THÈSE

Présentée à

L'Université Paris VIII

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE DE PARIS VIII

Spécialité **INFORMATIQUE**

Préparée au Laboratoire d'Informatique Avancée de Saint-Denis (LIASD)
Dans le cadre de l'École Doctorale Cognition Langage Interaction (CLI)

Présentée et soutenue publiquement

Par

Ludovic MENET

Le 24 juin 2010

Sur le sujet

Formalisation d'une approche d'Ingénierie Dirigée par les Modèles appliquée au domaine de la Gestion des Données de Référence.

Devant le jury composé de

Mme	Gabriella	SALZANO	Rapporteur
M.	Christian	PERCEBOIS	Rapporteur
Mme	Catherine	PELACHAUD	Directrice de thèse
M.	Olivier	CURÉ	Examinateur
Mme	Myriam	LAMOLLE	
M.	Abderahman	EL MAHMEDI	

À la mémoire de ma marraine Denise.

Résumé

Notre travail de recherche s'inscrit dans la problématique de la définition de modèles de données dans le cadre de la *Gestion des Données de Référence* ou Master Data Management. En effet, l'*Ingénierie Dirigée par les Modèles* (IDM) est un thème en pleine expansion aussi bien dans le monde académique que dans le monde industriel. Elle apporte un changement important dans la conception des applications en prenant en compte la pérennité des savoir-faire, des gains de productivité et en tirant profit des avantages des plateformes sans souffrir d'effets secondaires. L'architecture IDM se base sur la transformation de modèle pour aboutir à une solution technique sur une plateforme de notre choix a partir de modèles métier indépendants de toute plateforme. Dans cette thèse une démarche conceptuelle et expérimentale de l'approche IDM est appliquée à la définition de modèles de données pivot qui sont la base de la Gestion des Données de Référence ou Master Data Management (MDM). Ainsi utilisons nous UML (Unified Modeling Language), comme formalisme pour décrire les aspects indépendants de la plateforme (modèle métier), et nous proposons un métamodèle, sous la forme d'un profil UML, pour décrire les aspects dépendants de la plateforme MDM. Ensuite, nous présentons notre démarche pour passer d'un modèle métier vers un modèle de plateforme pour pouvoir générer le modèle pivot physique. Les apports de cette thèse sont : l'étude d'une approche IDM dans le contexte du MDM, la définition de transformations d'UML vers un modèle MDM (basé sur une structure XML Schema), d'autre part nous apportons un aspect inédit à l'IDM appliquée au MDM à savoir la définition d'une méthode de validation incrémentale de modèles permettant d'optimiser les phases de validation lors de la conception de modèles.

MOTS-CLÉS : IDM, Intégration de données, MDM, UML, Profil UML, XML, XML Schema, Méta-modèle, Validation.

Abstract

Our research work is in line with the problematic of data models definition in the framework of *Master Data Management*. Indeed, *Model Driven Engineering* (MDE) is a theme in great expansion in the academic world as well as in the industrial world. It brings an important change in the conception of applications taking in account the durability of savoir-faire and of gains of productivity, and taking profits of platforms advantages without suffering of secondary effects. The MDE architecture is based on the transformation of models to come to a technical solution on a chosen platform from independent business models of any platform. In this thesis, a conceptual and technical thought process of the MDE approach is applied to the definition of pivot data models, which are the base of Master Data Management (MDM). Thus, we use Unified Modeling Language (UML) as formalism to describe the independent aspects of the platform (business model), and we propose a meta-model, in the form of an UML profile, to describe the dependent aspects of the MDM platform. Then, we present our approach to move from a business model to a platform model to be able to generate the physical pivot model. The inputs of the thesis are: the study of a MDE approach in the MDM context, the definition of UML transformations towards a MDM model (based on a XML Schema structure), besides we bring a new aspect to MDE applied to MDM, that is to say the definition of a method for incremental model validation allowing the optimization of validation stages during model conception.

KEYWORDS: MDE, MDM, UML, UML Profile, XML, XML Schema, Meta-model, Validation.

Avant propos

Trois ans, déjà !

C'est une période qui peut paraître longue mais lorsque l'on se laisse prendre par cette belle aventure qu'est la thèse on se rend compte que ces trois années sont déjà finies, qu'il faut mettre un point final à celle-ci. La nostalgie de ces heures passées à étudier des travaux oh combien passionnants, à tenter d'apporter de nouvelles idées, à en échanger, et à publier des articles, se fait déjà ressentir en abordant ce mémoire.

Ce mémoire est le résultat de trois années et demie de travail qui m'ont permis d'explorer le monde de la recherche. Ce travail a été le fruit d'une collaboration, par l'intermédiaire d'une bourse CIFRE, entre le Laboratoire d'Informatique Avancée de Saint-Denis (LIASD) et l'éditeur logiciel Orchestra Networks.

Bien que ma thèse se soit déroulée dans le contexte d'une bourse CIFRE je tiens à préciser qu'elle a été le fruit d'un travail personnel. Je tiens de même à indiquer au lecteur que j'utiliserai tout au long de ce mémoire le sujet *nous* et de l'adjectif associé pour mentionner les actions que j'ai réalisées. Ce choix relève d'une habitude rédactionnelle que j'ai conservée tout au long de ma scolarité, de ma thèse et lors de la publication d'articles.

Je remercie l'ensemble des membres du jury,

S'il ne devrait y avoir que deux personnes à remercier, la première doit être ma co-directrice de thèse Mme Myriam Lamolle. Je ne la remercierai jamais assez pour son aide précieuse et sa patience qui m'ont permis de mener à bien ma thèse. Martial Doré, à la tête de la R&D de Orchestra Networks, est la seconde personne que je tiens particulièrement à remercier. J'ai énormément appris en travaillant dans son équipe. Un grand merci à ces deux personnes sans qui je n'aurai pu réaliser cette thèse.

J'exprime mes sincères remerciements à M. Abderrahman El Mahmedi, qui a accepté de participer au jury.

Je n'oublierai pas Mme Catherine Pelachaud qui a assumé le rôle de directrice de thèse en m'accueillant au sein de l'équipe LINC (Laboratoire d'INformatique et de Communication) qu'elle dirige.

Je suis extrêmement reconnaissant envers Mme Gabriella Salzano, pour avoir accepté d'être rapporteur de mon mémoire, ainsi que pour son regard critique concernant mes travaux.

Je remercie M. Christian Percebois, pour avoir accepté d'être rapporteur de ma thèse. Ses remarques pertinentes et constructives sur mon travail m'ont aidé à en améliorer la qualité. J'ai recueilli ses remarques et ses critiques avec intérêt.

Je remercie également M. Olivier CURÉ, pour avoir accepté d'examiner mon travail et de participer au jury de cette thèse.

Je tiens également à remercier,

Amar, ancien doctorant à l'IUT de Montreuil, qui m'a soutenu pendant toute la durée de mon stage de DEA et ma thèse; et sans oublier toutes les personnes de l'IUT de Montreuil qui m'ont accueillis, je m'excuse de ne pouvoir tous les mentionner. Je mentionnerai toutefois Fred, Madou, Philippe, Nicolas, Christophe, Sid, Max, Mamadou, Régis, Jean-Hugues, Anne, Alice.

Vincent, Eric, Christophe, Pierre, Yves, David, Jean-Baptiste, Manuel, Laure, Ghassen, Camille, André au sein d'Orchestra Networks pour leur soutient et ses discussions animées que nous avons eu et que nous continuerons d'avoir.

Et enfin je remercie du fond du cœur ma famille, mes amis et toutes les personnes que j'ai pu rencontrer durant ces trois années, qu'ils soient chaleureusement remerciés de m'avoir soutenu pour mener à bien cette thèse.

Table des matières

Résumé 4

Abstract 5

Avant propos 7

Chapitre 1. Introduction 26

1.1 Contexte et problématique 26

1.2 Principes et objectifs du MDM 29

Chapitre 2. Approches et systèmes d'intégration de données 33

2.1 Introduction 33

2.2 Les systèmes d'intégration existants 36

2.3 Fédération de schémas 39

2.3.1 Intégration par les vues 40

2.3.2 Modèles communs d'intégration de données 42

2.4 Traitement de requêtes 43

2.5 Architecture des systèmes existants 45

2.5.1 Architectures de médiations 45

2.5.1.1 TSIMMIS 45

2.5.1.2 MIX 46

2.5.1.3 DISCO 48

2.5.1.4 YAT 49

2.5.2 Systèmes d'intégration industriels 50

2.5.2.1 e-XMLMedia 50

2.5.2.2 XPERANTO 52

2.6 XML et les systèmes d'intégration de données 54

2.6.1 Apport d'un modèle unifié 54

2.6.2 Données semi-structurées et XML 54

2.7 Conclusion 57

Chapitre 3. La gestion des données de référence ou Master Data Management 61

3.1 Introduction 61

3.2 Définition de la donnée de référence 62

3.2.1 Donnée dupliquée au sein de plusieurs systèmes 63

3.2.2 Valorisation des données 66

3.2.3 Corollaire sur l'architecture MDM 67

3.3 Principes du MDM 67

3.3.1 Centralisation des données et la synchronisation des données 67

3.3.2 Confidentialité des données 68

3.3.3 Simplification de l'information 68

3.3.4 Qualité du contenu 68

3.3.5 Accessibilité 69

3.3.6 Flexibilité 69

3.3.7 Sécurité 69

3.3.8 Workflow intégré 69

3.4 Mise en place d'une architecture de type *MDM* 70

3.5 Solution MDM EBX.Platform 71

3.5.1 Architecture 71

3.5.2 Concepts 72

3.5.2.1 Héritage des données 73

3.5.2.2 Cycles de vie et versions des données 75

3.5.3 Propriétés d'un modèle d'adaptation 77

3.5.3.1 Les types de nœuds 77

3.5.3.1.1 Nœuds simples 77

3.5.3.1.2 Les nœuds simples multi-occurrencés 77

3.5.3.1.3 Les nœuds complexes 78

3.5.3.1.4 Les nœuds complexes multi-occurrencés 79

3.5.3.1.5 Les nœuds tables 79

3.5.3.2 Contraintes étendues 80

3.5.3.2.1 Facettes dynamiques 81

3.5.3.2.2 Contrainte d'intégrité sur les tables (clés étrangères) 81
3.5.4 Exemple de définition d'un modèle d'adaptation 82
3.6. Conclusion 86
Chapitre 4. L'Ingénierie Dirigée par les Modèles 89
4.1 Introduction 89
4.1.1 Architecture IDM 91
4.1.2 Concepts de base 92
4.2 Modèles 94
4.2.1 Modèle CIM 95
4.2.2 Modèle PIM 95
4.2.3 Modèle PSM 95
4.3 Métamodèles 96
4.3.1 Meta Object Facility 97
4.3.2 Eclipse Modeling Framework 99
4.3.3 Le rôle d'UML dans l'IDM 101
4.4 Transformation de modèles 106
4.4.1 Principes de la transformation de modèles 107
4.4.1.1 Définition des règles de transformation : 108
4.4.1.2 Expression des règles de transformation : 108
4.4.1.3 Exécution des règles de transformation 109
4.4.2 Les langages de transformation 110
4.4.2.1 MOF Query/View/Transformation 111
4.4.2.2 ATLAS Transformation Language (ATL) 112
4.4.2.3 MOFScript 115
4.5 Conclusion 121
Chapitre 5. Vers une représentation abstraite de modèles de données 122
5.1 Introduction 122
5.2 Intégration de métadonnées objet dans le modèle XML Schema 125
5.3 Profil UML associé à la sémantique de XML Schema 128
5.3.1 Propriétés d'un modèle XML Schema 130

5.3.1.1 Elément de déclaration d'un schéma 130
5.3.1.1.1 Stéréotype « Schema » 132
5.3.1.2 Eléments d'importation de schémas 133
5.3.1.2.1 Stéréotype « XSImport » 133
5.3.1.2.2 Stéréotype « Import » 134
5.3.1.2.3 Stéréotype « Redefine » 134
5.3.1.2.4 Stéréotype « Include » 134
5.3.2 Structure d'un modèle XML Schema 135
5.3.2.1 Types de données XML Schema 135
5.3.2.1 Eléments de déclaration d'éléments complexes 137
5.3.2.1.1 Stéréotype « ComplexType » 137
5.3.2.1.2 Stéréotype « Sequence » 138
5.3.2.1.3 Stéréotype « Choice » 139
5.3.2.1.4 Stéréotype « all » 139
5.3.2.2 Eléments de déclaration d'éléments simples 140
5.3.2.2.1 Stéréotype « XsElement» 140
5.3.2.2.2 Stéréotype « GlobalElement » 141
5.3.2.2.3 Stéréotype « SimpleType » 141
5.3.2.2.4 Stéréotype « List » 142
5.3.2.2.5 Stéréotype « Union » 142
5.3.2.3 Eléments de déclaration d'attributs 143
5.3.2.3.1 Stéréotype « Attribute » 143
5.3.2.3.2 Stéréotype « GlobalAttribute » 144
5.3.2.3.3 Stéréotype « AttributeGroup » 144
5.3.2.4 Eléments de déclaration de documentation 145
5.3.2.4.1 Stéréotype « XSDocumentation » 145
5.3.2.4.2 Stéréotype « Documentation » 146
5.3.2.4.3 Stéréotype « AppInfo » 146
5.3.3 Définition de contraintes XML Schema 147
5.3.3.1 Eléments de déclaration de contrainte d'unicité et de référence 147
5.3.3.1.1 Stéréotype « Ref » 147
5.3.3.1.2 Stéréotype « Key » 148
5.3.3.1.3 Stéréotype « Unique » 149
5.3.3.2.4 Stéréotype « KeyRef » 149

5.3.3.2 Eléments de déclaration de contraintes sur valeurs 150
5.3.3.2.1 Stéréotype « Facet » 150
5.3.3.2.2 Stéréotype « FacetEnumeration » 151
5.3.3.2.3 Stéréotype « FacetLength » 151
5.3.3.2.4 Stéréotype « FacetMinLength » 152
5.3.3.2.4 Stéréotype « FacetMaxLength » 152
5.3.3.2.5 Stéréotype « FacetPattern » 153
5.3.3.2.5 Stéréotype « FacetTotalDigits » 153
5.3.3.2.6 Stéréotype « FacetBoundary » 154
5.3.3.2.7 Stéréotypes « Facet{Min/Max}Boundary» 154
5.4 Profil Master Data Management 155
5.4.1 Définition d'un modèle d'adaptation 156
5.4.1.1 Stéréotype « AdaptationModel » 157
5.4.1.2 Stéréotype « Root » 159
5.4.2 Propriétés et structures avancées 160
5.4.2.1 Types de données étendus 160
5.4.2.2 Définition de services 161
5.4.2.2.1 Stéréotype << Service >> 162
5.4.2.3 Structures avancées d'un modèle d'adaptation 163
5.4.2.3.1 Stéréotype « MDMEntity » 165
5.4.2.3.2 Stéréotype « Domain » 165
5.4.2.3.3 Stéréotype « MDMSimpleElement » 166
5.4.2.3.4 Stéréotype « Table » 166
5.4.2.3.5 Stéréotype « PrimaryKey » 167
5.4.2.3.6 Stéréotype « Function » 168
5.4.2.3.7 Stéréotype « AutoIncrement » 168
5.4.3 Contraintes avancées 169
5.4.3.1 Stéréotype « DynamicFacet » 171
5.4.3.2 Stéréotype « DynamicEnumeration » 171
5.4.3.3 Stéréotype « DynamicLength » 171
5.4.3.4 Stéréotype « DynamicMinLength » 172
5.4.3.4 Stéréotype « DynamicMaxLength » 173
5.4.3.5 Stéréotype « DynamicBoundary » 173
5.4.3.6 Stéréotypes « Dynamic{Min/Max}Boundary» 174

5.4.3.7 Stéréotypes « Constraint» 174
5.4.3.8 Stéréotypes « ConstraintEnumeration» 175
5.4.4 Documentation avancée 176
5.4.4.1 Stéréotypes «Label» 177
5.4.4.2 Stéréotypes «Description» 177
5.4.4.3 Stéréotypes «DefaultErrorMessage» 178
5.4.4.4 Stéréotypes «MandatoryErrorMessage» 179
5.5 Conclusion 179
Chapitre 6. D'un modèle contemplatif à un modèle productif 181
6.1 Introduction 181
6.2 Spécification des mappings UML / XML Schema 184
6.2.1 Paquetages UML 185
6.2.2 Classes UML 185
6.2.3 Attributs 187
6.2.4 Cardinalités 189
6.2.5 Types de données natifs 189
6.2.6 Types dérivés 190
6.2.7 Généralisation / Spécialisation 191
6.2.8 Agrégation et composition 192
6.2.9 Notes et documentation 193
6.3 Implémentation des mappings 194
6.3.1 Transformation d'un modèle UML vers un modèle XML Schema 194
6.3.1.1 Association d'un métamodèle source 194
6.3.1.2 Importation de règles de transformation 195
6.3.1.3 Point d'entrée des règles de transformation 195
6.3.1.4 Règles de transformation 196
6.3.1.5 Propriétés et variables 198
6.3.1.6 Types de données natifs 198
6.3.1.7 Ecriture dans un fichier 198
6.3.1.8 Itérateurs 199
6.3.1.9 Instructions conditionnelles 199
6.3.1.10 Instruction de boucle conditionnelle 200

6.3.1.11 Opérations sur les modèles 200
6.3.1.12 Opérations sur les modèles UML..... 200
6.3.2 Transformation d'un modèle XML Schema vers un modèle UML..... 201
6.3.2.1 XML Metadata Interchange Format (XMI) 201
6.3.2.2 XSL Transformation (XSLT)..... 202
6.3.2.2.1 Déclaration d'une règle de construction..... 203
6.3.2.2.2 Application d'une règle de construction 204
6.3.2.2.3 Autres notions 205
6.4 Expérimentation des transformations..... 208
6.4.1 Définition d'un modèle XML Schema générique..... 210
6.4.2 Définition d'un modèle d'adaptation 214
6. Conclusion..... 217

Chapitre 7. Validation incrémentale de modèles 219

7.1 Introduction 219
7.2 Différentes approches de validation de modèles..... 220
7.2.1 Approches fondées sur OCL 220
7.2.2 Approches fondées sur les logiques de description 222
7.2.3 Approches fondées sur les graphes 224
7.3 Formalisation de la notion de métamodèle 225
7.3.1 Définitions..... 225
7.3.2 Formalisation de la notion de diagramme de classes..... 226
7.3.3 Définition des règles de validation par la logique du premier ordre..... 228
7.4 Validation incrémentale de modèles fondée sur des graphes..... 231
7.4.1 Représentation de modèles par graphes..... 231
7.4.2 Typologie d'actions et de règles de validation 234
7.4.3 Contextes de validation..... 235
7.4.4 Algorithme de validation incrémentale..... 239
7.5 Implémentation d'une méthode de validation incrémentale 240
7.5.1 Les logiques de description..... 241
7.5.1.1 La logique minimale *AL*..... 241
7.5.1.1.1 Les constructeurs de *AL* 241

7.5.2.1.2 La sémantique formelle de *AL* 242
7.5.1.1.3 Les extensions de *AL* 242
7.5.1.2 La logique de description SHIQ 244
7.5.2 Les niveaux de la logique de description 245
7.5.2.1 Le niveau terminologique (TBox) 245
7.5.2.1.1 Les entités atomiques. 246
7.5.2.1.2 Les concepts et rôles atomiques prédéfinis. 246
7.5.2.1.3 Les entités composées 246
7.5.2.2 Le niveau factuel (ABox) 246
7.5.3 Raisonnement par inférence 247
7.5.3.1 L'inférence au niveau terminologique 247
7.5.3.2 L'inférence au niveau factuel 247
7.5.3.3 Les moteurs d'inférences 248
7.5.5. Intégration à ArgoUML 249
7.5.6. Résultats expérimentaux 251
7.6. Conclusion 254

Chapitre 8. Conclusions et perspectives *255*

8.1 Contributions 255
8.1.1 Etude des approches d'intégration de données 256
8.1.2 Présentation d'une nouvelle approche d'intégration de données 257
8.1.3 Introduction d'une méthode d'Ingénierie Dirigée par les Modèles et premier profil UML dédié au MDM 258
8.1.4 Méthode de validation incrémentale dans le domaine de l'IDM 259
8.2 Perspectives 260
8.2.1 Normalisation du profil UML appliqué au MDM 260
8.2.2 Introduction d'une approche ontologique 260
8.2.3 Définition d'une approche de résolution d'erreurs de validation et expérimentation à grande échelle 261

Bibliographie *262*

Table des figures

Figure 2.1. Système d'intégration de données. 33
Figure 2.2. Les niveaux de schémas dans les systèmes d'intégration. 36
Figure 2.3. Architecture de médiation. 38
Figure 2.4. Approche GAV vs LAV. 41
Figure 2.5. Traitement de requêtes dans les systèmes d'intégration. 44
Figure 2.6. Architecture du système TSIMMIS. 46
Figure 2.7. Architecture du système MIX. 47
Figure 2.8. Architecture du système DISCO. 48
Figure 2.9. Architecture du système YAT...50
Figure 2.10. Architecture du système e-XMLMedia..52
Figure 2.11. Architecture du système XPERANTO. 53
Figure 2.12. Approche virtuelle...57
Figure 2.13. Approche matérialisée...58

Figure 3.1. La propagation de données en mode point à point...64
Figure 3.2. La propagation de données via le Master Data Management. 65
Figure 3.3. Valorisation des données dans une structure organisationnelle.............................66
Figure 3.4. Architecture EBX.Platform. 72
Figure 3.5. Arbre d'adaptation. 74
Figure 3.6 - Cycle de vie et traçabilité des données. 75
Figure 3.7. Exemple de déclaration d'un modèle d'adaptation contenant une seule racine. ...77
Figure 3.8. Exemple de déclaration d'un nœud simple 77
Figure 3.9. Exemple de déclaration d'un nœud simple multi-occurrencé...............................78
Figure 3.10. Exemple de déclaration d'un nœud complexe. 79
Figure 3.11. Exemple de déclaration d'un complexe multi-occurrencé..................................79
Figure 3.12. Exemple de déclaration d'une table contenant 5 champs. 80

Figure 3.13. Exemple d'utilisation d'une contrainte dynamique. 81
Figure 3.14. Exemple de définition d'une contrainte de clé étrangère.................................... 82
Figure 3.15. Modèle relationnel d'une base de données de publication d'ouvrages................ 83
Figure 3.16. Extrait d'un modèle d'adaptation.. 85
Figure 3.17. Visualisation d'une table dans EBX.Manager. 85
Figure 3.18. Visualisation d'un enregistrement. 86

Figure 4.1. L'impact de l'IDM dans le processus de développement logiciel......................... 91
Figure 4.2. Architecture IDM. 92
Figure 4.3. Architecture à 4 couches de métamodélisation... 96
Figure 4.4. Métamodèle EMOF... 98
Figure 4.5. Diagramme du package principal UML.. 99
Figure 4.6 Extrait du métamodèle ECore. 100
Figure 4.8. Exemple de contrainte OCL.. 105
Figure 4.9. Exécution des règles de transformation. 109
Figure 4.10. Architecture QVT des langages de spécification de modèles........................... 111
Figure 4.11. Extrait du métamodèle des règles ATL... 113
Figure 4.12. Patrons d'éléments (syntaxe abstraite).. 114
Figure 4.13. Transformation de modèle avec MOFScript... 116
Figure 4.14. MOFScript et l'architecture quatre couches du MOF....................................... 117
Figure 4.15. MOFScript et QVT. 118
Figure 4.16. Spécialisation de QVT par MOFScript... 119
Figure 4.17. Exemple de transformation MOFScript.. 120

Figure 5.1. Architecture 4 couches du MOF. 124
Figure 5.2. Spécificités objet d'un diagramme de classes UML... 126
Figure 5.3. Extension XML Schema représentant une métaconnaissance « objet ». 127
Figure 5.4. Relation de composition UML introduite dans XML Schema. 128

Figure 5.5. Extrait du profil XML Schema. 129
Figure 5.6. Extrait du profil XML : déclaration d'un schéma. 131
Figure 5.7. Définition d'un identifiant XML Schema. 131
Figure 5.8. Définition des propriétés d'un schéma XML. 132
Figure 5.9. Extrait du profil XML : import de schémas. 133
Figure 5.10. Import d'un schéma XML. 134
Figure 5.11. Redéfinition d'un schéma XML. 134
Figure 5.12. Inclusion d'un schéma XML. 135
Figure 5.13. Types de données XML Schema. 136
Figure 5.14. Extrait du profil XML : déclaration d'éléments complexes. 137
Figure 5.15. Élément complexe XML Schema. 138
Figure 5.16. Élément complexe définissant une suite ordonnée d'éléments. 138
Figure 5.17. Élément complexe définissant un choix d'éléments. 139
Figure 5.18. Élément complexe définissant un ensemble d'éléments non ordonnés 139
Figure 5.19. Extrait du profil XML : déclaration d'éléments simples. 140
Figure 5.20. Élément simple XML Schema. 141
Figure 5.21. Type simple nommé XML Schema. 141
Figure 5.22. Liste XML Schema. 142
Figure 5.23. Union de types XML Schema. 143
Figure 5.24. Extrait du profil XML : déclaration d'éléments simples. 143
Figure 5.25. Attribut XML Schema. 144
Figure 5.26. Attribut global à un schéma XML. 144
Figure 5.27. Groupe d'attributs XML Schema. 145
Figure 5.28. Extrait du profil XML : éléments de documentation XML Schema. 145
Figure 5.29. Documentation XML Schema. 146
Figure 5.30. Élément de documentation additionnelle. 146
Figure 5.31. Extrait du profil XML : éléments de définition de contrainte d'unicité et de référence. 147
Figure 5.32. Référence vers un élément XML Schema. 148

Figure 5.33. Définition d'une clé XML Schema. 148
Figure 5.34. Définition d'une contrainte d'unicité XML Schema. 149
Figure 5.35. Définition d'une référence vers une contrainte d'unicité XML Schema. 150
Figure 5.36. Extrait du profil XML : éléments de contrainte d'unicité et de référence. 150
Figure 5.37. Enumération XML Schema. 151
Figure 5.38. Contrainte sur la longueur d'une valeur. 152
Figure 5.39. Contrainte sur la longueur minimale d'une valeur. 152
Figure 5.40. Contrainte sur la longueur maximale d'une valeur. 153
Figure 5.41. Contrainte sur la « forme » d'une valeur. 153
Figure 5.42. Contrainte sur le nombre de chiffres d'un élément. 154
Figure 5.43. Contraintes permettant de borner la valeur d'un élément. 155
Figure 5.44. Extrait du profil MDM : propriétés d'un modèle d'adaptation. 157
Figure 5.45. Définition d'un modèle d'adaptation et de bindings. 159
Figure 5.46. Racine d'un modèle d'adaptation et définition d'un trigger. 160
Figure 5.47. Extrait du profil MDM : types de données étendus. 160
Figure 5.48. Extrait du profil MDM : définition de services. 162
Figure 5.49. Définition d'un service. 162
Figure 5.50. Extrait du profil MDM : éléments avancés d'un modèle d'adaptation. 164
Figure 5.51. Définition d'un domaine. 166
Figure 5.52. Définition d'un élément simple avancé. 166
Figure 5.53. Définition d'une table. 167
Figure 5.54. Définition d'un élément dont la valeur est calculée. 168
Figure 5.55. Définition d'un élément auto-incrémenté. 169
Figure 5.56. Extrait du profil MDM : contraintes avancées. 170
Figure 5.57. Enumération dynamique. 171
Figure 5.58. Contrainte dynamique sur la longueur d'une valeur. 172
Figure 5.59. Contrainte dynamique sur la longueur minimale d'une valeur. 173
Figure 5.60. Contrainte dynamique sur la longueur maximale d'une valeur. 173
Figure 5.61. Contraintes dynamiques permettant de borner la valeur d'un élément. 174

Figure 5.62. Définition d’une contrainte définie programmatiquement................................175
Figure 5.63. Définition d’une énumération définie programmatiquement............................175
Figure 5.64. Extrait du profil MDM : documentation avancée. 176
Figure 5.65. Libellé localisé d’un nœud dans un modèle d’adaptation. 177
Figure 5.66. Description localisée d’un nœud dans un modèle d’adaptation. 178
Figure 5.67. Message d’erreur portant sur des contraintes. 178
Figure 5.68. Message d’erreur sur une saisie obligatoire. 179

Figure 6.1. Mapping de niveau 1 entre UML et XML Schema. 183
Figure 6.2. Transformation d’un paquetage UML. 185
Figure 6.3. Transformation d’une classe UML. 186
Figure 6.4. Transformation d’une classe UML en un élément complexe global XML Schema. 186
Figure 6.5. Structure d’un élément complexe XML Schema. 187
Figure 6.6. Transformation d’une classe abstraite UML. 187
Figure 6.7. Transformation d’attributs UML vers des éléments simples XML Schema. 188
Figure 6.8. Transformation d’attributs UML en des éléments attributs XML Schema. 189
Figure 6.9. Transformation de cardinalités UML en cardinalités XML Schema. 189
Figure 6.10. Transformation de types de données natifs UML. 190
Figure 6.11. Transformation d’un type de données dérivé UML. 191
Figure 6.12. Transformation des propriétés de généralisation UML. 192
Figure 6.13. Transformation de relations de dépendance UML. 193
Figure 6.14. Transformation d’éléments de documentation UML. 194
Figure 6.15. Instruction MOFScript d’association d’un métamodèle source. 195
Figure 6.16. Instruction MOFScript d’importation de règles. 195
Figure 6.17. Point d’entrée d’un script de transformation MOFScript. 195
Figure 6.18. Point d’entrée MOFScript appliqué aux classes d’un modèle UML. 196
Figure 6.19. Point d’entrée MOFScript sans contexte. 196

Figure 6.20. Règles de transformation MOFScript. 197
Figure 6.21. Règles de transformation MOFScript avec valeur de retour........ 197
Figure 6.22. Règles de transformation MOFScript avec paramètres. 198
Figure 6.23. Propriétés et variables MOFScript. 198
Figure 6.24. Ecriture dans un fichier à partir d'une règle MOFScript. 199
Figure 6.25. Itérateur MOFScript. 199
Figure 6.26. Instructions conditionnelle MOFScript. 199
Figure 6.27. Boucle conditionnelles MOFScript. 200
Figure 6.28. Collection d'objets MOFScript d'un type donné. 200
Figure 6.29. Représentation XMI d'une classe UML. 202
Figure 6.30. Processus de transformation XSLT. 203
Figure 6.31. Déclaration d'un template XSLT. 203
Figure 6.32. Instruction d'écriture XSLT. 204
Figure 6.33. Instruction de copie XSLT. 204
Figure 6.33. Instruction d'exécution d'une règle XSLT. 204
Figure 6.34. Exemple d'application d'une règle XSLT. 205
Figure 6.34. Instruction conditionnelle XSLT. 205
Figure 6.35. Instruction de branchement conditionnel XSLT. 205
Figure 6.36. Attributs XSLT. 206
Figure 6.37. Instruction de copie ciblée XSLT. 206
Figure 6.38. Extrait d'un script de transformation d'un modèle XML Schema vers un modèle XMI. 208
Figure 6.39. Import / Export de modèles XML Schema dans ArgoUML. 209
Figure 6.40. Visualisation d'un modèle XML Schema dans ArgoUML. 210
Figure 6.41. Schéma XML de modélisation d'un bon de commande. 212
Figure 6.42. Modélisation UML d'un bon de commande défini avec XML Schema. 213
Figure 6.43. Modèle relationnel d'une base de données de publication d'ouvrages. 214
Figure 6.44. Exemple de modèle d'adaptation défini à l'aide du profil UML MDM. 215

Figure 6.45. Structure XML Schema d’un modèle d’adaptation de gestion de publications d’ouvrages. 217

Figure 7.1. Métamodèle des diagrammes de classes UML. 227
Figure 7.2. Formalisation du métamodèle des diagrammes de classes UML. 228
Figure 7.3. Exemple de formule de la logique du premier ordre. 229
Figure 7.4. Quantificateurs de la logique du premier ordre. 229
Figure 7.5. Règle de validation fondée sur la logique du premier ordre. 230
Figure 7.6. Métamodèle de la structure de graphe. 232
Figure 7.7. Mappings entre un diagramme de classes UML et un graphe. 233
Figure 7.8. Exemple de graphe. 234
Figure 7.9. Contexte de classe de validation. 236
Figure 7.10. Contexte de propriété de validation. 237
Figure 7.11. Contexte de référence de validation. 238
Figure 7.12. Algorithme de validation incrémentale. 240
Figure 7.13. Constructeurs *AL*. 241
Figure 7.14. Sémantique formelle de *AL*. 242
Figure 7.14. Exemple d’extension de *AL* [Baader *et al.*, 2003]. 243
Figure 7.15. Syntaxe $ALCQHIR^{+}$. 245
Figure 7.16. Exemple d’une base de connaissances. 245
Figure 7.17. Comparatif portant sur des moteurs d’inférence. 249
Figure 7.18. Processus de validation incrémentale introduit dans ArgoUML. 250
Figure 7.19. Rapport de validation ArgoUML. 251
Figure 7.20. Graphe d’un modèle de publication d’ouvrages. 252
Figure 7.21. Résultats sur un modèle de petite taille. 252
Figure 7.22. Résultats sur un modèle de grande taille. 253

Chapitre 1

Introduction

1.1 Contexte et problématique

Les avancées considérables en matière de réseaux et de bases de données ont conduit à la multiplicité et à l'expansion des systèmes d'information à grande échelle. Dans ces systèmes cohabitent des données et des services les manipulant. Ces derniers gèrent souvent des sources de données hétérogènes et réparties. Le rôle des systèmes d'intégration de données est donc de répondre aux besoins des utilisateurs au travers d'interfaces d'accès uniformes à ces sources. Cependant, la prise en compte de ces besoins amène une plus grande complexité des systèmes d'intégration. L'hétérogénéité des sources de données rend les processus d'intégration complexes et coûteux. Le défi de l'intégration de sources de données est de faire cohabiter ces sources hétérogènes, de plus en plus nombreuses, souvent réparties et indépendantes, dans un seul système uniforme, appelé système d'intégration, sans contraindre le comportement ni l'autonomie de chacune d'elles.

Dans les premières solutions proposées, deux approches sont clairement identifiées et mises en avant. L'approche virtuelle [Garcia-Molina *et al.*, 1997], ou par médiateur désigne une vision globale par l'intermédiaire d'un unique schéma de représentation d'un ensemble de sources de données hétérogènes. Ce schéma global peut être défini automatiquement à l'aide d'outils d'extraction de schémas. Dans ce contexte, les données sont stockées uniquement au niveau des sources. Les traitements sont donc synchronisés sur ces sources de données. Un médiateur connaît le schéma global et possède des vues abstraites sur les sources qui lui permettront lors d'interrogation par un utilisateur final de décomposer la requête initiale en sous-requêtes. Le médiateur soumet ces sous-requêtes à des adaptateurs qui ont pour fonction de traduire ces dernières dans des langages compréhensibles par les différentes sources de données. Une fois le traitement de ces requêtes réalisé par ces sources, les réponses suivent le cheminement inverse jusqu'à l'utilisateur.

Dans la seconde approche, dite d'intégration matérialisée [Abiteboul *et al.*, 2002], les données issues de sources hétérogènes sont copiées dans un entrepôt de données (ou référentiel). Les actions sur le référentiel sont asynchrones par rapport aux sources. La propagation des modifications apportées au référentiel vers les différentes sources de données doit passer par des procédures de mises à jour. Contrairement à l'approche virtuelle, les requêtes utilisateurs sont directement exécutées dans le référentiel, sans avoir à accéder

aux différentes sources de données. Ici, les données du référentiel sont déconnectées de celles contenues dans les sources hétérogènes. Les mises à jour des données du référentiel vers les sources et réciproquement sont déléguées à un intégrateur qui a pour fonction de réaliser la correspondance entre le schéma du référentiel et les sous-schémas des sources.

Les approches virtuelles et matérialisées ont fait l'objet de nombreuses études. Cependant, elles présentent encore des limitations en terme de standardisation, d'outils et de complexité de mise en place dans des contextes industriels. De plus, certains de ces systèmes se focalisent uniquement sur certaines problématiques telles que le traitement des requêtes ou l'intégration et la diffusion de données et ne traitent pas certains aspects importants de gestion de la données tels que :

- La mise en place de rôles et de droits d'accès pour accéder à la donnée.
- L'uniformisation de la représentation des données au sein d'un système d'information par l'intermédiaire d'un outil de gestion unique.
- La mise à disposition de moyens permettant d'auditer et d'historiser les données.
- La gestion de manière efficace et performante du cycle de vie des données et de l'accès concurrentiel entre différents utilisateurs.

Pour outrepasser ces problématiques individuelles, la gestion des données de référence [Régnier-Pécastaing *et al.*, 2008] ou Master Data Management (MDM) a été définie comme une approche visant à pallier ces limitations tout en se focalisant sur l'ensemble des problématiques de fédération de sources de données.

Le Master Data Management (MDM) [iWays 2009] [Oracle 2007] [IBM 2004] [Orchestranetworks MDM 2000] est une approche émergente d'intégration de données. Le MDM étant une discipline récente, très peu de travaux existent à ce jour. Par rapport à l'approche matérialisée, le MDM se focalise de plus sur l'unification des modèles et outils au sein d'un système d'information. Actuellement, la majorité des systèmes d'information sont caractérisés par une hétérogénéité en terme de données et de solutions de paramétrage. En effet, cette hétérogénéité se présente sous différents aspects : diversité des systèmes de stockage (bases de données, fichiers, annuaires, etc.), diversité des formats de données (tables, fichiers propriétaires, documents XML [W3C, 1998], etc.), diversité des solutions proposées pour gérer les différents types de données, diversité des acteurs exploitants les données de références (utilisateurs fonctionnels ou non), diversité des domaines d'application, diversité des activités (dites verticales pour des activités telles que la production ou l'approvisionnement, ou dites horizontales pour des activités telles que le marketing ou les ressources humaines), etc. D'autre part, utiliser un ensemble d'applications différentes afin de pouvoir gérer cette diversité dans les types de données entraîne inévitablement de la redondance tant au niveau des données que des outils. En l'absence de MDM, la propagation des mises à jour de données se réalise sans référentiel central ni modèle commun d'information.

La modélisation d'un modèle fédérateur de source de données hétérogènes doit prendre en compte quatre composantes. Premièrement, il est essentiel de parfaitement analyser et spécifier les modèles à intégrer. Cette analyse des différents modèles doit être le point de départ d'une conception la plus rigoureuse possible pour permettre la réalisation de

modèles répondant à des critères de plus en plus exigeants. La définition de modèles doit nécessiter aussi la prise en compte et l'intégration de spécificités de systèmes, de modules, d'applicatifs et/ou de composants déjà existants. Le développement de modèles, souvent volumineux, met aussi en relation des équipes. Ces équipes doivent pouvoir communiquer et échanger des informations. Ce dernier point doit prendre en compte trois dimensions :

(i) La culture des différentes équipes,
(ii) L'équipement informatique à disposition pour chacune d'elles,
(iii) La compétence de chaque participant.

Pour prendre en compte ces composantes, il est nécessaire d'utiliser le plus possible de standards. En effet, les standards assurent une indépendance par rapport aux outils et facilitent les échanges. Toutefois, les modèles de données à intégrer ne respectent généralement pas ces standards. Il faut donc être capable de transformer ces modèles de manière à extraire et sélectionner l'information pertinente. Cette transformation doit en particulier permettre le passage d'un modèle non standard à un modèle standard.

Nous considérons qu'un modèle est une description abstraite d'un système ou d'un processus. Il doit être expressif pour tirer parti de l'information contenue dans les données. En effet, pour pouvoir transformer un modèle existant en un modèle standard, il faut que le nouveau modèle puisse complètement prendre en compte la sémantique de l'information. En effet, il est indispensable que la transformation se fasse sans aucune perte d'information. Le nouveau modèle doit aussi avoir un niveau d'abstraction puissant. En particulier, il doit permettre de modéliser des informations de manière à ce qu'elles soient compréhensibles par des spécialistes d'horizons divers. Si on se positionne dans le domaine des processus d'affaire, il faut que le modèle de données utilisé permette d'extraire et de mettre en forme l'information nécessaire aux juristes. Enfin, le modèle doit être performant pour répondre aux besoins des utilisateurs en terme de traitements et de manipulation de données.

L'accroissement de la taille des modèles ainsi que l'accroissement de la complexité de ceux-ci nécessitent la mise en œuvre de méthodes et de notations dès les premières phases du cycle de vie. Ces méthodes et notations doivent apporter une aide importante aux équipes de conception pour pouvoir communiquer et échanger de l'information. Ces méthodes et notations doivent s'appuyer sur des outils offrant une assistance aux équipes pour la réalisation d'applications répondant à des critères de qualité de plus en plus stricts. Le besoin de communiquer et d'échanger est dû à deux évolutions du développement d'applications à savoir l'augmentation croissante de la taille et de la complexité des modèles et la nécessité de faire appel à des spécialistes du domaine qui ne sont pas forcément des informaticiens. En particulier, ces spécialistes interviennent lors de la phase de spécification de besoins et lors de la recherche des objets métiers et de l'étude de l'existant (informatique ou non informatique). Par exemple, lors de la définition de processus d'affaires dans le commerce électronique, il est nécessaire d'associer à la phase de spécification des juristes qui seront les garants de la régularité de la transaction commerciale.

Pour être exploitable, l'information donnée aux différents spécialistes doit répondre aux trois critères suivants :

- L'information présentée doit tenir compte de la spécificité culturelle de l'interlocuteur. En effet, lors de projet nécessitant l'intervention de plusieurs équipes,

il n'est pas évident que toutes les équipes aient la même culture. Cette différence est due au fait que les équipes peuvent être réparties au sein de différentes entreprises et/ou au sein de différents pays. C'est le cas typique des projets européens.

- L'information doit pouvoir être comprise par tous les membres d'une équipe, informaticiens ou non. Par exemple, lorsqu'un juriste veut vérifier la conformité d'un message dans une transaction commerciale, il faut lui présenter l'information sous la forme d'un message (l'objet réel) et non sous la forme d'attributs d'une classe représentant les éléments du message. Dans cet exemple, on remarquera que l'effort est fait au niveau de la présentation de l'information et non du contenu à proprement parler. Il n'est en aucun cas question de transformer une information existante en nouvelle information.
- L'information manipulée doit être indépendante des outils utilisés pour la créer. Cette indépendance a pour objectif d'éviter l'achat inutile d'outils. Par exemple, un spécialiste juridique n'a pas intérêt à acheter un atelier UML pour vérifier la définition de messages dans un processus d'affaire. En effet, il sera dans l'incapacité de traiter l'information brute et l'investissement sera disproportionné par rapport aux retombées escomptées.

Dans ce contexte, il est essentiel de s'appuyer sur des standards pour représenter l'information et pour y accéder.

1.2 Principes et objectifs du MDM

Deux conditions sont nécessaires à une architecture d'intégration de type MDM :

- il faut disposer d'un outil de MDM générique capable d'accueillir le modèle commun d'information pour toutes les natures de données. Sans ce niveau de généricité, il faudrait accepter des MDM par silos organisés autour des domaines d'information (Client, Produit, Organisation, paramètres fonctionnels, paramètres techniques, etc.) et les conséquences néfastes en terme de duplication des référentiels (ce que l'on cherche à éviter) et de duplication des fonctions de gouvernance (gestion des versions, interface homme-machine d'administration, etc.) ;
- il faut une méthode pour la modélisation et la négociation du modèle commun d'information faisant abstraction des formats « propriétaires » des différents systèmes.

Le premier point est assuré par la solution MDM, orientée modèles et basée sur la technologie XML Schema [W3C, 2001a] que nous utiliserons comme base de nos travaux [Orchestranetworks MDM 2000].

L'objectif principal de ce doctorat est de proposer une méthode d'*Ingénierie Dirigée par les Modèles* (IDM) appliquée au domaine du *Master Data Management*. Nous montrerons que cette approche est une solution adéquate au MDM.

Les travaux à mener pour intégrer une approche IDM afin d'uniformiser la modélisation des schémas sont de deux types à savoir la mise en place d'une solution de représentation générique et l'automatisation de transformation de modèles.

Pour le second point concernant la représentation générique, il s'agit de définir un formalisme abstrait permettant de manipuler des modèles XML Schema en utilisant les standards préconisés par l'OMG tels que MOF [MOF, 2006], XMI [Iyengar *et al.*, 1998], OCL [OMG, 2002b] et UML [UML, 1997]. L'introduction d'un formalisme standard de définition de modèles est une première étape dans notre approche IDM. La notation que nous avons choisie est la notation UML. Comme la plupart des notations actuelles, elle est basée sur un ensemble de représentations permettant de modéliser les aspects statique et dynamique des objets ainsi que l'aspect implémentation physique. Ces différents aspects sont représentés par des ensembles de diagrammes permettant d'exprimer différents points de vue. Toutefois, les outils actuels ne permettent pas d'exprimer des contraintes sur ces diagrammes et surtout ne permettent pas de vérifier la cohérence des différents diagrammes entre eux. Pourtant, cet aspect revêt un caractère essentiel lors de l'intégration de composants conçus et réalisés par des équipes réparties. En effet, l'intégrateur est alors seul responsable de la cohérence de l'ensemble. Sur des projets volumineux et complexes, cette intégration est difficile voire impossible sans l'aide d'outils.

La notion de Profil UML [Corba, 2002] [Greenfield, 2001] a été introduite dans le standard UML 1.3 comme un moyen permettant la structuration des extensions UML (stéréotypes, contraintes et valeurs étiquetées). Dans ce contexte, notre objectif est de structurer notre approche d'Ingénierie Dirigée par les Modèles, qui propose une extension du métamodèle UML, sous forme de deux profils UML :

- un premier profil UML dédié à la sémantique de XML Schema [W3C, 2001a],
- un second profil UML dédié à la sémantique des modèles d'adaptation définie dans le contexte du Master Data Management.

En ce qui concerne l'aspect transformation, il est question de définir un ensemble de règles de transformation, que nous appellerons *mapping* permettant d'établir une projection d'un modèle XML Schema vers un modèle UML et inversement. Dans cette étape de transformation, la génération de code est un processus important dans lequel il s'agira d'obtenir à partir d'une spécification abstraite une spécification concrète d'un modèle.

Nos travaux reposant sur l'intégration d'une approche IDM, nous devons assurer à tout instant la validité des modèles en cours de définition. Les approches classiques de validation de modèle reposent sur la vérification intégrale d'un modèle. En effet, lorsqu'un modèle est modifié, il est nécessaire de valider à nouveau l'ensemble du modèle pour vérifier si la modification apportée n'a pas entraîné une incohérence dans la structure du modèle. Cette approche est convenable dans des situations de modélisation de modèles de taille raisonnable, mais nous pouvons entrevoir que ce processus n'est pas envisageable dans des contextes industriels et lors d'un passage à l'échelle. Ce problème repose sur la non réutilisation des informations issues des contrôles effectués lors de chaque modification unitaire. Pour optimiser notre approche IDM, nous proposons d'introduire une approche incrémentale de validation de modèles.

Notre approche est fondée sur une formalisation de la notion de métamodèle dans le but d'exprimer des règles de validation et de cohérence structurelle sous la forme de formules de la logique du premier ordre. Une approche par graphes et des ensembles de règles on été définis afin d'introduire la notion de « *contexte de validation* ». Ces contextes de validation sont utilisés pour définir les sous-parties d'un graphe qui sont potentiellement impactées lors de chaque modification atomique d'un modèle.

Dans une approche de validation incrémentale, deux problématiques se posent :

(i) Quelles sont les règles à vérifier lors de chaque action sur le modèle ?
(ii) Quelles parties du modèle sont à vérifier ?

La solution à la première question est de considérer que, pour une action donnée, seul un ensemble de règles est à vérifier. En effet, une classification des règles à vérifier va nous permettre d'indiquer quelles sont les règles à valider lorsqu'une action précise se produit et de ne pas vérifier les règles non impliquées.

Concernant le second point, nous nous appuyons sur la définition de l'ensemble des parties du modèle pouvant invalider ou valider une règle de validation. En mettant en place ce mécanisme, nous optimiserons le processus de validation en considérant que seules les modifications affectant une partie du modèle peut en entrainer sa vérification.

Pour résumer, l'objectif de cette thèse est de formaliser une approche d'Ingénierie Dirigée par les Modèles dans le contexte du Master Data Management. Pour cela il faut concevoir une méthode permettant de représenter de manière abstraite un modèle de données en utilisant les standards préconisés par l'OMG et définir aussi des mécanismes permettant de générer automatiquement le modèle physique correspondant. Puis, il est aussi question de garantir l'intégrité des modèles définis de manière optimale en proposant une méthode de validation incrémentale de modèles dans le contexte du Master Data Management.

Ce mémoire est structuré de la manière suivante :

Le chapitre 2 recense les différentes approches des systèmes d'intégration de données.

Le chapitre 3 expose la problématique associée à la gestion de la donnée de référence au sein d'un Système d'Information.

Le chapitre 4 a pour objectif de présenter les principes de l'Ingénierie Dirigée par les Modèles (IDM).

Le chapitre 5 développe une mise en application d'une métamodélisation UML, un des principes fondamentaux de l'Ingénierie Dirigée par les Modèles, appliquée à la définition de modèles XML Schema et au domaine du Master Data Management.

Le chapitre 6 présente les règles de correspondance ou *mappings* permettant à la fois de produire un modèle XML Schema à partir d'un diagramme de classe UML générique ou spécialisé par l'intermédiaire de nos profils UML.

Le chapitre 7 introduit une méthode de validation incrémentale de modèles dans notre approche d'Ingénierie Dirigée par les Modèles afin d'optimiser les processus de vérification des structures de données.

Enfin, le chapitre 8 conclut ce mémoire en présentant les contributions apportées durant cette thèse, et présente les perspectives ouvertes à l'issue de nos travaux.

Chapitre 2

Approches et systèmes d'intégration de données

Résumé. L'objectif de ce chapitre est de présenter les différentes approches et systèmes d'intégration de données.

2.1 Introduction

Un système d'intégration de données tient le rôle d'un middleware[1] entre un ensemble de sources de données et des utilisateurs et/ou des applications ; il permet de fournir une vue unifiée de données provenant de sources multiples et hétérogènes, comme illustré par la figure 2.1. Ceci permet un accès aux données au travers d'une interface uniforme, indépendamment de leur structure et de leur localisation.

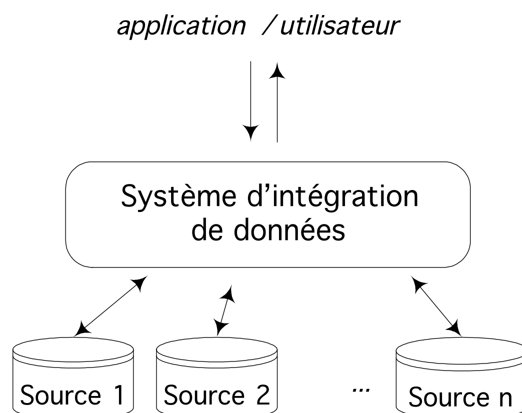


Figure 2.1. Système d'intégration de données.

[1] Désigne la couche intermédiaire entre les couches inférieures (les sources) et les couches supérieures (les applications).

L'élaboration d'un système d'intégration est une tâche difficile et a été le centre d'intérêt de nombreuses recherches en bases de données depuis une trentaine d'années. Les principales dimensions qui doivent être prises en compte dans de tels systèmes sont [Ozu *et al.*, 1999], [Hasselbring, 2000] :

- Distribution : les sources de données à intégrer peuvent être largement réparties. L'architecture des systèmes d'intégration de données doit prendre en compte cette répartition, notamment dans le traitement de requêtes, et donc, doit garantir un certain niveau de transparence.
- Autonomie : les sources de données et toutes leurs fonctionnalités existent avant la mise en place du système d'intégration lui-même. L'intégration de données est effectuée en gardant et respectant l'autonomie des sources dans le sens où celles-ci s'exécutent et évoluent indépendamment du système.
- Hétérogénéité : elle concerne aussi bien les sources de données que les caractéristiques de leurs plateformes d'exécution. Plusieurs types d'hétérogénéité existent par l'existence de différentes plateformes d'exécution tant au niveau matériel et logiciel (ex. systèmes d'exploitation) et au niveau des systèmes de communication (ex. protocoles de communication).

En ce qui concerne les schémas, les modèles de données et leurs sémantiques associées, cette hétérogénéité se traduit par des conflits d'intégration.

L'hétérogénéité se situe aussi au niveau des fonctionnalités offertes par chaque source en termes de capacité d'évaluation et de restriction d'accès. Certaines sources sont capables d'exécuter tout type d'opérations, comme celles implémentées au sein des bases de données, alors que d'autres sources ne peuvent effectuer qu'un simple parcours séquentiel sur leur contenu. Aussi certaines sources ne permettent-elles qu'un accès restreint aux données selon certains « patterns ».

Ces trois dimensions (distribution, autonomie et hétérogénéité) sont renforcées dans le contexte des systèmes d'information actuels, tenant compte des principaux niveaux d'interopérabilité. Cette interopérabilité concerne le système, le langage, la structure et la sémantique des sources de données à intégrer [Sheth, 1999]. En effet, les systèmes interopérables sont généralement composés de services et de données largement réparties qui cohabitent souvent d'une manière autonome, notamment à travers le Web. De plus, ces systèmes sont de plus en plus dynamiques dans le sens où de nouvelles sources peuvent être ajoutées ou supprimées à tout moment. A titre d'exemple, citons [Gardarin *et al.*, 2002], [Draper *et al.*, 2001], [Halevy *et al.*, 2003b]. Dans [Sheth, 1999], l'auteur présente un état de l'art complet sur les trois générations des systèmes d'informations, en mettant l'accent sur les différents niveaux d'interopérabilité supportant les différents types d'hétérogénéité précités.

Ce chapitre présente la problématique de l'intégration des données selon différents points de vue, en distinguant principalement les modèles et mécanismes utilisés pour l'intégration et les architectures des systèmes d'intégration. La section 2.2 présente d'une manière non exhaustive un ensemble de systèmes d'intégration de données basés sur une

architecture de médiation de données. Nous évoquons dans la section 2.3 la problématique de d'intégration de schémas et des mécanismes permettant de réponde aux besoins d'intégration. Les différents aspects et les principales étapes de traitement de requêtes dans les systèmes d'intégration sont montrés dans la section 2.4. Des systèmes d'intégration existants sont présentés dans la section 2.5 en mettant en avant leurs architectures et leurs principales caractéristiques. Pour finir, nous présentons le langage XML comme modèle pivot commun aux différentes sources et son apport notamment pour les systèmes d'intégration de données.

2.2 Les systèmes d'intégration existants

La figure 2.2 illustre les cinq niveaux de schémas de référence présents dans un système d'intégration, tels qu'ils sont identifiés dans [Sheth *et al.*, 1990a] :

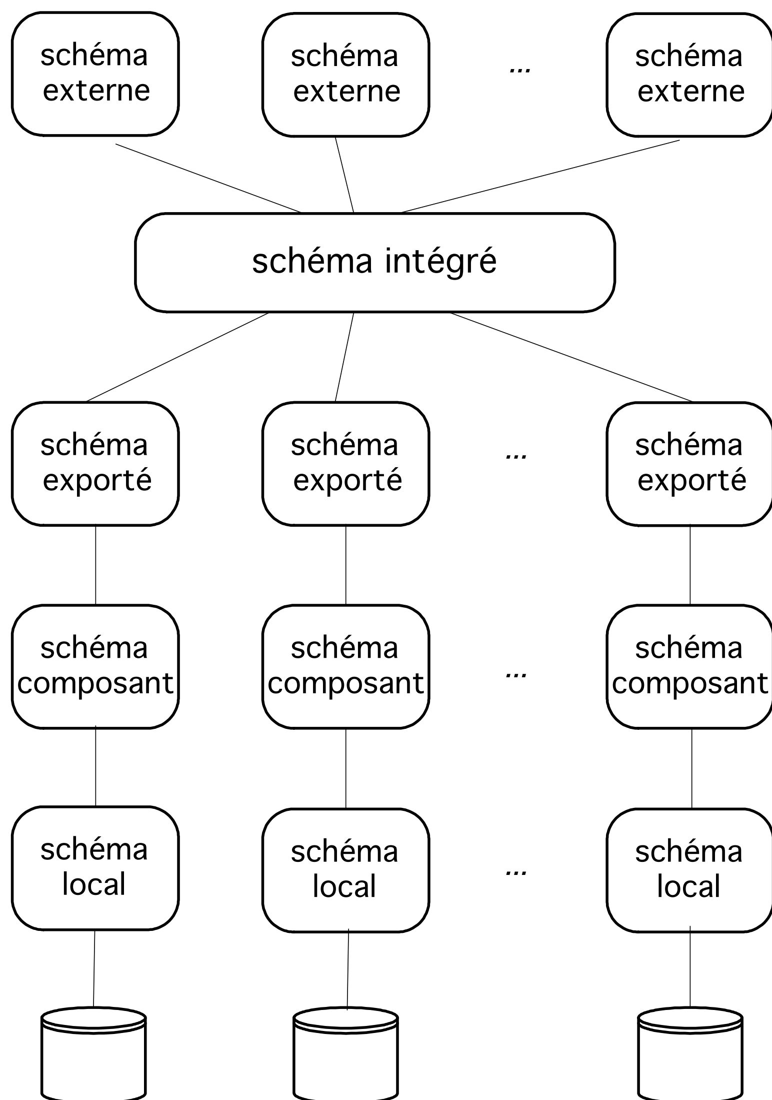


Figure 2.2. Les niveaux de schémas dans les systèmes d’intégration.

Le principe de l'intégration consiste alors à réaliser des correspondances entre les différents schémas. Les schémas appelés schémas composants[2] décrivent les schémas locaux des différentes sources dans un modèle de données commun au système qui peut être différent des modèles sources. Une source de données peut exporter la totalité ou une partie de ses données au travers d'un schéma exporté. Les différents schémas provenant de différentes sources sont unifiés et représentés dans un schéma cohérent appelé schéma intégré. Les données des schémas intégrés qui sont utilisés par les applications et/ou les utilisateurs sont représentées par des schémas vues [Sheth *et al.*, 1990b].

Généralement, selon l'approche d'intégration, le terme schéma intégré désigne le schéma global dans le cas où l'on aurait un seul schéma intégré, et le schéma fédéré dans le cas où le système est constitué de plusieurs schémas intégrés. En effet, dans la majorité des taxonomies proposées [Bouguettaya *et al.*, 1998], nous distinguons trois principales approches d'intégration :

- Approche à schéma global : dans cette première génération de systèmes (ANSI/SPARC) [Date, 1986], l'intégration est forte car les schémas exportés des sources sont intégrés en un seul schéma intégré. Par rapport à la figure 2.2, les schémas exportés sont identiques aux schémas composants. Même si cette architecture offre une transparence totale face à l'hétérogénéité et la répartition des sources, elle a été identifiée comme étant rigide ; sa mise en œuvre est difficile et elle ne prend en compte ni l'autonomie des sources ni la dynamicité des architectures.
- Approche à requêtes multibase : dans cette approche d'intégration, nous ne disposons pas de schéma intégré. Ces systèmes n'offrent pas de mécanisme de coopération et de partage d'information. L'accès aux données par les applications et/ou les utilisateurs se fait directement sur les sources au travers d'un langage de requêtes appelé langage multibases. Par rapport à la figure 2.2, cela revient à dire que les sources de données (les schémas composants) ne sont pas transparentes aux applications et/ou aux utilisateurs. Notons que dans cette approche, les sources de données sont généralement des bases de données classiques.
- Approche à schémas fédérés : dans cette approche, les sources de données exportent seulement une partie de leurs schémas. Cette approche est donc plus flexible et plus adéquate pour les systèmes actuels. De plus, les schémas intégrés peuvent être définis selon les besoins des applications et/ou des utilisateurs. Un exemple type d'une architecture basée sur cette approche est l'architecture de médiation [Wiederhold, 1999], [Wiederhold, 1992].

---

[2] Appelé component schema dans [Sheth et al., 1990b].

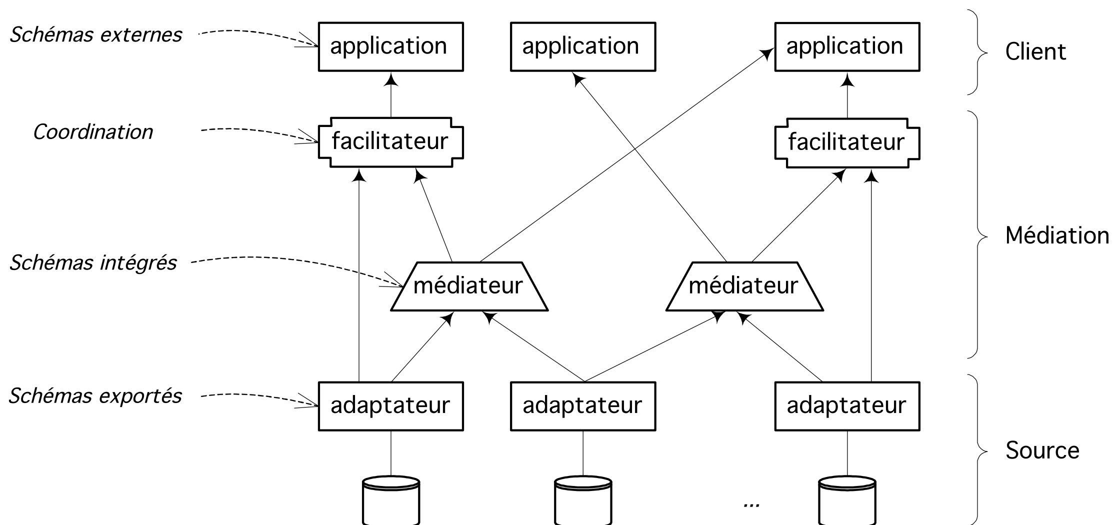


Figure 2.3. Architecture de médiation.

L'architecture de médiation (voir figure 2.3), initialement définie dans le cadre du projet DARPA [Wiederhold, 1999], [Wiederhold, 1992], est une architecture de fédération de données à trois couches :

- Couche source : cette couche regroupe l'ensemble des sources concernées par l'intégration. Chaque source est représentée par un composant appelé adaptateur (wrapper en anglais). Dans la figure 2.3, un adaptateur représente le schéma exporté d'une source. Ce composant offre un premier niveau d'intégration en permettant de réécrire les requêtes depuis le langage commun du système vers le langage natif des sources ; et inversement, de transformer le résultat des requêtes évaluées par les sources dans le modèle de données commun du système.
- Couche médiateur : cette couche, appelée aussi middleware, est constituée d'une hiérarchie de composants, dits médiateurs, et de facilitateurs. Les médiateurs sont les composants clés de l'architecture ; ils représentent le schéma intégré qui concilie l'hétérogénéité des schémas exportés des adaptateurs. Un médiateur est généralement doté d'un modèle de données et d'un langage de requêtes commun. Les facilitateurs sont des composants utilisés par les applications pour coordonner les interactions relatives aux requêtes avec les médiateurs ou les adaptateurs.
- Couche client : cette couche représente les acteurs qui utilisent les données fournies par le système, notamment les utilisateurs, les applications, les interfaces graphiques, etc.

L'architecture de médiation est une généralisation des architectures de systèmes de fédération de données. Elle ne résout pas explicitement les problèmes d'intégration de données, mais offre les composants de base qui permettent de construire des systèmes

d'intégration de données en prenant en compte la répartition, l'autonomie et l'hétérogénéité des sources. En effet, les différents composants de l'architecture, notamment les adaptateurs et les médiateurs, peuvent être largement répartis. L'ajout d'une nouvelle source de données se traduit par l'ajout d'un adaptateur à l'architecture existante. Même si le principe de médiation est le même, le partage des tâches entre médiateurs et adaptateurs varie d'un système à un autre comme nous allons le voir dans la section 2.5.

2.3 Fédération de schémas

L'intégration de données est le processus de standardisation des définitions de données et de structure de données par l'utilisation d'un schéma conceptuel commun sur une collection de sources de données [Heimbigner *et al.*, 1989].

Le processus d'intégration de données passe systématiquement par la résolution d'un certain nombre de conflits appelés conflits d'intégration. Par conséquent, l'hétérogénéité implique divers types de conflits entre les différentes sources. Principalement, il existe deux types de conflits à résoudre pour intégrer des schémas provenant de sources multiples et hétérogènes. Nous distinguons les conflits schématiques (ou syntaxiques) et les conflits sémantiques [Jarke *et al.*, 2000], [Vargun, 1999].

- Conflits schématiques : Les conflits schématiques entre les données apparaissent lorsque des concepts équivalents sont représentés différemment dans des bases de données locales. Ils sont associés aux noms, types de données, attributs, comme l'absence d'un attribut dans l'une des deux entités équivalentes appartenant à des schémas locaux différents. Nous distinguons deux cas de conflits liés aux schémas :
  - Conflits de noms : Les schémas des sources peuvent utiliser les mêmes noms pour désigner des concepts différents (cas d'homonymies) ; inversement, les sources peuvent utiliser deux noms différents pour désigner le même concept (cas de synonymie). Par exemple, si les entités auteurs des sources A et B utilisent tout deux un attribut "ville", mais que cet attribut signifie la ville de résidence pour les auteurs de la source A et la ville de naissance pour ceux de la source B alors il y a un conflit d'homonymie. En revanche, si les noms des auteurs sont représentés par un attribut "nom" dans la source A et "name" dans la source B alors nous sommes en présence d'un conflit de synonymie.
  - Conflits structurels : Lorsqu'une même information est représentée différemment dans deux sources de données, nous parlons de conflits structurels. Considérons une source A de type base de données relationnelle où les auteurs sont représentés par des tuples dans une table, et une source B de type base de données objet où les auteurs sont représentés par des objets. Cette hétérogénéité entraîne un conflit structurel car le concept d'auteur est représenté dans des modèles de données différents. Ce type de conflit se décline souvent en conflit de typage où la même information a des types différents selon les sources.

- Conflits sémantiques : Les conflits sémantiques sont liés au fait que les données des différentes sources peuvent être soumises à des interprétations diverses. Nous distinguons :
  - Conflits de nommage : Les conflits de nommage sont dus au fait que les sources utilisent des conventions de nommage distinctes. Les données sont alors décrites dans des formats différents. Des exemples classiques de ce type de conflits sont le format des dates (par exemple, "22/09/2006" et "2006-09-22") et les sigles (par exemple, "LIASD" et "Laboratoire d'Informatique Avancée de Saint Denis").
  - Conflits de mesure[3] : Les sources de données peuvent utiliser des unités de mesures différentes pour quantifier la même information. Par exemple, une source utilise le dollar pour estimer les prix des livres, alors que l'autre source utilise l'euro. Nous pouvons citer d'autres exemples : centigrade Celsius et Fahrenheit ou encore mètres, centimètres et pouces.

L'hétérogénéité au niveau schéma couvre seulement une série limitée des types de conflits liée sans doute à la structure des bases de données. Cependant, il existe une variété de conflits liés à l'interopérabilité et/ou à l'intégration de multiple bases de données générés par des Système de Gestion de Bases de données (SGBD) traditionnels [Kim *et al.*, 1993]. [Sheth, 1999] propose une autre classification de ces conflits impliqués dans l'interopérabilité, nous résumons ici les plus importants :

- Incompatibilité des domaines de définition (conflits de noms, conflits de représentations, conflits de contraintes d'intégrité, etc.).
- Incompatibilité des valeurs de données (inconsistance connue, temporelle et acceptable).
- Incompatibilité au niveau des abstractions (conflits de généralisation et d'agrégation).
- Incompatibilité schématique (conflits des attributs et des entités).

2.3.1 Intégration par les vues

Deux approches classiques permettent de décrire les règles d'appariements entre les différents schémas d'un système, et particulièrement entre les schémas exportés et les schémas intégrés : l'approche GAV (Gobal As View) [Garcia-Molina *et al.*, 1997] et l'approche LAV (Local As View) [Levy, 2001].

---

[3] Appelé aussi conflit d'échelle.

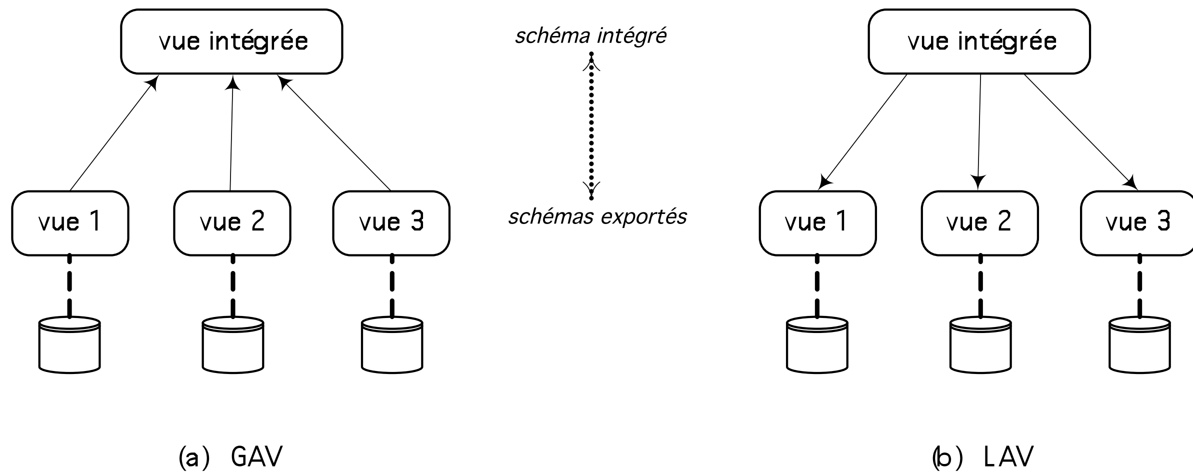


Figure 2.4. Approche GAV vs LAV.

Ces deux approches se basent sur le mécanisme de vues (figure 2.4). L'utilisation de ces dernières dans l'intégration apporte beaucoup de flexibilité [Abiteboul *et al.*, 1991], [Abiteboul *et al.*, 1999]. Elles permettent d'avoir plusieurs points de vues sur les sources de données, et sont aussi utilisées pour préserver la confidentialité des données entre les utilisateurs. Les vues sont alors utilisées pour représenter les différents schémas, notamment les schémas exportés et les schémas intégrés[4].

- Global-As-View : propose de définir le schéma global comme une vue des schémas locaux. Cette approche est simple pour la mise en œuvre des filtres mais tout changement intervenant dans les sources locales, tel que l'ajout d'une source, remet en cause la définition du schéma global et par conséquent celle des filtres. Cette approche est utilisée par exemple dans les projet TSIMMIS [Garcia-Molina *et al.*, 1997], e-XMLMedia [eXMLMedia, 1999].
- Local-As-View : propose de définir le schéma global suivant les besoins des utilisateurs et de définir les schémas locaux comme des vues sur le schéma global. Cette approche est plus complexe à mettre en œuvre que la précédente mais ne nécessite pas de redéfinition du schéma global en cas de changement intervenant dans les sources locales, chaque source étant intégrée indépendamment des autres. Cette approche est utilisée notamment dans Information Manifold [Kirk *et al.*, 1995], AGORA [Manolescu *et al.*, 2001], et XPERANTO [Carey *et al.*, 2000].

Ces deux approches ont toutes deux des points forts et des points faibles. Le principal avantage de l'approche GAV est la simplicité de la réécriture des requêtes (cf. section 2.4). Son inconvénient est qu'elle n'est pas performante lorsqu'il s'agit de construire un système d'intégration où les schémas des sources évoluent constamment. En effet, l'ajout d'une

[4] Les vues représentant les schémas intégrés sont aussi connues sous le nom de vues intégrées.

source de données demeure une tâche difficile et demande de considérer les relations de cette source avec les sources existantes, et de modifier les règles de correspondance. A l'opposé de l'approche GAV, l'approche LAV est appropriée dans les systèmes d'intégration de données avec des schémas évolutifs. L'ajout d'une source se fait indépendamment des autres sources de données. Cependant, la réécriture des requêtes nécessite des algorithmes plus compliqués. Notons qu'il existe aussi des approches hybrides, appelées GLAV, qui associent les deux approches LAV et GAV et qui permettent de passer d'une approche à l'autre [Cali *et al.*, 2002].

Selon la matérialisation ou non des données des vues, nous distinguons les systèmes d'intégration qui se basent sur les vues virtuelles et les systèmes d'intégration qui matérialisent les vues. Le deuxième type de système regroupe les systèmes de fouilles de données, c'est à dire les entrepôts de données (datawerehouse). L'entrepôt est construit en matérialisant les résultats des vues utilisées pour intégrer les sources. Cette matérialisation permet d'accélérer le traitement des requêtes car il n'est pas nécessaire d'accéder aux sources pour y répondre. Cependant, comme dans l'approche virtuelle il est nécessaire de réécrire les requêtes sur les vues matérialisées. Citons par exemple le cas de Xylème qui est un entrepôt de données basé sur XML, proposé par la société du même nom, fondé par une équipe de l'INRIA en 2000 [Abiteboul, 2004]. Xylème permet l'intégration et par conséquent l'exploitation de documents semi-structurés XML quelle que soit leur structuration.

L'avantage de la solution de la médiation par rapport à celle de l'entrepôt de données tient au fait qu'il n'y a plus de problème de taille de la base de données et de problème de cohérence. Les données restent en effet maintenues par les sources locales ; le médiateur se contente alors de maintenir le schéma global et les traducteurs permettent de réaliser les interrogations. Cependant, avec cette solution, les temps de recherche augmentent car le nombre de traitements à effectuer est plus important. En effet, le médiateur doit traduire les requêtes exprimées par les clients à l'aide du schéma global et du langage d'interrogation de la médiation pour qu'elles soient compréhensibles par les sources locales, et ceci, pour chacune des sources locales auxquelles il va s'adresser. Il doit ensuite communiquer les requêtes à chacune des sources locales, traduire et agréger les réponses qu'elles lui retournent pour présenter aux clients une réponse homogène exprimée à l'aide du schéma global.

2.3.2 Modèles communs d'intégration de données

Chaque système d'intégration fédéré utilise un modèle de données commun qui permet de représenter les données des différentes sources et l'interopérabilité entre les différents composants du système. Le modèle de données utilisé doit être assez expressif pour couvrir les modèles de données des sources de données sous-jacentes. La conversion des données des sources dans un modèle commun permet de prendre en compte les conflits structurels entre les sources. Plusieurs standards de modèles de données existants ont été utilisés comme modèle commun d'intégration, notamment les modèles relationnels, les modèles objets et les modèles semi-structurés :

- Le modèle de données relationnel : Les données d'une base de données relationnelle sont représentées sous forme d'une table structurelle [Codd, 1970], [Codd, 1972]. Le modèle de type est plat, dans le sens où un tuple contient des attributs de types atomiques. Les valeurs que peut prendre un attribut sont appelées valeurs d'un domaine. Un attribut peut aussi ne pas avoir de valeur. Les systèmes MAIRMED [Bright *et al.*, 1992] et XRERANTO [Carey *et al.*, 2000] utilisent le modèle relationnel pour l'intégration de données. Les vues relationnelles sont définies en utilisant le langage SQL.
- Le modèle de données objet : Le modèle objet, concurrent du modèle relationnel, fournit un pouvoir d'expression très puissant de modélisation en prenant en compte implicitement les contraintes d'intégrité référentielles. Bien que le modèle objet de l'ODMG (Object Database Management Group) [Eastman *et al.*, 1999] soit désigné comme le standard des bases de données objets, il n'existe pas de modèle objet universel accepté par toute la communauté. Le modèle objet a été adopté dans plusieurs systèmes d'intégration, notamment dans DISCO [Tomasic *et al.*, 1998], IRO-DB [Fankhausser *et al.*, 1998] et AMOS II [Risch *et al.*, 2001]. Les vues dans le modèle objet sont construites à partir d'opérateurs spéciaux (union, sélection, intersection, etc.) qui permettent la création de nouveaux types d'objets à partir d'autres types [Lacroix *et al.*, 1997].
- Le modèle semi-structuré : Les données semi-structurées sont des données qui s'auto-décrivent. Une source de données semi-structurée contient à la fois les données et la structure de ces données de manière non séparée. Cette caractéristique implique que ces données ne soient pas dotées d'un système de gestion ; ce qui change l'ordre établi dans la vision des modèles de données structurées. Il existe plusieurs systèmes qui reposent sur ce modèle, tels que TSIMMIS [Garcia-Molina *et al.*, 1997], YAT [Cluet *et al.*, 1998], STRUDEL [Fernandez *et al.*, 1998] qui sont représentés sous forme de graphes ou d'arbres. D'autres systèmes tel que Xylème [Abiteboul *et al.*, 2001] et MIX [Bornhovd, 1998] adoptent le modèle de données XML. La définition des vues sur les données semi-structurées a fait l'objet de plusieurs travaux de recherches [Milo *et al.*, 1999], [Cluet *et al.*, 2001].

2.4 Traitement de requêtes

Un système d'intégration de données permet d'interroger d'une manière uniforme et plus ou moins transparente plusieurs sources de données à travers un langage de requêtes [Zackari *et al.*, 1999]. L'un des objectifs majeur d'un système d'intégration est l'accès associatif transparent aux données intégrées représentées par le schéma intégré. Cet accès se fait simplement à travers un langage de requêtes. La figure 2.5 représente les différentes phases de traitement d'une requête globale définie sur le schéma intégré.

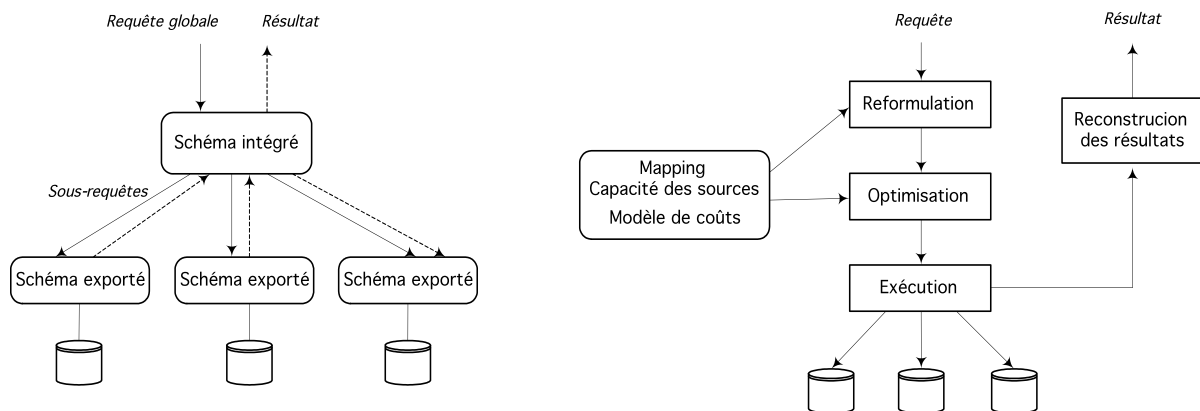


Figure 2.5. Traitement de requêtes dans les systèmes d'intégration.

En se basant sur les métadonnées de mapping entre le schéma intégré et les schémas exportés (ou encore entre les vues intégrées et les vues exportées), la phase de reformulation a pour objectif de produire des requêtes qui portent sur les schémas exportés des sources (i.e. chaque feuille de l'arbre de requête porte sur un adaptateur conformément à l'architecture de médiation). L'algorithme de décomposition ou de reformulation dépend de l'approche d'intégration entre les schémas exportés et intégrés adopté par le système [Lenzerini, 2002] :

- Cas de l'approche GAV : Dans cette approche, l'algorithme de reformulation est très simple. Il consiste à remplacer les vues intégrées de la requête globale par les requêtes de ces vues. Le résultat sera alors un arbre de requêtes où les feuilles représentent les vues exportées des sources [Li *et al.*, 2001].
- Cas de l'approche LAV : La reformulation des requêtes dans ce type de mapping est connue sous le nom de réécriture des requêtes en utilisant les vues (Answering queries using views). Pour savoir si une requête est incluse dans une autre, il faut pouvoir reformuler la deuxième en fonction de la première. Plusieurs algorithmes ont été proposés : l'algorithme bucket [Levy, 2001], puis l'algorithme inverse rule [Levy, 2001] qui a été proposé pour le système d'intégration Infomaster [Duscika *et al.*, 1997] et enfin l'algorithme MiniCon [Pottinger *et al.*, 2000] qui englobe les deux autres algorithmes et qui supporte le passage à l'échelle des vues.

Notons que la résolution des requêtes en utilisant les vues est aussi utilisée dans d'autres contextes à savoir l'optimisation des requêtes, les entrepôts de données et les caches sémantiques. Par exemple, dans les architectures client/serveur, elle est utilisée pour résoudre la requête cliente en utilisant les caches de données locales (matérialisées), qui peuvent être le résultat d'autres requêtes.

2.5 Architecture des systèmes existants

2.5.1 Architectures de médiations

Il existe de nombreux systèmes d'intégration de données. Dans ce qui suit, nous présentons de façon non exhaustive les systèmes d'intégration existants. Ces derniers nous ont paru intéressants pour les problématiques qu'ils soulèvent ou pour les solutions envisagées. Pour chaque système, nous présentons ses objectifs et ses apports majeurs ainsi que son architecture.

2.5.1.1 TSIMMIS

TSIMMIS (The Stanford-IBM Manager of Multiple Information Sources) [Chawathe *et al.*, 1994], [Garcia-Molina *et al.*, 1997]. Ce projet fait partie des pionniers dans la médiation des données structurées et semi-structurées. Il utilise une hiérarchie de médiateurs pour intégrer les sources de données hétérogènes. Il a introduit le formalisme OEM (Object Exchange Model) comme modèle commun où chaque source est transformée par un adaptateur pour fournir des données OEM aux médiateurs. La figure 2.6 montre l'architecture de TSIMMIS qui est composée des modules suivants :

- Classificateurs / extracteurs : ces composants permettent d'identifier les données provenant des sources non structurées tels que fichiers textes et pages Web HTML.
- Adaptateurs (wrapper sur la figure 2.6) : ils permettent la transformation des requêtes écrites dans le langage OEM-QML vers le langage spécifique des sources. Il convertit ensuite le résultat de la requête des sources dans le modèle de données OEM.
- Médiateurs : les médiateurs représentent des vues intégrées qui sont le résultat de la combinaison des vues des différentes sources ou d'autres médiateurs.
- Générateurs : Les adaptateurs et les médiateurs sont générés automatiquement à partir des définitions respectivement grâce aux deux composants : générateur de médiateur et générateur de wrapper.

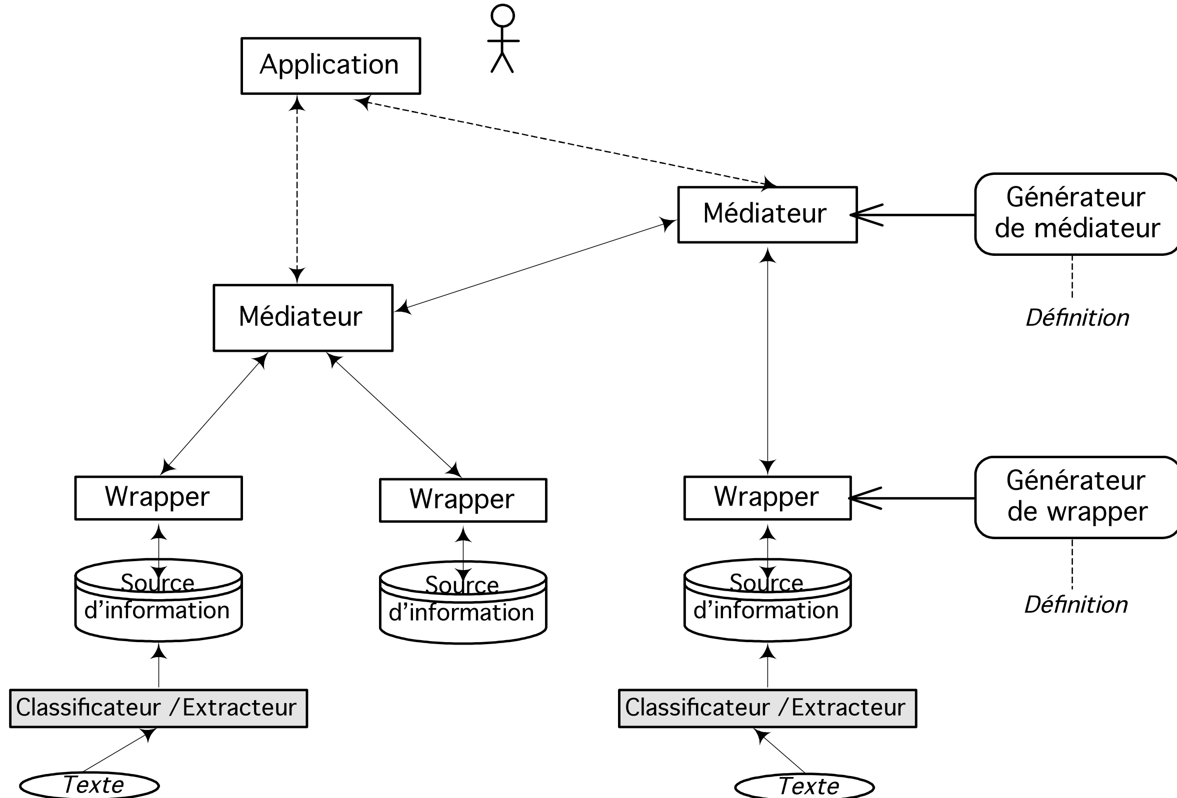


Figure 2.6. Architecture du système TSIMMIS.

TSIMMIS a été ensuite adapté dans l'approche GARLIC [Haas *et al.*, 1997] et LORE [Abiteboul *et al.*, 1997b] qui ont fait évoluer son format de données OEM vers XML [Goldman *et al.*, 1999]. Il s'est spécialement focalisé sur l'intégration des données multimédia (images, vidéos, cartes géographiques, textes, etc.). Le modèle de données adopté est le modèle orienté-objet de l'ODMG avec un langage de requêtes appelé GQL (GARLIC Query Language). L'apport de GARLIC est double, d'une part au niveau optimisation de requêtes et d'autre part au niveau architectural. Pour une requête donnée, l'optimiseur du médiateur coopère avec les wrappers (ou adaptateurs en français) concernés pour la recherche du plan en se basant sur les coûts et en prenant en compte les capacités des sources.

2.5.1.2 MIX

MIX (Mediation of Information Using XML) [Bornhovd, 1998] est un projet collaboratif entre l'UCSD Database Laboratory et le groupe Data Intensive Computing Environment de SDSC. L'objectif de MIX est de développer un système pour l'intégration de sources hétérogènes en utilisant XML [MIX, 1999]. L'idée de ce projet est de considérer

le Web comme une grande base de données distribuée et XML comme son modèle d'échange et XMAS (XML Matching And Structuring language) comme méta-langage de requêtes. Les acteurs du projet pensaient que dans un futur proche la majorité des sources de données permettraient l'exportation de leurs données au format XML. Cette hypothèse semble se vérifier car de nombreux logiciels le permettent aujourd'hui [Mignet *et al.*, 2003].

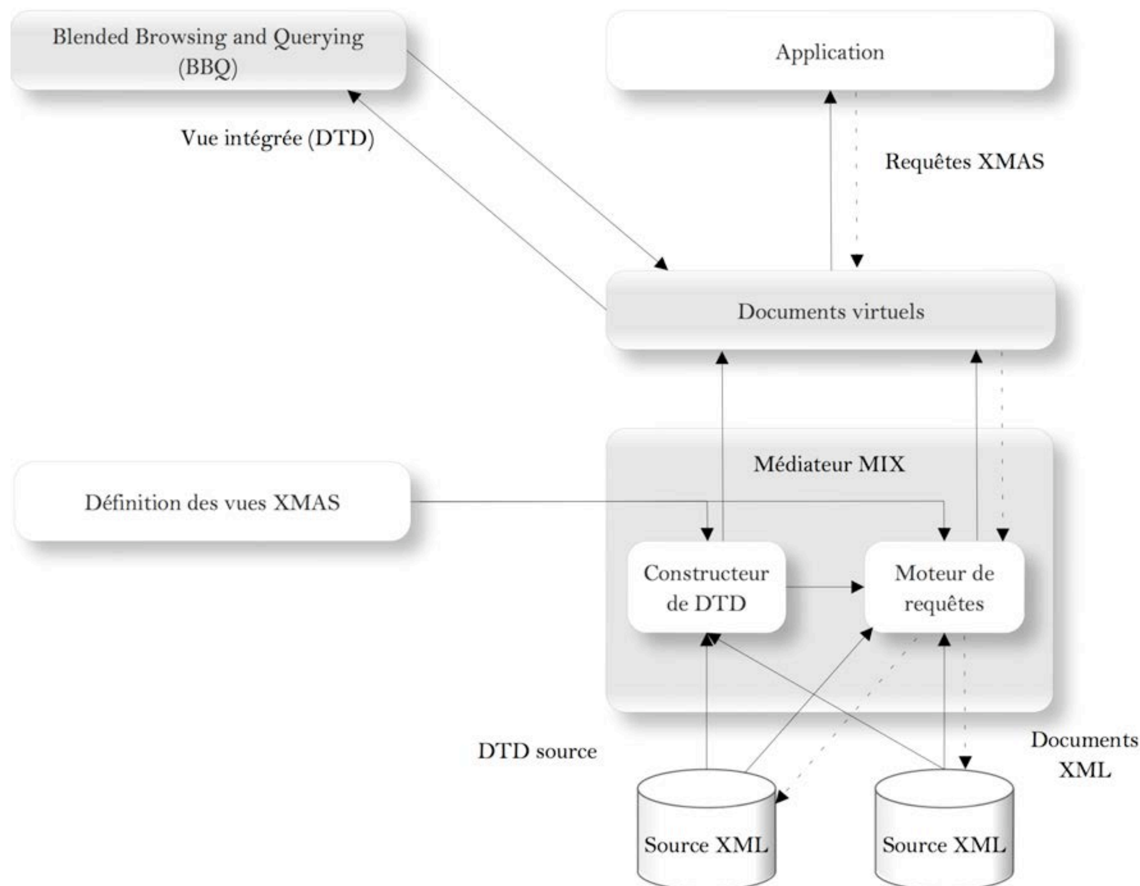


Figure 2.7. Architecture du système MIX.

Tel que le montre la figure 2.7, l'architecture de MIX est composée des modules suivants :

- Documents virtuels : constitue l'interface du système. Il utilise les DTD générés par les vues pour offrir aux utilisateurs du système des documents virtuels. Ces documents sont présentés sous forme d'arbres DOM virtuels qui permettent de fournir aux utilisateurs des documents XML virtuels.
- Médiateur MIX : constitue le noyau du système. Grâce à ce composant, le schéma médiateur est créé, et les requêtes envoyées à l'interface sont traitées. Pour cela, il est composé de deux modules à savoir *Constructeur de DTD* et *Moteur de requêtes.*

- Constructeur de DTD : est chargé de construire les DTD à partir de la définition des vues. Ces DTD permettent à l'interface (*Documents virtuels*) de fournir le schéma médiateur du système.
- Moteur de requêtes : est chargé du traitement des requêtes. Il décompose les requêtes sur les différentes sources pour les exécuter. Le résultat des requêtes sur ces sources est envoyé à l'interface afin d'instancier les documents XML virtuels qu'elle fournit.
- Blended Browsing and Querying (BBQ) : Cette interface graphique définit des vues ou des requêtes avec le langage XMAS. BBQ se base sur les DTD fournies par les sources afin de construire sa requête.

2.5.1.3 DISCO

DISCO (Distributed Information Search Component) a été développé dans les laboratoires de l'INRIA [Tomsic *et al.*, 1998]. C'est un système de médiation doté d'un modèle de données orienté-objet et du langage de requêtes OQL. Les principaux apports et les caractéristiques de DISCO sont les suivantes :

- L'intégration de données au niveau du médiateur est réalisée par des vues intégrées objets. Ces vues sont spécifiées selon l'approche GAV en utilisant le langage OQL et ne prennent pas en compte tous les conflits d'intégration.
- Pour l'exécution de requêtes sur plusieurs sources, DISCO a proposé des techniques et des algorithmes qui permettent de prendre en compte l'indisponibilité des sources au moment de l'évaluation en récupérant les résultats intermédiaires [Bonnet, 1999].

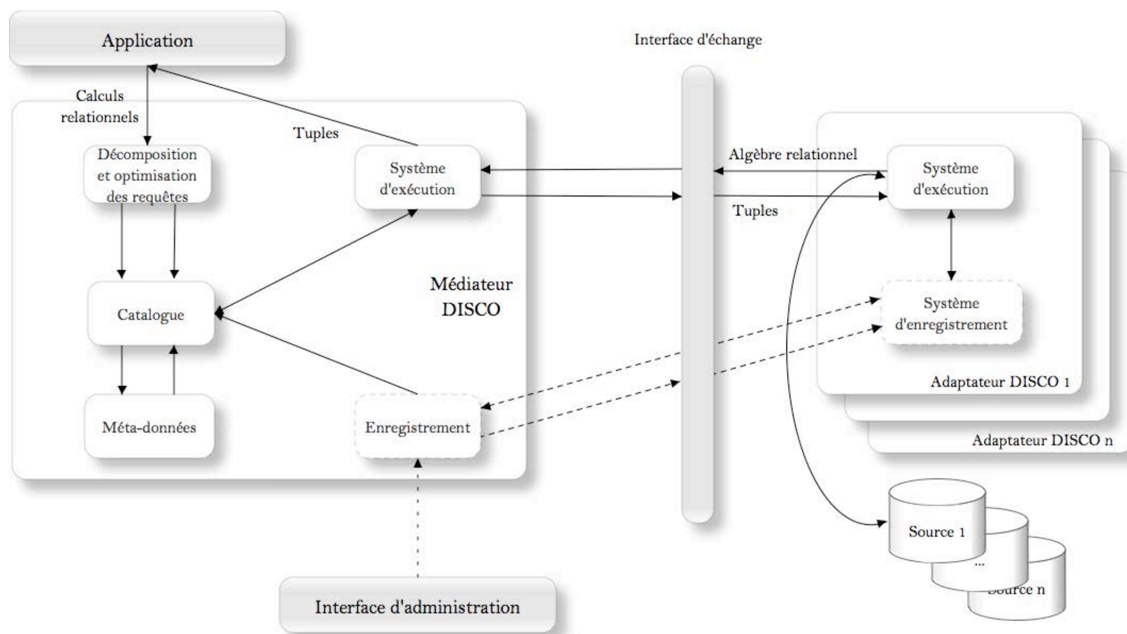


Figure 2.8. Architecture du système DISCO.

L'architecture fonctionnelle de DISCO, représentée par la figure 2.8 est composée de deux phases à savoir :

- Enregistrement des sources : durant cette phase, le médiateur fait appel à l'adaptateur localisé par l'administrateur. L'adaptateur envoie la description des sources contenant les capacités de traitement de ces l'adaptateurs, ainsi que les informations sur les coûts[5]. Ces informations permettent au médiateur d'évaluer le coût d'une requête avant son exécution.
- Traitement des requêtes : le médiateur récupère une requête d'une application ; cette requête est envoyée au module d'analyse sémantique et d'optimisation, qui analyse la requête afin de vérifier sa cohérence. Une fois la requête vérifiée, elle est décomposée en sous-requêtes envoyées à l'adaptateur. Enfin, l'exécution de la requête passe par trois étapes, (i) connexion adaptateur /médiateur, (ii) envoi et exécution des sous-requêtes par les adaptateurs et (iii) recomposition des résultats.

2.5.1.4 YAT

YAT est un système de médiation développé par l'équipe Verso de l'INRIA [Cluet *et al.*, 1998]. Il vise spécialement à définir des modèles et des langages pour l'intégration de données du Web. Il se base sur un modèle de données sous forme de graphe afin de représenter les données semi-structurées provenant des différentes sources. YAT possède un langage à base de règles appelé YATL [Cluet et al, 2000]. Ce langage offre surtout des possibilités de restructuration, de conversion de données et de fonctions Skolem pour manipuler des identifiants et créer des graphes complexes.

Dans l'architecture de YAT, le médiateur est représenté par le composant Runtime environment (voir figure 2.9). Ce composant utilise les wrappers pour importer les données, puis les exporter via d'autres wrappers. Ce médiateur est composé de :

- Patterns YAT / Module de gestion des règles YATL : permet de gérer l'ensemble des règles et des patterns du système.
- Fonctions externes et évaluation de prédicats : offre des fonctionnalités d'évaluation et des prédicats des programmes.
- Vérificateur de types : permet la vérification et l'inférence de types.
- Interpréteur YATL : équivalent à un évaluateur de requêtes, il est chargé de l'exécution des opérateurs et des fonctions de calcul.

[5] Les coûts et les statistiques sur les données des sources sont exportées par les wrappers en se basant sur un modèle de coûts proposé dans [Naacke et *al.*, 1998] pour les données objets.

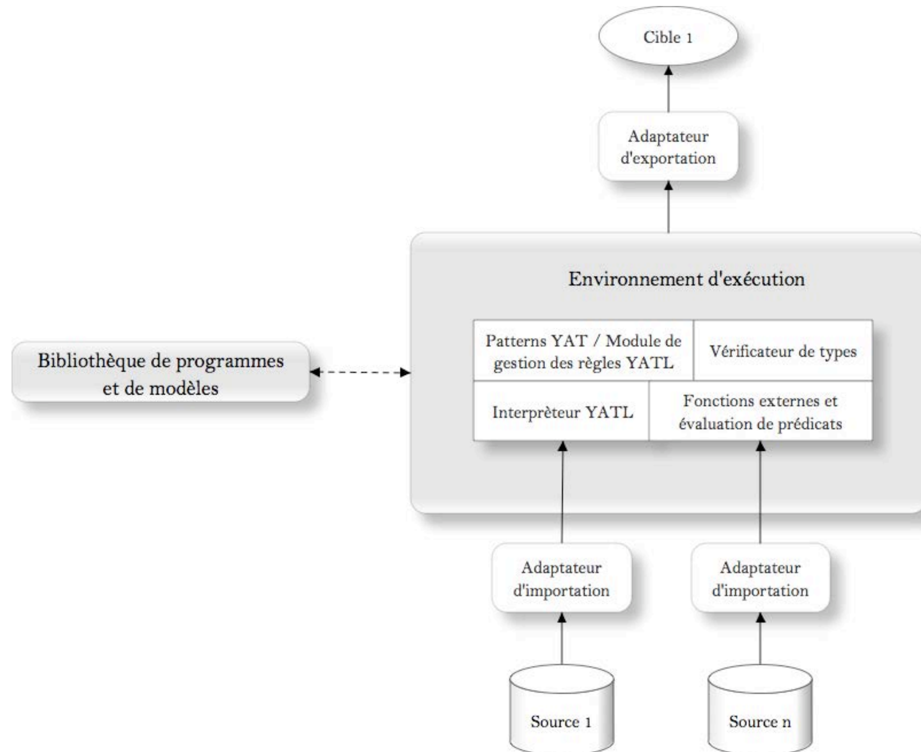


Figure 2.9. Architecture du système YAT.

2.5.2 Systèmes d'intégration industriels

Parallèlement aux projets de recherche, beaucoup d'entreprises ont développé des systèmes d'intégration de données basés sur des modèles de données existants. Nous présentons ici quelques produits permettant de construire des systèmes d'intégration de données basés sur une architecture de médiation.

2.5.2.1 e-XMLMedia

Le projet e-XMLMedia [eXMLMedia] avait pour objectif de développer et de distribuer des composants logiciels pour faciliter le développement d'applications intégrant des sources de données en utilisant XML comme standard d'échange. Ce système trouve son origine dans le projet Miro-Web [Fankhauser *et al.*, 1998]. Il est constitué de trois composants s'intégrant dans une architecture distribuée pour publier, échanger, stocker et interroger des documents XML dans un système d'information [Gardarin *et al.*, 2002] à savoir :

- XMLizer : permet de répondre à deux objectifs. En amont, il se charge d'extraire des données en provenance de documents XML puis de les intégrer au sein d'un SGBD relationnel, le tout en respectant le mode de description initial des éléments de contenu ainsi que les liens (père/fils) qu'ils entretiennent au sein du fichier XML. Ce composant qui joue le rôle d'adaptateur permet de résoudre les problèmes d'intégration syntaxique tels que certains conflits de noms.
- e-XML Repository : permet de stocker et d'interroger des documents XML dans un SGBD relationnel. Ce mapping est réalisé selon deux approches, générique et orienté-schéma. Dans l'approche dite générique, un schéma relationnel est créé à partir des documents XML et les données sont rangées dans des tables suivant des directives automatiquement créées. Dans la seconde approche dite orienté-schéma, les documents XML sont stockés dans un schéma existant suivant des directives qui sont données par l'administrateur de la base. e-XML Repository permet également d'interroger les documents en SQL ou en XQuery.
- e-XML Mediator : permet d'exécuter des requêtes sur des sources de données XML multiples et hétérogènes. Des métadonnées sont ajoutées afin de localiser de façon transparente les données dans les sources. En fait, ces métadonnées constituent des vues locales qui définissent les sources pour les intégrer dans un schéma selon une approche LAV. Le composant Médiateur e-XML se connecte aux sources de données via des adaptateurs qui assurent la traduction des données du format natif de chaque source vers XML ainsi que la traduction de la requête XML vers le langage de requête natif de la source. Le composant possède également une interface graphique pour l'aide à la spécification des requêtes.

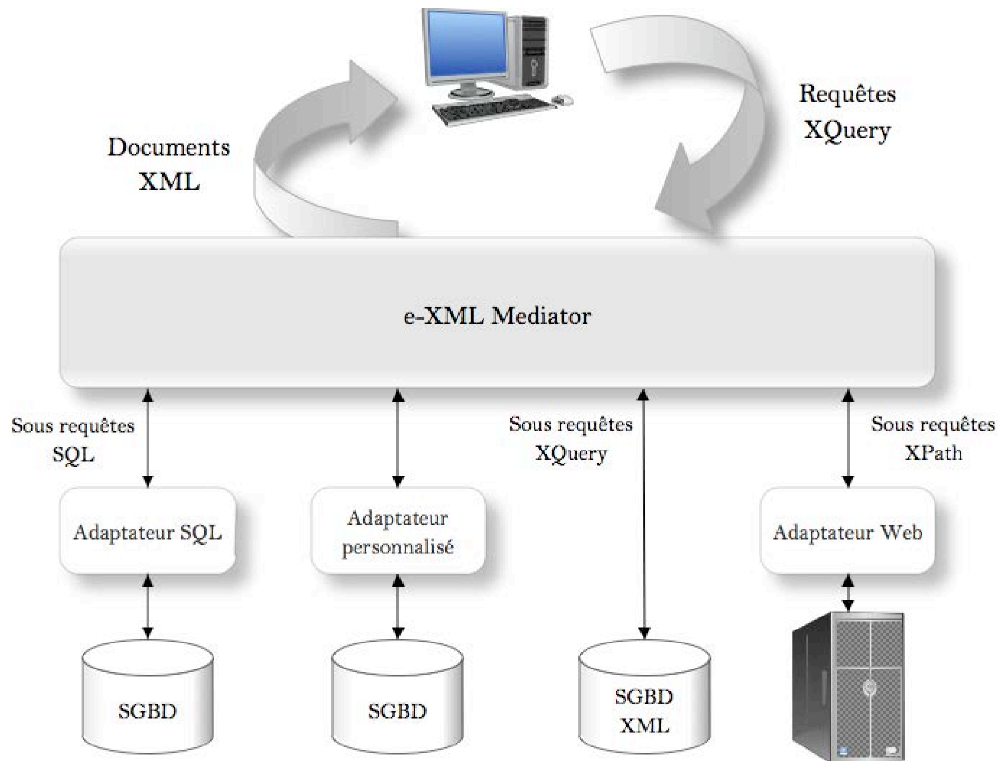


Figure 2.10. Architecture du système e-XMLMedia.

La force de cette approche réside d'une part dans sa capacité d'intégration de sources hétérogènes et d'autre part dans sa capacité à produire du XML structuré selon les besoins des utilisateurs à partir de toute source de données.

2.5.2.2 XPERANTO

XPERANTO (Xml Publishing of Entities, Relationships And Typed Object) [Carey at al., 2000] est conçu par IBM Almaden Research Center. C'est une interface d'interrogation (utilisant la syntaxe XML-QL) et de génération automatique de vues XML à partir de bases de données de type objet-relationnel. Une vue d'une base de données est une description de son schéma en utilisant des schémas XML du W3C [W3C, 2001a]. Cette vue n'est utilisée que par l'administrateur qui définit, grâce au langage de requêtes XML-QL, d'autres vues plus restrictives pour les utilisateurs. Les requêtes spécifiées sur la vue XML sont transformées dans une représentation intermédiaire appelé XQGM (XML Query Graph Model), langage neutre de représentation intermédiaire de requêtes sur XML, puis en SQL et exécutées par le langage de requêtes de la base de données.

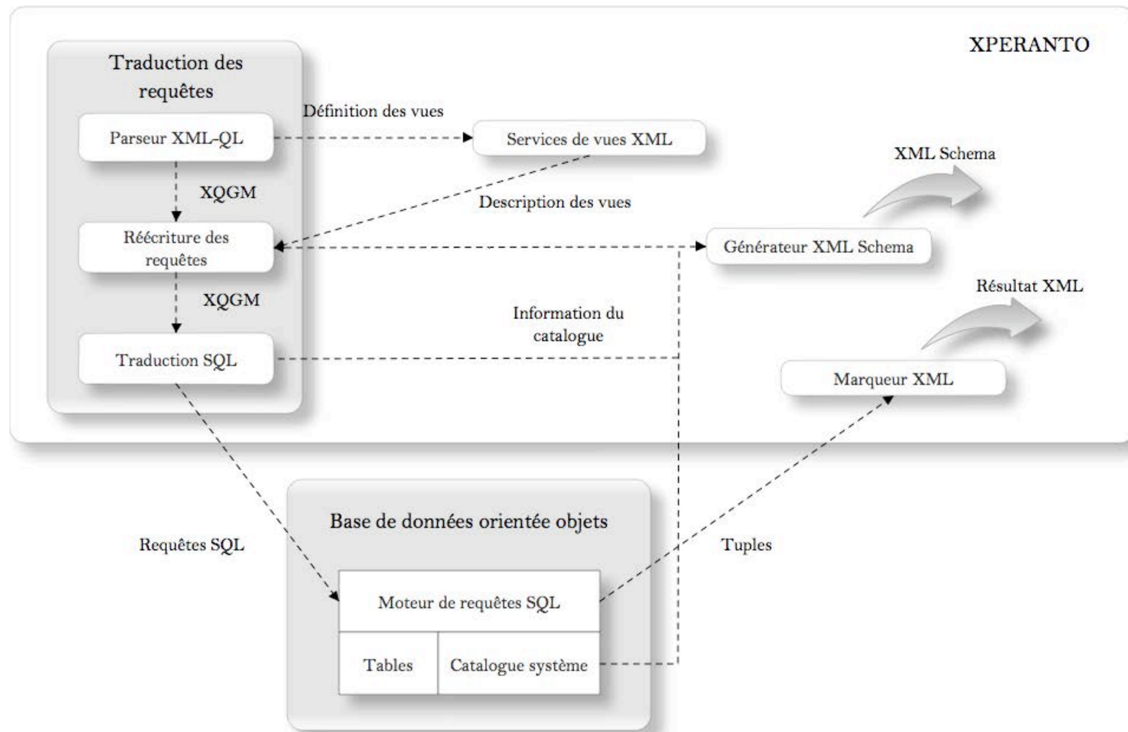


Figure 2.11. Architecture du système XPERANTO.

Nous pouvons constater sur la figure 2.11 que l'architecture de XPERANTO est composée des modules suivants :

- Query translation : traduit les requêtes XML-QL en SQL natif pour les SGBD-OR sous-jacents. Il est constitué (i) d'un analyseur XML-QL (XML-QL Parser) qui génère une représentation XQGM à partir d'une requête XML-QL, (ii) d'un composant de réécriture de requêtes (Query Rewrite) qui traduit la représentation XQGM en une représentation sémantiquement équivalente, après avoir résolu les références sur les vues et la composition des vues XML [W3C, 1998], (iii) d'un traducteur SQL (SQL Translation) qui traduit enfin la représentation XQGM en instructions SQL.
- XML View Services : stocke et charge les données pour la définition de vues XML-QL. Une fois ces vues définies, elles seront stockées dans des tables dédiées.
- XML Schema Generator : prend les informations du catalogue des bases de données en produisant des informations pour les vues et résultats de requêtes XML.
- XML Tagger : a pour rôle de convertir le résultat SQL (structure tabulaire) en un document XML structuré.

2.6 XML et les systèmes d'intégration de données

2.6.1 Apport d'un modèle unifié

Dans les systèmes de médiation, les besoins des utilisateurs sont représentés par un schéma de médiation pouvant être créé à partir des schémas des sources de données, ou indépendamment des sources par des experts du domaine. Certains problèmes liés à l'hétérogénéité du format de ces sources sont résolus grâce à l'utilisation d'un modèle commun. Ce dernier doit permettre la représentation de données provenant de sources hétérogènes pour les intégrer dans un schéma médiateur.

Afin de faciliter cette intégration, il faut procéder à la traduction du schéma. Ce processus de pré-intégration de schéma consiste à traduire les schémas des différentes sources dans un même modèle. La traduction des schémas sources dans le modèle commun est capitale en vue de leur intégration [Bellahsène *et al.*, 2001].

Dans ce qui suit, nous présentons les propriétés que doit posséder un tel modèle [Pitoura *et al.*, 1995]. Il doit être :

- Suffisamment puissant pour exprimer les modèles existants et pour capturer la sémantique exprimée explicitement et implicitement dans les différents schémas sources de données.
- Flexible pour pouvoir intégrer de nouveaux modèles.
- Minimal afin de comprendre un ensemble de constructeurs permettant aux différents modèles de réaliser des combinaisons.
- Simple pour faciliter la création de schéma médiateur.

Après la traduction des différents schémas dans un modèle commun, la phase d'intégration de schémas pourrait se résumer à faire l'union des schémas des sources si les mêmes concepts n'étaient pas représentés dans les différentes sources.

2.6.2 Données semi-structurées et XML

Les données semi-structurées sont des données qui suivent une certaine structure, mais dont la structure peut ne pas être rigide, normale ou complète. De plus, ces données ne se conforment généralement pas à un schéma fixe. C'est la raison pour laquelle ces données sont qualifiées de « sans schéma » ou auto-descriptives. Dans les données semi-structurées, les informations normalement associées à un schéma sont contenues dans les données elles-mêmes. Certaines formes de données semi-structurées n'ont pas de schéma séparé, d'autres

en possède mais n'impose que des contraintes faibles sur les données. Au contraire, les SGBD relationnels exigent un schéma prédéfini orienté table et toutes les données gérées par le système doivent respecter cette structure. Bien que les SGBD orientés objet permettent de définir une structure plus riche que les SGBD relationnels, ils requièrent encore l'adhésion de toutes les données à un schéma (orienté objet) prédéfini. Cependant, dans le cas d'un SGBD fondé sur des données semi-structurées et non basé sur une architecture XML [Schöning *et al.*, 2000], le schéma est découvert à partir des données et non imposé a priori. Actuellement, les données semi-structurées sont devenues de plus en plus importantes parce que :

- Les sources du Web sont considérées comme une base de données, mais sans être contraintes par un schéma.
- Un format fixe est mis à disposition pour l'échange de données parmi des bases de données disparates.
- XML devient un standard de représentation et d'échange des données sur le Web.

Les approches de la gestion des données semi-structurées sont fondées, pour la plupart, sur des langages de requêtes qui parcourent une représentation en forme d'arbre étiqueté. Sans schéma, nous ne pouvons identifier les données qu'en spécifiant leurs positions au sein de leur collection au lieu de citer des propriétés structurelles. Ceci signifie que les requêtes perdent leur nature déclarative traditionnelle et qu'elles perdent de leur aspect de navigation.

XML (eXtensible Markup Language) [W3C, 1998] est le résultat de la coopération d'un grand nombre d'entreprises et de chercheurs partenaires du consortium W3C. Leur objectif était de définir un formalisme permettant d'échanger facilement des documents complexes sur le Web, en dépassant les limites imposées par HTML. La norme existante pour l'échange de documents structurés, SGML (Standard Generalyzed Markup Language) [Goldfarb, 1991] (ISO 1986) s'est révélé trop complexe pour que des navigateurs puissent être facilement implémentés par les industriels [W3C, 1997]. XML permet de s'adapter à quasiment tous les domaines où nous avons besoin de structurer de l'information de façon portable.

XML influence déjà de nombreux aspects des technologies de l'information, avec notamment des interfaces graphiques, des systèmes embarqués, des systèmes distribués et des systèmes de gestion de bases de données. Par exemple, comme XML décrit la structure des données, il constitue un mécanisme bien utile de définition de la structure de bases de données hétérogènes et de sources de données diverses.

Dans ce qui suit, nous énumérons quelques uns des avantages de XML :

- Standard ouvert, indépendant de toute plateforme et non propriétaire : XML est indépendant autant de la plateforme que du fournisseur ce qui lui permet d'accepter du texte utilisant n'importe quel alphabet ; il comporte une méthode qui indique le langage et le codage utilisés.

- Extensibilité / réutilisation : A l'inverse d'HTML, XML est extensible par nature, ce qui permet aux utilisateurs de définir leurs propres balises en fonction des exigences de leurs applications particulières. L'extensibilité autorise aussi la mise en place de bibliothèques de balises XML et leur réutilisation dans de nombreuses applications.
- Séparation du contenu et de la présentation : XML sépare le contenu d'un document de la manière dont ce document sera présenté. Ceci facilite l'adaptation de la vue des données. XML est considéré comme un langage à « unique écriture, publication partout » car des facilités sont offertes, telles que les feuilles de style qui permettent de publier un même document de différentes manières, en fonction de différents formats et de divers médias.
- Support de l'intégration des données de sources multiples : La capacité à intégrer des données de sources multiples et hétérogènes est extrêmement difficile et nécessite du temps. Or, XML autorise la combinaison aisée de différentes sources. Des agents logiciels interviennent pour intégrer les données de bases de données à l'arrière-plan et d'autres applications, livrables ensuite à d'autres clients et/ou serveurs pour un traitement supplémentaire ou d'autres présentations.
- Capacité à décrire les données d'une grande variété d'applications : XML permet de décrire des données contenues dans une grande variété d'applications. De plus, étant autodescriptif, la réception et le traitement des données ne nécessitent plus de description intégrée et explicite de ces données.
- Des moteurs de recherche plus évolués : A l'heure actuelle, les moteurs de recherche fonctionnent sur des informations contenues dans des métabalises HTML ou sur la proximité de mots-clés par rapport à d'autres. Avec XML, les moteurs de recherche parcourent simplement les balises qui contiennent les descriptions.

La plupart des systèmes d'intégration de données actuels utilisent XML comme modèle commun [Bellahsène *et al.*, 2001] [Gardarin, 2002] du fait qu'il permet une représentation des données provenant de sources hétérogènes dont les modèles de données sont souvent différents. Aussi, de nombreuses sources de données sont-elles disponibles et de nombreuses applications permettent-elles d'exporter leurs données en XML.

XML permet de représenter des données semi-structurées en intégrant la plupart des concepts des modèles connus [Gardarin, 2002]. Il permet de composer des hiérarchies de données liées entre elles par des hyperliens [W3C, 2000f], [W3C, 2002]. Les relations peuvent être vues comme des instances d'éléments avec des attributs. Les hyperliens correspondent aux associations des modèles entité-association. Avec les schémas qui sont apparus récemment [W3C, 2001a], [W3C, 2001b], [W3C, 2001c], le typage des données élémentaires est possible. A cet égard, de nombreux types sont proposés permettant de décrire les données de manière précise. La structure est contenue dans les données sous forme de balise et de nom d'attribut.

Enfin, la standardisation de XML par le W3C et son succès dans l'industrie informatique en font un bon candidat comme langage commun pour un système d'intégration.

2.7 Conclusion

Une analyse des systèmes d'intégration de données présentés précédemment fait apparaître tout d'abord que même si tous ces systèmes suivent une architecture de médiation, les objectifs et donc les architectures de ces systèmes sont différents. Selon les objectifs, nous pouvons distinguer dans le contexte de l'interopérabilité de sources de données hétérogènes qu'il existe deux principales approches d'intégration de données à savoir l'approche virtuelle (ou par médiateur) et l'approche matérialisée (ou par entrepôt).

(i) L'approche virtuelle [Garcia-Molina *et al.*, 1997], ou par médiateur (figure 2.12), désigne une vision globale par l'intermédiaire d'un unique schéma de représentation d'un ensemble de sources de données hétérogènes. Ce schéma global peut être défini automatiquement à l'aide d'outils, ou extracteurs de schémas. Dans ce contexte, les données sont stockées uniquement au niveau des sources. Les traitements sont donc synchronisés sur ces sources de données. Un médiateur connaît le schéma global et possède des vues abstraites sur les sources qui lui permettront de décomposer la requête initiale en sous-requêtes. Le médiateur soumet ces sous-requêtes à des adaptateurs qui ont pour fonction de traduire ces dernières dans des langages compréhensibles par les différentes sources de données. Une fois le traitement de ces requêtes réalisé par ces sources, les réponses suivent le cheminement inverse jusqu'à l'utilisateur.

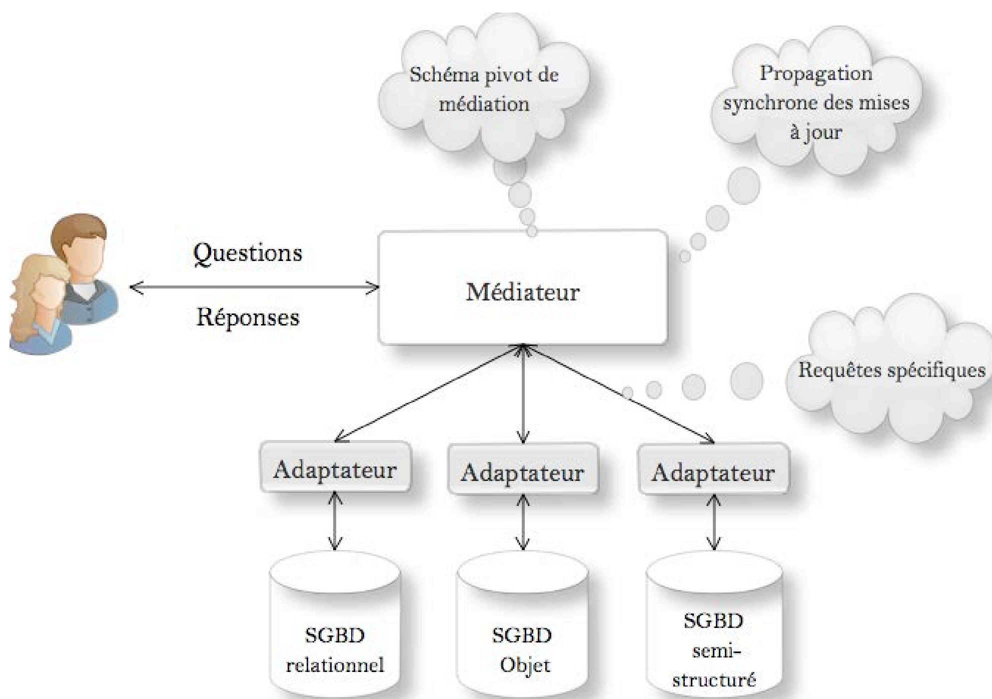


Figure 2.12. Approche virtuelle

(ii) Dans l'approche d'intégration matérialisée [Abitboul *et al.*, 2002], les données issues de sources hétérogènes sont copiées dans un entrepôt de données (ou référentiel). Les actions sur le référentiel sont asynchrones par rapport aux sources. La propagation des modifications apportées au référentiel vers les différentes sources de données doit passer par des procédures de mises à jour. Contrairement à l'approche virtuelle, les requêtes utilisateurs sont directement exécutées dans le référentiel, sans avoir à accéder aux différentes sources de données. Ici, les données du référentiel sont déconnectées de celles contenues dans les sources hétérogènes. Les mises à jour des données du référentiel vers les sources et réciproquement sont déléguées à un intégrateur qui a pour fonction de réaliser la correspondance entre le schéma du référentiel et les sous-schémas des sources.

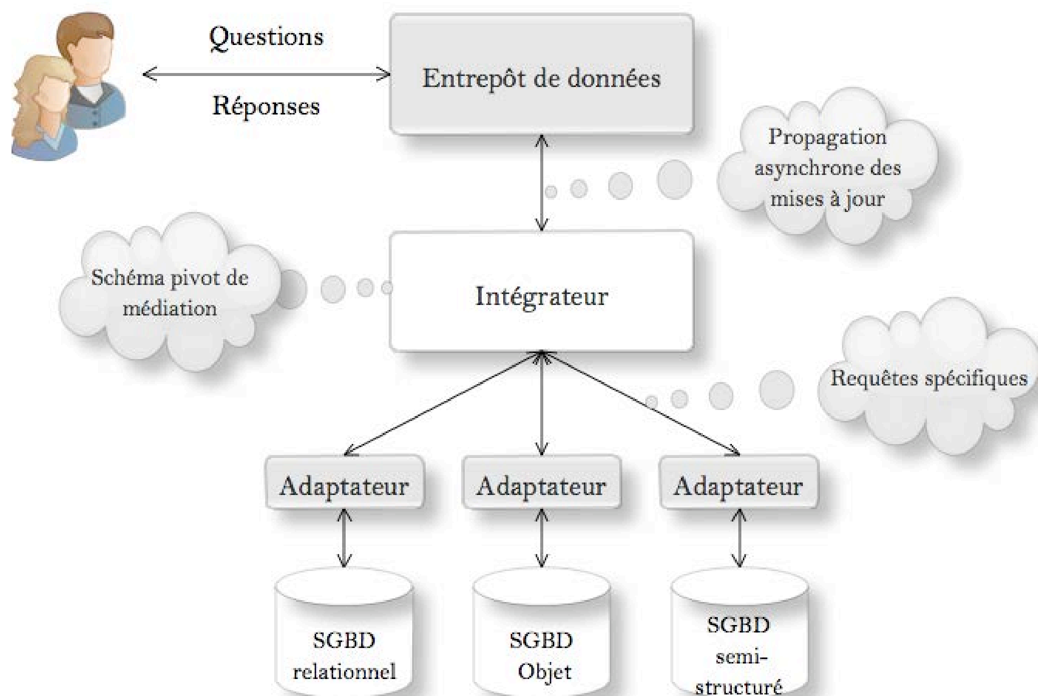


Figure 2.13. Approche matérialisée

A partir des systèmes d'intégration présentés précédemment, nous pouvons distinguer les systèmes concentrés sur le traitement des requêtes de ceux focalisés sur l'architecture des systèmes d'intégration.

Dans le premier cas (traitement des requêtes), les systèmes DISCO [Tomasic *et al.*, 1998] et GARLIC [Haas *et al.*, 1997] prennent en compte les méta-informations sur les sources. Cependant, ils abordent la problématique d'optimisation différemment par rapport à la représentation des méta-informations sur les sources, des stratégies de recherche et des modèles de coûts utilisés. Dans DISCO, les méta-informations sur les sources sont représentées dans un langage de description de capacités. L'adaptateur transmet ensuite ces

informations au médiateur. Un optimiseur de requêtes globales (au niveau médiateur) construit le modèle de coût en tenant compte des capacités des sources (selon une stratégie top-down). L'utilisation d'un langage de description des capacités des sources a été adoptée dans une majorité d'autres systèmes tels que TSIMMIS [Garcia-Molina *et al.*, 1997] et YAT [Cluet *et al.*, 1998] contrairement à GARLIC qui n'utilise pas un langage de description de métadonnées.

Dans le second cas (point de vue architectural), les systèmes d'intégration se sont principalement intéressés à la composition des médiateurs. Cette composition est montrée dans le système TSIMMIS. Ce projet fait partie des pionniers dans la médiation des données structurées et semi-structurées. Il utilise une hiérarchie de médiateurs pour intégrer les sources de données hétérogènes. Il a introduit le formalisme OEM (Object Exchange Model) comme modèle commun où chaque source est transformée par un adaptateur pour fournir des données OEM aux médiateurs.

L'avantage de l'approche virtuelle par rapport à l'approche matérielle tient au fait qu'il n'y a plus de problème de taille de la base de données et de problème de cohérence. Les données demeurent en effet dans les sources locales ; le médiateur se contente alors de maintenir le schéma global et les traducteurs permettent de réaliser les requêtes. Cependant, avec cette solution, les temps de recherche augmentent car le nombre de traitements à effectuer est plus important. En effet, le médiateur doit traduire les requêtes exprimées par les clients à l'aide du schéma global et du langage d'interrogation de la médiation pour qu'elles soient compréhensibles par les sources locales, et ceci, pour chacune des sources locales auxquelles il va s'adresser. Il doit ensuite communiquer les requêtes à chacune des sources locales, traduire et agréger les réponses qu'elles lui retournent pour présenter aux clients une réponse homogène exprimée à l'aide du schéma global. Il est à noter que ces deux approches peuvent être combinées afin de bénéficier de leurs avantages respectifs [eXMLMedia, 1999].

Les approches virtuelles et matérialisées ont longuement été étudiées par ces différents travaux qui présentent des limitations en terme de standardisation, d'outils et de complexité de mise en place dans des contextes industriels. De plus, certains de ces systèmes se focalisent uniquement sur certaines problématiques telles que le traitement des requêtes ou l'intégration et la diffusion de données et ne traitent pas certains aspects importants de gestion de la donnée tels que :

- La mise en place de rôles et de droits d'accès pour accéder à la donnée.
- L'uniformisation de la représentation des données au sein d'un système d'information par l'intermédiaire d'un outil de gestion unique.
- La mise à disposition de moyens permettant d'auditer et d'historiser les données.
- La gestion de manière efficace et performante du cycle de vie des données et de l'accès concurrentiel entre différents utilisateurs.

Pour outrepasser ces problématiques individuelles, le Master Data Management[6] (MDM) a été défini comme une approche visant à pallier ces limitations tout en se focalisant sur l'ensemble des problématiques de fédération de sources de données.

[6] Gestion des données de référence.

Chapitre 3

La gestion des données de référence ou Master Data Management

Résumé. Dans le chapitre précédent nous avons présenté les principales approches et les principaux systèmes d'intégration de données existants et nous avons conclu sur leurs limitations en terme de mise en application dans des contextes industriels. Le Master Data Management (MDM) est une approche émergente d'intégration de données qui a été définie pour combler ces limitations. Le MDM étant une discipline récente, très peu de travaux existent à ce jour. Par rapport aux approches existantes, le MDM se focalise de plus sur l'unification des modèles et outils, et une gestion optimisé du cycle de vie des données au sein d'un système d'information. Ce chapitre a pour objectif de présenter les principes du Master Data Management et les solutions existantes mettant en œuvre cette approche.

3.1 Introduction

Le Master Data Management est une approche émergente d'intégration de données basée sur l'approche matérialisée que nous avons présentée dans le chapitre précédent. Le MDM étant une discipline récente, très peu de travaux existent à ce jour ([iWays 2009], [Oracle 2007], [IBM 2004], [Orchestranetworks MDM 2000]). Par rapport à l'approche matérialisée, le MDM se focalise de plus sur l'unification des modèles et outils au sein d'un système d'information. Actuellement, la majorité des systèmes d'information est caractérisée par une hétérogénéité en terme de données et de solutions de paramétrage. En effet, cette hétérogénéité se présente sous différents aspects : diversité des systèmes de stockage (bases de données, fichiers, annuaires, etc.), diversité des formats de données (tables, fichiers propriétaires, documents XML, etc.), diversité des solutions proposées pour gérer les différents types de données, diversité des acteurs exploitant les données de références (utilisateurs fonctionnels ou non), diversité des domaines d'application (CRM[7], ERP[8], etc.), diversité des activités (dites verticales pour des activités telles que la production ou

[7] Customer Relationship Management.

[8] Enterprise Resource Planning.

l'approvisionnement, ou dites horizontales pour des activités telles que le marketing ou les ressources humaines), etc. D'autre part, utiliser un ensemble d'applications différentes afin de pouvoir gérer cette diversité dans les types de données entraîne inévitablement de la redondance tant au niveau des données que des outils. En l'absence de MDM, la propagation des mises à jour de données se réalise sans référentiel central ni modèle commun d'information, dans un style d'architecture souvent qualifié de «point à point». Ce style est assez courant en architecture ETL[9]/EAI[10] de base, sans MDM.

Deux conditions sont nécessaires à une architecture centralisée MDM :

(i) Il faut disposer d'un outil de MDM générique capable d'accueillir le modèle commun d'information pour toutes les natures de données. Sans ce niveau de généricité, il faudrait accepter des MDM par silos organisés autour des domaines d'information (Client, Produit, Organisation, paramètres fonctionnels, paramètres techniques, etc.) et les conséquences néfastes en terme de duplication des référentiels (ce que l'on cherche à éviter) et de duplication des fonctions de gouvernance (gestion des versions, interface homme-machine d'administration, etc.) ;

(ii) Il faut une méthode pour la modélisation et la négociation du modèle commun d'information faisant abstraction des formats « propriétaires » des différents systèmes.

Le premier point est assuré par les solutions MDM que nous présenterons au cours de ce chapitre. La solution de cette thèse étant l'introduction d'une approche d'Ingénierie Dirigée par les Modèles (IDM) pour définir des modèles de données pivot, nous montrerons que cette approche est une solution adéquate au second point.

Ce chapitre présente la problématique associée à la gestion de la donnée de référence au sein d'un Système d'Information. La section 3.2 définit la notion de donnée de référence. Dans la section 3.3 nous montrons la nécessité d'adopter une approche MDM au sein d'un Système d'Information. Dans la section 3.4 nous présentons comment migrer vers une architecture MDM. Nous étudierons dans la section 3.5 la solution MDM EBX.Platform sur laquelle s'est basé l'essentiel des travaux menés durant cette thèse. Pour finir nous présenterons les procédés de définitions d'un modèle de données pivot.

3.2 Définition de la donnée de référence

Une donnée est dans le périmètre des *données de référence* dans les situations suivantes, que nous détaillons dans cette section :

- Donnée dupliquée au sein de plusieurs systèmes.
- Donnée valorisée par un dispositif transverse aux systèmes applicatifs avant usage par les systèmes transactionnels.

---

[9] Extraction Transformation Loading.

[10] Enterprise Application Integration.

Les données de référence peuvent être liées entre elles par des associations et agrégations (un client est relié à une organisation, un produit est composé d'autres produits, etc.). Par conséquent, la conception d'un *modèle commun d'information* s'appuie sur des procédés complets de modélisation de l'information. Il ne s'agit pas simplement de concevoir une nomenclature de données sous la forme de liste de codes et libellés mais de définir un ensemble cohérent d'informations exploitable au travers d'un système d'information.

3.2.1 Donnée dupliquée au sein de plusieurs systèmes

La fiabilité de la valeur d'une donnée dupliquée dans de multiples systèmes impose la définition d'un référentiel maître pour ces données et l'exécution de règles de contrôle des mises à jour : droits d'accès, traçabilité d'usage, etc.

La solution préconisée est une centralisation de la donnée dans un référentiel maître et indépendant des systèmes applicatifs afin d'en garantir sa pérennité. Pour y parvenir, on s'appuie sur une solution de *Master Data Management.* Dans une approche MDM, il est question de centraliser à la fois les structures de données, sous la forme d'un modèle commun d'information, mais aussi les contenus, c'est-à-dire les valeurs des données pour chaque version (fonctionnelle et technique) et variante (contextes de valorisation multiples pour une même version).

En l'absence de MDM, la propagation des mises à jour de données se réalise sans référentiel central, ni modèle commun d'information, dans un style d'architecture souvent qualifié de « point à point ». Ce style est assez courant en architecture ETL / EAI de base, sans MDM. La figure 3.1 illustre, de manière simplifiée, cette architecture.

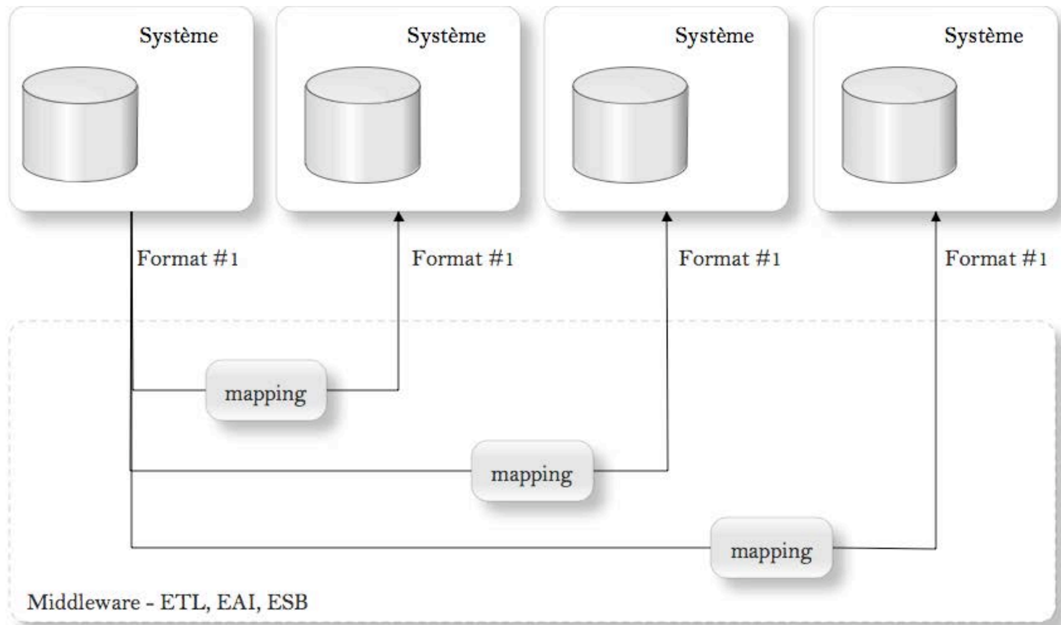


Figure 3.1. La propagation de données en mode point à point.

Dans ce cas l'absence, de référentiel central et de modèle commun d'information rend difficile, parfois impossible, la mise en place de règles mutualisées pour le contrôle de la qualité des données, de même qu'il n'est pas évident d'assurer une traçabilité des échanges. Au mieux, il peut exister une piste d'audit mais pas de référentiel structuré de stockage de l'information échangée, du fait de l'absence d'un modèle commun d'information.

A l'inverse, avec l'utilisation d'un MDM (figure 3.2), on dispose d'un référentiel central qui incarne le modèle commun d'information. Les propagations des mises à jour des données se mènent à partir du MDM. Le mode « point à point » est alors abandonné, de manière progressive. Ce style d'architecture allège les traitements de transformation de données et renforce les possibilités d'administration des données, en particulier en terme de traçabilité et de contrôle qualité sur les échanges. Toutefois, deux conditions sont nécessaires à ce style d'architecture :

- D'une part, il faut disposer d'un outil de MDM générique capable d'accueillir le modèle commun d'information pour toutes les natures de données. Sans ce niveau de généricité, il faudrait accepter des MDM par silos organisés autour des domaines d'information (Client, Produit, Organisation, paramètres fonctionnels, paramètres techniques, etc.) et les conséquences néfastes en terme de duplication des référentiels (ce que l'on cherche à éviter) et de duplication des fonctions de gouvernance (gestion des versions, interface homme-machine d'administration, etc.).
- D'autre part, il faut une méthode pour la modélisation et la négociation du modèle commun d'information qui masque les formats « propriétaires » des différents systèmes.

La figure 3.2 illustre l'architecture de gestion de la propagation de données, en prenant appui sur le MDM.

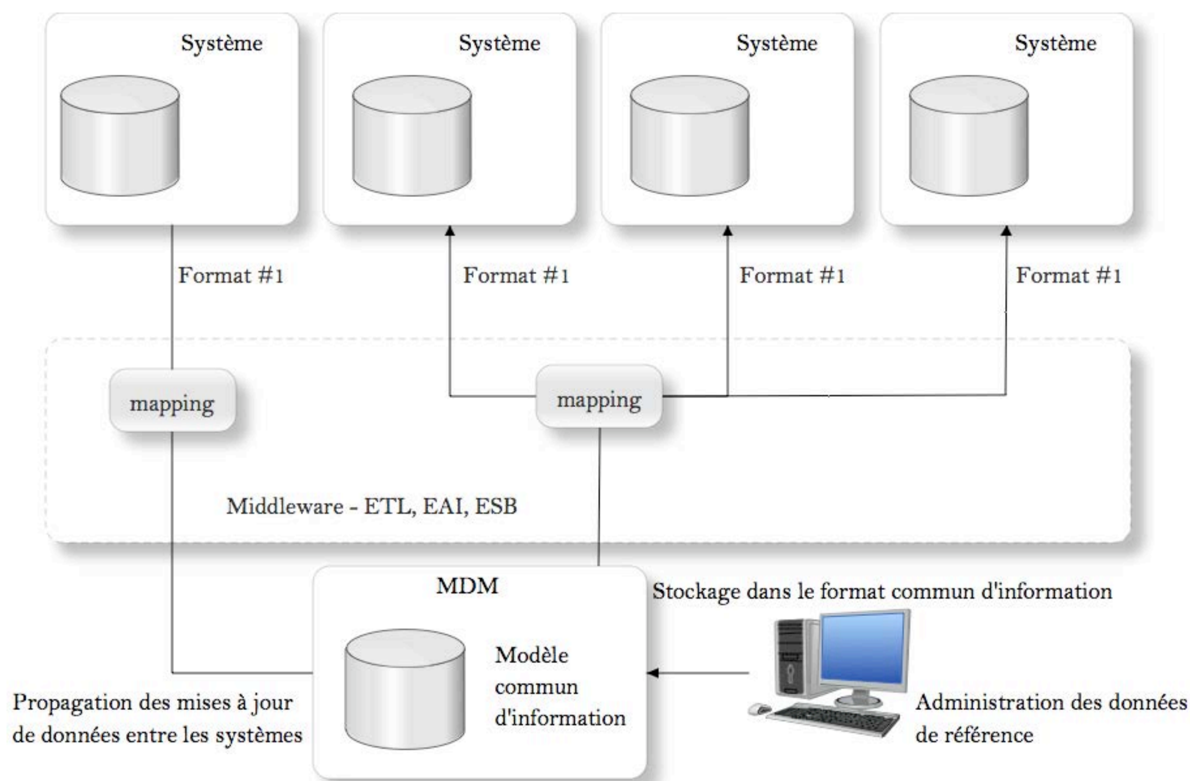


Figure 3.2. La propagation de données via le Master Data Management.

Dans la pratique, les échanges entre les systèmes et le MDM mettent en œuvre de multiples modèles d'interaction : synchrone, asynchrone, temps réel, *batch*, etc. L'infrastructure *middleware* (ETL, EAI, ESB) est donc complémentaire et indispensable au fonctionnement du MDM. De même, un outil évolué de transformation de données (*data mapping*) est nécessaire car il est rare que le modèle commun d'information se propage à l'intérieur de l'ensemble des systèmes. Ce n'est d'ailleurs pas toujours un objectif car une isolation entre les formats de données du MDM et ceux connus dans les systèmes peut être souhaitée. La profondeur d'utilisation du modèle commun d'information dépend du contexte de chaque entreprise.

Toute donnée dupliquée entre plusieurs systèmes est dans le domaine des données de référence, y compris si cette donnée n'est pas spontanément candidate au MDM. Nous pouvons cependant constater qu'avec la progression de l'urbanisation des systèmes d'information, la duplication de données diminue, ce qui permet au MDM de se concentrer sur les informations véritablement pivots (partagées) entre les applicatifs.

3.2.2 Valorisation des données

La donnée est valorisée par un dispositif transverse aux systèmes applicatifs, avant usage par les systèmes transactionnels. Cette valorisation se mène en tenant compte des contextes d'utilisation qui correspondent, par exemple, à une version fonctionnelle ou technique, une segmentation de marché, un canal d'utilisation, une langue, etc. Ce procédé de valorisation par contextes est courant pour la gestion des tables de type codes-libellés (énumérations). Cependant, avec l'ouverture des systèmes d'information, la prise en compte de ces multiples contextes s'avère plus complexe et nécessite un processus de gestion adapté, mutualisé et indépendant des systèmes applicatifs afin d'en assurer sa pérennité. Les contextes s'appliquent non seulement aux structures simples de données (par exemple code, libellé) mais aussi aux données de référence qui définissent des structures de produits, des organisations, etc.

Dans la figure 3.3 nous représentons une arborescence de contextes de valorisation qui décrit une structure organisationnelle par secteurs géographiques. Une même donnée de référence aura une valeur différente selon le contexte : siège social, régions, etc. Les valorisations bénéficient des principes d'héritage afin d'éviter la duplication inutile de données. Ces valorisations multiples d'une même donnée sont aussi nommées variantes.

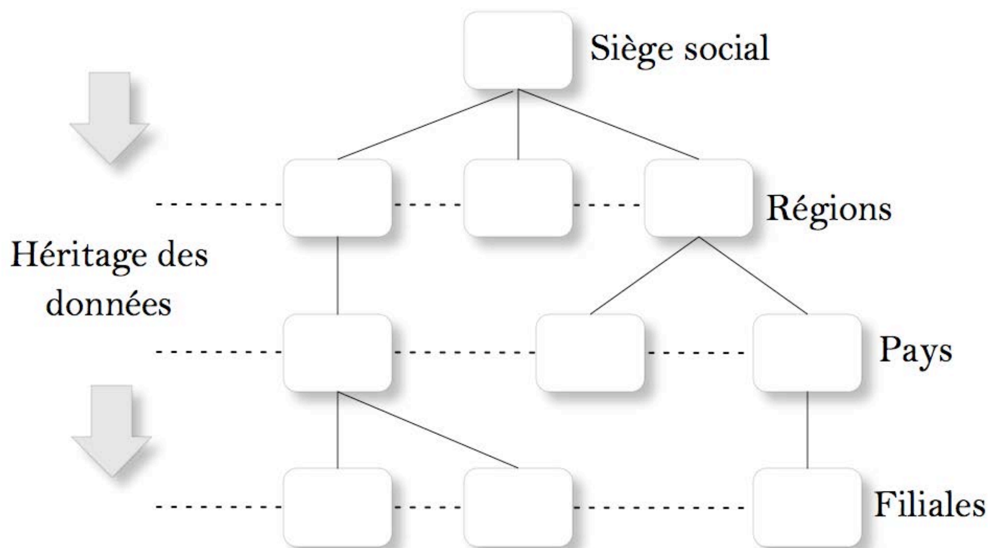


Figure 3.3. Valorisation des données dans une structure organisationnelle.

L'outil MDM doit permettre ce mode de valorisation avec une gestion des droits autorisant, par exemple, le responsable d'une filiale de modifier la valeur d'une donnée de référence pour son propre contexte, sans modifier ou consulter les valeurs de cette même donnée dans les autres contextes de valorisation.

3.2.3 Corollaire sur l'architecture MDM

Notre définition de la donnée de référence (duplication de données, valorisation avant usage par les systèmes transactionnels) ainsi que les préconisations que nous venons de formuler dictent le style d'architecture que nous retenons à savoir celui d'une centralisation des données de référence à l'aide d'un référentiel reposant sur le modèle commun d'information et stockant les valeurs des données de référence. Ce référentiel est soit utilisé directement par les systèmes applicatifs au travers d'une architecture orientée services, soit se propage dans les bases de données de ces systèmes à l'aide d'une chaîne d'intégration des données. Les autres styles d'architecture possibles, en particulier la propagation des mises à jour entre les systèmes selon un mode « point à point » sans centralisation des valeurs ou encore la mise en place d'un référentiel MDM « virtuel » limitant son action à une gestion d'index (vue réduite des données de référence limitée aux clés d'accès) ne rentrent pas dans le périmètre des procédés de modélisation décrits ici. Il est préférable d'opter pour la mise en place d'un référentiel complet des données de référence qui permettra de contrôler entièrement le cycle de vie des données, en particulier les valorisations par contexte (version comprise) et la traçabilité de leurs mises à jour. Ce référentiel central repose sur un modèle commun d'information qui rationalise les traitements de transformation de données entre les systèmes. Ceci évite ainsi de multiplier les transformations de données en mode « point à point » entre les systèmes. Ce style d'architecture met en œuvre une solution de *Master Data Management* qui fournit le référentiel de stockage, l'interface homme-machine d'administration des données et toutes les fonctions de gouvernance associées : gestion des versions, gestion des droits, interrogation, validation, processus d'approbation des mises à jour, interface avec le *middleware* d'intégration de données (ETL, EAI, ESB), etc.

Nous pouvons définir une donnée de référence comme étant une donnée nécessaire et devant être non redondante dans un Système d'Information. A partir de ce constat nous pouvons assimiler le Master Data Management à une couche supplémentaire de l'approche matérialisée vue dans le chapitre précédent.

3.3 Principes du MDM

3.3.1 Centralisation des données et la synchronisation des données

Le MDM centralise les données dans un unique référentiel et les propage entre les applications déployées au sein d'un système d'information. Les données sont identifiées de la même manière dans l'ensemble du Système d'Information. Il est ainsi possible d'appliquer une stratégie de mutualisation de l'information. Le MDM permet aux collaborateurs d'échanger les données en temps réel. Les modifications des données clés telles que les

coordonnées d'un client, d'un partenaire pourront alors être réalisées directement dans un même référentiel. Le référentiel collecte et distribue des données ayant des formats différents, ce qui permet d'alimenter toutes les applications (PGI, page web, etc.) au fil de l'eau. Avec des données maîtrisées et à jour, les utilisateurs peuvent accéder à l'information réelle.

3.3.2 Confidentialité des données

La confidentialité des données représente la gestion de l'accès aux données. Etant donné que chaque utilisateur peut avoir des droits différents, il faudra obligatoirement filtrer les données en fonction de l'utilisateur. Par exemple, lors d'une connexion à une application en tant qu'administrateur, un utilisateur est capable d'accéder aux informations d'autres usagers (login, mot de passe, données confidentielles, etc.). Inversement lorsqu'un utilisateur possède des droits restreints, celui-ci n'est pas autorisé à accéder à toutes les informations.

3.3.3 Simplification de l'information

En adoptant une approche de type *Enterprise Information Integration* [Halevy *et al.*, 2005], l'utilisateur obtient une vue métier unifiée. Les données sont stockées dans leur fichier ou base d'origine, puis en fonction de ses droits, l'utilisateur ne pourra accéder qu'aux données auxquelles il a le droit d'accéder. Cette approche permet d'éviter ainsi la surcharge d'information inutile à l'écran.

3.3.4 Qualité du contenu

Dans un système d'information, la qualité de l'information est la caractéristique la plus importante. Une donnée est dite de qualité si et seulement si elle est en phase avec la réalité et est conforme aux exigences spécifiées. Par exemple, dans le cas d'une commande client, si un client commande n articles, cette donnée pourra être qualifiée de qualité si le système a effectivement enregistré la bonne valeur et si celle-ci est conforme aux règles métiers imposées (ex. une date de fin doit être ultérieure à une date de début).

Les solutions de gestion de la qualité des données fonctionnent d'autant mieux qu'elles s'exécutent à partir d'un référentiel de données unifié, construit à partir d'un modèle commun d'information, c'est-à-dire du MDM. Le travail de modélisation de ce modèle s'impose comme une étape indispensable à la fois pour la qualité des données et pour le MDM.

3.3.5 Accessibilité

L'accessibilité signifie la disponibilité de l'information et la facilité d'accès. L'utilisateur doit être capable d'accéder à l'information sans avoir à passer par plusieurs applications. Les utilisateurs ont besoin d'accéder à l'information mise à jour en temps réel.

3.3.6 Flexibilité

Dans un système d'information, les données doivent pouvoir être évolutives. Une donnée est évolutive si elle peut être modifiée sans que l'utilisateur soit obligé de la redéfinir. La méthode Master Data Management permet de référencer les données de manière unique à l'aide d'un système de clé unique. Cette clé devient alors l'identifiant de la donnée, lui permettant d'évoluer en conservant son «identité».

3.3.7 Sécurité

La sécurité des données est un point fondamental. Bien que la plupart des personnes associent la sécurité à la confidentialité, il faudra étudier la fiabilité des données. La fiabilité des données est un mécanisme de sauvegarde permettant à l'utilisateur de faire une reprise des données en cas de faille du système.

3.3.8 Workflow intégré

Le Workflow automatise la gestion des flux d'information en fonction des spécifications d'un processus métier. En fonction de la succession des tâches définies, les tâches de traitement passent d'un acteur vers un autre selon un circuit conditionnel bien défini à l'avance. Le système facilite la tâche de l'utilisateur en lui présentant automatiquement les informations nécessaires à l'accomplissement de sa tâche. Ainsi l'utilisateur est apte à réaliser sa tâche avec aisance. Des systèmes de workflow sont omniprésents. Par exemple, lorsqu'un internaute souhaite publier une photographie sur un site web à caractère social, un administrateur doit procéder à une phase de validation qui suit les tâches suivantes :

(i) L'administrateur du site va être notifié qu'un utilisateur veut publier un contenu. Le système lui demandera d'accepter ou refuser le contenu.
(ii) Il devra contrôler que la photo répond bien aux exigences définies (taille, mise en place, contenu).
(iii) Auquel cas, la photo sera validée et publiée. C'est seulement à partir de ce moment que les autres internautes pourront voir cette photo.

Le Master Data Management a non seulement pour vocation de fédérer les données et leurs sources, mais aussi de mutualiser les outils nécessaires à l'exploitation et de proposer une gestion efficace et performante des données au sein d'un système d'information. Les évolutions du Master Data Management par rapport aux approches existantes rendent pertinente la mise en place d'une telle architecture dans un système d'information.

3.4 Mise en place d'une architecture de type *MDM*

Une mise en place progressive du référentiel des données de référence peut se mener avec un faible niveau d'intrusion vis-à-vis des systèmes existants. Les impacts se situent essentiellement au niveau de la rationalisation des processus d'intégration de données en place. Les intégrations en mode « point à point » sont remplacées par une intégration *via* le modèle commun d'information. Les bases de données de référence des systèmes existants ne sont pas modifiées ; elles sont alimentées par le référentiel central, de manière transparente. Les outils de valorisation des données de référence des systèmes existants seront généralement supprimés au profit de l'outil d'administration amené avec le référentiel central. C'est ici que l'on renforce l'intérêt de la solution MDM à l'échelle de l'entreprise, avec des fonctions de gouvernance identiques pour tous les modèles de données et toutes les natures d'information (métier, technique, fonctionnel, etc.) : génération automatique de l'interface homme-machine permettant la consultation par critères, mises à jour selon les règles de validation et d'habilitation, processus d'approbation des mises à jour, traçabilité des modifications, gestion des versions des données avec des fonctions de comparaison et fusion, déploiement des données dans les systèmes, définition des contrats d'abonnement aux modifications de données, exposition de services d'accès et de mise à jour des données, etc. En cas de refonte d'un applicatif, le nouveau système pourra s'appuyer directement sur le MDM en place afin d'accéder aux données de référence. L'usage d'un modèle commun d'information, dès le début du plan d'évolution vers le MDM, favorise une meilleure maîtrise des projets de refonte.

Nous connaissons les styles d'architecture logique des données relationnelles (avec les formes normales), l'orientation objet (avec les attributs multi-valués et l'héritage), le hiérarchique (types complexes de données en particulier aujourd'hui *via* XML), le multidimensionnel pour les systèmes décisionnels. Dans le cadre du MDM, il est question de mettre en place un référentiel de données pouvant être accédé en consultation mais aussi en mise à jour. De fait, le style d'architecture logique multidimensionnel est écarté pour l'implémentation du MDM car la dénormalisation rend difficile la mise à jour cohérente des données. A l'inverse, nous devons concilier les apports des autres styles d'architecture logique, d'un seul tenant, grâce à l'utilisation d'outils MDM de haute qualité capables d'adresser à la fois la gestion relationnelle (clef primaire et secondaire, contraintes d'intégrité référentielle), l'objet pour les attributs multi-valués y compris sur les listes de clefs étrangères et le hiérarchique dans la famille XML afin de construire des types de données

réutilisables et encapsulant des facettes comportementales de validation des données. On cherchera aussi à attacher des traitements (services) aux données afin de factoriser certaines facettes de comportement pour mieux les réutiliser ou encore invoquer des services à l'extérieur des modèles de données ; cette externalisation sera systématiquement recommandée dès que les contrôles encapsulent des règles métier.

3.5 Solution MDM EBX.Platform

De nos jours, les entreprises sont confrontées à de forte pressions pour réduire les coûts et accroître les marges. Les logiciels Master Data Management vont permettre au service informatique d'avoir le contrôle sur le système d'information. De plus, les utilisateurs finaux seront aptes à accéder à l'information via un unique point d'accès.

Dans cette section nous allons présenter la solution MDM EBX.Platform que nous développons au sein d'Orchestra Networks.

3.5.1 Architecture

Basé sur le standard XML Schema [W3C, 2001a], EBX.Platform simplifie la définition de modèles qui ont pour but d'unifier les données de référence d'une entreprise. En utilisant la technologie XML Schema, ces modèles peuvent être de tous types (simples, complexes) et de toutes natures (métiers, techniques, graphiques). Un des principaux avantages de XML Schema est de permettre la définition de modèles de données structurées et typées et ayant de puissantes propriétés de validation. La figure 3.4 présente l'architecture de notre solution MDM EBX.Plateform :

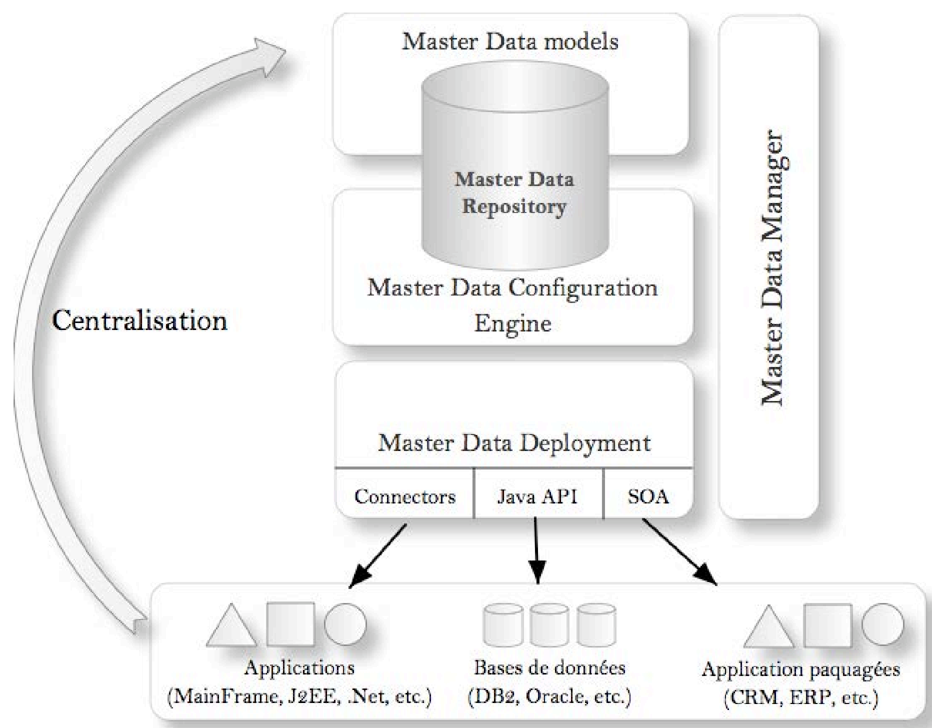


Figure 3.4. Architecture EBX.Platform.

EBX.Platform est constituée des 5 modules suivants :

- Master Data Models : module qui permet d'unifier, d'enrichir à l'aide de XML Schema et de manipuler les structures de données existantes (tables, fichiers, etc.).
- Master Data Repository : définit le référentiel interne dans lequel les contenus existants sont importés dans EBX.Platform.
- Master Data Configuration Engine : représente le moteur de configuration gérant les règles d'héritage entre les instances de données (ex : environnements, filiales, canaux, etc.) et leur cycle de vie.
- Master Data Manager : outil web convivial et sécurisé permettant d'accéder à la gestion des données de référence et des paramètres.
- Master Data : à partir de ce module, il est possible de propager à travers le Système d'Information les données de référence à l'aide de connecteurs (SGBD, fichiers, etc.), d'une API Java ou à l'aide de services web.

3.5.2 Concepts

EBX.Platform repose sur deux principes à savoir :

(i) Des modèles d’adaptation qui sont des documents XML Schema définissant la structure des données de référence,

(ii) Des adaptations qui sont des instances XML des modèles d’adaptation représentant le contenu des données de référence.

L’utilisation de XML Schema permet de préciser que chaque nœud du modèle de données correspond à un type de données existant et conforme au standard du W3C. D’autre part, le formalisme de XML Schema permet de spécifier des contraintes (énumération, longueur, bornes inférieures et supérieures, etc.), des informations relatives à l’adaptation et à l’instanciation (connecteurs d’accès, classe d’instanciation Java, restriction d’accès, etc.) et des informations de présentation (libellé, description, formatage, etc.) pour chaque nœud du schéma. Une adaptation est une instance du modèle d’adaptation. A tout nœud du modèle d’adaptation déclaré valorisable correspond un nœud dans l’adaptation.

3.5.2.1 Héritage des données

Il est souvent constaté que plus des trois quarts des données de référence sont identiques entre deux instances (par exemple un catalogue de produits entre un siège et une filiale). Il est important d'éviter la duplication de ces données afin de prévenir de longues procédures de saisies qui sont souvent fastidieuses et source d'erreurs. Pour ce faire, EBX.Platform s'appuie sur une technologie d’héritage. Ce mécanisme permet la création d'un "arbre d’instances" à partir d'un premier modèle de données de référence et la gestion de règles d’héritage entre les instances. Dans ce modèle de gestion chaque instance hérite de son parent. Si une hiérarchie de modèles d’adaptation est définie alors nous considérons qu’un arbre d’adaptation est manipulé comme l’illustre la figure 3.5.

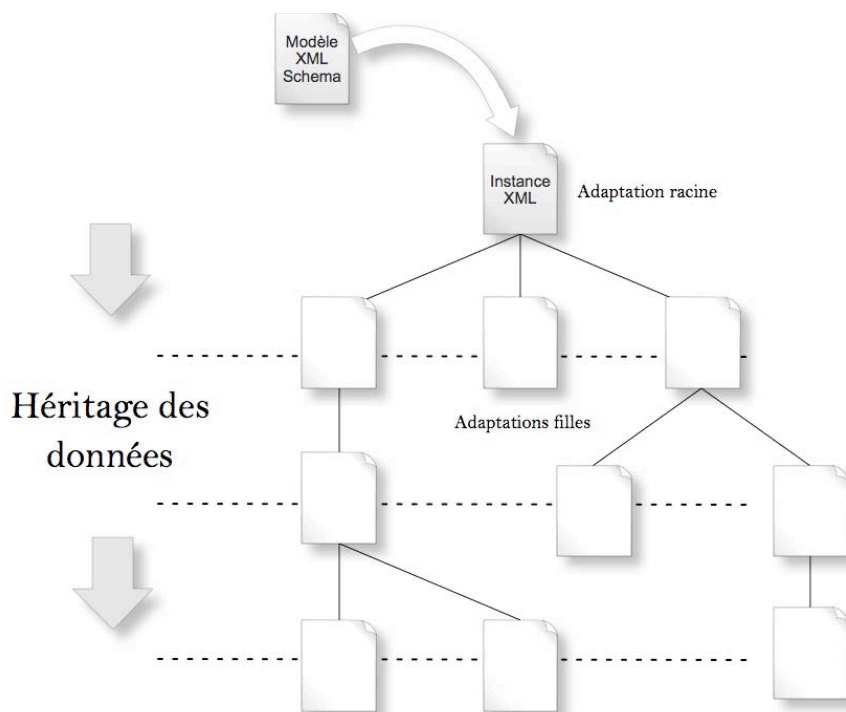


Figure 3.5. Arbre d'adaptation.

EBX.Platform offre les fonctions de valorisation des données selon des contextes d'usage. Ces contextes ne sont donc pas modélisés de manière figée. Ils sont déclarés dynamiquement par les utilisateurs de EBX.Platform ayant les droits adaptés. Exemple de contextes d'application : traduction de valeurs, produits spécifiques d'une filiale, etc. Ces contextes héritent les uns des autres selon un arbre de contextes. L'objectif est de factoriser les valorisations au niveau supérieur de l'arbre puis permettre les personnalisations par contexte, aussi bien en terme de modification de valeur, d'ajout de données (sur la base du principe des attributs dynamiques présenté plus haut dans la modélisation sémantique), d'occultation de valeurs (soit une colonne d'une table, soit un tuple complet, etc.). Une fonction de EBX.Platform permet, par exemple au moment du chargement des données, de prédéterminer automatiquement les valorisations pouvant faire l'objet d'une factorisation. C'est-à-dire identifier une valeur de donnée dans un contexte fils qui est identique à la valeur du contexte père. Dans ce cas, l'héritage permet de nettoyer des doublons. Cependant l'usage de l'héritage mobilise des ressources de calcul qui doivent être bien analysées dans l'étude des performances. Pour les cas d'héritage important (plusieurs centaines de nœuds), il est possible d'optimiser l'utilisation de l'application en dénormalisant le modèle de données au moment du passage du niveau logique vers le logiciel. Par exemple, la concentration, dans un même modèle d'adaptation, de l'ensemble des données qui ne sont pas concernées par l'héritage et dans un autre celles qui bénéficieront de l'héritage est possible. Les instances du second modèle feront des références (telles des contraintes de clé étrangère dans la sémantique des systèmes de gestion de bases de données) vers le premier.

3.5.2.2 Cycles de vie et versions des données

Les données de références sont souvent fondées sur des cycles de vie complexes. Là où les approches virtuelles et matérialisées se focalisent uniquement sur la fédération de sources de données, le Master Data Management permet de bénéficier en plus d'une gestion efficace du cycle de vie des données et garantit une traçabilité. Dans cette optique, EBX.Platform définit un mécanisme de branches de données dans un référentiel. En utilisant ce mécanisme (figure 3.6), il est possible de faire des modifications sur différentes branches et de les comparer/fusionner. EBX.Platform fournit également une fonctionnalité de version qui permet de prendre des «images immuables» de branches en vue de garder la trace des modifications effectuées. A tout moment, sous condition de disposer des droits, il est possible de créer une branche du référentiel, de noter des versions de données sur cette branche, de comparer des versions et des branches, de fusionner des branches, etc. Ces fonctions portent sur le contenu des données, c'est-à-dire leurs valorisations, et non sur les structures de données, c'est-à-dire les schémas XML.

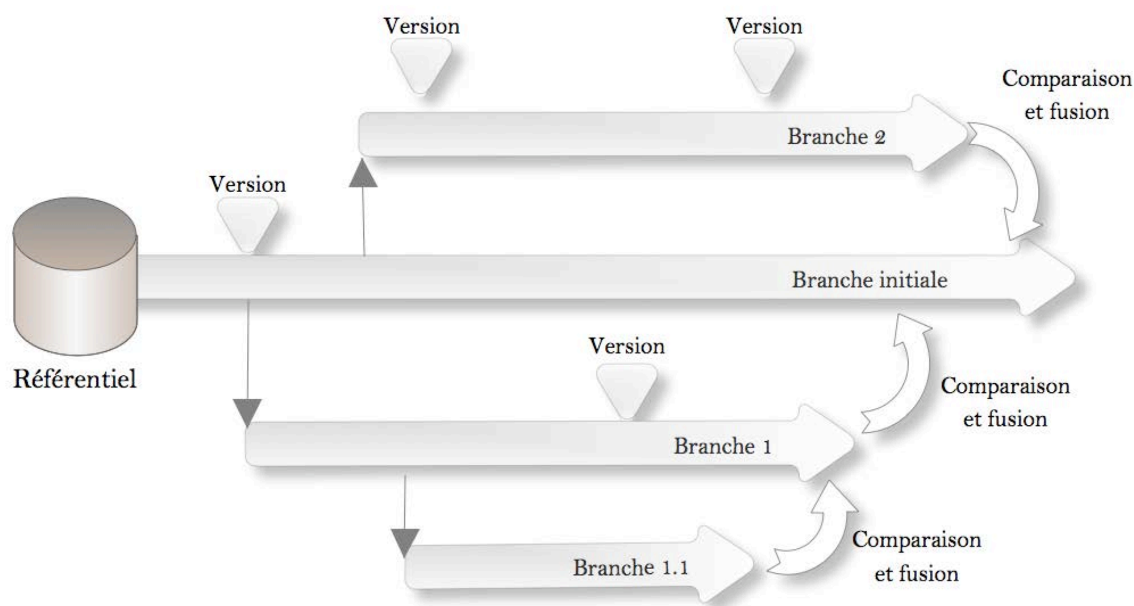


Figure 3.6 - Cycle de vie et traçabilité des données.

Le concept de branche est traditionnellement utilisé par les produits de gestion de version dans le domaine de l'ingénierie du logiciel à destination des informaticiens. Appliqué au domaine de la gestion des données de référence, il faut que l'outil de MDM offre une gestion de version similaire, au travers de branches et de versions sur les branches, avec une interface homme-machine orientée métier. Cette interface permet aux utilisateurs métier et informaticiens d'administrer les versions sans devoir maîtriser les outils techniques de l'ingénierie du logiciel.

Les cas d'usage des branches sont vastes. Les principes de départ suivants sont intéressants à prendre en compte :

- *Branche de travail* : Permet de valoriser les données sans que celles-ci soient encore disponibles pour les systèmes consommateurs. La branche de travail permet aussi de créer des versions *jetables* des valorisations de données, par exemple, en phase de développement pour des tests intermédiaires. Ces versions ne seront alors pas fusionnées sur les autres branches, par exemple celles en charge de la publication du référentiel vers les systèmes consommateurs des données.
- *Branche de publication* : Reçoit les données en provenance des branches de travail (mécanisme de fusion). La branche de publication n'est pas modifiable manuellement (gestion des permissions), seule la fusion permet de l'alimenter.
- *Branche d'administration* : Contient les métadonnées sur les autres branches du système. Dans cette branche se trouve les identifiants des autres branches de travail et de publication, les tables de transcodification, les autres référentiels techniques nécessaires comme les tables d'abonnement aux données, les déclarations des programmes de mapping, etc.

Les branches de travail et de publication peuvent aussi correspondre à des environnements logiques de travail qui reflètent le cycle de vie du développement des systèmes : développement, test unitaire, test intégré, qualification, pré-recette, formation, recette, site pilote, pré-production, production, etc. Les branches peuvent également être utilisées afin de prendre en compte des versions différentes des structures de données, c'est-à-dire des schémas XML (XSD) :

- Branche V1 contenant la version V1 de schémas XML.
- Branche V2 contenant la version V2 de schémas XML.

Les mécanismes de migration de données d'une version à l'autre se mènent alors entre les branches (voir aussi section suivante).

Une combinaison de ces différents cas d'usage permet de bâtir une solution appropriée selon le contexte de l'entreprise. L'outil MDM doit aussi permettre le déploiement entre référentiels physiques, par exemple d'une machine vers une autre, à l'aide de fonctions d'import/export, indispensables pour parcourir les environnements liés au cycle de vie du développement des systèmes. Ces fonctions d'import/export peuvent fonctionner par différentiel de valeurs et pas seulement en « annuler/remplacer » de la totalité du référentiel.

Dans EBX.Platform lorsqu'une version d'une branche est créée, c'est l'ensemble de celle-ci que est versionné, c'est-à-dire l'ensemble des instances relatives aux schémas XML définis dans cette branche du référentiel. Cette gestion de version est optimisée par l'outil ; elle n'a pas de conséquence sur les performances et les volumes.

Comme nous avons pu le voir, un modèle d'adaptation est un modèle de données défini au moyen de XML Schema. Pour être interprétés par EBX.Platform ces modèles doivent définir des propriétés spécifiques que nous présenterons par la suite.

3.5.3 Propriétés d'un modèle d'adaptation

Pour être accepté par EBX.Platform, un schéma XML doit au moins posséder une déclaration d'un élément racine et cette racine doit avoir l'attribut osd:access="--" :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:osd="urn:ebx-schemas:common_1.0" xmlns:fmt="urn:ebx-
  schemas:format_1.0">
     <xs:import namespace="urn:ebx-schemas:common_1.0"
        schemaLocation="http://schema.orchestranetworks.com/common_1.0.xsd"/>
     <xs:element name="root" osd:access="--">
     …
     </xs:element>
</xs:schema>
```

Figure 3.7. Exemple de déclaration d'un modèle d'adaptation contenant une seule racine.

Dans les modèles d'adaptation définis à l'aide de XML Schema, nous pouvons considérer principalement cinq types de nœuds à savoir simples, simples multi-occurrencés, complexes, complexes multi-occurrencés et tables.

3.5.3.1 Les types de nœuds

3.5.3.1.1 Nœuds simples

Les nœuds simples sont définis à l'aide de la balise *element* (figure 3.8) définie par XML Schema :

```
<xs:element name="noeud" type="xs:integer">
        <xs:annotation>
           <xs:documentation>
               <osd:label>Exemple de nœud simple</osd:label>
           </xs:documentation>
        </xs:annotation>
</xs:element>
```

Figure 3.8. Exemple de déclaration d'un nœud simple

3.5.3.1.2 Les nœuds simples multi-occurrencés

Les nœuds simples multi-occurrencés sont définis de la même manière qu'un nœud simple, mais permettent de définir une liste composée d'éléments identiques :

```
<xs:element name="noeud" type="xs:integer" minOccurs="0" maxOccurs= "100" >
    <xs:annotation>
        <xs:documentation>
            <osd:label>Exemple de nœud simple multi occurencé</osd:label>
        </xs:documentation>
    </xs:annotation>
</xs:element>
```

Figure 3.9. Exemple de déclaration d'un nœud simple multi-occurrencé

Les attributs *minOccurs* et *maxOccurs* (figure 3.9) indiquent le nombre d'éléments minimal et maximal composant la liste.

3.5.3.1.3 Les nœuds complexes

Les nœuds complexes sont définis par XML Schema par la balise *xs:complexType* (figure 3.10). En d'autres termes, ce sont des nœuds contenant d'autres nœuds (simples, complexes ou tables) :

```
<xs:element name="noeudComplex" >
    <xs:complexType>
        <xs:sequence>
            <xs:element name="element1" type="xs:integer">
                <xs:annotation>
                    <xs:documentation>
                        <osd:label>Element simple 1</osd:label>
                    </xs:documentation>
                </xs:annotation>
            </xs:element>
            <xs:element name="element2" type="xs:string">
                <xs:annotation>
                    <xs:documentation>
                        <osd:label> Element simple 2</osd:label>
                    </xs:documentation>
                </xs:annotation>
            </xs:element>
        </xs:sequence>
```

```
        </xs:complexType>
    </xs:element>
```

Figure 3.10. Exemple de déclaration d’un nœud complexe.

3.5.3.1.4 Les nœuds complexes multi-occurrencés

De la même manière que les éléments simples multi-occurrencés, il est possible de définir des éléments complexes multi-occurrencés (figure 3.11) :

```
    <xs:element name="noeudComplex” minOccurs=”0” maxOccurs= “100“>
        <xs:complexType>
            <xs:sequence>
                <xs:element name=”element1” type=”xs:integer”>
                    <xs:annotation>
                        <xs:documentation>
                            <osd:label>Element simple 1</osd:label>
                        </xs:documentation>
                    </xs:annotation>
                </xs :element>
                <xs:element name=”element2” type=”xs:string”>
                    <xs:annotation>
                        <xs:documentation>
                            <osd:label> Element simple 2</osd:label>
                        </xs:documentation>
                    </xs:annotation>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
```

Figure 3.11. Exemple de déclaration d’un complexe multi-occurrencé.

3.5.3.1.5 Les nœuds tables

Dans EBX.Platform, il est possible de définir des tables de la même manière que dans les bases de données relationnelles où les attributs de la table relationnelle sont déclarés comme une séquence d’éléments simples dans EBX.Platform (figure 3.12).

La déclaration d’une table s’effectue de la façon suivante :

```
<xs:element name="product" minOccurs="0" maxOccurs="unbounded"">
    <xs:annotation>
        <xs:documentation>
            <osd:label>Table des Produits</osd:label>
            <osd:description>Liste des produits du catalogue</osd:description>
        </xs:documentation>
    </xs:annotation>
    <xs:complexType>
    <xs:annotation>
        <xs:appinfo>
            <osd:table>
                <primaryKeys>/productRange /productCode</primaryKeys>
            </osd:table>
        </xs:appinfo>
    </xs:annotation>
        <xs:sequence>
            <xs:element name="productRange" type="xs:string"/><!-- clé -->
            <xs:element name="productCode" type="xs:string"/><!-- clé -->
            <xs:element name="productLabel" type="xs:string"/>
            <xs:element name="productDescription" type="xs:string"/>
            <xs:element name="productWeight" type="xs:int"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Figure 3.12. Exemple de déclaration d'une table contenant 5 champs.

Les contraintes sur les clés primaires sont définies à l'aide de facettes étendues spécifiques à EBX.Platform que nous présenterons par la suite.

EBX.Platform propose des contraintes étendues, interprétées en JAVA, lorsque XML Schema ne permet pas de spécifier des contrôles suffisamment avancés.

3.5.3.2 Contraintes étendues

Afin de conserver la conformité du document par rapport à la norme XML Schema, ces facettes sont définies sous l'élément *annotation/appinfo/otherFacets*. Nous reviendrons sur la validité de ces balises dans la quatrième partie de ce document. Les facettes suivantes ne

sont qu'une partie des facettes disponibles dans EBX.Platform. Nous présentons quelques facettes permettant d'appréhender les extensions apportées à XML Schema.

3.5.3.2.1 Facettes dynamiques

Les facettes dynamiques permettent d'ajouter des contraintes par rapport à un élément défini dans le modèle d'adaptation ou bien dans le même contexte et peuvent être de natures suivantes :

- *Length* : contrainte sur la longueur d'un élément ;
- *minLength* : contrainte sur la longueur minimale d'un élément ;
- *maxLength* : contrainte sur la longueur maximale d'un élément ;
- *maxInclusive* : contrainte sur la borne maximale incluse d'un élément ;
- *maxExclusive* : contrainte sur la borne maximale exclue d'un élément ;
- *minInclusive* : contrainte sur la borne minimale incluse d'un élément ;
- *minExclusive* : contrainte sur la borne minimale exclue d'un élément.

Par rapport aux balises standard définies par XML Schema, le nom de l'élément est conservé, cependant la propriété *value* est remplacée par la propriété *path*.

```
<xs:element name ="amount">
    <xs:annotation>
        <xs:appinfo>
            <osd:otherFacets>
                <osd:minInclusive path ="/path "/>
            </osd:otherFacets>
        </xs:appinfo>
    </xs:annotation>
…
</xs:element>
```

Figure 3.13. Exemple d'utilisation d'une contrainte dynamique.

Dans l'exemple de la figure 3.13, la borne de la facette *minInclusive* est définie dynamiquement, la valeur de la borne est détenue par le nœud donné en paramètre.

3.5.3.2.2 Contrainte d'intégrité sur les tables (clés étrangères)

De même que pour la définition de clés primaires rattachées à une table, EBX.Platform permet de définir des clés étrangères, au même sens que dans le domaine des

SGBD. Une référence à une clé primaire de table est définie par une balise étendue *tableRef* possédant les attributs suivants :

- *Container* : référence de l'instance contenant la table,
- *tablePath* : expression XPath indiquant le chemin de la table cible dans le modèle,
- *labelPaths* : expression XPath permettant de composer un libellé informatif,
- *displayKey* spécifie si la clé primaire est affichée en préfixe du libellé.

```
<xs:annotation>
    <xs:appinfo>
        <osd:otherFacets>
            <osd:tableRef>
                <tablePath>../catalog</tablePath>
                <labelPaths>/productLabel /productWeight</labelPaths>
                <displayKey>true</displayKey>
            </osd:tableRef>
        </osd:otherFacets>
    </xs:appinfo>
</xs:annotation>
```

Figure 3.14. Exemple de définition d'une contrainte de clé étrangère.

Les propriétés présentées dans cette section représentent les principes de base d'un modèle d'adaptation. D'autres propriétés existent permettant de spécifier des contraintes avancées et d'apporter plus de sémantique à la structure d'un modèle d'adaptation. Ces propriétés seront présentées dans la seconde partie de cette thèse lors de la définition du profil UML associé au Master Data Management.

3.5.4 Exemple de définition d'un modèle d'adaptation

Dans cette section nous allons présenter un exemple de modèle d'adaptation défini à l'aide des propriétés exposées dans la section précédente. Soit le modèle relationnel suivant :

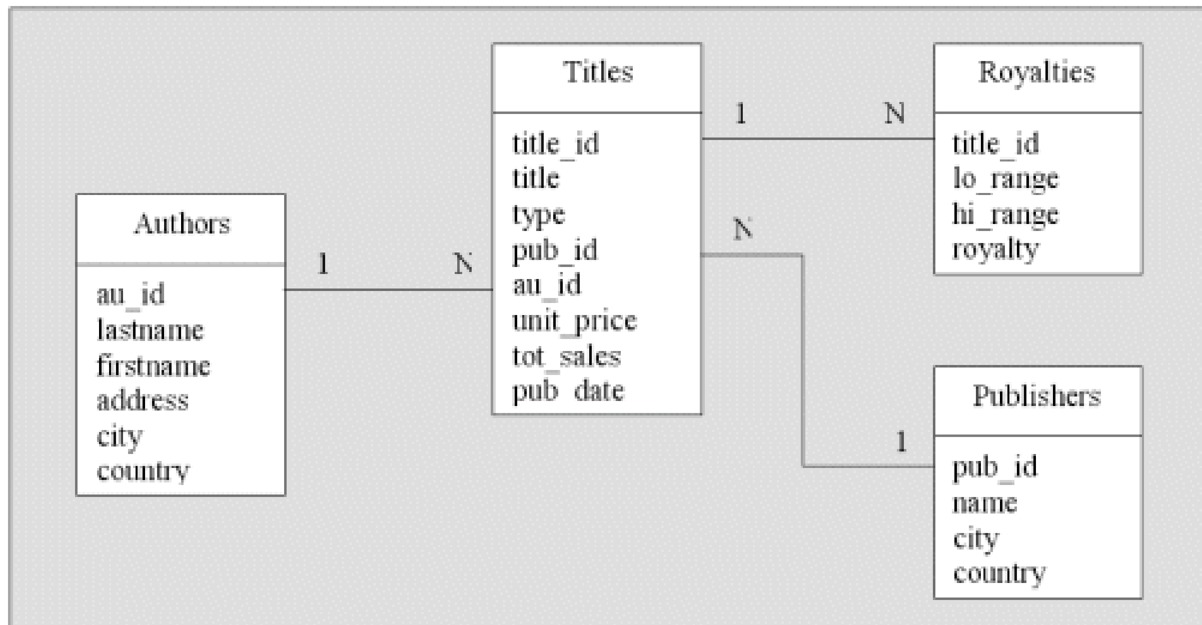


Figure 3.15. Modèle relationnel d'une base de données de publication d'ouvrages.

Le modèle relationnel présenté dans la figure 3.15 définit une base de données de publication d'ouvrages. Cette base de données est composée de quatre tables :

- *Publishers* est la table définissant les informations concernant des éditeurs. Cette table contient les numéros d'identification, noms, villes et pays des éditeurs.
- *Authors* est la table définissant les informations concernant des auteurs. Cette table contient les numéros d'identification, noms, prénoms, adresses, villes et pays des auteurs.
- *Titles* est la table définissant les informations concernant des ouvrages. Cette table contient les numéros d'identification, noms, types, identifiants des éditeurs, prix, etc.
- *Royalties* est la table contenant les informations sur les ventes d'ouvrages. Cette table contient les numéros d'identification des ouvrages et les bénéfices associés.

La figure 3.16 présente un extrait de la structure XML Schema correspondant au modèle d'adaptation de la base de données des publications.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="root" osd:access="--">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Titles" type="Title"   minOccurs="0"
          maxOccurs="unbounded"/>
        <xs:element name="Publishers" type="Publisher" minOccurs="0"
```

```
                maxOccurs="unbounded"/>
            <xs:element name="Authors" type="Author" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="Royalties" type="Royalty" minOccurs="0"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="Publisher">
    <xs:annotation>
        <xs:appinfo>
            <osd:table>
                <primaryKeys>/pub_id</primaryKeys>
            </osd:table>
        </xs:appinfo>
    </xs:annotation>
    <xs:sequence>
        <xs:element name="pub_id">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:maxLength value="4"/>
                    <xs:pattern value="[0-9]{4}"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
        <xs:element name="name">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:maxLength value="40"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
        <xs:element name="city" minOccurs="0">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:maxLength value="20"/>
```

```
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
...
</xs:schema>
```

Figure 3.16. Extrait d'un modèle d'adaptation.

Une des problématiques du Master Data Management est de minimiser les outils permettant de gérer les données de référence. EBX.Manager est un outil web conçu pour offrir une interface utilisateur riche et sécurisée permettant de gérer les données de référence. EBX.Manager est basé sur un mécanisme générant automatiquement une interface utilisateur à partir d'un modèle de données. Cela signifie qu'une fois qu'un modèle de données d'adaptation a été conçu, les utilisateurs sont en mesure de créer des instances, de modifier des valeurs et de valider les données directement à partir de EBX.Platform. Les figures 3.17 et 3.18 présentent la visualisation du modèle d'adaptation que nous venons de définir dans EBX.Manager.

Books publications Zoom

Authors

Book authors

New 1 -> 5 of 5 - [Key] - [Print] Show 10 by page

Form	Id	Lastname	Firstname	Address	City	C
Added	10-8-910	Sempf	Bill			
Added	24-1-865	Holzner	Steve			
Added	30-1-683	Aaron E.	Walsh			
Added	75-2-770	Burd	Barry			
Added	84-8-776	Valade	Janet			

1 -> 5 of 5 - [Key] - [Print] Show 10 by page

Figure 3.17. Visualisation d'une table dans EBX.Manager.

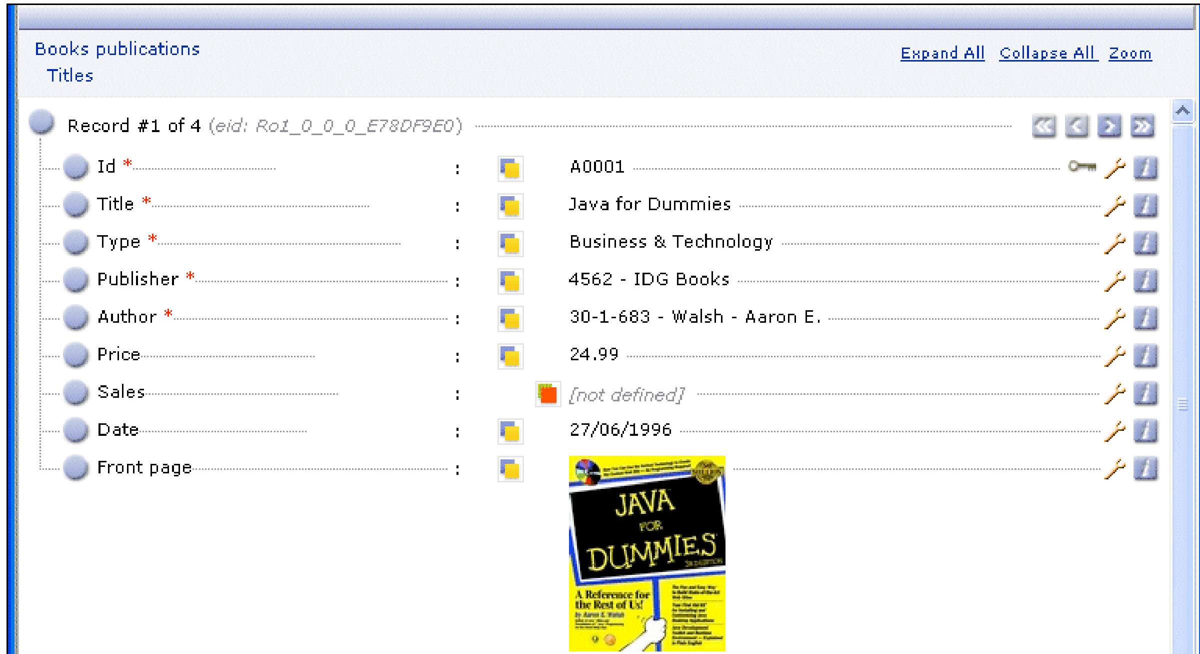


Figure 3.18. Visualisation d'un enregistrement.

3.6. Conclusion

Par rapport aux solutions existantes basées sur l'approche virtuelle et l'approche matérialisée, une solution de Master Data Management permet de fédérer non seulement les sources de données, mais aussi de gérer plus efficacement les données par une gestion optimisée du cycle de vie des données et l'utilisation d'un unique outil sécurisé. Nous avons présenté notre solution MDM EBX.Platform qui permet de répondre de manière générique à un ensemble de fonctionnalités liées aux problématiques du MDM :

- Gestion des versions de données. Une même donnée peut avoir des valeurs différentes selon le contexte dans lequel elle est valorisée (langues, segmentations de marché, environnements, profil applicatif, etc.).
- Héritage des valorisations selon un arbre de contextes. Par exemple, à partir d'une valorisation par défaut des données, il est possible de créer des contextes enfants qui permettront de surcharger les valeurs par défaut.
- Gestion des habilitations. Un mécanisme interne permet de définir des permissions portant sur des actions (action autorisée ou action non autorisée) et des droits d'accès (non visible, lecture et lecture/écriture). Les principales entités contrôlées par les permissions sont les branches, les versions, les adaptations, les tables et les nœuds d'adaptation. Les permissions peuvent être spécifiées de deux manières : soit par les

utilisateurs autorisés, au moyen de EBX.Manager, soit par les développeurs, au moyen des règles de permissions programmatiques et contextuelles.

- Sécurité. L'authentification à EBX.Platform s'effectue d'une manière sécurisée (HTTPS, SSO, etc.) par l'intermédiaire d'un protocole interne de gestion d'annuaires.
- Traçabilité des actes. Des services d'historisation sont disponibles afin d'auditer les actions intervenues dans le référentiel.
- Accessibilité et convivialité. EBX.Manager représente une interface utilisateur unique permettant d'accéder de manière simple et conviviale aux données.
- Interrogation des données. Les données peuvent être accédées de différentes manières : API, Webservices, interface utilisateur.
- Qualité des données. La définition de contraintes sur les données garantit une conformité de celles-ci par rapport au modèle. La gestion de l'héritage entre instances permet d'éviter les problématiques liées à la redondance de données.
- Historisation et version des données. EBX.Platform fournit un ensemble de fonctionnalités qui permettent de créer et gérer de multiples *branches* et *versions* de Master Data dans un référentiel. Grâce aux branches, il est possible d'effectuer des modifications concurrentes dans un référentiel (projets, environnements, brouillons, ...), de les comparer et fusionner de façon interactive. Une version permet de figer des "images" de branches afin de conserver un état stable et de détecter les modifications ultérieures.
- Workflow intégré. Le Workflow intégré permet de définir des processus métier pour faciliter la gestion des données de référence. Un processus métier est un enchaînement de tâches qui peuvent être soit automatiques, soit manuelles. A cela s'ajoute la possibilité d'aiguillage conditionnel entre ces tâches. Le module Workflow défini dans EBX.Platform s'axe sur trois priorités :
  - **Une interaction utilisateur intuitive.** Les utilisateurs ont une vue claire des tâches auxquelles ils participent et de celles qu'ils doivent réaliser.
  - **Fiabilité.** Les données de référence doivent être gérées de la manière la plus sûre possible ; par conséquent, il est essentiel que les interactions MDM soit effectuées de manière sécurisée, robuste et dans un Framework déclaratif. Les fonctionnalités de déclaration assurent une spécification explicite des processus, et renforcent la traçabilité de ces interactions.
  - **Flexibilité.** Bien qu'apparemment similaires, les demandes métier pour les processus MDM sont en réalité très variées, notamment pour ce qui concerne l'ordonnancement des tâches, la nature de ces tâches (automatique ou interaction utilisateur), et leurs objectifs à savoir la gestion du cycle de vie (branches, versions, et fusions) ou orientées sur les mises à jour et la validation des Master Data. De plus, le périmètre et le résultat attendu de chaque tâche peuvent être très varié.

L'intérêt de notre outil MDM EBX.Platform est d'offrir, de manière générique, ces fonctions à valeur ajoutée au sein d'un Système d'Information.

Concernant l'aspect modélisation d'un modèle d'adaptation celui-ci peut se mener suivant deux scénarii, soit directement à l'aide d'un éditeur XML Schema souvent associé à une notation (représentation des modèles) propriétaire, soit en utilisant un formalisme abstrait sous condition d'être capable de générer *via* une approche d'Ingénierie Dirigée par les Modèle (*IDM*) les modèles XML Schema nécessaire au fonctionnement de la solution MDM EBX.Platform. Le choix d'une architecture XML dans EBX.Platform permet de définir des modèles de données riches, structurés, fortement typés et d'associer des contraintes métier. Cependant, si l'utilisation de XML est appropriée à la définition des modèles, elle nécessite une connaissance approfondie de ce langage par les différents acteurs impliqués dans le processus de définition d'un modèle de données. Ceci nous a suggéré l'introduction d'une démarche guidant les concepteurs de modèles de données de façon qu'ils puissent se concentrer uniquement sur la modélisation et l'intégration de données et non sur la technologie à utiliser [Bézivin *et al.*, 2001]. L'adoption d'une approche objet et standard pour améliorer la compréhension du modèle et la sémantique associée (sémantique MDM dans nos perspectives) semble être une voie simple et efficace. Aussi, l'objectif principal de nos travaux est de suivre une approche d'Ingénierie Dirigée par les Modèles pour faire abstraction de la couche technologique (physique) au profit de la couche fonctionnelle (logique).

Dans le chapitre suivant, nous introduirons le concept d'Ingénierie Dirigée par les Modèles, nous présenterons ses principes et comment mettre en application cette approche.

Chapitre 4

L'Ingénierie Dirigée par les Modèles

Résumé. Dans le chapitre précédent nous avons présenté l'approche Master Data Management (MDM). Le MDM se focalise sur l'unification des modèles et outils, et une gestion optimisée du cycle de vie des données au sein d'un système d'information. Nous avons vu que la définition d'un modèle de données pivot s'avère être une étape délicate et relevant essentiellement de compétences techniques liées à la technologie spécifique utilisée. La solution permettant de s'abstraire de ces spécificités techniques est introduite par une approche d'Ingénierie Dirigée par les modèles. Ce chapitre a pour objectif de présenter les principes de l'Ingénierie Dirigée par les Modèles (IDM).

4.1 Introduction

L'Object Management Group (OMG), consortium de plus 1000 entreprises, initie la démarche d'Ingénierie Dirigée par les Modèles (IDM). Dans la littérature le terme « démarche », « norme » ou « approche » qualifie l'IDM [OMG, 2001].

Cette approche a pour but d'apporter une nouvelle vision unifiée permettant de concevoir des applications en séparant la logique métier de l'entreprise, de toute plateforme technique. En effet, la logique métier est stable et subit peu de modifications au cours du temps, contrairement à l'architecture technique. Il est donc évident de séparer les deux pour faire face à la complexité des systèmes d'information et aux coûts excessifs de migration technologique. Cette séparation autorise alors la capitalisation du savoir logiciel et du savoir-faire de l'entreprise. A ce niveau, l'approche objet et l'approche composant n'ont pas su tenir leurs promesses. Il devenait de plus en plus difficile de développer des logiciels basés sur ces technologies. Le recours à des procédés supplémentaires, comme le patron de conception ou la programmation orientée aspect, était alors nécessaire. L'IDM doit aussi offrir la possibilité de stopper l'empilement des technologies qui nécessite de conserver des compétences particulières pour faire cohabiter des systèmes divers et variés. Ceci est permis grâce au passage d'une approche interprétative à une approche transformationnelle. Dans l'approche interprétative, l'individu a un rôle actif dans la construction des systèmes informatiques alors que dans l'approche transformationnelle, il a un rôle simplifié et amoindri grâce à la construction automatisée.

Les principaux objectifs de l'IDM sont la portabilité, l'interopérabilité et la réutilisation. Dans ce contexte l'OMG a défini une approche spécifique de l'IDM appelée Model Driven Architecture (MDA) [Miller *et al.*, 2003] et basée sur les problématiques suivantes :

- Spécification d'un système indépendamment de la plateforme sur laquelle ce système doit être déployé,
- Spécification de plateformes,
- Définition d'une plateforme pour un système,
- Transformation des spécifications d'un système en un système associé à une plateforme particulière.

Les termes « *plateforme* » et « *plateforme indépendante*» sont intensément utilisés dans la littérature abordant l'IDM. La notion de *plateforme indépendante* fait référence à la capacité d'être indépendant des spécificités d'une plateforme. La technique la plus fréquemment utilisée pour être indépendant d'une plateforme est d'utiliser une technologie neutre représentée par une machine virtuelle. Une machine virtuelle peut aussi être considérée comme étant une plateforme. Tout modèle visant une machine virtuelle est alors considéré comme étant spécifique à cette plateforme, mais indépendant de la plateforme sous jacente. Cependant, d'une manière contradictoire, [Tekinerdogan *et al.*, 2003] indiquent que lors de la spécification d'un système indépendamment d'une plateforme, il est toutefois nécessaire d'avoir une idée de la plateforme visée par le développement de ce système. Autrement dit, la spécification d'un système est indépendante de toute les plateformes si ce système dépend uniquement de fonctionnalités qui sont communes ou pouvant être associées à toutes les plateformes.

La figure 4.1 montre l'impact d'une approche IDM dans le processus de développement d'un système.

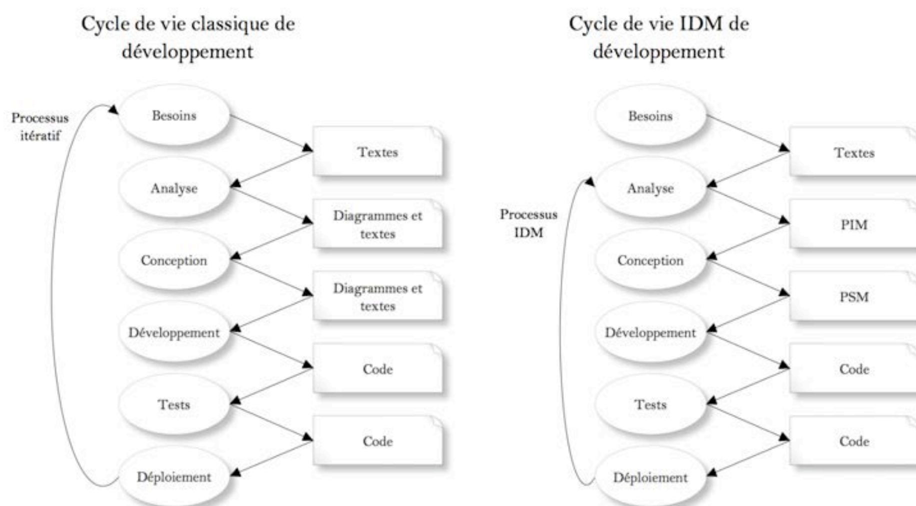


Figure 4.1. L'impact de l'IDM dans le processus de développement logiciel.

La partie gauche de la figure présente le cycle traditionnel de développement d'un système basé sur une approche itérative de développement. En théorie dans un processus itératif de développement, lorsqu'un cycle arrive à son terme et qu'un nouveau cycle est requis, il est nécessaire d'itérer de nouveau sur les besoins fonctionnels d'un système afin que les travaux réalisés dans les cycles précédents restent synchronisés. En pratique, le fait que seules les phases reliées au code du système soient automatisées rend cette approche itérative difficilement réalisable. Les retours d'expérience à l'issue d'un premier cycle doivent être pris en compte dans toutes les étapes d'un nouveau cycle de développement afin d'obtenir un système correspondant aux nouvelles exigences. Dans le cas où les nouvelles exigences sont remontées au niveau de la spécification des besoins, cela n'aura pas pour finalité la production d'un système entièrement en adéquation avec les nouveaux besoins. En effet, dans une approche classique de développement, les nouveaux besoins doivent être traduits manuellement de la plus haute couche d'abstraction jusqu'à la couche de code. C'est pour cette raison que dans les cycles classiques de développement les nouveaux besoins sont directement pris en compte dans les couches correspondantes au code du système. Avec une telle pratique nous obtenons une désynchronisation entre les hautes couches d'abstraction correspondant à la spécification d'un système et les basses couches correspondant au code du système.

L'Ingénierie Dirigée par les Modèles (partie droite de la figure 4.1) a été définie pour pallier les problèmes présents dans les approches classiques de développement. Dans une approche IDM, les niveaux d'analyse et de conception d'un système font partie intégrante du processus automatisé de développement. Il devient alors plus aisé dans une approche itérative de remonter les nouveaux besoins aux niveaux les plus hauts du processus de développement.

4.1.1 Architecture IDM

La Figure 4.2 présente l'architecture quatre couches de l'Ingénierie Dirigée par les Modèles :

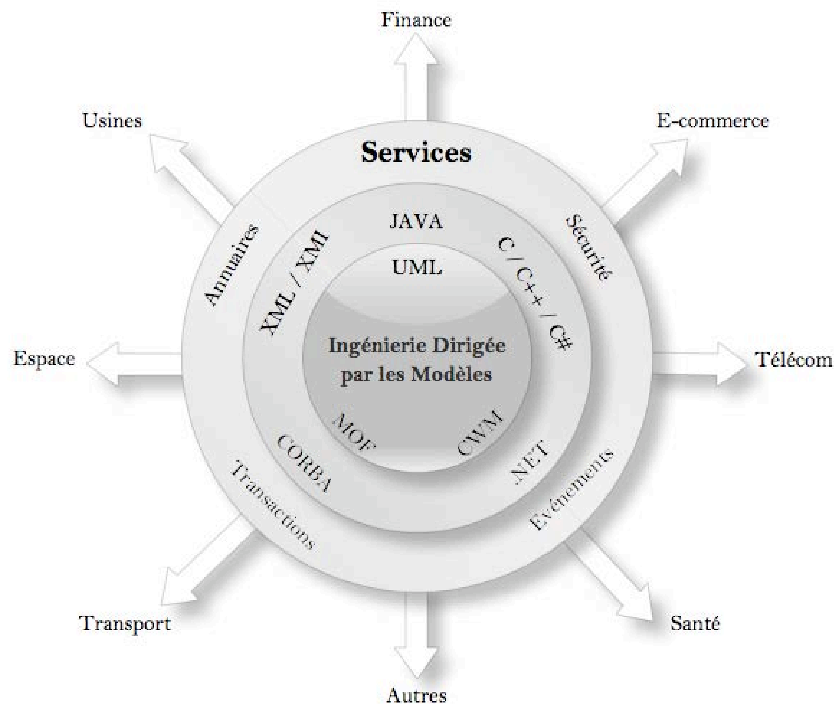


Figure 4.2. Architecture IDM.

Au cœur de l'Ingénierie Dirigée par les Modèles, on retrouve les technologies UML (Uniform Modelling Language) [Avgeriou *et al.* 2005], MOF (Meta Object Facility) [Overbeek 2006] et CWM (Common Warehouse Model) [Pool *et al.*, 2003] spécifiées par l'OMG pour modéliser la logique métier de l'application. Ce modèle métier est spécialisé ensuite dans une technologie middleware. On retrouve les standards actuels tels que les EJB [Sierra *et al.*, 2003], Corba [Schlidt *et al.*, 2000], .NET [Watkins *et al.*, 2002] et les WebServices [Castagna *et al.*, 2009]. L'anneau extérieur du cercle représente les services. Ceux-ci permettent par exemple de gérer la transaction, la persistance, les évènements. Enfin, la couche spécifique au domaine prend place à la périphérie du noyau. Elle se base sur des profils UML et permet de proposer des frameworks spécifiques au domaine d'application de l'application (Espace, Télécommunication, Santé, etc.).

4.1.2 Concepts de base

Dans cette section, nous nous proposons de définir les concepts de base de l'Ingénierie Dirigée par les Modèles afin d'appréhender de manière optimale cette approche :

- *Système* : on parle toujours d'IDM dans le cadre d'un système existant ou à réaliser. Un système peut tout inclure : un programme, une certaine combinaison de

différentes parties de différents systèmes, une fédération de systèmes autonomes, une fédération d'entreprises, etc. Toutefois, dans les systèmes, on s'intéresse surtout aux logiciels.

- *Point de vue :* un point de vue sur un système est une vision totale sur ce système s'intéressant à un aspect particulier. Par exemple, nous pouvons nous intéresser aux flux d'informations, et donc décrire uniquement le cheminement des données dans tout le système. La norme RM-ODP (Reference Model of Open Distributed Processing) définit cinq points de vue [Naumenko *et al.* 2001] :
  - *Point de vue Entreprise* : il représente la logique métier (le Pourquoi). Il exprime les objectifs, les droits et les obligations des entités qui composent une application.
  - *Point de vue Information* : il représente le système d'information du système (le Quoi). Il exprime la sémantique de l'application à l'aide de structures de données, de fonctions les manipulant et de propriétés d'intégrité sur ces données.
  - *Point de vue Traitement* : il représente la conception fonctionnelle de l'application (le Comment). Il exprime une vision définissant des entités fonctionnelles, en précisant leurs interactions et en déterminant les propriétés des fonctions correspondantes.
  - *Point de vue Ingénierie* : il représente les plates-formes support de l'application (le Où). Il décrit les fonctions et services de base que doit fournir l'infrastructure d'exécution pour que l'on puisse construire les mécanismes assurant l'interopérabilité en environnement hétérogène.
  - *Point de vue Technologie* : il représente les produits que va utiliser l'application (le Avec Quoi). Il expose les contraintes technologiques imposées par les mécanismes d'ingénierie correspondant à une infrastructure matérielle ou logicielle spécifique.
- *Architecture* : l'architecture d'un système est la spécification des parties et des connecteurs du système, ainsi que les règles d'interaction entre ces parties, utilisant les connecteurs.
- *Modèle* : le modèle d'un système est la spécification formelle des fonctions, de la structure et/ou du comportement de ce système dans son environnement, dans un certain but. Un modèle est souvent représenté par des schémas et du texte. Le texte peut être exprimé dans un langage de modélisation ou en langage naturel.
- *Application* : le terme application désigne une fonctionnalité qui est en cours de développement. Un système décrit une ou plusieurs application(s) supportée(s) par une ou plusieurs plates-formes.
- *Plateforme* : une plateforme est un ensemble de sous-systèmes et de technologies, qui fournit aux applications supportées, un ensemble cohérent de fonctionnalités à travers des interfaces et des gabarits masquant leurs détails d'implémentation. Les plateformes peuvent être de type objet (supporte les objets avec interfaces, requête/réponse aux services), de type batch (supporte une séquence de programmes indépendants s'exécutant consécutivement) ou de type flux de données (supporte un flux continu de données entre les composants logiciels). Les plateformes peuvent

supporter plusieurs technologies (tels que CORBA, Java 2 Entreprise Edition) et sont implémentées par plusieurs fournisseurs (tels que WebSphere, WebLogic).

L'Ingénierie Dirigée par les Modèles repose sur les trois formalisations suivantes :

(i) Modèles,
(ii) Méta-modèles,
(iii) Transformation de modèles.

Dans les sections suivantes nous présenterons ces trois concepts.

4.2 Modèles

L'Ingénierie Dirigée par les Modèles se focalise sur des modèles et comment ceux-ci peuvent être utilisés pour créer un système [Miller *et al.*, 2003].

La structure d'un modèle est définie par une syntaxe abstraite du langage dans lequel est implémenté ce modèle. La présentation d'un modèle est définie par la syntaxe concrète du langage de modélisation. Dans la section 4.3 nous présenterons de manière plus précise la notion de structure.

Les modèles offrent de nombreux avantages dont le plus important est de spécifier différents niveaux d'abstraction, facilitant la gestion de la complexité inhérente aux applications. Les modèles possédant un niveau élevé d'abstraction sont utilisés pour présenter l'architecture générale d'une application ou sa place dans une organisation, tandis que les modèles très concrets permettent de spécifier précisément des protocoles de communication réseau ou des algorithmes de synchronisation. Même si les modèles se situent à des niveaux d'abstraction différents, il est possible d'exprimer des relations de raffinement entre eux. De véritables liens de traçabilité et des relations garantissent la cohérence d'un ensemble de modèles impliqués dans une même application.

L'objectif majeur de l'IDM est l'élaboration de modèles pérennes, indépendant des détails techniques des plateformes d'exécution (J2EE, .Net, PHP, etc.), afin de permettre la génération automatique du code des applications et d'obtenir un gain significatif de productivité.

Le principe clé de l'IDM consiste en l'utilisation de modèles aux différentes phases du cycle de développement d'une application. Plus précisément, l'IDM préconise l'élaboration de modèles d'exigence (CIM), d'analyse et de conception (PIM) et de code (PSM).

Nous pouvons donc catégoriser les modèles de la manière suivante :

- Les modèles dits CIM (Computation Independent Model).
- Les modèles dits PIM (Platform Independent Model).
- Les modèles dits PSM (Platform Specific Model).

Dans les sections suivantes nous présenterons l'ensemble de ces types de modèles.

4.2.1 Modèle CIM

C'est le modèle métier ou le modèle du domaine d'application. Le CIM permet la vision du système dans l'environnement où il opèrera, mais sans rentrer dans le détail de la structure du système ni de son implémentation. Il aide à représenter ce que le système devra exactement faire. Il est utile, non seulement comme aide pour comprendre un problème, mais également comme source de vocabulaire partagé par d'autres modèles. L'indépendance technique de ce modèle lui permet de garder tout son intérêt au cours du temps et il est modifié uniquement si les connaissances ou les besoins métier changent. Le savoir-faire est recentré sur la spécification CIM au lieu de la technologie d'implémentation. Dans les constructions des PIM et des PSM, il est possible de suivre les exigences modélisées du CIM qui décrivent la situation dans lequel le système est utilisé, et réciproquement.

4.2.2 Modèle PIM

Un modèle PIM présente une indépendance vis-à-vis de toute plateforme technique (EJB, CORBA, .NET, etc.) et ne contient pas d'informations sur les technologies qui seront utilisées pour déployer l'application. C'est un modèle informatique qui représente une vue partielle d'un CIM. Le PIM représente la logique métier spécifique au système ou le modèle de conception. Il représente le fonctionnement des entités et des services. Il doit être pérenne au cours du temps. Il décrit le système, mais ne montre pas les détails de son utilisation sur la plateforme. A ce niveau, le formalisme utilisé pour exprimer un PIM est un diagramme de classes en UML qui peut être couplé avec un langage de contrainte comme OCL (Object Constraint Language) [Richters, 2002]. Il existe plusieurs niveaux de PIM. Le PIM peut contenir des informations sur la persistance, les transactions, la sécurité, etc. Ces concepts permettent de transformer plus précisément le modèle PIM vers le modèle PSM.

4.2.3 Modèle PSM

Un modèle PSM est dépendant de la plateforme technique spécifiée par les architectes d'un système. Le PSM sert essentiellement de base à la génération de code exécutable vers la ou les plateformes techniques. Le PSM décrit comment le système utilisera cette ou ces plateformes. Il existe plusieurs niveaux de PSM. Le premier, issu de la transformation d'un PIM, se représente par un schéma UML spécifique à une plateforme. Les autres PSM sont obtenus par transformations successives jusqu'à l'obtention du code dans un langage spécifique (Java, C++, C#, etc.) Un PSM d'implémentation contiendra par exemple des informations comme le code du programme, les types pour l'implémentation, les programmes liés, les descripteurs de déploiement.

4.3 Métamodèles

Dans une approche IDM, les spécificités techniques sous-jacentes à un modèle sont écartées. L'IDM préconise l'utilisation d'un mécanisme standard et abstrait pour définir des modèles. Ce mécanisme abstrait est dénoté par le terme *métamodèle.* Ainsi dans la pratique, tout modèle défini par l'intermédiaire d'une approche IDM doit posséder un métamodèle. Nous pouvons vulgariser la notion de métamodèle en la définissant comme étant le modèle d'un modèle. Pour supporter cette approche l'OMG a introduit une architecture 4 couches de métamodélisation :

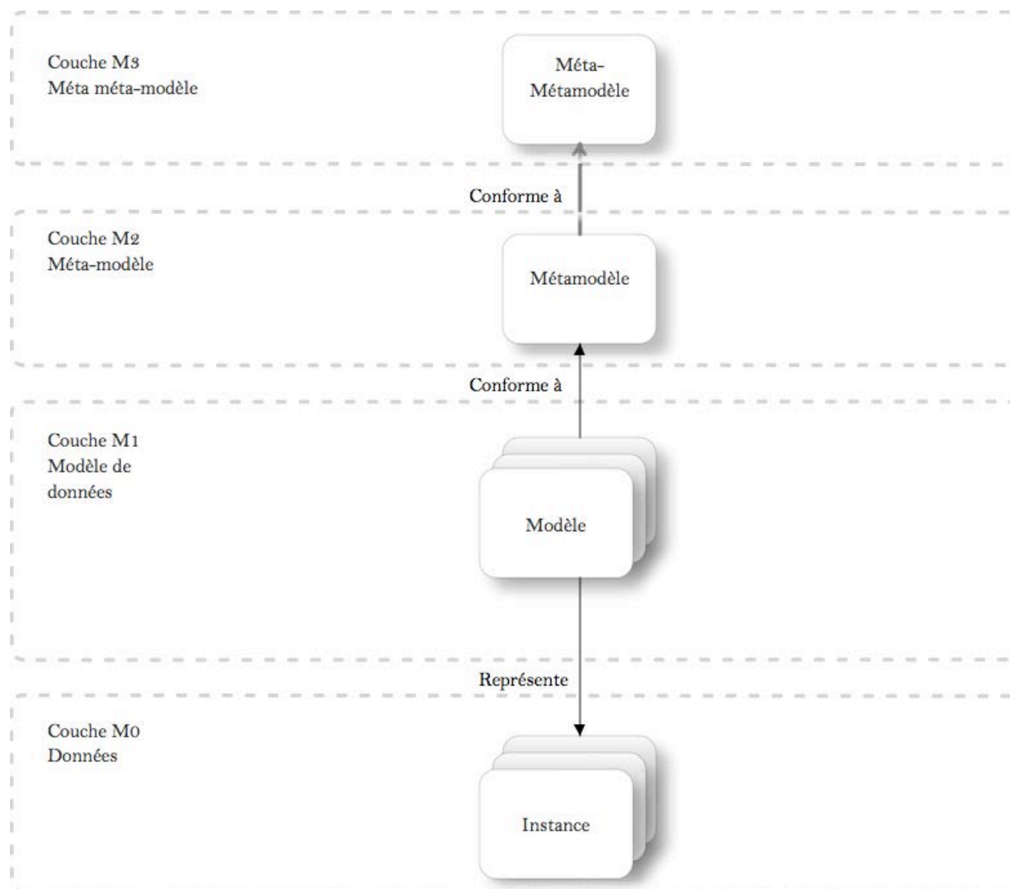


Figure 4.3. Architecture à 4 couches de métamodélisation.

Dans les sous-sections suivantes, nous présenterons le Meta Object Facility (MOF) [Overbeek 2006] qui a été défini par l'OMG comme standard pour la définition de métamodèles (section 4.3.1), son implémentation par l'Eclipse Modelling Framework (EMF)

[Salay *et al.*, 2007] (section 4.3.2), et une autre vision de la métamodélisation supportée par une approche UML (section 4.3.3).

4.3.1 Meta Object Facility

Le Meta Object Facility (MOF) a pour vocation de proposer une méthodologie pour décrire des *métadonnées*. Nous pouvons définir une métadonnée comme étant une donnée sur les données. La base du MOF repose sur un modèle définissant un langage standard de métamodélisation à utiliser dans un cadre IDM. Le modèle du MOF est décliné dans les deux versions suivantes :

- Essential MOF (EMOF) est un sous-ensemble du modèle MOF focalisé sur la structure des métamodèles. EMOF défini les méta-classes de bases telles que Class, *Property*, *Operation*, *Package*, *PrimitiveType* et *Enumeration*.
- Complete MOF (CMOF) représente le modèle MOF et donc inclut EMOF. CMOF définit des méta-classes supplémentaires qui héritent des méta-classes définies par EMOF.

Dans cette section, nous nous focaliserons sur la partie minimale du MOF à savoir EMOF. La figure 4.4 présente un extrait du métamodèle EMOF focalisé sur la représentation des classes du MOF. Nous pouvons noter que le métamodèle EMOF est situé au niveau M3 de l'architecture MOF (figure 4.3). Les classes spécifiées dans le métamodèle MOF sont utilisées pour représenter les méta-classes définies dans un métamodèle situé au niveau M2 (figure 4.3).

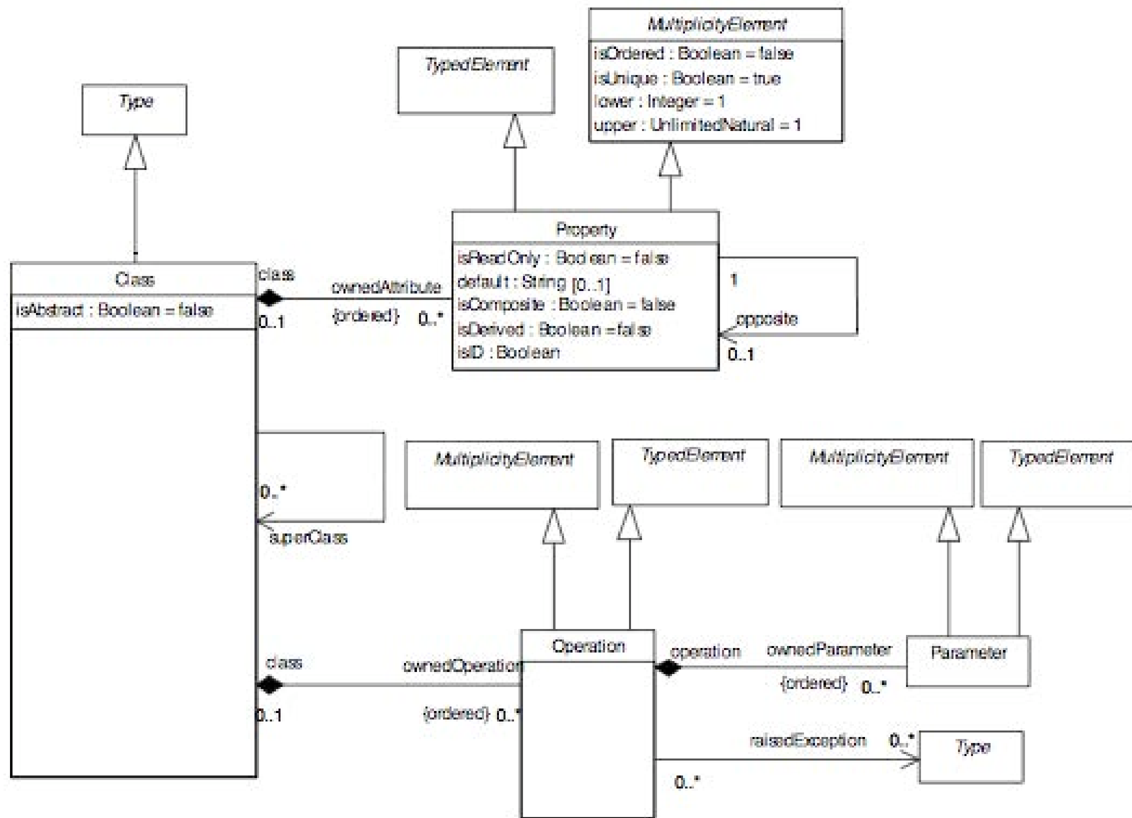


Figure 4.4. Métamodèle EMOF.

Nous pouvons prendre pour exemple le métamodèle d'UML qui est exprimé à partir du métamodèle EMOF. La figure 4.5 présente le diagramme de base du métamodèle UML. Nous pouvons constater que toutes les méta-classes définies dans le métamodèle UML sont des instances *MOF Class* et que leurs propriétés sont des instances *MOF Property*. Le diagramme présenté dans la figure 4.5 décrit les propriétés communes des entités (*Element*) définies dans UML. Selon le métamodèle UML des « *Element* » peuvent donc contenir d'autres « *Element* » et peuvent contenir des commentaires. En plus des sémantiques standards des classes, propriétés et associations, le métamodèle UML utilise d'une manière intensive les propriétés de sous-ensembles (« subsets ») et d'union. Par exemple les extrémités de l'association (« *ends* ») entre les entités « *Element* » et « *Comment* » sont des sous-ensembles des extrémités de l'association entre les entités « *Element* » et « *Element* ». En d'autres termes, l'entité « *owningElement* » est un sous-ensemble de « *owner* » et « *ownedComment* » est sous ensemble de « *ownedElement* ».

Cela signifie que toutes les valeurs d'une entité « *owningElement* » sont aussi contenues dans « *owner*» et que toutes les valeurs d'une entité « *ownedComment*» sont contenues dans « *ownedElement*».

La relation d'union permet de spécifier que les valeurs d'une propriété sont contenues au moins dans une des propriétés liées par cette relation d'union.

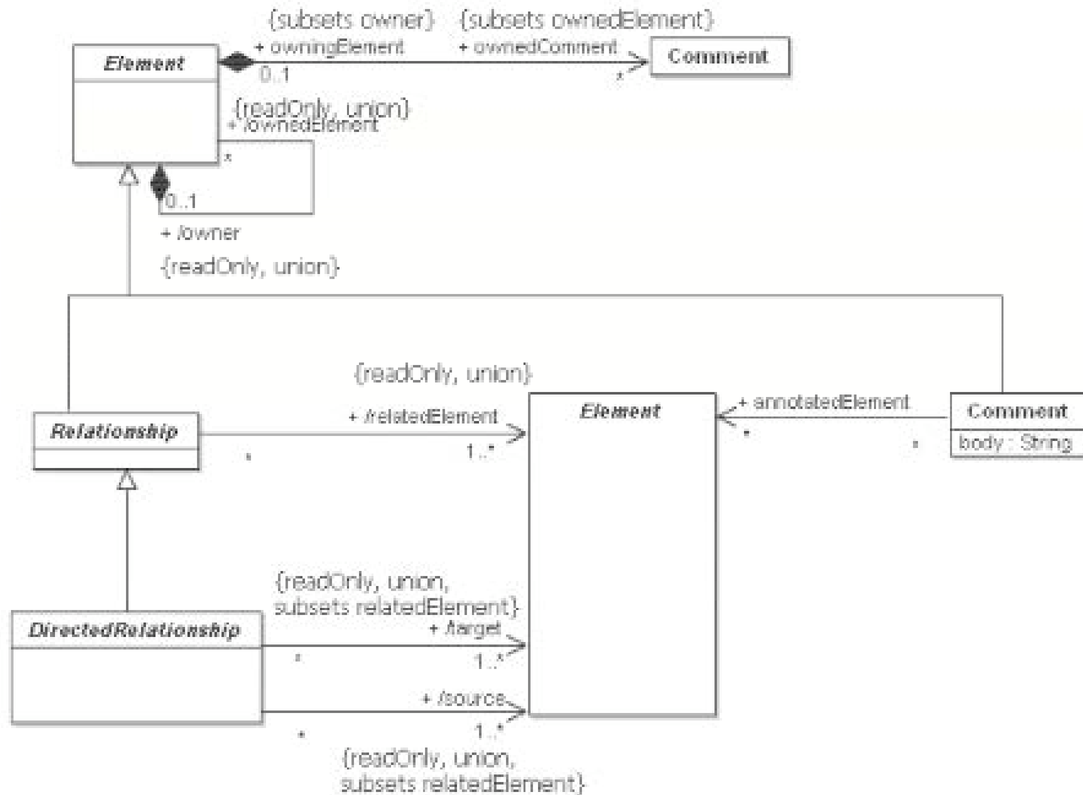


Figure 4.5. Diagramme du package principal UML.

Le fait que le métamodèle UML soit défini à partir des constructeurs du MOF rend celui-ci conforme au métamodèle du MOF. Il est aussi important de noter que le modèle du MOF est lui-même défini en utilisant les constructeurs du MOF. Les entités *Class* et *Property* sont aussi des instances de *Class.* Nous pouvons en conclure que le modèle du MOF est aussi son métamodèle.

4.3.2 Eclipse Modeling Framework

L'Eclipse Modeling Framework (EMF) [Steingerg *et al.*, 2009] est un framework de modélisation et de simplification de génération de code pour la construction d'outils et d'autres applications basés sur une structure de modèle de données. Depuis un modèle décrit en XMI (XML Metadata Interchange) [OMG 2005], EMF fournit des outils et un support de moteur d'exécution de production de classes Java pour un modèle, un jeu de classes qui permet la prévisualisation et l'édition du modèle ainsi qu'un éditeur. Les modèles peuvent

être spécifiés en utilisant des documents Java, UML, XML, puis sont importés dans EMF. Le plus important est qu'EMF fournit les fondements à l'interopérabilité avec d'autres outils ou applications basés sur EMF. EMF est conforme aux spécifications de l'EMOF. EMF propose un langage de métamodélisation appelé Ecore qui est un sous-ensemble de l'EMOF et qui diffère sensiblement de celui défini par l'EMOF. La figure 4.6 présente un extrait simplifié du métamodèle Ecore.

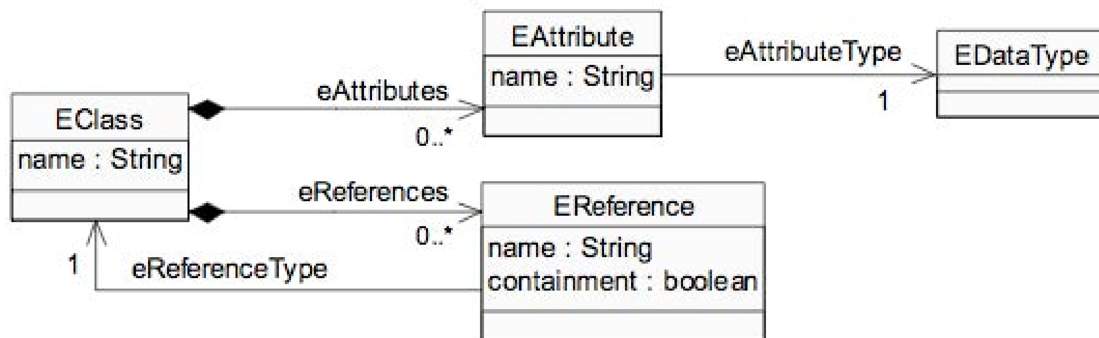


Figure 4.6 Extrait du métamodèle ECore.

Cet extrait du métamodèle Ecore présente quatre méta classes: *EClass*, *EAttribute*, *EDataType* et *EReference*. La méta-classe *EClass* est utilisée pour modéliser une classe. Une classe possède un nom, des attributs et des références. La méta-classe *EAttribute* est utilisée pour représenter des attributs. Un attribut possède un nom et un type. La méta-classe *EReference* représente la référence vers une classe utilisée par une des extrémités d'une association. Une référence possède un nom, un attribut indiquant si cette référence est une composition au sens UML, et un type de référence représentant un lien vers une classe. La méta-classe *EDataType* représente le type d'un attribut. Un type de donnée est soit un type primitif (entier, décimal, etc.) ou un objet au sens de la programmation orientée objet.

Nous pouvons constater que le nom des méta-classes Ecore est identique à la terminologie UML. Ecore préfixe d'un « E » toutes ses méta-classes. Ce préfixe permet de rendre plus facile la distinction entre les méta-classes Ecore et les méta classes UML. Le métamodèle Ecore fait de plus la distinction entre des entités de type *EAttributes* et *EReferences* tandis que l'EMOF utilise systématiquement la notion de *Property*. Cette distinction réside dans le fait que le type d'un élément *EAtttribute* est primitif, tandis que le type d'un élément *EReference* est de type *EClass*.

Les modèles produits à l'aide du framework EMF sont enregistrés dans le format XML Metadata Interchange (XMI). XMI est utilisé pour représenter des modèles sous la forme de documents XML. L'utilisation de XMI permet de rendre interopérables les modèles définis entre différentes applications et de faciliter les processus de transformation de modèles. Nous reviendrons sur les spécificités de XMI dans la seconde partie de cette thèse lorsque nous mettrons en application des processus de transformation.

EMF se calquant sur la sémantique d'UML, nous pouvons nous demander pourquoi EMF définit-il sont propre métamodèle. Dans la mesure où EMF est un sous-ensemble restreint et simplifié d'UML, il est alors effectivement nécessaire qu'EMF définisse un métamodèle simplifié. Par rapport à EMF, UML supporte la modélisation d'éléments ayant des points de variation sémantique. Par exemple, UML permet de définir le comportement d'une application de la même manière que la structure d'une classe. En outre, Ecore étant défini à l'origine pour simplifier la génération de code, EMF fourni un Framework permettant de générer du code à partir de modèles Ecore. Il est ainsi possible de générer à partir d'un modèle Ecore, le code Java correspondant. Pour se faire un mapping a été effectué entre le métamodèle Ecore et celui de Java.

Dans cette section, nous avons présenté l'EMF comme étant une implémentation du MOF pour la définition de modèles. L'Eclipse Modeling Framework est opérationnel pour être utilisé dans une méthodologie d'Ingénierie Dirigée par les Modèles, mais ne représente qu'un sous-ensemble de l'étendu des fonctionnalités proposées par UML. L'OMG préconise l'utilisation d'UML dans le cadre d'une approche d'Ingénierie Dirigée par les Modèles. Mais quel est exactement le rôle d'UML dans une approche IDM ? Dans la section suivante, nous présenterons les tenants et aboutissants d'UML dans un cadre IDM.

4.3.3 Le rôle d'UML dans l'IDM

L'Unified Modeling Language (UML) [Avgeriou *et al.* 2005], est le langage standard de modélisation de structures orientées objets. L'apparition des langages orientés objets ont donné naissance à différents langages de modélisation : OMT [Ebert *et al.*, 1994], Booch [Booch, 1993], OOSE [Jacobson *et al.*, 1994], etc. Chacune de ces méthodes décrit un formalisme pour construire des modèles objets. Dans les années 90, UML est né de la fusion des modèles OMT, Booch et OOSE. UML est un langage de modélisation objet de plus en plus utilisé. L'OMG a présenté UML comme étant le standard pour l'IDM, tandis que l'IDM permet d'utilisation d'autres langages de modélisation comme nous l'avons vu dans la section précédente.

UML est devenu un des standards les plus utilisés pour spécifier et documenter des systèmes d'information. Cependant, le fait qu'UML a pour but de proposer une notation générique peut limiter la modélisation de domaines particuliers pour lesquels des langages spécialisés peuvent être plus appropriés. UML fournit un ensemble de mécanismes d'extension permettant de remédier à ce problème de généricité et de s'adapter à des domaines particuliers. Dans cette section nous présenterons le mécanisme d'extension utilisé pour définir des *Profils UML*. Nous discuterons aussi de l'utilité et de la pertinence des profils UML dans le cadre de l'Ingénierie Dirigée par les modèles.

UML est un formalisme graphique pour spécifier, construire et documenter un système. UML peut donc être utilisé pour développer des systèmes dans des contextes différents et variés tels que la finance, les télécoms, l'Aeronautique, etc., et sur des implémentations différentes (Corba, J2EE, .NET, etc.). Cependant, dans certains cas, un

langage trop générique comme UML peut ne pas représenter la solution adéquate pour la modélisation d'applications liées à des domaines spécifiques. C'est le cas par exemple lorsque que la syntaxe et la sémantique des entités UML sont dans l'incapacité d'exprimer certains concepts spécifiques de systèmes particuliers ou lorsque l'on cherche à personnaliser les entités UML qui s'avèrent dans certains cas être trop générales.

Pour répondre à ces besoins de spécialisation, l'OMG spécifie deux approches possibles pour définir des modèles ayant pour particularité d'exprimer des sémantiques spécifiques à une plateforme.

La première approche consiste à définir un nouveau langage à utiliser à la place d'UML en utilisant les mécanismes fournis par l'OMG pour définir des langages de modélisation objet (c'est-à-dire en utilisant le même formalisme qui a été utilisé pour définir UML et son métamodèle). Dans une telle approche, la syntaxe et la sémantique des éléments du nouveau langage sont définies dans le but de correspondre aux spécificités de la plateforme cible.

La seconde approche est basée sur le principe de spécialisation d'UML. Dans cette approche les éléments du métamodèle UML sont spécialisés en imposant d'éventuelles nouvelles restrictions entre eux, tout en respectant le métamodèle UML et en conservant la sémantique des éléments non spécialisés. Autrement dit les propriétés des classes, associations, attributs, etc., demeurent inchangées ; de nouvelles contraintes seront simplement ajoutées aux éléments originaux. Il est aussi possible de définir au sein de profils UML de nouveaux symboles et icônes pour les éléments créés.

La première approche a été adoptée par les langages tel que le Common Warehouse Metamodel [Pool *et al.*, 2003] où la sémantique des constructeurs définis ne correspond à aucun élément du métamodèle UML. Ces nouveaux langages sont définis en utilisant le MOF.

Afin de supporter la seconde approche, UML fournit un ensemble d'extensions (stéréotypes, valeurs marquées, et contraintes) pour spécialiser ces éléments à la sémantique d'un domaine particulier. Un profil UML consistera donc à spécialiser les éléments existants du métamodèle UML.

Nous ne pouvons pas affirmer qu'une approche est meilleure que l'autre dans la mesure où elles ont chacune des avantages et des inconvénients. En effet, en définissant un nouveau langage, il est possible de spécifier une notation qui conviendra parfaitement à une plateforme particulière. Cependant, comme ces nouveaux langages ne respectent pas la sémantique d'UML, il ne sera alors pas possible d'utiliser des outils tiers existants pour créer des modèles, générer du code, faire de la génération inversée, etc. Réciproquement les profils UML, autorisant l'utilisation d'outils de modélisation existants, peuvent ne pas fournir une notation suffisamment adéquate requise par certains systèmes. Le choix de l'approche à adopter s'avère ainsi non trivial. Cependant, l'historique d'UML, le fait qu'UML soit préconisé par le MOF et la définition de ce formalisme comme étant le standard de modélisation dans le milieu du génie logiciel, sont autant d'arguments qui incitent à opter pour l'utilisation de profils UML dans un cadre d'Ingénierie Dirigée par les Modèles.

Par nature, un modèle UML ne peut pas être productif (dans le sens où cette méthode est un formalisme de modélisation et non un langage de programmation), compte tenu de la

multitude de langages existants et d'une sémantique trop générale. Il en découle le besoin de spécialiser, de profiler, UML pour des domaines précis. Un profil permet de spécialiser le formalisme UML pour un domaine particulier. De nombreux profils ont été développés pour des domaines ciblés ou des technologies particulières. Par exemple, les profils les plus répandus sont les profils CORBA et EJB. Dans le cas du profil EJB, il est par exemple possible de préciser qu'une classe au sens UML est une *EJB Session* ou un autre concept de ce domaine. Il est important de noter que les profils UML permettent la personnalisation de tout métamodèle définis à l'aide du MOF. D'une manière similaire, un profil UML peut aussi en spécialiser un autre. UML met en évidence différentes raisons pour lesquelles un développeur serait amené à spécialiser un métamodèle qu'il aurait défini :

- Pour obtenir une terminologie adaptée à une plateforme particulière ou à un domaine particulier.
- Pour définir une syntaxe pour des constructeurs qui n'auraient pas de notations dans le métamodèle existant.
- Pour définir une notation différente pour un élément déjà existant.
- Pour ajouter des concepts qui n'existent pas dans le métamodèle.
- Pour ajouter des contraintes restreignant l'utilisation du métamodèle et de ses constructeurs dans certains contextes.
- Pour ajouter des informations qui peuvent être utilisées lors de processus de transformation de modèles.

Un profil UML est défini comme un paquetage UML stéréotypé « profile » qui peut soit étendre un métamodèle ou un autre profil. Un profil UML est défini à l'aide de trois mécanismes : *stéréotypes*, *contraintes* et *valeurs marquées.*

Dans la figure 4.7, nous illustrons ces concepts par la définition d'un profil simple et minimal. Dans cet exemple, nous ajoutons deux nouveaux éléments que nous nommons *Teinture* et *Intensité.* De plus, nous associons à ces deux éléments des propriétés telles que la *couleur* pour un élément de type *Teinture* et une *intensité* numérique pour un élément de type *Intensité.* Dans cet exemple, nous restreignons l'utilisation de ces deux nouveaux éléments sur les méta-classes existantes *Class* et *Association.*

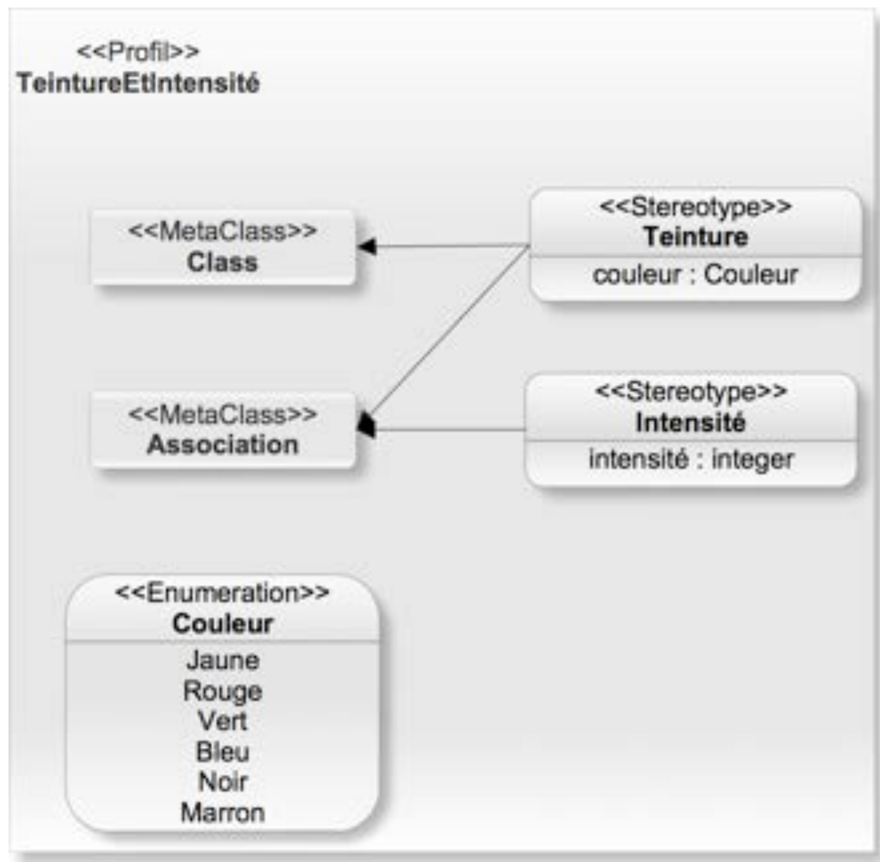


Figure 4.7. Exemple d'un profil UML.

Un stéréotype est défini par un nom et par l'ensemble des méta-classes auxquelles il est attaché. Graphiquement, les stéréotypes sont définis dans des boîtes du formalisme UML, stéréotypées « Stereotypes ». Dans notre exemple le profil UML indique que des éléments de type *class* et *association* peuvent être teintées, mais que seules les associations peuvent avoir une intensité. Les éléments du métamodèle sont spécifiés par les stéréotypes « Metaclass ». La notation d'extension est matérialisée par une flèche allant du stéréotype vers la méta-classe.

Il est possible d'ajouter des contraintes aux stéréotypes définis, imposant ainsi des restrictions aux éléments correspondants du métamodèle. Ces contraintes sont spécifiées à l'aide d'OCL [Richters, 2002]. Dans notre exemple nous spécifions qu'une association doit avoir la même couleur que les classes liées de la manière suivante :

```
context UML::InfrastructureLibrary::Core::
Constructs::Association
inv : self.isStereotyped("Coloured") implies
self.connection->forAll
(isStereotyped("Coloured")
```

```
        implies color=self.color)
```

Figure 4.8. Exemple de contrainte OCL.

Des valeurs marquées permettent de définir des méta attributs attachés à la méta classe du métamodèle étendu par un profil UML. Les valeurs marquées possèdent un nom et un type et sont associées à un stéréotype. Dans l'exemple de la figure 4.7., le stéréotype *Intensité* possède une valeur marquée *intensité* de type entier. Graphiquement, les valeurs marquées sont spécifiées comme étant des attributs de la classe définissant un stéréotype.

Un profil UML est donc un ensemble de stéréotypes, contraintes et valeurs marquées. Comme nous l'avons mentionné précédemment, le mécanisme de profil permet d'étendre la syntaxe et la sémantique des éléments. Les profils UML restent un moyen efficace lorsque l'on souhaite spécialiser la sémantique d'UML à un domaine particulier.

Différents profils ont été définis et publiquement disponibles. Certains de ces profils ont été adoptés et standardisés par l'OMG, tels que le profil pour CORBA, CCM (CORBA Component Model), EDOC (Enterprise Distributer Object Computing) et le profil EAI (Enterprise Application Integration). Cette liste est non exhaustive. De nombreux profils ont été définis et d'autres sont en cours d'approbation par l'OMG. L'intérêt de ces profils publics est de pouvoir être réutilisables par toute application de modélisation UML.

Dans le cadre d'une approche d'Ingénierie Dirigée par les Modèles, les activités les plus importantes sont la modélisation des différents aspects d'un système et la définition de transformations d'un modèle vers un autre d'une manière automatique. Nous aborderons la problématique de transformation de modèles dans la section 4.4.

Les profils UML peuvent tenir un rôle important dans la description de la plateforme cible et dans les règles de transformation entre les modèles. Si nous utilisons des profils UML pour spécifier le métamodèle d'une plateforme spécifique, cela garantira que les modèles dérivés seront en concordance avec UML. Nous pouvons affirmer que le succès d'une approche IDM réside dans l'utilisation maximum de standards.

Les mécanismes fournis par les profils UML conviennent à la description de modèles pour toute plateforme. Nous avons besoin de définir des correspondances (mappings) entre chaque élément d'un PIM, et les stéréotypes, contraintes, et valeurs marquées qui forment un profil UML. L'idée d'un profil UML dans une approche IDM est d'utiliser les stéréotypes définissant les concepts d'un PIM et de produire les éléments correspondants du PSM.

Dans cette section, nous avons présenté les profils UML comme étant un moyen efficace d'étendre le métamodèle UML pour le personnaliser à la sémantique d'une plateforme et d'un domaine spécifique. Les outils actuels de modélisation permettent la définition et l'utilisation de profils mais uniquement au niveau graphique. Cela signifie que la vérification des contraintes associées aux stéréotypes n'est pas ou très peu supportée ce qui nous amènera à proposer un mécanisme de définition de contraintes et plus précisément un formalisme de validation incrémentale que nous présenterons dans la seconde partie de cette thèse concernant nos contributions.

4.4 Transformation de modèles

Les deux principes fondamentaux de l'ingénierie des modèles sont la modélisation et la transformation de modèle. D'un point de vue général, on appelle *transformation de modèles* tout processus dont les entrées et les sorties sont des modèles. De nombreux outils, tant commerciaux que dans le monde de l'open source, sont aujourd'hui disponibles pour faire la transformation de modèles. Nous pouvons distinguer quatre catégories d'outils :

- Les outils de transformation génériques qui peuvent être utilisés pour faire de la transformation de modèles. Dans cette catégorie, se trouvent d'une part les outils de la famille XML, comme XSLT [Kay, 2001] ou Xquery [Robie, 2007], et d'autre part les outils de transformation de graphes [Guerra *et al.*, 2004]. Les premiers ont l'avantage d'être déjà largement utilisés dans le monde XML ce qui leur a permis d'atteindre un certain niveau de maturité. En revanche, l'expérience montre que ce type de langage est assez mal adapté pour des transformations de modèles complexes (c'est-à-dire allant au-delà des problématiques de transcodage syntaxique) car ils ne permettent pas de travailler au niveau de la sémantique des modèles manipulés mais simplement à celui d'un arbre couvrant le graphe de la syntaxe abstraite du modèle, ce qui impose de nombreuses « contorsions » qui rendent rapidement ce type de transformation de modèles complexe à élaborer, à valider et surtout à maintenir sur de longues périodes.
- Les ateliers de génie logiciel (AGL) intégrant des outils de transformation. Dans cette catégorie, on trouve une famille d'outils de transformation de modèles proposés par des industriels spécialisés dans le développement d'ateliers de génie logiciel. Par exemple, l'outil Arcstyler [OMG, 2002a] de Interactive Objects propose la notion de MDA-Cartridge qui encapsule une transformation de modèles écrite en JPython (langage de script construit à l'aide de Python et de Java). Ces MDA-Cartridges peuvent être configurées et combinées avant d'être exécutées par un interpréteur dédié. L'outil propose aussi une interface graphique pour définir de nouvelles MDA-Cartridges. Dans cette catégorie on trouve l'outil Objecteering [Softeam, 2009], qui propose un langage de script pour la transformation de modèles appelé J, et bien d'autres encore, y compris dans le monde de l'open source avec des outils comme Fujaba [Geiger *et al.*, 2006]. L'intérêt de cette catégorie d'outils de transformation de modèles est, d'une part, leur relative maturité et, d'autres part, leur excellente intégration dans l'atelier de génie logiciel qui les héberge. Leur principal inconvénient est le revers de la médaille de cette intégration poussée : il s'agit la plupart du temps de langages de transformation de modèles propriétaires sur lesquels il peut être risqué de miser sur le long terme. De plus, historiquement, ces langages de transformation de modèles ont été conçus comme des ajouts au départ marginaux à l'atelier de génie logiciel qui les héberge. Même s'ils ont aujourd'hui pris de l'importance, ils ne sont toujours pas vus comme les outils centraux qu'ils devraient être dans une véritable ingénierie dirigée par les modèles. En outre ces langages

montrent leur limitation lorsque les transformations de modèles deviennent complexes et qu'il faut les gérer, les faire évoluer et les maintenir [Giese *et al.*, 2009].

- Les outils conçus spécifiquement pour faire de la transformation de modèles et prévus pour être plus ou moins intégrables dans les environnements de développement normalisés. Ce sont des outils conçus spécifiquement pour faire de la transformation de modèles et prévus pour être plus ou moins intégrables dans les environnements de développement standards. Nous trouvons par exemple Mia Transformation [Mia, 2009], c'est un outil qui exécute des transformations de modèles prenant en charge différents formats d'entrée et de sortie (XMI, tout autre format de fichiers, API, dépôt). Les transformations sont exprimées comme des règles d'inférence semi-déclaratives (dans le sens où il n'y a pas de mécanisme de type de programmation logique), qui peuvent être enrichies en utilisant des scripts Java pour des services additionnels tels que la manipulation de chaînes. Dans le monde académique, on trouve de nombreux projets open source s'inscrivant dans cette approche : les outils ATL [Bézivin *et al.*, 2008] et MTL [Silaghi *et al.*, 2005], AndroMDA [Bohlen, 2007], ou encore UMT-QVT [UMT-QVT, 2005].
- Les outils de métamodélisation dans lesquels la transformation de modèles revient à l'exécution d'une application tels que MetaEdit+ [Tolvanen *et al.*, 2003]. Celui-ci permet de définir explicitement un métamodèle et, au même niveau, de programmer tous les outils nécessaires allant des éditeurs graphiques aux générateurs de code en passant par des transformations de modèles. Dans une catégorie similaire, l'outil MetaGen [Revault *et al.*, 1995] propose un environnement pour l'édition de métamodèles et un langage à base de règles pour exprimer des transformations de modèles. Les métamodèles sont compilés vers des classes Smalltalk qui peuvent être instanciées par un éditeur de modèles génériques. Plus récemment, l'environnement XMF-Mosaic [XMF Mosaic, 2007] de Xactium a été conçu comme un environnement complet pour la définition de langage. Au cœur de XMF-Mosaic se trouve un noyau exécutable de définition de langage (qui est d'ailleurs auto définissant). Tout est modélisé sur cette base, que ce soient les langages (tel que UML), les outils, les GUI, parseur et autres analyseurs XML, mais, à la différence de l'environnement MetaEdit+, obtenu par génération plutôt que par instanciation. Le langage de métamodélisation de base est un langage orienté objet qui a toute la puissance d'un langage de programmation complet ce qui en fait un outil particulièrement puissant pour la transformation.

4.4.1 Principes de la transformation de modèles

Le processus de transformation est composé de trois étapes :

- La définition des règles de transformation,
- L'expression des règles de transformation,
- L'exécution des règles de transformation.

Les deux dernières étapes définissent un système de transformation. Nous détaillons dans la suite les trois étapes une par une.

4.4.1.1 Définition des règles de transformation :

Etant donné un modèle source dans un langage *L1*, (tel que UML) et un modèle cible dans un langage *L2* (tel que Java), il s'agit dans cette étape d'élaborer une mise en correspondance des concepts de *L1* à ceux de *L2* (ex. une classe UML correspond à une ou plusieurs classes Java). Dès lors, on a recours à la technique de métamodélisation pour mettre en place une base de règles exhaustive et générique.

En outre, on peut définir un métamodèle de transformation permettant de définir un langage abstrait de transformations. Les modèles instances de ce métamodèle sont des spécifications de transformations entre deux métamodèles spécifiques. De cette manière, la spécification de transformation devient lisible (comprendre un modèle est plus facile que de comprendre du code), et réutilisable (le métamodèle représente les règles de transformation de manière abstraite indépendante du langage de sa mise en œuvre).

Les règles de transformation sont établies entre le métamodèle source et le métamodèle cible, c'est-à-dire entre l'ensemble des concepts du modèle source et celui du modèle cible. Le processus de transformation prend en entrée un ou plusieurs modèles conformes à des méta-modèles source et produit en sortie un ou plusieurs autres modèles conformes à un ou plusieurs métamodèles cibles, en utilisant les règles préalablement établies.

4.4.1.2 Expression des règles de transformation :

Pour exprimer les règles de transformation, un langage de spécification de règles est nécessaire. Actuellement, il n'existe pas de langage standard.

En 2002, l'OMG a émis un appel à proposition (RFP – Request For Proposal) pour la standardisation du processus de transformation. Ceci va produire le standard MOF/QVT (Queries/Views/Transformations) qui est une technique de transformation dont la syntaxe doit être définie en un métamodèle MOF. Il doit permettre :

- La correspondance entre les modèles définis en MOF.
- L'interrogation d'un modèle MOF afin de filtrer et sélectionner des éléments du modèle source à transformer.
- La création des vues sur des méta-modèles MOF, une vue sur un système modélisé est un modèle dérivé du modèle du système, ne révèle que certains aspects de ce dernier.

Un langage de transformation peut être déclaratif, impératif ou hybride. Dans la programmation déclarative, on décrit d'une part les données du problème à traiter et d'autre part les contraintes sur ces données. Le programme s'exécute à partir de la situation courante décrite par les données tout en respectant les contraintes. Le langage déclaratif

décrit ce qu'on devrait avoir à l'issue d'un certain nombre de données initiales. XSLT (Extensible Stylesheet Language Transformation) [W3C, 1999], est un exemple de langage déclaratif de transformation. Par opposition à un programme déclaratif, un programme impératif décrit comment le résultat devrait être obtenu en imposant une suite d'actions que la machine doit effectuer. Un langage hybride regroupe à la fois les paradigmes de programmation déclarative et impérative : l'ordre d'exécution des modules doit être spécifié tandis qu'au sein d'un même module, la détermination de l'ordre d'exécution des règles n'est pas à la charge de l'utilisateur.

4.4.1.3 Exécution des règles de transformation

Une fois spécifiées et exprimées, les règles requièrent un moteur d'exécution pour être exécutées. Ce moteur prend comme entrée le modèle et le métamodèle source, le métamodèle cible, ainsi que le modèle de transformation (les règles de transformation écrites dans le langage de transformation, basées sur les correspondances entre les deux méta-modèles source et cible) et son métamodèle (représentant la grammaire du langage de transformation). Il produit en sortie le modèle cible. La figure 4.9 illustre le processus de transformation de modèles.

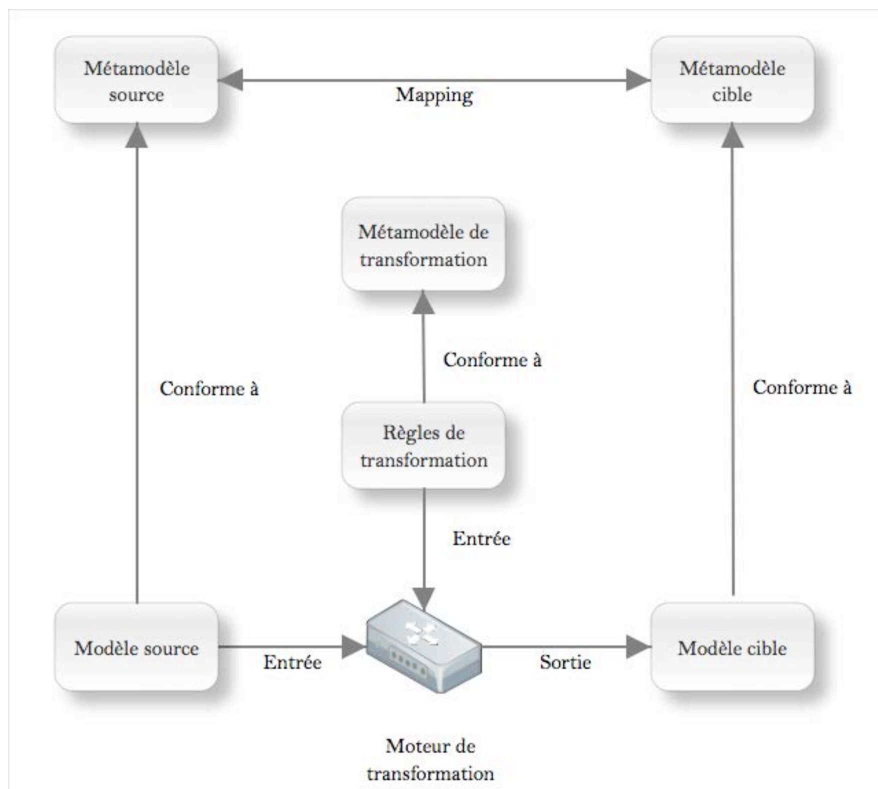


Figure 4.9. Exécution des règles de transformation.

La transformation de modèles peut avoir les caractéristiques suivantes :

- Une transformation est dite inversible ou sans perte si la transformation inverse existe.
- Une transformation est dite d'optimisation si elle transforme un modèle en un modèle équivalent optimisé selon une métrique donnée. Un exemple d'optimisation est l'amélioration de la réutilisabilité ou encore la lisibilité.
- Une transformation peut être caractérisée par le niveau d'abstraction des métamodèles source et cible. Plus on est proche de la plateforme d'exécution, moins on est abstrait, et inversement.
- Une transformation peut être caractérisée par sa position dans un processus de développement logiciel. Elle va donc être caractérisée par les acteurs qui l'appliquent et la stratégie de son application.
- Une transformation peut être également caractérisée par son langage d'expression. De plus, elle peut être totalement ou partiellement automatisée.

En outre, une approche de transformation de modèles est caractérisée par :

- les règles de transformation.
- la portée de l'application des règles qui est soit la source soit la cible.
- la relation entre le modèle source et le modèle cible qui précise si la transformation a comme résultat un autre modèle cible, ou un autre modèle mettant à jour le modèle source, ou encore le modèle source modifié.
- la stratégie de l'application des règles détermine la suite des éléments du modèle source sur lesquels les règles vont être exécutées.
- l'ordonnancement des règles détermine l'ordre d'exécution des règles.
- l'organisation des règles spécifie la structure des règles (orientées source, orientées cibles, indépendantes) ainsi que les relations de composition entre elles.
- la traçabilité précise la manière de garder la trace de la transformation. Les traces peuvent servir aux analyses, au débogage, à la synchronisation entre modèles, etc.
- Le sens de la transformation spécifie si les règles sont uni ou bi directionnelles.

4.4.2 Les langages de transformation

L'Ingénierie Dirigée par les Modèles se focalise sur les spécifications du MOF pour définir des langages de transformation de modèles. Dans cette section, nous présentons de façon non exhaustive les langages de transformation qui ont été définis à partir des recommandations du MOF. Nous présenterons les trois langages de transformation standardisés et basés sur le MOF :

- MOF Query/View/Transformation (QVT), le langage de transformation standard de modèles respectant le MOF.

- ATLAS Transformation Language (ATL), un langage de transformation calqué sur QVT, implémenté en Java et pouvant manipuler des modèles EMF.
- MOFScript, basé sur les spécifications de QVT, transforme des modèles en fichiers texte.

4.4.2.1 MOF Query/View/Transformation

Query/View/Transformation (QVT) [Kurtev, 2008] est le langage de transformation standard pour les modèles basés sur une architecture MOF. MOF QVT est considéré comme étant une partie des spécifications du MOF. QVT est basé sur une architecture de langages déclaratifs et procéduraux permettant une manipulation facilitée des éléments issus de modèles ou de métamodèles et la définition de règles de transformation sur ces éléments. L'architecture proposée par QVT est présentée par la figure 4.10 :

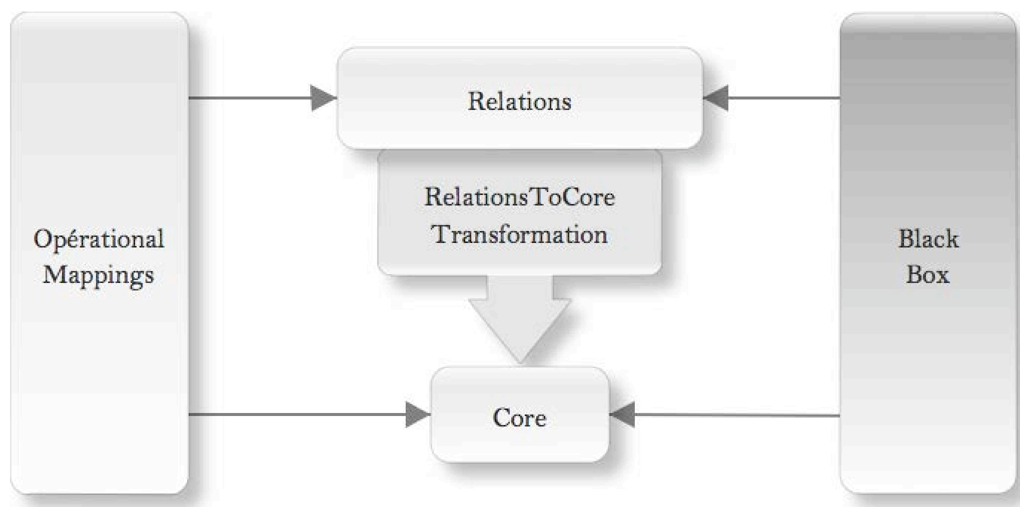


Figure 4.10. Architecture QVT des langages de spécification de modèles.

Parmi les langages déclaratifs, QVT propose un langage de relations (*Relations*) et un langage noyau (*Core*) permettant de spécifier des correspondances sur les éléments des modèles manipulés. Pour les langages procéduraux, QVT définit un langage d'opérations de correspondances (*Operational Mappings*) et un langage d'implémentation interne d'opérations (*Black box*). Ces langages procéduraux doivent donner accès aux relations et éventuellement aux opérations des modèles manipulés.

Le langage de relations de QVT est utilisé pour définir de manière déclarative des relations existantes entre les éléments d'un modèle source et les éléments d'un modèle cible suivant certaines contraintes. La vérification d'une relation et de ses contraintes conduira à la transformation de l'élément source en élément cible. Il est nécessaire que ce langage soit déclaratif afin que les diverses transformations nécessaires puissent se faire d'une manière parallèle. Par exemple, la transformation d'une classe UML fait appel non seulement à la

relation qui concerne cette classe, mais également à la relation qui concerne les éléments englobés par la classe selon le métamodèle (attributs et opérations).

Le langage des opérations doit être réflexif. Nous devons notamment pouvoir définir un ensemble de relations permettant de passer du langage de relations vers le langage noyau. Cet ensemble de relations est représenté par l'élément « RelationsToCore » de la figure 4.10.

Le langage noyau est un langage de bas niveau possédant une syntaxe assez simple, qui permet d'opérer un filtrage sur des ensembles de variables en évaluant des conditions issues des modèles traités. Le langage noyau est implémenté dans une sorte de machine virtuelle.

Le langage d'opérations de correspondances produit les mêmes effets que le langage de relations. Il s'agit d'une enveloppe procédurale du langage déclaratif des relations, permettant en particulier d'assurer une certaine facilité d'utilisation aux développeurs peu habitués aux langages déclaratifs. Des opérations peuvent être utilisées pour décrire des relations complexes ou encore pour décrire une transformation entière procédurale. Ce niveau de langage fait appel au langage OCL.

QVT à pour principal avantage de reposer sur les standards existants à savoir le MOF et OCL, permettant ainsi de développer des outils qui seront d'une part simples à utiliser pour un grand nombre d'utilisateurs, d'autre part compatibles avec un grand nombre d'outils de modélisation existants.

4.4.2.2 ATLAS Transformation Language (ATL)

ATL [Bézivin *et al.*, 2008] est un langage de transformations de modèles dans le contexte de l'Ingénierie Dirigée par les Modèles. La première réalisation d'ATL est basée sur MetaData Repository (MDR) [Hnetynka *et al.*, 2004] et Java, tandis que la deuxième est basée sur MDR, EMF, Java et Eclipse [Eclipse, 2009].

Une requête ATL est une expression en OCL qui peut retourner des types primitifs, des éléments d'un modèle, des collections, des n-uplets, ou une combinaison de ces types (par exemple, collections de n-uplets). La version actuelle d'ATL ne supporte ni la transformation incrémentale, ni la bidirectionnalité.

Cependant, les concepteurs d'ATL préconisent l'utilisation du support de traçabilité pour réaliser cette caractéristique en ATL. La transformation en ATL est unidirectionnelle. ATL est un langage hybride (déclaratif et impératif). L'approche déclarative d'ATL est basée sur OCL. L'approche impérative en ATL contient des instructions qui explicitent les étapes d'exécution dans une procédure (*helpers*). La figure 4.11 présente un extrait du métamodèle des règles de transformation ATL. Selon la syntaxe abstraite, une règle est définie par un nom et peut contenir les éléments *ActionBlock* et *OutPattern*. Une *MatchedRule* spécialise une *Rule* et contient un *InPattern*. Une *CalledRule* spécialise une *Rule* ou une *Operation* en OCL.

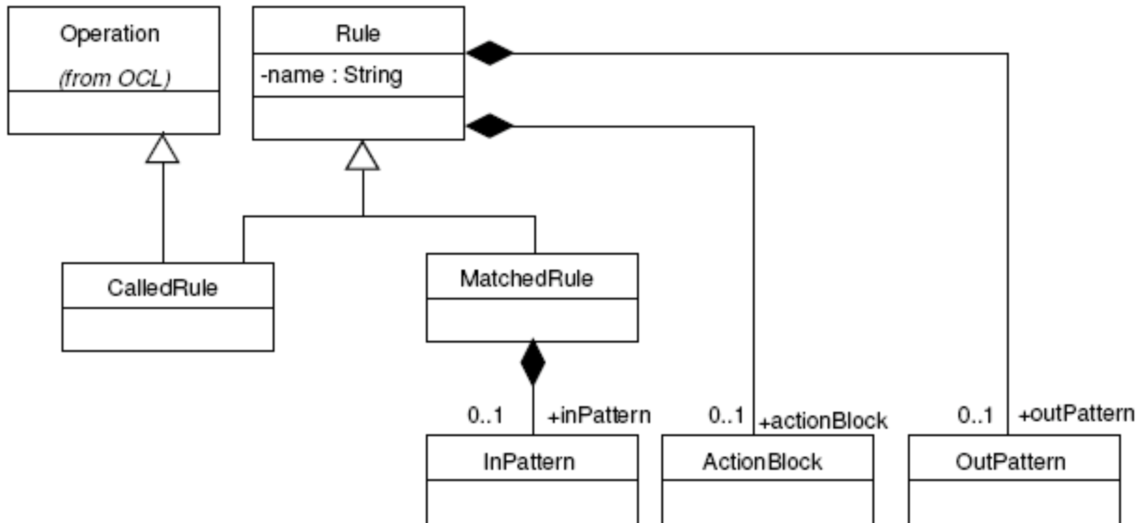


Figure 4.11. Extrait du métamodèle des règles ATL.

Une règle est explicitement appelée en utilisant son nom et ses paramètres (une *CalledRule*) ou exécutée comme un résultat de la reconnaissance d'un *inPattern* dans le modèle source (une *MatchedRule*). Le résultat de l'exécution d'une règle peut être déclaré en utilisant un *outPattern*, de manière déclarative, impérative ou les deux.

Une règle formée d'un *inPattern*, d'un *outPattern* et sans section impérative est définie comme une règle déclarative. Une règle formée avec un nom, des paramètres et une section impérative, mais sans un *outPattern* est définie comme une règle impérative. La combinaison des deux formes est simplement appelée règle hybride.

La Figure 4.12 présente les détails de la relation entre un *InPattern* et un *OutPattern*. L'élément *MatchedRule* spécifie un patron source (*inPattern*) comme un ensemble de types d'origine du métamodèle source associé à l'ensemble des noms de variables et optionnellement filtré en utilisant une expression logique en OCL. Un patron cible (*outPattern*) est un ensemble de types d'origine du métamodèle cible associé à des noms de variables et des liaisons (*bindings*). Quand une règle contenant un patron cible est exécutée, les éléments cibles sont créés. Une liaison spécifie la valeur utilisée pour initialiser une propriété d'une instance. En fait, un patron cible est un ensemble de noeuds qui peuvent être liés par des liaisons.

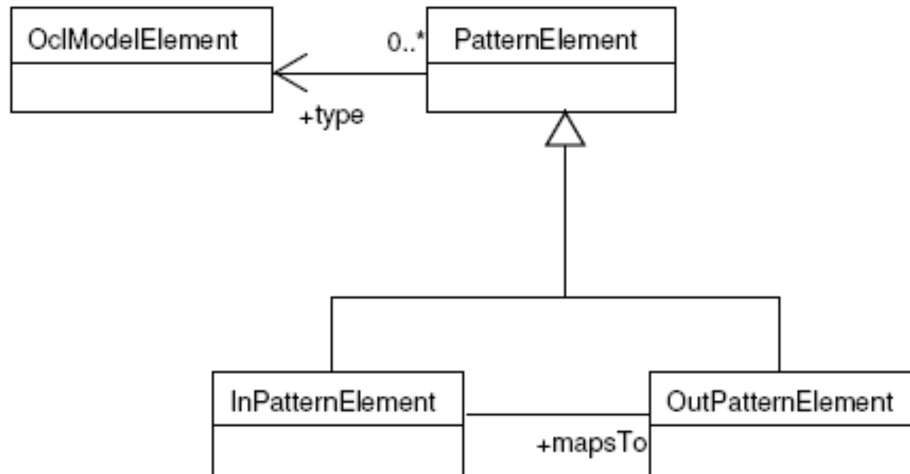


Figure 4.12. Patrons d'éléments (syntaxe abstraite).

Les constructions déclaratives ATL sont constituées d'un *InPattern* représenté par le mot clé *from*, d'un *outPattern* représenté par le mot clé *to*, et d'un ensemble de liaisons représentées par le symbole <-.

Les instructions impératives ATL peuvent être regroupées en :

- Expressions : basées sur OCL.
- Variables : Une déclaration d'une variable est faite par le mot clé *let*. Par exemple : *let* VarName : varType = initialValue ;
- Attribution : L'opérateur d'affectation <- peut être utilisé à cette fin. De plus, ATL fournit l'opérateur := pouvant être utilisé quand une résolution automatique n'est pas nécessaire. Ceci permet à l'opérande de gauche de prendre la valeur de celle de droite.
- Manipulation d'instances : Les instances peuvent être créées de manière implicite grâce à l'approche déclarative. De plus, il est possible d'expliciter la création ou la destruction d'une instance en utilisant les opérateurs *new* et *delete*.
- Déclarations conditionnelles *if*, *else*.
- Déclaration de boucles : ATL permet la déclaration de boucles *while*, *do... while*(condition) et *foreach*.

De plus, ATL fournit le mécanisme des *helpers* pour éviter la redondance de code et la création de grandes expressions OCL dans une règle. Ceci induit aussi une meilleure lisibilité des programmes ATL.

Un *helper* en ATL est une fonction qui peut recevoir des paramètres et retourner une valeur ou une instance d'un élément. Les *helpers* sont toujours utilisés dans le contexte d'une transformation ou pour d'autres *helpers*.

Selon la taxonomie créée par [Czarnecki *et al.*, 2003], ATL a les caractéristiques suivantes :

- Règles de transformation : ATL fait bien la distinction entre LHS (Left Hand Side), et RHS (Right Hand Side). Une règle en ATL est une combinaison de variables syntaxiquement typées et d'une logique exécutable basée sur OCL.
- Stratégie d'application des règles : en ATL, la stratégie est déterministe.
- Organisation de règles : ATL supporte la modularité.
- Traçabilité : ATL supporte la traçabilité par le biais de l'enregistrement des étapes de transformation réalisées par le moteur de transformation.
- Sens des transformations : une transformation en ATL est unidirectionnelle.

Par rapport à la classification générale, ATL est un langage permettant d'effectuer la transformation d'un modèle en un autre modèle avec les caractéristiques suivantes :

- ATL étant en principe déclaratif, il est aussi relationnel (les contraintes de ces relations sont spécifiées en OCL).
- ATL n'est pas directement basé sur la théorie des graphes.
- ATL contient des caractéristiques déclaratives et impératives.

4.4.2.3 MOFScript

Les langages de transformation tels que QVT et Atlas sont essentiellement focalisés sur des transformations de type « modèle à modèle ». Cette catégorie de transformation consiste à prendre en entrée un modèle source et des règles de transformation et de produire un modèle cible. Un besoin naturel est apparu par la suite, celui de pouvoir simplement générer un modèle productif à partir d'un modèle abstrait ; autrement dit, il s'agit de générer du code à partir d'un modèle et d'effectuer des transformations de type « modèle à texte ». Dans cette optique, l'OMG a proposé MOFScript comme Framework de transformation de modèles MOF vers du code.

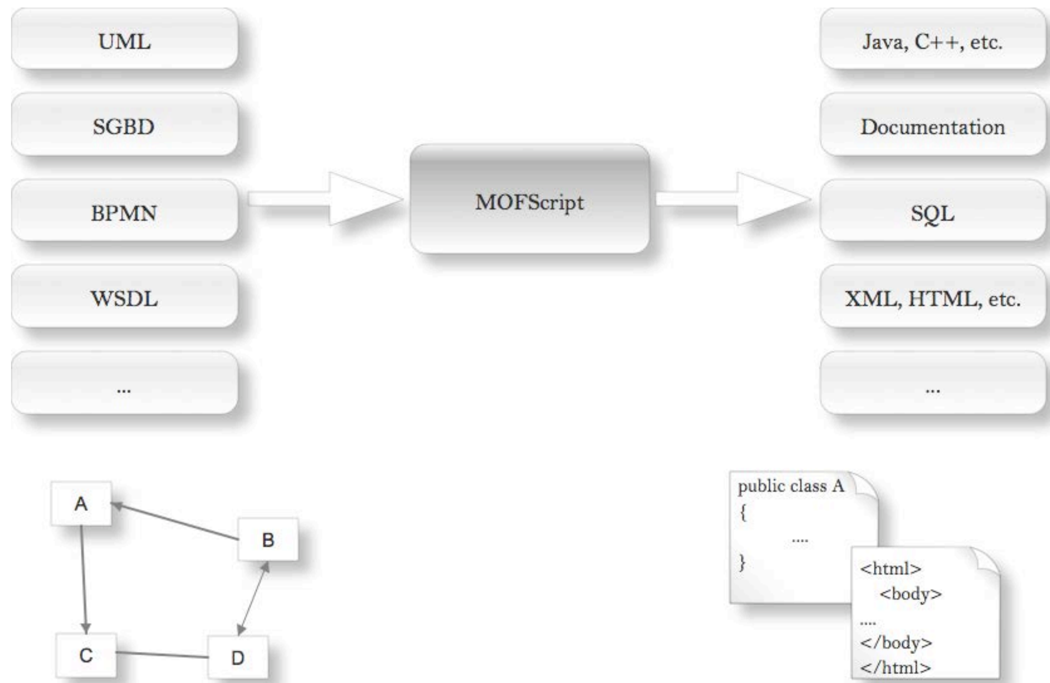


Figure 4.13. Transformation de modèle avec MOFScript.

La figure 4.13 présente le processus de transformation à l'aide du Framework MOFScript. Dans ce processus un modèle MOF est défini en entrée et un document texte est défini en sortie. La boîte MOFScript représente l'outil d'exécution des règles de transformation définies à l'aide du langage MOFScript. En utilisant MOFScript, il est aussi possible de spécifier plusieurs modèles MOF en entrée et de générer différents documents texte en sortie.

MOFScript est en parfait accord avec les spécifications du MOF et de son architecture quatre couches. La figure 4.14 montre la relation de MOFScript avec les différents niveaux de l'architecture MOF :

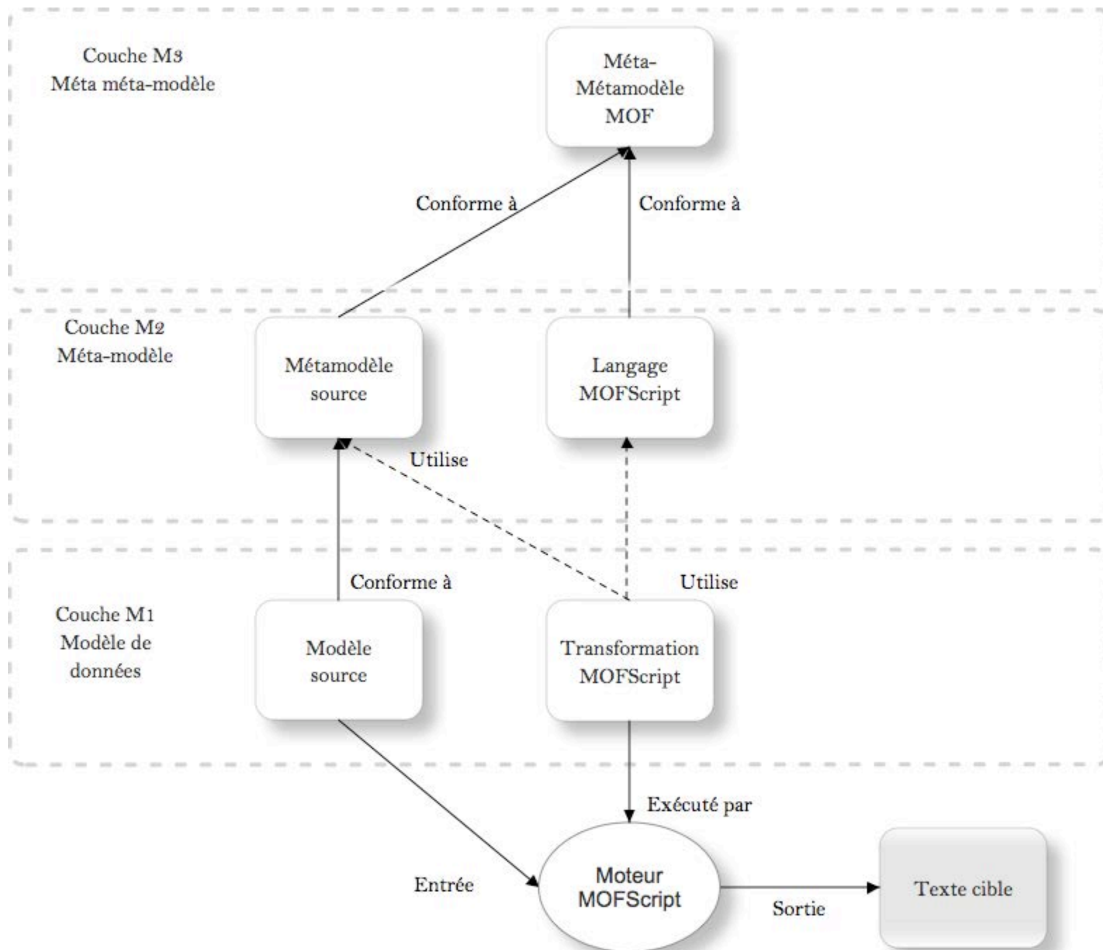


Figure 4.14. MOFScript et l'architecture quatre couches du MOF.

Dans cette figure, nous pouvons voir que la syntaxe abstraite, c'est-à-dire son métamodèle, est conforme au MOF (niveau M2). Une transformation MOFScript (niveau M1) est écrite en conformité à ce métamodèle. Les transformations sont exécutées par le Framework MOFScript sur un modèle MOF source. Nous pouvons noter que dans cette architecture le texte cible n'a pas de métamodèle défini explicitement ou n'a pas de correspondance dans la hiérarchie du MOF. En effet, le document texte cible ne représente pas obligatoirement un modèle MOF, ce document texte peut représenter du code JAVA, HTML, XML, XML Schema, etc.

Lors de la conception de MOFScript, trois besoins ont été identifiés :

- MOFScript devait définir un langage ressemblant fortement aux langages de programmation les plus fréquemment utilisés tel que JAVA.
- MOFScript étant fondé sur les spécificités du MOF, son implémentation doit être alignée sur celle de QVT.

- MOFScript définissant un langage, toutes les méthodes définies doivent être appelées explicitement, exceptée la méthode principale (méthode *main*()) d'un script MOFScript.

La figure 4.15 met en évidence la relation entre MOFScript et le langage QVT :

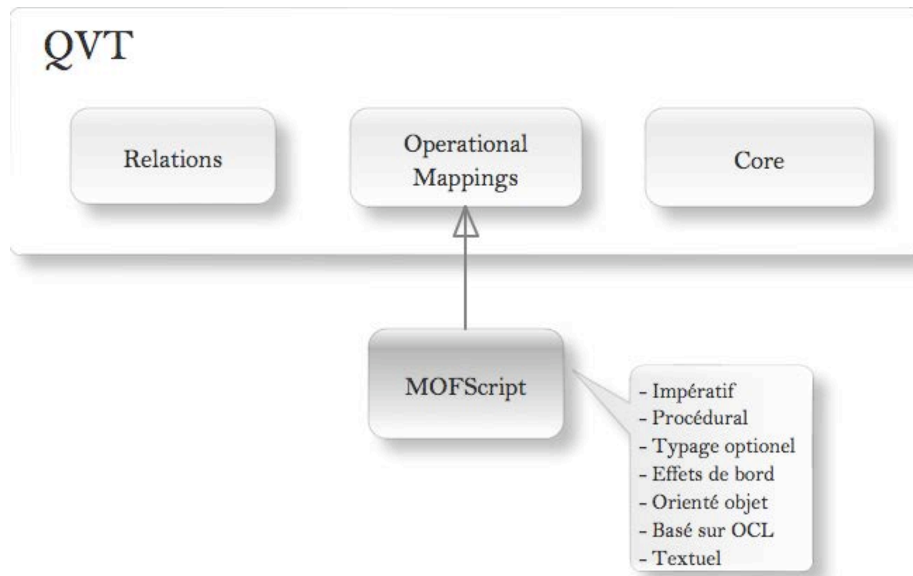


Figure 4.15. MOFScript et QVT.

MOFScript est une amélioration du module *« Operational Mappings »* que nous avons présenté dans la section précédente. Un ensemble de transformations définies dans un document MOFScript est une spécialisation des transformations QVT et peut contenir uniquement des règles définies à l'aide du langage MOFScript. Au niveau du métamodèle (M2), les règles MOFScript spécialisent les règles QVT. Tandis que QVT peut être utilisé de trois manières différentes (requêtes, vues et transformations), MOFScript est uniquement utilisé pour réaliser des transformations.

La figure 4.16 présente les relations entre QVT et MOFScript.

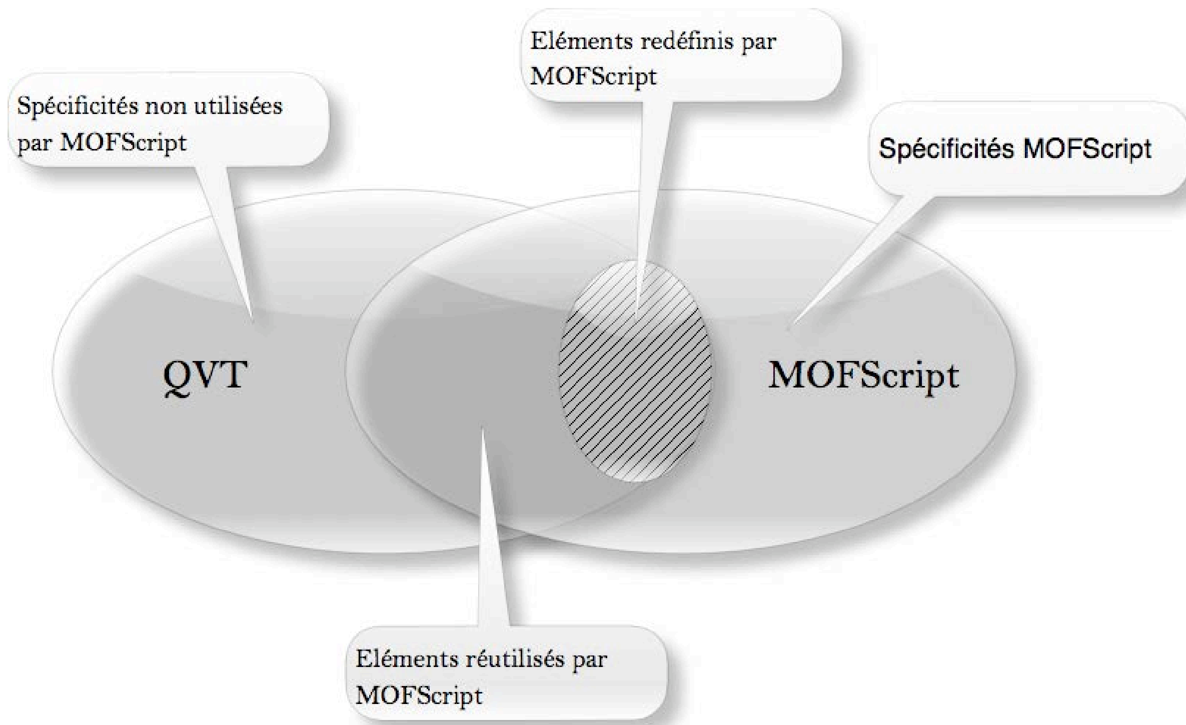


Figure 4.16. Spécialisation de QVT par MOFScript.

Bien que la spécification abstraite prévoit que MOFScript hérite de QVT et de OCL, l'implémentation et la syntaxe concrète de MOFScript trahissent cette relation d'héritage dans la mesure où certains éléments de QVT et d'OCL ne sont pas implémentés et d'autres sont redéfinis. Par exemple :

- Non réutilisés dans MOFScript : relations, noyau (*Core*), etc.
- Réutilisés par MOFScript : certaines parties des règles de correspondance (*Operational Mappings*), méthode *main()*, expressions OCL.
- Changement de notation : le mot clé *mapping* est extrait du module *rules de* QVT, *forEach*() remplace l'expression *forAll*(), etc.
- Parmi les nouveautés de MOFScript nous pouvons noter les éléments suivants :
- Bibliothèque sur les chaînes de caractères, gestion des espaces dans les chaînes de caractères (notion de *white space*), gestion des éléments XML, etc.
- Gestion avancée des types de données. En plus des types de bases tels que *Integer*, *String*, *Real*, *List*, etc., les variables qu'elles soient globales ou locales peuvent être optionnellement typées. Les variables non typées sont typées dynamiquement à partir des expressions dans lesquelles elles sont assignées.
- Définition d'une bibliothèque de fonctions. La plupart des méthodes sur les chaînes de caractères sont directement inspirées de celles définies par Java et XSLT.

- Les objets de type collection et les boucles conditionnelles sont inspirés des collections Java y compris l'expression conditionnelle *foreach*() définie depuis Java 1.5.
- Support des opérations définies par UML 2 :
  - *getAppliedStereotypes*() : List (Stereotype)
  - *getAppliedStereotype* (String name) : Stereotype
  - *hasStereotype* (String name) : Boolean
  - *hasValue* (StereoType st, String valueName): Object | *hasValue* (String stName, String valueName): Object
  - *getValue* (StereoType st, String valueName):Object | *getValue* (String stName, String valueName): Object

La figure 4.17. présente une mise en application de MOFScript sur un modèle UML respectant les spécifications du MOF.

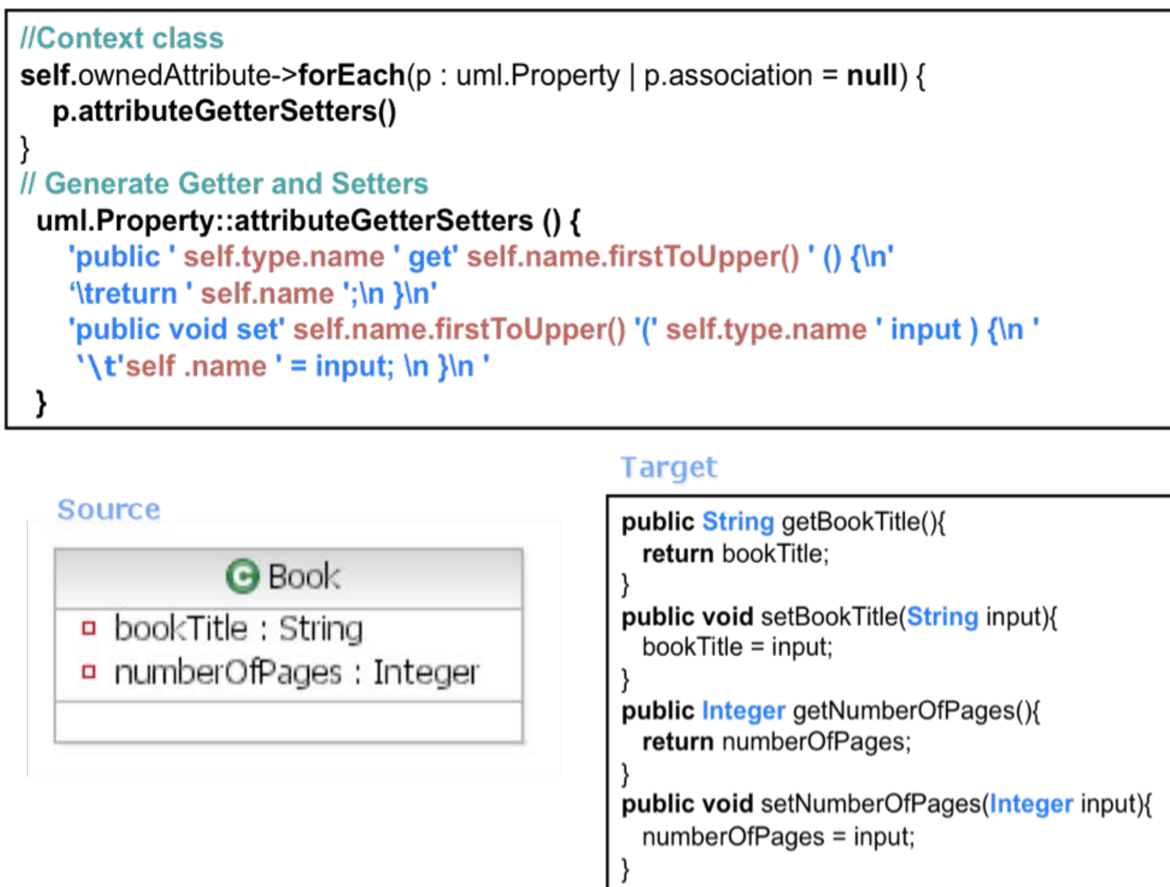


Figure 4.17. Exemple de transformation MOFScript.

Cet exemple met en avant la transformation d'une classe UML vers une classe JAVA. Le script de transformation prend en entrée le modèle UML, ici composé d'une classe

« *Book* » et produit la classe JAVA correspondante à partir des propriétés, opérations et attributs définis dans le modèle UML.

MOFScript a pour principaux avantages de proposer un framework performant de génération de code à partir de modèles MOF et de reposer sur les standards existants à savoir QVT et OCL. La richesse du langage MOFScript permet de manipuler tout modèle MOF et de générer du code dans des formalismes hétérogènes. Dans nos travaux il est question d'être capable à partir d'un modèle UML de générer de manière automatique un modèle XML Schema, ou autrement dit de transformer un modèle basé sur une architecture MOF vers un modèle textuel. Parmi les formalismes présentés dans ce chapitre, nous avons choisi d'utiliser le langage MOFScript. Notre choix a été motivé par le fait que les langages de transformation tels que QVT sont essentiellement focalisés sur des transformations de type « modèle à modèle » et que nous avons simplement besoin de générer un modèle textuel à partir d'un modèle abstrait ; autrement dit, il s'agit de générer du code à partir d'un modèle.

4.5 Conclusion

Dans ce chapitre nous avons présenté l'Ingénierie Dirigée par les Modèles (IDM). Nous avons positionné l'IDM comme étant une approche du développement logiciel qui vise à garantir la prise en compte des retours dans un cycle de développement du niveau du code vers le niveau d'analyse. Cela est possible grâce à des transformations automatiques de la description de plateformes abstraites et indépendantes (PIM) vers la description de plateformes concrètes et spécifique (PSM). Dans l'IDM, cette notion de description est englobée par la notion de modèles. Nous avons présenté le rôle et les différents types de modèles existants (section 4.2) dans le domaine de l'IDM. Nous avons aussi expliqué comment l'IDM utilise la métamodélisation (section 4.3) et les technologies de transformation de modèles (section 4.4) pour parvenir à la transition entre les différents niveaux d'abstraction.

Dans la suite de ce mémoire nous mettrons en application une méthodologie d'Ingénierie Dirigée par les Modèles dans le contexte du Master Data Management. L'application d'une telle méthodologie sera effectuée en trois étapes :

(i) Utilisation du formalisme UML et spécialisation de celui-ci par la définition de profils afin de modéliser un Master Data (chapitre 5).

(ii) Définition de règles de transformation pour passer d'un modèle abstrait vers un modèle spécifique de notre solution MDM EBX.Platform (chapitre 6).

(iii) Définition d'une approche de validation incrémentale de modèle afin d'optimiser les processus de validation lors de la définition et de la maintenance de modèles (chapitre 7).

Chapitre 5

Vers une représentation abstraite de modèles de données

Résumé. Dans le chapitre précédent nous avons positionné l'Ingénierie Dirigée par les Modèles comme étant une solution permettant de s'abstraire des spécificités techniques liées à une plateforme donnée. Cela est possible grâce à l'utilisation d'un standard de représentation de métamodèles (Meta Object Facility) et d'un formalisme de représentation abstrait de modèles de données (UML). Dans ce chapitre nous mettrons en application une métamodélisation UML, un des principes fondamentaux de l'Ingénierie Dirigée par les Modèles, appliquée à la définition de modèles XML Schema et au domaine du Master Data Management.

5.1 Introduction

L'introduction d'une approche d'Ingénierie Dirigée par les Modèles (IDM) est un point clé dans l'adoption d'une vision unifiée permettant de concevoir des applications en séparant la logique métier de toute plateforme technique. La mise en place d'une couche d'abstraction entre la logique métier et les plateformes techniques est assurée par la définition de métamodèles et l'utilisation d'un formalisme standard abstrait de définition de modèles. Dans le chapitre précédent, nous avons vu qu'UML représente un formalisme abstrait adéquat permettant de représenter des modèles indépendamment d'une plateforme donnée. Il fournit les fondements pour spécifier, construire, visualiser et décrire les artefacts d'un système logiciel. Afin d'assurer cela, UML se base sur une notation graphique dont la syntaxe est à la fois simple, intuitive et expressive.

UML est un langage de modélisation à destination d'un grand nombre de domaines d'application. Cependant, par rapport au formalisme UML standard, les développeurs souhaitent souvent rajouter des caractéristiques pour tenir compte de la spécificité de leur domaine d'application. Afin de satisfaire ce besoin, UML est doté d'un mécanisme d'extensibilité fondé sur des stéréotypes, des contraintes et des valeurs étiquetées. Un tel mécanisme permet de personnaliser le métamodèle UML pour qu'il prenne en considération les besoins de modélisation spécifiques. Le résultat de cette personnalisation est un *profil*

UML. La notion de *profil* UML a été introduite dans le standard UML 1.3 comme un moyen permettant la structuration des extensions UML (stéréotypes, contraintes et valeurs étiquetées).

Un profil UML peut être décrit simplement comme un ensemble d'éléments de modélisation ajoutés au métamodèle UML. De plus, il existe dans UML la notion de règles. Ces règles permettent de décrire et d'automatiser un savoir-faire. De cette façon, les profils UML constituent un moyen efficace pour spécifier et guider le processus de développement UML. Durant le développement, à chaque phase, les profils permettent d'exprimer comment utiliser UML, quels sont les produits de développement attendus, et quelles règles le modèle doit respecter. En reprenant cette approche, un profil UML peut être décrit par :

- les éléments UML utilisés (éléments d'UML pertinents pour un domaine donné),
- les extensions UML ajoutées (stéréotypes, contraintes, valeurs étiquetées),
- les règles de validation (règles vérifiant des critères de cohérence sur un modèle pour un profil donné),
- les règles de présentation (les diagrammes UML doivent présenter certaines informations et en cacher d'autres),
- les règles de transformation (règles de génération de code et patrons permettant d'assister ou d'automatiser le développement).

Plusieurs profils UML ont été standardisés ou sont en cours de standardisation tels que SPEM (modélisation des procédés logiciels) [SPEM, 2008], EDOC (modélisation des applications distribuées) [EDOC, 2004], MARTE (modélisation pour des analyses quantitatives en temps réel) [OMG, 2006], Performance and Time (modélisation des systèmes embarqués) [PST, 2005], UML pour CORBA [Corba, 2002], UML pour les EJB [Greenfield, 2001], etc.

En effet, pour faciliter la définition et la formalisation d'UML, les différents concepts d'UML sont modélisés eux-mêmes en UML. Cette définition récursive, appelée métamodélisation, décrit de manière formelle les éléments de modélisation d'UML ainsi que la syntaxe et la sémantique de la notation qui permet de les manipuler. Le métamodèle devient, entre autres, un outil de vérification qui facilite l'identification des éventuelles incohérences, notamment par l'utilisation de règles de bonne modélisation, exprimées en langage Object Constraint Language [Richters, 2002].

Dans ce cadre, la métamodélisation de modèles d'adaptation est une première étape à l'introduction d'une approche IDM. L'objectif de ce chapitre est de structurer notre approche d'Ingénierie Dirigée par les Modèles en proposant une extension du métamodèle UML sous forme de deux profils UML :

- un premier profil UML dédié à la sémantique de XML Schema (section 5.3),
- un second profil UML dédié à la sémantique des modèles d'adaptation définie dans le contexte du Master Data Management (section 5.4).

La définition de ces concepts au niveau métamodèle UML permettra de guider et de valider une représentation IDM.

Rappelons que la métamodélisation a été standardisée par l'OMG qui a préconisé l'utilisation du Meta Object Facility (MOF) pour la définition de métamodèles. La métamodélisation d'un modèle d'adaptation a pour but de représenter de manière abstraite la sémantique dédiée à la représentation d'un modèle de données pivot associé au domaine du MDM. L'OMG recommande d'utiliser les formalismes UML pour définir un métamodèle et OCL pour spécifier les contraintes entre les éléments de celui-ci. Cependant, ces deux formalismes ne sont pas suffisants et présentent un nombre limité d'entités pour représenter des modèles associés à une technologie particulière dits Platform Specific Model (PSM) [Cattel, 1994]. Pour pallier les limitations du métamodèle UML pour représenter des modèles PSM, il est possible de spécialiser le métamodèle UML en ajoutant des éléments et des contraintes. Cet affinement est possible par l'intermédiaire de profils UML, ce qui est en fait une spécialisation de la couche M2 de l'architecture MOF (figure 5.1) aux sémantiques XML Schéma et Master Data Management.

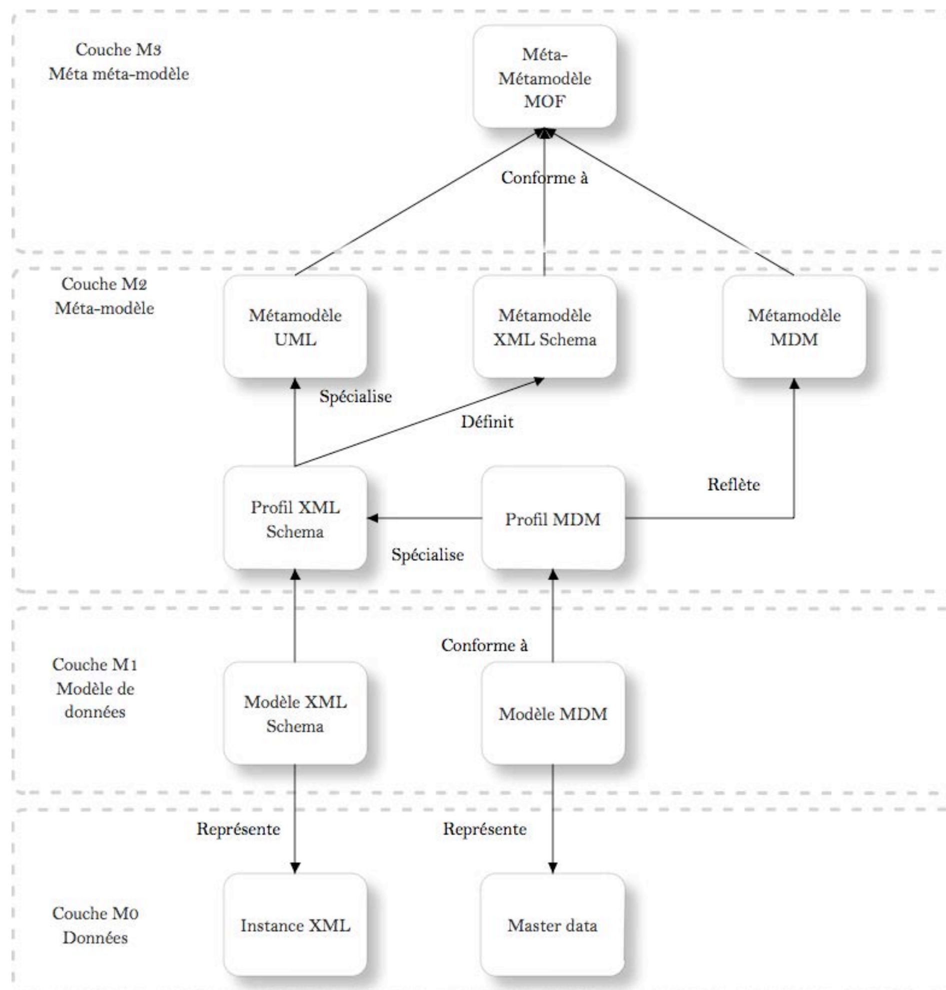


Figure 5.1. Architecture 4 couches du MOF.

Notre objectif est de faciliter et de standardiser la modélisation de modèles d'adaptation basés sur le formalisme XML Schema et dédiés au domaine du MDM. A ce jour, la modélisation graphique de modèles XML Schema n'est pas standardisée. Il existe certes des outils pour la modélisation de modèles XML mais ceux-ci sont restreints à la sémantique de XML Schema (Altova XML Spy, etc.). En effet, ces outils sont dans l'incapacité de guider l'utilisateur dans l'emploi des concepts introduits par les modèles d'adaptation, et plus généralement dans l'utilisation d'extensions spécifiques. De plus, ces outils de modélisation proposent des représentations graphiques différentes d'une solution à une autre, ce qui représente une source potentielle de confusion lors des phases de modélisation où différents acteurs peuvent intervenir. L'introduction d'un formalisme standard de définition de modèles est un moyen de rendre la modélisation plus accessible. UML est un langage de modélisation « objet » de plus en plus utilisé et reconnu aujourd'hui comme un standard dans le domaine du génie logiciel, ce qui en fait un candidat idéal pour la modélisation des modèles d'adaptation. La spécialisation du langage UML par l'intermédiaire de profils est un moyen de standardiser et de rendre générique la définition de modèles d'adaptation. Ces modèles étant à l'origine défini à l'aide de XML Schema et dédiés au domaine du Master Data Management, nous définissons dans ce chapitre deux profils UML où le premier est dédié à la sémantique de XML Schema et le second appliqué à la sémantique du Master Data Management. Dans l'objectif de faciliter et d'optimiser l'appariement entre XML Schema et UML, il est nécessaire, dans un premier temps, d'enrichir chacun de ces formalismes avec les spécificités provenant de l'autre. C'est une première étape d'homogénéisation permettant de définir un appariement entre ces deux formalismes.

5.2 Intégration de métadonnées objet dans le modèle XML Schema

UML est un formalisme de modélisation objet qui définit des notions telles que la généralisation, la composition et l'agrégation. Bien qu'il soit possible de matérialiser ces notions de manière implicite dans un modèle XML Schema, nous proposons d'introduire des métadonnées matérialisant ces notions de manière explicite dans les modèles XML Schema.

La Figure 5.3 illustre l'utilisation de ces concepts par l'intermédiaire d'un diagramme de classes UML :

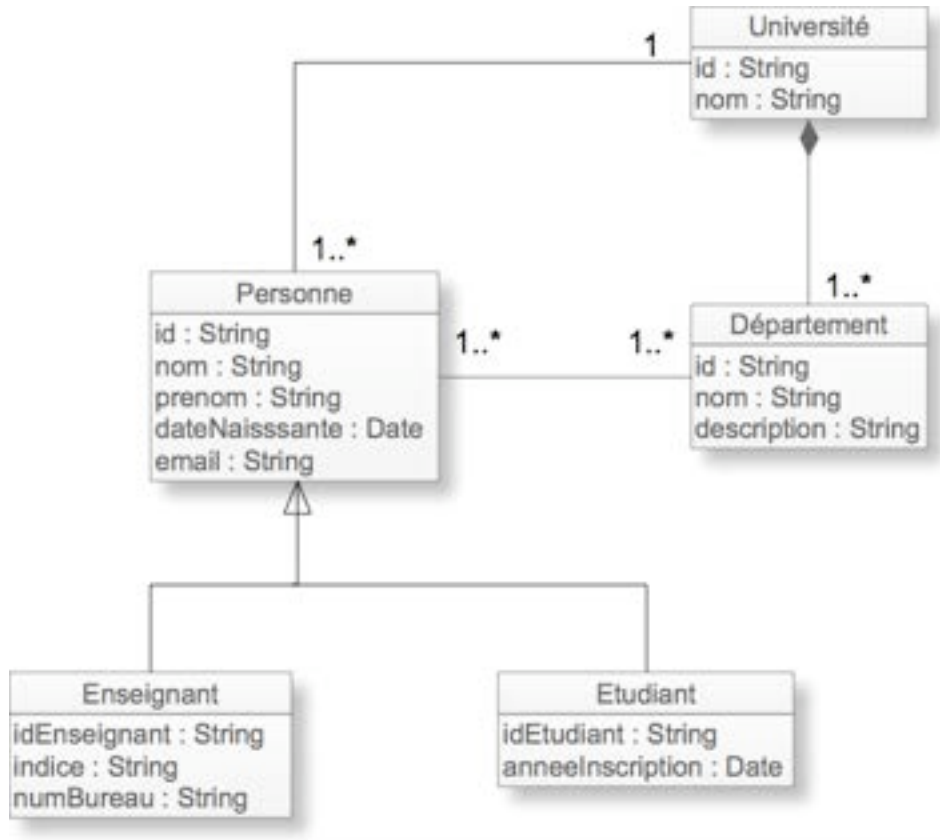


Figure 5.2. Spécificités objet d'un diagramme de classes UML.

Ce diagramme présente les concepts *Personne*, *Enseignant*, *Etudiant*, *Département* et *Université*. Les concepts *Etudiant* et *Enseignant* héritent du concept *Personne*. Ces deux premiers concepts sont donc des spécialisations du concept *Personne*. A l'inverse, ce dernier est une généralisation des concepts *Etudiant* et *Enseignant*. Le concept *Université* est défini comme étant une composition d'instances du concept *Département*.

La généralisation peut se définir comme étant le regroupement d'attributs, d'opérations et d'associations, communs à plusieurs concepts. La figure 5.2 illustre la notion de généralisation des concepts *Etudiant* et *Enseignant* par l'intermédiaire du concept *Personne* qui a pour fonction de mutualiser les attributs communs dans notre exemple. A l'inverse, la spécialisation est l'affinement d'un concept en sous-ensembles. En reprenant le même exemple, les concepts *Etudiant* et *Enseignant* sont des spécialisations du concept *Personne*. En effet, les concepts *Etudiant* et *Enseignant* possèdent des propriétés supplémentaires telles qu'un numéro d'étudiant pour le concept *Etudiant*, et un numéro de bureau et un indice de salaire pour le concept *Enseignant*.

La notion de composition indique qu'un concept appartient à un autre concept. Par exemple, si l'on considère une université composée de différents départements, alors on peut définir le concept *Université* comme étant une composition de départements. L'hypothèse de

composition entre concepts a pour conséquence de créer une dépendance entre ceux-ci. En effet, la notion de composition sous-entend qu'il ne peut y avoir d'instances du concept *Département* sans instances du concept *Université*.

Dans un premier temps, nous nous proposons d'introduire des métadonnées matérialisant ces spécificités « objet » dans le métamodèle XML Schema (notées *nom_concept_objet_UML* à la ligne 3 de la figure 5.3). L'ajout de ces notions objet est une première étape d'homogénéisation des formalismes UML et XML Schema. Pour cela, nous utilisons les mécanismes d'extensions préconisés par XML Schema, soit, pour chaque métaconnaissance, une description sous la forme de l'extension suivante :

```
<xs:annotation>                          [1]
  <xs:appinfo>                           [2]
    <osd:nom_concept_objet_UML/>         [3]
  </xs:appinfo>                          [4]
</xs:annotation>…                        [5]
```

Figure 5.3. Extension XML Schema représentant une métaconnaissance « objet ».

L'ajout de ces métadonnées dans les modèles d'adaptation permet d'inclure des spécificités objet d'UML et de mettre en évidence des relations entre certains concepts. La figure 5.4 illustre la métaconnaissance XML Schema représentant la notion de composition entre les concepts *Université* et *Département*.

```
<xs:complexType name="Université">
  <xs:sequence>
     <xs:element name="id" type="xs:string" minOccurs="1"
      maxOccurs="1"/>
     <xs:element name="nom" type="xs:string" minOccurs="1"
      maxOccurs="1"/>
     <xs:element name="personnes" type="Personne" minOccurs="1"
      maxOccurs="unbounded" />
     <xs:element name="départements" type="Département"
      minOccurs="0" maxOccurs="unbounded">
       <xs:annotation>
         <xs:appinfo>
           <osd:composition/>
         </xs:appinfo>
       </xs:annotation>
```

```
            </xs:element>
        </xs:sequence>
    </xs:complexType>
```

Figure 5.4. Relation de composition UML introduite dans XML Schema.

Au delà de notre processus d'appariement entre XML Schema et UML, ces métadonnées contribuent à optimiser certains traitements tels que la factorisation de données, l'optimisation d'arbres, la suppression d'instances devenues inutiles [Menet *et al.*, 2007]. En effet, l'ajout de ces métadonnées dans le modèle d'adaptation permet de mettre en évidence des relations sémantico-structurelles entre certains concepts. En effet, si l'on considère, dans notre exemple de la figure 5.2, la notion de composition entre les concepts *Université* et *Département*, une instance du concept *Département* ne peut pas exister sans instance du concept *Université*. De ce fait, lors de la suppression d'une instance appartenant au concept *Université*, toutes les instances de *Département* dépendantes de celle-ci seront supprimées.

5.3 Profil UML associé à la sémantique de XML Schema

L'introduction dans la section précédente de spécificités objet d'UML dans le métamodèle XML Schéma est une première étape dans le processus d'homogénéisation de ces deux formalismes. La seconde étape consiste à intégrer la sémantique de XML Schema dans UML. Nous présentons dans cette section le profil UML associé à la sémantique de XML Schema. Le mécanisme d'extension d'UML nous permet d'étendre son formalisme à la sémantique de XML Schema. Cette extension est définie par des stéréotypes et des valeurs marquées. Les stéréotypes sont utilisés pour définir un nouveau type d'élément à partir d'un élément existant du métamodèle UML. Les valeurs marquées sont interprétées comme des attributs d'une métaclasse UML et permettent d'associer à une instance d'un stéréotype des valeurs prédéfinies. La figure 5.5 présente un extrait du profil XML Schema que nous avons défini.

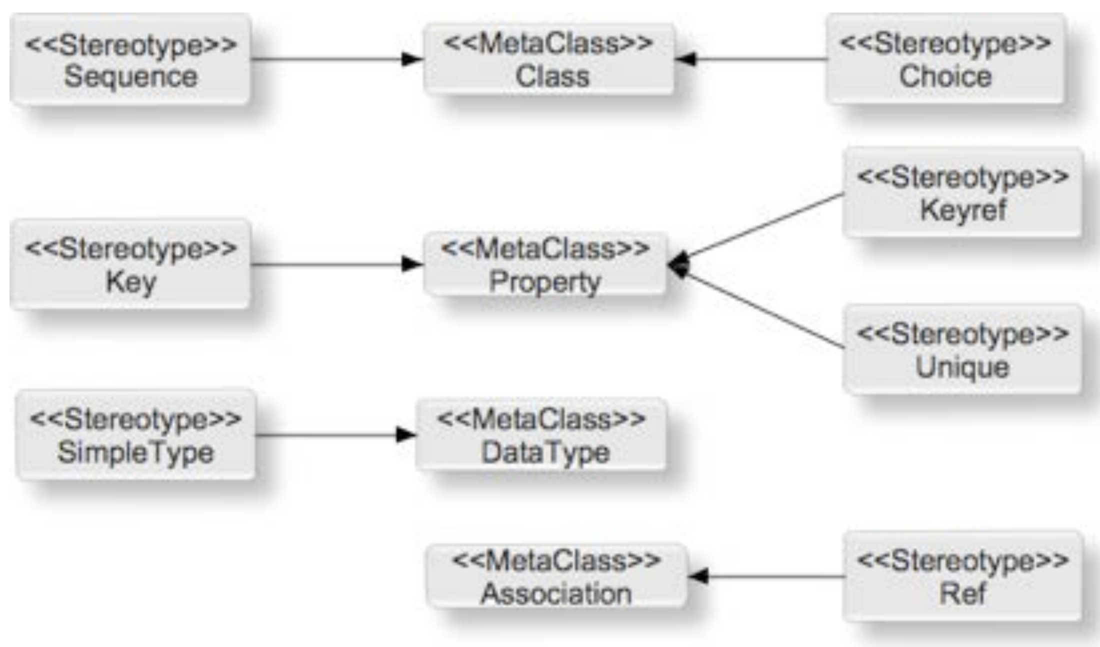


Figure 5.5. Extrait du profil XML Schema[11].

Dans la suite de cette section nous présenterons en détail l'ensemble de ce profil UML. Les stéréotypes de la figure 5.5 (notés *<<Stereotype>>*) héritent respectivement de l'élément *Class*, *DataType*, *Property* et *Association* du métamodèle UML. Par conséquent, chacun de ces stéréotypes sera instancié par le constructeur du métamodèle de la même manière que les éléments *Class*, *DataType*, *Property* ou *Association*. Des valeurs marquées peuvent, de plus, être associées à des stéréotypes. Celles-ci spécifient des paires clés-valeurs pour fixer un ensemble de propriétés d'éléments existants ou de stéréotypes définis. La définition de ces stéréotypes permet d'introduire plus de sémantique, extérieure à UML, nous permettant ainsi de représenter un modèle XML Schema à l'aide de diagrammes UML. Cependant, l'utilisation de diagrammes de classes nous impose d'appliquer des restrictions concernant leur définition. En effet, certains concepts d'UML tels que les opérations, les interfaces ou encore les classes internes ne peuvent pas être représentés avec XML Schéma et doivent par conséquent être exclus lors de la définition d'un modèle XML Schéma via notre profil UML.

Dans la suite de cette section nous proposons de présenter l'ensemble du profil UML que nous avons défini autour de trois axes :

(i) Présentation de la partie du profil UML regroupant les propriétés d'un schéma XML,
(ii) Présentation de la partie du profil UML regroupant la structure d'un modèle XML Schema,

[11] Pour plus d'informations sur la sémantique de XML Schema, voir http://www.w3.org/TR/xmlschema-0/

(iii) Présentation de la partie du profil UML regroupant la définition de contraintes XML Schema.

Pour ce faire, nous décidons d'adopter les conventions de présentation suivantes :

- Présentation du diagramme de classes correspondant à la partie du profil UML décrite.
- Présentation de chaque stéréotype en cinq points :
  - Description du stéréotype,
  - Définition de ses propriétés (valeurs marquées),
  - Définition des métaclasses UML étendues,
  - Définition des stéréotypes étendus,
  - Définition des règles de « bonne modélisation », c'est-à-dire définition des contraintes de modélisation associées au stéréotype,
  - Exemple de notation XML Schema.

5.3.1 Propriétés d'un modèle XML Schema

Dans cette section, nous présentons les éléments de notre profil permettant de spécifier les propriétés d'un modèle XML Schema. Nous présenterons les éléments de déclaration d'un schéma et d'import de modèles XML Schema.

5.3.1.1 Elément de déclaration d'un schéma

La figure 5.6 présente les stéréotypes permettant de définir les propriétés d'un schéma XML.

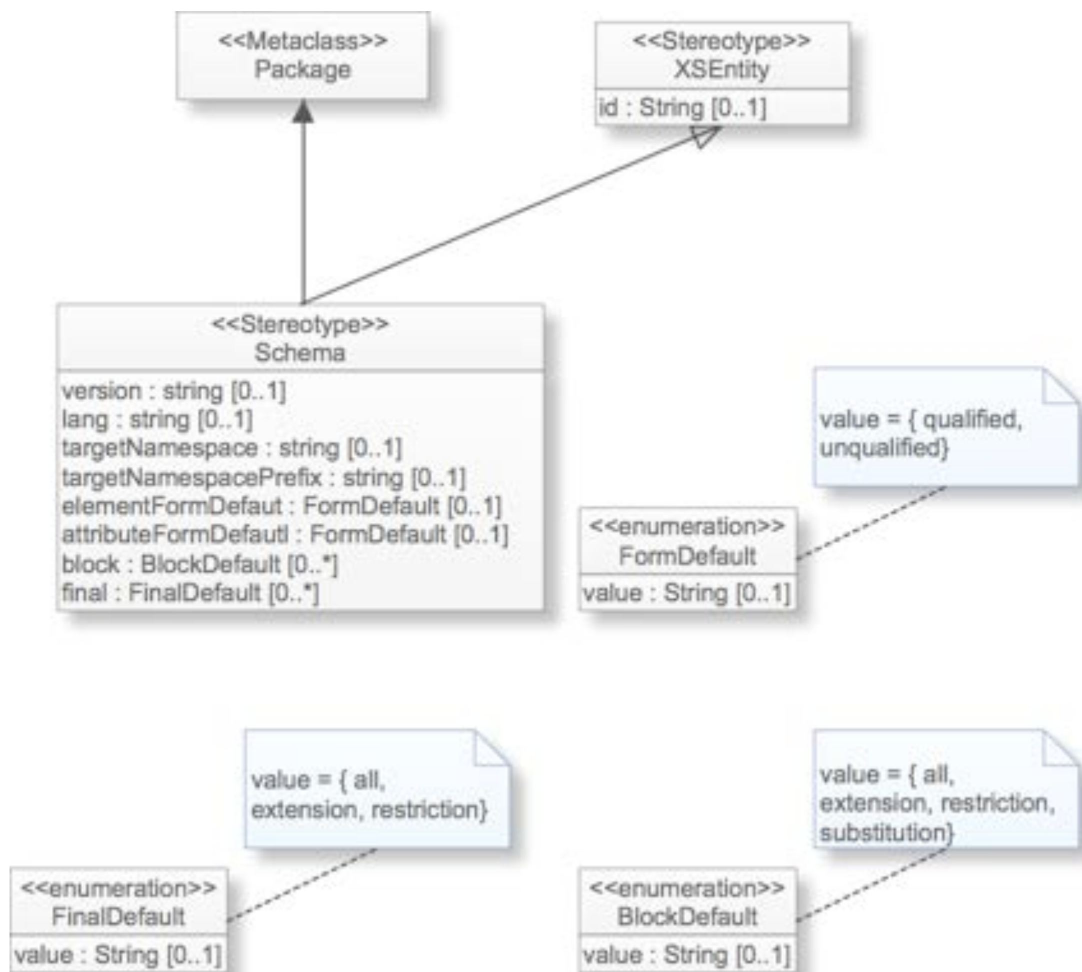


Figure 5.6. Extrait du profil XML : déclaration d'un schéma.

Dans notre profil UML nous avons défini un stéréotype abstrait noté *XSEntity* dans la figure 5.6. Ce stéréotype permet de définir une propriété commune à un ensemble de stéréotypes appartenant à notre profil. Nous verrons par la suite que cet élément est intensivement utilisé dans la définition de notre profil. Le stéréotype *XSEntity* défini une valeur marquée *id*. Cette valeur marquée permet d'associer un identifiant à des élément XML Schema. La figure 5.7 illustre l'utilisation de cet attribut.

```
<xs:simpleType name="typeExample » id="idExample">
    <xs:restriction base="xs:string"/>
</xs:simpleType>
```

Figure 5.7. Définition d'un identifiant XML Schema.

5.3.1.1.1 Stéréotype « Schema »

- Description : ce stéréotype définit la racine d'un schéma XML. Cet élément contient un ensemble de propriétés applicable au modèle XML Schema,
- Propriétés :
  - Version : définit le numéro de version du schéma,
  - Lang : définit la localisation du schéma (Français, anglais, etc.),
  - TargetNamespace : cible une URI[12] représentant l'espace de nommage qui contiendra les définitions du schéma,
  - TargetNamespacePrefix : définit un préfixe qui sera associé à l'espace de nommage spécifié par la propriété *targetNamespace*,
  - ElementFormDefault : propriété indiquant que les éléments définis dans une instance du schéma doivent indiquer explicitement leur espace de nommage,
  - AttributeFormDefault : propriété identique à *ElementFormDefault* mais concerne les attributs définis dans une instance du schéma,
  - Block : permet d'interdire l'utilisation d'un ensemble d'éléments dans le schéma,
  - Final : permet d'interdire l'extension d'éléments dans le schéma.
- Métaclasses étendues : Package,
- Stéréotypes étendus : <<XSEntity>>,
- Contraintes : ce stéréotype doit être défini seul sur un élément de type package.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace=http://www.example.com/example
   attributeFormDefault="qualified">
      . . .
</xs:schema>
```

Figure 5.8. Définition des propriétés d'un schéma XML.

Les cardinalités renseignées dans les diagrammes de classes spécifient si une propriété est obligatoire ou non. Les attributs *elementFormDefault*, *attributeFormDefault*, *block* et *final* sont des énumérations au sens UML ; ces propriétés prennent leurs valeurs dans des ensemble finis. Par exemple la valeur marquée *attributeFormDefault* est de type *FormDefault*. La classe *FormDefault* est définie comme étant une énumération (par le stéréotype <<Enumeration>>) et ses valeurs sont définies dans la note UML associée à la classe. Ainsi

---

[12] Uniform Resource Identifier (URI) est une courte chaîne de caractères identifiant une ressource sur un réseau (par exemple une ressource Web) physique ou abstraite, et dont la syntaxe respecte une norme d'Internet mise en place pour le Web.

dans notre profil l'attribut *attributeFormDefault* peut uniquement avoir pour valeur *qualified* ou *unqualified*.

5.3.1.2 Eléments d'importation de schémas

La figure 5.9 présente les stéréotypes permettant de définir des schémas à importer, à redéfinir ou à inclure dans un modèle XML Schema.

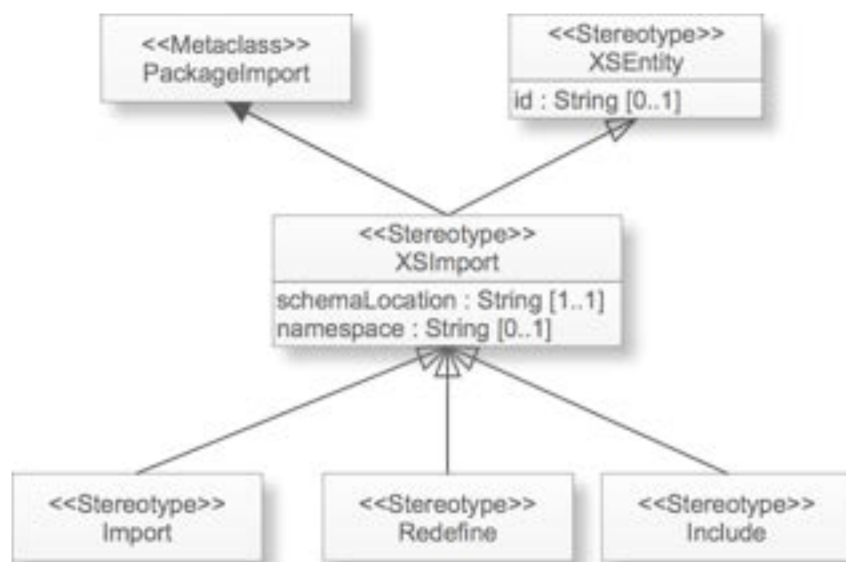


Figure 5.9. Extrait du profil XML : import de schémas.

Nous pouvons noter qu'à première vue les notions d'import et d'inclusion semblent être similaires. La différence entre ces deux termes tient dans le fait que l'import doit être utilisé dans le cas où l'on souhaite se référer à des déclarations ou à des définitions qui sont dans un espace de nommage différent du schéma courant. De plus la notion d'import permet de surcharger les concepts importés. Dans le cas où ces définitions et déclarations se situent ou se situeront dans le même espace de nommage la notion d'inclusion doit être utilisée.

5.3.1.2.1 Stéréotype « XSImport »

- Description : ce stéréotype abstrait décrit les propriétés d'importation, de redéfinition et d'inclusion de schémas,
- Propriétés :
  - SchemaLocation : cible une URI représentant un schéma à utiliser.
  - Namespace : indique l'espace de nommage du schéma à utiliser.

- Métaclasses étendues : PackageImport,
- Stéréotypes étendus : <<XSEntity>>.

5.3.1.2.2 Stéréotype « Import »

- Description : ce stéréotype décrit les propriétés d'importation de schémas,
- Métaclasses étendues : PackageImport,
- Stéréotypes étendus : <<XSImport>>,
- Propriétés : propriétés héritées du stéréotype <<XSImport>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.com/example">
        <xs:import namespace="http://www.example.com/namespace"
            schemaLocation="http://www.example.com/xml.xsd"/>
 . . .
</xs:schema>
```

Figure 5.10. Import d'un schéma XML.

5.3.1.2.3 Stéréotype « Redefine »

- Description : ce stéréotype décrit les propriétés de redéfinition de schémas,
- Métaclasses étendues : PackageImport,
- Stéréotypes étendus : <<XSImport>>,
- Propriétés : propriétés héritées du stéréotype <<XSImport>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.com/example">
        <xs:redefine schemaLocation="./schemaAredefinir.xsd"/>
         . . .
</xs:schema>
```

Figure 5.11. Redéfinition d'un schéma XML.

5.3.1.2.4 Stéréotype « Include »

- Description : ce stéréotype décrit les propriétés d'inclusion de schémas,
- Métaclasses étendues : PackageImport,

- Stéréotypes étendus : XSImport,
- Propriétés : propriétés héritées du stéréotype XSImport,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.com/example">
        <xs:include schemaLocation="schemaAinclure.xsd"/>
        . . .
</xs:schema>
```

Figure 5.12. Inclusion d'un schéma XML.

5.3.2 Structure d'un modèle XML Schema

Dans cette section nous présentons les éléments de notre profil permettant de définir la structure d'un modèle XML Schema. Nous présenterons les éléments de déclaration de types de données, de types complexes, de types simples, d'attributs et de documentation.

5.3.2.1 Types de données XML Schema

La figure 5.13 présente les types XML Schema définis à partir des types de données UML.

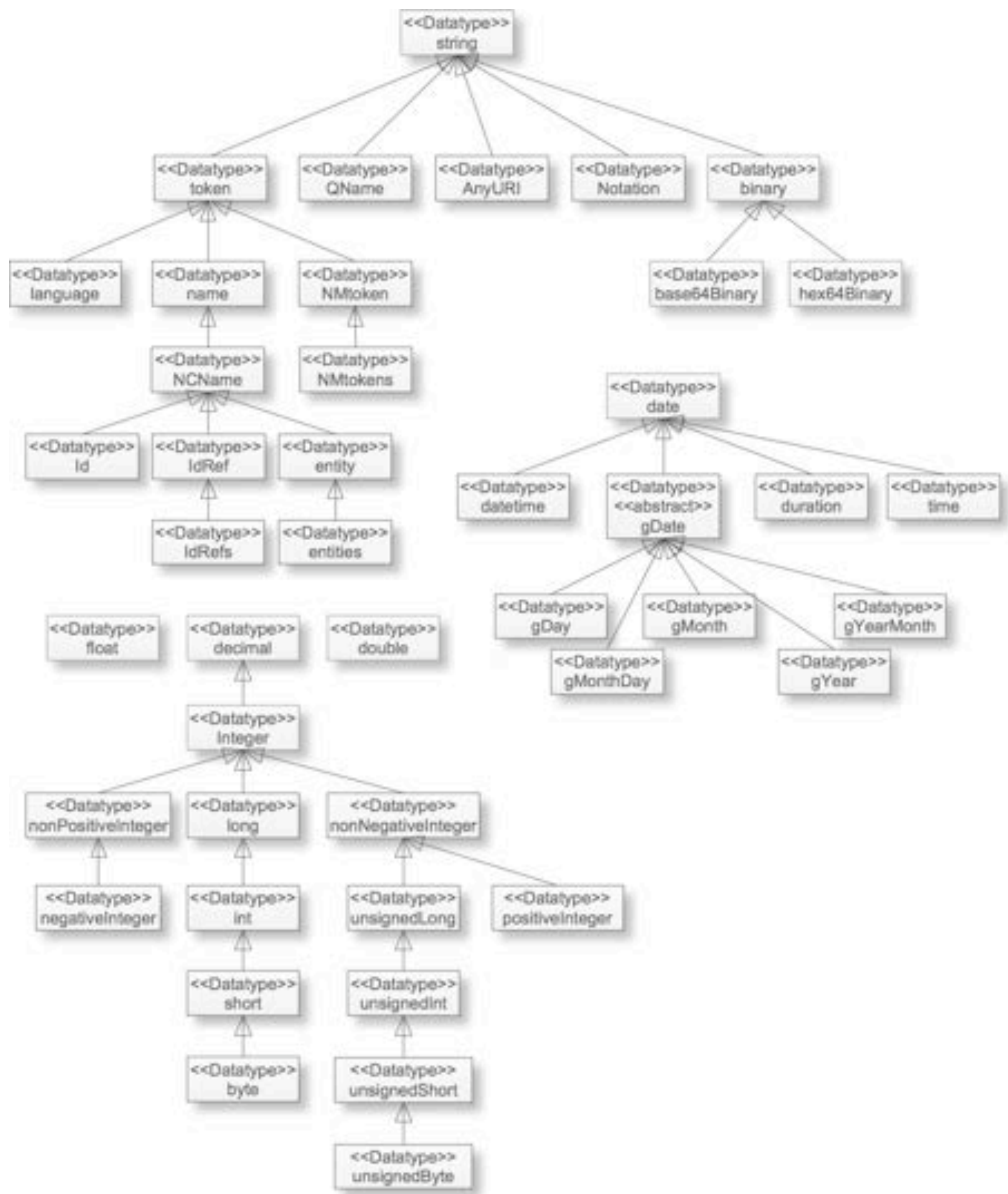


Figure 5.13. Types de données XML Schema.

5.3.2.1 Eléments de déclaration d'éléments complexes

La figure 5.14 présente les stéréotypes permettant de définir des éléments complexes XML Schema.

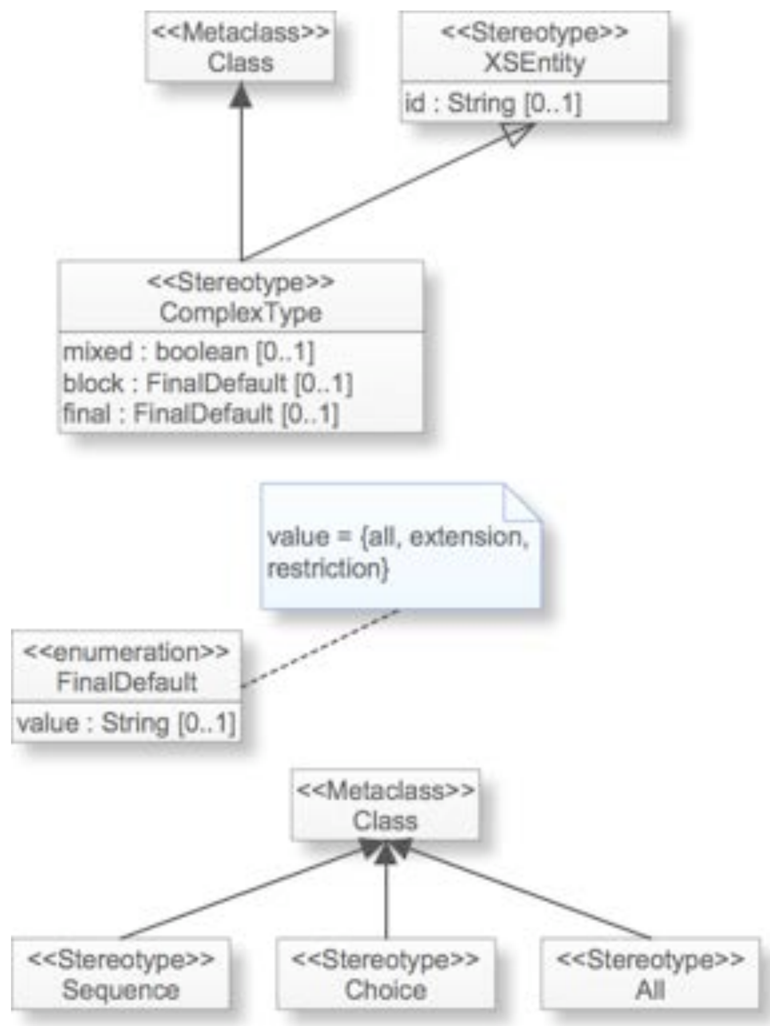


Figure 5.14. Extrait du profil XML : déclaration d'éléments complexes.

5.3.2.1.1 Stéréotype « ComplexType »

- Description : ce stéréotype définit un type complexe XML Schema,
- Métaclasses étendues : Class,
- Stéréotypes étendus : <<XSEntity>>,
- Propriétés :

– Mixed : permet d'indiquer qu'un élément complexe a un contenu comportant du texte et des sous-éléments.
– Block : permet d'interdire l'utilisation d'un ensemble d'éléments,
– Final : permet d'interdire l'extension de cet élément dans le schéma.

- Contraintes : ne peut pas être combiné aux stéréotypes <<group>>, <<attributeGroup>>, <<globalAttribute>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType name="complex1">
        <xs:sequence>
            <xs:element name="elt1" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Figure 5.15. Élément complexe XML Schema.

5.3.2.1.2 Stéréotype « Sequence »

- Description : ce stéréotype indique que l'élément complexe définit une suite ordonnancée de sous-éléments,
- Métaclasses étendues : Class,
- Contraintes : ne peut être combiné aux stéréotypes <<all>>, <<choice>>, <<attributeGroup>> et <<globalAttribute>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType name="complex2">
        <xs:sequence>
            <xs:element name="elt1" type="xs:string"/>
            <xs:element name="elt2" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Figure 5.16. Élément complexe définissant une suite ordonnée d'éléments.

5.3.2.1.3 Stéréotype « Choice »

- Description : ce stéréotype indique qu'un seul des sous-éléments doit être présent dans une instance de cet élément complexe,
- Métaclasses étendues : Class,
- Contraintes : ne peut être combiné aux stéréotypes <<all>>, <<sequence>>, <<attributeGroup>> et <<globalAttribute>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType name="complex3">
        <xs:choice>
            <xs:element name="elt1" type="xs:string"/>
            <xs:element name="elt2" type="xs:string"/>
        </xs:choice>
    </xs:complexType>
</xs:schema>
```

Figure 5.17. Élément complexe définissant un choix d'éléments.

5.3.2.1.4 Stéréotype « all »

- Description : ce stéréotype définit un ensemble de sous-éléments sans contrainte d'ordonnancement,
- Métaclasses étendues : Class,
- Contraintes : ne peut être combiné aux stéréotypes <<sequence>>, <<choice>>, <<attributeGroup>> et <<globalAttribute>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType name="complex3">
        <xs:all>
            <xs:element name="elt1" type="xs:string"/>
            <xs:element name="elt2" type="xs:string"/>
        </xs:all>
    </xs:complexType>
</xs:schema>
```

Figure 5.18. Élément complexe définissant un ensemble d'éléments non ordonnés

5.3.2.2 Eléments de déclaration d'éléments simples

La figure 5.19 présente les stéréotypes permettant de définir des types et des éléments simples :

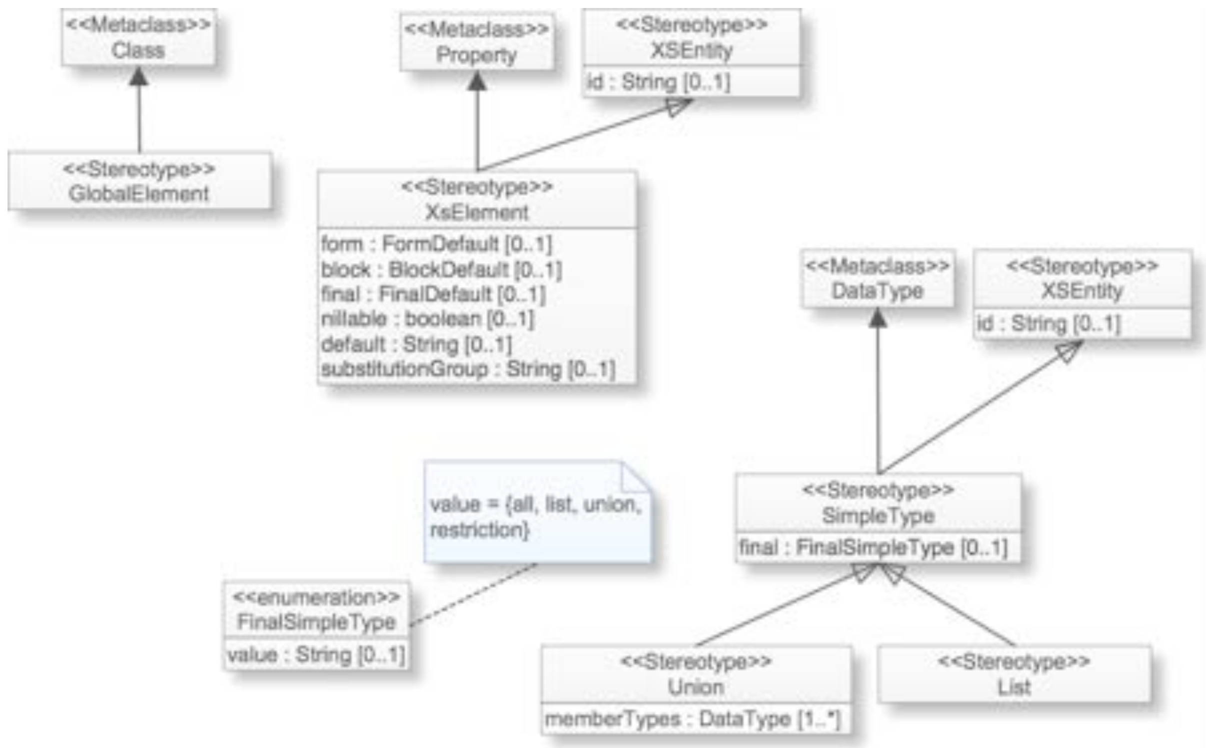


Figure 5.19. Extrait du profil XML : déclaration d'éléments simples.

5.3.2.2.1 Stéréotype « XsElement»

- Description : ce stéréotype définit les propriétés communes aux éléments XML Schema,
- Métaclasses étendues : Property,
- Stéréotypes étendus : <<XSEntity>>,
- Propriétés :
  - Form : permet d'indiquer que les instances de cet élément doivent spécifier explicitement leur espace de nommage,
  - Block : permet d'interdire l'utilisation d'un ensemble de propriétés sur cet élément,
  - Final : permet d'interdire l'extension de cet élément dans le schéma.

  - Nillable : permet d'indiquer qu'une instance de cet élément peut définir une valeur 'null',
  - Default : permet de spécifier une valeur par défaut,
  - SubstitutionGroup : permet de spécifier qu'une instance de cet élément peut être utilisée à la place de l'élément indiqué.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="elt3" type="xs:string" nillable="true"
        block="substitution" form="qualified"
        default="uneValeurParDéfaut"/>
</xs:schema>
```

Figure 5.20. Élément simple XML Schema.

5.3.2.2.2 Stéréotype « GlobalElement »

- Description : ce stéréotype indique la définition d'éléments simples localisés à la racine d'un modèle XML Schema,
- Métaclasses étendues : Class.

5.3.2.2.3 Stéréotype « SimpleType »

- Description : ce stéréotype indique la définition d'un type simple nommé,
- Métaclasses étendues : DataType,
- Stéréotypes étendus : <<XSEntity>>,
- Propriétés :
  - Final : permet d'interdire l'extension de cet élément dans le schéma.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:simpleType name="typeSimple1">
                <xs:restriction base="xs:string">
        </xs:simpleType>
</xs:schema>
```

Figure 5.21. Type simple nommé XML Schema.

5.3.2.2.4 Stéréotype « List »

- Description : ce stéréotype indique la définition d'une liste,
- Métaclasses étendues : DataType,
- Stéréotypes étendus : <<SimpleType>>,
- Propriétés : propriétés héritées du stéréotype <<SimpleType>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:simpleType name="liste">
              <xs:list itemType="xs:integer"/>
        </xs:simpleType>
</xs:schema>
```

Figure 5.22. Liste XML Schema.

5.3.2.2.5 Stéréotype « Union »

- Description : ce stéréotype indique la définition d'une union de types simples,
- Métaclasses étendues : DataType,
- Stéréotypes étendus : <<SimpleType>>,
- Propriétés : propriétés héritées du stéréotype <<SimpleType>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:simpleType name="tailleVetement">>
          <xs:union memberTypes=" tailleParNombre taille "/>
    </xs:simpleType>
    <xs:simpleType name="tailleParNombre">
          <xs:restriction base="xs:positiveInteger"/>
    </xs:simpleType>
    <xs:simpleType name="taille">
          <xs:restriction base="xs:string">
          <xs:enumeration value="petit"/>
          <xs:enumeration value="moyen"/>
          <xs:enumeration value="grand"/>
          </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Figure 5.23. Union de types XML Schema.

5.3.2.3 Eléments de déclaration d'attributs

La figure 5.24 présente les stéréotypes permettant de définir des attributs XML Schema :

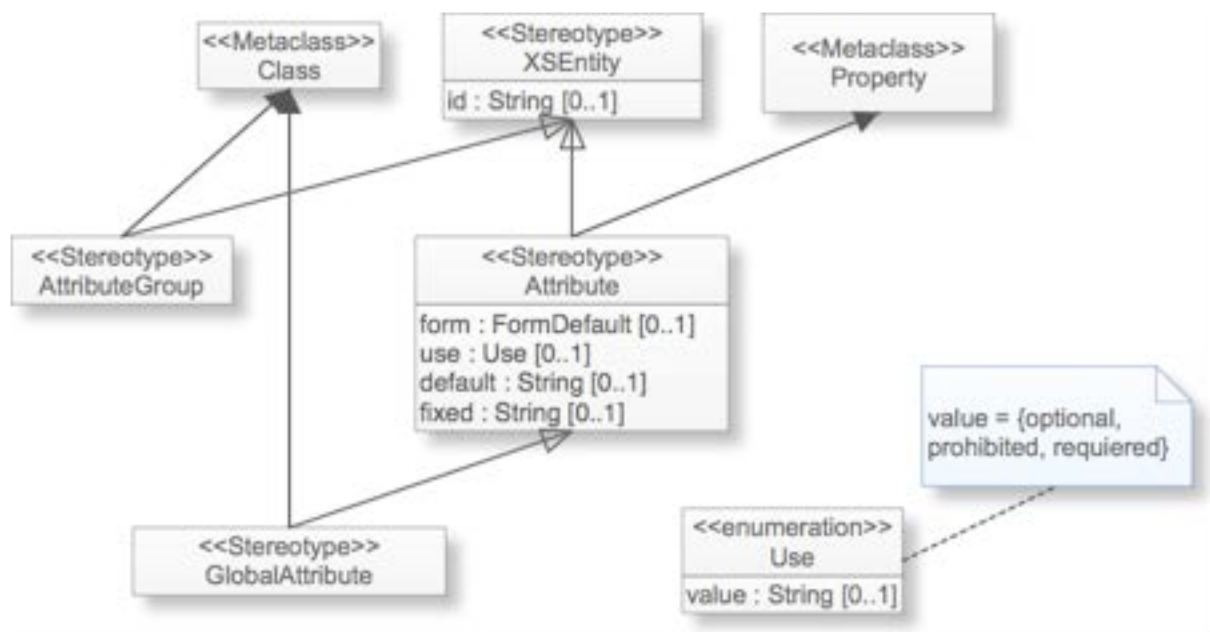


Figure 5.24. Extrait du profil XML : déclaration d'éléments simples.

5.3.2.3.1 Stéréotype « Attribute »

- Description : indique la définition d'un attribut,
- Métaclasses étendues : Property,
- Stéréotypes étendus : <<XSEntity>>,
- Propriétés :
  - Form : permet d'indiquer que les instances de cet attribut doivent spécifier explicitement leur espace de nommage,
  - Use : indique la nature de l'attribut. L'utilisation de cet attribut peut être optionnelle, obligatoire ou bien interdite.
  - Default : permet de spécifier une valeur par défaut,
  - Fixed : permet de définir une valeur fixe pour cet attribut.

- Contraintes : ne peut être combiné aux stéréotypes <<key>>, <<keyref>> et <<unique>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="elt4" type="xs:string">
            <xs:atribute name="att1" type="xs:string" use="required">
        </xs:element>
</xs:schema>
```

Figure 5.25. Attribut XML Schema.

5.3.2.3.2 Stéréotype « GlobalAttribute »

- Description : indique la définition d'un attribut global au schéma,
- Métaclasses étendues : Class, Property
- Stéréotypes étendus : <<Attribute>>,
- Propriétés : propriétés héritées du stéréotype <<Attribute>>,
- Contraintes : ne peut être combiné aux stéréotypes <<all>>, <<choice>>, <<sequence>>, <<attributeGroup>> et <<complexType>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:atribute name="att1" type="xs:string" use="required">
        <xs:atribute name="att2" type="xs:date" use="optional">
        <xs:atribute name="att3" type="xs:integer">
</xs:schema>
```

Figure 5.26. Attribut global à un schéma XML.

5.3.2.3.3 Stéréotype « AttributeGroup »

- Description : indique la définition d'un groupe d'attributs,
- Métaclasses étendues : Class,
- Stéréotypes étendus : <<XSEntity>>,
- Propriétés : propriétés héritées du stéréotype <<XSEntity>>,
- Contraintes : ne peut être combiné aux stéréotypes <<all>>, <<choice>>, <<sequence>>, <<GlobalAttribute>> et <<complexType>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
        <xs:attributeGroup name="attGroup">
            <xs:atribute name="att1" type="xs:string">
            <xs:atribute name="att2" type="xs:date">
            <xs:atribute name="att3" type="xs:integer">
        </xs:attributeGroup>
    </xs:schema>
```

Figure 5.27. Groupe d'attributs XML Schema.

5.3.2.4 Eléments de déclaration de documentation

La figure 5.28 présente les stéréotypes permettant de définir des éléments de documentation XML Schema :

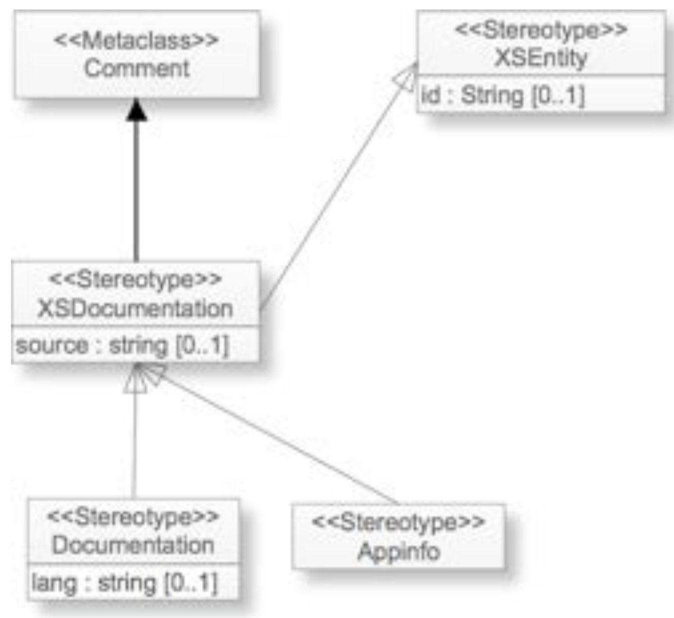


Figure 5.28. Extrait du profil XML : éléments de documentation XML Schema.

5.3.2.4.1 Stéréotype « XSDocumentation »

- Description : stéréotype abstrait définissant un élément de documentation XML Schema,
- Métaclasses étendues : Comment,
- Stéréotypes étendus : <<XSEntity>>,
- Propriétés :

- Source: définit une URI vers une ressource externe de documentation.

5.3.2.4.2 Stéréotype « Documentation »

- Description : stéréotype définissant un élément de documentation XML Schema,
- Métaclasses étendues : Comment,
- Stéréotypes étendus : <<XSDocumentation>>,
- Propriétés :
  - Propriétés héritées du stéréotype XSDocumentation,
  - Lang : spécifie la langue de l'élément de documentation.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:annotation>
        <xs:documentation xml:lang="fr">
        Exemple de documentation d'un schéma XML
        </xs:documentation>
    </xs:annotation>
</xs:schema>
```

Figure 5.29. Documentation XML Schema.

5.3.2.4.3 Stéréotype « AppInfo »

- Description : stéréotype définissant un élément de documentation additionnelle XML Schema,
- Métaclasses étendues : Comment,
- Stéréotypes étendus : <<XSDocumentation>>,
- Propriétés : Propriétés héritées du stéréotype <<XSDocumentation>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:annotation>
        <xs:appInfo>Documentation additionnelle</xs:appInfo>
    </xs:annotation>
</xs:schema>
```

Figure 5.30. Élément de documentation additionnelle.

5.3.3 Définition de contraintes XML Schema

Dans cette section, nous présentons les éléments de notre profil permettant d'associer des contraintes à des éléments XML Schema. Nous présenterons les éléments de déclaration de contrainte d'unicité et de référence ainsi que les contraintes portant sur la valeur.

5.3.3.1 Eléments de déclaration de contrainte d'unicité et de référence

La figure 5.31 présente les stéréotypes permettant de définir des références vers des éléments et des contraintes d'unicité XML Schema :

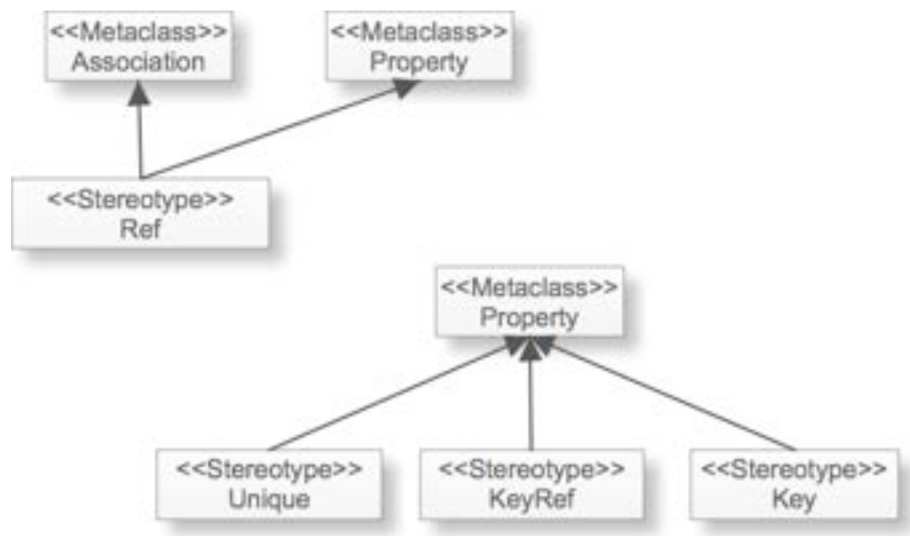


Figure 5.31. Extrait du profil XML : éléments de définition de contrainte d'unicité et de référence.

5.3.3.1.1 Stéréotype « Ref »

- Description : stéréotype définissant une référence vers un élément défini dans le schéma,
- Métaclasses étendues : Property, Association,
- Contraintes : ne peut être combiné aux stéréotypes <<key>>, <<keyref>> et <<unique>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="nom" type="xs:string"/>
```

```
        <xs:element name="adresse" type="xs:string"/>
        <xs:element name="personne">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="nom"/>
              <xs:element ref="adresse"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
</xs:schema>
```

Figure 5.32. Référence vers un élément XML Schema.

5.3.3.1.2 Stéréotype « Key »

- Description : définit un élément ou un attribut en tant que clé, la valeur de cet élément devra être unique, présente et non nulle dans l'instance du schéma.
- Métaclasses étendues : Property,
- Contraintes : ne peut être combiné aux stéréotypes <<keyref>> et <<unique>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="personne">
          <xs:complexType>
            <xs:sequence>
                  <xs:element name="id" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:key name="idKey">
            <xs:selector="personne"/>
            <xs:field xpath="id"/>
        </xs:key>
</xs:schema>
```

Figure 5.33. Définition d'une clé XML Schema.

5.3.3.1.3 Stéréotype « Unique »

- Description : indique que la valeur d'un élément doit être unique.
- Métaclasses étendues : Property,
- Contraintes : ne peut être combiné aux stéréotypes <<key>>, <<keyref>> et <<ref>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="personne">
     <xs:complexType>
      <xs:sequence>
           <xs:element name="id" type="xs:string"/>
      </xs:sequence>
     </xs:complexType>
       <xs:unique name="idKey">
            <xs:selector="."/>
            <xs:field xpath="./id"/>
       </xs:unique>
    </xs:element>
</xs:schema>
```

Figure 5.34. Définition d'une contrainte d'unicité XML Schema.

5.3.3.2.4 Stéréotype « KeyRef »

- Description : indique que la valeur d'un élément ou d'un attribut correspond à celle d'un élément défini en tant que clé ou étant unique.
- Métaclasses étendues : Property,
- Contraintes : ne peut être combiné aux stéréotypes <<key>>, <<ref>> et <<unique>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="personne">
     <xs:complexType>
      <xs:sequence>
          <xs:element name="id" type="xs:string"/>
      </xs:sequence>
```

```
        </xs:complexType>
      </xs:element>
      <xs:key name="idKey">
        <xs:selector="personne"/>
        <xs:field xpath="id"/>
      </xs:key>
      <xs:keyref name="idKeyRef" refer="idKey"/>
</xs:schema>
```

Figure 5.35. Définition d'une référence vers une contrainte d'unicité XML Schema.

5.3.3.2 Eléments de déclaration de contraintes sur valeurs

La figure 5.36 présente les stéréotypes permettant de définir des contraintes sur les valeurs des éléments définis dans un schéma :

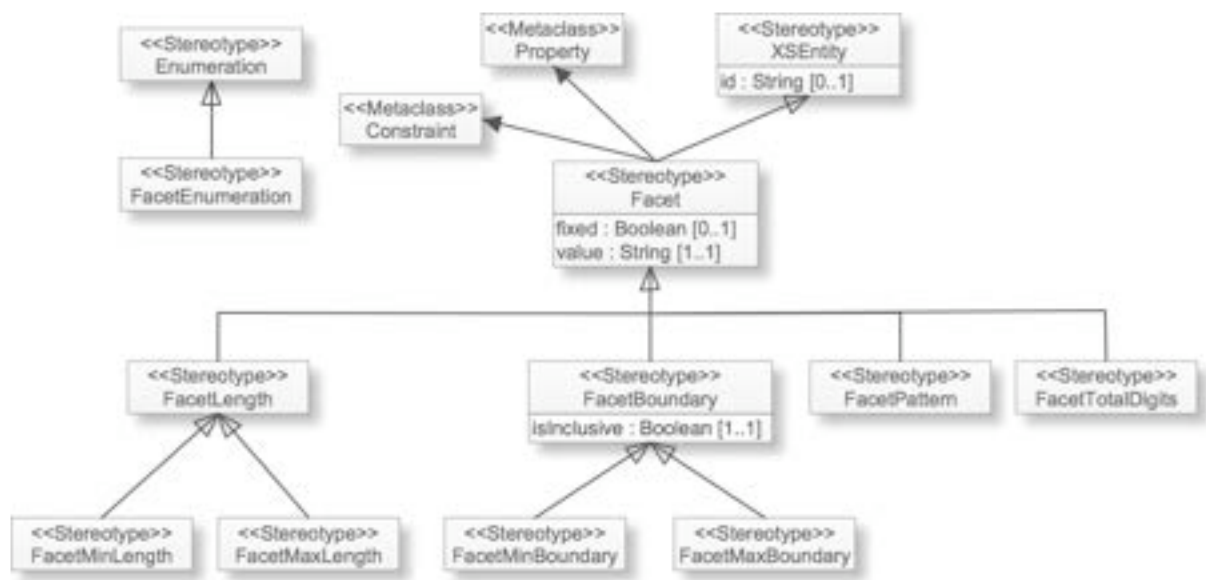


Figure 5.36. Extrait du profil XML : éléments de contrainte d'unicité et de référence.

5.3.3.2.1 Stéréotype « Facet »

- Description : stéréotype abstrait définissant une contrainte XML Schema,
- Métaclasses étendues : Property, Constraint

- Stéréotypes étendus : <<XSEntity>>,
- Propriétés :
  - Fixed : indique si la valeur de la contrainte peut être redéfinie,
  - Valeur : spécifie la valeur de la contrainte.
- Contraintes : ne peut être combiné aux stéréotypes <<key>>, <<keyref>> et <<unique>>.

5.3.3.2.2 Stéréotype « FacetEnumeration »

- Description : définit une énumération contenant l'ensemble des valeurs pouvant être affectées à un élément,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<Enumeration>>,
- Propriétés : propriétés héritées du stéréotype <<Facet>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="tailleVetement">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="petit"/>
                <xs:enumeration value="moyen"/>
                <xs:enumeration value="grand"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:element>
</xs:schema>
```

Figure 5.37. Enumération XML Schema.

5.3.3.2.3 Stéréotype « FacetLength »

- Description : définit une contrainte sur la longueur de la valeur d'un élément,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<Facet>>,
- Propriétés : propriétés héritées du stéréotype <<Facet>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="tailleVetement">
```

```
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:length value="2"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
    </xs:schema>
```

Figure 5.38. Contrainte sur la longueur d'une valeur.

5.3.3.2.4 Stéréotype « FacetMinLength »

- Description : définit une contrainte sur la longueur minimale de la valeur d'un élément,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<Facet>>,
- Propriétés : propriétés héritées du stéréotype <<Facet>>,
- Exemple XML Schema :

```
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="password">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:minLength value="6"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
    </xs:schema>
```

Figure 5.39. Contrainte sur la longueur minimale d'une valeur.

5.3.3.2.4 Stéréotype « FacetMaxLength »

- Description : définit une contrainte sur la longueur maximale de la valeur d'un élément,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<Facet>>,
- Propriétés : propriétés héritées du stéréotype <<Facet>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="name">
        <xs:simpleType>
          <xs:restriction base="xs:string">
             <xs:maxLength value="20"/>
          </xs:restriction>
        </xs:simpleType>
    </xs:element>
</xs:schema>
```

Figure 5.40. Contrainte sur la longueur maximale d'une valeur.

5.3.3.2.5 Stéréotype « FacetPattern »

- Description : définit une contrainte sur la forme de la valeur d'un élément,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<Facet>>,
- Propriétés : propriétés héritées du stéréotype <<Facet>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="id">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:pattern value="[0-9]{4}"/>
          </xs:restriction>
        </xs:simpleType>
    </xs:element>
</xs:schema>
```

Figure 5.41. Contrainte sur la « forme » d'une valeur.

5.3.3.2.5 Stéréotype « FacetTotalDigits »

- Description : définit une contrainte sur le nombre de chiffres composant la valeur d'un élément,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus :<<Facet>>,

- Propriétés : propriétés héritées du stéréotype <<Facet>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="codePostal">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:totalDigits value="5"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:element>
</xs:schema>
```

Figure 5.42. Contrainte sur le nombre de chiffres d'un élément.

5.3.3.2.6 Stéréotype « FacetBoundary »

- Description : stéréotype abstrait définissant une borne appliquée à la valeur d'un élément,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<Facet>>,
- Propriétés :
  - Propriétés héritées du stéréotype <<Facet>>.
  - IsInclusive : indique si la borne est incluse dans la restriction.

5.3.3.2.7 Stéréotypes « Facet{Min/Max}Boundary»

- Description : définit une borne minimale ou maximale appliquée à la valeur d'un élément,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<FacetBoundary>>,
- Propriétés : propriétés héritées du stéréotype <<FacetBoudary>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="size">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:minInclusive value="36"/>
                <xs:maxInclusive value="50"/>
```

```
            </xs:restriction>
        </xs:simpleType>
    </xs:element>
</xs:schema>
```

Figure 5.43. Contraintes permettant de borner la valeur d'un élément.

À partir du profil UML que nous venons de présenter il est ainsi possible de définir à l'aide d'un diagramme de classes un modèle XML Schema. Nous présenterons des exemples complets d'utilisation de ce profil dans le chapitre suivant qui sera consacré aux appariements entre UML et XML Schema.

5.4 Profil Master Data Management

Nous avons vu dans les chapitres précédents l'importance d'adopter une approche d'Ingénierie Dirigée par les Modèles pour définir des modèles de données. Dans le cadre de nos travaux, notre approche IDM vise à s'appliquer au domaine du Master Data Management. Nous avons aussi vu dans le troisième chapitre de cette thèse que les modèles de données relatifs à notre solution de Master Data Management sont basés sur des modèles XML Schema. Ces modèles que nous avons appelés modèles d'adaptation sont des modèles conformes aux spécifications XML Schema et disposent d'une sémantique additionnelle associée au Master Data Management. La seconde étape de notre approche IDM vise à définir un second profil dédié au domaine du Master Data Management, et se basant sur le profil XML Schema que avons présenté dans la section précédente. Dans cette section nous présentons un profil UML dédié au Master Data Management en se basant sur les propriétés définies dans notre solution MDM EBX.Platform[13] présenté dans le troisième chapitre de cette thèse. Le profil MDM que nous avons développé est défini en quatre parties :

(i) Stéréotypes permettant de définir un modèle d'adaptation à partir d'un modèle XML Schema.
(ii) Stéréotypes représentant les propriétés avancées définis par notre solution MDM.
(iii) Stéréotypes représentant des contraintes avancées définies à partir des contraintes XML Schema.
(iv) Stéréotypes représentant des éléments de documentation permettant d'étendre les entités de documentation XML Schema.

Un modèle d'adaptation est un modèle de données enrichi pour les données de référence ou Master Data. Globalement, les objectifs sont de garantir la cohérence de ces données et de faciliter leur gestion. Concrètement, le modèle d'adaptation est un document

[13] La documentation en ligne est disponible à l'adresse http://doc.orchestranetworks.com/

conforme au standard XML Schema (recommandation W3C). Les principales caractéristiques standards qui sont supportées sont les suivantes :

- une riche bibliothèque de types de données (types simples : integer, boolean, decimal, date, time, ...),
- la possibilité de définir des structures complexes (complex types),
- la possibilité de définir des listes simples d'éléments (listes agrégées),
- la spécification de contraintes de validation (facettes XML Schema) telles que énumérations, unicité, bornes minimum et maximum, etc.

Notre solution utilise également les capacités d'extension d'XML Schema afin de définir des informations utiles telles que :

- types pré-définis (locale, resource, html, ...),
- définition de tables et contraintes de clé étrangère,
- mapping de données et de Java beans,
- contraintes de validation avancées (facettes étendues) telles que des énumérations dynamiques,
- informations de présentation étendues : libellé, description, messages d'erreur, ...

Notre solution MDM EBX.Platform supporte un sous-ensemble de la recommandation du W3C ; certaines caractéristiques du standard sont en effet inutiles pour les Master Data.

Le profil UML que nous proposons de définir décrit ces éléments permettant de représenter les extensions XML Schéma.

5.4.1 Définition d’un modèle d’adaptation

La figure 5.44 présente les stéréotypes permettant de définir un modèle d’adaptation à partir d’un modèle XML Schema :

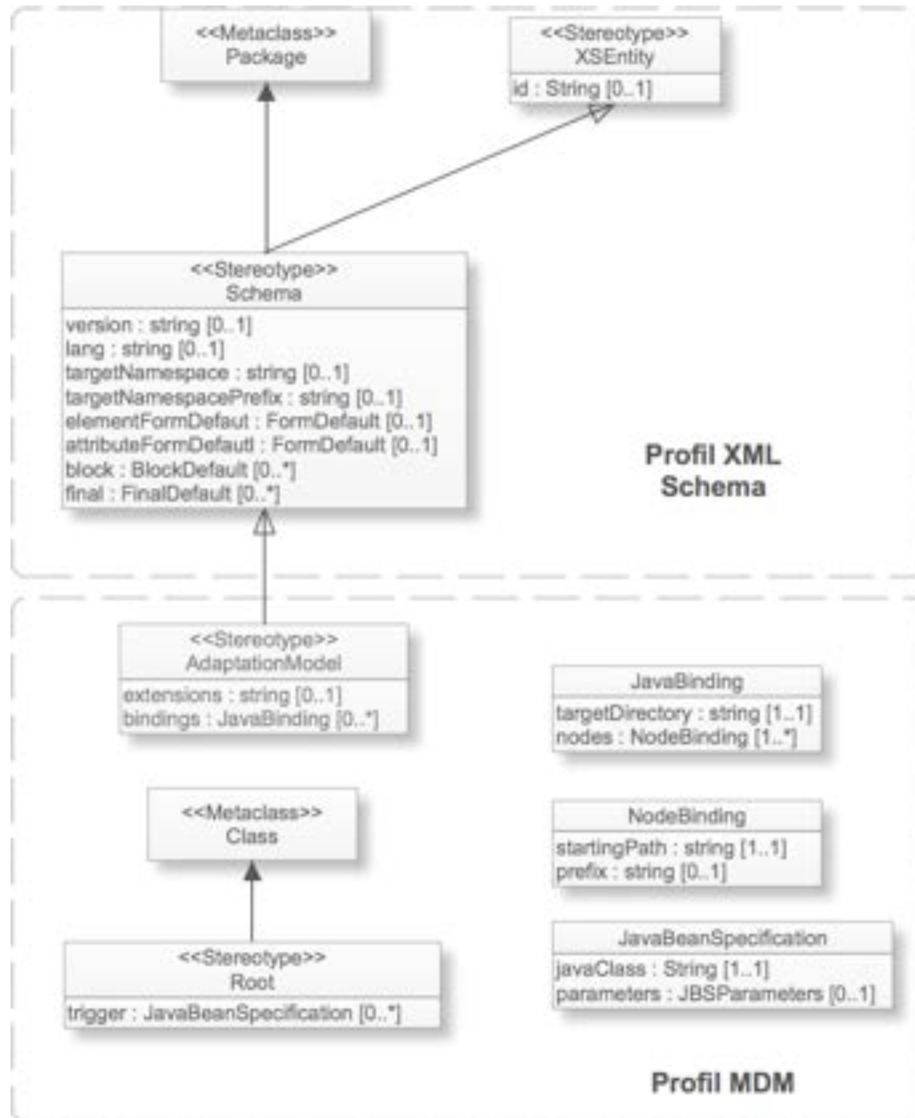


Figure 5.44. Extrait du profil MDM : propriétés d'un modèle d'adaptation.

5.4.1.1 Stéréotype « AdaptationModel »

- Description : stéréotype spécifiant la définition d'un modèle d'adaptation.
- Métaclasses étendues : Package,
- Stéréotypes étendus : <<Schema>>,
- Propriétés :
  - Propriétés héritées du stéréotype <<Schema>>,

– Bindings : dans notre solution EBX.Platfrom il est possible de générer des types Java à partir d'un modèle d'adaptation. Cet attribut spécifie quels sont les types Java à générer à partir du schéma XML. Le fait d'assurer un lien entre la structure du schéma XML et le code Java apporte de nombreux avantages à savoir : (i) **Assistance au développement** par la complétion automatique lors de l'écriture de chemins d'accès à des paramètres (dans la mesure où l'environnement de développement le supporte), (ii) **Vérification du code d'appel** : l'ensemble des appels aux paramètres d'adaptation est vérifié dès la compilation du code, (iii) **Mesure d'impact** : toute modification du modèle d'adaptation impacte immédiatement l'état de compilation du code, (iv) **Référencement** : en utilisant l'outil de référencement de l'environnement de développement, il est facile de lister où est utilisé un paramètre donné. Les classes *JavaBinding* et *NodeBinding* de la figure 5.42 permettent de spécifier les paramètres des types Java à générer. La spécification des types Java à générer à partir du schéma XML est incluse directement dans le schéma XML principal du modèle d'adaptation. Chaque élément *binding* définit une cible de génération. Il doit être situé à l'emplacement *xs:schema/xs:annotation/xs:appinfo/ebxbnd:binding* (notation XPath) où le préfixe *ebxbnd* fait référence à l'espace de nommage identifié par une URI spécifique. Plusieurs éléments *binding* peuvent être définis s'il y a des cibles de génération distinctes. L'attribut *targetDirectory* de la classe *Javabinding* définit le répertoire racine de génération des types Java (généralement, il s'agit du répertoire des codes sources d'un projet par exemple). Un chemin relatif est interprété relativement au répertoire courant d'exécution de la machine virtuelle Java (et non par rapport à la localisation du schéma). En définissant des *bindings*, nous obtenons la capacité d'associer des constantes Java aux chemins d'un modèle d'adaptation représenté par un modèle XML Schema. Pour cela, nous générons une ou plusieurs interfaces à partir d'un nœud du schéma (pouvant être le nœud racine "/"). Les noms d'interface sont décrits dans les balises *javaPathConstants* avec l'attribut *typeName* et le nœud associé est décrit dans la balise *nodes* avec l'attribut *root*.

- Contraintes : Ce stéréotype ne peut pas être combiné à d'autres stéréotypes applicables à un package.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:annotation>
        <xs:appinfo>
          <ebxbnd:binding targetDirectory="chemin d'un dossier"
              <javaPathConstants typeName="com.org.NomClasseJava">
                  <nodes root="/root" prefix=""/>
              </javaPathConstants>
          </ebxbnd:binding>
        </xs:appinfo>
```

```
    </xs:annotation>
    …
</xs:schema>
```

Figure 5.45. Définition d'un modèle d'adaptation et de bindings.

5.4.1.2 Stéréotype « Root »

- Description : Pour être accepté par notre solution MDM, un schéma XML doit inclure la déclaration d'un élément global comportant l'attribut *osd:access="--"*. Cet élément global n'est autre que la racine (Root) d'un modèle d'adaptation. Nous définissons donc un stéréotype à cet effet.
- Métaclasses étendues : Class,
- Propriétés :
  - Trigger : Il est possible d'associer à des instances d'un modèle d'adaptation des méthodes qui seront exécutées automatiquement par une classe Java lorsque certaines opérations sont effectuées telles que les opération de création, mise à jour, suppression, etc. L'attribut *trigger* permet de spécifier les propriétés de la classe Java à invoquer. Ces propriétés sont définies par la classe *JavaBeanSpecification*. Cette classe possède les attributs *javaClass* et *parameters* qui définissent respectivement le nom qualifié (sous la forme nom_package.nom_classe_java) de la class Java à invoquer et les paramètres associés en conformité à la spécification JavaBean[14].
- Contraintes : ce stéréotype ne peut pas être combiné à d'autres stéréotypes définis dans ce profil.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" osd:access="--">
        <xs:annotation>
            <xs:appinfo>
                <osd:trigger class="com.foo.MyInstanceTrigger">
                    <param1>...</param1>
                    <param...n>...</param...n>
                </osd:trigger>
            </xs:appinfo>
```

[14] La spécification JavaBean est disponible en ligne à http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp

```
        </xs:annotation>
      </xs:element>
      …
  </xs:schema>
```

Figure 5.46. Racine d'un modèle d'adaptation et définition d'un trigger.

5.4.2 Propriétés et structures avancées

5.4.2.1 Types de données étendus

Afin de répondre à des problématiques spécifiques de manipulation et de présentation de données, nous avons défini dans notre solution MDM des types de données étendus. Ces types sont définis à partir des types de données XML Schema. La figure 5.47 présente les types de données étendus que nous avons définis.

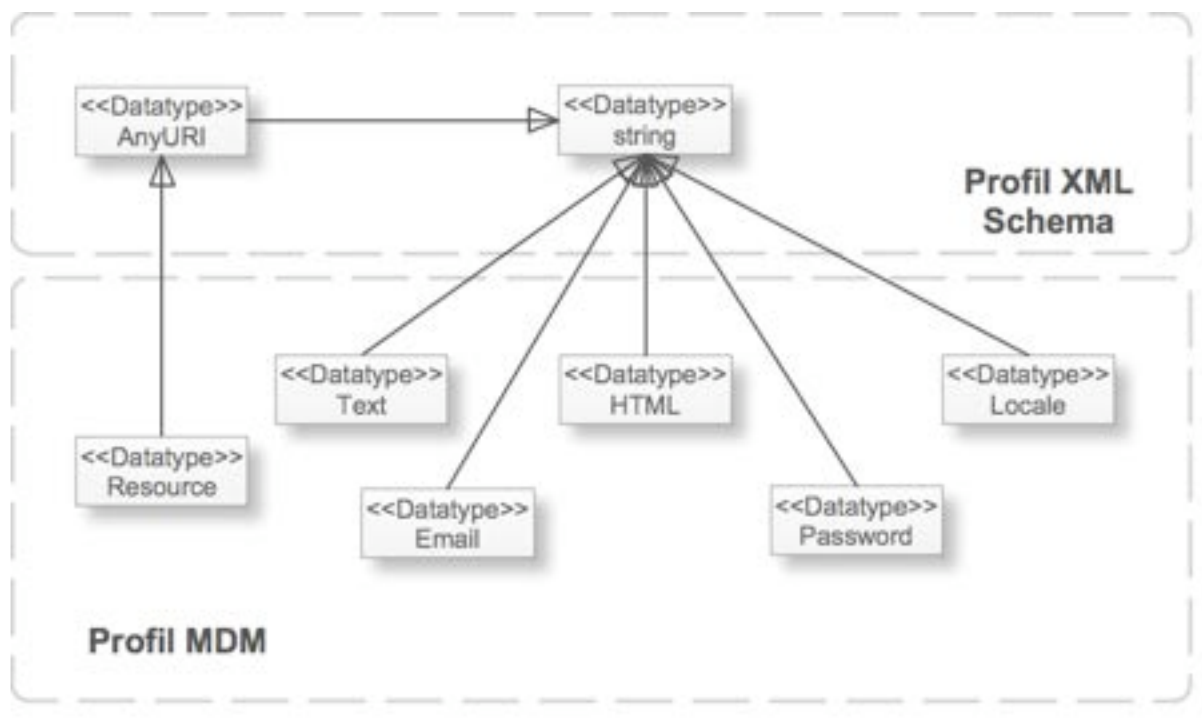


Figure 5.47. Extrait du profil MDM : types de données étendus.

- Text : représente une information textuelle. Vis-à-vis de la chaîne de caractères simple (xs:string), l'interface utilisateur qu'offre par défaut EBX.Manager est un éditeur multi-ligne.
- Html : une instance de type *html* représente une chaîne de caractères au format HTML. Un éditeur “wysiwyg”[15] dédié est disponible dans EBX.Manager pour éditer ces instances.
- Email : une instance de type *email* représente une adresse email comme défini dans la norme RFC822[16].
- Password : une instance de type *password* représente un mot de passe crypté. Ce type est associé à un composant graphique spécifique à la saisie de mots de passe (avec confirmation de la saisie).
- Resource : une instance de type *resource* représente une ressource EBX.Platform. Une ressource est un fichier de type image, html, css, ou javascript, localisée à l'intérieur d'un module EBX.Platform.
- Locale : une instance de type *locale* représente une région géographique, politique ou culturelle spécifique.

5.4.2.2 Définition de services

Notre solution MDM propose un ensemble de fonctionnalités permettant de gérer des données de référence. Dans certains cas, des fonctionnalités spécifiques et non présentes dans notre outil peuvent être requises telles que la mise à jour automatique de systèmes distants ou la définition de pistes d’audit spécialisées. Nous avons défini la notion de *service* dans le but d’intégrer dans notre solution MDM des applications Java/JSP. Les services permettent d’enrichir notre outil et ainsi fournir aux utilisateurs des fonctionnalités additionnelles.

La figure 5.48 présente l’extrait de notre profil permettant de définir les services à inclure dans un modèle d’adaptation :

---

[15] Acronyme de la locution anglaise *What you see is what you get*. Désigne les interfaces utilisateur graphiques permettant de composer visuellement des graphiques complexes.

[16] http://www.w3.org/Protocols/rfc822/

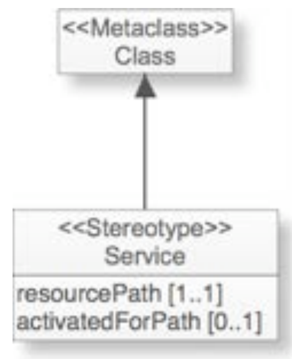


Figure 5.48. Extrait du profil MDM : définition de services.

5.4.2.2.1 Stéréotype << Service >>

- Description : définit un service permettant de spécifier des fonctionnalités additionnelles à importer.
- Métaclasses étendues : Class,
- Propriétés :
  - ResourcePath : spécifie le chemin du service à importer.
  - ActivatedForPath : spécifie sur quels éléments du schéma le service est disponible. Par défaut, un service est disponible sur l'instance du modèle d'adaptation.
- Contraintes : ce stéréotype ne peut pas être combiné à d'autres stéréotypes définis sur un élément de type classe.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType name="serviceExample">
        <xs:annotation>
            <xs:appinfo>
                <osd:service resourcePath="/importXML/importXML.jsp"
                    activatedForPath="/root/table1" />
            </xs:appinfo>
        </xs:annotation>
    </xs: complexType >
    …
</xs:schema>
```

Figure 5.49. Définition d'un service.

5.4.2.3 Structures avancées d'un modèle d'adaptation

Afin de répondre à des besoins avancés de gestion de données (représentation graphique, appariements avec des objets Java, etc.), nous définissons différentes entités permettant d'enrichir la structure d'un modèle XML Schema.

La figure 5.50 présente l'extrait de notre profil permettant de définir des éléments avancés au sein d'un modèle d'adaptation :

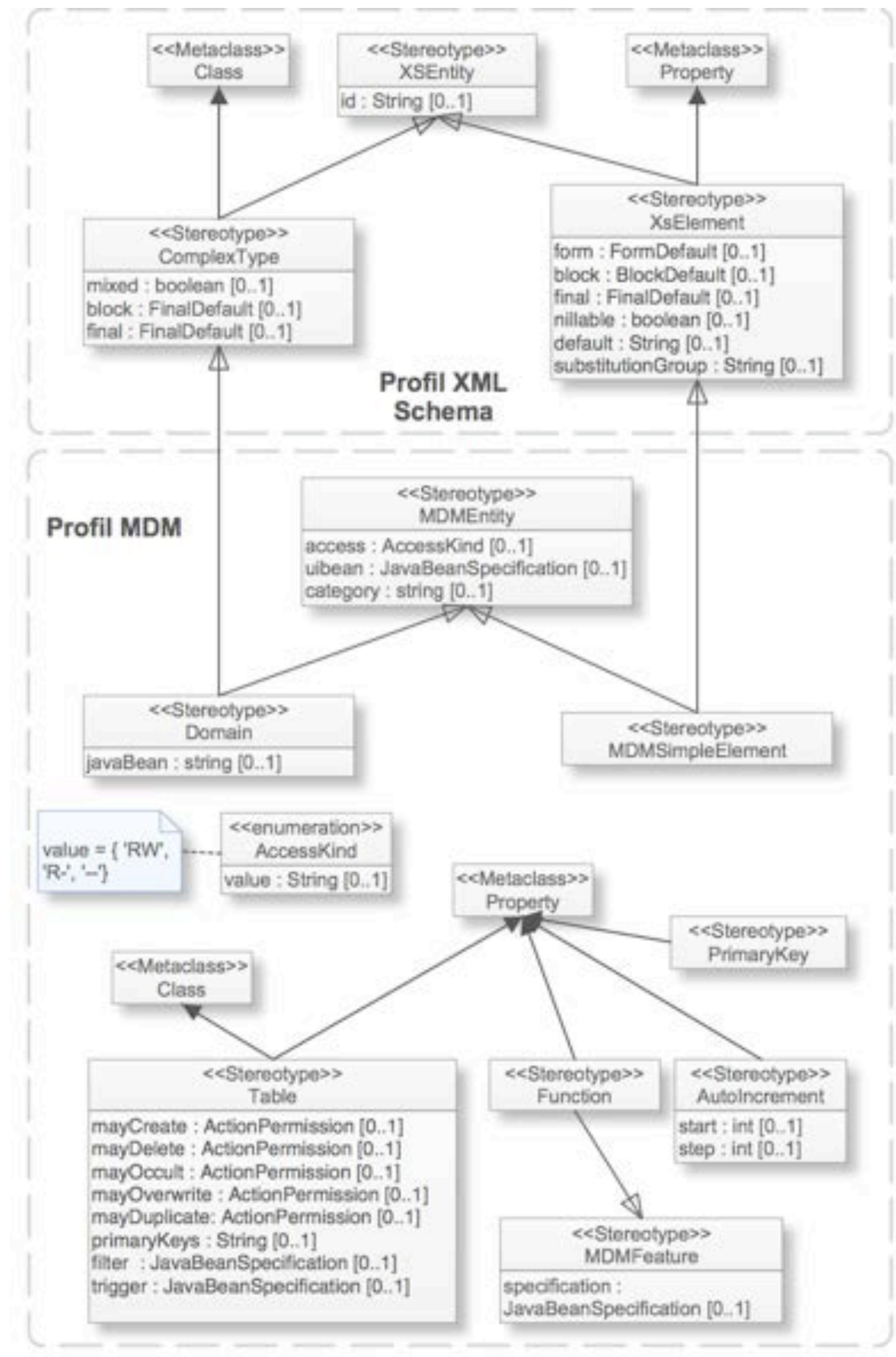


Figure 5.50. Extrait du profil MDM : éléments avancés d'un modèle d'adaptation.

5.4.2.3.1 Stéréotype « MDMEntity »

- Description : ce stéréotype générique définit un élément abstrait au sein d'un modèle d'adaptation. Ce type d'élément possède des propriétés permettant de répondre à des besoins métier de manipulation et de présentation de données.
- Propriétés :
  - Access: définit les propriétés d'accès d'un élément dans un modèle d'adaptation. Cette propriété permet d'indiquer qu'un nœud est accessible soit en lecture ou en lecture/écriture.
  - Uibean : définit un composant de saisie graphique personnalisé. Par défaut dans notre solution MDM, l'interface utilisateur est automatiquement générée à partir de la structure d'un modèle d'adaptation. Cette propriété permet de définir un composant graphique personnalisé défini par une classe Java.
  - Category : cette propriété permet de définir des catégories d'éléments dans un modèle d'adaptation. La définition de catégories peut être utilisée pour filtrer les éléments d'un modèle.

5.4.2.3.2 Stéréotype « Domain »

- Description : définit un élément complexe possédant des propriétés spécifiques à un master data.
- Métaclasses étendues : Class,
- Stéréotypes étendus : <<MDMEntity>>, <<ComplexType>>
- Propriétés :
  - Propriétés héritées des stéréotypes <<MDMEntity>> et <<ComplexType>>.
  - Javabean : Il est possible d'instancier une classe Java particulière lors de l'instanciation d'un type complexe. Ceci se fait par l'ajout de l'attribut *osd:class* dans la définition du nœud complexe.
- Contraintes : ce stéréotype peut uniquement être combiné au stéréotype <<Table>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType name="monDomaine" osd:access="RW"
        osd:class="com.foo.monJavaBean">
        <xs:annotation>
            <xs:appinfo>
                <osd:uiBean class="com.foo.monUIBean"/>
            </xs:appinfo>
        </xs:annotation>
    <xs:sequence>…</xs:sequence>
```

```
        </xs: complexType >
        …
    </xs:schema>
```

Figure 5.51. Définition d'un domaine.

5.4.2.3.3 Stéréotype « MDMSimpleElement »

- Description : définit un élément simple possédant des propriétés spécifiques à un Master Data.
- Métaclasses étendues : Property,
- Stéréotypes étendus : <<MDMEntity>>, <<ComplexType>>
- Propriétés :
  - Propriétés héritées des stéréotypes <<MDMEntity>> et <<XsElement>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="monElement" type="xs:string"
    osd:category="maCatégorie">
      <xs:annotation>
        <xs:appinfo>
          <osd:uiBean class="com.bar.monUIBean"/>
        </xs:appinfo>
      </xs:annotation>
  </xs:element>
  …
</xs:schema>
```

Figure 5.52. Définition d'un élément simple avancé.

5.4.2.3.4 Stéréotype « Table »

- Description : définit un élément complexe qui devra être interprété comme une table au sens SGBD.
- Métaclasses étendues : Class, Property
- Propriétés :
  - Propriétés héritées des stéréotypes <<MDMEntity>> et <<ComplexType>>.
  - PrimaryKeys : spécifie les éléments composant la clé primaire de la table.

- Filter : spécifie un filtre sur la table. Le filtre est défini programmatiquement par une classe Java.
- Trigger : associe à la table des méthodes qui seront exécutées automatiquement par une classe Java lorsque certaines opérations sont effectuées telles que les opération de création, mise à jour, suppression, etc.
- MayCreate : spécifie un droit de création d'enregistrements dans la table.
- MayDelete : spécifie un droit de suppression d'enregistrements dans la table.
- MayOccult : spécifie un droit d'occultation d'enregistrements hérités dans une adaptation fille.
- MayOverwrite : spécifie un droit de surcharge d'enregistrements hérités dans une adaptation fille.
- MayDuplicate : spécifie un droit de duplication d'enregistrements dans la table.
- Contraintes : ce stéréotype peut uniquement être combiné au le stéréotype <<*Domain*>>.

• Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="maTable" maxOccurs="unbounded">
    <xs:annotation>
      <xs:appinfo>
        <osd:table>
          <primaryKeys>/col1 /col2</primaryKeys>
          <mayDuplicate>never</mayDuplicate>
        </osd:table>
      </xs:appinfo>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="col1" type="xs:string"/>
      <xs:element name="col1" type="xs:string"/>
      <xs:element name="col3" type="xs:string"/>
    </xs:sequence>
  </xs: complexType >
  …
</xs:schema>
```

Figure 5.53. Définition d'une table.

5.4.2.3.5 Stéréotype « PrimaryKey »

- Description : indique que l'élément associé fait partie de la clé primaire d'une table.
- Métaclasses étendues : Property,
- Contraintes : ce stéréotype peut uniquement être défini sur des propriétés appartenant à une classe de type <<Table>>.

5.4.2.3.6 Stéréotype « Function »

- Description : indique que la valeur de l'élément doit être alimentée de manière spécifique par une classe Java. Par exemple, une valeur peut provenir d'un système de persistance externe (base de données relationnelle, système central, etc.). Elle peut aussi être le résultat d'un calcul.
- Métaclasses étendues : Property,
- Stéréotypes étendus : << MDMFeature >>
- Propriétés :
  - Propriétés héritées du stéréotype <<MDMFeature>>
- Contraintes : ce stéréotype peut uniquement être appliqué à des éléments simples.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="monElement" type="xs:integer">
        <xs:annotation>
            <xs:appinfo>
                <osd:function class="com.bar.maFonction"/>
            </xs:appinfo>
        </xs:annotation>
    </xs:element>
    ...
</xs:schema>
```

Figure 5.54. Définition d'un élément dont la valeur est calculée.

5.4.2.3.7 Stéréotype « AutoIncrement »

- Description : indique que la valeur de l'élément est auto-incrémentée.
- Métaclasses étendues : Property,
- Propriétés :
  - Start : indique la valeur de départ de l'incrément,
  - Step : indique le pas pour la prochaine valeur de l'incrément.

- Contraintes : ce stéréotype peut uniquement être appliqué à des éléments de type *int* ou *integer* à l'intérieur d'une table, et peut uniquement être combiné au stéréotype <<PrimaryKey>>.
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name=" monElement" type="xs:integer">
        <xs:annotation>
            <xs:appinfo>
                <osd:autoIncrement>
                    <start>1</start>
                    <step>2</step>
                </osd:autoIncrement>
            </xs:appinfo>
        </xs:annotation>
    </xs:element>
    …
</xs:schema>
```

Figure 5.55. Définition d'un élément auto-incrémenté.

5.4.3 Contraintes avancées

Dans cette section nous présentons les éléments de notre profil permettant d'associer des contraintes avancées et dynamiques à des nœuds d'un modèle d'adaptation. La figure 5.56 présente la partie du profil MDM définissant les contraintes avancées.

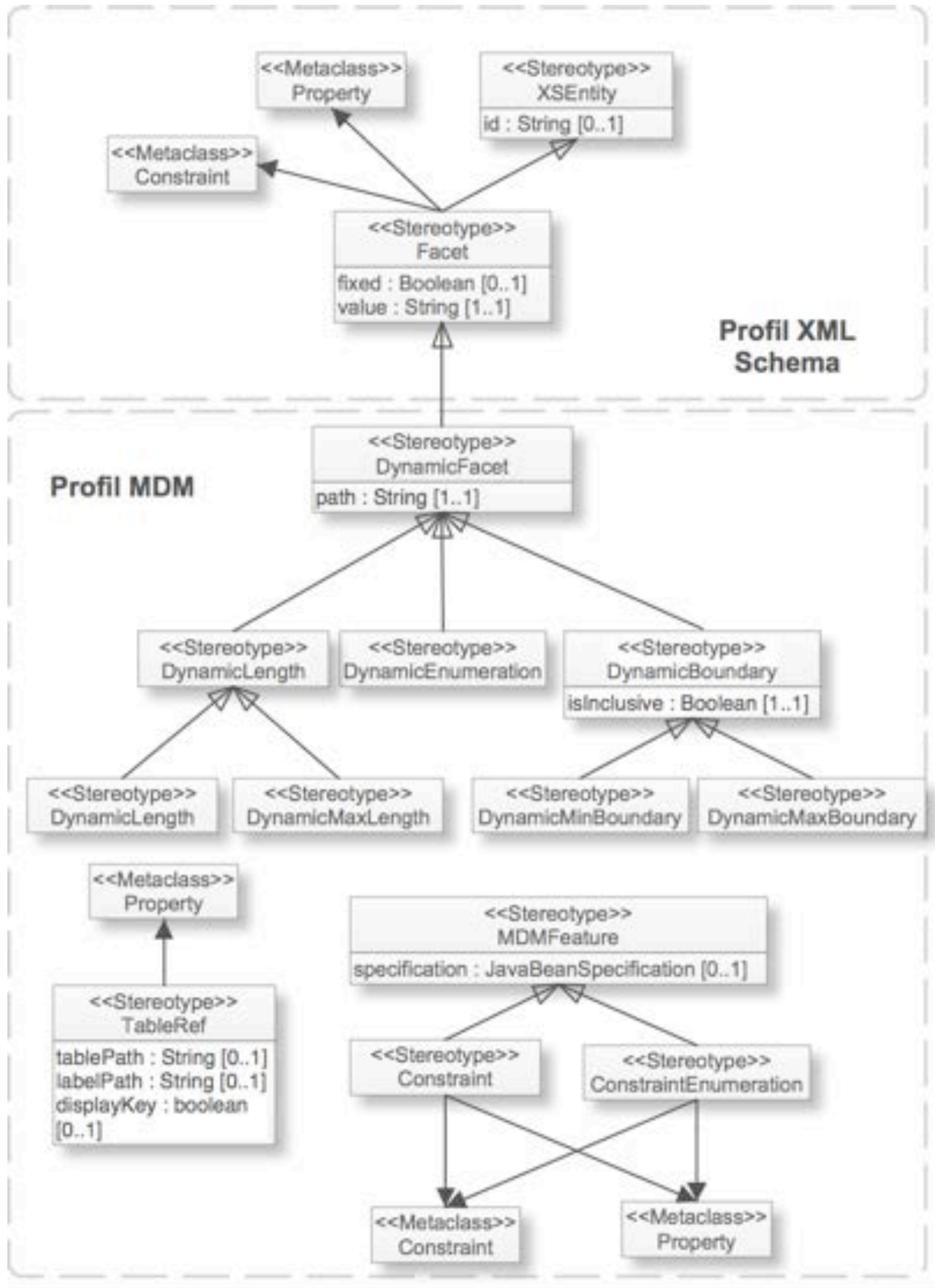


Figure 5.56. Extrait du profil MDM : contraintes avancées.

5.4.3.1 Stéréotype « DynamicFacet »

- Description : ce stéréotype abstrait définit une contrainte dynamique. Une contrainte dynamique supporte le référencement de la valeur d'un nœud. Il est ainsi possible de modifier les contraintes du modèle de données dynamiquement dans les adaptations.
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<Facet>>,
- Propriétés :
  - path : spécifie le chemin du nœud référencé.

5.4.3.2 Stéréotype « DynamicEnumeration »

- Description : définit une énumération dynamique alimentée par un nœud du modèle,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<DynamicFacet>>,
- Propriétés : propriétés héritées du stéréotype <<DynamicFacet>>,
- Contraintes : le nœud référencé doit être une énumération,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="tailleVetement" type="xs:string">
        <xs:annotation>
            <xs:appinfo>
                <osd:otherFacets>
                    <osd:enumeration path="chemin_enumeration"/>
                </osd:otherFacets>
            </xs:appinfo>
        </xs:annotation>
    </xs:element>
</xs:schema>
```

Figure 5.57. Enumération dynamique.

5.4.3.3 Stéréotype « DynamicLength »

- Description : définit une contrainte dynamique sur la longueur de la valeur d'un élément,

- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<DynamicFacet>>,
- Propriétés : propriétés héritées du stéréotype <<DynamicFacet>>,
- Contraintes : le nœud référencé doit être un entier,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="tailleVetement" type="xs:string">
        <xs:annotation>
            <xs:appinfo>
                <osd:otherFacets>
                    <osd:length path="chemin_nœud_référencé"/>
                </osd:otherFacets >
            </xs:appinfo>
        </xs:annotation>
    </xs:element>
</xs:schema>
```

Figure 5.58. Contrainte dynamique sur la longueur d'une valeur.

5.4.3.4 Stéréotype « DynamicMinLength »

- Description : définit une contrainte dynamique sur la longueur minimale de la valeur d'un élément,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<DynamicFacet>>,
- Propriétés : propriétés héritées du stéréotype <<DynamicFacet>>,
- Contraintes : le nœud référencé doit être un entier,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="tailleVetement" type="xs:string">
        <xs:annotation>
            <xs:appinfo>
                <osd:otherFacets>
                    <osd:minLength path="chemin_nœud_référencé"/>
                </osd:otherFacets >
```

```
                </xs:appinfo>q
            </xs:annotation>
        </xs:element>
    </xs:schema>
```

Figure 5.59. Contrainte dynamique sur la longueur minimale d'une valeur.

5.4.3.4 Stéréotype « DynamicMaxLength »

- Description : définit une contrainte sur la longueur maximale de la valeur d'un élément,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<DynamicFacet>>,
- Propriétés : propriétés héritées du stéréotype <<DynamicFacet>>,
- Contraintes : le nœud référencé doit être un entier,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="tailleVetement" type="xs:string">
        <xs:annotation>
            <xs:appinfo>
                <osd:otherFacets>
                    <osd:maxLength path="chemin_nœud_référencé"/>
                </osd:otherFacets >
            </xs:appinfo>
        </xs:annotation>
    </xs:element>
</xs:schema>
```

Figure 5.60. Contrainte dynamique sur la longueur maximale d'une valeur.

5.4.3.5 Stéréotype « DynamicBoundary »

- Description : stéréotype abstrait définissant une borne dynamique appliquée à la valeur d'un élément,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<DynamicFacet>>,
- Propriétés :

- Propriétés héritées du stéréotype <<DynamicFacet>>,
- IsInclusive : indique si la borne est incluse dans la restriction.

5.4.3.6 Stéréotypes « Dynamic{Min/Max}Boundary»

- Description : définit une borne minimale ou maximale appliquée à la valeur d'un élément,
- Méta-classes étendues : Property, Constraint
- Stéréotypes étendus : <<DynamicFacet>>,
- Propriétés : propriétés héritées du stéréotype <<DynamicFacet>>,
- Contraintes : le nœud référencé doit être de type numérique,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="size">
        <xs:annotation>
            <xs:appinfo>
                <osd:otherFacets>
                    <osd:maxInclusive path="chemin_nœud_référencé"/>
                    <osd:minExclusive path="chemin_nœud_référencé"/>
                </osd:otherFacets >
            </xs:appinfo>
        </xs:annotation>
    </xs:element>
</xs:schema>
```

Figure 5.61. Contraintes dynamiques permettant de borner la valeur d'un élément.

5.4.3.7 Stéréotypes « Constraint»

- Description : spécifie une contrainte définie par une classe Java,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<MDMFeature>>,
- Propriétés : propriétés héritées du stéréotype <<MDMFeature>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="size">
```

```
            <xs:annotation>
                <xs:appinfo>
                    <osd:otherFacets>
                        <osd:constraint class="com.foo.maContrainte"/>
                    </osd:otherFacets >
                </xs:appinfo>
            </xs:annotation>
        </xs:element>
    </xs:schema>
```

Figure 5.62. Définition d'une contrainte définie programmatiquement.

5.4.3.8 Stéréotypes « ConstraintEnumeration»

- Description : spécifie une énumération alimentée par une classe Java,
- Métaclasses étendues : Property, Constraint
- Stéréotypes étendus : <<MDMFeature>>,
- Propriétés : propriétés héritées du stéréotype <<MDMFeature>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="size">
        <xs:annotation>
          <xs:appinfo>
              <osd:otherFacets>
                  <osd:constraintEnumeration
                    class="com.foo.monEnumeration"/>
              </osd:otherFacets >
          </xs:appinfo>
        </xs:annotation>
    </xs:element>
</xs:schema>
```

Figure 5.63. Définition d'une énumération définie programmatiquement.

5.4.4 Documentation avancée

Nous définissons dans notre solution MDM des éléments de documentation avancés permettant de spécifier des informations additionnelles qui seront exploitées pour l'affichage de nœuds dans notre outil EBX.Manager. La figure 5.64 présente la partie du profil MDM définissant les éléments de documentation avancée.

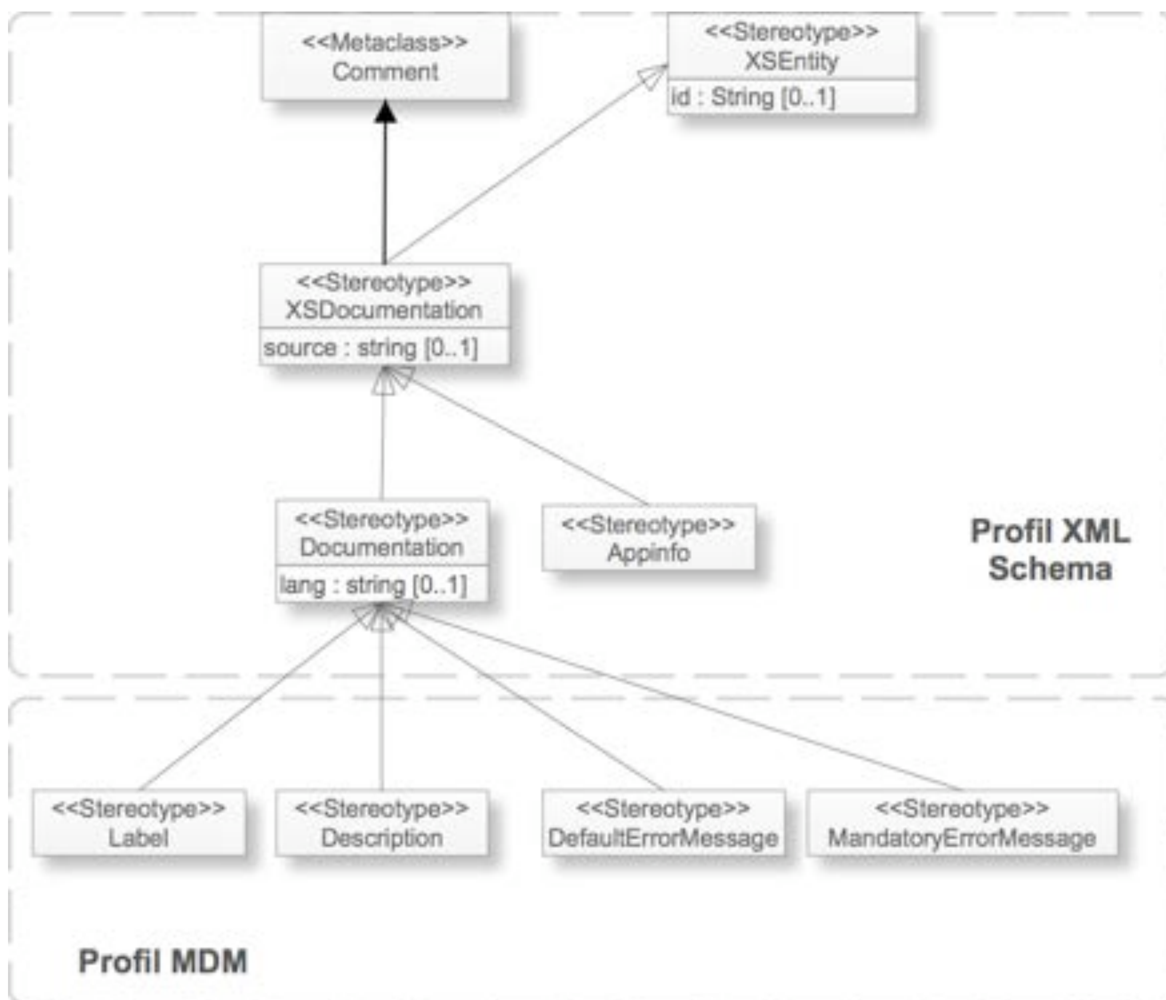


Figure 5.64. Extrait du profil MDM : documentation avancée.

5.4.4.1 Stéréotypes «Label»

- Description : définit un libellé personnalisé et pouvant être localisé,
- Métaclasses étendues : Comment,
- Stéréotypes étendus : <<Documentation>>,
- Propriétés : propriétés héritées du stéréotype <<Documentation>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="size">
        <xs:annotation>
            <xs:documentation>
                <osd:label xml:lang="fr_FR">taille</osd:label>
                <osd:label xml:lang="en_US">size</osd:label>
            </xs:documentation>
        </xs:annotation>
    </xs:element>
</xs:schema>
```

Figure 5.65. Libellé localisé d'un nœud dans un modèle d'adaptation.

5.4.4.2 Stéréotypes «Description»

- Description : définit une description personnalisée et pouvant être localisée,
- Métaclasses étendues : Comment,
- Stéréotypes étendus : <<Documentation>>,
- Propriétés : propriétés héritées du stéréotype <<Documentation>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="size">
      <xs:annotation>
         <xs:documentation>
            <osd:description xml:lang="fr_FR">ce nœud représente la
             taille d'un vêtement.
            </osd:description>
            <osd:description xml:lang="en_US">this node represents a
            wear size.
```

```
                </osd:description>
            </xs:documentation>
        </xs:annotation>
    </xs:element>
</xs:schema>
```

Figure 5.66. Description localisée d'un nœud dans un modèle d'adaptation.

5.4.4.3 Stéréotypes «DefaultErrorMessage»

- Description : définit un message d'erreur localisé associé à une contrainte,
- Métaclasses étendues : Comment,
- Stéréotypes étendus : <<Documentation>>,
- Propriétés : propriétés héritées du stéréotype <<Documentation>>,
- Contraintes : peut être uniquement défini sur un élément définissant une contrainte XML Schema ou dynamique,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name ="zipCode">
        <xs:simpleType>
            <xs:restriction base ="xs:string">
                <xs:minInclusive value ="01000">
                    <xs:annotation>
                        <xs:documentation>
                            <osd:defaultErrorMessage xml:lang="en_US">
                            Postal code not valid.
                            </osd:defaultErrorMessage>
                        </xs:documentation>
                    </xs:annotation>
                </xs:minInclusive>
            </xs:restriction>
        </xs:simpleType>
    </xs:element>
    …
</xs:schema>
```

Figure 5.67. Message d'erreur portant sur des contraintes.

5.4.4.4 Stéréotypes «MandatoryErrorMessage»

- Description : définit un message d'erreur localisé associé à une contrainte de cardinalité. Si le nœud spécifie l'attribut *minOccurs*="1" (comportement par défaut), alors un message d'erreur *obligatoire* est affiché si l'utilisateur ne renseigne pas le champ.
- Métaclasses étendues : Comment,
- Stéréotypes étendus : <<Documentation>>,
- Propriétés : propriétés héritées du stéréotype <<Documentation>>,
- Exemple XML Schema :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="size">
        <xs:annotation>
            <xs:documentation>
                <osd:mandatoryErrorMessage xml:lang="fr_FR">
                 le champs doit être saisi.
                </osd:mandatoryErrorMessage>
            </xs:documentation>
        </xs:annotation>
    </xs:element>
</xs:schema>
```

Figure 5.68. Message d'erreur sur une saisie obligatoire.

5.5 Conclusion

Dans ce chapitre nous avons présenté la première étape permettant d'introduire une approche d'Ingénierie Dirigée par les Modèles visant à optimiser et standardiser la définition de modèles de données. Nous avons introduit une couche d'abstraction permettant de représenter un modèle d'adaptation indépendamment d'un domaine d'application par l'intermédiaire de deux profils à savoir un profil XSD Schema et un profil MDM. L'enrichissement sémantique que nous avons réalisé à la fois dans XML Schema et UML dans le but d'établir des correspondances entre ces deux technologies nous permet de spécifier des mappings entre UML et XML Schema. Dans notre approche IDM, la définition de mappings permet d'assurer le passage de la solution fonctionnelle (UML) à la solution technologique (XML Schema).

L'approche que nous avons suivie par la définition de deux profils distincts exprimés en UML est applicable de manière générale à toute modélisation de modèles XML Schema et s'applique aussi au domaine spécialisé du Master Data Management. Couplée à des méthodes de transformation que nous présenterons dans le chapitre suivant, l'utilisation de nos profils UML permet de s'abstraire de toute spécificité technique liée à la définition des modèles XML Schema appliqués au MDM.

Chapitre 6

D’un modèle contemplatif à un modèle productif

Résumé. Dans le chapitre précédent nous avons mis en application une métamodélisation UML appliquée à la définition de modèles XML Schema et au domaine du Master Data Management. Cette métamodélisation UML est définie à l’aide de deux profils UML dédiés respectivement à la définition de modèles XML Schéma génériques et à la définition de modèles d’adaptation. Dans ce chapitre nous présenterons les règles d'appariements ou *mappings* permettant de produire un modèle XML Schema à partir d’un diagramme de classes UML générique ou spécialisé par l’intermédiaire de nos profils UML.

6.1 Introduction

L'Ingénierie Dirigée par les Modèles (IDM) que nous avons présentée dans le chapitre 4 est une approche de conception qui vise à conserver les modèles comme point de référence dans un processus de développement logiciel. L’un des points clé d’une approche IDM est la possibilité de transformer des modèles d’un espace de modélisation ou d’un niveau d’abstraction vers un autre. En effet, la gestion manuelle des modèles s’avère coûteuse et les modèles sont bien souvent délaissés dans les phases de conception. Il devient donc nécessaire de proposer des mécanismes de gestion automatique de modèles permettant, tout au long de leur cycle de vie de les représenter sous des formes par lesquelles ils répondront au mieux aux attentes des utilisateurs.

Dans une approche IDM, il est possible d’adapter des modèles à différentes plateformes spécifiques par l’intermédiaire de transformations ou de générer un modèle technique à partir d’un modèle abstrait. C’est sur ce dernier point que nous avons focalisé nos travaux. En effet, dans le chapitre précédent, nous avons mis en application une métamodélisation UML permettant à la fois de représenter des modèles XML Schema et des modèles liés au domaine du Master Data Management. En utilisant les profils UML que nous avons définis dans le chapitre précédent, nous avons la capacité de définir d’une manière abstraite de tels modèles. Un diagramme de classes UML demeure dans le domaine du contemplatif dans la mesure où il représente de manière abstraite la sémantique d’un

domaine. Afin d'obtenir un modèle exploitable techniquement et productif, nous devons être en mesure de générer automatiquement les modèles spécifiques liés aux modèles abstraits. La génération de modèles est possible à l'aide de règles de transformation. Pour définir des règles de transformation entre deux modèles il faut préalablement établir des appariements ou *mappings* entre les métamodèles impliqués dans les processus de transformation. De nombreux travaux sur les problématiques de mappings entre métamodèles ont été menés. [Levendovszky, 2002] définit le mapping comme étant un ensemble de règles de transformation de modèles permettant de traduire des instances d'un métamodèle source en instance d'un métamodèle cible.

[Baïna *et al.*, 2006] se basent sur les travaux de [Kalfoglou *et al.*, 2003] pour apporter une définition mathématique à l'interopérabilité des applications dans un système d'entreprise. Après étude de ces travaux, nous nous apercevons que cette définition peut aussi s'appliquer aux mapping de modèles. Considérons A et B deux métamodèles ; A et B sont dits interopérables si et seulement s'il existe un mapping bijectif de M_A vers M_B, que nous noterons f. La bijection de f nous assure que nous pouvons construire une instance du modèle B à partir de l'instanciation du modèle de A (en utilisant f) et construire une instance du modèle A à partir de l'instanciation du modèle de B. A partir de cette définition, trois niveaux d'interopérabilité sont définis [Baïna *et al.*, 2006] et des mappings sont identifiés entre les langages A et B à savoir :

- Niveau 2 : Il existe un isomorphisme total entre M_A et M_B. De ce fait, tout concept de M_A a son équivalent dans M_B et inversement, ce qui signifie que M_A et M_B sont équivalents ;
- Niveau 1 : Il existe un isomorphisme partiel entre M_A et M_B. Il existe donc une sous-partie de M_A que l'on notera M'_A et une sous-partie de M_B (M'_B) telles que l'interopérabilité entre M'_A et M'_B est de niveau 2, ces sous-parties sont donc équivalentes ;
- Niveau 0 : Il n'existe pas d'isomorphisme partiel entre M_A et M_B. Cependant, il se peut que des mappings non bijectifs existent entre M_A et M_B. Dans ce cas, nous ne pouvons pas parler d'interopérabilité sémantique en A et B.

L'interopérabilité de niveau 2 est la plus difficile à établir car il est très rare que deux métamodèles soient totalement équivalents. Dans ce chapitre, nous établirons une correspondance partielle entre M_A et M_B en calculant la proportion d'équivalence de M'_A et M'_B par rapport à M_A et M_B obtenant ainsi une interopérabilité de niveau 1 entre UML et XML Schema (figure 6.1).

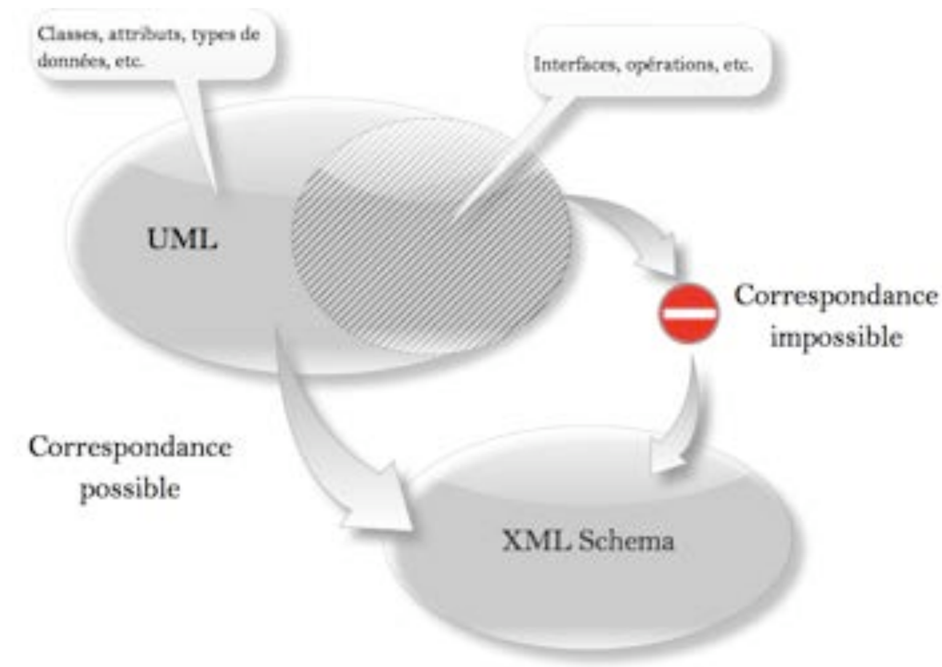


Figure 6.1. Mapping de niveau 1 entre UML et XML Schema.

Dans le cadre de mappings entre UML et XML Schema, nous pouvons citer les travaux de [Carlson, 2001] qui aborde la transformation de modèles UML en schéma XML et sa réciproque, obtenant ainsi un mapping bijectif entre ces deux formalismes. Le mapping réalisé utilise comme fondement un profil UML définissant des concepts spécifiques à XML Schema. Ce profil est utilisé afin d'étendre la sémantique d'un modèle UML à la sémantique d'un modèle XML Schema. Cette approche permet un mapping d'une grande partie des concepts introduits par XML Schema, mais ne prend pas en compte certains concepts tels que la notion de groupe (*list*, *union*), de contrainte d'identité (*key*, *keyref*, *unique*), de contrainte de généricité (*substitutionGroup*), etc. De plus, certains concepts importants d'UML tels que l'agrégation, la composition, l'association et la documentation ne sont pas pris en compte lors d'une transformation d'un modèle UML vers un modèle XML Schema. Ce mapping a été mis en application par HyperModel [Carslon, 2006]. HyperModel est un plug-in pour l'IDE Eclipse implémentant le mapping bijectif UML/XML. Cet outil est fonctionnellement opérationnel mais souffre de quelques limitations lors d'une transformation d'un modèle UML vers un modèle XML Schema car :

- Certains concepts ne sont pas appariés (agrégation, composition, etc.) ;
- Des éléments sont appariés plusieurs fois entraînant une redondance d'information et une inconsistance dans le modèle résultat ;
- Une perte d'information, notamment concernant les contraintes de cardinalité, se produit sur certains modèles ;

- Des modèles XML Schemas générés ne sont pas valides au regard de la spécification du W3C.

[Routledge *et al.*, 2002] abordent le mapping de manière traditionnelle entre UML et XML Schema par l'intermédiaire de l'approche à trois niveaux issus du monde des bases de données à savoir les niveaux conceptuel, logique et physique. Dans le contexte d'un diagramme de classes UML, le niveau conceptuel décrit les objets et leurs relations. Le niveau logique représente les structures de données XML Schema sous la forme d'un profil UML. Le niveau physique représente directement le modèle XML Schema. De la même manière que les travaux de Carlson, certains éléments spécifiques d'UML tels que l'agrégation, la composition et d'autres ne sont pas pris en compte.

D'autres travaux ont été réalisés dans le même contexte par [Conrad *et al.*, 2000], [Wu et Hsieh, 2002] et [Kurtev *et al.*, 2003] mais souffrent également des mêmes limitations que les travaux que nous avons présentés précédemment.

Notre objectif est de combler les limitations de ces travaux. Dans la section 6.2 nous présenterons les mappings que nous avons établis entre éléments UML et XML Schema, y compris les notions décrivant des relations de dépendance. Dans la section 6.3 nous implémenterons les règles de transformation issues des mappings entre les entités UML et XML Schema. Dans la section 6.4 nous mettrons en application ces règles de transformation par la définition d'une application spécifique de modélisation.

6.2 Spécification des mappings UML / XML Schema

Nous avons défini dans le chapitre précédent deux profils UML permettant de définir des modèles XML Schema génériques ou soit appliqués au domaine du Master Data Management. Les mappings issus de ces profils revêtent un aspect trivial dans la mesure où lors de leur construction nous avons adopté un formalisme de description liant les entités XML Schema (et MDM) aux entités UML. Il est cependant utile de décrire les mappings par défaut entre un modèle UML générique et un modèle XML Schema. Dans ce contexte, chaque stéréotype que nous avons introduit permettra d'une part d'agrémenter les mappings par défaut et modifiera sensiblement la manière dont sont transformés les éléments ; d'autre part, chaque stéréotype ajoutera ou non des métadonnées qui définissent les caractéristiques supplémentaires des entités XML Schema.

Dans cette section, nous présentons les mappings par défaut entre un modèle UML et un modèle XML schéma.

6.2.1 Paquetages UML

Chaque paquetage UML est transformé vers un modèle XML Schema. Lorsqu'un paquetage fait référence à des éléments provenant d'un autre paquetage alors la déclaration *xs:include* est utilisée. Par défaut, le nom du modèle XML Schema généré sera le même que celui du paquetage.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:import namespace="http://www.example.com/namespace"
      schemaLocation="./autrePaquetage.xsd"/>
  . . .
</xs:schema>
```

Figure 6.2. Transformation d'un paquetage UML.

6.2.2 Classes UML

Une classe UML est associée à un élément XML *<xs:element>* ayant une structure complexe figure 6.3.

```
<xs:element name="Classe">
   <xs:complex>
      <xs:sequence>
          …
      </xs:sequence>
```

```
        </xs:complex>
    </xs:element>
```

Figure 6.3. Transformation d’une classe UML.

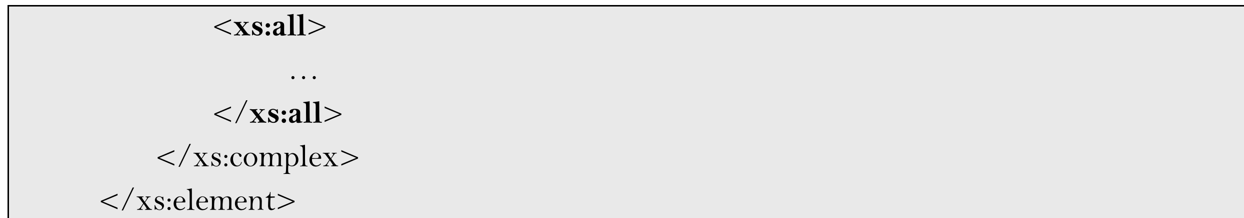
Certains stéréotypes que nous avons définis dans notre profil permettent d’ajouter des informations supplémentaires sur les classes. Ainsi, le stéréotype <<ComplexType>> permet d’indiquer que la classe UML doit être transformée en un élément de type complexe global au schéma XML *<xs:complexType>* (figure 6.4).

```
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:complexType name="ClasseComplexe">
            <xs:sequence>
                ...
            </xs:sequence>
        </xs:complexType>
    </xs:schema>
```

Figure 6.4. Transformation d’une classe UML en un élément complexe global XML Schema.

Les stéréotypes <<*Sequence*>>, <<*Choice*>> et <<*All*>>, permettent, par exemple, d’indiquer que la classe doit être transformée en un élément complexe XML ayant respectivement une structure de la forme *<xs:sequence>*, *<xs:choice>* ou *<xs:all>* (figure 6.5).

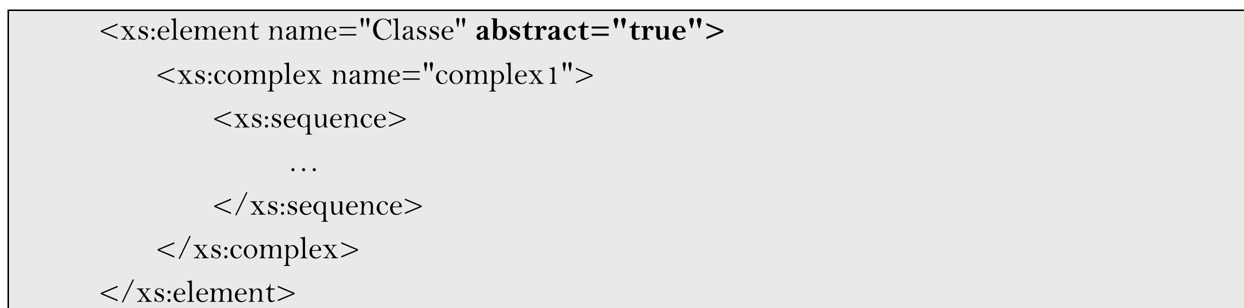
```
    <xs:element name="Classe">
        <xs:complex name="complex1">
```



```
            <xs:all>
                  …
            </xs:all>
      </xs:complex>
</xs:element>
```

Figure 6.5. Structure d'un élément complexe XML Schema.

De plus, si une classe UML est déclarée comme étant abstraite, cette propriété est mappée par l'attribut *'abstract=true'* dans l'élément XML Schema correspondant (figure 6.6).

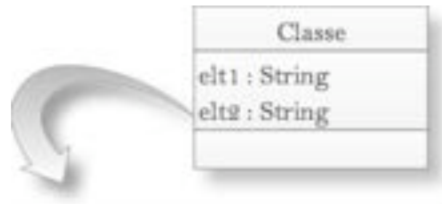


```
<xs:element name="Classe" abstract="true">
      <xs:complex name="complex1">
            <xs:sequence>
                  …
            </xs:sequence>
      </xs:complex>
</xs:element>
```

Figure 6.6. Transformation d'une classe abstraite UML.

6.2.3 Attributs

Les attributs UML sont transformés en éléments simples XML *<xs:element>* ayant pour valeur d'attribut *<xs:type>* un type de données. L'élément XML généré est englobé soit par un élément *<xs:sequence>*, *<xs:any>*, *<xs:choice>* ou *<xs:all>* selon le stéréotype appliqué à la classe (figure 6.7).




```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="Classe">
        <xs:complex>
            <xs:sequence>
                <xs:element name="elt1" type="xs:string"/>
                <xs:element name="elt2" type="xs:string"/>
            </xs:sequence>
        </xs:complex>
    </xs:element>
</xs:schema>
```

Figure 6.7. Transformation d'attributs UML vers des éléments simples XML Schema.

L'utilisation du stéréotype <<*Attribute*>> permet de spécifier que l'attribut UML doit être transformé en attribut XML <*xs:attribute*> et non en élément <*xs:element*> (figure 6.8).

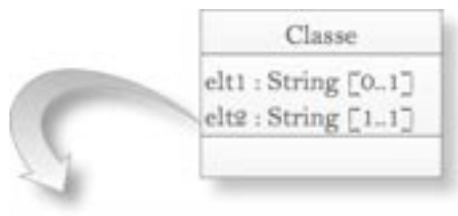
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="Classe">
        <xs:complex>
            <xs:sequence>
                <xs:element name="elt1" type="xs:string"/>
            </xs:sequence>
            <xs:atribute name="att1" type="xs:string">
        </xs:complex>
    </xs:element>
```

```
</xs:schema>
```

Figure 6.8. Transformation d'attributs UML en des éléments attributs XML Schema.

6.2.4 Cardinalités

Les cardinalités UML sont transformées en attributs *<xs:minOccurs>* et *<xs:maxOccurs>* portés par l'élément correspondant. Les cardinalités peuvent porter sur les associations, les agrégations, les compositions et les attributs (figure 6.9).

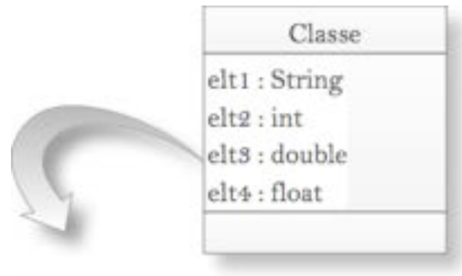



```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="Classe">
        <xs:complex>
            <xs:sequence>
                <xs:element name="elt1" type="xs:string" minOccurs="0"
                 maxOccurs="1"/>
                <xs:element name="elt2" type="xs:string" minOccurs="1"
                 maxOccurs="1"/>
            </xs:sequence>
        </xs:complex>
    </xs:element>
</xs:schema>
```

Figure 6.9. Transformation de cardinalités UML en cardinalités XML Schema.

6.2.5 Types de données natifs

Les types de données natifs UML tels que *int*, *double*, *float*, *string*, etc., sont mappés en entités XML correspondantes, soit respectivement *<xs:integer>*, *<xs:double>*, *<xs:float>*, *<xs:string>*, etc. (figure 6.10).

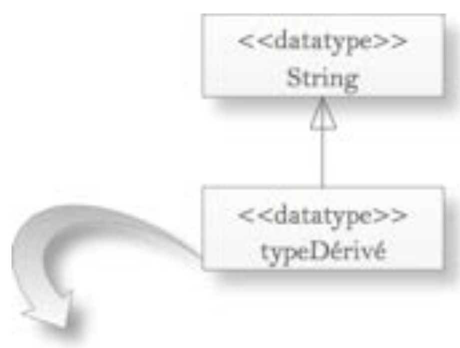



```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="Classe">
        <xs:complex>
          <xs:sequence>
            <xs:element name="elt1" type="xs:string"/>
            <xs:element name="elt2" type="xs:integer"/>
            <xs:element name="elt3" type="xs:double"/>
            <xs:element name="elt4" type="xs:float"/>
          </xs:sequence>
        </xs:complex>
    </xs:element>
</xs:schema>
```

Figure 6.10. Transformation de types de données natifs UML.

6.2.6 Types dérivés

Un type dérivé permet de spécifier des contraintes sur des types ou de définir de nouveaux types métier. Les types dérivés sont mappés en éléments XML Schema *<xs:simpleType>* (figure 6.11).



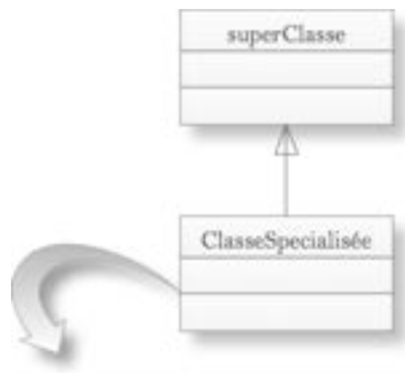
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:simpleType name="typeDérivé">
                <xs:restriction base="xs:string">
        </xs:simpleType>
</xs:schema>
```

Figure 6.11. Transformation d'un type de données dérivé UML.

Dans notre mapping, nous incluons les contraintes XML sur les types. Ainsi est il possible pour un type dérivé de spécifier des contraintes conformes à la spécification du W3C telles que *length*, *min/max length*, *enumeration*, *fractionDigits*, *totalDigits*, *min/max Inclusive* et *min/max Exclusive*.

6.2.7 Généralisation / Spécialisation

La généralisation et la spécialisation entre deux classes UML sont mappés en éléments XML Schema *<xs:complexContent>* et *<xs:extension>* (figure 6.12).




```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="superClasse">
            <xs:complex>
                <xs:sequence>
                    …
                 </xs:sequence>
            </xs:complex>
      </xs:element>
      <xs:element name="ClasseSpécialisée">
```

```
            <xs:complexContent>
                <xs:extension base= superClasse>
                    <xs:sequence>
                    …
                    </xs:sequence>
                </xs:extension>
            </xs:complexContent>
        </xs:element>
    </xs:schema>
```

Figure 6.12. Transformation des propriétés de généralisation UML.

6.2.8 Agrégation et composition

Nous avons défini dans le chapitre précédent des extensions XML permettant de matérialiser des relations « objet ». Nous utilisons les extensions que nous avons introduites pour traduire en XML Schema l’agrégation (respectivement composition) par un élément possédant l’extension suivante :

```
    <xs:annotation>
      <xs:appinfo>
            <osd:aggregation />  < !-- respectivement composition -->
      </xs:appinfo>
    </xs:annotation>
```

Les cardinalités UML définies pour les associations, agrégation et composition sont représentées par les attributs XML *<xs:minOccurs>* et *<xs:maxOccurs>* (figure 6.13).

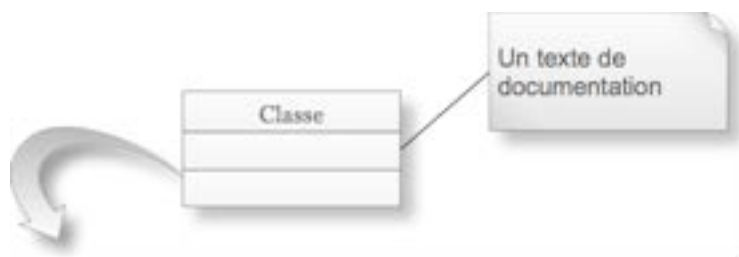
```
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="Foo">
            <xs:complex>
```

```
                    <xs:sequence>
                        <xs:element name="Bar"
                            minOccurs="0" maxOccurs="unbounded">
                            <xs:annotation>
                                <xs:appinfo>
                                        <osd:composition/>
                                </xs:appinfo>
                            </xs:annotation>
                            …
                        </xs:element/>
                    </xs:sequence>
                </xs:complex>
            </xs:element>
        </xs:schema>
```

Figure 6.13. Transformation de relations de dépendance UML.

6.2.9 Notes et documentation

Une documentation ou une note est une variante affaiblie de contrainte ; c'est un texte à l'usage du lecteur mais non exécutable par la machine. Dans un diagramme de classes UML, une note est rattachée à l'élément auquel elle fait référence (figure 6.14).




```
        <xs:element name="Classe">
            <xs:annotation>
                <xs:documentation>
                    Un texte de documentation.
                </xs:documentation>
            </xs:annotation>
```

```
    …
</xs:element>
```

Figure 6.14. Transformation d'éléments de documentation UML.

Par l'intermédiaire de ces mappings et des profils que nous avons définis, il est possible de générer un modèle XML Schema à partir d'un modèle UML. Dans une méthodologie d'Ingénierie Dirigée par les Modèles nous définissons aussi des transformations permettant de réaliser des processus de *reverse engineering*[17]. Pour ce faire, nous implémentons des mappings permettant de générer un modèle XML Schema à partir d'un modèle UML et, inversement, de générer un modèle UML à partir d'un modèle XML Schema.

6.3 Implémentation des mappings

6.3.1 Transformation d'un modèle UML vers un modèle XML Schema

Dans le chapitre 4 de ce mémoire, nous avons présenté différents formalismes de transformation focalisés sur des modèles basés sur l'architecture du MOF. Il est question dans un premier temps d'être capable à partir d'un modèle UML de générer de manière automatique un modèle XML Schema, ou autrement dit de transformer un modèle basé sur une architecture MOF vers un modèle textuel. Parmi les formalismes présentés, nous avons choisi d'utiliser le langage MOFScript (cf. section 3.4.2.2). Notre choix a été motivé par le fait que les langages de transformation tels que QVT sont essentiellement focalisés sur des transformations de type « modèle à modèle » et que nous avons simplement besoin de générer un modèle textuel à partir d'un modèle abstrait ; autrement dit, il s'agit de générer du code à partir d'un modèle.

Dans cette section, nous décrivons les spécificités de MOFScript que nous avons utilisées pour implémenter nos règles de transformation.

6.3.1.1 Association d'un métamodèle source

MOFScript a pour but de transformer un modèle vers du texte. Les modèles sources sont des instances de métamodèles devant être basés sur l'architecture du MOF. Il est

[17] Processus de génération inversée permettant de générer un modèle UML à partir d'un modèle XML Schema.

possible de définir le métamodèle utilisé avec l'instruction *texttransformation*. Dans notre cas, nous avons décidé de nous fonder sur le métamodèle d'UML (figure 6.15).

```
texttransformation exempleDeTransformation (in
                uml:"http://www.eclipse.org/uml2/1.0.0/UML"))
```

Figure 6.15. Instruction MOFScript d'association d'un métamodèle source.

6.3.1.2 Importation de règles de transformation

Il est possible d'importer des règles de transformation, provenant d'autres scripts, en tant que bibliothèques ou en tant qu'extensions du script courant. L'instruction *import* permet de spécifier les bibliothèques à importer dans un script. MOFScript offre différentes syntaxes pour importer des scripts (figure 6.16). Il est ainsi possible d'associer un nom à une instruction d'importation ou de spécifier uniquement le chemin du script à importer.

```
import nomImportation("std/stdLib2.m2t")
import "std/stdLib2.m2t"
```

Figure 6.16. Instruction MOFScript d'importation de règles.

6.3.1.3 Point d'entrée des règles de transformation

Les *points d'entrée* des règles de transformation définissent où commence l'exécution de la transformation définie dans un script. Ce point d'entré est similaire à la méthode *main()* existante en Java. Le point d'entrée est appliqué dans un contexte donné qui indique quel élément du métamodèle représente le point de départ de la transformation à exécuter. Dans la figure 6.17, nous définissons le point d'entrée sur un modèle UML à l'aide de l'instruction *uml.Model::main()*.

```
uml.Model::main ()
{
  self.ownedMember->forEach(p:uml.Package)
  {
    p.transformerPackage()
  }
}
```

Figure 6.17. Point d'entrée d'un script de transformation MOFScript.

Un point d'entrée peut s'appliquer sur plusieurs instances d'une entité d'un métamodèle. Dans ce cas, le point d'entrée sera exécuté pour chaque instance de l'entité spécifiée. La figure 6.18 présente la définition d'un point d'entrée sur toutes les instances de l'entité *Class* du métamodèle UML.

```
uml.Class::main ()
{
 self.transformerClasse()
}
```

Figure 6.18. Point d'entrée MOFScript appliqué aux classes d'un modèle UML.

Dans l'exemple de la figure 6.18, nous exécutons une transformation spécifique pour chaque classe d'un modèle UML.

Il est à noter que des règles de transformation peuvent être définies sans nécessairement spécifier de contextes particuliers. Dans ce cas, le point d'entrée défini sera exécuté une seule fois. Un point d'entrée sans contexte est défini avec ou sans le mot clé *module* (figures 6.19).

```
module::main ()
{
   uml.objectsOfType (uml.Package)
}
OU
main ()
{
 }
```

Figure 6.19. Point d'entrée MOFScript sans contexte.

6.3.1.4 Règles de transformation

Une règle de transformation MOFScript peut être assimilée à une méthode Java dans la mesure où une règle définit un ensemble d'instructions exécutées lorsque la règle est appelée. Une règle peut retourner un objet qui peut être soit un type de données natif MOFScript, soit un type du métamodèle source. La figure 6.20 présente le squelette d'un script permettant de générer un modèle XML Schema à partir d'un modèle UML. L'instruction *file* permet de spécifier le chemin du modèle XML Schema à générer.

```
uml.Package::transformerPackage()
{
      file(self.obtenirNomPaquetage() + ".xsd")
      ….ensemble d'instructions…
      self.ownedMember->forEach(c:uml.Class)
            c.transformerClasse()
}
uml.Class::transformerClasse()
{
      ….ensemble d'instructions…
      self.traiterStereotypes()
      self.ownedAttribute->forEach(p : uml.Property)
      {
            p.transformerAttributs()
      }
      self.transformerAssociations()
      ….ensemble d'instructions…
}
```

Figure 6.20. Règles de transformation MOFScript.

Dans ce script de transformation, nous itérons sur tous les paquetages du modèle UML source. Pour chacun de ces paquetages, nous créons un document XML Schema. Le script traite ensuite les classes d'un paquetage, leurs attributs et leurs associations.

De la même manière que dans les langages de programmation classiques, il est possible de définir des règles retournant une valeur à l'aide de l'instruction *result*. Cette valeur peut être réutilisée par les règles appelantes :

```
uml.Package::obtenirNomPaquetage(): String
{
      result = self.owner.getFullName() + "."
}
```

Figure 6.21. Règles de transformation MOFScript avec valeur de retour.

Une règle de transformation peut aussi définir des paramètres :

```
uml.Model::regleAvecParamètres (s1:String, i1:Integer)
```

```
{
        …ensemble d'instructions…
}
```

Figure 6.22. Règles de transformation MOFScript avec paramètres.

6.3.1.5 Propriétés et variables

MOFScript permet de définir des propriétés et des variables globales ou locales au sein d'une règle ou d'un bloc d'instructions. Dans la sémantique de MOFScript, une propriété est une constante invariable pouvant être ou non typée. Si une propriété ou une variable n'est pas typée au moment de sa déclaration alors elle sera typée dynamiquement selon la valeur assignée (figure 6.23).

```
property maConstante:String = "valeur"
var maVariable = 7
```

Figure 6.23. Propriétés et variables MOFScript.

6.3.1.6 Types de données natifs

MOFScript définit les types de données suivants :

- String : représente une valeur textuelle,
- Integer : représente un nombre entier,
- Real : représente un nombre réel,
- Boolean : représente un booléen,
- Hashtable : représente une table de hachage.
- List : représente une liste,
- Object : représente un type générique.

6.3.1.7 Ecriture dans un fichier

L'instruction *print* est utilisée pour écrire dans un fichier ou sur la sortie standard d'un environnement d'exécution :

```
File modele ("modelXML.xsd")
modele.println ("<xs:schéma …>");
```

Figure 6.24. Ecriture dans un fichier à partir d'une règle MOFScript.

6.3.1.8 Itérateurs

MOFScript définit des itérateurs pouvant être utilisés pour itérer sur des ensembles d'éléments d'un modèle source. L'instruction *forEach* définit un itérateur sur un ensemble d'éléments. L'instruction *forEach* peut être restreinte à un type donné du modèle source.

```
uml.Class::transformerClasse()
{
    self.ownedAttribute->forEach(p : uml.Property)
    {
        p.transformerAttributs()
    }
    … ensemble d'instructions
}
```

Figure 6.25. Itérateur MOFScript.

6.3.1.9 Instructions conditionnelles

MOFScript définit des instructions conditionnelles *if / else* classiques :

```
if (c.hasStereotype ("complexType"))
{
  …ensemble d'instructions
}
else
{
  …ensemble d'instructions
}
```

Figure 6.26. Instructions conditionnelle MOFScript.

6.3.1.10 Instruction de boucle conditionnelle

Une boucle conditionnelle est définie par l'instruction *while*. Cette instruction permet d'exécuter un ensemble d'instructions tant qu'une condition est respectée :

```
var  i : Integer = 10
while (i > 0)
{
  i -=1
}
```

Figure 6.27. Boucle conditionnelles MOFScript.

6.3.1.11 Opérations sur les modèles

MOFScript définit les opérations suivantes applicables à un modèle source :

- objectsOfType(type) : retourne une collection d'instances d'un type donné à l'intérieur d'un modèle. Cette méthode est particulièrement utile lorsque l'on définit un point d'entrée sans contexte et que l'on souhaite effectuer un traitement particulier sur un ensemble d'éléments (figure 6.28).

```
main ()
{
     uml.objectsOfType(uml.Package) -> forEach(p)
     {
       p.transformerPaquetage()
     }
}
```

Figure 6.28. Collection d'objets MOFScript d'un type donné.

- store(uri) : enregistre le modèle source dans l'emplacement spécifié par l'URI donnée.

6.3.1.12 Opérations sur les modèles UML

MOFScript définit un ensemble d'opérations applicables aux modèles UML :

- *hasStereotype*(String nomStéréotype) : indique si une entité UML possède le stéréotype passé en paramètre,
- *getAppliedStereotypes*() : permet d'obtenir l'ensemble des stéréotypes appliqués à un élément du modèle source UML,
- *getAppliedStereotype*() : permet d'obtenir le stéréorype appliqué sur un élément UML,
- *hasValue*(*Stereotype unStereotype* OU *String nomStereotype*, *String nonValeur*) : indique si un élément a un stéréotype définissant une valeur pour une propriété donnée,
- *getValue*(*Stereotype unStereotype* OU *String nomStereotype*, *String nonValeur*) : retourne la valeur d'une propriété d'un stéréotype porté par un élément.

Nous utilisons l'ensemble de ces opérations sur les modèles UML pour gérer les transformations spécifiques associées à l'utilisation des stéréotypes que nous avons définis dans le chapitre précédent. A partir des spécificités et des fonctionnalités offertes par MOFScript, nous pouvons générer de manière automatique un modèle XML Schema à partir d'un modèle UML[18].

6.3.2 Transformation d'un modèle XML Schema vers un modèle UML

Dans la section précédente, nous avons montré comment générer un document XML Schema de manière automatique à partir d'un modèle UML à l'aide de MOFScript. Dans cette section, nous présentons un processus de rétro-ingénierie permettant d'obtenir un modèle UML à partir d'un modèle XML Schema. Cette seconde étape, dans notre approche de transformation de modèles, revêt un aspect plus délicat dans la mesure où nous cherchons à définir un modèle graphique abstrait à partir d'un modèle technique en dehors de tout standard. Cela nécessite donc de pouvoir représenter les modèles XML Schema et UML dans un formalisme commun.

6.3.2.1 XML Metadata Interchange Format (XMI)

XMI est un formalisme de représentation de modèle recommandé par l'OMG créé dans le but de pouvoir échanger des modèles entre outils de modélisation. La représentation UML d'un modèle représente le niveau abstrait de celui-ci ; le niveau concret est matérialisé par la représentation XMI de ce modèle. XMI représente un format pivot pour représenter

[18] Nous choisissons de ne pas présenter en détails les scripts que nous avons développés dans la mesure où il ne représente qu'un aspect technique des mappings que nous avons établis.

et manipuler un modèle UML. La figure 6.29 présente une classe UML et sa représentation possible en XMI.

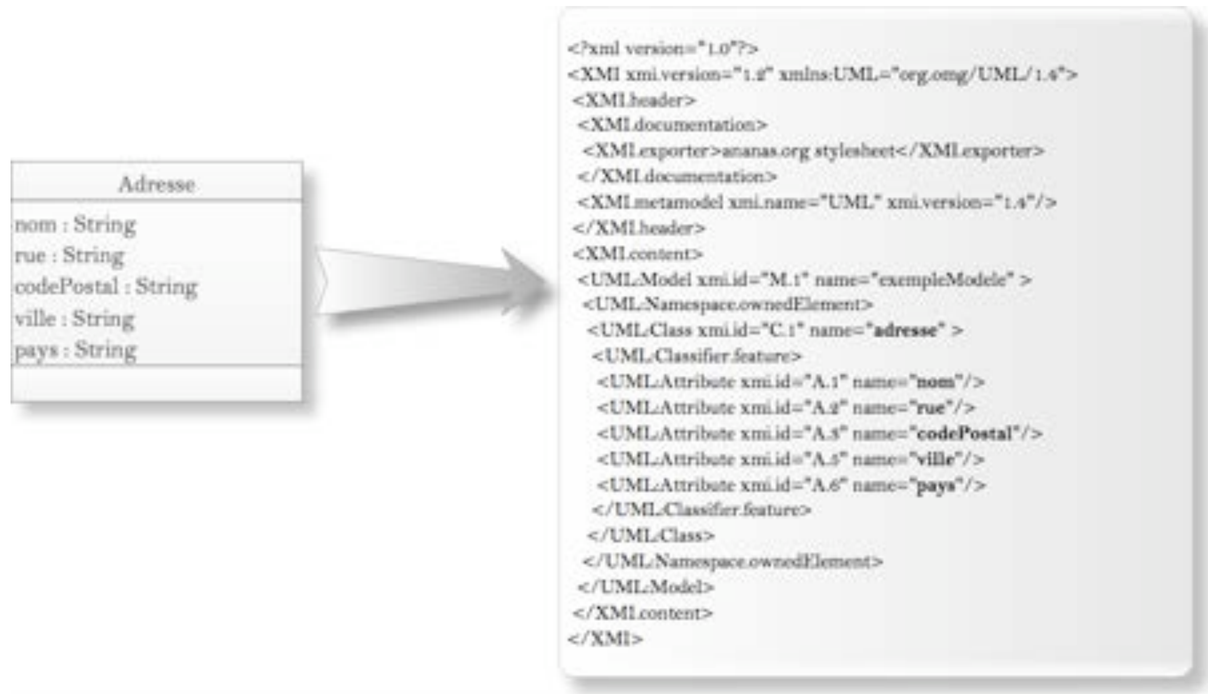


Figure 6.29. Représentation XMI d’une classe UML.

Le Document XMI contient deux parties importantes : un entête (ou *header)* et un contenu (ou *content*). La partie *header* définit les informations sur le modèle représenté (propriétaire, description, etc.). La partie *content* définit le contenu du modèle, c'est-à-dire les classes, les stéréotypes, les associations, etc. De nombreux logiciels de modélisation incluent les fonctionnalités d’import et d’export de modèles UML au format XMI. Cependant, ces logiciels peuvent pour un même modèle générer des documents XMI valides mais ayant des structures différentes, rendant l’échange de modèles difficile, dans les faits, entre ces différents outils. Cet inconvénient majeur nous oblige à appliquer nos propres mappings afin de générer des documents XML Schema, conformes aux normes du W3C, dans le but de faciliter l’échange et l’interopérabilité de modèles entre applications.

6.3.2.2 XSL Transformation (XSLT)

XMI permet de représenter un modèle UML sous la forme d’un document XML permettant ainsi d’établir une corrélation entre un modèle XML Schema et un modèle UML. XSLT, quant à lui, est un langage permettant d’appliquer des règles de transformation sur un document XML pour obtenir un autre document XML. Ces règles de transformation

sont décrites dans des feuilles de styles XSL. La figure 6.30 décrit le processus de transformation :

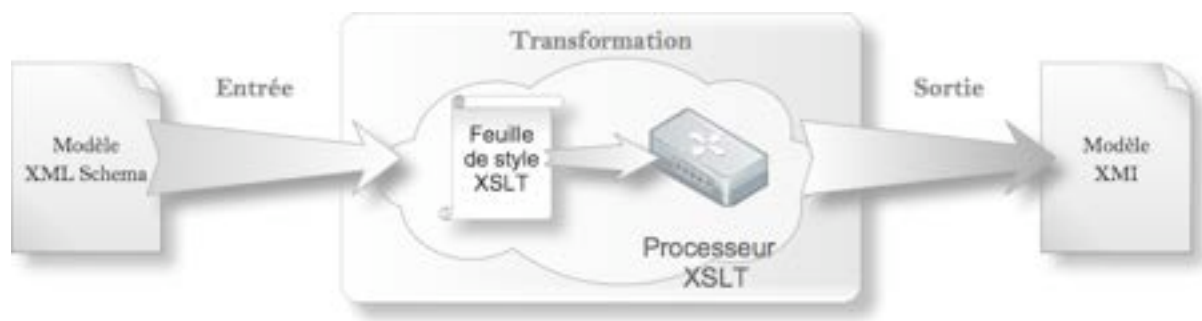


Figure 6.30. Processus de transformation XSLT.

Conçu à l'origine comme le composant de transformation du langage de formatage XSL [W3C, 2001d] du W3C, XSLT *(XSL Transformation)* [W3C, 1999a] est devenu un langage de transformation de documents XML indépendant. Il permet de transformer un document XML en un autre document XML, HTML, XTHML, mais ce pourrait être tout aussi bien un document d'un autre format, par exemple du texte pur, un document pdf, etc. Ce langage de transformation structurelle utilise XPath [W3C, 1999b] pour sélectionner et filtrer les nœuds d'arbres. XSLT est un langage déclaratif à base de *règles* spécifiant comment sera le résultat et non comment sont effectuées les transformations sur les nœuds de l'arbre source.

6.3.2.2.1 Déclaration d'une règle de construction

La déclaration d'une règle de construction ou *template* est définie comme suit :

```
<xsl:template match="pattern">
        ensemble d'instructions…
</xsl:template>
```

Figure 6.31. Déclaration d'un template XSLT.

pattern est une expression XPath qui spécifie le ou les éléments sur lesquels va s'appliquer la règle. Un *template* définit une séquence d'éléments XSL qui contient des opérations de transformation (comme la sélection ou l'insertion de nœuds) à effectuer pour obtenir le document résultat. Parmi ces actions, nous avons :

- l'écriture de la valeur du nœud racine d'un élément, d'un attribut, d'un commentaire, ou d'une instruction de traitement *(processing instruction)* en utilisant l'instruction

xsl:value-of select="expressionXPath". Si cette valeur doit être mise comme valeur d'un attribut résultant, il est impossible d'employer cette instruction sans quoi le document XSL serait mal formé. La solution est de placer directement l'expression XPath entourée d'accolades dans la partie valeur de l'attribut.

```
<xsl:template match="xs:complexType">
  <UML:Class xmi.id="{generate-id()}" name="{@name}">
    …
   </UML:Class>
</xsl:template>
```

Figure 6.32. Instruction d'écriture XSLT.

Cet exemple illustre un *template* appliqué à l'élément *xs:complexType* d'un modèle XML Schema en entrée. Il crée un nouvel élément XMI *UML:Class* dans l'arbre de sortie dont la valeur de l'attribut *name* est calculée à partir de l'instruction *@name*. L'instruction @ permet d'obtenir la valeur d'un attribut.

- La copie des nœuds à partir de l'arbre d'entrée.

Cet exemple définit une règle de construction appliquée aux éléments *xs:simpleType* et *xs:complexType*, en les copiant sans changement dans l'arbre de résultat.

```
<xsl:template match="xs:simpleType|xs:complexType">
    <xsl:copy-of select="."/>
</xsl:template>
```

Figure 6.33. Instruction de copie XSLT.

6.3.2.2.2 Application d'une règle de construction

L'exécution d'une règle de construction *template* est définie comme suit :

```
<xsl:template match="pattern">
          <xsl:apply-templates select="expressionXPath"/>
</xsl:template>
```

Figure 6.33. Instruction d'exécution d'une règle XSLT.

L'élément *apply-templates* permet d'appliquer les *templates* d'une feuille de style sur les fils du nœud courant et les nœuds textuels. *expressionXPath* traite les nœuds sélectionnés par la règle spécifiée au lieu de traiter tous les éléments.

Exemple :

```
<xsl:template match="xs:schema">
        <xsl:apply-templates select="xs:simpleType"/>
</xsl:template>
```

Figure 6.34. Exemple d'application d'une règle XSLT.

Dans l'exemple précédent, nous voulons sélectionner tous les sous-éléments *xs:simpleType* du nœud courant *xs:schema* et nous appliquons la règle appropriée à chacun d'eux.

6.3.2.2.3 Autres notions

Dans les expressions XPath, il est possible de conditionner des traitements avec les instructions suivantes :

- *If* (mais sans partie *else*) comme suit :

```
<xsl:if test="expressionXPath">
  ensemble d'instructions…
</xsl:if>
```

Figure 6.34. Instruction conditionnelle XSLT.

- Switch :

```
<xsl:choose>
   <xsl:when test="expressionXPath1">  contenu 1…</xsl:when>
   <xsl:when test="expressionXPath2">  contenu 2…</xsl:when>
   …
  <xsl:otherwise>  contenu n </xsl:otherwise>
</xsl:choose>
```

Figure 6.35. Instruction de branchement conditionnel XSLT.

Il est possible de répéter un traitement sur un ensemble d'éléments sélectionnés avec une expression XPath via *xsl:for-each*. Dans une boucle[19] ou lors d'un appel récursif de règles, les éléments traités peuvent être ordonnés *(xsl:sort)* en fonction de la valeur d'un attribut ou d'un nœud (attribut *select*). Les informations du document résultant peuvent aussi être numérotées *(xsl:number)*. Parfois, il est impossible d'écrire directement le nom de l'élément ou de l'attribut à produire car il dépend du document en entrée. Dans ce cas, il est nécessaire d'utiliser les instructions *xsl:element* ou *xsl:attribute*.

```
<xsl:template match="element">
  <xsl:element name="@attribut1">
        <xsl:attribute name="@attribut2">
                  <xsl:value-of select="../element2"/>
        </xsl:attribute>
    </xsl:element>
</xsl:template>
```

Figure 6.36. Attributs XSLT.

Le contenu d'un sous-arbre source peut aussi être copié dans l'arbre résultant en précisant quels éléments doivent être copiés avec *xsl:copy* (et *xsl:copy-of* pour copier un ensemble de nœuds).

```
<xsl:template match="/ |@*|node()"/>
   <xsl:copy>
       <xsl:apply-template select="/ |@*|node()"/>
   </xsl:copy>
</xsl:template>
```

Figure 6.37. Instruction de copie ciblée XSLT.

Un script XSL peut aussi contenir des variables (*xsl:variable*), globales ou locales à une règle. Ces variables peuvent être initialisées avec une valeur constante ou avec la valeur d'un nœud (via l'attribut *select*) et peuvent être utilisées n'importe où, selon leur visibilité, en préfixant leur nom avec le caractère dollar *($)*. Mais ces valeurs ne peuvent pas être modifiées sauf par surcharge des variables.

---

[19] Ce doit être la première instruction de la boucle.

A partir de ces instructions XSLT nous avons défini des scripts de transformation permettant de générer un document XMI à partir d'un modèle XML Schema. La figure 6.38 présente un extrait des scripts de transformation que nous avons établis.

```
<?xml version="1.0"?>
    <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:UML="org.omg/UML/1.4"
                exclude-result-prefixes="xs"
                version="1.0">
    <xsl:output indent="yes"/>
    <xsl:template match="xs:schema">
     <XMI xmi.version="1.2">
      <XMI.header>
       <XMI.documentation>
        <XMI.exporter>Transformation XML Schema vers XMI</XMI.exporter>
       </XMI.documentation>
       <XMI.metamodel xmi.name="UML" xmi.version="1.4"/>
      </XMI.header>
      <XMI.content>
       <UML:Model xmi.id="{generate-id()}"
         name="{substring-after(@targetNamespace,'http://psol.com/uml/')}"
         visibility="public" isSpecification="false"
         isRoot="false" isLeaf="false" isAbstract="false">
         <UML:Namespace.ownedElement>
           <xsl:apply-templates/>
         </UML:Namespace.ownedElement>
       </UML:Model>
      </XMI.content>
     </XMI>
    </xsl:template>
    <xsl:template match="xs:element">
     <UML:Class xmi.id="{generate-id()}" name="{@name}"
        visibility="public" isSpecification="false" isRoot="true"
        isLeaf="true" isAbstract="false" isActive="false">
        <xsl:apply-templates/>
```

```
    </UML:Class>
   </xsl:template>
   <xsl:template match="xs:sequence">
    <UML:Classifier.feature>
     <xsl:apply-templates/>
    </UML:Classifier.feature>
   </xsl:template>
   <xsl:template match="xs:sequence/xs:element">
    <UML:Attribute xmi.id="{generate-id(.)}" name="{@name}"
            visibility="private" isSpecification="false"
            ownerScope="instance"/>
   </xsl:template>
   ….
</xsl:stylesheet>
```

Figure 6.38. Extrait d'un script de transformation d'un modèle XML Schema vers un modèle XMI.

6.4 Expérimentation des transformations

Pour mettre en application notre approche, nous avons développé un outil de modélisation permettant de définir des modèles d'adaptation et par extension des modèles XML Schema. L'outil que nous avons développé est basé sur l'AGL (Atelier de génie Logiciel) ArgoUML [ArgoUML, 2002]. ArgoUML est un logiciel libre de modélisation UML. Sur la base d'ArgoUML nous avons développé un module incluant les profils UML que nous avons présentés dans le chapitre précédent ainsi que les fonctionnalités d'import et d'export de modèles XML Schema. La fonctionnalité d'import nous permet de générer un diagramme de classes UML à partir d'un modèle XML Schema. La fonctionnalité d'export permet de générer le code XML Schema d'un diagramme de classes défini avec nos profils UML.

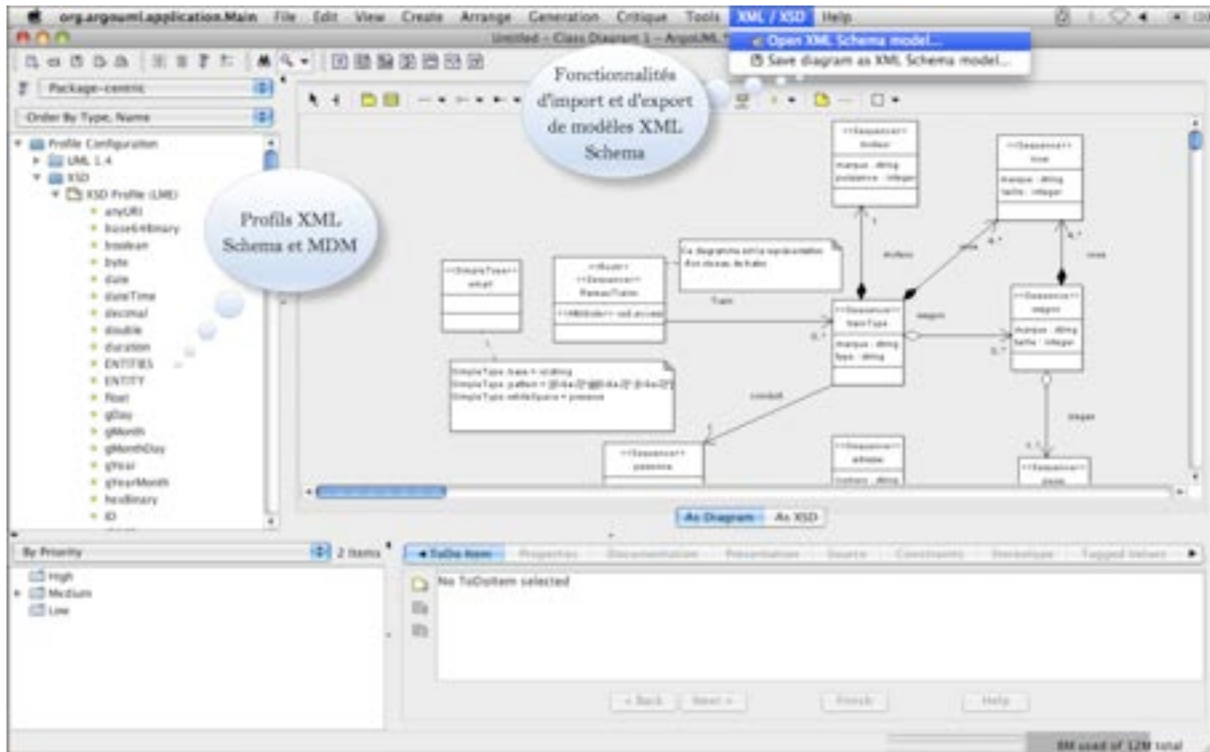


Figure 6.39. Import / Export de modèles XML Schema dans ArgoUML.

La figure 6.39 présente l'interface d'ArgoUML. Les profils UML que nous avons définis et intégrés dans l'outil sont accessibles dans la partie gauche de l'interface. Le navigateur de profils permet de visualiser les stéréotypes et les types de données que nous avons définis.

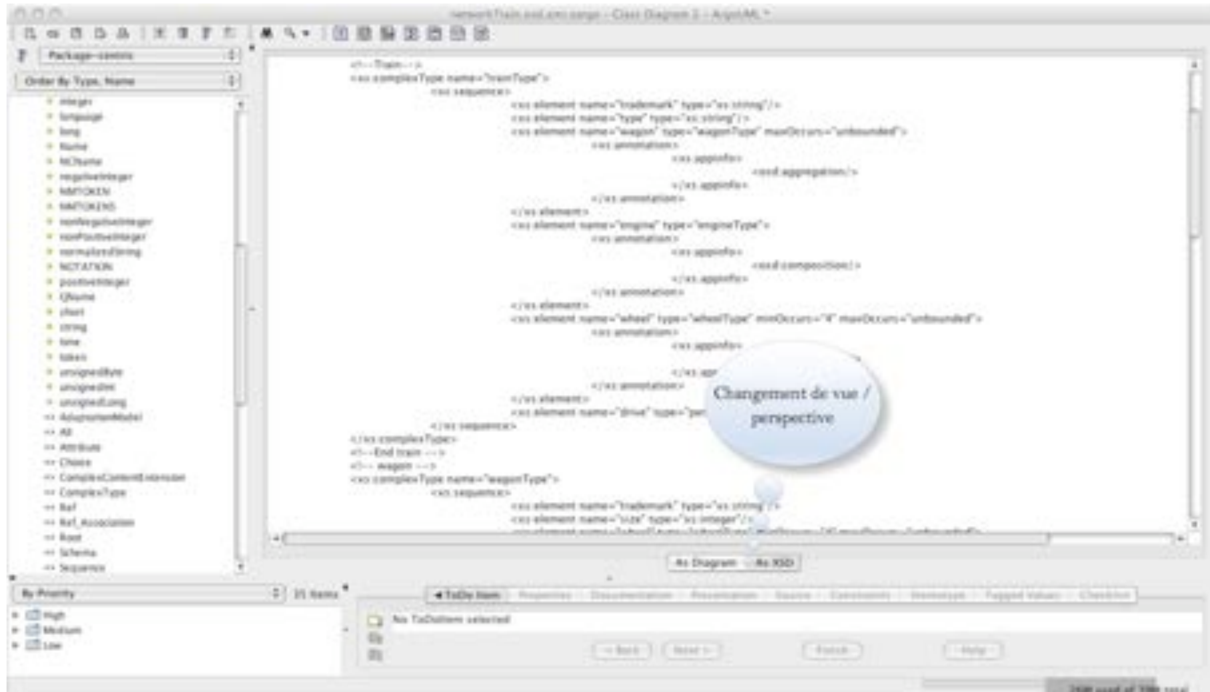


Figure 6.40. Visualisation d’un modèle XML Schema dans ArgoUML.

L’extension que nous avons développée permet de générer un modèle XML Schema à partir d’un modèle UML. Il est possible de visualiser directement le schéma XML généré en changeant de perspective (figure 6.40).

Dans les paragraphes suivants, nous illustrerons l’utilisation de nos profils UML par deux exemples. Le premier exemple est consacré à la définition d’un modèle XML Schema générique ; le second exemple présentera la définition d’un modèle d’adaptation.

6.4.1 Définition d’un modèle XML Schema générique

La figure 6.41 présente un modèle contenant la description d'un bon de commande émis par une personne et géré par une application de facturation.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:documentation xml:lang="fr">
     modélisation d’un bon de commande
    </xs:documentation>
  </xs:annotation>
```

```
<xs:element name="commande" type="commandeType"/>
<xs:element name="commentaire" type="xs:string"/>
<xs:complexType name="commandeType">
      <xs:sequence>
            <xs:element name="adresseLivraison" type="Adresse"/>
            <xs:element name="adresseFacturation" type="Adresse"/>
            <xs:element ref="commentaire" minOccurs="0"/>
            <xs:element name="produits"  type="ProduitType" minOccurs="0"
maxOccurs="unbounded">
      </xs:sequence>
      <xs:attribute name="dateCommande" type="xs:date"/>
</xs:complexType>
<xs:complexType name="Adresse">
      <xs:sequence>
            <xs:element name="nom"   type="xs:string"/>
            <xs:element name="prenom" type="xs:string"/>
            <xs:element name="rue"   type="xs:string"/>
            <xs:element name="ville"  type="xs:string"/>
            <xs:element name="codePostal"    type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="pays" type="xs:NMTOKEN"
         default="FR"/>
</xs:complexType>
<xs:complexType name="ProduitsType>
    <xs:sequence>
         <xs:element name="nomProduit" type="xs:string"/>
         <xs:element name="quantite">
             <xs:simpleType>
                <xs:restriction base="xs:positiveInteger">
                 <xs:maxExclusive value="100"/>
                </xs:restriction>
             </xs:simpleType>
         </xs:element>
         <xs:element name="prix"  type="xs:decimal"/>
         <xs:element ref="commentaire"   minOccurs="0"/>
         <xs:element name="dateExpedition" type="xs:date" minOccurs="0"/>
```

```
      </xs:sequence>
      <xs:attribute name="reference" type="Reference" use="required"/>
    </xs:complexType>
    <xs:simpleType name="Reference">
      <xs:restriction base="xs:string">
        <xs:pattern value="\d{3}-[A-Z]{2}"/>
      </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Figure 6.41. Schéma XML de modélisation d'un bon de commande.

Le bon de commande est constitué d'un élément principal, *commandeType*, et des sous-éléments *adresseLivraison*, *adresseFacturation*, *commentaire* et *produits*. Ces éléments (à l'exception de l'élément *comment*) contiennent à leur tour d'autres sous-éléments, et ainsi de suite, jusqu'à ce qu'un sous-élément comme par exemple *prix* contienne un type simple plutôt qu'un sous-élément. On admet que les éléments qui contiennent des sous-éléments ou portent des attributs sont de type complexe, tandis que les éléments qui contiennent des nombres (ou des chaînes de caractères, des dates, etc.) sans autre sous-élément sont de type simple. Quelques éléments ont des attributs ; les attributs sont toujours de type simple. Dans une instance de document, les types complexes et une partie des types simples qui y sont utilisés sont définis dans le schéma associé. Les autres types simples sont ceux qui sont prédéfinis par XML Schema.

En utilisant le profil UML dédié à la sémantique d'XML que nous avons défini dans le chapitre précédent, nous pouvons modéliser cette structure XML Schema d'un bon de commande à l'aide d'un diagramme de classes. La figure 6.42 présente le diagramme de classes associé à la figure 6.41.

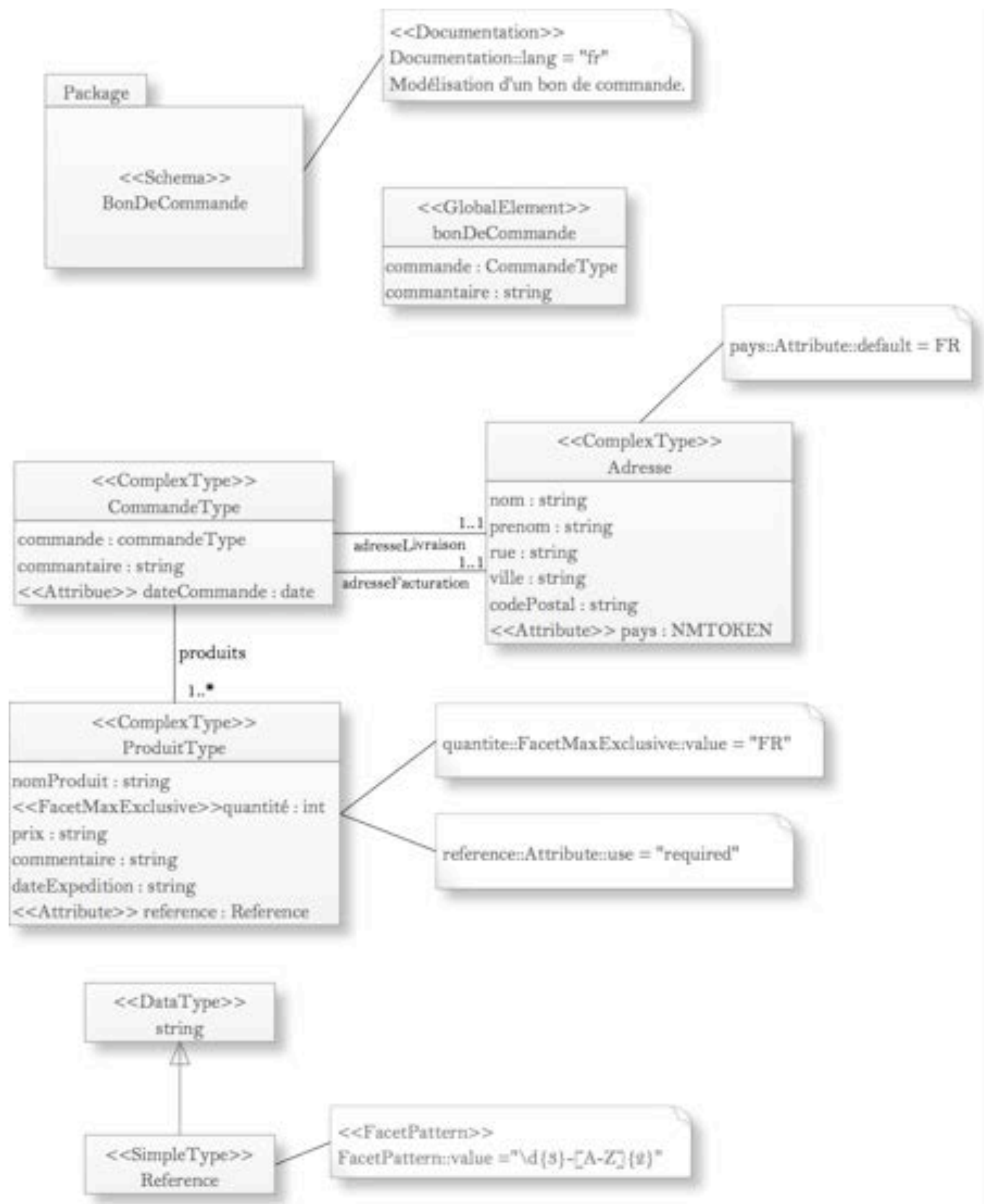


Figure 6.42. Modélisation UML d'un bon de commande défini avec XML Schema.

A partir de ce diagramme et des règles de transformation que nous avons définies, il est possible de générer de manière automatique le modèle XML Schema présenté par la figure 6.41.

6.4.2 Définition d'un modèle d'adaptation

Nous proposons maintenant de définir à l'aide d'un diagramme de classes le modèle de publication d'ouvrages que nous avons présenté dans le troisième chapitre de ce mémoire.

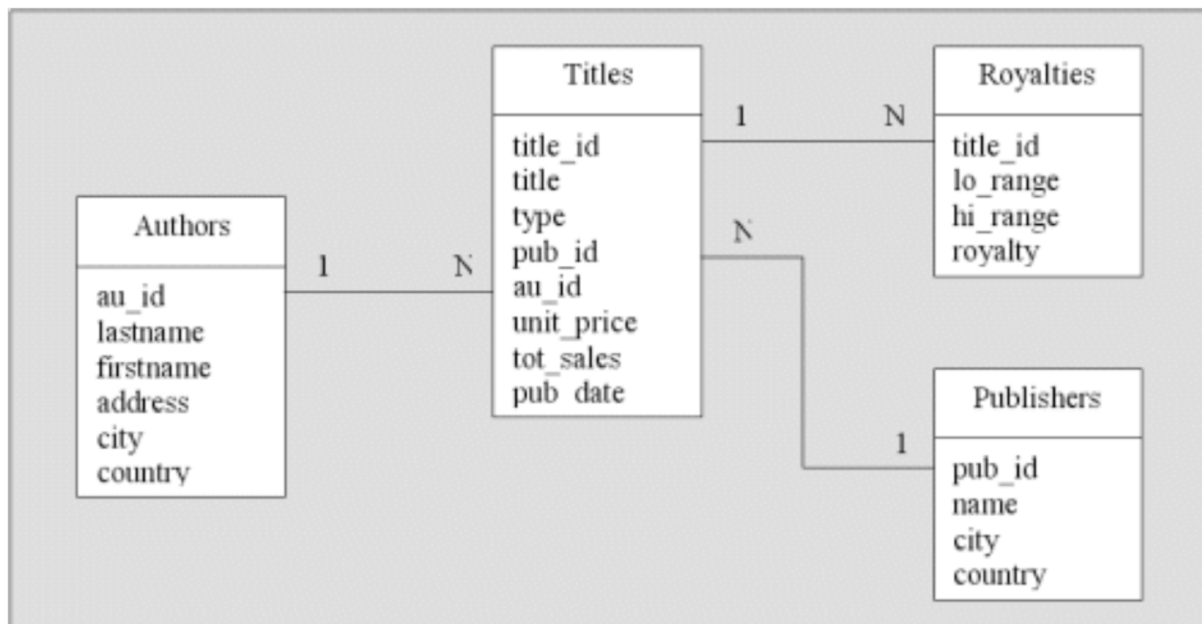


Figure 6.43. Modèle relationnel d'une base de données de publication d'ouvrages.

Le modèle relationnel présenté dans la figure 6.43 définit une base de données de publication d'ouvrages. Cette base de données est composée de quatre tables :

- *Publishers* est la table définissant les informations concernant des éditeurs. Cette table contient les numéros d'identification, noms, villes et pays des éditeurs.
- *Authors* est la table définissant les informations concernant des auteurs. Cette table contient les numéros d'identification, noms, prénoms, adresses, villes et pays des auteurs.
- *Titles* est la table définissant les informations concernant des ouvrages. Cette table contient les numéros d'identification, noms, types, identifiants des éditeurs, prix, etc.
- *Royalties* est la table contenant les informations sur les ventes d'ouvrages. Cette table contient les numéros d'identification des ouvrages et les bénéfices associés.

La figure 6.44. présente le diagramme de classes UML représentant ce modèle d'adaptation à l'aide du profil MDM que nous avons défini.

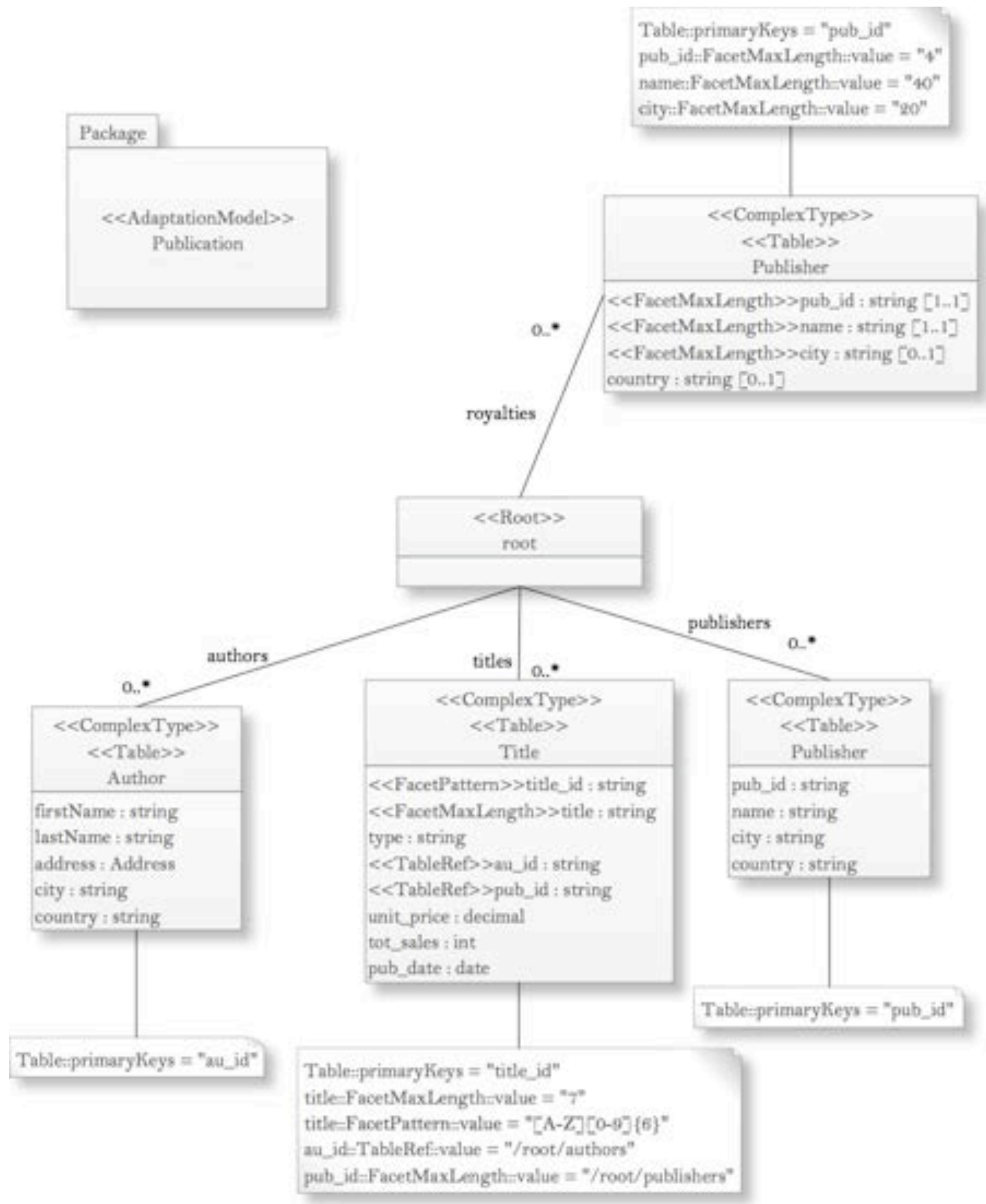


Figure 6.44. Exemple de modèle d'adaptation défini à l'aide du profil UML MDM.

La figure 6.45 présente un extrait de la structure XML Schema correspondant au modèle d'adaptation de la base de données des publications.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="root" osd:access="--">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Titles" type="Title"   minOccurs="0"
         maxOccurs="unbounded"/>
        <xs:element name="Publishers" type="Publisher" minOccurs="0"
         maxOccurs="unbounded"/>
        <xs:element name="Authors" type="Author" minOccurs="0"
         maxOccurs="unbounded"/>
        <xs:element name="Royalties" type="Royalty" minOccurs="0"
         maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="Publisher">
    <xs:annotation>
      <xs:appinfo>
        <osd:table>
          <primaryKeys>/pub_id</primaryKeys>
        </osd:table>
      </xs:appinfo>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="pub_id">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:maxLength value="4"/>
            <xs:pattern value="[0-9]{4}"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="name">
        <xs:simpleType>
```

```
                <xs:restriction base="xs:string">
                    <xs:maxLength value="40"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
        <xs:element name="city" minOccurs="0">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:maxLength value="20"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:element>
        <xs:element name="country" type="xs:string" minOccurs="0">
    </xs:sequence>
  </xs:complexType>
  …
</xs:schema>
```

Figure 6.45. Structure XML Schema d'un modèle d'adaptation de gestion de publications d'ouvrages.

6. Conclusion

Dans ce chapitre, nous avons présenté un processus de transformation permettant de générer de manière automatique un modèle PSM, dans notre optique XML Schema, à partir d'un modèle abstrait UML. La démarche à suivre se base sur les profils UML présentés dans le chapitre précédent et des formalismes standards de transformation tels que MOFScript et XSLT. Ces profils représentent l'ensemble des concepts statiques des plateformes cibles à savoir XML Schema et les modèles appliqués au Master Data Management. Ensuite, nous avons implémenté les règles de transformation que nous avons définies au cours de ce chapitre. Couplée à des méthodes de transformation, l'utilisation de nos profils UML permet de s'abstraire de toute spécificité technique liée à la définition des modèles XML Schema appliqués au MDM. Notre méthodologie d'Ingénierie Dirigée par les Modèles s'applique aussi bien à des modèles XML Schema génériques qu'à des modèles XML Schema appliqués au Master Data Management. De ce fait, notre méthodologie est apte à s'appliquer à tous les domaines se basant sur des modèles XML Schema.

Dans le chapitre suivant, nous aborderons les problématiques de validation incrémentale de modèles afin d'optimiser les processus de validation durant les phases de

conception. En effet, la cohérence structurelle et les processus de validation représentent des aspects essentiels dans les phases de définition de modèles de données. Les approches classiques de validation de modèle [Nentwich *et al.*, 2003] [Sabetzadeh *et al.*, 2005] consistent à vérifier l'intégralité d'un modèle. Lorsqu'un modèle est modifié, il est nécessaire de valider à nouveau l'ensemble du modèle pour vérifier si la modification apportée n'a pas entrainée une incohérence dans la structure du modèle. Cette approche est convenable dans des situations de modélisation de modèles de taille raisonnable mais nous pouvons entrevoir que ce processus n'est pas envisageable dans des contextes industriels et lors d'un passage à l'échelle, caractéristiques indissociables dans un cadre d'application du Master Data Management. La principale cause de ce problème est que les informations issues des contrôles effectués lors de chaque modification unitaire ne sont pas exploitées lors des processus ultérieurs de validation globale. Aussi, dans la suite de ce mémoire, nous nous proposons d'introduire une approche incrémentale de validation de modèles.

Chapitre 7

Validation incrémentale de modèles

Résumé. Dans ce chapitre, nous introduisons dans notre approche d'Ingénierie Dirigée par les Modèles une méthode de validation incrémentale afin d'optimiser le processus de vérification de modèles.

7.1 Introduction

Nous avons montré dans les chapitres précédents de ce mémoire qu'une approche axée sur l'Ingénierie Dirigée par les Modèles (IDM) est une solution possible pour réduire les difficultés rencontrées lors de la modélisation de structures de données complexes. En effet, l'objectif d'une approche IDM est de déplacer la complexité de réalisation d'une application logicielle vers la spécification de cette application. Il s'agit donc de faire abstraction du langage de programmation et de se fonder sur un processus de modélisation abstrait axé sur l'utilisation de différents standards tels que MOF [MOF, 2006], XMI [Iyengar *et al.*, 1998], OCL [OMG, 2002b] et UML [UML, 1997]. Les travaux que nous avons précédemment menés pour intégrer une approche IDM à notre problématique ont été de deux types à savoir une partie concernant la modélisation [Menet *et al.*, 2008a] et l'autre concernant la transformation de modèles [Menet *et al.*, 2008b] [Menet, 2008].

Dans ces deux aspects, il est nécessaire de pouvoir valider les modifications faites au fur et à mesure pour obtenir des versions stables. Il n'est plus à démontrer que la cohérence structurelle et les processus de validation représentent des aspects essentiels dans les phases de définition de modèles de données. Les approches classiques de validation de modèles consistent à vérifier l'intégralité d'un modèle. En effet, lorsqu'un élément du modèle est modifié il est nécessaire de valider à nouveau son ensemble pour vérifier si la modification apportée n'a pas entrainée une incohérence dans la structure globale du modèle. Cette approche est convenable dans des situations de définitions de modèles de taille raisonnable, mais nous pouvons entrevoir que ce processus n'est pas envisageable dans des contextes industriels et lors d'un passage à l'échelle. La principale cause de ce problème est que les informations issues des contrôles effectués lors de chaque modification unitaire ne sont pas exploitées lors des processus ultérieurs de validation globale. Dans ce chapitre, nous proposons d'introduire une approche incrémentale de validation de modèles appliquée aux diagrammes de classes UML dans le contexte du Master Data Management.

Pour se faire, nous procédons en plusieurs étapes :

(i) Formalisation du métamodèle des diagrammes de classes UML afin de pouvoir exprimer des règles de validation,

(ii) Définition de contextes de validation basés sur une représentation en graphes. Ces contextes permettent de localiser les parties d'un modèle impactées par la modification d'un élément,

(iii) Définition de l'algorithme de validation incrémentale,

(iv) Mise en pratique et validation de notre approche en intégrant notre méthode de validation incrémentale dans un outil de modélisation UML.

Ce chapitre est organisé de la façon suivante : dans la section 7.2 nous présentons de manière non exhaustive des approches de validation de modèles existantes ; dans la section 7.3, nous formalisons la notion de métamodèle en objet mathématique. Ceci nous permet de définir des règles et des contraintes sous la forme d'expressions écrites sous la forme de formules de la logique du premier ordre. La section 7.4 décrit notre approche de validation incrémentale fondée sur la notion de contexte de validation. Enfin, dans la section 7.5, nous présentons une mise en application de notre méthode de validation incrémentale en exploitant les logiques de description et les moteurs d'inférence.

7.2 Différentes approches de validation de modèles

Nous présentons de manière non exhaustive dans cette section les différentes approches de validation de modèles existantes dans le domaine de l'Ingénierie Dirigée par les Modèles.

7.2.1 Approches fondées sur OCL

Les diagrammes de classes UML ne permettent pas de décrire tous les aspects relevant de la spécification d'un système ; pourtant, il est nécessaire d'avoir la possibilité de décrire des contraintes supplémentaires sur les objets d'un modèle. Par exemple, une règle de gestion telle que « *Une infirmière ne peut plus accéder aux dossiers de ses patients 15 jours après leur sortie de l'hôpital* » n'est pas exprimable directement. Ces contraintes qui sont des informations importantes ne peuvent pas être spécifiées à l'aide des éléments de modélisation définis par UML. Elles sont souvent exprimées en langage naturel ; cependant, l'utilisation d'un langage non formel peut introduire des ambiguïtés. Au contraire, l'utilisation d'un langage formel supprime ces ambiguïtés en se basant sur une grammaire précise. Dans cette optique, l'OMG a adopté le langage OCL (Object Constraint Language) [OMG, 2003c] pour décrire une partie de la sémantique d'UML et comme complément des différents diagrammes UML permettant d'exprimer des contraintes [OMG, 2003b].

Initialement développé par IBM, OCL est un langage formel pour l'expression des contraintes et présente les avantages suivants :

- c'est un langage simple à écrire et à comprendre, et raisonnablement puissant ;
- c'est un langage intermédiaire entre langage naturel et langage mathématique ;
- il est fortement utilisé pour la description des méta–modèles dont celui d'UML.

Comme OCL est un langage déclaratif sans effet de bord, les contraintes OCL exprimées dans le modèle ne modifient pas les instances du modèle. OCL permet l'expression des contraintes suivantes :

- les invariants au sein d'une classe ou d'un type : contraintes qui doivent toujours être vérifiées pour s'assurer du bon fonctionnement des instances de la classe ou du type concerné ;
- les contraintes au sein d'une opération : contraintes qui doivent toujours être vérifiées pour s'assurer de la bonne exécution de l'opération concernée ;
- les pré et post-conditions d'opération : contraintes qui doivent être respectivement vérifiées avant et après l'exécution d'une opération ;
- les gardes : contraintes sur la modification de l'état d'un objet ;
- les expressions de navigation : contraintes pour représenter les chemins au sein de la structure de classes.

OCL souffre cependant d'un certain nombre de lacunes [Olivé, 2007] à savoir :

- il est peu lisible pour des contraintes complexes ;
- il n'est pas aussi rigoureux qu'un langage de spécification comme Z [Spivey, 1998] ou B (pas de preuves possibles) [Abrial, 1995];
- sa puissance d'expression est limitée.

Dans le contexte de l'IDM, le langage OCL peut être utilisé pour définir des contraintes sur des métamodèles. Ces expressions OCL contraignent la structure des modèles. Différents travaux utilisent une approche OCL pour définir des méthodes de validation de modèles. Dans le cadre d'une approche basée sur OCL, Neptune [Millan *et al.*, 2008] est une méthode inspirée par la méthodologie Unified Process [Jacobson *et al.*, 1999]. Cette méthode couvre les phases d'analyse et de conception des systèmes logiciels. Elle consiste à modéliser le système par un ensemble de diagrammes UML qui doivent satisfaire des règles de validation établies à plusieurs niveaux du processus de création d'un modèle. Au sein de l'approche Neptune, les règles de validation sont à la fois définies en OCL au niveau métamodèle et au niveau modèle. Une règle OCL définie au niveau du métamodèle permet de définir des contraintes sur la structure des modèles basés sur le métamodèle. Une règle définie au niveau modèle permet de spécifier des contraintes qui seront appliquées aux instances de ce modèle. Si nous devions opter pour une approche OCL pour valider des modèles, les règles OCL devraient être définies au niveau des métamodèles que nous avons définis par l'intermédiaire des profils UML présentés dans le chapitre 5 de ce mémoire. Une des qualités de Neptune est de permettre l'édition, l'ajout et la suppression de règles à tout

moment lors de la définition d'un modèle. Cependant Neptune a pour limitation de ne pas gérer les liens entre règles de validation. Or, la gestion des liens entre règles de validation permet à un système de savoir quelles sont les règles pouvant être contradictoires. En outre, le processus de validation est à la charge de l'utilisateur. Autrement dit, le mécanisme de validation n'est pas exécuté en tâche de fond. L'utilisateur doit demander explicitement à l'application de procéder à la validation du modèle en cours d'édition. Il est enfin important de noter qu'en dépit du fait que l'utilisateur puisse choisir soit de valider l'intégralité d'un modèle ou bien des sous-parties, le mécanisme de validation de Neptune n'est pas incrémental et donc pose des problèmes de performances en terme de temps de validation sur des modèles de taille importante.

Toujours dans le cadre des approches de validation basées sur OCL, UML Analyzer [Egyed, 2007a] met en œuvre un système de validation incrémentale appliqué à des modèles UML. Contrairement à Neptune, l'activité de vérification d'un modèle n'est pas à la charge de l'utilisateur mais est faite au cours de l'édition des modèles de manière automatique. Cette approche a la capacité de supporter un passage à l'échelle sur des modèles de taille industrielle. UML Analyzer utilise une méthode de validation de modèle basée sur OCL mais propose un formalisme de définition de règles de validation spécifique. Cependant, il est fortement semblable à OCL, ce qui nous a fait considérer cette approche dans notre étude des différentes solutions existantes fondées sur OCL. Le mécanisme de validation incrémentale introduit par UML Analyzer est basé sur la notion de *portée* d'une règle de validation. La portée d'une règle permet de déterminer l'ensemble des éléments d'un modèle et l'ensemble des événements pouvant impacter la validité de la règle.

Les approches de validation de modèles fondées sur OCL ont pour avantage d'être correctement intégrées dans un environnement de modélisation dans la mesure où il n'est pas nécessaire de transformer un modèle source dans un formalisme mathématique pour procéder à sa vérification. De plus, des approches telles que UML Analyzer se révèlent performantes sur des modèles de taille importante. Cependant, en dépit du fait qu'OCL soit préconisé pour définir des contraintes de validation de modèles, son utilisation possible dans les ateliers de modélisation est limitée et peu répandue.

7.2.2 Approches fondées sur les logiques de description

Les approches fondées sur les logiques s'appuient sur des langages de la logique du premier ordre pour représenter à la fois les règles de validation et les métamodèles qui constituent le fondement des modèles à valider. Des mécanismes d'inférence sont utilisés pour détecter des violations à ces règles de validation. Ces mécanismes se basent sur une transformation des modèles à vérifier vers une représentation logique (ou mathématique), et sur une vérification des règles de validation.

De nombreux travaux ont abordé la problématique de validation incrémentale de modèles avec des approches logiques. [Nentwich *et al.*, 2003] utilisent une logique du premier ordre pour représenter des vues sur des modèles et des règles de validation entre ces

vues. La détection des erreurs de validation est effectuée à l'aide d'un démonstrateur de théorèmes s'utilisant de manière semi-automatique.

[Sabetzadeh *et al.*, 2005] utilisent le moteur d'inférence JESS [Friedman-Hill, 2001] pour détecter les erreurs de validation sur des modèles UML. Les règles sont définies sous la forme de conditions qui, lorsqu'elles sont satisfaites, déclenchent des actions de résolution que l'utilisateur peut utiliser. Le déclenchement des vérifications est géré en tâche de fond et cela de manière transparente. L'algorithme d'inférence de JESS (RETE) a une complexité algorithmique croissant exponentiellement avec la taille du modèle [Schmedding *et al.*, 2007]. La principale limitation de cette approche réside dans la dépendance au métamodèle et aux actions de modification du modèle. En effet, chaque diagramme de classes UML est converti de manière spécifique vers le formalisme interne du moteur JESS. De plus, cette dépendance soulève des questions sur la gestion des règles inter-modèles.

[Sourrouille *et al.*, 2002] utilisent le moteur d'inférence Sherlock [Bahl *et al.* 2007] pour gérer la validation de modèles UML. Le système Sherlock est composé d'un moteur d'inférence centralisé et d'un ensemble d'agents Sherlock. Sherlock fonctionne en trois étapes :

(i) identification des dépendances dans un modèle,
(ii) construction du graphe des dépendances,
(iii) localisation des erreurs.

Les règles sont écrites sous la forme d'expressions conditionnelles associées à des actions spécifiques à réaliser en cas d'erreur. Sherlock nécessite une description du métamodèle UML. L'intégration entre l'outil de modélisation et le moteur d'inférence se fait par l'intermédiaire de documents XML.

Les approches à base de logiques permettent de couvrir toutes les activités de la validation de modèles : spécification des règles avec un langage de logique, détection des incohérences avec des mécanismes d'inférence et traitement des incohérences par la spécification de règles de correction. De plus, de nombreux moteurs d'inférence tels que Racer [Haarslev, 2001], FaCT [Patel-Schneider *et al.*, 1999], FaCT++ [Tsarkov *et al.* 2006] et Pellet [Sirin *et al.*, 2007] existent et permettent de raisonner sur des modèles. Racer, FaCT et FaCT++ se conforment à DIG [Bechhofer *et al.*, 2003], un protocole standard pour interroger un moteur d'inférence par des requêtes http. Racer se présente comme un logiciel serveur avec lequel l'interaction a lieu par communication réseau. Racer offre également un fonctionnement sous forme de grappe de calcul grâce à RacerProxy. Cette particularité confère à Racer un avantage pour la gestion de fortes demandes de calculs.

Cependant, une première limitation de ces approches réside dans la complexité algorithmique des mécanismes de détection des erreurs de validation. Ils ne permettent pas un passage à l'échelle sur des modèles de grande taille. De ce fait, une approche incrémentale est nécessaire pour réduire l'impact du coût algorithmique. Une autre limitation concerne la problématique de l'intégration de l'outil de modélisation et du moteur d'inférence qui a un impact sur l'automatisation et la transparence globale des activités de gestion des erreurs de validation. Dans les cas où la détection des erreurs de validation est effectuée en tâche de fond de manière transparente pour l'utilisateur, il est en effet nécessaire de maintenir une

synchronisation permanente entre le modèle de l'outil de modélisation et sa représentation dans le moteur d'inférence.

Nous nous intéressons à présent sur les approches de validation de modèles fondées sur la théorie des grammaires de graphe.

7.2.3 Approches fondées sur les graphes

Les approches fondées sur les grammaires de graphe sont basées sur une théorie longuement étudiée dans la littérature [Rozenberg, 1997] [Bardohl *et al.*, 1999]. Le principe de ces approches est de considérer les modèles comme des graphes et les erreurs de validation comme des « motifs » dans ces graphes (ou patterns). Dans cette section, nous présentons en détail deux approches à base de grammaires de graphes et concluons sur leurs apports et leurs limitations.

FUJABA [Geiger *et al.*, 2006] est un atelier UML permettant la génération de prototypes exécutables à partir de diagrammes de classes. FUJABA se fonde sur des techniques de transformation de graphes pour définir et détecter des erreurs de validation. Une étude préliminaire sur les dépendances entre règles de validation a été menée, mais elle ne fonctionne que sur un cas particulier où deux règles sont en conflit (la résolution d'une règle de validation entraîne la violation d'une autre règle de validation). Avec cette solution, il est nécessaire de posséder une bonne connaissance du métamodèle UML, d'OCL et des techniques de réécriture de graphes pour définir des règles de validation. Il n'y a pas d'études existantes sur les performances de ce système mais la détection d'un pattern dans un graphe est un problème connu pour être NP-complet [Varro *et al.*, 2005].

L'approche AGG [Taentzer, 2004] est la référence des approches fondées sur les graphes. Cette approche permet la définition, la détection et la résolution d'erreurs de validation. AGG dispose d’un outil de gestion de graphes mettant en application la théorie des graphes typés [Folli *et al.*, 2007] [Hermann *et al.*, 2008]. Une contribution importante de cette approche est de permettre la détection et la gestion des dépendances entre règles de validation. Cependant, l'analyse des dépendances entre les règles de validation rend la flexibilité de la gestion des règles de validation plus difficile. En effet, la complexité de l'algorithme d'analyse d’impacts lors d’une modification est coûteuse. Ceci est dû à la nécessité d'exécuter cet algorithme à chaque modification de l'ensemble des règles de validation. D'un point de vue de l'automatisation des vérifications, le déclenchement des détections est manuelle, à l'inverse de la méthode employée par FUJABA.

D'autres travaux se sont focalisés sur une approche de validation à base de grammaires de graphes. Nous pouvons citer les travaux de [Goedicke *et al.*, 1999] qui utilisent également l'outil de transformations de graphes AGG mais n'abordent pas la problématique de dépendance entre règles de validation. D'autre part, l'activité de détection des erreurs de validation nécessite une coordination manuelle des acteurs impliqués dans le processus de création d'un modèle, elle n'est pas réalisée de manière automatique tel qu'il est fait dans FUJABA. [Engels *et al.*, 2002] ont montré comment à partir d'un ensemble ordonné de règles de transformations de graphes, il était possible d'assurer des propriétés de

validation. L'intérêt de cette approche est de montrer qu'il est possible d'aborder des problématiques de validation avec des techniques d'analyse de la structure des modèles.

Les approches basées sur des grammaires de graphes ont l'avantage de reposer sur un formalisme robuste et longuement étudié [Rozenberg, 1997] [Bardohl *et al.*, 1999] qui permet de définir des règles de validation dans un langage proche des langages de modélisation. Cette approche permet aussi d'aborder la problématique des dépendances entre règles de validation. La difficulté majeure de cette approche réside dans la complexité algorithmique de détection d'un pattern en un problème NP-complet. D'autre part, l'algorithme d'analyse de la structure d'un modèle est également complexe, ce qui limite dans la pratique l'ajout et la modification de l'ensemble des règles de validation. Une des solutions pratiques pour limiter cette complexité algorithmique est de fournir un moteur incrémental de transformation de graphes [Varro *et al.*, 2005].

De ces trois approches (fondées sur OCL, logique de description et graphe) de validation de modèles que nous venons de présenter, nous avons choisi de combiner les approches utilisant les logiques de description et les approches basées sur les grammaires de graphes. Nous utilisons la puissance des logiques de description pour définir les règles de validation s'appliquant aux profils UML que nous avons présentés dans les chapitres précédents. Quant aux approches par grammaires de graphes, nous les exploitons pour représenter et manipuler les modèles à valider.

7.3 Formalisation de la notion de métamodèle

La validation de modèles est une étape importante dans le processus de modélisation. La première étape de notre méthode consiste à formaliser la notion de métamodèle. La formalisation en un objet mathématique nous permettra par la suite de pouvoir exprimer des règles et contraintes et de pouvoir raisonner sur un modèle.

Dans cette section, nous nous intéressons à la formalisation de la notion de métamodèle et nous appliquerons ce formalisme au métamodèle représentant les diagrammes de classes UML.

7.3.1 Définitions

Nous formalisons la notion de métamodèle comme étant un ensemble de *classes*, d'*attributs*, de *références*. Nous utilisons cette notion pour définir un métamodèle (*MM*) comme étant un 8-uplet que l'on peut définir de la manière suivante : $MM = \{MC, P, R, T, MetaClass, PropType, RefType, Super\}$ où :

- MC est l'ensemble fini des métaclasses du métamodèle,
- P est l'ensemble fini des propriétés (attributs) du métamodèle,
- R est l'ensemble fini des références (associations),

- *T* est l'ensemble fini des types primitifs utilisés dans le métamodèle,
- *MetaClass* est la fonction renvoyant la méta-classe de chaque propriété et référence,
- *PropType* est la fonction qui associe à chaque propriété son type primitif,
- *RefType* est la fonction qui associe à chaque référence son type de métaclasse,
- *PropValue* est la fonction qui associe à chaque propriété d'une métaclasse une valeur,
- *RefValue* est la fonction qui associe à chaque référence d'une métaclasse une valeur,
- *Super* ⊂ MCxMC est la relation d'héritage binaire, irréflexive et antisymétrique définie entre les métaclasses du métamodèle.

A partir de ces propriétés, nous pouvons définir qu'un métamodèle est un ensemble de métaclasses, de propriétés, d'associations et de relations d'héritage.

Nous utilisons ces définitions pour formaliser le métamodèle des diagrammes de classes UML.

7.3.2 Formalisation de la notion de diagramme de classes

La figure 7.1 présente le métamodèle simplifié des diagrammes de classes UML.

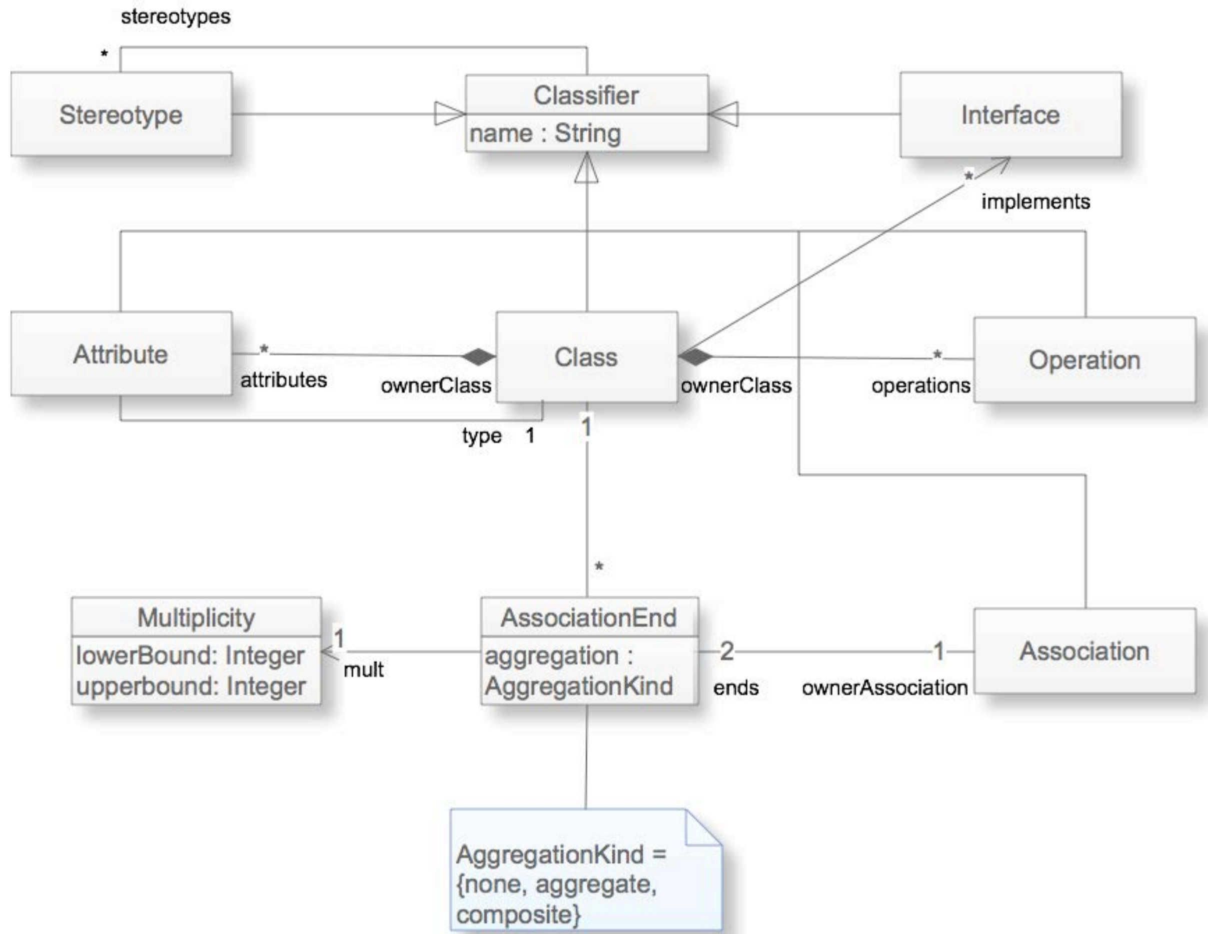


Figure 7.1. Métamodèle des diagrammes de classes UML.

Dans nos travaux actuels, nous excluons du métamodèle les notions d'*interface* et d'*opération*. En effet, rappelons que nous utilisons des diagrammes de classes pour représenter des modèles XML Schema ; or, ces concepts ne peuvent pas être représentés avec XML Schéma et doivent par conséquent être exclus lors de la définition d'un modèle via notre profil UML. [Barbier *et al.*, 2001] ont démontré l'inadéquation de cette méta-modélisation à l'aide d'attributs, notamment du fait de l'absence de contraintes OCL associées à ces attributs (le langage OCL est un sous-ensemble d'UML). Cependant [Henderson-Sellers, 2001] précise que l'utilisation d'OCL dans ces conditions n'est pas appropriée : il ne sert qu'à palier temporairement les inconsistances du langage. Or, OCL devrait être utilisé pour exprimer des contraintes supplémentaires et non pour une solution temporaire. De plus, OCL étant supporté partiellement ou totalement par très peu d'outils, nous avons décidé de conserver cette méta-modélisation et de nous orienter sur des règles et des contraintes exprimées par des formules de la logique du premier ordre.

D'après les définitions présentées en 7.3.1 nous pouvons formaliser le métamodèle des diagrammes de classes UML de la manière suivante :

MC = {Classifier, Class, Attribute, AssociationEnd, Association, Stereotype, Multiplicity}
P = {name, lowerBound, upperBound, aggregation}
R = {type, ownerClass, attributs, mult, ends, ownerAssociation, superClass, stereotypes}
T = {String, Integer}
MetaClass(name) = {Classifier}
MetaClass(lowerBound) = {Multiplicity}
MetaClass(upperBound) = {Multiplicity}
MetaClass(aggregation) = {AssociationEnd}
PropType(name) = {String}
PropType(aggregation) = {String}
PropType(lowerBound) = {Integer}
PropType(upperBound) = {Integer}
MetaClass(type) = {Attribute}, **RefType**(type) = {Class}
MetaClass(owerClass) = {Attribute}, **RefType**(ownerClass) = {Class}
MetaClass(attributes) = {Class}, **RefType**(attributes) = {Attribute}
MetaClass(mult) = {AssociationEnd}, **RefType**(mult) = {Multiplicity}
MetaClass(ends) = {Association}, **RefType**(ends) = {AssociationEnd}
MetaClass(ownerAssociation) = {AssociationEnd}, RefType(ownerAssociation) = {Association}
MetaClass(superClass) = {Class}, **RefType**(superClass) = {Class}
MetaClass(stereotypes) = {Classifier}, **RefType**(stereotypes) = {Stereotype}
Super = {(Class, Classifier), (Attribute, Classifier), (Stereotype, Classifier), (Association, Classifier)}

Figure 7.2. Formalisation du métamodèle des diagrammes de classes UML.

7.3.3 Définition des règles de validation par la logique du premier ordre

La logique du premier ordre, aussi appelée calcul des prédicats, est la logique des formules mathématiques usuelles, telles que :

$$\forall x, y\ \exists z\ (x + z = y + 1), \text{ ou}$$
$$\forall \varepsilon > 0\ \exists \delta > 0\ (|x - x_0| < \delta \Rightarrow |f(x) - f(x_0)| < \varepsilon).$$

Figure 7.3. Exemple de formule de la logique du premier ordre.

L'intérêt de cette formalisation tient à la possibilité de démontrer des théorèmes portant sur les démonstrations prises elles-mêmes comme objet d'étude.

L'examen d'un texte mathématique quelconque permet de constater que les formules qui y apparaissent obéissent à un même schéma général, à savoir assembler, à l'aide de connecteurs booléens $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$ et de quantificateurs $\forall$ et $\exists$, des formules simples du type $t1 = t2$, $t1 < t2$,..., d'une façon générale $R(t1, ..., tk)$ où R désigne une relation k-aire, et où t1, ..., tk représentent des éléments de la structure considérée et sont eux-mêmes soit des variables, soit des noms d'éléments particuliers, soit des combinaisons de variables et de noms à l'aide d'opérations ou de fonctions, sur le modèle de :

$$\forall x_1\ \forall x_2\ (x_1 < x_2 \Leftrightarrow \exists x_3\ (x_1 + x_3 = x_2))$$

Figure 7.4. Quantificateurs de la logique du premier ordre.

Dans la logique du premier ordre deux options sont retenues :

(i) La première est que, le but étant d'exprimer les propriétés de structures variées, il est plus commode d'introduire une famille de logiques plutôt qu'une logique unique. Ces logiques sont toutes bâties sur le même modèle mais chacune dépend d'un choix spécifique des opérations et des relations considérées.

(ii) La seconde option est d'établir une claire distinction entre les symboles qui figurent dans une formule et les objets mathématiques qu'ils représentent. Même si le contexte suggère qu'une relation d'ordre est définie, voire plus précisément un certain ordre, par exemple l'ordre canonique des entiers naturels, il est utile pour la suite de maintenir les formules à un niveau purement syntaxique, afin notamment de pouvoir interpréter une même formule dans plusieurs contextes distincts et, par exemple, pouvoir déclarer que la même formule (7.4) est vraie dans *N* et fausse dans *Z*. De la sorte, la formule elle-même, qui n'est qu'un mot, n'est ni vraie ni fausse hors d'un contexte spécifique. Pour rendre cette distinction visible, on utilisera, au moins dans un premier temps, des notations distinctes, typiquement pour une relation (ensemble de k-uplets) et pour le symbole qui la représente ; pour ne pas compliquer, lorsqu'une notation pour une relation ou une opération est usuelle, on utilisera par défaut la même notation en gras pour le symbole correspondant.

Nous décidons d'adopter la logique du premier ordre comme modèle de définition des règles de validation. Brièvement, adopter une logique consiste à la fois à fixer une famille de formules et adopter une notion de preuve adaptée. Dans le cas de la logique du premier ordre, les choix ont été faits pour calquer au mieux l'usage mathématique, et il n'est donc pas

surprenant que la logique obtenue apparaisse comme la meilleure approximation formelle possible du discours mathématique. Néanmoins, comme pour toute modélisation, il est au moins naturel d'examiner l'adéquation du modèle à la réalité qu'il simule et les bénéfices attendus de la modélisation.

Le pouvoir d'expression des logiques du premier ordre est grand mais néanmoins limité, sauf dans le contexte de la théorie des ensembles.

L'intérêt majeur de la logique du premier ordre tient dans l'existence d'une notion de prouvabilité. En effet, la logique du premier ordre est munie d'une notion de prouvabilité possédant deux qualités essentielles : d'abord et avant tout, son adéquation avec la logique du bon sens, et, d'autre part, l'existence d'un théorème de complétude par rapport à la sémantique naturelle. Aucune autre logique développée à ce jour ne possède ces qualités. Par exemple, les logiques du second ordre ne peuvent satisfaire aucun théorème de complétude. D'une façon générale, [Cate *et al.*, 2007] ont démontré qu'en un sens précis les logiques du premier ordre sont les seules qui puissent à la fois vérifier un théorème de compacité et un théorème de complétude.

La formalisation du métamodèle UML nous permet d'exprimer des contraintes sous la forme de formules de la logique du premier ordre. Par exemple, nous voulons définir une contrainte qui spécifie que le stéréotype *ComplexType* doit être uniquement défini sur des éléments ayant pour métaclasse la métaclasse *Class*. Cette règle est la suivante :

$$\exists\, y, x \mid ((\text{Class}(y, \text{“Stereotype”}) \cap \text{Ref}(x, \text{“stereotype”}, y) \cap \text{Prop}(y, \text{“name”}, \text{“ComplexType”})) \Rightarrow \text{Class}(x, \text{“Class”}))$$

Figure 7.5. Règle de validation fondée sur la logique du premier ordre.

Avec:

- Class(m_e, m_c) est le prédicat tel que Class(m_e, m_c) $\Leftrightarrow$ MetaClass(m_e) = m_c,
- Prop(m_e, p, val) est le prédicat tel que Prop(m_e, p, val) $\Leftrightarrow$ PropValue(m_e, p) = val,
- Ref(m_e, r, val) est le prédicat tel que Ref(m_e, r, val) $\Leftrightarrow$ RefValue(m_e, r) = val.

A l'aide de formules de la logique du premier ordre, nous avons défini les règles de validation représentant les contraintes structurelles spécifiées par nos profils UML et par le métamodèle UML. Nous avons choisi dans ce chapitre de ne pas présenter en détail ces règles car elles n'expriment qu'une traduction triviale des contraintes présentées dans le chapitre 5.

7.4 Validation incrémentale de modèles fondée sur des graphes

Les approches classiques de validation de modèles consistent à vérifier la validité complète d'un modèle. Lorsqu'un modèle est modifié, il est nécessaire de valider à nouveau l'ensemble du modèle pour vérifier si la modification apportée n'a pas entrainé une erreur dans la structure du modèle. Dans cette section, nous proposons d'introduire une approche incrémentale de validation.

Dans une approche de validation incrémentale deux problématiques se posent :

(i) Quelles sont les règles à vérifier lors de chaque action sur le modèle ?
(ii) Quelles parties du modèle sont à vérifier ?

La solution à la première question est de considérer que pour une action donnée seul un ensemble de règles est à vérifier. En effet, une classification des règles à vérifier va nous permettre d'indiquer quelles sont les règles à valider lorsqu'une action précise se produit et de ne pas vérifier les règles non impliquées.

Concernant le second point, la notion de *contexte* que nous allons introduire va nous permettre pour un modèle donné, de connaître l'ensemble des parties du modèle pouvant invalider ou valider une règle de validation. En mettant en place ce mécanisme de contexte, nous optimisons le processus de validation en considérant que seules les modifications affectant un type de contexte peut en entraîner sa vérification.

Sur ces principes, nous mettrons ensuite en place un algorithme de validation incrémentale.

7.4.1 Représentation de modèles par graphes

De par leur structure adaptée aux algorithmes de parcours et par leur facilité de mise à jour, nous avons choisi de représenter un modèle sous la forme d'un graphe orienté. L'utilisation d'un graphe nous permet de partitionner un modèle et de raisonner sur l'intégralité de celui-ci ou sur ses sous-parties. Pour ce faire, nous devons transformer les diagrammes de classes en graphes. La figure 7.6 présente le métamodèle de graphe que nous avons défini pour représenter un diagramme de classes UML.

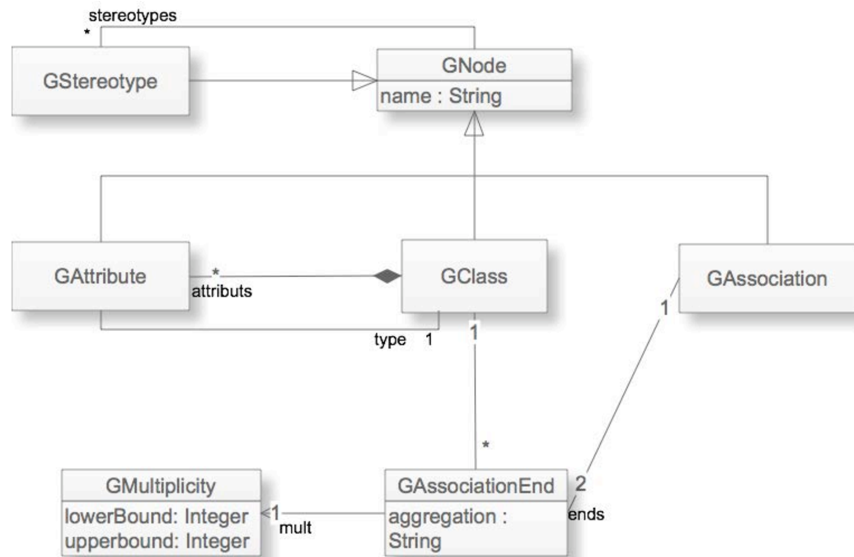


Figure 7.6. Métamodèle de la structure de graphe.

Pour que le processus de transformation d'un diagramme de classes vers un graphe soit le plus simple possible, nous avons fait en sorte que le métamodèle que nous avons défini soit similaire au métamodèle des diagrammes de classes UML. La figure 7.7 présente les règles de correspondance entre ces deux métamodèles et la figure 7.8 présente une partie du graphe correspondant au modèle fictif de publication d'ouvrages que nous avons présenté dans les chapitres précédents.

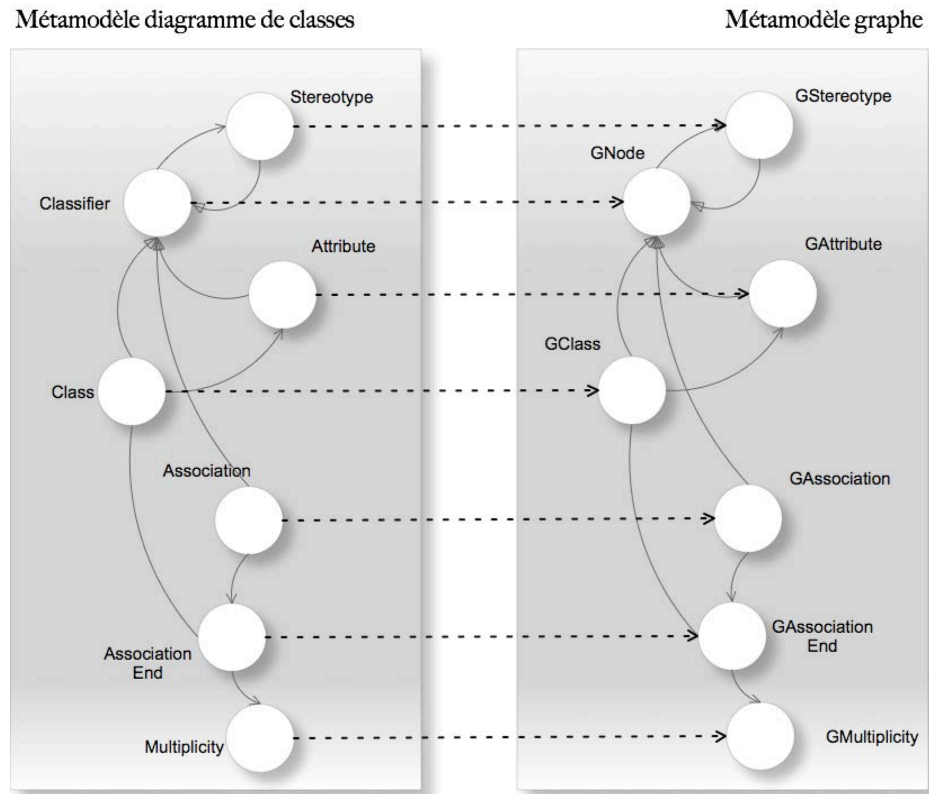


Figure 7.7. Mappings entre un diagramme de classes UML et un graphe.

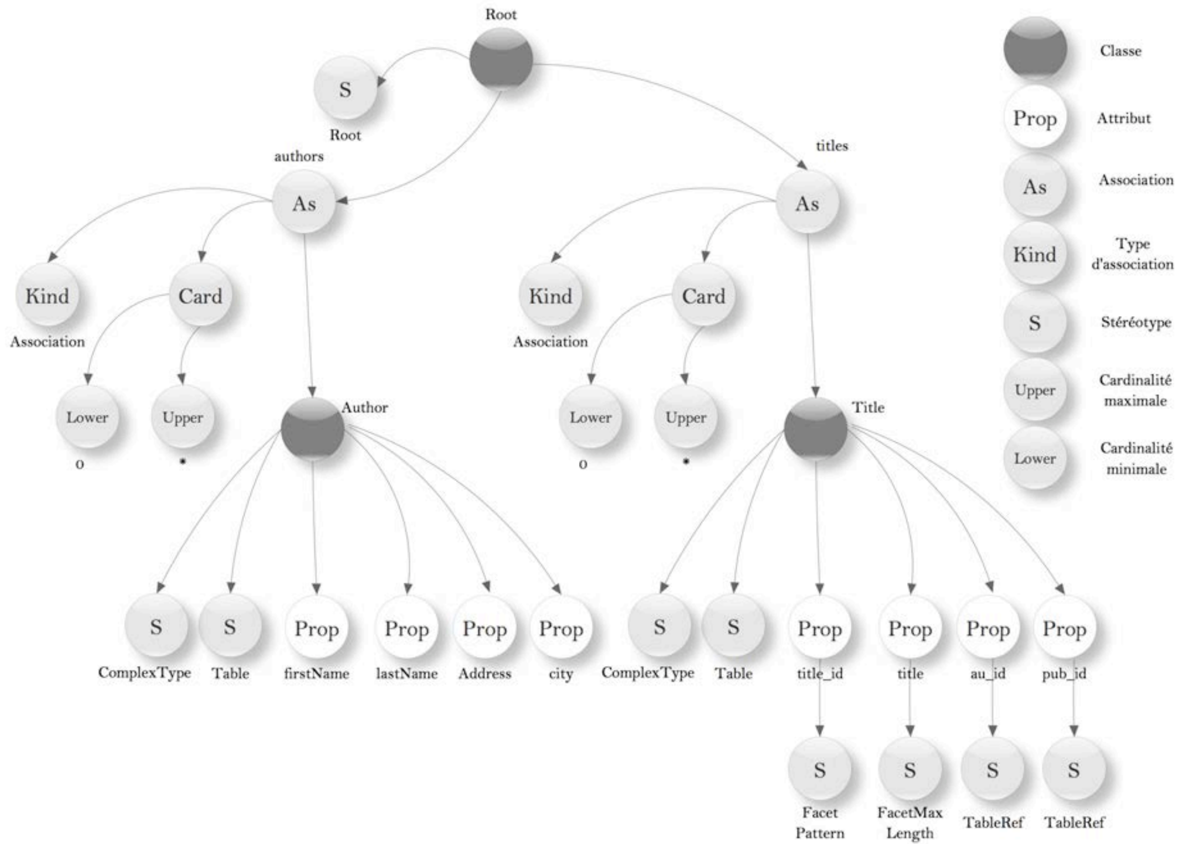


Figure 7.8. Exemple de graphe.

Il est à noter que pour des raisons de simplicité de représentation et de manipulation nous avons fusionné les éléments *GAssociationEnd* et *GAssociation.*

7.4.2 Typologie d'actions et de règles de validation

Dans le cadre d'un processus de définition de modèle de données, nous définissons des types d'action possibles et des catégories de règles de validation. La figure 7.9 présente les types d'action possibles sur les éléments d'un modèle.

Element / Action	Classe	Propriété (attribut)	Référence (association)
Création	✓	✓	✓
Suppression	✓	✓	✓

Modification	✓	✓	✓

Figure. 7.9. Types d'action possibles sur les éléments d'un modèle.

Nous appelons *événements* l'ensemble des combinaisons actions/éléments possibles. Par exemple, la modification d'une classe, la modification d'une propriété, la suppression d'une référence, etc., représentent des événements.

Cette typologie des actions nous permet de spécifier quels sont les événements déclencheur d'un ensemble de règles à vérifier. Nous définissons 4 ensembles de règles qui nous permettent de classifier les règles de validation :

- *RC* est l'ensemble des règles définies sur un modèle,
- RC_{CL} est l'ensemble des règles définies et appliquées à un élément de type *Classe*,
- RC_{Prop} est l'ensemble des règles définies et appliquées à un élément de type *Propriété*,
- RC_{Ref} est l'ensemble des règles définies et appliquées à un élément de type *Référence*.

Avec $RC = RC_{CL} \cap RC_{Prop} \cap RC_{Ref}$

La classification des règles de validation et la représentation en graphe que nous avons adoptées nous permettent de définir les notions de contextes de validation.

7.4.3 Contextes de validation

Notre objectif est de déterminer quelles sont les sous-parties d'un modèle à vérifier lorsqu'un événement sur celui-ci est déclenché. Pour cela, nous définissons la notion de contexte de validation comme étant un ensemble d'éléments (sous-graphes) à valider liés à un type d'événement. Ainsi définissons-nous trois types de contextes de validation :

- *Contexte de classe*, représente l'ensemble des éléments dépendants d'un événement sur un élément de type *Classe*,
- *Contexte de propriété*, représente l'ensemble des éléments dépendants d'un événement sur un élément de type *Propriété*,
- *Contexte de référence*, représente l'ensemble des éléments dépendants d'un événement sur un élément de type *Référence*.

Ces différents contextes sont respectivement représentés par les figures 7.9, 7.10 et 7.11.

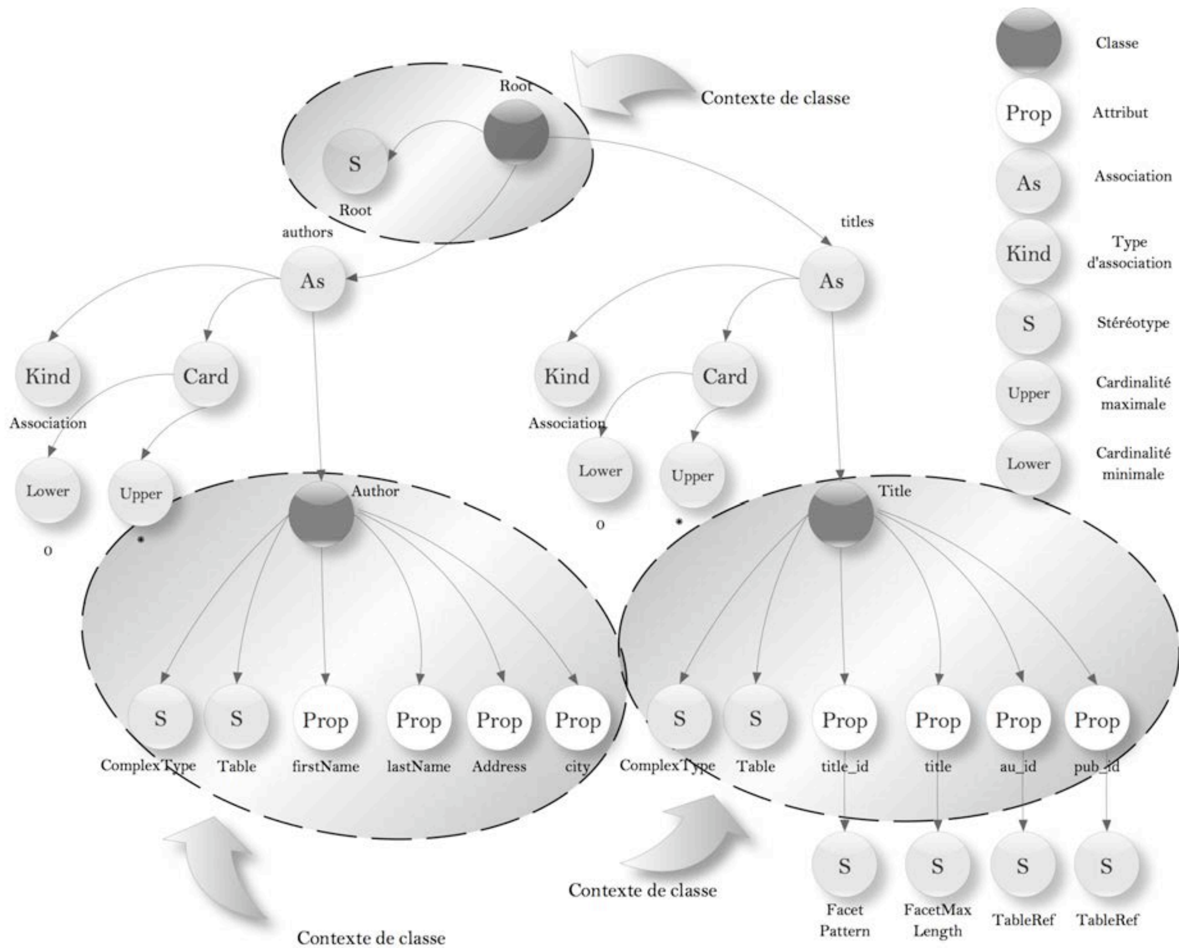


Figure 7.9. Contexte de classe de validation.

Nous reprenons dans la figure 7.9 l'exemple de la publication d'ouvrages présenté dans le chapitre 3 de ce mémoire. Cette figure illustre la notion de contexte de classe. Dans cet exemple, trois contextes de classe sont présentés :

- Le contexte de classe associé à la classe *Root* est composé de la classe *Root* et de son stéréotype (*Root*).
- Le contexte de classe associé à la classe *Author* est composé de la classe *Author*, de ses stéréotypes (*ComplexType* et *Table*) et de ses attributs (*firstName*, *lastName*, *Address* et *city*).
- Le contexte de classe associé à la classe *Title* est composé de la classe *Title*, de ses stéréotypes (*ComplexType* et *Table*) et de ses attributs (*title_id*, *title*, *au_id* et *pub_id*).

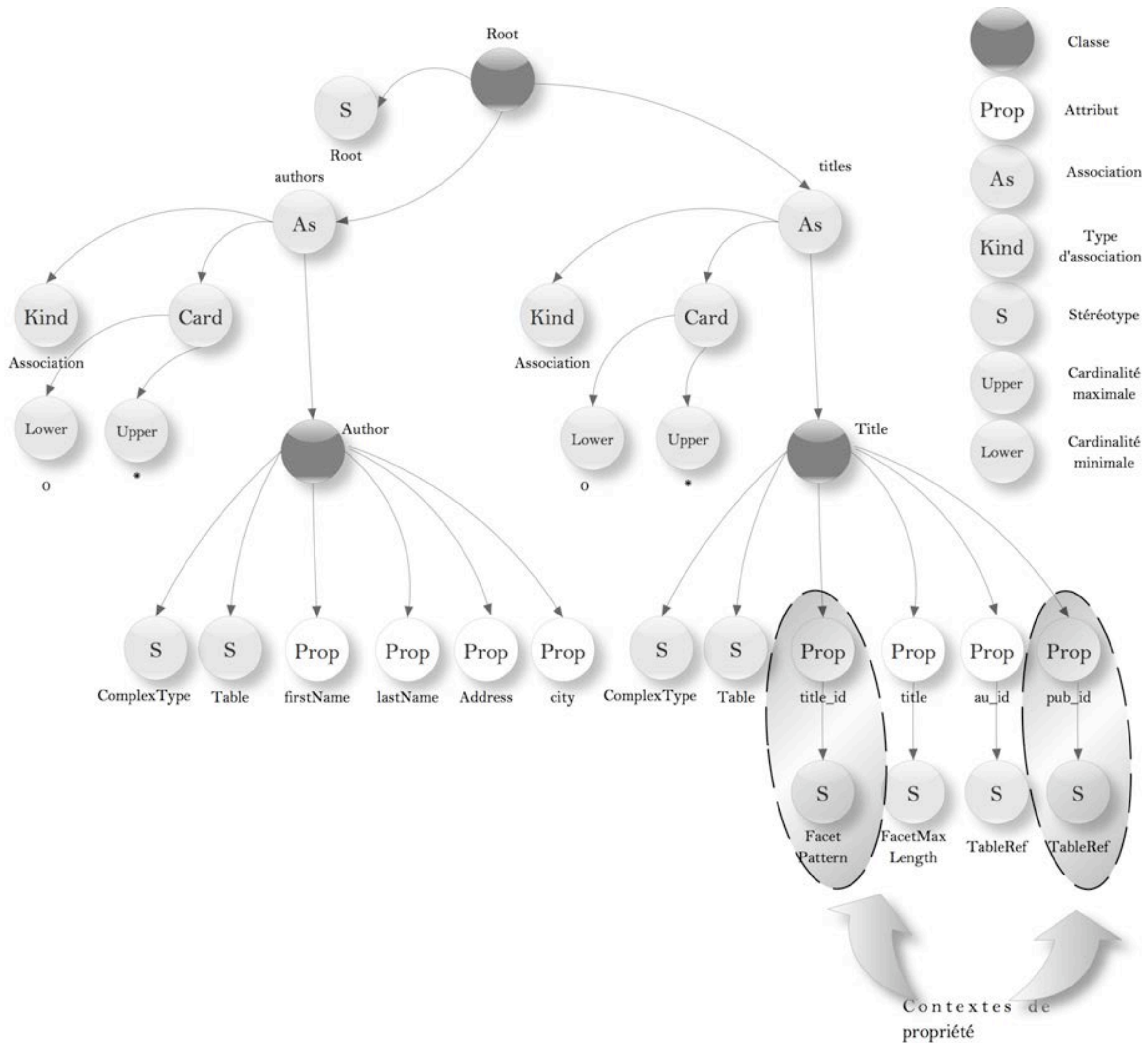


Figure 7.10. Contexte de propriété de validation.

Dans l'exemple 7.10, deux contextes de propriété sont présentés :

- Le contexte de propriété associé à l'attribut *Title_id* est composé de l'attribut *Title_id* et de son stéréotype (*FacetPattern*).
- Le contexte de propriété associé à la classe *Pub_id* est composé de l'attribut *Pub_id* et de son stéréotype (*TableRef*).

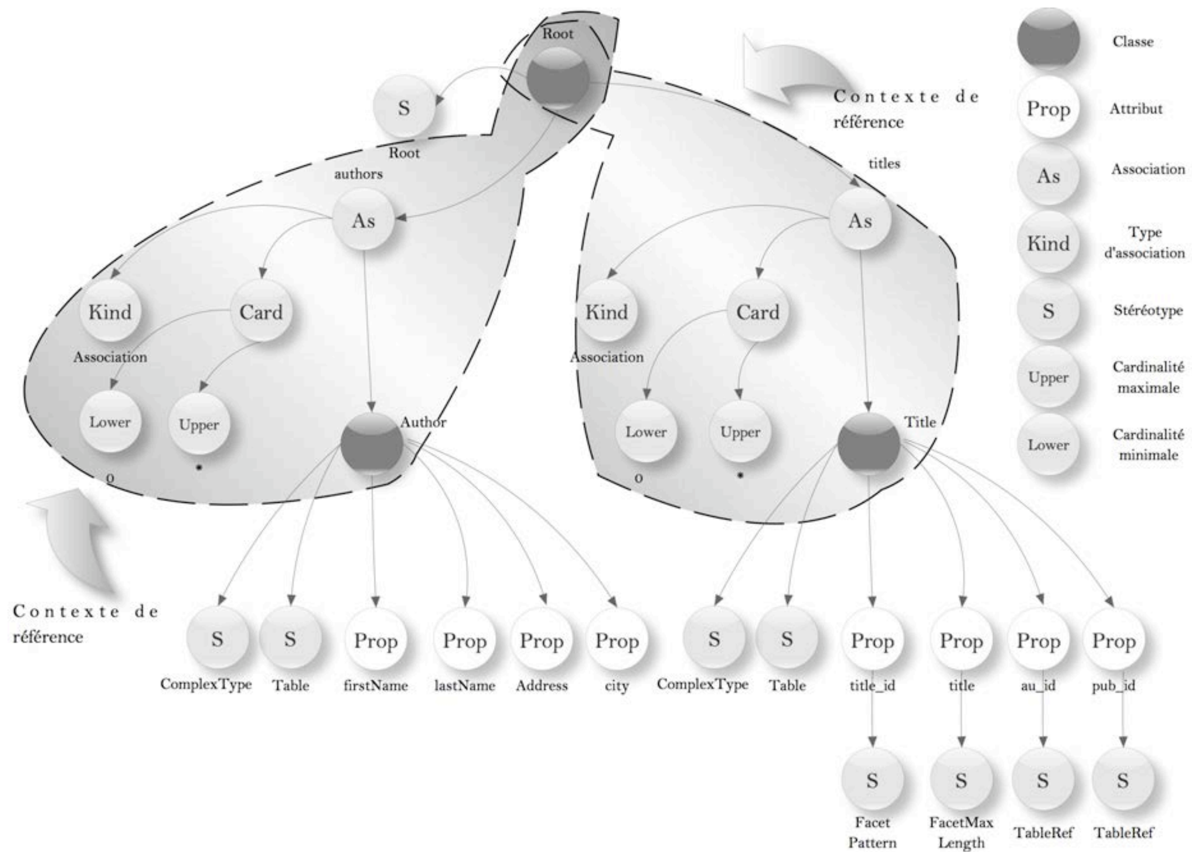


Figure 7.11. Contexte de référence de validation.

Enfin, deux contextes de référence sont présents dans la figure 7.11 :

- Le contexte de référence associé à l'association *Authors* est composé de l'association *Authors*, de son type (*Association*) et de ses cardinalités.
- Le contexte de référence associé à l'association *Titles* est composé de l'association *Titles*, de son type (*Association*) et de ses cardinalités.

Les contextes de validation que nous venons de définir permettent de localiser les parties d'un modèle à valider lorsqu'un événement sur celui-ci intervient. La décomposition d'un modèle en sous-parties permet d'optimiser le processus de validation en associant à chaque action atomique un type de contexte qui correspondra à un sous-graphe représentant la partie du modèle à vérifier. A partir des événements et des contextes de validation que nous avons proposés, nous pouvons définir un algorithme de validation incrémentale.

7.4.4 Algorithme de validation incrémentale.

La définition de notre algorithme de validation incrémentale étant fondée sur les notions d'événement et de contexte de validation, lorsqu'un événement est déclenché le processus de validation incrémentale se déroule de la manière suivante :

(i) Calculer les contextes associés à l'événement,
(ii) Calculer les règles de validation associées à cet événement,
(iii) Valider les contextes dépendants de cet événement.

```
1.  // Cette fonction s'applique à un graphe
2.  Procedure handleEvent(Event newEvent)
3.  BeginProcedure
4.  Context c;
5.  List dependencies;
6.
7.  //Evénement de création
8.  If(newEvent.isCreationEvent())
9.  //Création du contexte dépendant de l'événement
10. c = Context.createNewContext(newEvent)
11. //Ajout du nouveau contexte au graphe
12. this.addContext(c);
13. //Vérification du nouveau contexte
14. c.validate(newEvent) ;
15. //Modification ou suppression d'un élément
16. Else
17. // Mise à jour de l'élément dans le graphe
18. this.update(newEvent);
19. EndIf
20.
21. // Evaluation des contextes dépendants
22. dependencies = this.getContexts(newEvent) ;
23. While(dependencies.hasNext())
24. c = dependencies.next();
25. // Validation du contexte dépendant
26. c.validate(newEvent) ;
27. EndWhile
```

```
28. EndElse
29. EndProcedure
30.
31. // Cette procédure s'applique à un contexte
32. Procedure validate(Event newEvent)
33. BeginProcedure
34. // Récupération des règles associées à l'événement sur l'élément modifié
35. List rules = this.getRules(newEvent);
36.
37. // Vérification des règles avec le contexte courant
38. While(rules.hasNext())
39.     Rule r = rules.next();
40.     r.check(this,newEvent);
41. EndWhile
42. EndProcedure
```

Figure 7.12. Algorithme de validation incrémentale.

Dans notre algorithme d'évaluation de contexte nous gérons les éventuels cycles qui pourraient intervenir en appliquant une stratégie de coloration de graphe [Gravier, 1997].

7.5 Implémentation d'une méthode de validation incrémentale

Nous proposons dans cette section de présenter comment mettre en application notre méthode de validation incrémentale en utilisant des raisonneurs fondés sur les logiques de description.

Au sein des logiques de description, l'accent est mis sur les services de raisonnement. L'objectif principal des logiques de description consiste à pouvoir raisonner efficacement pour minimiser les temps de calcul. Par conséquent, la communauté scientifique a publié de nombreuses recherches qui portent sur l'étude du rapport expressivité/performance des différentes logiques de description [Nardi *et al.*, 2003]. La principale qualité des logiques de description réside dans leurs algorithmes d'inférence dont la complexité est souvent inférieure aux complexités des démonstrateurs de preuves de la logique de premier ordre [Tsarkov *et al.*, 2003]. Les logiques de description utilisent une approche ontologique, c'est-à-dire que pour décrire les individus d'un domaine, elles requièrent tout d'abord la définition des catégories générales d'individus et les relations logiques que les individus ou catégories

peuvent entretenir entre eux. Cette approche ontologique est naturelle pour le raisonnement puisque même si la majorité des interactions se déroulent au niveau des individus, la plus grande partie du raisonnement se produit au niveau des catégories [Russell *et al.*, 2002].

Cette section est organisée comme suit :

La section 7.5.1 présente de manière brève les fondements des logiques de description ;

La section 7.5.2 présente les bases du raisonnement ontologique que sont le niveau terminologique et le niveau factuel d'une base de connaissances ;

La section 7.5.3 met en avant les différents mécanismes d'inférence basés sur les logiques de description ;

La section 7.5.4 présente l'intégration de notre méthode de validation incrémentale à l'atelier de génie logiciel ArgoUML.

7.5.1 Les logiques de description

7.5.1.1 La logique minimale *AL*

Dans cette section nous décrivons la logique de description minimale *AL* introduite par [Schmidt-SchauB *et al.*, 1991] et qui représente la base des logiques de description que nous utiliserons. Nous présenterons les constructeurs ainsi que la sémantique de *AL* et les mécanismes d'extension permettant de créer des logiques de description plus complexes.

7.5.1.1.1 Les constructeurs de *AL*

La figure 7.13 illustre les constructeurs définis par *AL :*

1.	C,D → A	(Concept atomique)
2.	| T	(Concept universel)
3.	| ⊥	(Concept spécifique)
4.	| ¬A	(Opérateur de négation)
5.	| C ∩ D	(Opérateur d'intersection)
6.	| ∃ R.T	(Quantificateur existentiel)
7.	| ∀ R.C	(Quantificateur universel)

Figure 7.13. Constructeurs *AL*.

Le constructeur $C \cap D$ permet de faire la conjonction de deux concepts composés, ce qui représente l'ensemble des individus qui sont à la fois membres du concept C et du concept D.

Le constructeur $\neg A$ est utilisé pour évoquer la négation d'un concept atomique, c'est-à-dire les individus qui n'appartiennent pas au concept atomique *A*.

Le quantificateur existentiel non typé $\exists R.T$ désigne l'ensemble des individus, membres du domaine d'un rôle *R* dans *T*.

Le quantificateur universel $\forall R.C$ désigne l'ensemble des individus du domaine d'un rôle *R* qui sont en relation, par le biais de *R*, avec un individu du concept C.

La sous-section qui suit décrit la sémantique formelle d'*AL*.

7.5.2.1.2 La sémantique formelle de *AL*

La figure 7.14 présente la sémantique formelle de *AL :*

1. $T^I = \Delta^I$
2. $\bot^I = \varnothing$
3. $(\neg A)^I = \Delta^I \setminus A^I$
4. $(C \cap D)^I = C^I \cap D^I$
5. $(\exists R.T)^I = \{a \in \Delta^I \mid \exists b \in \Delta^I : (a, b) \in R^I, b \in T^I\}$
6. $(\forall R.C)^I = \{a \in \Delta^I \mid \forall b \in \Delta^I : (a, b) \in R^I \Rightarrow b \in C^I\}$

Figure 7.14. Sémantique formelle de *AL*.

Afin de supporter la notion de concepts composés, la fonction d'interprétation est étendue par les règles décrites à la figure 7.15. Deux concepts *C* et *D* sont équivalent ssi et seulement si $C^I = D^I$ pour toute interprétation *I* d'un modèle de *T*.

7.5.1.1.3 Les extensions de *AL*

Dans de nombreux cas la logique de description *AL* n'est pas assez expressive pour exprimer certains concepts et rôles. Pour répondre à des besoins d'expressivité supplémentaires il est possible de définir des logiques de description étendant *AL*.

Il existe trois manière d'étendre une logique de description [Baader *et al.*, 2003] :

(i) Ajouter des constructeurs de concepts,
(ii) Ajouter des constructeurs de rôles,
(iii) Définir des contraintes sur l'interprétation des rôles.

7.5.1.1.3.1 L'extension par ajout de constructeurs de concepts ou de rôles

La figures 7.14 illustre des exemples de constructeurs permettant d'étendre *AL* [Baader *et al.*, 2003]. La première colonne contient la lettre qui désigne le constructeur, la deuxième sa syntaxe d'utilisation et la dernière sa sémantique. La nomenclature des logiques de description dicte que pour chaque constructeur ajouté, il faut associer la lettre correspondante au nom de la logique originale. Par exemple, la logique *AL*, enrichie de

l'union (U) et de la quantification existentielle complète (ε), se nomme *ALU*ε. Il faut noter que l'appellation *ALC* équivaut à *ALU*ε dans la mesure où l'union et la quantification existentielle complète s'expriment par la négation complète et inversement, car $\neg(C \cup D) \equiv (\neg C \cap \neg D)$ et $\neg(\exists R.C) \equiv \neg \forall R.\neg C$.

$[\mathcal{O}]$	$\{a_1, a_2, ..., a_n\}$	$\{a_1^{\mathcal{I}}, a_2^{\mathcal{I}}, ..., a_n^{\mathcal{I}}\}$
$[\mathcal{U}]$	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
$[\mathcal{E}]$	$\exists R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \{\exists b.(a,b) \in R^{\mathcal{I}}\} \wedge b \in C^{\mathcal{I}}\}$
$[\mathcal{C}]$	$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
$[\mathcal{I}]$	R_1^{-1}	$\{(y,x) \mid (x,y) \in R_1^{\mathcal{I}}\}$
$[\mathcal{H}]$	$R_1 \sqsubseteq R_2$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
$[\mathcal{F}]$	$= 1R$	$\{x \in \Delta^{\mathcal{I}} \vert \vert \{y \in \Delta^{\mathcal{I}} \mid (x,y) \in R^{\mathcal{I}}\} \vert = 1\}$
	$\geq 2R$	$\{x \in \Delta^{\mathcal{I}} \vert \vert \{y \in \Delta^{\mathcal{I}} \mid (x,y) \in R^{\mathcal{I}}\} \vert \geq 2\}$
$[\mathcal{N}]$	$\geq nR$	$\{a, b \in \Delta^{\mathcal{I}} \vert \vert (a,b) \in R^{\mathcal{I}}\} \vert \geq n\}$
	$\leq nR$	$\{a, b \in \Delta^{\mathcal{I}} \vert \vert (a,b) \in R^{\mathcal{I}}\} \vert \leq n\}$
	$= nR$	$\{a, b \in \Delta^{\mathcal{I}} \vert \vert (a,b) \in R^{\mathcal{I}}\} \vert = n\}$
$[\mathcal{Q}]$	$\geq nR.C$	$\{a, b \in \Delta^{\mathcal{I}} \vert \vert (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \vert \geq n\}$
	$\leq nR.C$	$\{a, b \in \Delta^{\mathcal{I}} \vert \vert (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \vert \leq n\}$
	$= nR.C$	$\{a, b \in \Delta^{\mathcal{I}} \vert \vert (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \vert = n\}$

Figure 7.14. Exemple d'extension de *AL* [Baader *et al.*, 2003].

Le constructeur *O* permet la description de concepts par l'énumération d'individus nommés, *U* désigne l'union de concepts arbitraires, ε la quantification existentielle complète, *C* la négation complète, *I* les rôles inverses et *H* la hiérarchie de rôles. Les constructeurs *F*, *Q* et *N* sont trois variantes de restriction de cardinalité sur le rôle.

Il est donc possible à partir de ces différentes extensions de définir différentes logiques de description qui auront pour nom la juxtaposition du nom de la logique de description de base et des initiales des extensions ajoutées.

7.5.1.1.3.2 L'extension par ajout de contraintes sur l'interprétation des rôles

La spécification d'un ensemble de rôles transitifs N_{R+}, constitue $R+$ une extension par ajout de contraintes sur l'interprétation des rôles (désignée par la lettre $R+$), qui permet l'expression de rôles transitifs tels qu'*ancêtreDe* ou *frèreDe* [Baader *et al.*, 2003]. La lettre *S* désigne la logique *ALC* additionnée de $R+$.

7.5.1.1.3.3 L'extension des types primitifs

Une dernière extension, symbolisée par la lettre (*D*), ajoute le support des types primitifs [Horrocks *et al.*, 2003]. Cette extension augmente *AL* d'un second domaine d'interprétation Δ^{I}_{D} disjoint avec Δ^{I} et qui représente l'ensemble des valeurs de type primitif. Le domaine Δ^{I}_{D} définit plusieurs sous-domaines tels que les entiers, les chaînes de

caractères, les entiers positifs, etc. Les éléments de ces domaines se nomment individus primitifs. De plus, l'extension ajoute un nouveau type de rôle, défini comme une relation binaire sur $\Delta^I \times \Delta^I{}_D$ et appelé rôles à valeurs primitives. La lettre *U* représente l'ensemble de ces rôles.

Dans la section suivante nous nous intéressons plus particulièrement à la logique de description *SHIQ* [Horrocks *et al.*, 2000] qui est utilisé par des moteurs d'inférence que nous présenterons plus loin.

7.5.1.2 La logique de description SHIQ

La logique de description *SHIQ* [Horrocks *et al.*, 2000] regroupe l'ensemble des constructeurs *ALC* noté *S* auxquels est ajouté l'ensemble $\{\geq n\ r.C, \leq n\ r.C\}$ notée *Q*. *H* signifie l'existence d'une hiérarchie de rôles et *I* la possibilité de définir des rôles inverses et transitifs.

L'intérêt d'utiliser SHIQ réside dans le fait que cette logique de description redéfinit de manière efficace la logique *DLR* [Calvanese *et al.*, 1998a] qui permet de raisonner sur des modèles de données conceptuels [Calvanese *et al.*, 1998b]. Ainsi, si nous considérons qu'une *TBox* représente un modèle de données et qu'une *Abox* ses données, nous pouvons utiliser la logique de description SHIQ pour raisonner sur un modèle.

SHIQ peut être décomposé et peut aussi être appelé *ALCQHIR⁺*. Dans la logique de description *ALCQHIR⁺* nous considérons les ensembles suivants :

- C un ensemble de concepts,
- R un ensemble de rôles,
- O un ensemble d'individus.

La figure 7.15 présente la sémantique de *ALCNHR⁺* :

Syntaxe	Sémantique
A	$A^I \subseteq \Delta^I$
$\neg C$	$\Delta^I \setminus C^I$
$C \sqcap D$	$C^I \cap D^I$
$C \sqcup D$	$C^I \cup D^I$
$\exists$ R.C	$\{a \in \Delta^I \mid \exists\, b \in \Delta^I : (a, b) \in R^I, b \in C^I\}$
$\forall$ R.C	$\{a \in \Delta^I \mid \forall\, b \in \Delta^I : (a, b) \in R^I \Rightarrow b \in C^I\}$
$\exists \geq n$ S.C	$\{a \in \Delta^I \mid \Vert\{y \mid (x, y) \in S^I, y \in C^I\}\Vert \geq n\}$
$\exists \leq n$ S.C	$\{a \in \Delta^I \mid \Vert\{y \mid (x, y) \in S^I, y \in C^I\}\Vert \leq n\}$

R	$R^I \subseteq \Delta^I \times \Delta^I$

Figure 7.15. Syntaxe *ALCQHIR*⁺.

7.5.2 Les niveaux de la logique de description

La modélisation des connaissances d'un domaine avec les logiques de description se réalise en deux niveaux. Le premier, le niveau terminologique ou *TBox*, décrit les connaissances générales d'un domaine alors que le second, le niveau factuel ou *ABox*, représente une configuration précise. Une *TBox* comprend la définition des concepts et des rôles, alors qu'une *ABox* décrit les individus en les nommant et en spécifiant en terme de concepts et de rôles, des assertions qui portent sur ces individus nommés. Plusieurs *ABox* peuvent être associés à une même *TBox* ; chacune représente une configuration constituée d'individus, et utilise les concepts et rôles de la *TBox* pour l'exprimer.

7.5.2.1 Le niveau terminologique (TBox)

Le coté gauche de la figure 7.16 présente un exemple de *TBox*.

TBox	ABox
Mâle ⊆ T ¬Femelle Femelle ⊆ T ¬Mâle Animal ≡ Mâle ∪ Femelle Humain ⊆ Animal Homme ≡ Humain ∩ Mâle Femme ≡ Humain ∩ Femelle Mère ≡ Femme ∩ ∃relationEnfant Père ≡ Homme ∩ ∃relationEnfant relationEnfant ⊆ T_R	Humain (john) Mâle (John) Femme (Jane) Humain (Alice) relationEnfant(Alice, John)

Figure 7.16. Exemple d'une base de connaissances.

Avec :

- ⊆ représente l'opérateur d'inclusion,
- ≡ représente l'opérateur d'équivalence,
- ∪ représente l'opérateur *ou*,

- ∩ représente l'opérateur d'intersection (*et*),
- ∃ représente l'opérateur existentiel,

Les prochains paragraphes explicitent divers aspects des *TBox* en se référant à l'exemple de la figure 7.16.

7.5.2.1.1 Les entités atomiques.

Les concepts atomiques et rôles atomiques constituent les entités élémentaires d'une *TBox*. Les noms débutant par une lettre majuscule désignent les concepts, alors que ceux débutant par une lettre minuscule dénomment les rôles (par exemple : les concepts *Femelle*, *Mâle*, *Homme* et *Femme*, et le rôle *relationEnfant*).

7.5.2.1.2 Les concepts et rôles atomiques prédéfinis.

Les logiques de description prédéfinissent quatre concepts atomiques minimaux : le concept et le rôle, les plus généraux de leur catégorie respective, et le concept ainsi que le rôle les plus spécifiques (c'est-à-dire l'ensemble vide).

7.5.2.1.3 Les entités composées

Les concepts et rôles atomiques peuvent être combinés au moyen de constructeurs pour former respectivement des concepts et des rôles composés. Par exemple, le concept composé *Mâle* ∩ *Femelle* résulte de l'application du constructeur ∩ aux concepts atomiques *Mâle* et *Femelle*. Le concept *Mâle* ∩ *Femelle* s'interprète comme l'ensemble des individus qui appartiennent aux concepts *Mâle* et *Femelle*. Les différentes logiques de description se distinguent par les constructeurs qu'elles proposent. Plus les logiques de description sont expressives, plus les chances sont grandes que les problèmes d'inférence soient non décidables ou de complexité très élevée. Par contre, les logiques de description trop peu expressives démontrent une inaptitude à représenter des domaines complexes.

7.5.2.2 Le niveau factuel (ABox)

Une *ABox* contient un ensemble d'assertions sur les individus :

(i) des assertions d'appartenance,
(ii) des assertions de rôle.

Chaque *ABox* doit être associée à une *TBox*, car les assertions s'expriment en terme des concepts et des rôles de la *TBox*. La partie droite de la figure 7.16 illustre un exemple de *ABox*. Une *ABox* désigne des individus impliqués dans des assertions par des noms qu'elle leur donne. L'exemple de la figure 7.16 comprend les individus nommés suivants : *John*, *Jane* et *Alice*. En terme d'assertions d'appartenance, la figure 7.16 indique que *John* est un humain mâle, *Jane* est une femme et *Alice* est un humain. Concernant les relations de rôle, nous

avons défini dans cet exemple la relation *relationEnfant()* spécifiant un couple parent/enfant avec une assertion définissant que *Alice* et *John* ont une relation parent/enfant.

De manière plus concrète, nous considérons dans nos travaux que les métamodèles que nous avons définis à l'aide de nos profils UML représentent des *TBox* et les instances de ces métamodèles représentent des *ABox*.

7.5.3 Raisonnement par inférence

L'inférence s'effectue au niveau terminologique ou factuel que nous exposons respectivement dans les sections 7.5.3.1 et 7.5.3.2. La section 7.5.3.3 présente un tableau comparatif des différents moteurs d'inférence synthétisant les capacités de raisonnement.

7.5.3.1 L'inférence au niveau terminologique

Quatre principaux problèmes d'inférence se présentent au niveau terminologique :

- **Satisfaisabilité** : Un concept *C* d'une terminologie *T* peut être satisfait si et seulement s'il existe un modèle *M* de *T* tel que $C^M = \emptyset$.
- **Subsomption** : Un concept C est subsumé par un concept D pour une terminologie si et seulement si $C^M \sqsubseteq D^M$ pour tout modèle *M* de *T*.
- **Équivalence** : Un concept C est équivalent à un concept D pour une terminologie et seulement si $C^M \equiv D^M$ pour chaque modèle *M* de *T*.
- **Disjonction** : Des concepts C et D sont disjoints pour une terminologie *T* si et seulement si $C^M \setminus D^M = \emptyset$; pour chaque modèle *M* de *T*.

Les moteurs d'inférence actuels tirent généralement profit du fait que les quatre types de problèmes d'inférence peuvent être réduits à des problèmes de subsomption ou à des problèmes de satisfaisabilité. Il a été prouvé par [Baader et Nutt, 2003] que les moteurs d'inférence nécessitent souvent qu'un seul algorithme pour raisonner au niveau terminologique. D'ailleurs, les deux grandes classes d'algorithmes de raisonnement pour les logiques de description correspondent aux façons de réduire respectivement des problèmes d'inférence à des problèmes de subsomption et de satisfaisabilité [Baader *et al.*, 2003].

7.5.3.2 L'inférence au niveau factuel

Le niveau factuel comprend quatre principaux problèmes d'inférence:

- **Cohérence** : Une *ABox A* est cohérente par rapport à une *TBox T* si et seulement s'il existe un modèle *M* de *A* et *T*.
- **Vérification d'instance** : ce problème consiste à vérifier par inférence qu'une assertion *S(a)* est vraie pour tout modèle d'une *ABox A* et d'une TBox *T*.
- **Vérification de rôle** : ce problème consiste à vérifier par inférence si une relation *R(a, b)* est vraie pour tout modèle d'une *ABox A* et d'une *TBox T*.

- **Problème de récupération** : ce problème consister à déduire pour une *ABox A* et un concept *C* d'une terminologie *T* tous les individus d'un modèle *M* de *T*.

7.5.3.3 Les moteurs d'inférences

La figure 7.16 dresse une comparaison des principaux moteurs d'inférence pour les logiques de description : FaCT [Horrocks, 1998], Racer [Haarslev et Möller, 2001], FaCT++ [Tsarkov et Horrocks, 2004], Hoolet [Hoolet, 2004], Pellet [Parsia *et al.*, 2004], Surnia [Surnia, 2003] et F-OWL [Zou *et al.*, 2004]. Le critère "passage-à-échelle" mesure la capacité à demeurer efficace proportionnellement à la complexité des ontologies. La figure 7.16 reprend les données de [Zou *et al.*, 2004] pour comparer ces différents moteurs d'inférence. [Bechhofer *et al.*, 2003]. Racer, FaCT et FaCT++ se conforment à DIG [Bechhofer *et al.*, 2003], un protocole standard pour interroger un moteur d'inférence par des requêtes http. Ce protocole se base sur le langage *SHOIQ(D-N)*. DIG comporte cependant un certain nombre de problèmes [Dickinson, 2004] :

- une faible documentation, incomplète pour certains aspects tels que les codes d'erreurs et les réponses aux requêtes,
- un manque de régularité dans la nomination des entités,
- des tests de conformité inexistants,
- un non support de certains services de raisonnement tels que la non vérification de l'égalité d'individus nommés dans le cas où l'hypothèse de nom unique (HNU), et peu de moteurs d'inférence se conforment à DIG actuellement (figure 7.17).

Moteur		Racer	Fact	Fact++	Pellet
Logique description		$SHIQ(D)^{-}$	*SHIQ, SHF*	*SHIF(D)*	*SHIN(D), SHON(D)*
OWL		OWL-DL	OWL-DL	OWL-LITE	OWL-DL
Implémentation		C++	Common Lisp	C++	Java
Inférence		Tbox/Abox	Tbox	Tbox	Tbox/Abox
API Java		Oui	Oui	Oui	Oui
DIG		Oui	Oui	Oui	Non
Mise-à-l'échelle		Bonne	Bonne	Bonne	Bonne
Décidabilité		Oui	Oui	Oui	Oui
Moteur		F-OWL	Hoolet	Surnia	
Logique description		*SHIQ(D), RDF*	Logique des prédicats	Logique des prédicats	
OWL		OWL-FULL	OWL-DL	OWL-FULL	

Implémentation		Java	Java	Python
Inférence		Tbox/Abox	Tbox/Abox	Tbox/Abox
API Java		Oui	Oui	Non
DIG		Non	Non	Non
Mise-à-l'échelle		Faible	Faible	Faible
Décidabilité		Non	Non	Non

Figure 7.17. Comparatif portant sur des moteurs d'inférence.

Tous ces moteurs raisonnent autant sur des *ABox* que sur des *TBox*, mis à part FaCT et FaCT++ qui se spécialisent en raisonnant sur des TBox seulement. FaCT++ est une variante de FaCT implantée en C++ pour améliorer ses performances.

Les moteurs d'inférence Surnia [Surnia, 2003] et F-OWL [Zou *et al.*, 2004] et Hoolet [Hoolet, 2004] raisonnent sur des logiques de description expressives. Hoolet et Surnia raisonnent sur l'expressivité totale de la logique des prédicats, alors que F-OWL infère sur la logique *SHIQ(D)*. Ce moteur se base sur des méthodes expérimentales de raisonnement qui montrent des performances intéressantes pour des problèmes simples mais limitées pour une utilisation à grande échelle en raison de leur faible performance et de la non-décidabilité de leurs algorithmes. Comme l'indique le critère "Passage à l'échelle" dans le tableau, les moteurs les plus performants actuellement sont Racer, FaCT, FaCT++ et Pellet. FaCT, FaCT++, Racer et Pellet disposent probablement de la plus grande notoriété actuellement, chacun d'eux étant utilisé dans de nombreux projets. Il est à noter que Pellet est intégré à l'éditeur d'ontologies Protégé [Noy *et al.*, 2003].

La plupart des moteurs mentionnés dans cette section procurent une interface de programmation (API) (autre qu'une interface DIG) pour faciliter l'accessibilité par un programme Java (voir le tableau). Ce critère importe particulièrement puisque comme nous l'avons mentionné précédemment DIG comporte plusieurs limitations. L'existence d'une API Java est très important dans notre choix portant sur le moteur d'inférence à utiliser dans la mesure où nous voulons intégrer notre méthode de validation incrémentale dans la plateforme ArgoUML développé en Java. Il s'agit pour nous de faire communiquer via une API Java notre module de validation avec le moteur d'inférence.

7.5.5. Intégration à ArgoUML

Après études des différents moteurs d'inférence que nous avons présenté dans la section précédente, nous avons choisi d'utiliser Racer pour implémenter notre méthode de validation incrémentale. Racer apparaît être le moteur d'inférence le plus adéquat par rapport à nos besoins dans la mesure où il permet à la fois de raisonner sur des *ABox* et des *TBox* ; il possède d'excellentes performances de raisonnement sur des modèles de taille importante et enfin il dispose d'une API Java pour pouvoir être exploité par des applications tierces. Pour mettre en application notre approche nous avons utilisé l'AGL open source

ArgoUML que nous avons présenté dans le chapitre précédent. Dans cette optique, nous avons fait évoluer le processus de validation d'ArgoUML afin qu'il puisse prendre en considération les restrictions appliquées à la sémantique de XML Schema et notre méthode de validation incrémentale. La figure 7.18 illustre le mécanisme de validation que nous avons intégré dans ArgoUML.

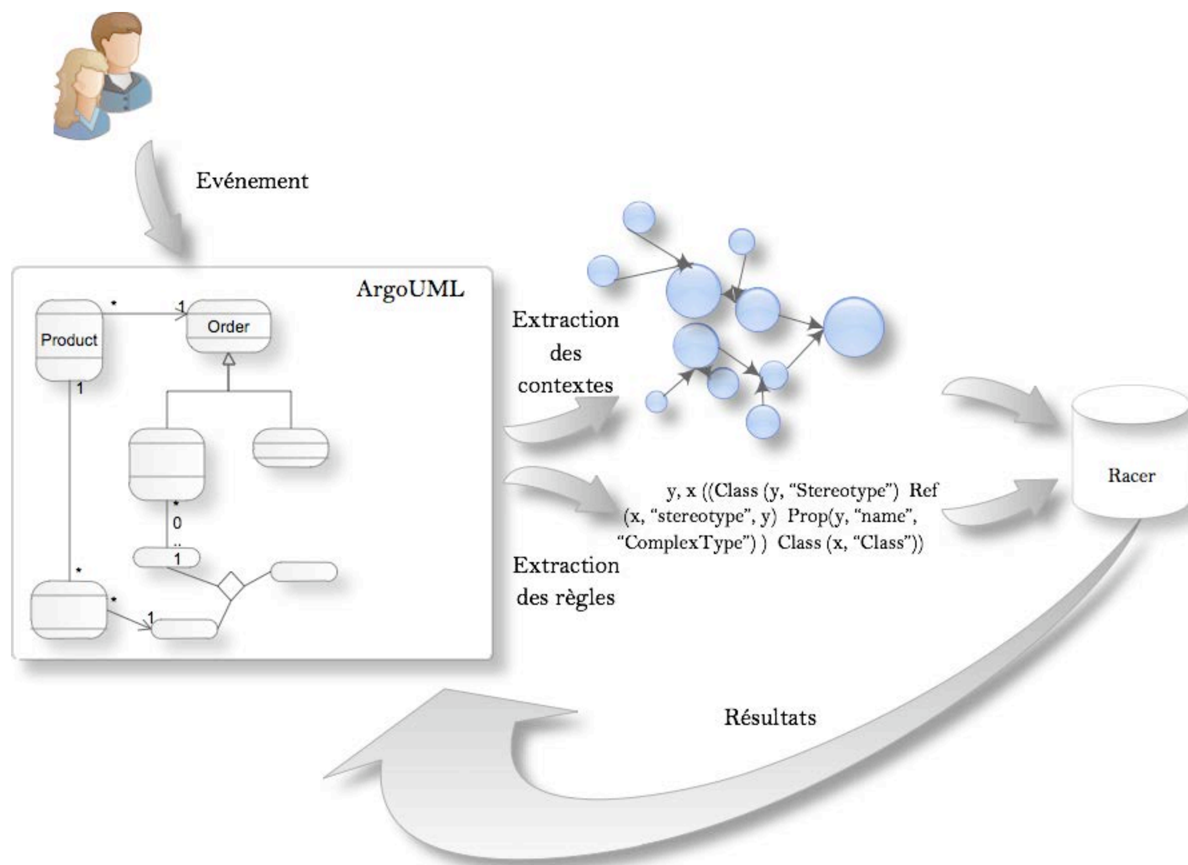


Figure 7.18. Processus de validation incrémentale introduit dans ArgoUML.

Dans la figure 7.18, nous pouvons voir comment est articulé notre processus de validation autour d'ArgoUML et de Racer. Comme nous l'avons mentionné précédemment, une des raisons pour laquelle nous avons opté pour Racer est que ce moteur d'inférence dispose d'une API Java qui permet de l'exploiter par l'intermédiaire d'une application externe, dans notre cas ArgoUML. Dans notre processus de validation, nous appliquons l'algorithme que nous avons défini dans la section 7.4.4. Lorsqu'un événement intervient sur un modèle, le moteur de validation que nous avons créé est chargé en tâche de fond pour traiter automatiquement cet événement et le valider. La validation de cet événement est réalisé par le calcul des contextes dépendants de celui-ci et des règles appliquées sur les

éléments composant ces contextes. L'ensemble des contextes et des règles est transmis via une API Java à Racer qui est chargé de « raisonner » sur les parties du modèle à valider. Le résultat du processus d'inférence réalisé par Racer est retourné à ArgoUML. Nous traitons les messages envoyés par Racer pour générer un rapport de validation exprimant les éventuelles erreurs issues de l'événement sur le modèle. La figure 7.19 illustre l'affichage d'un rapport de validation au sein de ArgoUML.

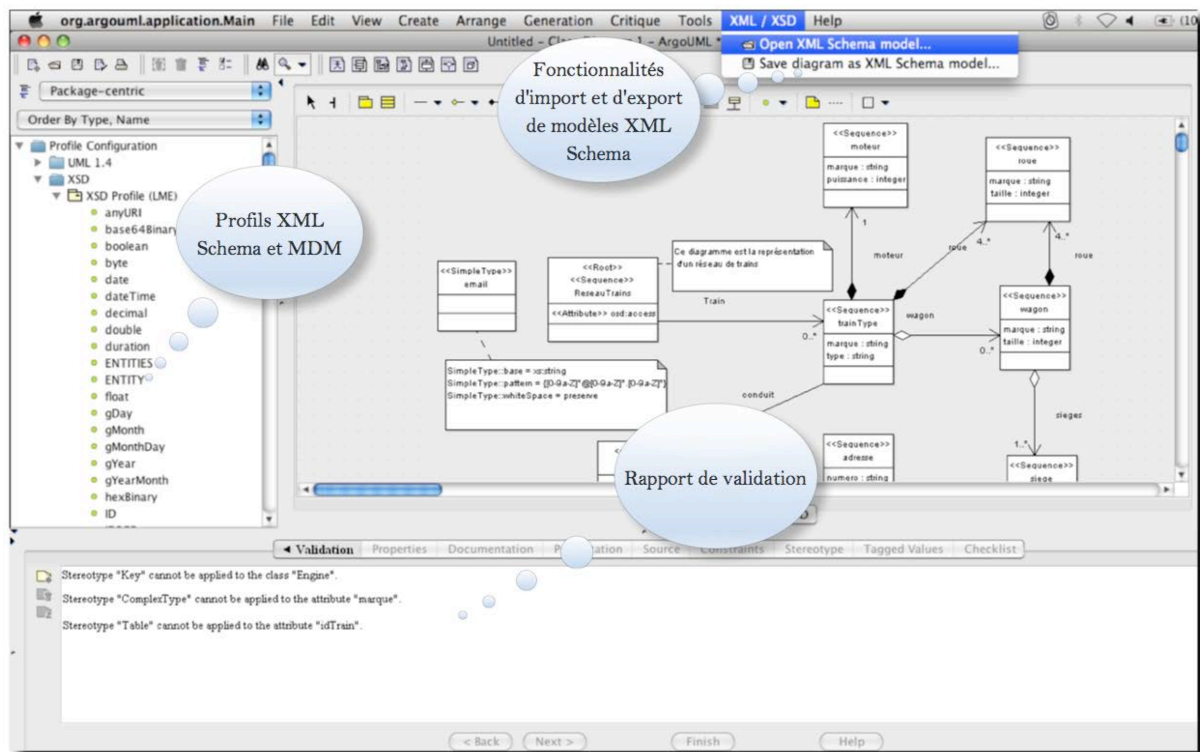


Figure 7.19. Rapport de validation ArgoUML.

7.5.6. Résultats expérimentaux

Dans cette section nous proposons d'expérimenter notre approche de validation incrémentale sur deux modèles de tailles différentes. Pour que ces résultats soient représentatifs nous choisissons de comparer les performances de notre approche de validation incrémentale avec une validation classique non incrémentale. Les critères de comparaison que nous avons retenus sont le temps d'une première validation d'un modèle, le temps d'une revalidation complète, le temps d'une troisième validation après avoir effectué une action d'insertion, de suppression ou de modification sur un modèle. Nous choisissons de ne pas considérer les aspects associés aux thématiques d'occupation mémoire dans la mesure où ces aspects peuvent fortement varier d'une machine à une autre.

Pour une première expérimentation nous choisissons un modèle de petite taille celui correspondant à notre modèle de publication d'ouvrages que nous avons utilisé tout au long de ce mémoire.

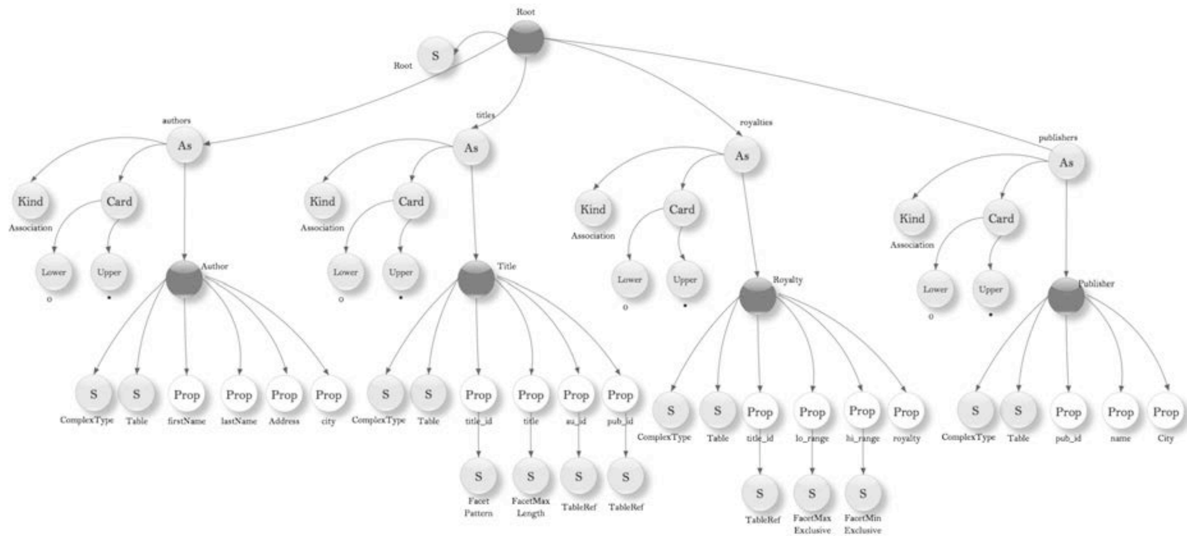


Figure 7.20. Graphe d'un modèle de publication d'ouvrages.

La figure 7.20 présente le graphe associé au modèle de publication d'ouvrages. La volumétrie de ce graphe est la suivante :

- Nombre de nœuds : 56 ;
- Nombre d'arcs : 55.

Evénement	Validation classique Temps (ms)	Validation incrémentale Temps (ms)
Première validation	187 ms	190 ms
Seconde validation sans modifications	180 ms	16 ms
Validation après insertion d'un élément simple	189 ms	40 ms
Validation après modification d'un élément simple	181 ms	31 ms
Validation après suppression d'un élément complexe	123 ms	31 ms

Figure 7.21. Résultats sur un modèle de petite taille.

À l'issue de cette première expérimentation sur un modèle de petite taille, nous pouvons constater que notre approche permet d'optimiser les temps de validation quelque soit l'événement s'étant produit sur le modèle. En utilisant une méthode classique de validation qui consiste à revalider systématiquement le modèle dans son intégralité nous pouvons remarquer que les performances demeurent constantes, excepté lorsqu'un élément complexe est supprimé dans la mesure où la taille du modèle a été réduite suite à cette suppression.

En utilisant notre approche notre étude montre que les temps de validation sont drastiquement diminués et cela pour tout évènement produit. Dans le cas d'une seconde validation sans modifications du modèle notre approche permet de réutiliser les informations issues de la première validation ce qui nous permet d'obtenir une réduction de temps de l'ordre du dixième du temps nécessaire par une méthode de validation classique. De même lorsqu'un événement intervient sur le modèle les validations suivantes sont en moyenne cinq fois plus rapides. En effet notre algorithme de validation permet de détecter quelles sont les parties du modèle à valider, ainsi seules ces parties sont soumises à revalidation.

En dépit du fait que le modèle utilisé pour cette première expérimentation soit de petite taille, les résultats obtenus par notre méthode de validation incrémentale sont encourageants concernant l'application de notre approche sur des modèles de tailles importantes.

La figure 7.22 présente les résultats de notre méthode de validation incrémentale appliquée à un modèle de grande taille. La volumétrie du graphe associé est la suivante :

- Nombre de nœuds : 5000;
- Nombre d'arcs : 4999.

Evénement	Validation classique Temps (ms)	Validation incrémentale Temps (ms)
Première validation	2777 ms	2808 ms
Seconde validation sans modifications	2802 ms	20 ms
Validation après insertion d'un élément simple	2910 ms	61 ms
Validation après modification d'un élément simple	2951 ms	34 ms
Validation après suppression d'un élément complexe	2658 ms	228 ms

Figure 7.22. Résultats sur un modèle de grande taille.

Sur un modèle de grande taille notre approche demeure très performante par rapport à une méthode classique de validation. L'optimisation apportée varie entre le centième et le dixième de temps nécessaire à une validation classique. Le gain le plus évident concerne la revalidation lorsqu'aucun élément du modèle n'a été modifié. Nous pouvons constater dans notre approche qu'à l'issue de la suppression d'un élément complexe nous obtenons un temps de revalidation plus important que ceux associés aux évènements d'insertion et de modification. Cela est dû au processus d'évaluation des contextes associés à l'élément supprimé qui peut être plus ou moins performant selon la complexité du modèle manipulé.

7.6. Conclusion

Dans les chapitres précédents de ce mémoire, nous avons abordé l'utilisation d'une démarche IDM pour définir des modèles de données XML Schéma. Cette démarche a permis d'obtenir une couche d'abstraction par l'intermédiaire d'un profil UML dédié à la sémantique de XML schéma. Pour passer d'un niveau abstrait à un niveau concret, nous avons défini des mappings et les avons mis en pratique via une application que nous avons développée. Nous avons proposé dans ce chapitre une approche incrémentale de validation de modèle afin de compléter notre méthode IDM. Nous avons défini un cadre mathématique pour définir d'une manière formelle des métamodèles et nous exploitons la puissance d'expressivité de la logique du premier ordre pour définir des règles de validation. La formalisation mathématique que nous avons introduite permet d'appliquer notre méthode de validation incrémentale à tout type de modèle. Notre approche de validation incrémentale est basée sur des notions de contexte de validation et d'événement qui nous permettent de localiser au mieux les parties d'un modèle à valider lorsqu'une modification intervient, cela nous permet d'optimiser, comme ont démontré nos expérimentations, le processus de validation sur les modèles de taille importante.

Chapitre 8

Conclusions et perspectives

8.1 Contributions

Dans ce mémoire, nous avons présenté une approche d'Ingénierie Dirigée par les Modèles (IDM) appliquée à une nouvelle approche d'intégration de données qu'est la gestion des données de référence ou *Master Data Management (MDM)* [Menet *et al.*, 2009a] [Menet *et al.*, 2009c]. Nos travaux se sont déroulés en plusieurs étapes. La première étape a consisté à étudier les avantages et les inconvénients des différentes approches d'intégration de données, à présenter leurs limitations principales afin de justifier la nécessité d'une nouvelle approche de fédération de données. Dans un second temps, nous avons présenté le *MDM* qui est une approche récente et émergente comme une solution aux limitations des différentes méthodes d'intégration de données existantes. Le *MDM* représente de manière indéniable une approche adéquate aux problèmes d'intégration de données ; cependant, nous nous sommes heurtées aux difficultés présentes lors des phases de conception de modèles de données pivot. En effet, il s'avère que la définition de modèle de données pivot nécessite une connaissance approfondie de la technologie sous-jacente par les acteurs impliqués dans le processus de définition d'un modèle de données. C'est pourquoi, nous avons proposé l'introduction d'une démarche guidant les concepteurs de modèle de données de façon qu'ils puissent se concentrer uniquement sur la modélisation et l'intégration de données et non sur la technologie à utiliser. L'adoption d'une approche objet et standard pour améliorer la compréhension du modèle et la sémantique associée (sémantique *MDM* dans nos perspectives) semble être une voie simple et efficace. Aussi, l'objectif principal de cette thèse a été de proposer une approche d'Ingénierie Dirigée par les Modèles pour faire abstraction de la couche technologique (physique) au profit de la couche fonctionnelle (logique), soit, en d'autres termes, d'apporter une nouvelle vision unifiée permettant de concevoir des modèles en séparant la logique métier de l'entreprise de toute plateforme technique. Pour ce faire, notre démarche a été divisée en deux parties :

- définition de profils UML permettant de représenter de manière abstraite la sémantique associée aux modèles de données pivot [Menet *et al.*, 2008a] [Menet, 2008c],

- définition de règles de transformation afin de rendre automatique et transparent le passage du niveau fonctionnel au niveau technologique [Menet, 2008a] [Menet, 2008b] [Menet *et al.*, 2008c].

Il est à noter qu'en complément de notre approche d'Ingénierie Dirigée par les Modèles, nous avons proposé aussi une nouvelle méthode de validation de modèles fondée sur une approche incrémentale [Menet *et al.*, 2009b]. En effet, quels que soient les domaines dans lesquels ont été introduites des méthodologies IDM, l'aspect *incrémental de validation de modèles* est très peu, voire nullement abordé. Nous pensons que cet aspect doit faire partie d'une approche IDM dans la mesure où nous devons garantir l'intégrité d'un modèle par rapport à la plateforme sur lequel celui-ci est déployé. Notre approche de validation s'est essentiellement focalisée sur des moyens d'optimisation des processus de validation pour des modèles de taille importante, à l'échelle industrielle.

8.1.1 Etude des approches d'intégration de données

Dans le second chapitre de ce mémoire, dans le contexte de l'interopérabilité de sources de données hétérogènes, nous avons fait apparaître, selon les objectifs visés, deux principales approches d'intégration de données à savoir l'approche virtuelle (ou par médiateur) et l'approche matérialisée (ou par entrepôt).

L'approche virtuelle, ou par médiateur, désigne une vision globale par l'intermédiaire d'un unique schéma de représentation d'un ensemble de sources de données hétérogènes. Ce schéma global peut être défini automatiquement à l'aide d'outils, ou extracteurs de schémas. Dans ce contexte, les données sont stockées uniquement au niveau des sources. Les traitements sont donc synchronisés sur ces sources de données. Un médiateur connaît le schéma global et possède des vues abstraites sur les sources qui lui permettront de décomposer la requête initiale en sous-requêtes. Le médiateur soumet ces sous-requêtes à des adaptateurs qui ont pour fonction de traduire ces dernières dans des langages compréhensibles par les différentes sources de données. Une fois le traitement de ces requêtes réalisé par ces sources, les réponses suivent le cheminement inverse jusqu'à l'utilisateur.

Dans l'approche d'intégration matérialisée, les données issues de sources hétérogènes sont copiées dans un entrepôt de données (ou référentiel). Les actions sur le référentiel sont asynchrones par rapport aux sources. La propagation des modifications apportées au référentiel vers les différentes sources de données doit passer par des procédures de mises à jour. Contrairement à l'approche virtuelle, les requêtes utilisateurs sont directement exécutées dans le référentiel, sans avoir à accéder aux différentes sources de données. Ici, les données du référentiel sont déconnectées de celles contenues dans les sources hétérogènes. Les mises à jour des données du référentiel vers les sources (et réciproquement) sont déléguées à un intégrateur qui a pour fonction de réaliser la correspondance entre le schéma du référentiel et les sous-schémas des sources.

L'avantage de l'approche virtuelle par rapport à l'approche matérielle tient au fait qu'il n'y a plus de problème de taille de la base de données et de problème de cohérence. Les

données demeurent, en effet, dans les sources locales ; le médiateur se contente alors de maintenir le schéma global et les traducteurs permettent de réaliser les requêtes. Cependant, avec cette solution, les temps de recherche augmentent car le nombre de traitements à effectuer est plus important. En effet, le médiateur doit traduire les requêtes exprimées par les clients à l'aide du schéma global et du langage d'interrogation de la médiation pour qu'elles soient compréhensibles par les sources locales, et ceci, pour chacune des sources locales auxquelles il va s'adresser. Il doit ensuite communiquer les requêtes à chacune des sources locales, traduire et agréger les réponses qu'elles lui retournent pour présenter aux clients une réponse homogène exprimée à l'aide du schéma global.

Les approches virtuelles et matérialisées ont longuement été étudiées dans différents travaux et présentent d'indéniables qualités mais comportent aussi des limitations en terme de standardisation, d'outils, de complexité de mise en place dans des contextes industriels. De plus, certains de ces systèmes se focalisent uniquement sur certaines problématiques telles que le traitement des requêtes ou l'intégration et la diffusion de données et ne traitent pas certains aspects importants de gestion des données elles-mêmes telles que :

(i) la mise en place de rôles et de droits d'accès pour accéder à une donnée,
(ii) l'uniformisation de la représentation des données au sein d'un système d'information par l'intermédiaire d'un outil de gestion unique,
(iii) la mise à disposition de moyens permettant d'auditer et d'historiser les données,
(iv) la gestion de manière efficace et performante du cycle de vie des données et de l'accès concurrentiel entre différents utilisateurs.

8.1.2 Présentation d'une nouvelle approche d'intégration de données

Pour outrepasser les limitations des approches existantes d'intégration de données, le *Master Data Management* (MDM) a été défini comme une approche visant à pallier leurs lacunes tout en se focalisant sur l'ensemble des problématiques de fédération de sources de données. Nous avons présenté dans le troisième chapitre de ce mémoire le Master Data Management comme étant une approche émergente d'intégration de données basée sur l'approche matérialisée. Le MDM étant une discipline récente, très peu de travaux existent à ce jour. Par rapport à l'approche matérialisée, le MDM se focalise de plus sur l'unification des modèles et outils au sein d'un système d'information. Nous avons présenté notre solution MDM *EBX.Platform* qui permet de répondre de manière générique à un ensemble de fonctionnalités liées aux problématiques du MDM :

- *Gestion des versions de données.* Une même donnée peut avoir des valeurs différentes selon le contexte dans lequel elle est valorisée,
- *Héritage des valorisations selon un arbre de contextes.* Par exemple, à partir d'une valorisation par défaut des données, il est possible de créer des contextes enfants qui permettront de surcharger les valeurs par défaut,

- *Gestion des habilitations.* Des rôles et des droits d'accès peuvent être définis afin de garantir ou de restreindre l'accès aux données,
- *Sécurité.* L'authentification à EBX.Platform s'effectue d'une manière sécurisée (HTTPS, SSO, etc.) par l'intermédiaire d'un protocole interne de gestion d'annuaires,
- *Accessibilité et convivialité.* EBX.Manager représente une interface utilisateur unique permettant d'accéder de manière simple et conviviale aux données,
- *Interrogation des données.* Une seule et unique application est utilisée pour accéder aux données,
- *Qualité des données.* La définition de contraintes sur les données garantit une conformité de celles-ci par rapport au modèle. La gestion de l'héritage entre instances permet d'éviter les problématiques liées à la redondance de données,
- *Historisation et version des données.* Il est possible d'auditer un référentiel en exploitant des mécanismes d'historisation et de gestion de différentes versions des données conservées au cours du temps.

8.1.3 Introduction d'une méthode d'Ingénierie Dirigée par les Modèles et premier profil UML dédié au MDM

La contribution de cette thèse a été l'introduction d'une approche d'Ingénierie Dirigée par les Modèles (IDM), dont nous avons présenté les principes dans le quatrième chapitre de ce mémoire, pour définir des modèles de données pivot dans le domaine du Master Data Management [Menet, 2008d], et ainsi faire abstraction de formats « propriétaires ». Comme base de nos travaux nous avons exploité l'outil MDM générique *EBX.Platform* basé sur une architecture XML. Les travaux que nous avons menés pour intégrer une approche IDM afin d'uniformiser la modélisation des schémas ont été de deux types à savoir la mise en place d'une solution de représentation générique et l'automatisation de transformations de modèles.

Concernant la représentation générique, nous avons défini dans le cinquième chapitre de ce mémoire un formalisme abstrait permettant de manipuler des modèles *XML Schema.* Dans ce contexte, notre approche s'est orientée vers les standards préconisés par l'OMG en proposant une extension du métamodèle UML, sous la forme de deux profils UML :

(i) un premier profil UML dédié à la sémantique de documents XML Schema,
(ii) un second profil UML dédié à la sémantique des modèles d'adaptation définie dans le contexte du Master Data Management.

Notre contribution majeure à la phase de modélisation a été de définir *le premier profil UML* associé au Master Data Management. En effet, le Master Data Management étant une discipline émergente dans le domaine de l'intégration de données, très peu de travaux existent à ce jour.

Dans le sixième chapitre de ce mémoire, il a été aussi question de définir un ensemble *de règles de transformation* permettant d'établir une projection d'un modèle XML Schema vers

un modèle UML et inversement tout cela en restant dans l'optique d'utiliser les standards recommandés par l'OMG tels que XMI et XSLT.

8.1.4 Méthode de validation incrémentale dans le domaine de l'IDM

Dans la littérature associée au domaine de l'Ingénierie Dirigée par les Modèles, une grande place est consacrée à la cohérence des modèles ou autrement dit à la validité des modèles par rapport à différentes contraintes posées par un métamodèle. Cependant, les approches classiques de validation de modèles reposent sur la vérification intégrale d'un modèle quel que soit le nombre de modifications intervenues. En effet, lorsqu'un modèle est modifié, il est nécessaire de valider à nouveau l'ensemble du modèle pour vérifier si la modification apportée n'a pas entraînée une incohérence dans la structure du modèle, et non un sous-ensemble de celui-ci qui serait uniquement impacté par une modification. Pour répondre à ce problème, nous avons proposé dans le septième chapitre de ce mémoire une nouvelle méthode de validation basée sur une approche incrémentale. Notre approche de validation incrémentale a pour but de réutiliser les informations issues des contrôles effectués lors de chaque modification unitaire et ainsi procéder à la validation de sous-parties d'un modèle uniquement lorsqu'il est nécessaire [Menet *et al.*, 2009b]..

L'approche que nous avons présentée est fondée sur une *formalisation* de la notion de métamodèle en un objet mathématique le rendant manipulable et dans le but d'exprimer des règles de validation et de cohérence structurelle sous la forme de formules de la *logique du premier ordre*. Pour représenter les sous-parties d'un modèle qui sont potentiellement impactées lors de chaque modification atomique, nous avons utilisé une approche par graphes et nous avons introduit la notion de *contexte de validation*. Cette notion de contexte de validation nous permet de calculer au plus près les parties d'un modèle à vérifier. Après avoir introduit cette notion de contexte de validation, il a fallu déterminer quelles sont les règles de validation à vérifier sur un contexte donnée. Pour répondre à cette problématique, nous avons considéré que pour une action donnée seul un ensemble de règles est à vérifier. En effet, une classification des règles à vérifier nous permet d'indiquer quelles sont les règles à valider lorsqu'une action précise se produit et de ne pas vérifier les règles non impliquées.

En mettant en place ce mécanisme de contexte, nous offrons une méthodologie permettant d'optimiser le processus de validation en considérant que seules les modifications affectant un type de contexte peut en entraîner sa vérification. Une des plus-values de notre approche est *d'être applicable à tout type de modèles de données*, à condition de formaliser le métamodèle associé aux modèles à valider.

8.2 Perspectives

Les travaux que nous avons menés tout au long de cette thèse ont permis de résoudre certains problèmes liés à la définition de modèles de données ; mais ils ouvrent aussi de nouvelles perspectives dans le domaine de l'Ingénierie Dirigée par les Modèles.

8.2.1 Normalisation du profil UML appliqué au MDM

Nous avons initié durant nos travaux le premier profil UML appliqué à la sémantique du Master Data Management. Il s'agirait dans la continuité de nos travaux de normaliser le profil UML que nous avons défini. Cela conduira à standardiser la définition et l'utilisation de modèles appliqués au Master Data Management. Pour ce faire, nous proposerons nos profils UML au W3C qui est chargé de la normalisation de différents standards. La standardisation de nos profils permettra de s'inscrire dans les problématiques d'interopérabilité et d'accessibilité qui sont les principaux objectifs du W3C. En construisant des modèles selon les standards du W3C, le concepteur obtiendra la garantie que tous ses modèles seront conçus selon des normes et seront interprétés de la même manière quelle que que soit la plateforme de déploiement.

8.2.2 Introduction d'une approche ontologique

L'approche de validation incrémentale que nous avons proposée est basée sur des graphes et sur la logique du premier ordre. Nous comptons, dans la suite de nos travaux, améliorer notre méthode en exploitant les ontologies [Leenheer *et al.*, 2008] et le langage de représentation d'ontologies *OWL* [OWL, 2009]. Le langage d'ontologie Web *OWL* est conçu pour décrire des classes et leurs relations, lesquelles sont inhérentes aux documents et applications Web. OWL est actuellement utilisé afin de :

(i) Formaliser un domaine en définissant des classes et des relations ainsi que des propriétés sur ces classes et ces relations,
(ii) Définir des individus et affirmer des propriétés les concernant,
(iii) Raisonner sur ces classes et individus dans la mesure où le permet la sémantique formelle du langage OWL.

Nous avons utilisé les graphes afin de répondre à ces trois problématiques. Nous souhaitons utiliser les ontologies afin de représenter de manière standard des métamodèles (domaines) et les relations entre les entités (individus) les composant. Il s'agira aussi d'exploiter le principe d'ontologie modulaire afin de fragmenter un modèle en sous-parties ce qui permettra de réutiliser notre notion de contextes de validation dans un cadre ontologique.

8.2.3 Définition d'une approche de résolution d'erreurs de validation et expérimentation à grande échelle

Les problématiques de validation incrémentale peuvent faire à elles seules l'objet de thèses. Nous n'avons introduit qu'une petite partie de la problématique de validation incrémentale dans notre méthodologie d'Ingénierie Dirigée par les Modèles. En effet, notre méthode ne propose pas une résolution automatique des erreurs levées lors du processus de validation. L'utilisateur doit manuellement corriger les erreurs qui sont détectées lors de chaque événement. Pour améliorer notre méthode de validation, il serait intéressant de diriger nos travaux vers des méthodes de calcul d'ensembles de modifications correctrices [Egyed, 2007b] qui permettront de corriger de manière automatique les erreurs détectées, ou de proposer les corrections adéquates. Il s'agit aussi d'être capable de détecter les règles de validation s'excluant mutuellement [Mens *et al.*, 2006].

Dans la continuité de la méthodologie de validation incrémentale que nous avons introduite il s'agirait de pousser notre étude sur les aspects de performance sur des modèles utilisés à l'échelle industrielle. En effet, nous n'avons pas pu expérimenter notre méthode sur des cas concrets dans des contextes réels. Cependant au vu des première expérimentations que nous avons effectuées et des différentes approches existantes sur le sujet [Tsiolakis *et al.*, 2000] [Finkelstein *et al.*, 1994], il est fort probable que notre approche reste dans le même ordre de grandeur en terme d'optimisation.

Concernant les questions relatives à la complexité algorithmique de notre approche nous pouvons considérer que celle-ci est liée à la fois à la nature du modèle à vérifier et à la logique de description utilisée. En effet, la complexité liée au calcul des contextes de validation est liée à la taille d'un modèle donné et aux différentes relations définies dans celui-ci. Quant à la partie raisonnement, dans la mesure où nous avons utilisé une logique de description décidable et ayant une complexité qui, dans le pire des cas, a une complexité exponentielle, nous ne pouvons qu'être optimiste sur la complexité algorithmique de notre approche.

Bibliographie

Abiteboul *et al.*, (2004) Abiteboul, S., "Distributed information management with XML and Web services", Gemo Report number 317, Record 26(3), p54-66, 2004.

Abiteboul *et al.*, (2002) Abiteboul S., Cluet S., M.-C. Rousset, "The Xyleme Project", Computer Networks 39, 2002.

Abiteboul *et al.*, (2001) Abiteboul, S., Cluet, S., Ferran, G., and Rousset, M.C., "The Xyleme Project", Technical report, INRIA, Gemo Team, Novembre 2001.

Abiteboul *et al.*, (1999) Abiteboul, S., Buneman, P., Suciu, D., "Data on the Web: From Relations to Semistructured Data and XML", Morgan Kaufmann Publishers, (1999).

Abiteboul *et al.*, (1997b) Abiteboul, S., Goldman, R., Quass, D., Widom, J., "Lore: A database Management System for Semistructured Data", SIGMOD, 1997.

Abiteboul *et al.*, (1991), Abiteboul, S., Bonner, A., "Objects and views", In Proceedings of the SIGMOD Conference on Management of Data, (San Francisco, California, March, 1991.

Abrial, (1995) J.R. Abrial, "The B book", University Press. 1995.

ArgoUML, (2002) ArgoUML, http://argouml.tigris.org/, 2002.

Altova, (2002) Altova XMLSpy, http://www.altova.com/xmlspy

Ambler, (2004) Ambler S, "Persistence Modeling in the UML", Issue of Software Development, http://www.sdmagazine.com, 1999.

Avgeriou *et al.*, (2005) P. Avgeriou, N. Guelfi, and N. Medvidovic, "Software architecture description and UML", 2005, pp. 23-32.

Baader, (2003) F. Baader and W. Nutt, "Basic description logics", 2003.

Bahl *et al.*, (2007) P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, M. Zhang, "Towards Highly Reliable Enterprise Network Services Via Inference of Multi-level Dependencies", SIGCOMM'07, août 27 -31, 2007.

Baïna *et al.*, (2006) Baïna S., Panetto H., Benali K., "Apport de l'approche MDA pour une interopérabilité sémantique", Ingénierie des Systèmes d'Information (ISI). Volume 11. p.11-29. Octobre 2006.

Barbier *et al.*, (2001) F. Barbier, B. Henderson-Sellers, "The whole-part relationship in object modelling: A definition in cOlOr", Information and Software Technology, 43(1) : pp. 19-39, 2001.

Bardohl *et al.*, (1999) R. Bardohl, M. Minas, A. Schürr, and G. Taentzer, "Application of graph transformation to visual languages," in Handbook of Graph Grammars and Computing by Graph Transformation, H. Ehrig, G. Engels, H. J. Kreowski, and G. Rozenberg, Eds. World Scientific, vol. II: Applications, Languages and Tools, pp. 105-180, 1999.

Bechhofer *et al.*, (2003) S. Bechhofer, R. Möller, P. Crowther. "The dig description logic interface", in Proceedings of the 2003 Description Logic Workshop (DL 2003).

Bellahsène *et al.*, (2001) Bellahsène, Z., Baril, X., « XML et les systèmes d'intégration de données », Ingénierie des Systèmes d'Information (ISI-NIS), pp. 11-32, 2001.

Bézivin *et al.*, (2008) F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool", Science of Computer Programming, vol. 72, no. 1-2, pp. 31-39, Juin 2008.

Bézivin *et al.*, (2001) J. Bézivin , O. Gerbé, "Towards a Precise Definition of the OMG/MDA Framework", Automated Software Engineering (ASE'01), IEEE Computer Society Press, San Diego, USA. 2001.

Bohlen, (2007) M. Bohlen, "AndroMDA Model Driven Architecture Framework", http://galaxy.andromda.org/docs-3.2/, 2007.

Bonnet, (1999) P. Bonnet. « Prise en compte des sources de données indisponibles dans les Systèmes de Médiation », mémoire de doctorat, Université de Savoie, 1999.

Booch, (1993) G. Booch, "Object-Oriented Analysis and Design with Applications (2nd Edition)", Addison-Wesley Professional, Septembre 1993.

Jacobson *et al.*, (1994) I. Jacobson, M. Christerson, and L. L. Constantine, “The OOSE method: a use case-driven approach”, pp. 247-270, 1994.

Bornhovd, (1998) C. Bornhivd, “MIX: A Representation Model for the Integration of Web Based Data”, Technical report, Darmastat University of Technology, Germany, 1998.

Bouguettaya *et al.*, (1998) A. Bouguettaya, B. Benatallah, and A. Elmagarmid, “Interconnecting Heterogeneous Information Systems”, Kluwer Academic Press, Norwell, MA. 1998.

Bright *et al.*, (1992) M.W. Bright, A.R. Hurson, S.H. Pakzard, “A Taxonomy and Current issues in Multidatabase Systems”, In IEEE Computer Society, pp. 50-60, Mars 1992.

Cattel, (1994) R.G.G. Cattel, “The Object Database standard : ODMG-93”, Morgan Kaufmann, San Francisco, Californie, 1994.

Cattell *et al.*, (1999) Cattell R.G.G., Barry D., “The Object Data Standard : ODMG 3.0”, Morgan Kauffman Publishers, 1999.

Castagna *et al.*, (2009), G. Castagna, N. Gesbert, and L. Padovani, “A theory of contracts for web services”, ACM Trans. Program. Lang. Syst., vol. 31, no. 5, pp. 1-61, 2009.

Cali *et al.*, (2002) Cali, A., Calvanese, D., Giacomo, G.D., Lenzerini, M., “On the Expressive Power of Data Integration Systems”, in Proceedings of the International Conference on Conceptual Modeling, Springer-Verlag, Londre, Angleterre, 2002.

Calvanese *et al.*, (1998a), D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati, “Description logic framework for information intégration”, in Proceedings of KR-98, 1998.

Calvanese *et al.*, (1998b), D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati, “Source integration in data warehousing”, in Proceedings of DEXA-98. IEEE Computer Society Press, 1998.

Carey et al., (2000), Carey, M.J., Florescu, D., Ives, Z.G., Shnmugasundaram, J., Shekita, E.J., Subramanian, S.N., “XPERANTO: Publishing Object-Relational data as XML”, in Proceedings of the WebDB (Informal Proceedings), Dallas, Texas, pp. 105-110, 2000.

Carlson, (2001), Carlson D., "Modeling XML Applications with UML: Practical e-Business Applications", Addison-Wesley Inc., 2001.

Carlson, (2006), Carlson D., "Semantic Models for XML Schema with UML Tooling", In Proceedings of the 2nd International Workshop on Semantic Web Enabled Software Engineering, 2006.

Cate *et al.*, (2007), B. T. Cate, J. van Benthem, and J. Väänänen, "Lindström theorems for fragments of first-order logic", In LICS '07, Washington, DC, USA, IEEE Computer Society, pp. 280–292, 2007.

Cluet *et al.*, (1998) Cluet, S., Delobel, C., Siméon, J., and Smaga, K., "your Mediators Need Data Conversion!", in Proceedings of the International Conference on Management Data, pp. 177-188. 1998.

Chawathe *et al.*, (1994), S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, J. Papakonstantinou, J. Ulman, J. Widom, "The TSIMMIS Project – Integration of Heterogeneous Information Sources", 10th Anniversary Meeting of Information Processing Society of Japan. Tokyo, Japon, 1994.

Cluet *et al.*, (2000) Cluet, S., Siméon, J. Yatl, "A Functional and Declarative Language for XML", Technical report, INRIA-Bell Labs, 2000.

Codd, (1970) Codd, E.F., "A Relational Model of Data for Large Shared Data Bank", Communications of the ACM, pp. 377-387, Juin 1970.

Codd, (1972) Codd, E.F. Relational Completeness of Databases Sublanguages. Prentice Hall, Englewood. 1972.

Conrad *et al.*, (2000) Conrad R., Scheffner, D. and Freytag, J.C., "XML conceptual modeling using UML", in Proceedings of the 19th International Conference on Conceptual Modeling (ER'2000), 2000.

Corba, (2002) http://www.omg.org/technology/documents/formal/profile_corba.htm

Czarnecki *et al.*, (2003) K. Czarnecki and S. Helsen, "Classification of model transformation approaches", 2003.

Dickinson, (2004) I. Dickinson, “Implementation experience with the dig 1.1 specification”, Rapport technique HPL-2004-85, Digital Media Systems Laboratory, Hewlett-Packard, Bristol, 2004.

Draper *et al.*, (2001) Draper, D., Halevy, A., Weld, D., “The Nimble Data Integration System”, In 17th International Conference on Data Engineering (ICDE), (2001), pp. 155-160.

Duscika *et al.*, (1997) Duscika, M., Genesretii, M.R., “Query Planning in Infomaster”, Proceedings of the Symposium on Applied Computing (ACM). San Jose, Canada, 1997.

Ebert *et al.*, (1994) J. Ebert and G. Engels, “Structural and behavioural views on OMT-classes”, pp. 142-157, 1994.

Eclipse, (2009) http://www.eclipse.org/

EDOC, (2004) http://www.omg.org/technology/documents/formal/edoc.htm

Eastman *et al.*, (1999) Eastman, J., Jordan, D., Russeli, C., Schadow, O., Stanienda, T., Velez, F., Berler, M., Cattell, R.G.G., and Barry, D.K. “The Object Data Standard: ODMG 3.0”, Morgan Kaufmann Publishers, 1999.

eXMLMedia,(2009), e-XMLMedia, http://e-xmlmedia.com/inf/index.htm, 2009.

Egyed, (2007a) A. Egyed, "Uml/analyzer: A tool for the instant consistency checking of uml models", Software Engineering, International Conference, vol. 0, pp. 793-796, 2007.

Egyed, (2007b) A. Egyed., “Fixing inconsistencies in uml design models”, in Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Washington, DC, USA: IEEE Computer Society, pp. 292-301, 2007.

Engels *et al.*, (2002) G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen, "Consistency-preserving model evolution through transformations", pp. 212-227, 2002.

Fankhausser *et al.*, (1998) Fankhausser, P., Gardarin, G., Lopez, M., Munoz, J., Tomasic, A., “Experiences in Federated Databases: From IRO-DB to MIRO-Web”, in Proceedings of the International Conference on Very Large Data Bases (VLDB), New York, 1998.

Fernandez *et al.*, (1998) Fernandez, M., Florescu, D., Kang, J., Levy, A., and Suciu, D., “Catching the Boat with Strudel: Experiences with a Web-Site Management System”,

In Proceedings of the International Conference on Management of Data (SIMOD), ACM Press, Seattle, Washington, USA, pp. 414-425, 1998.

Finkelstein *et al.*, (1994) A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency handling in multiperspective specifications," IEEE Transactions on Software Engineering, vol. 20, no. 8, pp. 569-578, October 1994.

Folli *et al.*, (2007) A. Folli and T. Mens, "Refactoring of uml models using agg," ECEASST, vol. 8, 2007.

Friedman-Hill, (2001) E. Friedman-Hill. Jess, "The expert system shell for the Java platform", http://herzberg.ca.sandia.gov/jess/, août 2001.

Garcia-Molina *et al.*, (1997) Garcia-Molina, H., Papakanstantinou, Y., Quass, D., Rajarman, A., Sagiv, Y., Ullman, J., Vassalos, V., and Widom, J., "The TSIMMIS Approach to Mediation: Data Models and Languages", Journal of Intelligent information Systems (JIIS), Pays Bas, pp. 117-132, 1997.

Gardarin *et al.*, (2002) Gardarin, G., Mensch, A., and Tomasic, A., "An Introduction of the eXML Data Integration Suite", In International Conference on Extending Database Technology (EDBT), pp. 297-306, Prague, République Tchèque, Mars 2002.

Geiger *et al.*, (2006) L. Geiger and A. Zündorf, "Tool modeling with fujaba", Electronic Notes in Theoretical Computer Science, vol. 148, no. 1, pp. 173-186, février 2006.

Giese *et al.*, (2009) H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization", Software and Systems Modeling, vol. 8, no. 1, pp. 21-43, Février 2009.

Gravier, (1997) S. Gravier, "Coloration et produits de graphes". Mémoire de doctorat. Université Joseph Fourier, Grenoble. 1997.

Guerra *et al.*, (2004) E. Guerra and J. de Lara, "Event-driven grammars: Towards the integration of meta-modelling and graph transformation", pp. 54-69, 2004.

Goedicke *et al.*, (1999) M. Goedicke, T. Meyer, and G. Taentzer, "Viewpoint-oriented software development by distributed graph transformation: towards a basis for living with inconsistencies", pp. 92-99, 1999.

Goldfarb, (1991) Goldfarb, C., "The SGML Handbook", Clarendon Press. 1991.

Goldman *et al.*, (1999) Goldman, R., Widom, J. "Enabling query formulation and optimization in semi-structured databases", in Proceedings of 23rd International Conference on Very Large Data Bases (VLDB'97), Morgan Kaufmann, pp. 436-445, 1997.

Greenfield, (2001) J. Greenfield, "UML Profile For EJB", Rational Software Corp., Mai 2001.

Halevy *et al.*, (2003b) Halevy, A.Y., Ives, Z.G., Suciu, D., Tatarinov, I., "Schema mediation in peer data management systems", in Proceedings of International Conference on Data Engineering (ICDE), Bangalore, Inde, Mars 2003.

Haarslev *et al.*, (2000) V. Haarslev and R. Möller, "Expressive ABox reasoning with number restrictions, role hierarchies, and transitively closed roles", International Conference on Principles of Knowledge Representation and Reasoning (KR'2000), pp. 273–284. 2000.

Haarslev *et al.*, (2001) V. Haarslev, R. Möller, "Description of the racer system and its applications", in Proceedings of the International Workshop on Description Logics (DL-2001), Stanford, Californie, pp. 132-141, Août 2001.

Haas *et al.*, (1997) Haas, L. Miller, R., Niswonger, B., Rotii, M., Schwarz, P., Wimmers, E.L., "Transforming Heterogeneous Data with Database Middleware: Beyond Integration", IEEE Data Engineering Bulletin, 1997.

Halevy *et al.*, (2005) Halevy, A. Y., N. Ashish, D. Bitton, M. Carey, D. Draper, J. Pollock, A. Rosenthal, and V. Sikka, "Enterprise information integration: successes, challenges and controversies" In Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD'05), New York, NY, USA, pp. 778-787, 2005.

Hasselbring, (2000) Hasselbring, W., "Information System Integration", Communications of the ACM, pp. 33-38, 2000.

Henderson-Sellers, (2001) B. Henderson-Sellers, "Some Problems with the UML V1.3 Metamodel", 34th Annual Hawaii International Conference on System Sciences, 3-6 janvier 2001, Maui, Hawaii, 2001.

Hermann *et al.*, (2008) F. Hermann, H. Ehrig, and G. Taentzer, "A typed attributed graph grammar with inheritance for the abstract syntax of uml class and sequence diagrams," Electronic Notes in Theoretical Computer Science, vol. 211, pp. 261-269, April 2008.

Hoolet, (2004) http://owl.man.ac.uk/hoolet/

Horrocks, (1998) I. Horrocks, "The FaCT system", dans de Swart, H. (éditeur), Automated Reasoning with Analytic Tableaux and Related Methods : International Conference (Tableaux'98), n°1397 in Lecture Notes in Artificial Intelligence. Springer-Verlag, pp. 307-312. 1998.

Horrocks *et al.*, (2000) I. Horrocks, U. Sattler, and S. Tobies, "Reasoning with individuals for the description logic SHIQ", In David MacAllester, editor, Proceedings of the 17th International Conference on Automated Deduction (CADE-17), n°1831 in Lecture Notes in Computer Science, Germany, Springer-Verlag, 2000.

Heimbigner *et al.*, (1989) Heimbigner, D., McLeod, D., "A Federated Architecture for Information Management", ACM Transaction, pp. 253-278, 1995.

Hnetynka *et al.*, (2004) P. Hnetynka and M. Pise, "Hand-written vs. mof-based metadata repositories: the SOFA experience", in IEEE international conference and workshop on the engineering of computer-based systems N°11, Brno, République Tchèque, pp. 329-336, 2004.

IBM, (2004) IBM MDM, http://www01.ibm.com/software/data/ips/products/masterdata, 2004.

iWays, (2009) iWays Master Data Center, http://www.iwaysoftware.com, 2009.

Iyengar *et al.*, (1998) Iyengar S., Brodsky A., "XML Metadata Interchange (XMI) Proposal to the OMG Object Analysis & Design Task", Object Management Group (OMG), http://www.omg.org, 1998.

Jacobson *et al.*, (1999) I. Jacobson, G. Booch, and J. Rumbaugh, "The Unified Software Development Process", Addison-Wesley Object Technology Series, Addison-Wesley Professional, Février 1999.

Kalfoglou *et al.*, (2003) Y. Kalfoglou and M. Schorlemmer, "Ontology mapping: the state of the art", The Knowledge Engineering Review, vol. 18, n°01, pp. 1-31, January 2003.

Kay, (2001) M. H. Kay, "XSLT Programmer's Reference",2nd Edition, Peer Information, Avril 2001.

Kleppe *et al.*, (2003) A. Kleppe, J. Warmer, W. Bast, "MDA Explained: The Model Driven Architecture Practice and Promise", Addison Wesley Professional, 2003.

Kurtev *et al.*, (2003) Kurtev, I., Berg, K.V., Aksit M., "UML to XML-Schema Transformation: a Case Study", in Managing Alternative Model Transformations in MDA, Forum on specification and Design Languages, 2003.

Kurtev, (2008) L. Kurtev, "State of the art of QVT: A model transformation language standard", pp. 377-393, 2008.

Jarke *et al.*, (2000) Jarke, M., Lenzerini, M., Vassiliadis, P., Vassiliou, Y. "Fundamentals of Data Warehouses", Vernon Hoffner, Lawrence Technological University, (2000).

Lacroix *et al.*, (1997) Lacroix, Z., Delobel, C., Brèche, P., "Object Views and Database Restructuring", in Proceedings of the International Workshop on Database Programming Languages, Springer Verlag, (Colorado, USA, August, 1997), pp. 80-101, 1997.

Leenheer *et al.*, (2008) P. Leenheer and T. Mens, "Ontology evolution", in Ontology Management, ser. Computing for Human Experience, M. Hepp, P. Leenheer, A. Moor, and Y. Sure, Eds. Boston, MA: Springer US, vol. 7, ch. 5, pp. 131-176, 2008.

Lemesle, (1998) Lemesle, R., "Transformation Rules Based on Meta-Modelling", EDOC'98, La Jolla, California, 3-5 November 1998, pp. 113-122.

Lenzerini, (2002) Lenzerini, M., "Data Integration: A Theoretical Perspective", in Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 233-246. 2002.

Levendovszky *et al.*, (2002) Levendovszky T., Karsai G., Maroti M., Ledeczi A., Charaf H., "Model Reuse with Metamodel-Based Transformations", In Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools, pp. 166 -178, 2002.

Levy, (2001) Levy, A., "Answering Queries Using Views: A Survey", The International Journal on Very Large Data Bases, pp. 270-294. 2001.

Li *et al.*, (2001) Li, C., Chang, E., "On answering queries in the presence of limited access patterns", in Proceedings of the 8th International Conference on Database Theory, pp. 219-233. 2001.

Manolescu *et al.*, (2001) Manolescu, I., Florescu, D., Kossmann, D., "Answering XML Queries over Heterogeneous Data Sources", in Proceedings of 27th International Conference on Very Larges Data Bases, p241-250. Rome, Italie, Septembre, 2001.

Menet *et al.*, (2007) Menet L., Lamolle M., "Meta-modeling "object": expression of semantic constraints in complex data structures", International Symposium on Innovative Management Practices *(ERIMA'07)*, Biarritz, France, Mars 2007.

Menet *et al.*, (2008a) Menet L., Lamolle M., "Designing XML Pivot Models for Master Data Integration via UML Profile", International Conference on Enterprise Information Systems (ICEIS'08), Volume DISI, Barcelone, Espagne, 12-16 juin 2008. pp. 461-464, 2008.

Menet *et al.*, (2008b) Menet L., Lamolle M., "Towards a Bidirectional Transformation of UML and XML Models", in Proceedings of the 2008 International Conference on E-Learning, E-Business, Enterprise Information System and E-Government (EEE'08) 14-17 juillet, 2008, Las Vegas, Nevada, USA, 2008.

Menet *et al.*, (2008c) Menet L., Lamolle M., "Extension conceptuelle de méta-modèles XML et UML pour une transformation bidirectionnelle de modèles", International Conference of Web and Information Technologies (ICWIT'08), 29-30 juin 2008, Sidi Bel Abbes, Algérie.

Menet (2008a) Menet L., “Vers une transformation bidirectionnelle de modèles UML et XML Schema”, Manifestation des Jeunes Chercheurs en Sciences et Technologies de l’Information et de la Communication (MajecStic'08), Marseille, France, 29-31 octobre 2008

Menet (2008b) Menet L., “Enrichissement sémantique de méta-modèles XML et UML pour une transformation bidirectionnelle de modèles », Informatique des Organisations et Systèmes d'Information et de Décision (INFORSID'08), Fontainebleau, France, 27-30 mai 2008.

Menet, (2008c) Menet L., “How to share knowledge on XML Schema models using UML” In Proceedings of 12th International Symposium on the Management of Industrial and Corporate Knowledge (ISMICK'08), Rio de Janeiro, 3-5 novembre 2008.

Menet (2008d) Menet L., “Managing Master Data with XML Schema and UML”, International Workshop on Advanced Information Systems for Enterprises (IWAISE'08), 19-20 avril 2008, Constantine, Algérie.

Menet *et al.*, (2009a) Menet L., Lamolle M., “Gestion des Données de Références par une Approche d’Ingénierie Dirigée par les Modèles”, Ingénierie d'Entreprise et des Systèmes d'Information, IESI'09, en association avec INFORSID'09, Toulouse, France, 26 Mai 2009

Menet *et al.*, (2009b) Menet L., Lamolle M., “Incremental validation of models in a MDE approach applied to the modeling of complex data structures”, in proceedings of the 2009 International Conference on Semantic Web & Web Services, SWWS 2009, July13-16 juillet, 2009, Las Vegas, Nevada, USA, pp 107-113, 2009.

Menet *et al.*, (2009c) Menet L., Lamolle M., “A Model Driven Engineering Approach Applied to Master Data Management”, On the Move to Meaningful Internet Systems: OTM 2009, workshop on Ambient Data Integration (ADI), Vilamoura, Portugal, pp 19-28, 1-6 novembre, 2009.

Mens *et al.*, (2006) T. Mens, R. Van Der Straeten, and M. D'Hondt, "Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis", in

Proceedings of the International Conference MoDELS/UML, Lecture Notes in Computer Science, vol. 4199. Springer-Verlag, Octobre 2006.

Mia, (2009) Mia-software, http://www.mia-software.com/

Mignet *et al.*, (2003) Mignet, L., Barbosa, D., Veltri, P., "The XML Web: a First Study", in Proceedings of the Twelfth International Word Wide Web Conference (WWW'03), 2003.

Miller *et al.*, (2003) Joaquin Miller, Jishnu Mukerji, "MDA Guide", Object Management Group, Inc., Version 1.0.1, omg/03-06-01, June 2003.

Milo *et al.*, (1999) Milo, T., Suciu, D., "Type inference for Queries on Semi-structured Data", in Proceedings of the Symposium on Principles of database systems (ACM SIGMOD-SIGACT), ACM Press, pp. 215-226, 1999.

Naumenko *et al.*, (2001) A. Naumenko and A. Wegmann, "A formal foundation of the rm-odp conceptual framework", 2001.

Millan *et al.*, (2008) T. Millan, L. Sabatier, P. Bazex, C. Percebois. « NEPTUNE II Une plate-forme pour la vérification et la transformation de modèles », Dans : Génie Logiciel, GL & IS, Meudon - France, Numéro spécial « Une plate-forme pour la vérification et la transformation de modèles », Vol. 85, p. 30-34, juin 2008.

MOF, (2006) MetaObject Facility, http://www.omg.org/mof/

Nardi *et al.*, (2003) D. Nardi, R. Brachman, "An introduction to description logics", The Description Logic Handbook : Theory, Implementation and Applications. Cambridge University Press, pp. 544, 2003.

Nentwich *et al.*, (2003) C. Nentwich, W. Emmerich, and A. Finkelstein, "Flexible consistency checking", ACM Transactions on Software Engineering and Methodology, vol. 12, no. 1, pp. 28-63, janvier 2003.

Noy *et al.*, (2003) N. F. Noy, M. Crubezy, R. W. Fergerson, H. Knublauch, S. W. Tu, J. Vendetti, and M. A. Musen, "Protégé-2000: an open-source ontology-development and knowledge-acquisition environment", Annual Symposium proceedings / AMIA Symposium. AMIA Symposium, 2003.

Olivé, (2007) A. Olivé, "Conceptual Modeling of Information Systems". Springer, 2007.

OMG, (1999) Object Database Management Group, "The Object Data Standard : ODMG 3.0", Morgan Kauffman Publishers, 1999.

OMG, (2001) Object Management Group, "Model Driven Architecture", http://www.omg.org/mda/

OMG, (2002a) OMG, "ArcStyler MDA-Business, Transformer Tutorial", 2002.

OMG, (2002b) Object Management Group, "Response to the UML 2.0 OCL", http://www.omg.org/docs/ad/02-05-09.pdf ,2002.

OMG, (2005) Object Management Group, "MOF 2.0/XMI Mapping Speci cation", September 2005, Version 2.1, formal/05-09-01. 2.3.2., 2005.

OMG, (2006) Object Management Group. UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE). Request For Proposals, 2006.

Oracle, (2007) Oracle MDM suite, http://www.oracle.com/master-data-management/, 2007.

Orchestranetworks, (2000) Orchestranetworks, http://www.orchestranetworks.com, 2000.

Overbeek, (2006) I. J. F. Overbeek, "Meta object facility (mof) investigation of the state", 2006.

OWL, (2009), Web Ontology Language, http://www.w3.org/TR/owl2-overview/, 2009.

Ozu *et al.*, (1999) Ozu, M., Valduriez, P, "Principals of Distributed Database Systems", (2nd edition), Pertinence-Hall, Inc., 1999.

Padadimitriou, (2004) C.H. Padadimitriou, "Computational complexity", Addison-Wesley Publishing Company, Massachusetts, USA, 1994.

Parsia *et al.*, (2004) B. Parsia, E. Sirin, "Pellet : An OWL DL reasoner", Haarslev, V. et Möller, R., Proceedings of the International Workshop on Description Logics (DL'04), 2004.

Patel-Schneider *et al.*, (1999) P. F. Patel-Schneider and I. Horrocks, "Dlp and fact," in TABLEAUX '99: Proceedings of the International Conference on Automated

Reasoning with Analytic Tableaux and Related Methods, London, UK: Springer-Verlag, pp. 19-23, 1999.

Pitoura *et al.*, (1995) Pittoura, E., Bukhres, O., Elmagarmid, A., "Object Orientation in Multidatabase Systems", ACM Computing Surveys, pp. 141-195. 1995.

Pottinger *et al.*, (2000) Pottinger, R., Levy, A.Y., "A Scalable Algorithm for answering Queries Using Views", In Proceedings of the International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann Publishers Inc., pp. 484-495. 2000.

Pool *et al.*, (2003) J. Poole, D. Chang, D. Tolbert, and D. Mellor, "Common Warehouse Metamodel Developer's Guide", Wiley. Janvier 2003.

PST, (2005), UML Profile for Schedulability, Performance, and Time, version 1.1, http://www.omg.org/technology/documents/formal/schedulability.htm

Régnier-Pécastaing *et al.*, (2008) F. Régnier-Pécastaing, M. Gabassi, J. Finet, « MDM : Enjeux et méthodes de la gestion des données », collection InfoPro – Management des Systèmes d'Information, ISBN 978-2-10-051910-1, éditions Dunod, 2008.

Revault *et al.*, (1995) N. Revault, H.A. Sahraoui, G. Blain, J-F. Perrot, "A Metamodeling Technique: the MetaGen System", in proceedings of TOOLS Europe'95 proceedings, TOOLS 16, Prentice Hall. 1995.

Risch *et al.*, (2001) Risch, T., Josifovski, V., "Distributed Data Integration by Object-oriented Mediator Servers", Concurrency and Computation: Practice and Experience, 13(11), pp. 933-953. 2001.

Richters, (2002) M. Richters, M. Gogolla, "OCL: Syntax, semantics, and tools", pp. 447-450, 2002.

Robie, (2007) J. Robie, "Xml processing and data integration with xquery", IEEE Internet Computing, vol. 11, no. 4, pp. 62-67, 2007.

Routledge *et al.*, (2002) Routledge N., Bird L., Goodchild A., "UML and XML schema", In Proceedings of the 13th Australasian Database Conference, pp 157-166, Melbourne, Australie, 2002.

Rozenberg, (1997) G. Rozenberg, "Handbook of Grammars and Computing by Graph Transformation", World Scientific Publishing Company, janvier 1997.

Russel *et al.*, (2002) S.J. Russell, S. J. P. Norvig, "Artificial Intelligence : A Modern Approach", seconde édition, Prentice Hall. 2002.

Sabetzadeh *et al.*, (2005) M. Sabetzadeh and S. Easterbrook, "An Algebraic Framework for Merging Incomplete and Inconsistent Views", in Proceedings of the 13th IEEE International Requirements Engineering Conference (RE'05), Washington, DC, USA: IEEE Computer Society, pp. 306-318, septembre 2005.

Salay *et al.*, (2007) R. Salay, M. Chechik, S. Easterbrook, Z. Diskin, P. Mccormick, S. Nejati, M. Sabetzadeh, and P. Viriyakattiyaporn, "An eclipse-based tool framework for software model management", in eclipse'07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange. New York, NY, USA: ACM, pp. 55-59, 2007.

Schlidt *et al.*, (2000) D. G. Schmidt and F. Kuhns, "An overview of the real-time corba specification", Computer, vol. 33, no. 6, pp. 56-63, 2000.

Schmedding *et al.*, (2007) F. Schmedding, N. Sawas, and G. Lausen, "Adapting the rete-algorithm to evaluate f-logic rules", pp. 166-173, 2007.

Schmidt-SchauB *et al.*, (1991) M. Schmidt-SchauB, G. Smolka, "Attributive concept descriptions with complements", Artificial Intelligence, 48(l):l-26, 1991.

Schöning *et al.*, (2000) H. Schöning and J. Wäsch, "Tamino - an internet database system", pp. 383-387, 2000.

Sheth, (1999) Sheth, A.P., "Changing Focus on Interoperability in Information Systems: From System, Syntax, Structure to Semantics", Interoperating Geographic Information Systems, M.F. GoodChild, M.J. Egenhofer, R. Fegeas, and C.A. Kottman (eds), Kluwer Publishers. 1999.

Sheth *et al.*, (1990a) Sheth, A.P., Larson J., "Federated Database Systems and Managing Distributed, Heterogeneous and Autonomous Databases", ACM Transaction on database systems, pp. 140-173, 1990.

Sheth *et al.*, (1990b) Sheth A., Larson J., “Federated database systems for managing Distributed, heterogeneous and autonomous databases”, ACM Computing Surveys, 1990.

Silaghi *et al.*, (2005) R. Silaghi, F. Fondement, and A. Strohmeier, "Weaving mtl model transformations", pp. 123-138, 2005.

Sierra *et al.*, (2003) K. Sierra and B. Bates, “Head First EJB”, (Brain-Friendly Study Guides; Enterprise JavaBeans), O'Reilly Media, Inc., October 2003.

Sirin *et al.*, (2007) E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical owl-dl reasoner", Web Semantics: Science, Services and Agents on the World Wide Web, vol. 5, no. 2, pp. 51-53, June 2007.

Sourrouille *et al.*, (2002) J. L. Sourrouille and G. Caplat, "Constraint checking in uml modeling", in Proceedings of the 14th international conference on Software engineering and knowledge engineering (SEKE'02), New York, NY, USA: ACM, pp. 217-224, 2002.

SPEM, (2008), Software Process Engineering Meta-Model, version 2.0, http://www.omg.org/spec/SPEM/2.0/

Spivey, (1998) J.M. Spivey, “The Z notation : a reference manual”, Prentice Hall, 1998.

Steingerg *et al.*, (2009) D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, “EMF: Eclipse Modeling Framework”, 2nd Edition (Eclipse), Addison-Wesley Longman, Amsterdam, January 2009.

Softeam, (2009) Softeam Objecteering Software, http://www.objecterring.com, 2009.

Surnia, (2003), Surnia http://www.w3.org/2003/08/surnia/, 2003.

Taentzer, (2004) G. Taentzer, “Agg: A graph transformation environment for modeling and validation of software”, pp. 446-453, 2004.

Tekinerdogan *et al.*, (2003) B. Tekinerdogan, S. Bilir, and C. Abatlevi., “Integrating platform selection rules in the model driven architecture approach”, In Uwe Aßmann, Mehmet Ak¸sit, and Arend Rensink, editors, Model Driven Architecture: European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004,

Enschede, The Netherlands, June 2003 and Linköping, Sweden, June 2004. Revised Selected Papers, volume 3599 of Lecture Notes in Computer Science, pp. 159–173. Springer-Verlag, August 2004.

Tomasic *et al.*, (1998) Tomasic, A., Raschid, L., and Valduriez, P., "Scaling Access to Heterogeneous Data Sources with Disco", IEEE Transactions on Knowledge and Data Engineering, 10(5), pp. 808-823. 1998.

Tolvanen *et al.*, (2003) J. P. Tolvanen and M. Rossi, "Metaedit+: defining and using domain-specific modeling languages and code generators", in Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '03:), New York, NY, USA: ACM, pp. 92-93, 2003.

Tsarkov *et al.*, (2003) D. Tsarkov, I. Horrocks, "DL reasoner *vs.* first-order prover", in Proceedings of the 2003 Description Logic Workshop (DL'03), pp. 152-159, 2003.

Tsarkov *et al.*, (2006) D. Tsarkov and I. Horrocks, "Fact++ description logic reasoner: System description", Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 4130 LNAI, pp. 292-297, 2006.

Tsiolakis *et al.*, (2000) A. Tsiolakis, H. Ehrig, "Consistency Analysis of UML Class and Sequence Diagrams using Attributed Graph Grammars", Proceedings of Workshop on Graph Transformation Systems (GRATRA), pp.77-86, March 2000.

UML, (1997) Unified Modeling Language, http://www.uml.org, 1997.

UMT-QVT, (2005) UMT-QVT, "UML Model Based Tool", http://umt-qvt.sourceforge.net/, 2005.

Vargun, (1999) Vargun, A., "Semantic Aspects of Heterogeneous Data Base", 1999.

Varro *et al.*, (2005) G. Varro, A. Schürr, and D. Varro, "Benchmarking for graph transformation," in Proc. IEEE Symp. Visual Languages (VL/HCC), A. Amber and K. Zhang, Eds., University of Texas, Dallas, Los Alamitos: IEEE Computer Society Press, septembre 2005.

W3C, (1998) World Wide Web Consortium. Bray, T., Paoli, J., Sperberg-McQueen, and C.M., (1998) Maler, E. (editors), "Extensible Markup Language (XML) version 1.0", W3C Recommendation. 10 février 1998.

W3C, (1999) XSLT, Extensible Stylesheet Language Transformation, http://www.w3.org/TR/xslt, 1999.

W3C, (2001a) World Wide Web Consortium. Fallside, D.C. (editor), "XML Schema Part 0: Primer", W3C Recommendation 2 May 2001, Disponible en ligne http://www.w3.org/TR/2001/REC-xmlschema-0-20010502, 2001.

W3C, (2001b) World Wide Web Consortium. Fallside, D.C. (editor), "XML Schema Part 1: Structures", W3C Recommendation 2 May 2001, Disponible en ligne http://www.w3.org/TR/2001/REC-xmlschema-1-20010502 2001.

W3C, (2001c) World Wide Web Consortium, "XML Schema Part 2: Datatypes", W3C Recommendation 2 May 2001, Disponible en ligne http://www.w3.org/TR/2001/REC-xmlschema-2-20010502, 2001.

W3C, (2001d) World Wide Web Consortium. Sharon A., et al., (editors), "Extensible Stylesheet Language (XSL) version 1.0", W3C Candidate Recommendation 15 October 2001. Disponible en ligne http://w3.org/TR/2001/REC-xsl-20011015, 2001.

W3C, (2002) World Wide Web Consortium, DeRose, S, Daniel, R., Grossos, P., Maler, E., Marsh, J., Walsh, N. "XML Pointer Language (XPointer) version 1.0", W3C Working Draft 16 August 2002. Disponible en ligne http://www.w3.org/TR/2002/WD-xptr-20020816, 2002.

W3C, (2001f) World Wide Web Consortium. DeRose, S., Maler, E., Orchard, D. (editors), "XML Linking Language (XLink) version 1.0", W3C Recommendation 27 June 2001. Disponible en ligne http://www.w3.org/TR/2000/REC-xlink-20010627, 2001.

Watkins *et al.*, (2002) D. Watkins, M. Hammond, and B. Abrams, "Programming in the .Net Environment (Microsoft .Net Development)", Addison-Wesley Longman, Amsterdam, novembre 2002.

Wiederhold, (1999) Wiederhold, G., “Mediation to Deal with Heterogeneous Data Sources”, In Proceedings oft he Interoperability in Large-Scale Heterogeneous Systems Conference (intertop), pp. 1-16, 1999.

Wiederhold, (1992) Wiederhold, G., “Mediators in the Architecture of Future Information Systems”, In Readings in Agents, p38-49, 1992.

Wu *et al.*, (2002) Wu I.C., Hsieh S.H., ”An UML-XML-RDB Model Mapping Solution for Facilitating Information Standardization and Sharing in Construction Industry”, In Proceedings of the National Institute of Standards and Technology. Gaithersburg, Maryland, Septembre, 2002.

XMF Mosaic, (2007) XMF Mos aic, “The Xactium XMF Mosaic”, http://www.modelbased.net/www.xactium.com, 2007.

Zackari *et al.*, (1999) Zackari, G., G.I., Florescu, D., Friedman, M., al, “An Adaptive Query Execution System for Data Integration”, In Proceedings of the International Conference on the Management of Data (SIGMOD'99), 1999.

Zou *et al.*, (2004) Y. Zou, T. Finin, H. Chen, “F-owl : an inference engine for semantic web”, Third NASA-Goddard/IEEE Workshop on Formal Approaches to Agent-Based Systems. Greenbelt, Maryland, 2004.