REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOULOUD MAMMERI, TIZI-OUZOU

FACULTE DE GENIE ELECTRIQUE ET D'INFORMATIQUE

DEPARTEMENT D'INFORMATIQUE

MEMOIRE DE MAGISTER

En Informatique

Option : **Ingénierie des Systèmes Informatiques**

Présenté par :

Mlle DJEMA Nassima

Thème :

Contribution à la Maintenance des Applications Distribuées : Un Outil pour la Restructuration et la Maintenance des Bases de Données

Devant le jury d'examen composé de:

Mr	AHMED OUAMER Rachid	Professeur	U.M.M.T.O	Président
Mr	RASSOUL Idir	M.C.A	U.M.M.T.O	Rapporteur
Mme	AOUGHLIS Farida	M.C.A	U.M.M.T.O	Examinatrice
Mr	DAOUI Mhamed	M.C.A	U.M.M.T.O	Examinateur

Soutenu publiquement le : 08 Avril 2015

REMERCIEMENTS

Mes remerciements s'adressent en premier à Monsieur RASSOUL Idir, maître de conférences à l'UMMTO, pour son encadrement et ses précieux conseils. Sans ses encouragements, son immense patiente et ses critiques constructives, ce travail n'aurait jamais vu le jour.

Je tiens à présenter mes vifs remerciements à Monsieur AHMED OUAMER Rachid Professeur à l'UMMTO de me faire l'honneur de présider le jury et d'examiner ce travail.

Mes remerciements vont également à Madame AOUGHLIS Farida et Monsieur DAOUI Mhamed, maîtres de conférences à l'UMMTO, de m'avoir fait l'honneur d'accepter d'être membre du jury et de consacrer de leurs temps pour l'évaluation de ce modeste travail.

Je profite aussi pour remercier tous les enseignants de la poste graduation, pour tous les efforts consentis dans la transmission des connaissances et leur savoir faire.

Je remercie mes camarades de promotion, pour leurs disponibilités et leurs gentillesses. Grâce à eux, mon parcours à l'université de Mouloud Mammeri fût très agréable.

Ma profonde gratitude va à mes parents et à mon frère, sans lesquelles je ne serai pas là où j'en suis aujourd'hui. Merci pour leurs encouragements, leurs patience qui n'a pas de limite et surtout pour leurs soutien inestimable.

SOMMAIRE

INTRODUCTION GENERALE 10

CHAPITRE I : Contexte de la problématique

I.1. Introduction ……………………………………………... 15
I.2. Le système complexe ……………………………….... 15
I.3. Maintenance des applications informatiques ……….... 16
I.4. Modèles de données …………………………………... 20
I.5. Ingénierie dirigée par les modèles ………………….... 33
I.6. Conclusion …………………………………………….. 39

CHAPITRE II : Transformation des modèles

II.1. Introduction …………………………………….......... 41
II.2. Définitions de Transformation ……………………….. 42
II.3. Taxonomie des transformations…………………….... 43
II.4. Types de transformation de modèles……………….... 45
II.5. Approches de transformation…………………………. 47
II.6. Langages de transformation de modèles…………….... 50
II.7. Mécanisme de transformation des modèles ………….. 53
II.8. Génération de transformations de modèles…………… 55
II.9. Conclusion…………………………………………….. 58

CHAPITRE III : Refactoring des modèles

III.1. Introduction ……………………………………..….. 60
III.2. Principe du refactoring…………………………..…….. 62
III.3. Refactoring des modèles…………………….…...…. 63
III.4. Processus de refactoring ………………….……….. 64
III.5. Refactoring des diagrammes de classes…...………… 70
III.6. Approche de refactoring……………..……………… 72
III.7. Conclusion………………………..………………… 77

Chapitre IV : Cohérence et évolution des modèles

IV.1. Introduction ……..………………………………….. 79
IV.2. Cohérence dans les applications de base de données... 80
IV.3. Intégration des données……………......…………….. 82
IV.4. Evolution des modèles……………..………..……… 82
IV.5. Conclusion……………………………………………... 91

CHAPITRE V : Architecture de l'outil de maintenance

V.1. Introduction…………………………………………... 93
V.2. Principe de l'outil de maintenance …………….….. 93
V.3. L'analyse de concept formel (AFC) et l'analyse relationnelle……………………………………………... 95
V.4. Architecture de l'outil de maintenance………………. 98
V.5. Résultat ………………………………………………. 107
V.5. Conclusion…………………………………………… 108

CONCLUSION GENERALE…………………………………………….. 109

REFERENCES BIBLIOGRAPHIQUES ……..……………………….. 110

Liste des figures

Figure I.1 : Axes de la problématique 15

Figure I.2 : L'iceberg ………………………………………………………. 19

Figure I.3 : schéma du modèle E-A ……………………………………….. 22

Figure I.4 : Rôles d'une association dans modèle E-A ……………………… 23

Figure I.5 : Relation de généralisation/spécialisation ……………………….. 27

Figure I.6: Document XML et sa structure d'arborescence………………….. 30

Figure I.7 : Relations entre système, modèle, métamodèle et langage ……... 36

Figure I.8 : *Pile de modélisation* …………………………………………… 37

Figure I.9.*Principe du Processus MDA* ……………………………………… 39

Figure II.1 : *Schéma de base d'une transformation de modèles* ……………… 43

Figure II.2: transformation endogène à gauche et transformation exogène à droite ………………………………………………………………… 44

Figure II.3 : à gauche Transformation verticale, à droite transformation horizontale ……………………………………………………………… 45

Figure II.4 : Composantes de Q/V/T………..…………………………….. 51

Figure II.5 : Mécanisme de transformation des modèles ………………… 54

Figure III.1: Code source Java avant l'application du refactoring ………… 61

Figure III.2: Code source Java après l'application du refactoring ………… 61

Figure III.3 : Exemple de refactoring de modèle de classe ……………........ 64

Figure III.4 : Processus de refactoring 64

Figure III.5 : synchronisation des différents diagrammes d'un modèle……… 70

Figure IV.1: Eléments de l'approche Bounif ……………………………… 84

Figure IV.2 : Structure générale de l'approche de Cicchetti et al. …………. 86

Figure IV.3 : méta modèle de différence …………………………………… 87

Figure IV.4 : Modélisation classique définie en trois niveaux d'abstraction. 88

Figure IV.5 : Propagation vers l'aval des modifications déduites de (R1 ⟶ R1') et apportées à SC0……………………………………………… 89

Figure IV.6 : Propagation vers l'amont et vers l'aval des modifications déduites de (R2 ⟶ R2') et apportées à SL0………………………………... 90

Figure V.1 : Représentation de l'approche …………………………………... 94

Figure V.2 : Partie d'un modèle statique du domaine bancaire……………… 96

Figure V.3 : (a) codage des clases d'un modèle UML par un contexte binaire (b) le treillis de concept correspondant ………………………… 97

Figure V.4 Architecture de l'outil de maintenance ………………………. 98

Figue V.5 : processus de refactoring de modèle de classe UML en utilisant l'ARC ……………………………………………………………………. 102

Figure V.6 : Modèle globale du système d'information complexe.............. 108

Liste des Algorithmes

Algorithme 1 : Détermination de la pertinence d'un concept……………….. 105

Algorithme 2 : Génération d'un modèle structuré à partir d'une FTR……….. 106

Résumé

Un système complexe est défini comme un système composé d'un grand ensemble cohérent d'entités hétérogènes collaborant de manière unitaire, et en interaction permanente. Ceci se traduit par des systèmes d'information complexes et hétérogènes. La maintenance de ces systèmes revient en conséquence à la maintenance des bases de données. Ces bases de données sont hétérogènes, leur hétérogénéité vient du fait qu'elles ont été développées par plusieurs personnes qui selon leurs connaissances utilisent différents langages de développement, différentes plateformes et différents formalismes de représentations des données. Dans ce cadre l'ingénierie dirigée par les modèles (IDM) a pour principal objectif de relever un certain nombre de défis du génie logiciel (qualité, productivité, séparation des préoccupations, coût, etc.) en suivant une approche à base de modèles. Il s'agit d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles.

En s'appuyant sur le principe clé de l'IDM, la transformation des modèles, et les techniques de propagation dans les modèles, nous proposons une architecture d'un outil de maintenance des applications hétérogènes.

Dans ce présent travail, nous proposons un outil de maintenance des applications informatiques hétérogènes.

Mots- Clés : Système d'information, bases de données hétérogènes, modèles de données, modélisations, logiciel, aide à la maintenance, maintenance des bases de données hétérogènes.

Abstract

A complex system is defined as a system consisting of a large coherent set of heterogeneous entities collaborating in a unitary manner and in permanent interaction. This results in complex and heterogeneous information systems. Maintaining these systems comes accordingly to the maintenance of databases. These databases are heterogeneous; their heterogeneity is that they were developed by several people based on their knowledge using different development languages, different platforms and formalisms representations of data. In this context the model driven engineering (MDE) has the main objective to raise a number of software engineering challenges (quality, productivity, separation of concerns, cost, etc.) following a model-based approach. This is a form of generative engineering in which all or part of an application is generated from models.

By relying on the key principle of the MDE, processing models, and propagation techniques in models, we propose an architecture of a maintenance tool heterogeneous applications.

In the present work, we propose a maintenance tool heterogeneous computing applications.

Key-words: Information system, heterogeneous databases, data models, modeling, software, maintenance assistance, maintenance of heterogeneous databases.

Introduction générale

Introduction générale

Un système complexe est défini comme un système composé d'un grand ensemble cohérent d'entités hétérogènes collaborant de manière unitaire, et en interaction permanente. Un système d'information complexe est alors défini comme un ensemble cohérent de systèmes d'information hétérogènes. La maintenance de ce système revient alors à la maintenance des bases de données, composantes principales de ces systèmes d'information. Ces bases de données sont hétérogènes, leur hétérogénéité vient du fait qu'elles ont été développé par plusieurs personnes qui selon leurs connaissances utilisent différents langages de développement, différentes plateformes et différents formalismes de représentation des données.

La maintenance est l'un des problèmes majeurs en génie logiciel. Pourtant relativement peu d'informations sont diffusées quant à son amplitude réelle et quantitative. Dans le domaine de la recherche, la maintenance n'est pas non plus un sujet vraiment prisé. D'ailleurs, dans les premiers cycles de vie la maintenance n'apparaissait même pas. Aujourd'hui certains voient encore la maintenance et la recherche comme deux termes quelque peu contradictoires puisque l'un est tourné vers le passé et l'autre est tourné vers l'avenir.

Les modifications dans les bases de données, qu'elles soient des mises à jour, des ajouts ou des suppressions, ont un effet non seulement sur les constituants en relation avec l'artefact modifié au même niveau d'abstraction mais aussi avec les artefacts en relation d'un niveau d'abstraction supérieur. Autrement dit dans les bases de données, le niveau d'abstraction supérieur c'est les modèles de données. On parle alors de la maintenance des modèles ou de l'évolution des schémas de base de données.

Plusieurs études ont été faites sur l'impact des changements dans les bases de données. La plupart de ces études se concentrent sur une base ou un ensemble des bases de données homogènes, et peu d'entre elles se penchent sur la problématique de propagation dans les bases de données hétérogènes.

Parmi ces études on trouve celles qui propagent les modifications en se basant sur des algorithmes de comparaison des modèles pour créer le modèle de différence. D'autres utilisent des entrepôts de schéma pour regrouper tous les schémas hétérogènes des bases et construire des ontologies de besoins.

Toutes ces approches peuvent être caractérisées comme des approches ascendantes, car elles partent des données et propagent les modifications aux niveaux supérieurs, pour faire évoluer les modèles de représentations de ces données.

L'ingénierie dirigée par les modèles (IDM), ou Model Driven Engineering (MDE) en anglais, a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique. L'IDM a pour principal objectif de relever un certain nombre de défis du génie logiciel (qualité, productivité, séparation des préoccupations, coût, etc.) en suivant une approche à base de modèles [Bézivin et al. , 04]. Il s'agit d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles.

Dans cette nouvelle perspective, les modèles occupent une place de premier plan parmi les artéfacts de développement des systèmes, mais doivent en contrepartie être suffisamment précis et riches afin de pouvoir être interprétés ou transformés par des machines. Le processus de développement des systèmes peut alors être vu comme une séquence de transformations de modèles partiellement ordonnée [Diaw et al., 08], chaque transformation prenant un ou des modèles en entrée et produisant un ou des modèles en sortie, jusqu'à l'obtention d'artéfacts exécutables.

En s'appuyant sur le principe clé de l'IDM, la transformation des modèles, et les techniques de propagation dans les modèles, nous proposant une architecture d'un outil de maintenance des applications hétérogènes. Notre outil aura pour

principale fonction la création de modèle unique d'un système d'information complexe. C'est sur ce modèle que le chargé de maintenance appliquera ses changements qui se propageront ensuite sur tout le modèle.

Dans notre approche, au lieu de faire évoluer les différents schémas après une modification du code, d'une manière ascendante, le mainteneur aura une vue globale du système et procèdera à une maintenance descendante.

Notre outil va créer le modèle global du système à partir des différents modèles qui le constituent, et cela par un processus de transformation et de refactoring entre ces modèles. Le modèle généré sera un modèle de classe, avec comme classe du haut niveau du schéma résultant, une classe abstraite qui regroupera les attributs communs au différents modèles, et la base du schéma les classes propres à chaque modèle. Dans un schéma, une modification apportée à la classe abstraite se propagera sur tout le schéma, alors que celle apportée sur la classe de base n'aura aucun effet sur les autres classes.

Le mémoire est structuré en cinq chapitres comme suit :

Dans le chapitre I, nous présentons le contexte de la problématique. Nous définissons le concept des systèmes complexes, puis nous développons la maintenance des applications informatiques, les modèles de représentation des bases de données et enfin l'ingénierie dirigée par les modèles (IDM).

Le chapitre II sera consacré à l'étude approfondie sur la transformation des modèles, la taxonomie de ces transformations, les approches utilisées pour réaliser ces transformations et enfin à un panorama des langages et des outils manipulés.

Dans le chapitre III, nous introduisons l'ensemble des notions relatives au refactoring des modèles. Nous présentons le processus de refactorisation des modèles et quelques approches utilisées pour leur refactoring.

Le chapitre IV est dédié à une présentation sur les modèles de cohérence des données, les protocoles de cohérence et la gestion de l'évolution des modèles.

Dans le chapitre V, nous présentons notre contribution. Il s'agit de l'architecture de l'outil de maintenance des applications distribuées notamment les bases de données distribuées (BDD).

Enfin, nous terminons ce travail par une conclusion générale et des références bibliographiques.

Chapitre I

Contexte de la problématique

I.1. Introduction

Notre problématique se situe au carrefour de quatre grands axes de l'informatique. Le premier point définira le concept des systèmes complexes, puis en second on présentera la maintenance des applications informatiques ainsi que ces différents types.

Le troisième point va nous permettre de rappeler quelques notions sur les modèles de représentation des bases de données.

Enfin, nous clôturerons ce chapitre par des généralités sur l'Ingénierie dirigée par les modèles (IDM).

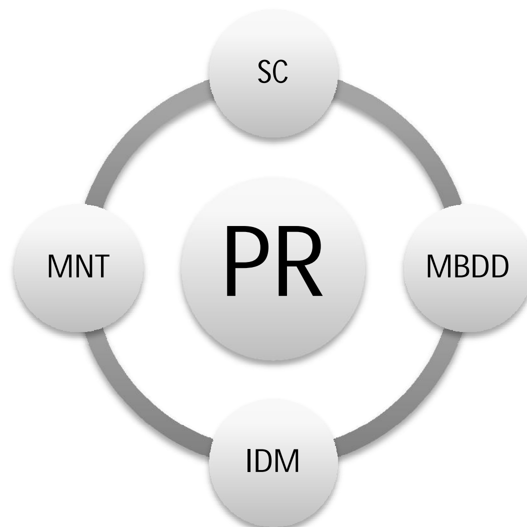


Figure I.1 : Axes de la problématique

I.2. Le système complexe

Un système complexe [Bouarioua, 13] est défini comme un système composé d'un grand ensemble cohérent d'entités hétérogènes collaborant de manière unitaire, et en interaction permanente, pour assurer une ou plusieurs fonctions du système dans son environnement. Les propriétés d'un système résultent de

l'interaction de ses constituants. Des propriétés émergentes sont parfois souhaitées et parfois indésirables. Nous pouvons dans ce cas dire que l'intérêt de la conception en ingénierie des systèmes est d'obtenir les comportements émergents attendus, en maintenant les comportements émergents non-attendus dans les limites acceptables. L'étude comportementale d'un système est alors incontournable.

Différents types de systèmes ont été identifiés dans la littérature parmi lesquels les systèmes distribués, qui sont des systèmes répartis, fonctionnant généralement les uns avec les autres dans un réseau.

La complexité est le facteur qui rend difficile la tâche de conception d'un système. Cette complexité dépend des points suivants :

Le nombre et la nature des éléments constituant le système : dans un système complexe, les éléments sont hétérogènes et le nombre d'éléments peut être élevé ou, éventuellement, variable.

La nature de son organisation interne : cette organisation concerne les relations entre les éléments du système, qu'ils soient organisés dans des réseaux ou des hiérarchies, etc.

Le couplage avec l'environnement : plus le système interagit avec son environnement, plus il y a des incertitudes sur son comportement.

I.3. Maintenance des applications informatiques

La maintenance s'oppose au développement. Alors que les problèmes qu'elle pose sont très importants, pendant longtemps ils ont tout simplement été ignorés. Bien qu'aujourd'hui la situation se soit améliorée, la maintenance reste l'une des principales zones d'ombre du génie logiciel.

En génie logiciel, presque tous les efforts pour diminuer les coûts de la maintenance consistent à améliorer la qualité des logiciels développés. Cette approche est indispensable. Pourtant elle est loin d'être suffisante.

La maintenance est définie comme étant l'ensemble des activités effectuées sur un logiciel après sa livraison ou sa mise en opération [Favre, 95].

La maintenance correspond à des tâches de correction suite à des erreurs intervenues dans les logiciels.

La maintenance des logiciels ne consiste pas uniquement en la correction d'erreurs. Les modifications d'un programme dû à un changement de ses spécifications ou de son environnement relèvent également de la maintenance. La correction d'erreurs ne constitue qu'un faible pourcentage de l'activité de maintenance.

Le standard IEEE de la maintenance du logiciel définit la maintenance comme étant la modification du logiciel après sa livraison pour corriger des défauts, améliorer les performances ou tout autre attribut, ou adapter le produit logiciel aux changements affectant son environnement [Zitouni, 96].

Selon les raisons amenant à modifier le logiciel au cours de la maintenance on distingue quatre types de maintenance: corrective, adaptative, perfective [Favre, 95], [Zitouni, 96] et préventive [Favre, 95]. Ces types de maintenance sont décrits dans les sections suivantes.

I.3.1 Maintenance corrective

Elle regroupe les différentes opérations effectuées après la détection des erreurs ou de défauts dans le fonctionnement du système. Ces opérations permettent de corriger les erreurs omises lors de la phase de test. Elles visent à éliminer les défaillances d'implémentation, les échecs du traitement et d'exécution, et assurer le bon fonctionnement du logiciel. A titre d'exemple, la correction des programmes qui ne fonctionnent pas et ceux qui produisent des faux résultats.

Notons que ce type de maintenance n'existerait pas si l'on était capable de développer des logiciels sans erreurs. Notons aussi que de la maintenance corrective ne génère pas de nouveaux profits. Elle est cependant indispensable.

I.3.2 Maintenance adaptative

Provoquée par une modification dans l'environnement du travail logiciel et/ou matériel suite à des besoins nouveaux ou changeant. Ce type de maintenance est indispensable afin de s'adapter aux besoins évolutifs du logiciel. Un changement dans la structure du code est un changement logiciel et un changement du système d'exploitation ou du matériel (sur lequel le logiciel est opérationnel) est un changement matériel.

I.3.3 Maintenance perfective

Effectuée à la demande des utilisateurs, elle réfère aux modifications apportées au logiciel afin d'augmenter ses performance ou ses fonctionnalités et d'éliminer l'inefficacité du traitement. Elle vise l'amélioration de la qualité du logiciel, la documentation ou autre facteurs de qualité. A titre d'exemple, l'optimisation du code source en vue d'augmenter la rapidité de son exécution. La maintenance perfective prend 60 à 70% de l'effort total de la maintenance.

Ce type de maintenance n'est pas toujours indispensable. Avant d'entreprendre une modification, il est donc important de comparer le coût de l'opération avec les bénéfices attendus.

I.3.4 Maintenance préventive

La maintenance préventive est l'ensemble des activités de maintenance ayant pour but d'améliorer la facilité de maintenance des logiciels.

Ce type de maintenance ne génère aucun bénéfice direct. De telles activités ont uniquement pour but de faciliter, dans le futur, les autres activités de maintenance. La maintenance préventive n'est pas déclenchée par un événement externe mais plutôt par une décision prise dans les services de maintenance.

La maintenance est l'un des problèmes majeur en génie logiciel. Pourtant relativement peu d'informations sont diffusées quant à son amplitude réelle et

quantitative. Dans le domaine de la recherche, la maintenance n'est pas non plus un sujet vraiment prisé. D'ailleurs dans les premiers cycles de vie, la maintenance n'apparaissait même pas [Favre, 95]. Aujourd'hui, certains voient encore maintenance et recherche comme deux termes quelque peu contradictoires puisque l'un est tourné vers le passé et l'autre tourné vers l'avenir. Il est souvent dit "Plutôt que de maintenir des logiciels de mauvaises qualités il serait préférable d'en construire de nouveaux". Cette affirmation est tout à fait sensée dans un contexte de recherche. Hélas elle n'est pas réaliste dans l'industrie si l'on considère des logiciels de grandes tailles. C'est justement le propos du génie logiciel.

Pour combattre ce manque d'intérêt, des images fortes ont souvent été employées pour faire ressortir le problème de la maintenance. Par exemple la maintenance a été comparée à un "iceberg". Cette image suggère que le coût de développement n'est que la partie visible du problème. La maintenance est méconnue et la sous-estimer c'est s'exposer à certains risques...

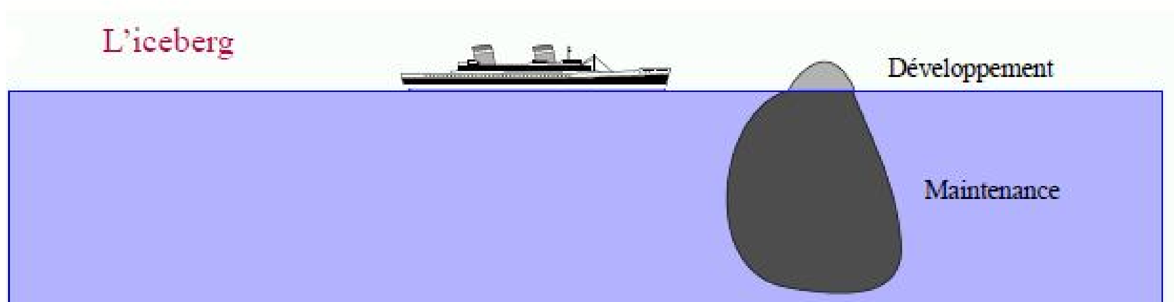


Figure I.2 : L'iceberg [Favre, 95].

De plus, les effets que peut engendrer une opération de modification, ne sont généralement pas limités au seul artefact cible de cette opération. Ils se propagent à d'autres artefacts avec lesquels le constituant modifié est lié par différents types de relations.

Parmi les applications logicielles concernées par la fonction de maintenance, on trouve les bases de données. Les modifications dans les bases de données,

quelles soient des mises à jour, des ajouts ou des suppressions, ont un effet non seulement sur les constituant en relation avec l'artefact modifié au même niveau d'abstraction mais aussi avec les artefacts en relation d'un niveau d'abstraction supérieur. Autrement dit dans les bases de données le modèle d'abstraction supérieur, c'est les modèles de données.

Dans les sections qui suivent nous allons présenter quelques modèles de représentation des données, puis introduire les principes de la discipline qui a pour artefact principal le modèle, à savoir l'ingénierie dirigée par les modèles

I.4. Modèles de données

Les modèles d'information expriment une vue abstraite, dépendante de l'observateur, sur une réalité spécifique et avec un but précis. Chaque modèle conceptuel, logique ou physique peut avoir différents formalismes de représentation. Alors que les modèles conceptuels (tels que E/A) donnent une assez grande liberté d'expression, il est communément admis que les modèles logiques de données (puis physiques) posent un cadre limitant et réduisent l'expressivité du domaine étudié. Très souvent, les problèmes de performance d'accès aux données en sont la cause.

L'histoire des modèles d'information a très vite confronté deux grandes familles de modèles. D'une part les modèles relationnels et d'autre part les graphes conceptuels.

Les modèles relationnels, adaptés à la gestion de grandes masses de données, tentent peu à peu de s'adapter aux nouveaux usages que l'Internet suggère, à savoir l'utilisation d'une masse de données réparties dans un grand graphe global, mais sans grand succès. La structure même de l'Internet, et la pauvreté sémantique du modèle relationnel nous pousse à penser que ce modèle n'a pas les propriétés nécessaires pour aborder les problématiques de demain.

I.4.1 Le modèle relationnel

Le modèle relationnel de données proposé par Ted Codd 1970 [Choquet, 11] pour apporter plus d'indépendance aux éléments de données en opposition aux modèles en réseau et en graphes. Il propose de représenter les données en les structurant dans des tables contenant des colonnes et des tuples ; le nom de la table définissant le domaine de l'information stockée. Les relations entre les tables, effectuée par un mécanisme de clé primaire - clé étrangère, permettent de mettre en relation un ou plusieurs éléments (n-aires). Une relation au sens de T.Codd, ne peut cependant pas représenter de relation sémantique entre deux concepts puisqu'elle n'est pas stockée physiquement dans la base de données. Elle représente par contre un concept mathématique comme T.Codd l'a définit :

Définition : Sachant les ensembles E_1, E_2, . . ., E_n, R est une relation portant sur n ensembles si c'est un ensemble de n-uplet, le premier élément appartenant à E_1, le deuxième à E_2, etc. Un tableau qui représente une relation n-aire R a les propriétés suivantes :

– chaque ligne représente un n-uplet de R,

– chaque ligne est distincte,

– l'ordre des ligne est signifiante, il correspond à l'ordre des ensembles E_1, E_2, …..,E_n des domaines dans lesquels R est défini,

– le sens de chaque colonne est partiellement apporté par le label de celle-ci.

Le modèle logique relationnel basé sur des modèles conceptuels comme l'entité-association.

I.4.2 Le modèle Entité association

Dans le modèle E-A (Entité-Association) ou E-R (*Entity Relationship*), les objets du système d'information sont représentés par des entités, et les liens

par des associations, aussi appelées relations. Nous voyons ici les concepts qui vont nous permettre d'élaborer des schémas conceptuels E-A.

Figure I.3 : schéma du modèle E-A

I.4.2.1 Les entités

Elles correspondent aux objets du système d'information. Par exemple, l'entité CLIENT rassemble toutes les informations communes aux clients de l'entreprise.

Les entités sont nommées et leur noms sont unique dans le modèle et s'écrivent en majuscule.

Dans le système d'information, l'entité CLIENT représentera l'ensemble des clients. Chaque client constitue alors une **occurrence** de l'entité. Une occurrence représente un « exemplaire » de l'objet.

I.4.2.2 Propriétés

Une entité possède toujours au moins une propriété. Chacune de ces propriétés doit pouvoir être valorisée de manière unique. Les propriétés des entités sont issues du dictionnaire des données.

- ***Identifiant***

Parmi les propriétés d'une entité, il y en a une qui joue un rôle particulier :

L'identifiant d'une entité est une propriété telle qu'à chaque valeur de la propriété corresponde une et une seule occurrence de l'entité. Par conséquent :

- on ne peut trouver deux occurrences d'une entité ayant le même identifiant.
- l'identifiant se confond avec l'entité : l'entité existe si on peut valoriser l'identifiant
- si l'entité ne possède qu'une propriété, cette propriété est l'identifiant
- toutes les propriétés sont en dépendance fonctionnelle élémentaire et directe avec l'identifiant
- L'identifiant est représenté souligné dans le modèle.

I .4.2.3 Les associations

Une association est un lien entre des entités. Contrairement aux entités, les associations n'ont pas d'existence propre mais elles peuvent porter des propriétés.

Les associations sont nommées, généralement à l'aide d'un verbe d'action.

Une association peut lier plus de deux entités

Le nombre d'entités impliquées dans l'association détermine sa **dimension** : pour les associations faisant intervenir deux entités, on parle d'association de dimension deux ou associations binaires. Pour les associations faisant intervenir plus de deux entités, on parle d'associations n-aires.

- ***Rôles d'une association***

Chaque « patte » d'une association vers une entité représente le rôle joué par l'association. Ces rôles peuvent être nommés, cela peut aider à clarifier le modèle dans certaines situations complexes.

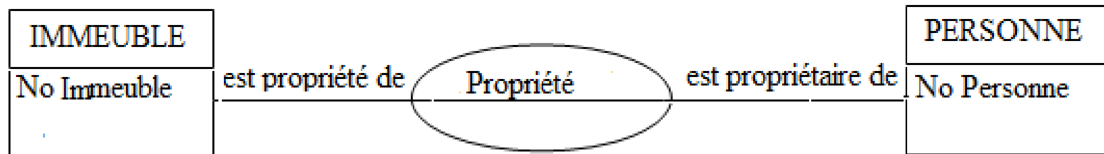


Figure I.4 : Rôles d'une association dans le modèle E-A

Les rôles sont généralement faciles à identifier sur des associations binaires, ce n'est pas le cas des associations n-aires.

I.4.2.4 Cardinalités

La cardinalité représente le nombre d'occurrences minimum et maximum d'une association par rapport à une entité.

La cardinalité minimale représente le nombre de fois « au minimum » où une occurrence de l'association participe aux occurrences de l'entité. Cette cardinalité est choisie parmi 0 ou 1.

La cardinalité maximale représente le nombre de fois « au maximum » où une occurrence de l'association participe aux occurrences de l'entité. Cette cardinalité est choisie parmi 1 ou n où *n* indique une cardinalité maximale supérieure à 1 mais non quantifiée. Si la valeur de *n* est connue, on peut la mentionner.

Les cardinalités sont mentionnées par couple du côté de l'entité à considérer. La cardinalité minimale est représentée en premier, la maximale en second. La cardinalité minimale à 1 ou à 0 dépend des règles de gestion.

I.4.3. Le modèle de classe

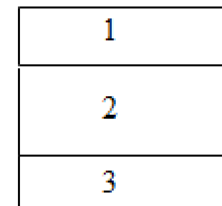
Le modèle de classe est l'un des neuf diagrammes du langage de modélisation unifié (UML), ce modèle représente les entités du système d'information en termes de classes et de relations entre ces classes.

Pour représenter les entités du système d'information, le modèle de classe utilise six concepts : Classe, attribut, identifiant, opération, relation, généralisation / spécialisation.

I.4.3.1 Classe

Une classe est une description abstraite (condensée) d'un ensemble d'objets du domaine de l'application, qui définit une structure, des caractéristiques et des comportements communs de cet ensemble d'objets, ainsi que leurs relations.

Les classes sont représentées par des rectangles compartimentés :



Le 1er représente le nom de la classe,

Le 2ème représente les attributs de la classe,

Le 3ème représente les opérations de la classe

Généralement il y a trois types de classes :

Classes Abstraites, interfaces et les Classes normales.

I.4.3.2 Attribut de classe

Un attribut est une caractéristique intrinsèque partagée par tous les objets d'une classe, il possède un nom unique dans la classe. On peut associer à l'attribut le type des valeurs qu'il peut prendre ainsi qu'une valeur initiale**.** L'identifiant est un attribut particulier, qui permet de repérer de façon unique chaque objet, instance de la classe.

UML définit 3 niveaux de visibilité pour les attributs :

1- public (+) : l'élément est visible pour tous les clients de la classe.
2- protégé (#) : l'élément est visible pour les sous-classes de la classe.
3- privé (-) : l'élément n'est visible que par les objets de la classe dans laquelle il est déclaré.

I.4.3.3 Opération d’une classe

Une opération est une fonctionnalité assurée par une classe pour modifier des attributs et produire un résultat. La description des opérations peut préciser les paramètres d’entrées et de sorties ainsi que leurs types et cardinalités. On peut associer à l’opération son type de retour.

La liste des paramètres et le type de retour d’une opération est appelé signature d’une opération.

Une méthode est une implémentation (définition) d’une opération. Elle définit comment l’opération se comporte, pas seulement ce qu’elle fait. Une opération est dite abstraite si la méthode n’est pas spécifiée.

Les niveaux de visibilité définit par UML pour une opération sont les même que ceux définis pour un attribut.

I.4.3.4 Relations entre classe

S’il existe des liens entre objets, cela se traduit nécessairement par des relations qui existent entre leurs classes respectives. Les liens entre les objets doivent être considérés comme des instances de relations entre classes. Il existe plusieurs types de relations entre classes: l’association, la généralisation/spécialisation, la dépendance.

- *L’association*

L’association est la relation la plus courante et la plus riche du point de vue sémantique. Une association est une relation statique n-aire (le plus souvent : elle est binaire) : c’est-à-dire qu’elle relie plusieurs classes entre elles. L’association existe entre les classes et non entre les instances : elle est introduite pour montrer une structure et non pour montrer des échanges de données. L'association est instanciable dans un diagramme d'objets ou de collaboration, sous forme de liens entre objets issus de classes associées.

- La généralisation / spécialisation

Le principe de généralisation / spécialisation permet d'identifier parmi les objets d'une classe (générique) des sous-ensembles d'objets (des classes spécialisées) ayant des définitions spécifiques. La classe plus spécifique (appelée aussi classe fille, classe dérivée, classe spécialisée, classe descendante…) est cohérente avec la classe plus générale (appelée aussi classe mère, classe générale …), c'est-à-dire qu'elle contient par *héritage* tous les attributs, les membres, les relations de la classe générale, et peut contenir d'autres. Les relations de généralisation peuvent être découvertes de 2 manières:

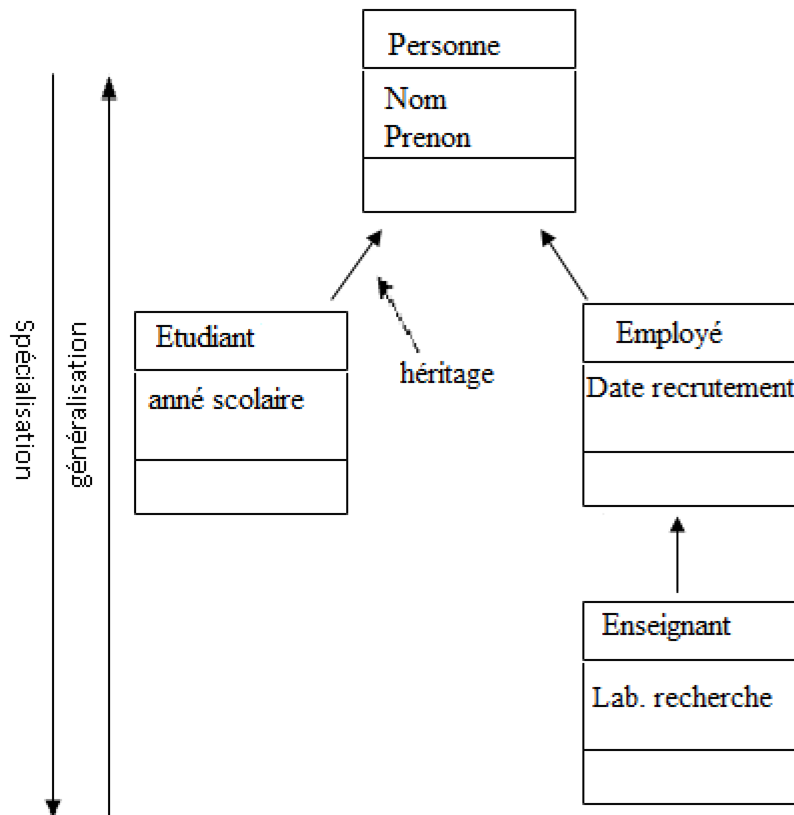


Figure I.5 : Relation de généralisation/spécialisation

La généralisation : il s'agit de prendre des classes existantes déjà mises en évidences) et de créer de nouvelles classes qui regroupent leurs parties communes ; il faut aller du plus spécifique au plus général.

La spécialisation : il s'agit de sélectionner des classes existantes (déjà identifiées) et d'en dériver des nouvelles classes plus spécialisées, en spécifiant simplement les différences. Ces deux démarches, même si elles sont fondamentalement différentes, mènent au même résultat, à savoir la constitution d'une hiérarchie de classes reliées par des relations de généralisation. La relation de généralisation est très puissante car elle permet de construire simplement de nouvelles classes à partir de classes existantes. Cependant, elle est très contraignante dans le sens où elle définit une relation très forte entre les classes. Ainsi, l'évolution d'une classe de base entraîne une évolution de toutes les classes qui en dérivent. Cet effet boule de neige peut avoir des conséquences aussi bien positives (quand c'est l'effet recherché) que négatives.

- La dépendance

Les relations de dépendance sont utilisées lorsqu'il existe une relation sémantique entre plusieurs éléments qui n'est pas de nature structurelle. Une relation de dépendance définit une relation unidirectionnelle entre un élément source et un élément cible. Une dépendance est une relation entre deux éléments de modélisation dans laquelle toute modification effectuée sur un élément de modélisation (l'élément influent) affecte l'autre élément (élément dépendant). UML définit 4 types de relation de dépendance. Pour chaque type de dépendance, un mot clé ou stéréotype entre guillemets peut être ajouté à la relation de dépendance pour préciser sa nature. Les 4 types de relation sont l'abstraction, liaison, permission et utilisation.

I.4.4. Modèle XML

Le modèle de données des bases de données XML natives "BDXMLN" est relativement éloigné d'un modèle relationnel "classique" ou d'un modèle conceptuel (E/A). Un document XML ayant une structure d'arborescence. Lors de son stockage dans une base de données XML native, un document XML est classé dans une collection. Une collection regroupe un jeu de documents XML. Une base de données peut contenir plusieurs collections. Ces collections sont organisées sous forme d'arborescence. Cette organisation est similaire à un système de fichiers : les collections correspondent aux dossiers, et les documents XML correspondent aux fichiers.

I.4.4.1 Bases de données XML natives :

Une base de données doit répondre à plusieurs critères pour prétendre être de type XML natif [Staken ,01]: Elle doit définir un modèle logique pour tout document XML et se baser sur ce modèle pour stocker et extraire des documents. Le modèle doit au minimum inclure les éléments, les attributs, les PCDATA (types d'éléments textuels devront être traités par l'analyseur), les CDATA (types d'éléments textuels qui ne doivent pas être traités par l'analyseur) et l'ordre interne du document. De plus, elle doit considérer le document comme unité fondamentale de stockage.

Cela change radicalement des BD relationnelles, pour qui une unité de stockage correspond à une ligne d'enregistrement d'une table donnée. Enfin, l'unité de stockage physique n'est pas spécifique, elle peut être bâtie sur les bases des données traditionnelles (relationnelle ou orienté objet) ou bien utilisée d'autres techniques de stockage. Peu importe, tant que ce dernier n'influence pas l'intégrité des données traitées.

Les bases de données XML natives sont des bases conçues spécialement pour stocker des documents XML. Comme toutes les autres bases, elles possèdent des fonctionnalités telles que les transactions, la sécurité, les accès multi utilisateurs, un ensemble d'API(s), des langages de requêtes comme XQuery (standard édicté par W3C (*World Wide Web Consortium*).). La seule différence par rapport aux autres bases c'est qu'elles sont basées sur XML [Bourett, 03].

La figure I.6 montre un document XML et sa représentation hiérarchique, un véritable système de base de données XML native, utilise la structure d'arbre comme étant le modèle de base pour le stockage et le traitement.

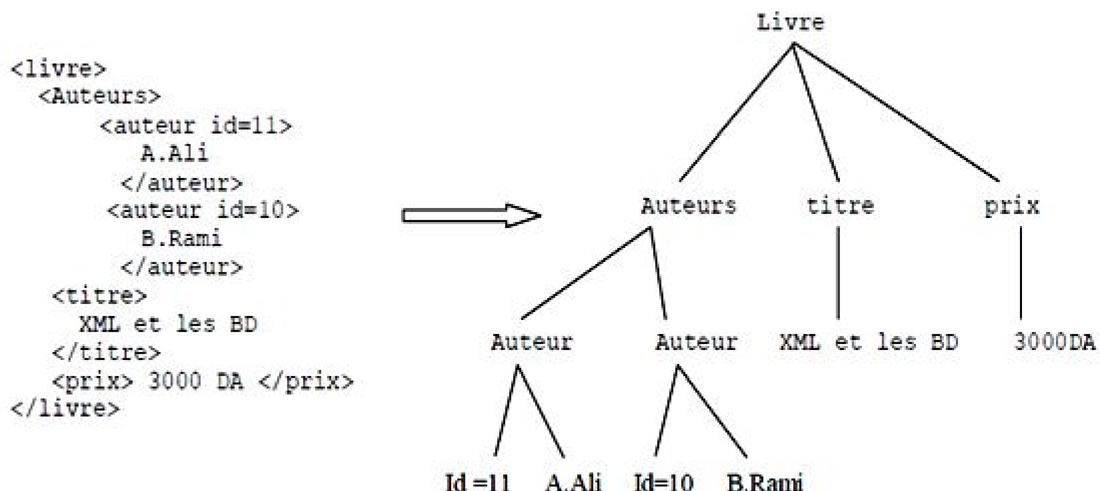


Figure I.6: Document XML et sa structure d'arborescence

Les bases de données XML natives sont plus utiles pour le stockage des contenus orientés document, car elles préservent l'ordre interne du document, les instructions de traitement, les commentaires, les sections CDATA. Etc. en outre les bases XML natives supportent les langages de requêtes XML qui permettent d'effectuer des recherches difficiles à effectuer dans un langage tel que SQL.

La base de données XML native est utilisée également pour le stockage des contenus orientés données et le stockage de documents qui ne possèdent pas de DTD(s) (*Document Type Definition*) ou tout autre type de schéma XML (les documents sans schémas). Cela veut dire qu'une base de données XML native peut stocker n'importe quel document XML sans configuration préalable.

Dans une base de données XML, il existe trois types de langage de requêtes : les langages de requêtes basés sur des modèles, les langages de requêtes basés sur SQL et les langages de requêtes XML.

I.4.4.2 Classification des bases de données XML natives

Il existe deux grandes catégories d'architectures des bases de données XML natives, les architectures basées sur le texte et celles qui sont basées sur un modèle :

- **Bases de données XML natives basées sur le texte**

Une base de données XML native basée sur le texte stocke le document XML en tant que texte. Cela peut être un fichier dans un système de fichiers, un BLOB (Binary Large Objec) dans une base relationnelle, ou un format propriétaire.

Les index sont communs à toutes les bases de données XML natives basées sur le texte. Ils permettent au moteur de recherche de naviguer facilement en tout point d'un document XML quelconque. Cela procure à ce genre de base un avantage considérable en matière de vitesse quand on recherche des documents entiers ou des fragments de documents.

Ce type de base de données XML native présente l'inconvénient de repasser à chaque fois le document.

➢ Bases de données XML natives basées sur le modèle

La seconde catégorie est constituée des bases XML natives basées sur un modèle. Plutôt que de stocker un document XML en tant que texte, elles construisent un modèle objet interne du document et stockent ce modèle. La manière dont le modèle est stocké dépend de la base. Certains produits stockent le modèle dans une base relationnelle ou orientée objet.

D'autres bases utilisent un format de stockage propriétaire adapté à leur modèle. Les bases XML natives basées sur un modèle et construites sur d'autres bases possèdent vraisemblablement des performances similaires à ces bases sous-jacentes lors de la recherche des documents, et c'est pour cette raison évidente qu'elles reposent sur ces systèmes pour retrouver les données.

Cependant, la conception de la base (tout particulièrement dans le cas des bases XML natives construites sur des bases relationnelles) laisse place à des variations.

Lorsque l'on recherche les données dans l'ordre où elles sont stockées, les bases XML natives basées sur un modèle et qui utilisent un format de stockage propriétaire possèdent vraisemblablement des performances similaires aux bases XML natives basées sur le texte.

Ceci est dû au fait que la plupart de ces bases utilisent des pointeurs physiques entre les nœuds, ce qui devrait fournir des performances similaires à la recherche textuelle. (La vitesse dépend aussi du format de sortie. Les systèmes basés sur le texte sont manifestement plus rapides pour renvoyer des documents textuels, tandis que les systèmes basés sur un modèle sont indubitablement plus rapides pour renvoyer des arbres par l'analyseur DOM (*Document Object Model*), en supposant que leur modèle coïncide explicitement au DOM.)

À l’instar des bases XML natives basées sur le texte, les bases natives basées sur un modèle rencontrent probablement des problèmes de performance lorsque l’on recherche et retourne des données dans un format quelconque autre que celui sous lequel ces données sont stockées. Cependant, elle présente l’avantage de combiner de fragments de données à partir de différents documents.

I.5. Ingénierie dirigée par les modèles

Suite à l’approche objet des années 80 et de son principe du « tout est objet », l’ingénierie du logiciel s’oriente aujourd’hui vers l’ingénierie dirigée par les modèles (IDM) et le principe du « tout est modèle ». [COMBEMALE, 08].

La modélisation constitue la première étape dans la conception ou la configuration d’outil documentaire ou, dans un cadre plus général, d’un système d’information. Ce premier pas est d’une importance cruciale. Le modèle réalisé permet en effet d’identifier d’une manière claire, précise et exhaustive les besoins que l’outil viendra satisfaire.

I.5.1 Principe de l’IDM

L’ingénierie dirigée par les modèles (IDM), ou Model Driven Engineering (MDE) en anglais, a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique. L'IDM a pour principal objectif de relever un certain nombre de défis du génie logiciel (qualité, productivité, séparation des préoccupations, coût, etc.) en suivant une approche à base de modèles [Bézivin , 04]. Il s’agit d’une forme d’ingénierie générative dans laquelle tout ou partie d’une application est engendrée à partir de modèles.

Dans cette nouvelle perspective, les modèles occupent une place de premier plan parmi les artéfacts de développement des systèmes, mais doivent en

contrepartie être suffisamment précis et riches afin de pouvoir être interprétés ou transformés par des machines. Le processus de développement des systèmes peut alors être vu comme une séquence de transformations de modèles partiellement ordonnée [Diaw et al. 08], chaque transformation prenant un ou des modèles en entrée et produisant un ou des modèles en sortie, jusqu'à l'obtention d'artéfacts exécutables.

Alors que l'approche objet est fondée sur deux relations essentielles, « InstanceDe » qui permet d'introduire la notion de classe et « HériteDe » permet d'introduire la notion de superclasse, l'IDM est basée sur un autre jeu de concepts et de relations. La première relation, appelée *ReprésentationDe* est liée à la notion de *modèle*, alors que la relation *EstConformeA* permet de définir la notion de modèle par rapport à celle de *métamodèle*. [Lajmi, 10]

Le concept central de l'IDM est la notion de modèle pour laquelle il n'existe pas à ce jour de définition universelle. Néanmoins, de nombreux travaux s'accordent à un relatif consensus d'une certaine compréhension.

I.5.2 Modèle, métamodèle et méta métamodèle

Trois concepts fondamentaux sont à la base de l'Ingénierie Dirigée par les Modèles : les modèles, les métamodèles, le méta métamodèle. La figure I.7 schématise les relations entre ces concepts dans une architecture à trois niveaux d'abstraction.

I.5.2.1 Le Modèle (M_1):

Il n'existe pas de définition universelle pour le concept de modèle, cependant on peut trouver un consensus relatif autour du fait qu'un *modèle* est une représentation simplifiée (ou abstraction) d'un système réel. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé.

On déduit de cette définition la première relation majeure de l'IDM, entre le modèle et le système qu'il représente, « représentationDe », nommée dans [Bézivin, 04] par μ et représentée sur la figure I.7.

I.5.2.2. Le Méta modèle (M_2):

Un métamodèle est un modèle qui définit le langage d'expression d'un modèle, c.-à-d. le langage de modélisation. Dans un métamodèle, on définit les concepts ainsi que les relations entre les concepts permettant d'exprimer des modèles [Gerbé et al., 01]. Un modèle est instance, d'un métamodèle.

La notion de métamodèle conduit à l'identification de la seconde relation, liant le modèle au métamodèle, appelée « conformeA », et nommée dans [Bézivin, 04] par χ et représentée sur la figure I.7

I.5.2.3. Le Méta métamodèle (M_3):

Les éléments du langage d'expression d'un métamodèle doivent aussi satisfaire un ensemble de règles et contraintes. Ces règles et contraintes sont exprimées sous forme d'un méta langage ou méta métamodèle.

Le méta-métamodèle définit la structure de tous les métamodèles qui se trouvent au niveau M2.

Le langage utilisé en M_3 est réflexif, c'est-à-dire qu'il s'auto-décrit, ce qui permet de dire que le niveau M_3 est le dernier niveau de la hiérarchie (conforme à lui-même) figure I.7.

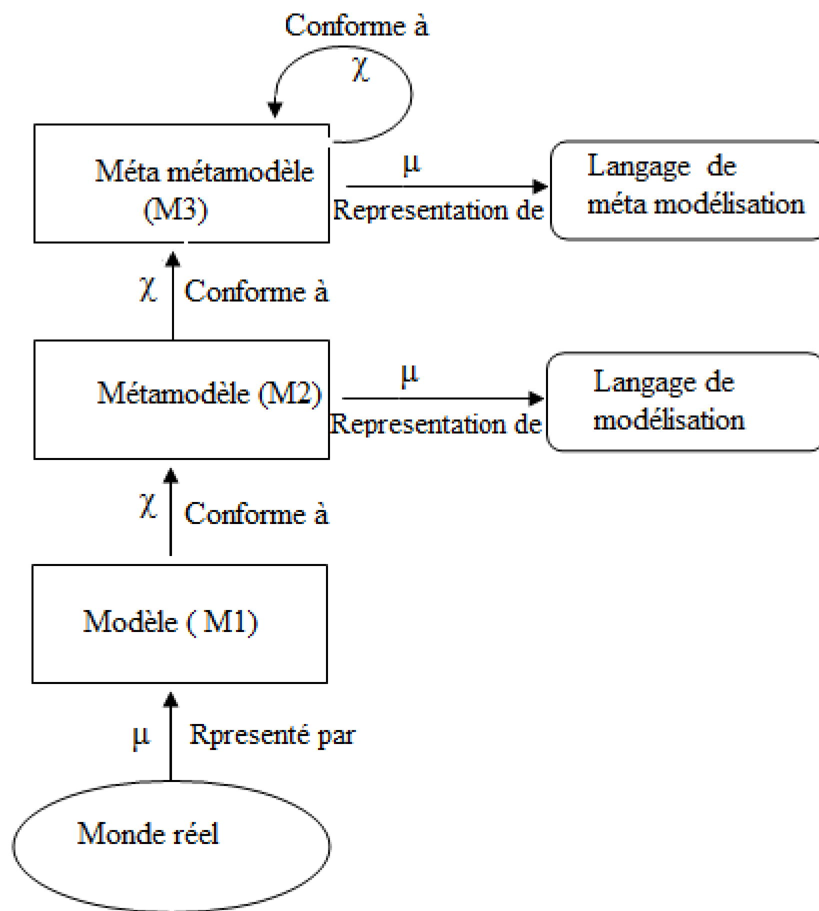


Figure I.7 : Relations entre système, modèle, métamodèle et langage [Diaw et al. 08]

I.5.3 Architecture dirigée par les modèles

L'ADM (Architecture dirigée par les modèles) ou le MDA (Model-Driven Architecture) en anglais est une initiative de l'OMG (Object Management Group) rendue publique en 2000. C'est une proposition à la fois d'une architecture et d'une démarche de développement. L'idée de base du MDA est la séparation des spécifications fonctionnelles d'un système des détails de son implémentation sur une plate-forme donnée.

Pour cela, le MDA classe les modèles en modèles indépendants des plates-formes appelés PIM (Platform-Independent Models) et en modèles spécifiques appelés PSM (Platform-Specific Models). [Combemale, 08]

L'approche MDA permet de déployer un même modèle de type PIM sur plusieurs plates-formes (modèles PSM) grâce à des projections standardisées. Elle permet aux applications d'inter opérer en reliant leurs modèles et favorise l'adaptabilité aux évolutions des plates-formes et des techniques. La mise en œuvre du MDA est entièrement basée sur les modèles et leurs transformations.

L'initiative MDA a donné lieu à une standardisation des approches pour la modélisation sous la forme d'une structure en 4 niveaux de modélisation (appelée communément *Pile de modélisation*) présentée dans la figure I.8. La proposition initiale était d'utiliser le langage UML et ses différentes vues comme unique langage de modélisation. Cependant, il a fallu rapidement ajouter la possibilité d'étendre le langage UML, par exemple en créant des profils, afin d'exprimer de nouveaux concepts relatifs à des domaines d'application spécifiques.

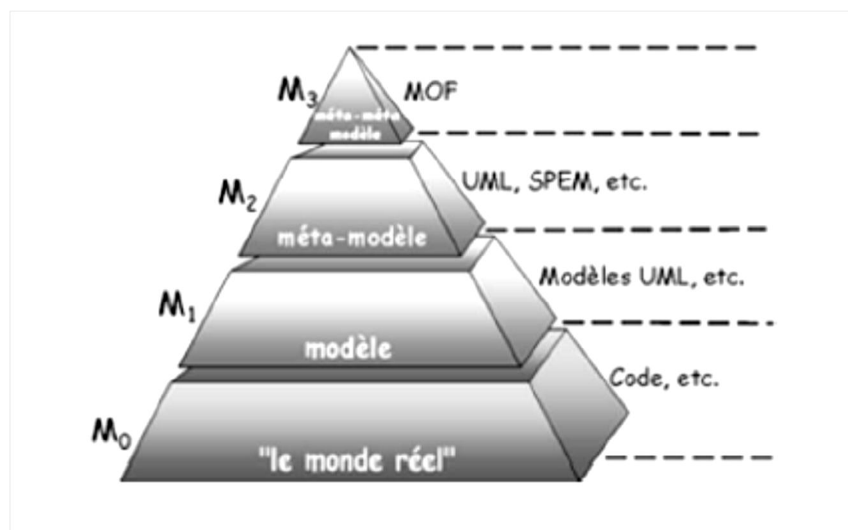


Figure I.8 : *Pile de modélisation* [Diaw et al. 08]

Ces extensions devenant de plus en plus importantes, la communauté MDA a élargi son point de vue en considérant les langages de modélisation spécifiques à un domaine (DSML en Anglais). La figure I.9 donne une vue générale d'un processus MDA en faisant apparaître les différents niveaux d'abstraction associés aux modèles, depuis les modèles de besoins jusqu'au code qui s'exécute.

I.5.3.1 niveaux d'abstraction du MDA

- *CIM (Computation Independent Model)* : appelé aussi modèle de domaine ou modèle métier. Il modélise les exigences du système. Il montre le système dans l'environnement organisationnel dans lequel il va être exécuté. Son but est d'aider à la compréhension du problème. Les exigences exprimées dans le CIM doivent être traçables dans le PIM et le PSM.

- *PIM (Platform Independent Model) :* décrit le système sans montrer les détails de son utilisation sur une plate-forme particulière. Un PIM peut être construit en se basant sur les points de vue entreprise. Les PIMs représentent par exemple les différentes entités fonctionnelles d'un système avec leurs interactions. Ils représentent l'intérêt de l'application : la logique métier. Un PIM doit être raffiné par les détails d'une ou plusieurs architecture(s) particulière(s) pour obtenir un PSM.

- *PDM (Platform Description Model) :* Après la définition du PIM, l'architecte doit choisir une plateforme technique qui permet d'implanter le système avec la qualité architecturale décrite dans le PIM. Cette plateforme doit être décrite dans un modèle afin de

permettre son intégration dans la construction du logiciel. Ce modèle est appelé PDM.

- *PSM (Platform Specific Model)* : est un modèle dépendant d'une plateforme technologique. C'est le modèle produit par la transformation du PIM. Il spécifie comment le système va utiliser la plate-forme choisie. Les PSM servent essentiellement de base à la génération de code exécutable vers ces mêmes plateformes techniques.

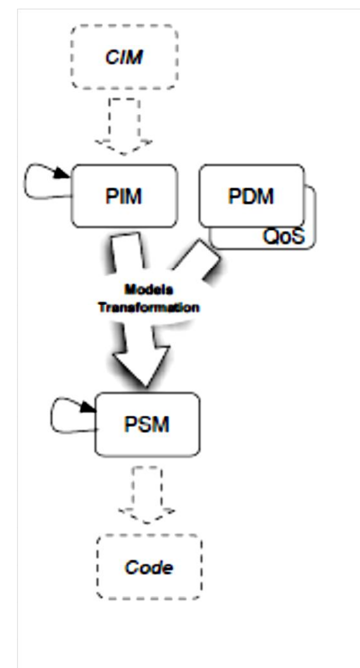


Figure I.9. *Principe du Processus MDA*

I.6. Conclusion

Apres avoir rappelé ce qu'est un système complexe et défini les différentes catégories de la maintenance des applications informatiques, nous avons présenté les modèles de représentations des données les plus utilisés, suivi de la discipline qui prend le modèle comme artefact principal de son architecture à savoir l'IDM.

Dans le chapitre qui suit nous allons détailler l'opération de base sur laquelle repose l'Architecture Dirigée par les Modèles (ADM) qui est la transformation des modèles.

Chapitre II

Transformation des modèles

II.1. Introduction :

Comme nous l'avons vu précédemment la transformation des modèles est un aspect clé de l'ingénierie dirigée par les modèles. Cette transformation peut avoir des sémantiques différentes en fonction des utilisations : raffinage, restructuration, réingénierie ...etc.

L'opération de transformation manuelle ou automatique n'est pas vraiment nouvelle pour le développeur d'application informatique. En effet, quand la programmation n'est autre qu'une transformation manuelle d'un schéma de conception en un langage de programmation de haut niveau (langage C), la compilation de ce langage (génération du code en langage machine), est une transformation automatique d'un langage de haut niveau à un langage de bas niveau.

Dans ce chapitre nous présenterons quelques définitions de la notion de transformation se trouvant dans la littérature ainsi que celle émanant de notre vision, puis nous verrons la taxonomie de ces transformation, les différentes approches utilisées pour réaliser ces transformations et au final un panorama des langages et outils manipulés pour une transformation.

II.2 Definitions de Transformation

Fabian Büttner définit une transformation comme " A transformation T : $M_I \longrightarrow M_F$ from a source metamodel M_I to a target metamodel M_F , a transformation model M_T is a metamodel that includes M_I and M_F , and additional structural modeling elements and constraints in order to capture the execution semantics of T ". [Büttner, 12].

Pour Büttner une transformation T : $M_I \longrightarrow M_F$ doit être conforme au méta modèle de transformation M_T . ce méta modèle inclut le modèle source M_I le modèle cible M_F , les élément de structure des modèles et un ensemble de contraintes dans le but d'exécuter la transformation voulu.

Dolques reprend la définition de Anneke Kleppe dans [Dolques ; 09] , qui définit les transformations de modèles comme:

– «Une *transformation* est la génération automatique d'un modèle cible à partir d'un modèle source, suivant une définition de la transformation » ;

– «Une *définition de transformation* est un ensemble de règles de transformation qui, réunies, décrivent comment un modèle dans un langage source peut être transformé dans un langage cible » ;

– « Une *règle de transformation* est une description de la manière dont une ou plusieurs structures du langage source peuvent être transformées en une ou plusieurs structures du langage cible ».

Dans le contexte de MDA Kherraf dans [Kherraf, 11] donne la définition suivante, "*the process of converting a model into another model of the same system*"

On définit une transformation de modèle comme une opération qui a un nom : M_SM_C (M_S 2 M_C), ou M_ST (M_S 2 T) avec M_S, M_C des modèles candidats typés par un métamodèle, et T par un format texte. Une transformation se caractérise par sa direction de transformation qui précise

le modèle source et cible et par ses règles de transformation qui décrivent la correspondance entre les entités du modèle source et celles du modèle cible.
Une transformation des entités du modèle source met en jeu deux étapes. La première étape permet d'identifier les correspondances entre les concepts des modèles source et cible au niveau de leurs métamodèles, ce qui induit l'existence *d'une fonction de transformation* applicable à toutes les instances du métamodèle source. La seconde étape consiste à appliquer la transformation du modèle source afin de générer automatiquement le modèle cible par un programme appelé *moteur de transformation* ou d'exécution. La figure II.1 illustre ces deux étapes d'une transformation de modèles.

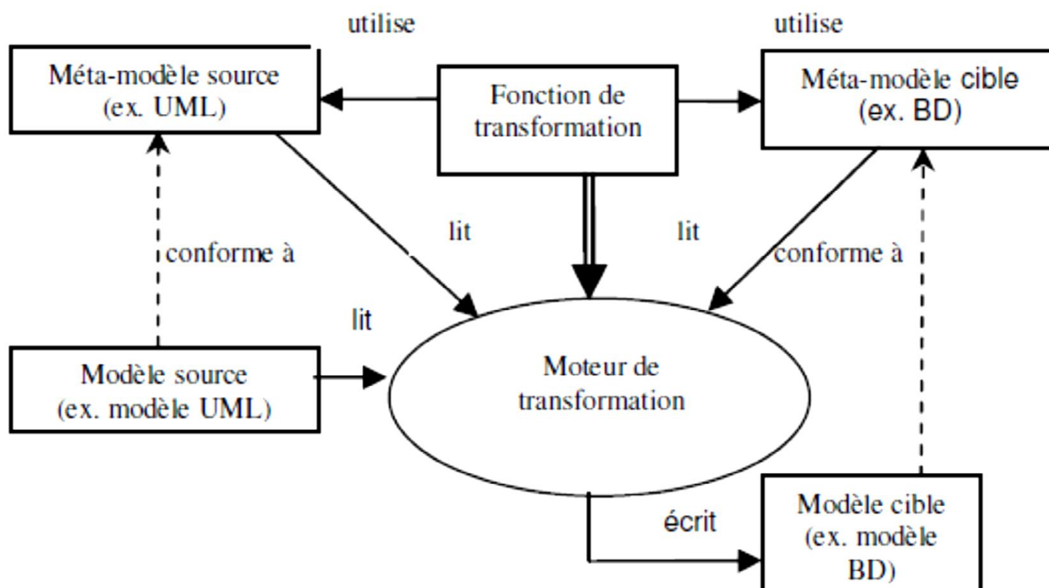


Figure II.1 : *Schéma de base d'une transformation de modèles* [Diaw et al. 08]

II.3. Taxonomie des transformations

Comme nous l'avons vu, les modèles candidats à une opération de transformation sont typés par des métamodèles. Dans le cas où le métamodèle de M_S (MM_S) est le même que celui de M_C (MM_C) on parle alors de transformation endogène, sinon c'est une transformation exogène.

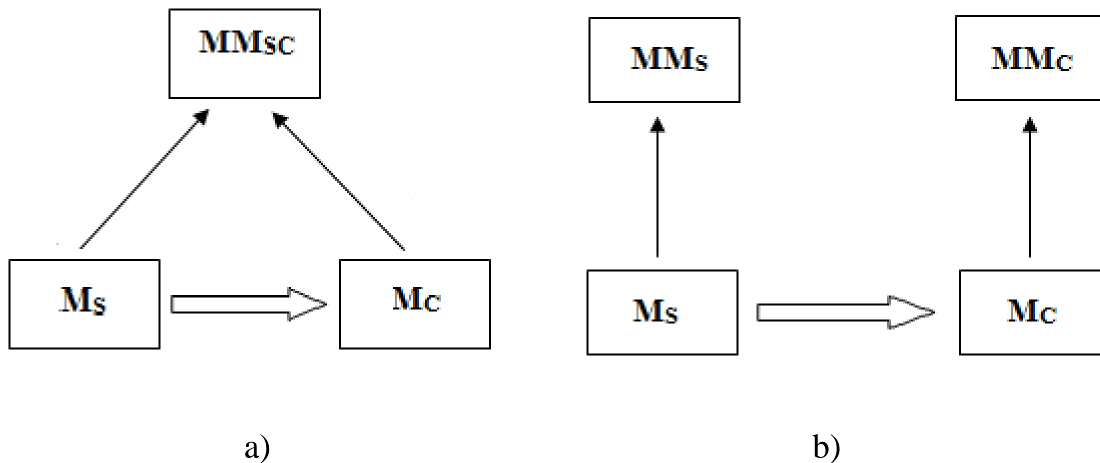


Figure II.2: a) Transformation endogène. b) Transformation exogène

Par ailleurs, le niveau d'abstraction, critère important pour l'approche MDA, permet de classifier les transformations de modèles. Une transformation entre deux modèles du même niveau d'abstraction est dite horizontale. Par exemple le passage d'un modèle indépendant de toute plateforme (PIM) à un autre (PIM) ou bien d'un modèle spécifique à une plateforme (PSM) à un autre (PSM) est dit transformation horizontale. La transformation entre deux modèles de deux niveaux d'abstraction différents est appelée transformation verticale, comme le passage d'un (PIM) à un (PSM) ou inversement.
Une transformation permettant de réduire la différence des niveaux d'abstraction est appelée transformation d'ingénierie directe. Une transformation augmentant le niveau d'abstraction est appelée transformation d'ingénierie inverse. La figure qui suit résume cette classification.

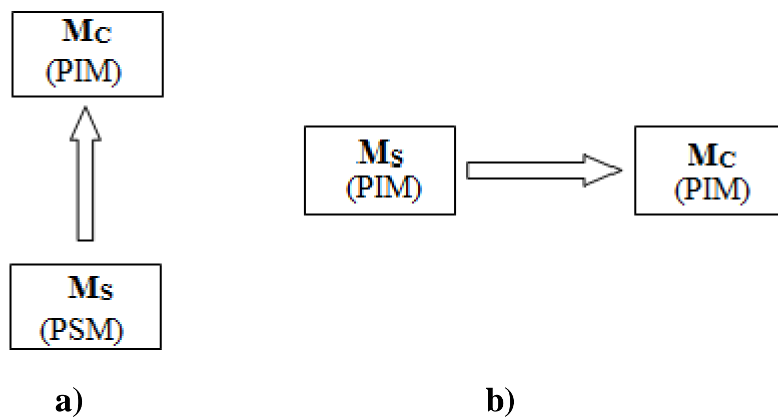


Figure II.3 : a) Transformation verticale ; b) Transformation horizontale

II.4. Types de transformation de modèles

On distingue dans ce qui suit, 3 axes de classification des types de transformation des modèles :

II.4.1. Les transformations selon le modèle source et cible : on trouve deux types de transformation dans cet axe.

Les transformations « Model to Text » qui ont pour but principal de générer du code à partir d'un modèle (transformer un modèle d'un certain niveau d'abstraction vers un niveau d'abstraction moins élevé) appelé aussi transformation verticale.

Les transformations « model to model ». Dans ce type de transformation on génère un nouveau modèle définit par un métamodèle à partir du modèle source afin d'étudier les différentes vues du système, son optimisation, la vérification de ses propriétés et sa validation.

II.4.2. Les transformations selon le nombre d'éléments transformés

On distingue les types de transformation suivants :

– Une transformation simple (1 vers 1) qui associe à tout élément du modèle source au plus un élément du modèle cible.

– Une transformation multiple (M vers N) prend en entrée un ensemble d'éléments du modèle source et produit un ensemble d'éléments du modèle cible. Les transformations de décomposition de modèles (1 vers N) et de fusion de modèles (N vers 1) sont des cas particuliers de transformations multiples.

– Une transformation de mise à jour, encore appelée transformation sur place, consiste à modifier un modèle par ajout, modification ou suppression d'une partie de ses éléments. Dans ce type de transformation, les modèles source et cible sont confondus. Une telle transformation agit directement sur le modèle source sans créer de modèle cible. Un exemple typique d'une telle transformation est la restructuration de modèles (Model Refactoring) qui est l'objet du chapitre suivant**.**

II.4.3. Les transformation selon la structure du modèle

Selon l'approche utilisée, un modèle peut être considéré comme un arbre ou un graphe :

- *Transformation d'arbre :* Dans ce type les modèles sources sont conformes au métamodèle XML. Un modèle XML est structuré de manière arborescente. Dans ce cas la transformation d'arbre peut être en bloc, disposant de la totalité de l'arbre source et des règles de transformation, ou en streaming où la transformation est réalisée au fur et à mesure sans connaitre la totalité du document source.

- *Transformation de graphe :* Le principe de transformation des graphes dépend du graphe considéré. Si l'on considère les multi-graphes (qui admettent plusieurs arcs parallèles entre deux nœuds) ou les graphes

comme des structures relationnels (il existe au plus un arc entre deux nœuds), alors les traitements de transformation se font par des approches algébriques.

II.5. Approches de transformation

Six catégories d'approches de transformations de modèles sont retenues: approches par programmation, approches par template, approches par structure, approches relationnelle, approches par transformations de graphes et les approches par modélisation.

II.5.1. Approches par programmation

Dans ces approches, la transformation est décrite sous forme d'un programme informatique à l'image de n'importe quelle application informatique. Pour écrire un programme de transformation, on utilise:

- *Les langages de programmation dit standard* et plus particulièrement les langages orientés objet comme java. Cette approche reste très utilisée car elle réutilise l'expérience accumulée et l'outillage des langages existants.

-*Les langages dédiés à un atelier de génie logiciel* comme *J* dans *objecteering*. Ces langages sont propriétaires et inutilisables en dehors d'un atelier de génie logiciel.

II.5.2. Approches par templates

Les approches basées sur des *templates* sont utilisées dans la génération du code source à partir des modèles. La transformation du PSM en code source représente un bon exemple de cette approche. Les modèles contiennent un méta programme qui peut accéder au code source. Généralement, les langages basés sur des *templates* incorporent le patron de conception *visitor* permettant ainsi de traverser la structure interne du modèle [Biehl et al. , 10].

II.5.3. Approches par structure

Ces approches consistent à définir des canevas des modèles cibles souhaités. Ces canevas sont des *modèles cibles paramétrés*. L'exécution d'une transformation consiste à prendre un modèle cible paramétré et à remplacer ses paramètres par les valeurs d'un modèle source.

II.5.4. Approches relationnelle

Ces approches utilisent les contraintes pour spécifier les relations entre les éléments du modèle source et ceux du modèle cible en utilisant une logique déclarative basée sur des relations mathématiques.

II.5.5 Approches par transformation de graphes

Ces approches conviennent lorsque les modèles sont représentés par des graphes. Elles expriment les transformations sous une forme déclarative. Les règles de transformation sont définies sur des parties de modèle et non pas sur des éléments basiques. Une opération de filtrage de motifs (*Pattern Matching*) est ensuite lancée. Le moteur de transformation compare à chaque fois des fragments du modèle source pour trouver des règles applicables. Ce fragment est ensuite remplacé par son équivalent dans la règle appliquée. Quelques exemples d'approches basées sur les transformations de graphes : VIATRA, ATOM3, GreAT, UMLX, BOTL, etc.

II.5.6. Approches par modélisation

Elles consistent quant à elles à appliquer les concepts de l'ingénierie des modèles aux transformations des modèles elles-mêmes. L'objectif est de modéliser les transformations de modèles et de rendre les modèles de transformation pérennes et productifs, en exprimant leur indépendance vis-à-vis des plates-formes d'exécution. Le standard MOF 2.0 QVT (Queries, Views, Transformations) de l'OMG a été élaboré dans ce cadre et a pour but de définir un métamodèle permettant l'élaboration des modèles de transformation de modèles.

Les langages utilisés dans les modèles de transformation sont des langages dédiés à la transformation des modèles et peuvent être déclaratifs, impératifs ou hybride.

- *Les langages impératifs* spécifient un flux de contrôle séquentiel. Ils fournissent des moyens capables de décrire comment le langage de transformation devrait s'exécuter. Les instructions et les concepts de ce langage sont similaires aux langages de programmation orienté objet comme le java ou le C++. Ils offrent un haut niveau de contrôle, ce qui donne de la flexibilité aux programmeurs et permet une implémentation efficace.

  Les transformations à l'aide de ces langages sont décrites comme une séquence d'actions, ce qui est bien utile dans le cas où l'ordre des règles de transformation devrait être contrôlé explicitement [Bex et al., 02]**.** En effet, l'approche impérative est plus appropriée grâce aux notions de séquence, de sélection et d'itération. On peut citer le langage Kermeta.
- *Les langages déclaratifs* ne garantissent pas un contrôle explicite. Au lieu de décrire comment la transformation doit être exécutée, l'accent est mis sur ce qui doit être transformé par l'opération de transformation. Cette approche décrit la relation entre le modèle source et le modèle cible et peut être interprétée d'une manière bidirectionnelle. Les langages déclaratifs sont compacts et les descriptions des transformations sont courtes et concises. Le langage Tefkat est un exemple de langage déclaratif.
- *Langages hybrides :* Les principes utilisés dans les deux types de langages précédemment vus peuvent donc coexister au niveau des composants de transformation. Une transformation impérative peut utiliser des sous-transformations déclaratives et dans l'autre sens une transformation impérative peut être encapsulée pour fournir une interface déclarative. Dans ce type de langage on peut citer le langage ATL (ATLAS Transformation Language).

II.6. Langages de transformation de modèles

Chaque approche de transformation de modèle requiert un langage de transformation. Dans cette perspective, K. Czarnecki a proposé une série de critères qui doivent caractériser un langage de transformation [Czarnecki, 03]. Le langage doit :

- être exécutable;
- avoir une implémentation efficace.
- être expressif et non ambiguë.
- fournir une description précise, concise et claire des règles de transformations.
- différencier les règles de sélection des éléments du modèle source et les règles de transformation.
- proposer des constructions graphiques (une notation visuelle est plus concise et intuitive qu'une notation textuelle).
- proposer la possibilité de faire des transformations composites à l'aide des opérateurs de séquences, de conditions et de répétions.
- offrir la possibilité de faire des transformations bidirectionnelles et une mise à jour incrémentale des transformations (possibilité de reporter les changements effectués sur un modèle source dans le modèle cible).

Le QVT (Queries, Views, Transformations) et l'ATL (Atlas Transformation Language) sont les langages les plus utilisés à ce jour.

II.6.1. QVT (Queries, Views, Transformations)

Le QVT est un langage de transformation de modèles standardisé par l'OMG (Object Management Group, 2011). Il permet la transformation de modèle à modèle et du modèle vers du texte en utilisant particulièrement le métamodèle d'UML. Donc, il est possible de générer du code source ou même de la documentation grâce à ce langage.

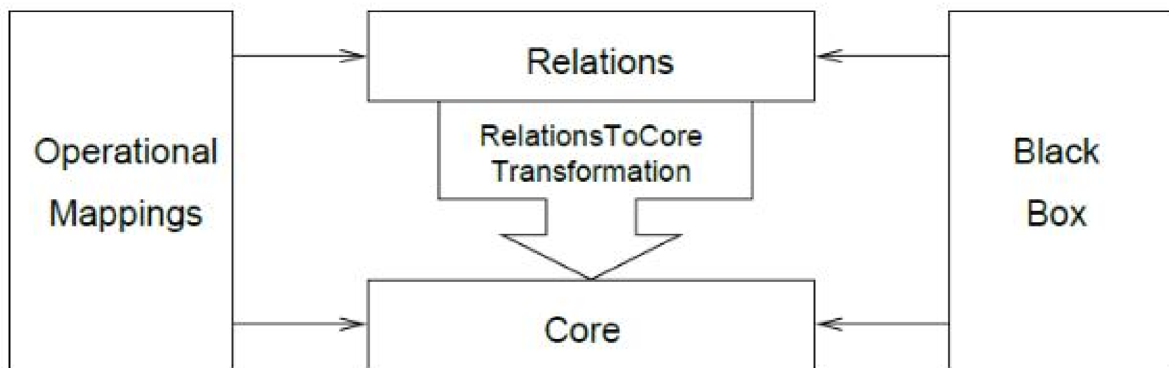


Figure II.4 : Composantes du langage QVT

Le langage QVT, comme le montre la Figure II.4, est constitué de quatre composantes. Trois de celles-ci sont des spécifications de langage dédié. Ces spécifications se nomment *Core*, *Relation* et *Operational Mapping*.

QVT relational est un langage déclaratif de haut niveau permettant de faire des transformations de modèle à modèle. Les transformations sont définies sous forme de règles qui, après la transformation, doivent être respectées totalement par le modèle cible.
Une opération de transformation est une spécification d'un ensemble de relations entre le métamodèle source et le métamodèle cible, permettant ainsi de vérifier la correspondance, de faire respecter la cohérence et de synchroniser d'autres modèles candidats à cette même opération de transformation. Le QVT permet de faire des transformations bidirectionnelles et du *pattern matching* (filtrage par motif). Il supporte le filtrage complexe des éléments du modèle à l'aide de l'OCL (Object Constraint Language). La traçabilité est créée implicitement et la sémantique est définie par le langage QVT *Core*.

QVT* Core** est une version simplifiée du langage ***QVT Relational. Le *pattern matching* du langage *Core* est limité. Contrairement au langage *Relational*, les traces de la transformation ne sont pas effectuées automatiquement. Le programmeur doit expliciter ce qui doit être tracé. Un fait intéressant de l'interpréteur QVT est qu'il transforme les instructions de langage *Relational* en instructions de langage *Core* pour faciliter l'interprétation. Ceci est analogue aux principes des machines virtuelles qui transforment les instructions d'un langage en instructions plus simples.

QVT Operational est un langage de transformation impératif qui étend le langage QVT *Relational* avec des instructions impératives. Au lieu de déclarer comment le modèle source doit être, le programmeur déclare les transformations à appliquer. Le QVT *Operational* permet seulement des transformations unidirectionnelles et utilise en grande partie le langage OCL. Il est possible d'utiliser presque toutes les fonctionnalités d'OCL. Par contre, l'utilisation d'OCL doit s'imbriquer dans des fonctions appelées *mappings*, *queries*, et *helpers* de QVT, un peu comme dans les langages traditionnels où les instructions doivent être contenues dans une méthode.
Le dernier composant du langage QVT est le *Black Box Implementation*. Il permet d'étendre le langage QVT avec des fonctionnalités qui seraient difficiles à construire à l'aide des langages proposés par QVT ou avec OCL.

II.6.2. ATL (Atlas Transformation Language)

ATL est un langage de transformation hybride inspiré du langage QVT. Il permet à la fois des constructions déclaratives et impératives. Toutefois, les auteurs d'ATL [Allilaire et al. ,06] recommandent l'utilisation du style déclaratif qui simplifie l'écriture des transformations.
Les transformations dans ATL sont unidirectionnelles mais il est possible de faire des transformations bidirectionnelles à l'aide d'un ensemble de règles implémentées à la fois dans les deux directions : le modèle source et le modèle cible.

Les transformations dans ATL se composent de trois modules : Header, Helpers et Rules.

Header est utilisé pour déclarer des informations générales tels que les noms du module et de la transformation, des métamodèles source et cible et des bibliothèques utilisées dans la transformation.

Helpers sont des instructions génériques basées sur l'OCL pour éviter la redondance du code de transformation.

Rules sont des expressions qui décrivent la manière de générer les éléments du modèle cible (basé sur le métamodèle cible) à partir des éléments du modèle source (basé sur le métamodèle source).
Il existe une implémentation d'ATL dans l'IDE d'Eclipse dotée d'un moteur de règles responsable de la compilation et l'exécution des règles de transformation.

II.7. Mécanisme de transformation des modèles

Une transformation de modèles se réalise en général selon trois étapes [Djebbi, 04].

II.7.1. Définition des règles de transformation

Cette étape consiste à mettre en correspondance les éléments du modèle source et les éléments du modèle cible. Il faut donc définir un métamodèle pour mettre en place des règles de transformation génériques. Les instances de ce métamodèle sont des spécifications (règles de transformation) de transformation entre deux métamodèles source et cible. Le processus de transformation prend en entrée un modèle source conforme au métamodèle source et produit en sortie, à l'aide des règles de transformation, un modèle cible conforme au métamodèle source.

II.7.2. Expression des règles de transformation

Il faut un langage de transformation (exemple QVT ou ATL) capable de spécifier, filtrer et sélectionner les éléments de chacun des modèles source et cible pour ensuite les écrire dans un format interprétable par un moteur d'exécution de règles de transformation.

II.7.3. Exécution des règles de transformation

Les règles de transformation, une fois spécifiées et exprimées, requièrent un moteur d'exécution. Celui-ci prend en entrée le modèle et son métamodèle source, les règles de transformation ainsi que le métamodèle du modèle cible. Il produit en sortie le modèle cible.

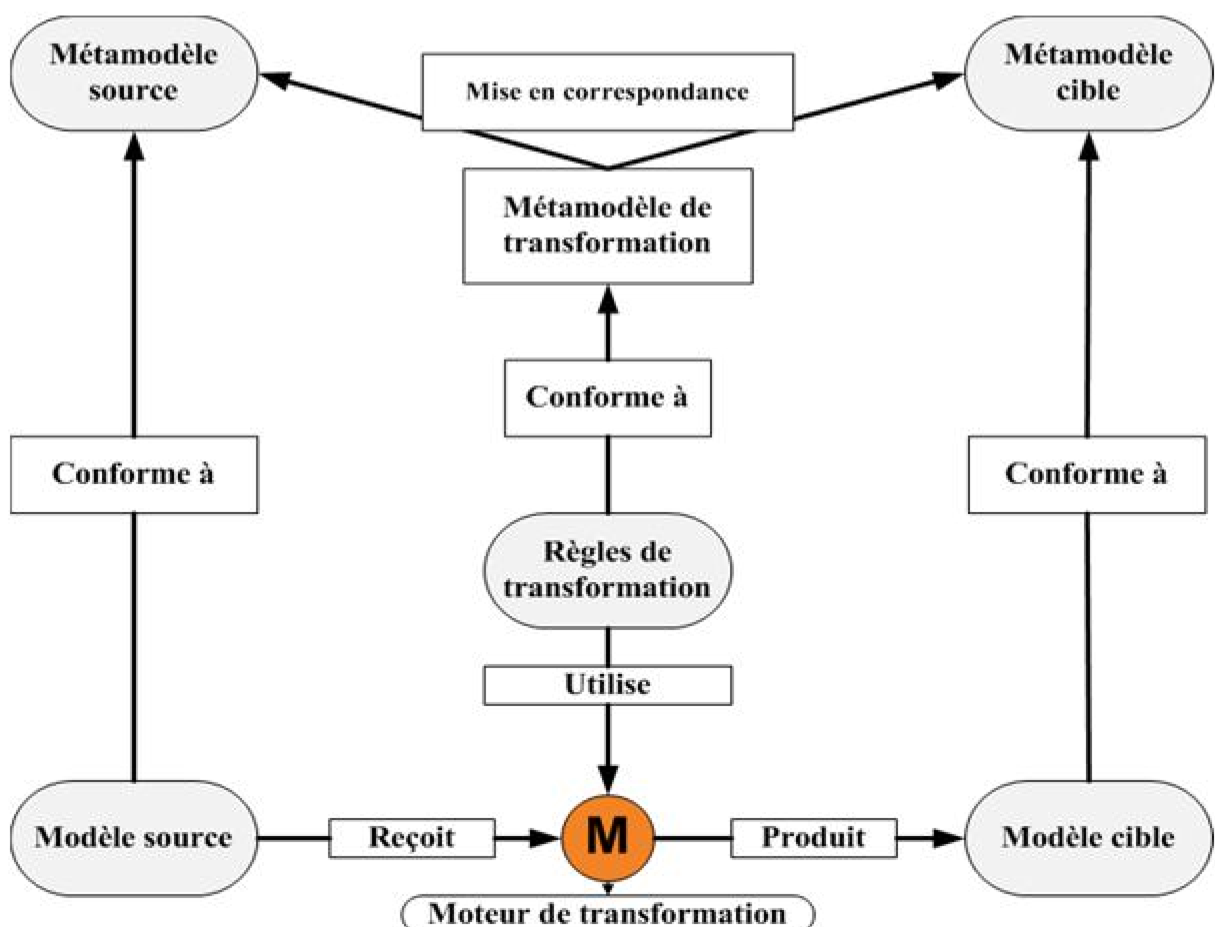


Figure II.5 : Mécanisme de transformation des modèles. [Djebbi, 04]

II.8. Génération de transformations de modèles

Les transformations de modèle requièrent du temps et des compétences particulières. Il est nécessaire pour le développeur de maîtriser à la fois un langage de transformation de modèles et les métamodèles source et cible de la transformation. Dans certains cas, la difficulté pour réaliser une transformation, réside plus dans la taille de la transformation et des métamodèles que dans sa complexité. C'est pour faciliter la création de ces transformations qu'ont été créés deux types d'approches : le premier type se base sur les similarités entre métamodèles et le second type se base sur des modèles d'exemples pour générer des transformations plus complexes. Nous ne présenterons que le premier type.

II.8.1 Génération de transformations de modèle basée sur l'alignement de Métamodèles

Inspirée des méthodes d'alignement de schémas, la génération de transformations de modèles à partir d'alignement de métamodèles permet la déduction de la transformation automatiquement à partir des seules informations contenues dans les métamodèles sources et cibles. [Deloques, 09]. Ce type d'approche se trouve donc approprié dans le cadre de migrations de modèles, c'est-à-dire lorsque les métamodèles sources et cibles sont deux langages proches dédiés à un même but.

Une première approche est présentée dans [Lopes et al., 06] où les auteurs présentent une approche de génération automatique de transformation dans laquelle nous pouvons dégager deux parties :

– Dans une première partie, les auteurs définissent un algorithme d'alignement automatique de métamodèles appelé « SAMT4MDE ». Il s'agit d'établir un ensemble de correspondances entre les éléments du métamodèle source et du métamodèle cible.

Pour cela, l'approche propose de comparer les éléments des métamodèles de même type et d'appliquer sur chaque couple possible une fonction mesurant la similarité entre les éléments. La fonction est définie différemment selon que les éléments à comparer sont des classes, des types primitifs ou des énumérations, mais pour chaque couple elle donne un résultat permettant d'évaluer son degré de similarité.
L'ensemble des couples ayant un degré de similarité suffisamment élevé par rapport à une borne définie formeront alors un alignement.
– Dans une seconde partie, les auteurs définissent un outil de génération de transformations à partir d'un alignement appelé « MT4MDE ». Cet outil est basé sur un métamodèle d'alignement permettant de représenter les différentes correspondances entre les éléments des métamodèles. À partir d'un modèle d'alignement, l'outil génère alors une transformation de modèle en code ATL.
La première partie de cette approche a été par la suite améliorée dans [Lopes *et al.*, 09] . Les modifications portent principalement sur la fonction d'évaluation du degré de similarité, qui évalue non seulement des similarités de valeur au niveau des noms ou des attributs, mais elle évalue aussi la similarité au niveau de la structure des éléments en prenant en compte les éléments qui leur sont liés.

Dans [DEL FABRO et al. 07] les auteurs présentent une approche similaire à la précédente. Il s'agit ici de générer par transformations de modèles un modèle dit de tissage (*weaving model* en anglais) similaire aux modèles d'alignement vus précédemment.
Ce modèle s'obtient grâce à un processus d'alignement qui comme dans les précédentes approches définit d'abord une fonction de similarité appliquée à tous les couples d'éléments des métamodèles source et cible. Cette fonction de similarité permet de définir un degré de similarité pour chaque couple, il est ici calculé à partir de comparaison de chaînes de caractères, de

comparaison de noms à l'aide d'un dictionnaire de synonymes et d'éléments de structure.

Le *similarity flooding* est utilisé dans [DO et al., 02] pour propager la similarité. Le principe du *similarity flooding* est basé sur l'hypothèse que les voisins respectifs d'éléments reconnus comme similaires ont de grandes chances d'être eux aussi similaires. Ainsi les voisins d'éléments ayant une similarité importante voient leur similarité augmenter ce qui permet, par propagation, de compléter l'alignement. L'étape finale consiste à prendre les couples dont la similarité est la plus élevée pour obtenir alors le modèle de tissage permettant la génération d'une transformation. L'approche présentée dans [FALLERI ET AL., 08] se concentre sur la génération d'alignement et reprend la plupart des éléments de l'approche de Del Fabro et Valduriez mais de manière plus détaillée. Il y est notamment présenté les différentes possibilités pour traduire les métamodèles dans un format de données exploitable pour l'algorithme *similarity flooding* et l'impact sur les résultats.

La dernière approche, présentée dans [KAPPEL et al., 06] prend le problème d'un autre point de vue . Les auteurs considèrent que le problème de l'alignement de métamodèles est difficile car les métamodèles ne se contentent pas de décrire un ensemble de concepts mais sont aussi guidés par la manière d'implémenter ces concepts sous forme d'un langage.

Ils considèrent par ailleurs que l'alignement d'ontologies est plus facile car les ontologies sont plus appropriées pour représenter l'ensemble des concepts d'un langage sans se soucier de son implémentation. Les auteurs proposent alors un processus permettant la transposition d'un métamodèle en une ontologie. Cette transposition permet d'utiliser les techniques d'alignement d'ontologies, ici COMA++. Une fois l'alignement obtenu, il est alors possible de remonter vers les métamodèles pour établir un lien entre les éléments du métamodèle source et ceux du métamodèle cible.

II.9. Conclusion

L'étude des transformations de modèle nous a permis de voir l'importance qu'elles ont pour l'automatisation des processus de développement. Les approches pour transformer un modèle se sont multipliées, chacune d'elles a sa propre vision du modèle. Nous avons aussi pu découvrir quelque uns des différents langages qui nous sont apparus intéressants pour ce concept. Puis nous avons présenté les approches pour l'automatisation de ce concept. Dans le chapitre qui suit nous détaillerons le « refactoring » : technique spécifique de transformation sur un seul modèle.

Chapitre III

Refactoring des modèles

III.1. Introduction

Les activités de conception logicielle ne se limitent pas à la création de nouvelles applications en partant de rien ; très souvent le concepteur doit modifier et faire évoluer le comportement et les fonctionnalités d'une application existante.

Il est maintenant reconnu comme étant une bonne pratique de diviser une évolution en deux étapes:

1. Sans introduire de nouveau comportement au niveau conceptuel, restructurer le logiciel pour améliorer des facteurs de qualité tels que la maintenabilité, la lisibilité, l'efficacité etc.

2. Tirer parti de la conception améliorée pour modifier le comportement du logiciel.

Cette première étape appelée *refactoring* est maintenant considérée comme essentielle pendant le développement et la maintenance d'un logiciel.

[Fowler ,99] définit le refactoring comme: "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour".

Le refactoring est venu pour améliorer la qualité du code orienté-objet. Il représente un processus de changement d'un système logiciel visant à améliorer la structure interne du code sans altérer son comportement externe. Le but est d'améliorer la qualité du code, en convertissant du code dégradé dans une forme plus modulaire et structurée. Ainsi, les modifications ultérieures pourront s'effectuer avec une facilité accrue.
Les travaux sur les refactorings se sont dès le départ dirigés vers la transformation de programmes orientés objet pour deux raisons :

- Comparée à des approches du développement plus traditionnelles, la programmation orientée objet facilite les refactorings car elle explicite les informations structurelles nécessaires.

- Le refactoring est plus particulièrement important dans la programmation orientée objet. Certains membres de la communauté accordent une grande importance à la conception et re-conception du logiciel pour le rendre plus réutilisable. Dans certains cas, la meilleure façon d'améliorer le design d'un programme est de le réécrire ; dans d'autres cas, sa restructuration peut être plus aisé.

```
  protected LectureVO[] handleFindLecture
     (java.lang.String title, domain.Weekday day, domain.Time time)
     throws java.lang.Exception
*  { SearchCriteria c = new SearchCriteria();
*    c.setDay(day);
*    c.setTitle(title);
*    c.setTime(time);
     Collection coll =
      getLectureDao().findLecture(LectureDao.TRANSFORM_LECTUREVO,c);
     LectureVO[] lectures = new LectureVO[coll.size()];
     return (LectureVO[])coll.toArray(lectures); }
```

Figure III.1: Code source Java avant l'application du refactoring

```
  protected LectureVO[] handleFindLecture
     (java.lang.String title, domain.Weekday day, domain.Time time)
     throws java.lang.Exception
*  { SearchCriteria c = this.initialise(title,day,time);
     Collection coll =
      getLectureDao().findLecture(LectureDao.TRANSFORM_LECTUREVO,c);
     LectureVO[] lectures = new LectureVO[coll.size()];
     return (LectureVO[])coll.toArray(lectures); }
* protected SearchCriteria initialise
*    (java.lang.String title, domain.Weekday day, domain.Time time)
*    throws java.lang.Exception
* { SearchCriteria c = new SearchCriteria();
*    c.setDay(day);
*    c.setTitle(title);
*    c.setTime(time);
*    return c; }
```

Figure III.2: Code source Java après l'application du refactoring

[Brant et al. , 99] présentent les refactoring comme un outil essentiel pour gérer l'évolution d'un logiciel. Selon eux, puisque les méthodes traditionnelles de développement fondées sur le cycle de vie en cascade placent la maintenance en dernière phase du cycle de vie d'un logiciel, elles ne prennent pas en compte l'évolution de ce logiciel. Ils remarquent aussi que d'autres méthodes s'inspirant plutôt du cycle de vie en spirale offrent un meilleur support pour l'évolution du logiciel. Ces méthodes encouragent l'utilisation des langages de quatrième génération (langages de programmation conçus dans un but spécifique) et des environnements de développement intégrés et sont donc plus appropriées à l'utilisation des refactoring.

III.2. Principe du refactoring

Dans la littérature le terme anglais « refactoring » n'est pas le seul qu'on peut trouver pour décrire ce concept ; les termes refactorisation, restructuration, ou des fois même le terme factorisation sont utilisés pour parler du refactoring, bien que « factorisation » n'englobe pas toute les opérations du refactoring.

La refactorisation sert à améliorer la qualité du logiciel notamment l'extensibilité, modularité, la réutilisabilité, la complexité et la maintenabilité.

La majorité des recherches effectuées sur la refactorisation se concentrent sur le code et se sont moins intéressées à la phase de conception [Chikofsky et al. ,90], [Opdyke, 92], [Marticorena et al., 06]. Cependant, dans l'ingénierie logicielle dirigée par les modèles (IDM), les modèles jouent un rôle prépondérant. Il est donc nécessaire et utile d'étudier les techniques de refactorisation au niveau des modèles.

Cette approche prometteuse consiste à créer des primitives de refactorisation indépendantes du langage. Elle offre l'avantage de pouvoir réutiliser les primitives de refactorisation sur plusieurs plateformes. Dans le cadre général de l'IDM, le procédé de refactorisation peut être exprimé comme un problème de transformation de modèles. En effet, il faut transformer le modèle source qui nécessite une amélioration du point de vue conceptuel en un modèle cible. C'est une transformation endogène où le modèle source et cible obéissent au même métamodèle.

III.3. Refactoring des modèles

Le refactoring des modèles est une technique permettant d'améliorer la qualité intrinsèque d'un modèle tout en conservant son comportement externe et donc sa sémantique. Une grande difficulté, au niveau des modèles, est justement de pouvoir définir les deux notions que sont qualité et comportement de modèle.

De la première dépend la satisfaction globale d'un modèle ou d'une partie de celui-ci : un refactoring s'appliquera en priorité sur une sous-partie d'un modèle qui possède une mauvaise qualité. Exemple d'un refactoring de modèle de classe Figure III.3

De la seconde dépend la légitimité du refactoring : si un refactoring modifie le comportement externe d'un modèle, de par sa nature, il ne peut pas s'appliquer.

Le refactoring de modèle se focalisera donc sur les défauts de types pragmatiques. Le refactoring agit uniquement au sein d'un niveau de modélisation M_1. Le modèle doit rester conforme à son métamodèle avant et après sa transformation.

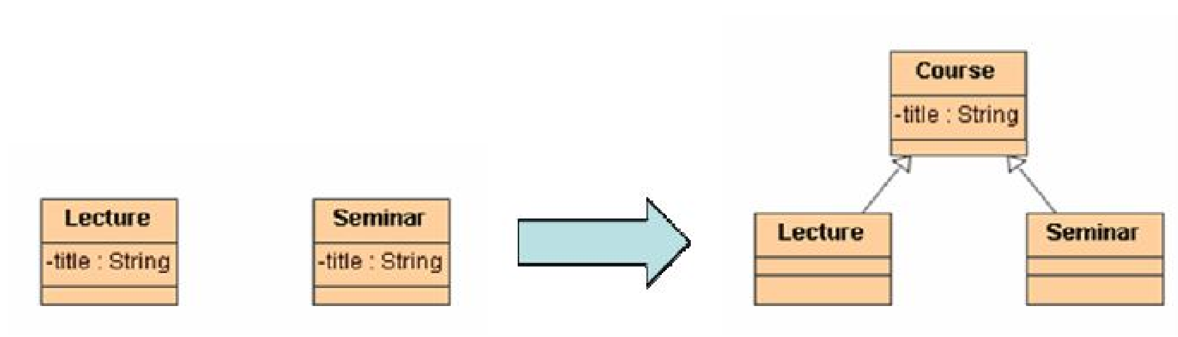


Figure III.3 : Exemple de refactoring de modèle de classe

III.4. Processus de refactoring

Le processus de refactoring des modèles est constitué d'un nombre d'étapes basés sur le processus de refactoring des programmes.

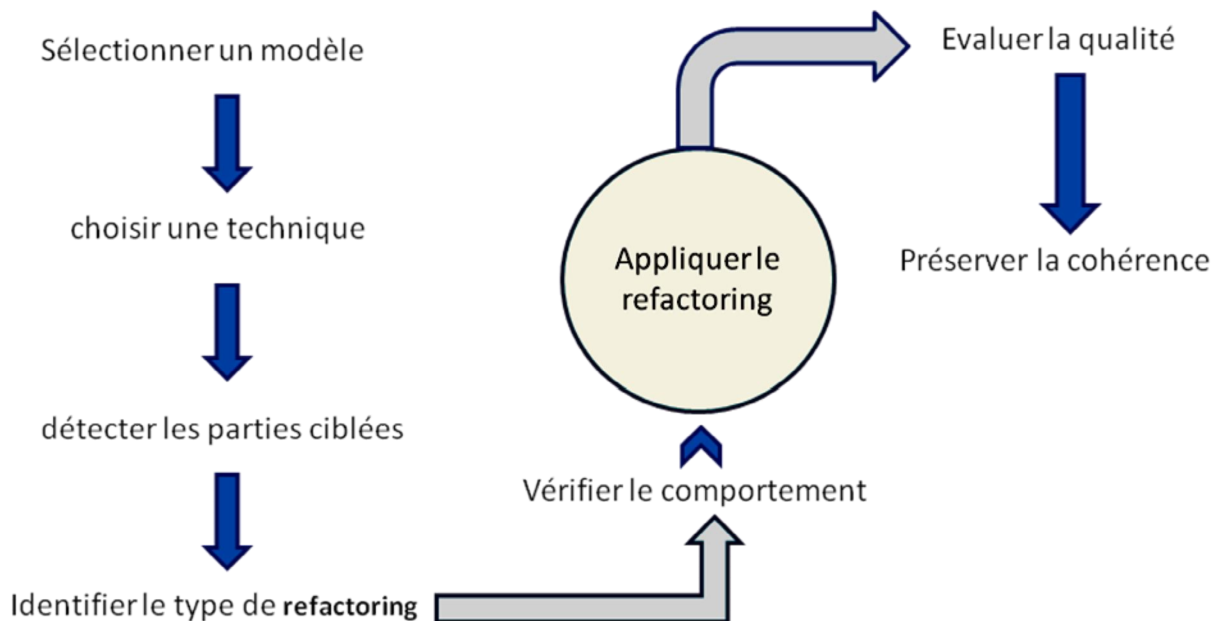


Figure III.4 : Processus de refactoring

III.4.1. Sélection du modèle sur lequel s'applique le refactoring

Il n'est pas facile de maîtriser une infrastructure permettant la transformation de modèles. Pour cette raison, il est important de vérifier si les bénéfices que peut apporter un processus de refactoring sont suffisamment importants pour les coûts associés. Le processus du refactoring peut se gérer de deux manières différentes lors de l'évolution d'un modèle, chacune possédant ses avantages et inconvénients:

- Appliquer des refactorings avant chaque modification majeure du modèle : ceux ci permettront de préparer le terrain pour de gros changements en essayant d'obtenir un modèle mieux structuré et plus modulaire. L'avantage est qu'il est seulement nécessaire de penser au refactoring avant des changements conséquents. L'inconvénient est que ce travail de préparation peut durer relativement longtemps.

- Appliquer de manière continue des refactoring pendant l'évolution du modèle : cette façon de procéder permet de conserver la qualité du modèle tout au long de son cycle de vie. L'inconvénient est qu'il est alors nécessaire de régulièrement consacrer un certain temps aux activités du refactoring.

III.4.2. Choix du modèle de transformation qui sera utilisé pour spécifier les règles de transformation.

Trois principales techniques de transformation de modèles sont utilisées pour le refactoring des modèles : l'approche logique, la transformation de graphes et la transformation par programmation impérative.

- **Transformation logique**. La transformation logique repose sur une base de connaissances regroupant les différents éléments du modèle. Cette base de connaissances, aussi appelée model repository [Störrle, 07], doit être représentée dans un langage logique tel que Prolog ou

SQL. Lorsque cette base de connaissances est construite, il est possible d'y exécuter diverses opérations telles que des requêtes, des transformations, des mesures et des vérifications. Ceci permet de mettre en place tout le processus nécessaire pour supporter les différentes activités de refactoring de modèles. D'autres méthodes se basent sur Prolog pour permettre l'utilisation d'un style complètement déclaratif afin de transformer des modèles [Schätz et al., 08].

- **Transformation de graphes**. Comme expliqué précédemment, les refactoring sont composés de pré-conditions et post-conditions qui garantissent la conservation du comportement si elles sont satisfaites [Opdyke, 1992]. Selon cette vue, il y a une correspondance directe entre refactoring et transformations de graphes. Un refactoring de modèle correspond à une règle de production de graphe. La partie gauche de cette règle correspond à une pré-condition et la partie droite à une post-condition. L'application même du refactoring correspond à une transformation de graphe. Cette façon de procéder correspond bien à l'idéologie de l'IDM qui consiste à se passer autant que possible de code pour privilégier des approches visuelles.
- **Transformation par programmation impérative**. Plusieurs langages de transformation de modèles ont été développés sur la base du paradigme de programmation impérative : on modifie un modèle à travers des effets de bord. Quelques exemples très connus de cette approche sont, entre autres, Kermeta, Operational QVT [Object Management Group, 2008] et ATL. Les deux derniers sont des langages de transformation de type modèle à modèle qui font partie de M2M, un sous-projet d'Eclipse Modeling Project.

Le langage Kermeta a été utilisé par [Moha et al., 09] pour définir des refactoring de modèles génériques indépendants du métamodèle utilisé.

III.4.3 Identification des parties du modèle où le refactoring se réalise

Pour savoir où appliquer un refactoring sur un modèle, il faut d'abord être conscient des qualités et faiblesses de celui-ci. Evaluer la qualité d'un modèle se révèle être une tâche très délicate. Il faut pour cela créer des critères de qualité: performance, lisibilité, modularité, etc. Il arrive que certains critères de qualité soient contradictoires et ne puissent pas être conjointement satisfaits. Pour contourner ce problème, il faut pondérer ces critères en fonction des objectifs à atteindre. Par exemple, pour un logiciel temps-réel, le critère de performance sera plus pondéré que celui de lisibilité. L'évaluation de la qualité d'un modèle est une question de qualité requise et d'équilibre.

Dans le contexte du refactoring de code, des outils existent pour analyser des problèmes de qualité : les *« bad smells »*. Les *bad smells* sont des structures dans le code qui suggèrent la possibilité de refactoring [Fowler, 99]. Une technique similaire est utilisée pour détecter les endroits qui semblent propices à un refactoring dans le cadre des modèles : les « *model smells* » ou « *design critics* » [Robbins, 98]. Ce sont des défauts pragmatiques qui ont pour objectif d'indiquer les parties des modèles qui sont mal structurées. Les *model smells* peuvent se baser sur des heuristiques particulières ainsi que sur des métriques de modèles [Kim, 08]. Les métriques de modèles, à l'instar de ce qui se fait sur le code, correspondent à des valeurs numériques qui permettent d'évaluer certains aspects du modèle.

II.4.4 Identification du type de refactoring à utiliser sur la partie du modèle

Le choix du refactoring à appliquer est relatif au critère de qualité que l'on veut améliorer à un endroit précis du modèle. Prenons le cas concret d'un *model smell* qui nous indique une data class, c'est-à-dire une classe qui contient significativement plus de données que de comportement et donc plus d'attributs que d'opérations. Les refactorings à appliquer seront ceux qui tenteront de relocaliser des opérations dans cette classe, ou de déplacer des attributs de cette classe, afin d'améliorer son équilibre.

De même que nous cherchons à trouver le refactoring le plus apte à corriger un défaut dans un modèle, il peut être nécessaire d'évaluer la qualité d'un modèle après son refactoring pour voir si, dans l'ensemble, celle-ci s'est améliorée. La façon la plus simple de procéder est de vérifier à nouveau la présence de *model smells* et de voir si la situation globale est meilleure ou pire qu'avant le refactoring. Il est tout à fait possible que plusieurs nouveaux *model smells* apparaissent suite à un refactoring prévu pour en corriger un autre. Le tout est une question d'ordonnancement ainsi que d'équilibre entre les *model smells* corrigés, les *model smells* créés et les pondérations des uns et des autres.
Un autre aspect qui peut influencer le choix du refactoring est la présence de dépendances entre les refactoring. Parfois, le fait d'appliquer un refactoring sur un modèle peut empêcher l'application d'un autre (dépendance parallèle). Inversement, l'application d'un refactoring peut introduire une nouvelle opportunité de refactoring (dépendance séquentielle). Une analyse formelle des refactoring permet d'identifier ces dépendances [Rouane et al., 07].

III.4.5 Vérification de la conservation du comportement après le refactoring

Une des méthodes permettant de préserver le comportement d'un modèle lors d'un refactoring est de transformer ce modèle dans une entité dont le comportement est vérifiable. Il est par exemple possible de transformer un modèle en code source ou en utilisant une représentation formelle. Le comportement de ces deux représentations est vérifiable : le code source à l'aide des tests unitaires et la représentation formelle à l'aide de propriétés mathématiques du formalisme choisi. Il reste ensuite à tester si le comportement est le même avant et après le refactoring.

Dans la pratique, pour garantir la préservation du comportement, nous avons besoin d'un large panel de comportements qui devraient être vérifiés. Les logiciels temps-réel ont des contraintes de temps d'exécution, les logiciels embarqués ont des contraintes de ressources et les systèmes critiques ont des

contraintes de sécurité. Toutes ces contraintes devraient être préservées par le refactoring.

III.4.6 Application du refactoring

III.4.7 Evaluation des effets du refactoring sur la qualité de l'application

Après l'application du refactoring, il est nécessaire de vérifier que la qualité de l'application dont le modèle a subi un refactoring n'a pas était détériorée et que le refactoring n'a pas entrainé d'autres refactoring. Le but du refactoring étant de faciliter la maintenance et non pas d'impliquer d'autre maintenance.

III.4.8 Préservation de la consistance entre les modèles

En règle générale, un modèle est construit à partir de différentes vues, représentées par plusieurs diagrammes. L'approche UML, entre autres, utilise cette représentation. Si un refactoring est appliqué sur un diagramme au sein d'un modèle UML, il est probable que celui-ci ne soit plus cohérent avec certains autres diagrammes qui composent le modèle.

Si la situation se présente, il existe principalement deux façons de procéder :

(i)- Le refactoring transforme le diagramme sans tenir compte des incohérences qui pourraient survenir dans les autres diagrammes.

(ii)- Le modèle composé de tous les diagrammes affectés par le refactoring est analysé pour s'assurer que le refactoring n'introduit pas de nouvelles incohérences. Pour ce faire, il faut tenir compte de tous les métamodèles des modèles concernés. Figure III.5

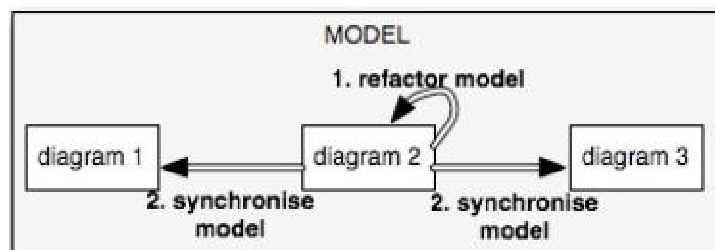


Figure III.5 : synchronisation des différents diagrammes d'un modèle.

III.5. Refactoring des diagrammes de classes

Les refactoring présentés par Opdyke dans sa thèse [Opdyke, 92], s'appliquent essentiellement à trois concepts : les classes, les méthodes et les variables.

Nous avons donc naturellement commencé la transposition des refactoring exitants vers UML, en nous intéressant aux diagrammes de classes.

Les refactoring présentés ici peuvent être classés en quatre types d'opérations de base : ajout, suppression, généralisation et spécialisation d'éléments de modèle. Les deux derniers types déplacent des éléments à travers la hiérarchie d'héritage, le long des relations de généralisation. La plupart des éléments de modèle composant un diagramme de classes peuvent avoir une connexion directe à des éléments d'autres vues. Certains refactoring qui s'appliquent aux diagrammes de classe ont donc potentiellement un impact sur d'autres vues d'UML.

III.5.1 Ajout et suppression d'éléments

L'ajout de membres (attributs et méthodes) et d'associations à une classe est possible quand le nouveau membre ou la nouvelle association n'a la même signature d'aucun autre membre ou association de la classe considérée ou d'une super classe ou sous-classe de celle-ci.

La suppression d'associations et de membres n'est possible que si l'élément supprimé n'est pas référencé dans le modèle entier ; une méthode peut par exemple être référencée dans un diagramme d'interaction si elle est liée à un message ou un stimulus, ou encore dans un diagramme d'états si elle est liée à une action ou un événement.

L'ajout et la suppression de classes devient particulièrement intéressant quand la hiérarchie d'héritage est prise en compte. On peut insérer une instance de GeneralizableElement au milieu d'une relation de généralisation, entre deux éléments parents. L'élément inséré ne doit pas

introduire de comportement, et surtout être du même type que les deux autres par le biais d'instances, de membres etc.

III.5.2 Généralisation et spécialisation

Le refactoring de généralisation peut être appliqué aux éléments possédés par les classes, comme les attributs, les AssociationEnd, les méthodes, les opérations, ou même les diagrammes d'états. Il consiste à la fusion de plusieurs éléments en un seul qui est ensuite transféré vers une super-classe commune.

Puisque les membres privés ne sont pas accessibles depuis les sous-classes, on ne peut pas les déplacer ainsi. Cette transformation implique que toutes les sous-classes directes de la super-classe partagent un élément équivalent: pour les attributs, les AssociationEnd ou les opérations cette équivalence peut être vérifiée structurellement, mais le problème est plus difficile pour les méthodes et les statecharts.

Le refactoring de spécialisation est l'inverse du précédent : il envoie un élément d'une classe dans toutes ses sous-classes. Informellement, il préserve le comportement si la classe d'origine n'est pas le *contexte de référence* de l'élément, c.-à-d. si l'élément n'est utilisé (p.ex. par des contraintes) que via des instances de sous-classes de la classe d'origine. Ce contexte de référence peut être obtenu dans les diagrammes d'objets ou, si le corps des méthodes est disponible, par analyse des actions de lecture et d'écriture des attributs.

On peut obtenir le contexte de référence des AssociationEnd grâce aux diagrammes d'objets et de collaborations et celui des méthodes grâce aux diagrammes d'interactions et d'états.

D'autres problèmes peuvent survenir si on néglige l'existence de l'héritage multiple en UML. Il faut vérifier que les classes qui vont recevoir l'élément transféré n'ont pas de sous-classe commune, c'est-à-dire que le schéma

d'héritage traditionnellement problématique en diamant ne se présente pas. S'il existe une sous-classe commune, après la transformation elle hériterait de deux membres équivalents, ce qui serait une erreur conceptuelle.

III.6. Approches de refactoring

Nous allons présenter les principales approches existantes pour le refactoring des modèles.

III.6.1. Approche de Lieberherr

L'objectif de cette approche est la factorisation maximale des attributs, en minimisant l'utilisation de l'héritage multiple. Cette approche utilise le nom ainsi que le type des attributs. Il faut noter que le type d'un attribut ne peut être la classe où il est introduit, ce qui est une limitation. Elle peut ajouter des classes abstraites dans le modèle initial.
Deux types d'algorithmes sont introduits par cette approche de reconstruction : un algorithme global [Lieberherr et al. ,90], et un algorithme incrémental [Lieberherr et al. ,91] .

III.6.2. Approche de Moore

Cette approche s'appuie, dans un premier temps, sur le nom des méthodes. Ensuite elle s'étend au code de ces méthodes en factorisant les parties dupliquées dans des méthodes supplémentaires qui seront ajoutées dans des classes abstraites. Dans les deux cas, l'algorithme de reconstruction utilisé est le même. L'objectif de l'approche est la factorisation maximale des méthodes.

Moore présente une approche de reconstruction de modèles de classes développée dans le contexte du langage Self [Moore et al., 96]. Un graphe biparti est construit avec deux ensembles de nœuds : les classes et les features (les features désignent indifféremment les attributs et les méthodes, qui ne sont pas différenciés en Self). Chaque classe est reliée à ces features et celles dont elle hérite.

À partir de ce graphe biparti, un nouveau graphe biparti est construit en remplacent les nœuds représentant les classes par des nœuds représentant un ensemble de classes. Ces nœuds peuvent être reliés si un ensemble de classes est inclus dans un autre.

Maintenant, un ensemble de classe est relié à une feature si celle-là a était relié à toute les classes de cette ensemble dans le premier graphe.

Le graphe représente donc maintenant la relation d'ordre partiel induite par l'inclusion des ensembles de classes. Enfin, la réduction transitive de ce graphe est calculée. Le graphe résultant forme donc la hiérarchie de classes finale produite par cette approche. Le modèle de classes produit par cette approche est un sous-ordre particulier de treillis. [Moore et al., 96]

III.6.3 Approche de Chen

L'objectif de cette approche est la factorisation maximale des attributs et méthodes. Cette approche s'appuie sur le nom et la signature des attributs et méthodes. Les auteurs respectent la théorie des types et demandent que les redéfinitions de méthodes suivent la règle de la covariance du type de retour et la contravariance du type des arguments. Cette approche peut ajouter des classes et des méthodes dans le modèle initial.

L'article de Chen [Chen et al., 96] décrit un algorithme de reconstruction globale très semblable à celui de Moore [Moore et al., 96]**.** . Le modèle de classes produit par cette approche est un sous-ordre particulier de treillis.

III.6.4 Approche de Cook

Cette approche a pour objectif de construire une hiérarchie d'interfaces en factorisant le nombre maximal des méthodes issues d'une hiérarchie de classes. L'approche s'appuie uniquement sur le nom des attributs et peut ajouter des classes abstraites dans le modèle initial.

La méthode utilisée est présentée dans [Cook, 92]. C'est une méthode qui permet de restructurer une hiérarchie d'interfaces extraite de code Smalltalk. L'auteur applique tout d'abord une étape de prétraitement pour extraire les interfaces d'un ensemble de classes Smalltalk. En effet, Smalltalk permet de désactiver l'héritage de certaines méthodes. Une interface correspond donc à l'ensemble des méthodes réellement possédées (y compris celles héritées) par une classe. À la fin de cette étape de prétraitement, on dispose donc d'un ensemble de couples (*classe, méthodes*) où *méthodes* est l'ensemble des méthodes réellement possédées par *classe*. L'auteur calcule ensuite la relation inverse (*méthode, classes*) où *classes* est l'ensemble des classes où apparaît *méthode*. Enfin, l'auteur calcule l'ensemble des couples (*classes, méthodes*) des *méthodes* possédées par un ensemble donné de *classes*. Finalement l'ensemble des couples (*classes, méthodes*) est ordonné par un tri topologique basé sur l'ordre partiel induit par l'inclusion des ensembles de *classes*. Le modèle de classes produit par cette approche est un sous-ordre particulier de treillis

III.6.5. Approche de Casais

Cette approche s'appuie sur le nom et le code des méthodes pour leur factorisation maximale. Elle peut ajouter des classes abstraites dans le modèle initial ainsi que des méthodes abstraites

E.Casais [Casais, 92] présente une approche où la hiérarchie existante est modifiée à chaque ajout d'une nouvelle classe dans la hiérarchie. Cette approche qui a été développée pour le langage Eiffel, utilise la propriété qu'a ce langage pour bloquer l'héritage de certaines méthodes, pour modifier la hiérarchie existante. Plus précisément, le procédé est appliqué à chaque ajout d'une classe **C** par le concepteur.

L'algorithme prend en entrée la classe **C** que l'on veut rajouter, et l'ensemble des ses super classes directes **S** que le concepteur a définies. L'idée est que si **C** bloque l'héritage de certaines méthodes de ses super

classes, ou si **C** redéfinit des méthodes déjà concrètes, on peut considérer que la hiérarchie de classes peut être améliorée. Ainsi pour chacune des super classes de **C**, s ∈ **S** qui pose problème, on crée une classe **f** qui factorise les méthodes communes de **C** et s, et qui contient les méthodes de même nom qui sont concrètes dans **C** ainsi que dans s (si, bien sûr le corps de ces deux méthodes est différent, car dans le cas inverse ce n'est pas une redéfinition).

Ce procédé est ensuite appliqué de manière récursive, en ajoutant **f** comme sous-classe de l'ensemble des super classes directes de **C**. Quand les relations d'héritage des classes ne deviennent plus problématiques (il n'y a plus de classes dans les situations précédemment décrites), les ensembles de méthodes déclarées par les classes sont remis à jour. Le modèle de classes produit par cette approche est un graphe sans circuit. Il ne peut exister qu'un niveau de redéfinition entre les méthodes (une méthode abstraite est redéfinie par une méthode concrète, une méthode concrète ne peut être redéfinie).

III.6.6. Approche AFC

L'objectif de cette approche est de factoriser le maximum des attributs et méthodes. Pour cela cette approche s'appuie sur le nom ou la signature des attributs et méthodes. Elle peut parfois s'appuyer sur des relations de spécialisation connues sur ces attributs, leur type, ou les méthodes [Dao et al. , 04]. Elle peut ajouter des nouveaux attributs, classes et méthodes (notamment des méthodes abstraites) dans le modèle initial.

L'Analyse Formelle de Concept (AFC) a été utilisée par plusieurs approches de reconstruction des hiérarchies de classes [Miralles et al., 2014]. Le fonctionnement général de ces approches reste toutefois similaire. Dans la description de cet algorithme nous nommerons feature les éléments contenus par les classes, qui peuvent être des attributs ou des méthodes.

Tout d'abord une relation binaire est construite entre les classes et les features. Un couple (*classe,feature*) est ajouté dans la relation binaire si *classe* introduit la *feature* ou l'hérite. Cette relation binaire, ainsi que l'ensemble de classes et l'ensemble de features permettent de créer un contexte formel. L'AFC est appliquée sur ce contexte. Le résultat de l'application de l'AFC est un treillis de concepts où chaque feature présente dans la hiérarchie initiale n'est introduite qu'une seule fois, éliminant ainsi les redondances.

Le modèle de classes produit par cette approche est un treillis. Ce treillis peut aussi être filtré pour obtenir un sous-ordre de treillis, nommé sous-hiérarchie de Galois. Dans ce cas, on obtient un nombre minimal de classes nécessaires pour introduire sans redondances toutes les features.

III.6.7. Approche ARC

L'objectif de l'approche ARC est la factorisation maximale des attributs et méthodes. [Rouane et al., 07] . Cette approche peut en outre être appliquée aux associations (par exemple celles dans les modèles UML). Cette approche s'appuie sur le nom et la signature des attributs et méthodes, le nom des associations et le type d'extrémités des associations, elle peut ajouter des nouveaux attributs, classes et méthodes dans le modèle initial. L'algorithme de reconstruction prend en paramètre une famille relationnelle de contextes, comportant des contextes qui décrivent les différents éléments à restructurer (classes, associations, attributs et méthodes), ainsi que plusieurs contextes présentant les relations entre ces éléments (classes qui typent les attributs et les méthodes, possession des attributs par les classes,. . .). L'ARC est ensuite appliqué sur cette famille de contextes relationnels, pour donner naissance à une famille de treillis. Cette famille de treillis est enfin analysée pour donner naissance à un modèle de classes, comportant des nouveaux attributs, classes, associations et méthodes. Le

modèle de classes produit par cette approche est un ensemble de treillis. Ces treillis peuvent aussi être filtrés pour obtenir des sous hiérarchies de Galois (comme pour les approches à base d'AFC).

III.7. Conclusion

Après avoir défini le modèle refactoring, nous avons présenté les étapes qui constituent le processus de refactorisation des modèles, puis nous avons finalisé par quelques approches utilisées pour le refactoring des modèles.
Dans le chapitre qui suit nous allons présenter la cohérence des données dans une base de données et la cohérence entre une base et son modèle, puis nous allons voir les approches utilisées pour contrer chaque type d'incohérence.

Chapitre IV

Cohérence et évolution des modèles

IV .1. Introduction

Les applications de bases de données, au cours de leur cycle de vie, comportent généralement des phases de maintenance correspondant soit à des corrections, soit, plus généralement à des modifications d'évolution. Ces modifications proviennent de nouveaux besoins de type fonctionnel (satisfaire les exigences des utilisateurs), organisationnel (modification du cadre de travail dans lequel s'inscrit l'application) ou technique (modifications de contraintes techniques ou matérielles). Pour satisfaire ces nouveaux besoins, le concepteur va apporter des modifications techniques à son application.

Pour Sheeren ces modifications sont la cause du manque de cohérence entre les données des bases de données [Sheeren, 05]. En effet les bases n’ont pas les mêmes rythmes de mise à jour et la politique d’entretien des données n’est pas nécessairement la même (mise à jour à la demande, mise à jour en continu, etc.). Les données d’évolution ne sont donc pas intégrées au même moment et des incohérences entre les jeux de données peuvent apparaître. Pour éviter ce genre d’incohérence les approches d’intégration de données qui seront définies dans la section IV.3 peuvent être utilisées.
La maintenance des bases de données peut se faire pendant la phase de modélisation, les concepteurs ont tendance à faire évoluer les différents modèles sur lesquels ils travaillent afin d'ajouter de nouvelles fonctionnalités. Ceci risque d'engendrer des problèmes d’incohérence structurelle, à partir du moment où les modèles sont la représentation structurelle des données et que la modification de l'un d'eux peut entrainer l'incohérence du système en entier, d'où la nécessité de répercuter et d’adapter les modifications, ou tout au moins d'identifier les éléments de modèles qui seront impactés par les changements.

IV.2. Cohérence dans les applications de base de données

On distingue deux niveaux de cohérence dans les applications de base de données : la cohérence horizontale et la cohérence verticale.

IV.2.1 La cohérence horizontale concerne des données dans une base de données, elle peut être de type individuelle ou mutuelle [Pontisso, 09].

- **La cohérence individuelle**

Un intervalle de validité temporelle est défini pour chaque objet de la base de données pour maintenir la fraîcheur des objets. Cette cohérence des objets par rapport à leur fraîcheur est aussi appelée cohérence absolue.

C'est alors au système gérant la base de données de contrôler la mise à jour des objets de façon à ce qu'ils ne dépassent pas leurs intervalles de validité temporelle et que lorsqu'une transaction est validée (réalisation d'un commit), les valeurs utilisées par cette transaction n'auraient pas dépassé leurs domaines de validité, c'est-à-dire que les données ne soient pas devenues obsolètes.

Dans [Pontisso, 09], les auteurs cherchent la plupart du temps à déterminer un ordonnancement des transactions permettant de garantir une fraîcheur déterminée des données. L'approche retenue se base souvent sur des transactions périodiques mais les variantes sont multiples.

Ces approches se concentrent uniquement sur le maintien de la fraîcheur de chaque objet pris individuellement sans prendre en compte leur fraîcheur mutuelle. Cela peut alors aboutir à une vue incohérente du système.

- **La cohérence mutuelle**

Les objets d'un système peuvent être liés les uns aux autres. Il est donc important de préserver la fraîcheur mutuelle des objets en plus de leur fraîcheur individuelle. La prise en compte des relations entre objets aboutit à une notion de cohérence mutuelle.

La cohérence mutuelle impose que les données respectent d'une part leur fraîcheur individuelle mais également que leurs âges ne soient pas trop dispersés.

Chaque objet du système est daté. Une notion de dispersion mesurant la différence entre les âges de deux objets est ajoutée. Les auteurs s'intéressent alors à la cohérence temporelle des données en fonction de l'âge et de la dispersion des données. Les objets sont temporellement cohérents si leurs âges et leurs dispersions sont suffisamment petits pour correspondre aux besoins de l'application.

L'objectif du maintien de la cohérence mutuelle est de garantir que les mises à jour des données produisent une vue cohérente du système.

IV.2.2 la cohérence verticale : Elle concerne la cohérence du modèle de représentation et des données de la base de données, appelée aussi cohérence structurelle.

La cohérence d'une base est spécifiée au moyen de contraintes d'intégrités, certaines de ces contraintes sont inhérentes au modèle de données, elles portent sur la correspondance structurelle et sémantique entre les données de la base et son modèle de représentation.
Une évolution du modèle doit être propagée sur les données de la base (et inversement), afin que celles-ci respectent toujours les contraintes de correspondance structurelle et sémantique entre les données et le modèle.

Par exemple, à la suite de l'ajout de l'attribut **Att** dans la classe **C**, les représentations des objets associés à **C** et aux classes qui héritent **Att** de **C** doivent être modifiées. La modification consiste à ajouter l'attribut **Att** à la représentation de chacun de ces objets et à initialiser sa valeur.

IV.3. Intégration des données.

Un système d'intégration a pour objectif d'assurer à un utilisateur un accès à des sources multiples, réparties et hétérogènes, à travers une interface unique. En effet, l'avantage premier de créer un tel système est que l'utilisateur se préoccupe davantage de ce qu'il veut obtenir plutôt que comment l'obtenir, l'objectif étant l'obtention d'informations.

Ces systèmes peuvent être utilisés pour gérer la cohérence des données dans les bases de données hétérogène. Il permet de dispenser le mainteneur de chercher la donnée à modifier dans les différentes sources de données, et interroger chacune de ces sources de données en utilisant sa propre interface pour la modification.

On distingue deux approches pour l'élaboration d'un système d'intégration: la première est l'approche «non matérialisée», appelée aussi approche virtuelle qui consiste à migrer les requêtes vers les sources de données via un médiateur unique globale et des adaptateurs au niveau de chaque source.

La seconde est l'approche «matérialisée», appelée aussi approche d'entreposage qui consiste à migrer les données pour les centraliser dans une source cible. Dans cette approche un médiateur interagit avec un entrepôt de données regroupant toutes les données des différentes sources.

IV.4. Evolution des modèles

Afin de maintenir la cohérence entre les modèles du système lors de leurs évolutions, des opérations de migration peuvent être appliquées afin de propager les changements d'un modèle à un autre. L'opération de migration est

une opération de transformation qui concerne seulement les éléments nécessitant un ajustement.

Comme cela sera détaillé dans les sections suivantes, ces éléments sont spécifiés dans un modèle de différence, propagés via un module de transformation ou bien représentés dans une ontologie des besoins

IV.4.1 Approches d'évolution de modèle

Plusieurs approches dans la littérature traitent une ou des parties du problème d'évolution de modèle. Dans cette section nous nous intéressons à quelques-unes d'autres elles.

IV.4.1.1 Approche de Bounif et al. [Bounif et al., 09]

H. Bounif, aborde le problème de l'évolution des modèles en anticipant les changements potentiels sur ces dernier [Bounif et al., 09] . L'approche adopte la perspective, à priori, c'est-à-dire de planifier à l'avance une solution possible qui fait évoluer le modèle de la base de données au fil du temps. Cette solution fonctionne avec un ensemble d'hypothèses et de techniques. L'approche est élaborée sur la base de différents éléments représentés dans la Figure IV.1.

Tout d'abord, elle propose de développer un dépôt de modèles. Ce dépôt contient un grand nombre de modèles hétérogènes ainsi que leurs versions. Ce dépôt sert également, dans une série d'opérations qui sont importantes, pour l'étude des modèles comme la sélection des modèles pertinents et la construction des matrices de changement. Ces matrices servent à donner des renseignements sur les composantes de modèles, à savoir leurs fréquences et le type de changements qu'ils ont subi.

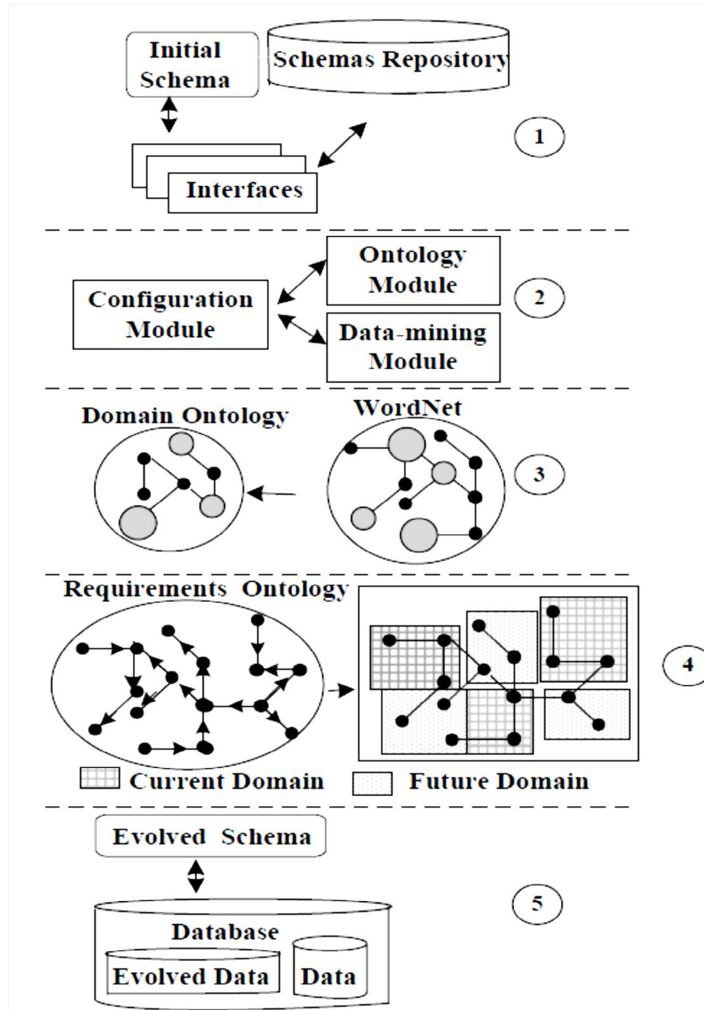


Figure IV.1: Eléments de l'approche Bounif [Bounif et al., 09].

Puis elle propose de développer une ontologie des besoins. Cette ontologie du domaine décrit les concepts actuels d'une base de données ainsi que les concepts qui représentent les potentiels futurs besoins et qui pourraient faire partie de l'évolution de la base. La méthode de construction de cette ontologie comprend plusieurs phases, comme la préparation des données pour cette ontologie, la création du dictionnaire de données et la mise en place des perspectives de changements.

Ces perspectives stockent les concepts qui sont associés à d'autres concepts par des relations résultant de l'application du processus de data mining, l'évaluation de l'ontologie de domaine en utilisant une structure graphique et enfin la construction de l'ontologie en utilisant un ensemble de procédures.

Enfin elle propose de développer des modèles prédits, qu'elle définit comme des modèles de base de données qui contiennent deux types de métadonnées : les métadonnées, qui représentent les entités actuelles de la base de données et les métadonnées qui représentent les entités potentielles futures.

Le modèle prédit possède deux méthodes de représentation: la multi-représentation et les dépôts de modèles prédits. Le modèle de la base de données avec la multi-représentation est construit comme un modèle avec ses versions. Le modèle avec les dépôts de modèle prédits est construit comme un modèle mais entouré de dépôts pour les futurs besoins.

IV.4.1.2 Approche de Cicchetti et al. [Cicchetti et al. ,08] :

Les auteurs s'intéressent à l'évolution des modèles homogènes et proposent une approche à base de transformation de modèles permettant d'adapter des modèles existants à la suite de modifications effectuées sur un modèle tiers. Cette approche opère par exécution séquentielle de deux transformations (Figure IV.2) qui consistent à obtenir en premier lieu un méta modèle de différence (MMD) à partir duquel le modèle de différence contenant les changements sera instancié, suivi ensuite d'un modèle contenant les opérations de migration.

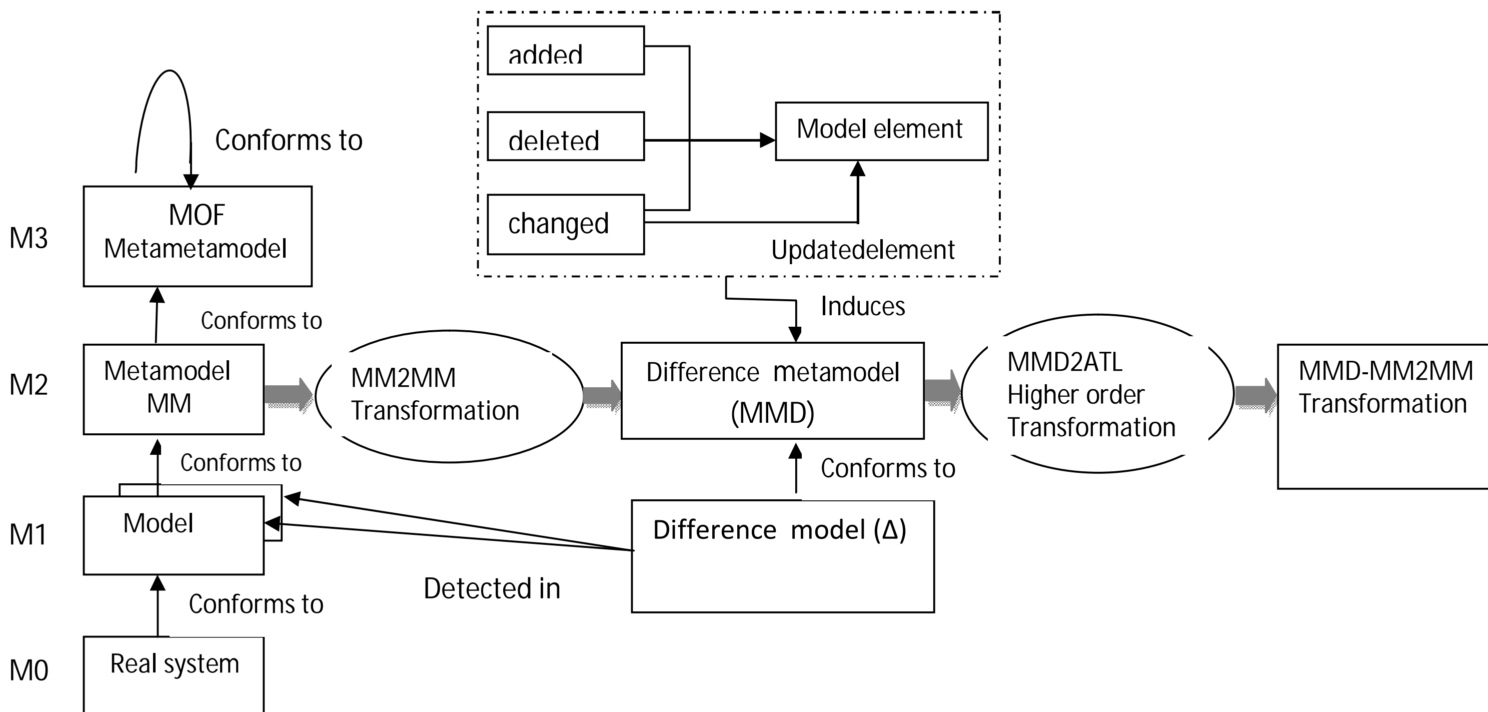


Figure IV.2 : Structure générale de l'approche de Cicchetti et al. [Cicchetti et al. ,08]

Le modèle de différence (Δ), est conforme à un méta modèle de différence (MMD) généré à partir des méta modèles sources (MM). Il permet d'identifier les éléments de modèles sur lesquels les modifications doivent être appliquées, les éléments qui ne sont pas affectés par les changements ne sont pas référencés dans le modèle de différence.

Afin de représenter les différences entre les deux modèles, l'approche (ciblant les modèles homogènes) propose la création d'un méta modèle de différences, étendant leur méta modèle commun de base par application d'une transformation MM2MMD. Pour chaque méta classe (MC) du méta modèle source, les méta classes supplémentaires «AddedMC», « DeletedMC», et «ChangedMC» sont générées.

De façon générale le méta modèle de différences n'est autre que le méta modèle de base incluant les concepts d'ajout, de suppression et de modification pour chaque méta classe.

Dans cette approche le méta modèle de différence dépend du méta modèle source. Dans [Rivera et al. , 08] les auteurs proposent un méta modèle de différence indépendant du méta modèle source, composé de 3 méta classes fondamentales : « Addedelt », « Deletedelt » et « Modifiedelt ». Ainsi, chaque élément du modèle de différence (DiffElement) appartiendra à l'une des méta classes de la Figure IV.3, selon que l'élément ait été ajouté, supprimé ou modifié.

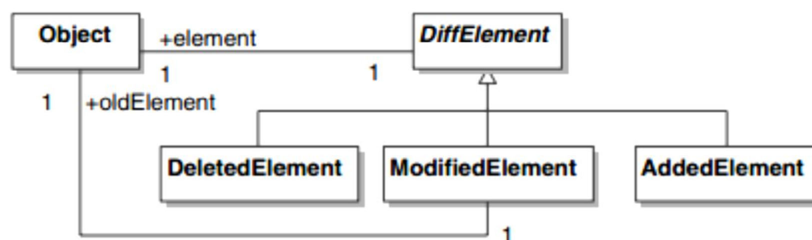


Figure IV.3 : méta modèle de différence [Rivera et al. , 08].

L'opération de migration est effectuée en exécutant la transformation MMD_MM2MM, qui est générée automatiquement à travers une transformation HOT (Higher Order Transformation) (transformation qui produit une autre transformation MMD2ATL) [Bézevin , 05].

Le code ATL de migration prend en entrée le modèle d'entrée ainsi que le modèle de différence et produit un autre modèle en fonction des changements repérés

IV.4.1.3 Approche DB-MAIN [Englebert, 95]

Les concepteurs de l'atelier DB-MAIN trouvent qu'une application bien documentée reste le bon moyen pour le concepteur pour faciliter la

maintenance de l'existant. Ils considèrent l'analyse des modifications dans le cadre de la modélisation classique qui définit essentiellement trois niveaux d'abstraction : conceptuel, logique et physique. (Figure IV.4).

Les modifications des besoins interviennent dans un de ces niveaux. Le schéma conceptuel répond aux besoins R1, le schéma logique aux besoins R2 et l'application aux besoins R3. Le problème de l'évolution sera de les propager aux autres niveaux (en amont et/ou en aval).

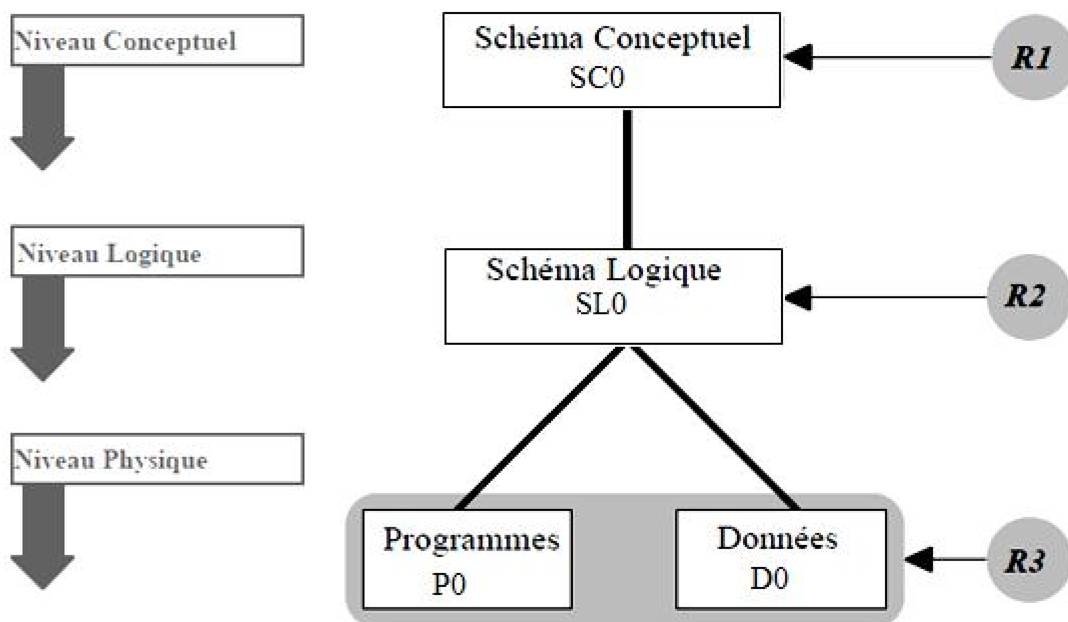


Figure IV.4 : Modélisation classique définie en trois niveaux d'abstraction.

Dans un premier scénario, ils supposent que tous les niveaux de la modélisation d'une application existent (SC0 est le schéma conceptuel, SL0 le schéma logique, P0 et DO les programmes et les données du niveau physique).

Ils vont construire l’historique du processus de production de SL0 à partir de SC0, en enregistrant dans une fonction appelée Histo , toutes les opérations appliquées à SC0 pour obtenir SL0.

Ensuite, les besoins R1 (auxquels répond le schéma SC0) évoluant, se transforment en R1'. Ce changement est traduit par l'analyste en modifications de SC0, qui devient ainsi SC1 (Figure IV.5).

Là, l'historique est utilisé pour transformer SC1 en SL1 : en faisant appel à la fonction Histo à partir de SC1, on obtient SL1 proche de SL0 mais dans lequel les modifications de SC1 ont été propagées, ces modifications concernent des ajouts ou des retraits de concepts.

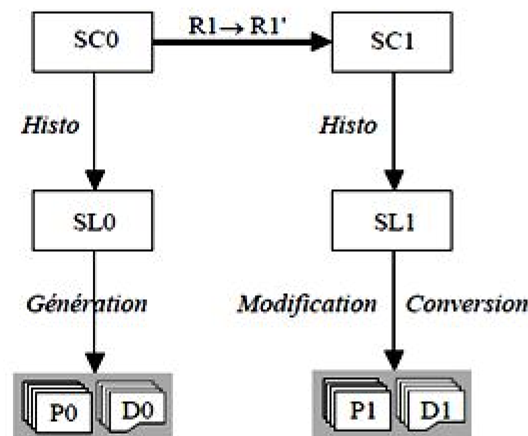


Figure IV.5 : Propagation vers l'aval des modifications déduites de (R1⟶ R1') et apportées à SC0.

P0 et D0 doivent aussi tenir compte des nouvelles spécifications. La conversion des données ce fait à travers des convertisseurs sous la forme de scripts DDL, ou encore de programmes dans les situations plus complexes.

Pour les programmes ils proposent donc le développement d'un outil d'annotation des programmes qui signalerait au programmeur les sections de code à modifier, et qui lui indiquerait la nature des modifications à effectuer.

Dans un deuxième scénario, les modifications sont apportées à SL0 (par exemple ajouter une colonne à une table) suite au changement des besoins R2 en R2' (Figure IV.6).

Dans un premier temps, le contexte est le même que dans le premier scénario : on dispose d'un historique (Histo) contenant les transformations réalisées pour passer de SC0 à SL0. D'autre part, ils s'appuient sur le fait que les transformations conceptuel/logique sont pour la plupart réversibles. Ils désignent par Histo' le scénario à jouer pour remonter de SL1 à SC1. Histo' est l'inverse de Histo (SC0 vers SL0) c'est-à-dire un historique où la séquence a été inversée et où chaque transformation est remplacée par son inverse. Cette inversion d'historique permet de trouver SC1 proche de SC0 avec les modifications de SL1 propagées.

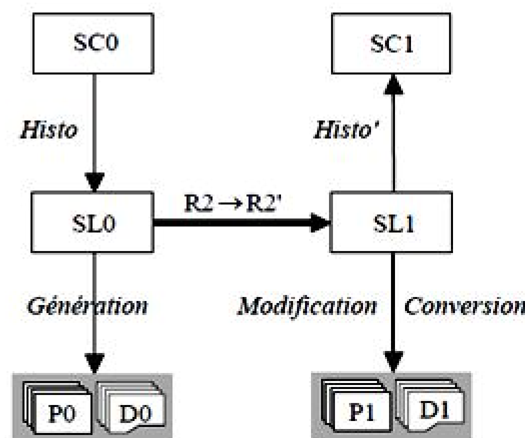


Figure IV.6 : Propagation vers l'amont et vers l'aval des modifications déduites de (R2 ⟶ R2') et apportées à SL0.

Dans un troisième scénario, SC0 et/ou SL0 manquent. Avant d'appliquer un des deux scénarios précédents, il faut retrouver les schémas logique et physique ainsi qu'un historique possible à partir de P0 et D0. Il s'agit donc d'un problème de rétro-ingénierie.

La rétro-ingénierie est une méthode qui comporte deux processus principaux, à savoir l'extraction des structures de données et la conceptualisation de ces structures. Cette division correspond à peu près à l'inverse des processus de conception physique et de conception logique habituellement admis pour l'ingénierie des bases de données.

IV.6. Conclusion :

Dans ce chapitre nous avons vue que les modifications apportées aux bases de données ou à leurs modèles ne sont pas sans risque de tomber dans une incohérence structurelle du système. Pour éviter ces états d'incohérences nous avons présenté quelques approches d'évolution de modèles qui permettent la propagation de ces modifications entre les modèles.

Dans le chapitre suivant nous allons présenter notre approche, pour assister le charger de la maintenance dans les taches d'évolutions des bases de données dans un système d'information complexe.

Chapitre V

Architecture de l'outil de maintenance

V.1. Introduction

Dans ce chapitre nous présenterons le principe de notre approche, ensuite nous donnerons quelque définitions sur l'analyse relationnelle des concepts, méthode sur laquelle repose l'approche que nous utiliserons pour le processus du refactoring entre les modèles, et qui sera détaillée après la présentation de l'architecture de l'outil de maintenance.

V.2. Principe de l'outil de maintenance

Afin de faciliter l'activité de maintenance d'un système d'information complexe composé d'un ensemble de systèmes d'information hétérogènes, notre approche présentée dans la Figure V.1, vise à déduire un modèle unique de ce système à partir de l'ensemble des modèles hétérogène qui le constitue.

A travers ce modèle unique, le chargé de la maintenance aura une vue globale sur le système à maintenir et pourra estimer dans un premier temps l'effet qu'aura une modification d'une composante du système sur l'ensemble.

L'outil de maintenance aura comme entrée l'ensemble des modèles hétérogènes des systèmes d'informations, puis par un processus de transformations des modèles, il génèrera un ensemble de modèles de classes, qui seront ensuite restructurés grâce au processus de refactoring pour au final donner une représentation de la structure globale du système.

On trouve deux étapes dans le processus de transformation :

- La première qui a pour objectif d'éliminer l'hétérogénéité des formalismes de représentations, consiste à transformer les différents modèles des systèmes d'information en des modèles de classes, par le processus d'appariement de leurs métamodèles.

- La seconde qui a pour objectif de créer le modèle unique, va appliquer le concept du refactoring sur l'ensemble de classes de ces modèles de classe. L'approche de refactoring utilisée dans cette outil est l'Analyse de Concepts Relationnelle (ARC) que nous introduisant dans la section suivante.

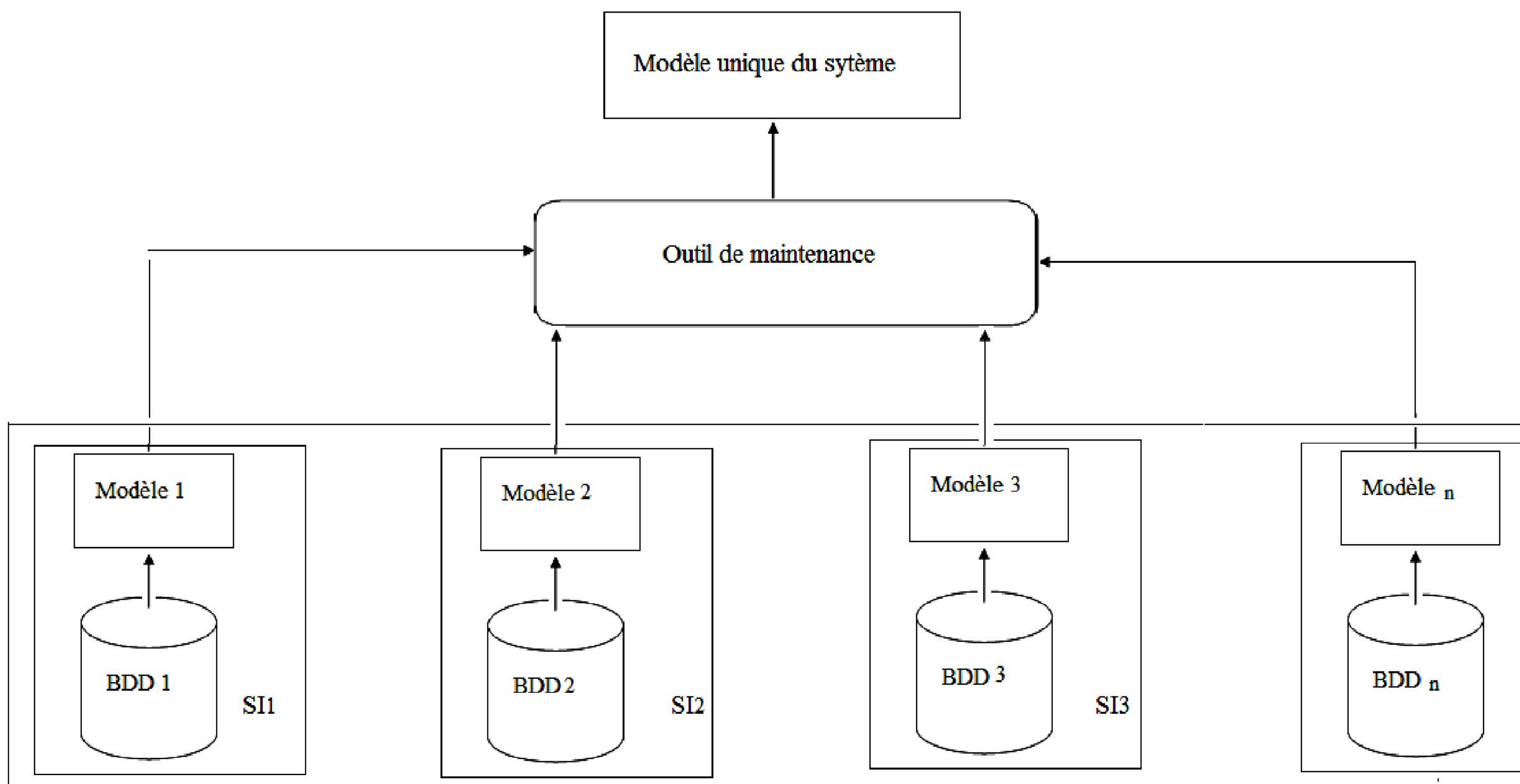


Figure V.1 : Représentation de l'approche de maintenance d'un système d'information complexe.

V.3. L'analyse de concept formelle (AFC) et l'analyse relationnelle de concept (ARC)

IV.3.1. Analyse formelle de concepts (AFC) :

L'AFC a l'avantage d'offrir un cadre théorique très approprié pour la restructuration des hiérarchies : d'une part, le treillis de concepts représente une formalisation de la structure de la hiérarchie de classes, tout en intégrant l'ensemble des abstractions qu'il est possible d'inférer à partir d'une collection de classes et d'autre part, il assure une factorisation maximale, c'est-à-dire, une absence totale de redondances, aux modèles qui en sont dérivés.

L'AFC est une approche vers la découverte et la structuration de connaissances. Elle fournit une méthode algébrique de dérivation des hiérarchies d'abstractions à partir d'un ensemble d'entités. L'AFC représente les entités par le biais d'un tableau de données (contexte formel), noté K, faisant apparaître un ensemble O d'individus (objets formels), un ensemble A de caractéristiques (attributs formels) et exprimant la relation binaire I d'incidence objet-attribut.

Dans les bases de données le contenu est d'abord modélisé sous la forme d'un contexte formel qui décrit une relation binaire entre un ensemble d'objets et un ensemble d'attributs (objet x attributs). Les objets correspondent aux classes et les attributs correspondent aux propriétés de ces classes.

La formalisation des bases est donnée par le contexte formel K_B =(C, A, I), où C est un ensemble de classes, A est un ensemble d'attribut et méthodes de ces classes et I une relation incidence entre C et A vérifiant les propriétés : $I \subseteq C \times A$ et $(c, a) \in I$ ou (cIa) où c, a sont tels que $c \in C$ et $a \in A$ signifie que la classe c est caractérisé sémantiquement par l'attribut a.

Un concept formel dans la formalisation des bases de notre collection K_B est un ensemble de classes partageant un ensemble d'attributs ou de méthodes. Un concept formel est défini comme suit.

Definition : (Concept formel). Soit $K_B = (C, A, I)$ un contexte formel. Un concept formel est un couple (X, Y) tel que $X \subseteq C$, $Y \subseteq A$. X et Y sont respectivement appelées extension et intension du concept formel (X, Y).

Le tableau illustré par la Figure V.3 montre un contexte $K = (O, A, I)$ d'une partie du diagramme UML du domaine bancaire représenté dans la Figure V.2, les classes représentent des types de comptes, soit chèque (CheckAccount) ou hypothèque (MortgageAccount), et leurs détenteurs respectifs (CheckBookHolder et Mortgager).
Dans ce contexte les individus sont les classes du diagramme UML et les attributs formels sont les propriétés (attributs et opérations) de ces classes.

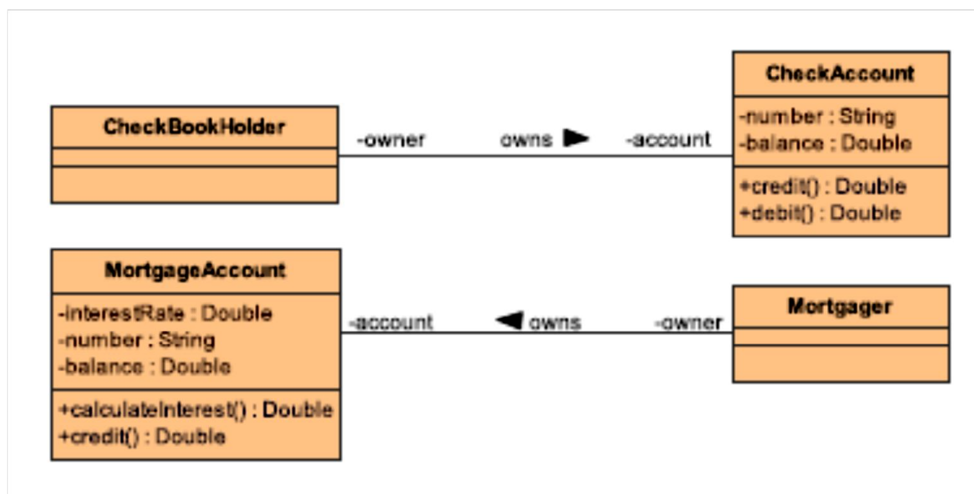
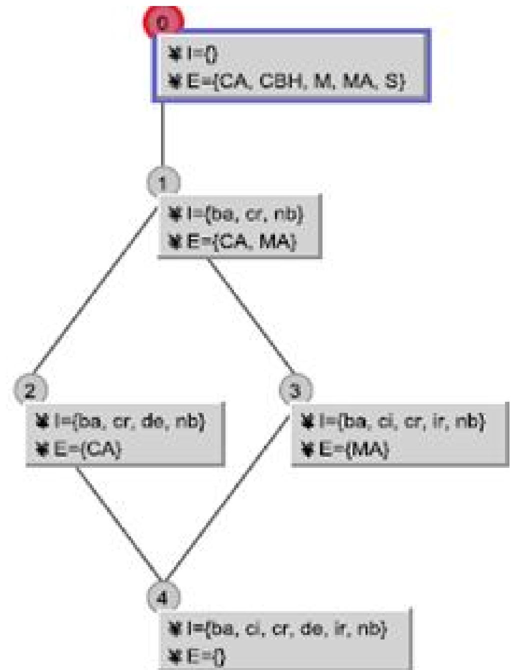


Figure V.2 : Partie d'un modèle statique du domaine bancaire.

	number (nb)	balance (ba)	interestRate (ir)	credit() (cr)	debit() (de)	calculateInterest() (ci)
CheckBookHolder (CBH)						
CheckAccount (CA)	X	X		X	X	
String (S)						
Mortgager (M)						
MortgageAccount (MA)	X	X	X	X		X

(a)



(b)

Figure V.3 : (a) codage des clases d'un modèle UML par un contexte binaire

(b) le treillis de concept correspondant

l'AFC n'étudie qu'une collection unique d'individus, même si plusieurs contextes peuvent être composés au-dessus de celle-ci comme proposé dans [Prediger et al., 99]. Cependant, pouvoir détecter des abstractions non seulement sur des classes mais aussi sur les associations, requiert une représentation de celles-ci en tant qu'individus dans un contexte. De plus, de par la nature intégrée du modèle UML, l'analyse indépendante de ces contextes n'est pas envisageable. Il se dessine donc plusieurs tâches d'analyse conceptuelle, une par sorte d'entité UML, qui doivent non seulement s'exécuter de façon simultanée, mais aussi leurs exécutions respectives doivent être synchronisées.

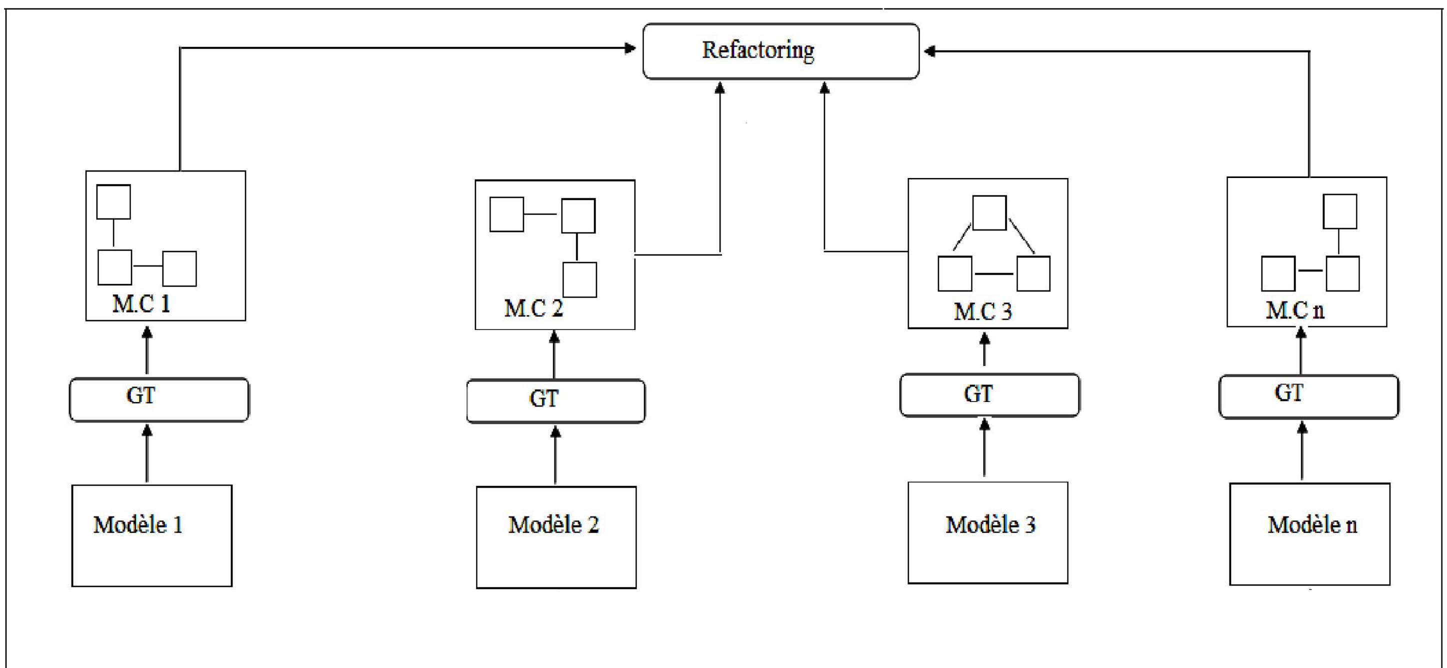
Le format de données, qui permet de réunir plusieurs sortes d'individus et d'exprimer différents rapports que ceux-ci entretiennent, est introduit par l'analyse relationnelle des concepts (ARC). Ainsi, l'analyse a pour entrée une structure appelée famille de contextes relationnels (FCR) qui est composée d'une collection de contextes standards et d'un ensemble de relations binaires inter contextes.

Une FCR est donc une paire (K , R) où K est un ensemble de contextes $K_j = (O_j , A_j , I_j)$ et $R = \{r_k\}$ un ensemble de relations $r_k \subseteq O_i \times O_j$, avec O_i (domaine de r_k) et O_j (co -domaine de r_k) des ensembles d'individus.

Une FCR a la forme d'une base de données relationnelle avec, d'une part, une partie schéma composée par les contextes avec leurs listes de caractéristiques en tant qu'entités et les relations en tant qu'associations, et d'autre part, un contenu représenté par les individus et leurs liens.

V.4. Architecture de l'outil de maintenance

La Figure V.4 représente l'architecture de l'outil de maintenance.



La Figure V.4 Architecture de l'outil de maintenance

Comme nous l'avons mentionné dans la section V.2, l'outil se base principalement sur les opérations de transformations des modèles, il comporte deux étapes: la première transforme les modèles et la seconde refactorise les modèles résultants de la première étape.

V.4.1 Le module de génération de transformation GT :

Les transformations de modèle permet au développeur d'être indépendant des environnements et des langages de développement, mais en contre partis il doit maîtriser à la fois un langage de transformation de modèles et les métamodèles source et cible de la transformation. Dans notre cas la difficulté pour réaliser une transformation réside dans la taille de la transformation (la taille des modèles source étant inconnu), ainsi que dans les métamodèles de ces modèles source (la plupart étant connu comme E/A, XML, Relationel,…..).

C'est pour faciliter la création de ces transformations que nous avons choisi l'approche génération automatique de transformations de modèles de Lopes [Lopes *et al.*, 06]. Cette approche se base sur l'alignement de métamodèles et est fortement inspirée des méthodes d'alignement de schémas. Nous pouvons dégager deux parties dans cette approche :

- Dans une première partie, un ensemble de correspondances entre les éléments du métamodèle source et du métamodèle cible sont établis grâce à l'algorithme d'alignement automatique de métamodèles SAMT4MDE. Dans notre cas nous voulons transformer tous les modèles en des modèles de classe. Pour cela, l'approche propose de comparer les éléments des métamodèles de même type et d'appliquer sur chaque couple possible une fonction mesurant la similarité entre les éléments. La fonction est définie différemment selon que les éléments à comparer sont des classes, des types primitifs ou des énumérations, mais pour chaque couple elle donne un résultat permettant d'évaluer son degré de similarité. L'ensemble des couples ayant un degré de

similarité suffisamment élevé par rapport à une borne définie formeront alors un alignement.

- Dans une seconde partie, les auteurs définissent un outil de génération de transformations à partir d'un alignement appelé MT4MDE. Cet outil est basé sur un métamodèle d'alignement permettant de représenter les différentes correspondances entre les éléments des méta-modèles. À partir d'un modèle d'alignement, l'outil génère alors une transformation de modèle en code ATL.

V.4.2. Refactoring entre les modèles

Après avoir homogénéisé l'ensemble des modèles de notre système d'information complexe, et obtenu un ensemble de modèles de classes, on passe maintenant à la deuxième étape qui est le refactoring entre ces modèles.

Pour cela on prend l'ensemble de classes des modèles auxquelles on applique l'approche ARC (Analyse Relationnelle des Concepts).

D'abord on doit :

1. Détecter les classes identiques
2. Eliminer l'hétérogénéité sémantique

Puis l'ARC qui se déroule en trois autre étapes (Figure V.5).

3. Extraire une famille de contextes relationnelle (FCR)
4. Construire une famille de treillis relationnelle (FTR)
5. Génération du modèle UML

Détecter les classes identiques et éliminer l'hétérogénéité sémantique

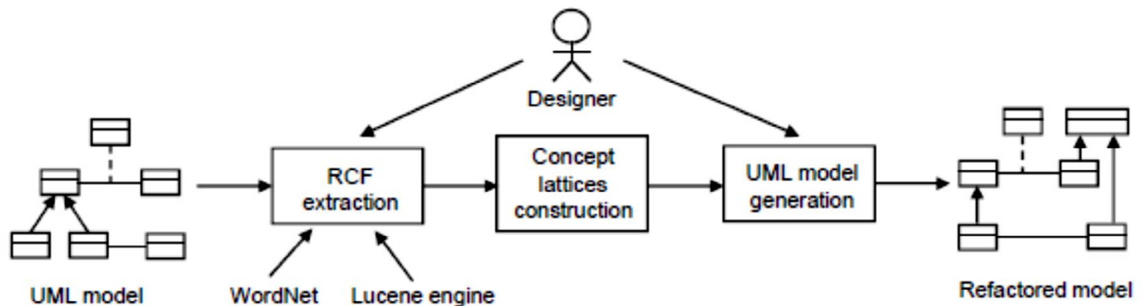
Une données peut être représenter dans plusieurs de ces modèles, elle peut être sous le même identifiant (identique) ou des synonymes, cela engendre des redondances de classe qu'il faut éliminer.

Aussi pour arriver a une abstraction optimale, il faut éliminer les problèmes causés par la polysémie du langage naturel qui est utilisé pour nommer les éléments du modèle.

Afin de détecter les conflits de noms entre les éléments UML, nous établissons une mesure de similarité entre ceux-ci et détectons les cas suspects qui sont ensuite rapportés à l'expert humain. La méthode que nous utiliserons pour le calcul de similarité entre noms suit la distance entre termes atomiques au sein de WORDNET [Fellbaum, 98]. Rappelons que pour chaque terme, WORDNET emmagasine une définition par acception du terme ainsi que les synonymes (synset) les hyponymes, les formes dérivées, etc pour chaque acception.

Dans une étape de prétraitement pour le calcul de la similarité, les données du modèle UML sont organisées en index, un par type d'élément. A cette fin, les noms symboliques composites sont soumis à une tokenisation, c'est-à-dire à un découpage en termes atomiques. Par exemple, le nom de classe CheckBook est séparé en Check et Book. Ensuite, les sens possibles des différents termes atomiques sont extraits de WORDNET. Pour check, par exemple, un des synsets est {check, assay, arrest} alors que pour book on trouve {book, record, script}. Le couple formé, d'une part, par le nom de l'élément UML et, d'autre part, par ses termes atomiques avec leurs synsets est placé dans l'index approprié. Ensuite, la similarité repose sur un appariement optimal entre les pools de termes atomiques des deux noms. L'appariement utilise à son tour une distance d'arbre entre les synsets qui est mesurée au sein des hiérarchies disponibles dans WORDNET. Par exemple, étant donnés deux noms du modèle invoiceAmount et customerOrderSum, l'appariement retenu en se

basant sur les distances WORDNET entre les termes composant ces deux noms est représenté par les deux paires (amount,sum) et (invoice,order).



Figue V.5 : Processus de refactoring de modèle de classe UML en utilisant l'ARC [Rouane,07]

Transformation d'un modèle UML vers une FCR : Plusieurs contextes seront construits à partir des éléments des modèles de classe sur lesquelles reposerait notre refactoring. Ces éléments sont les classes, les associations et les attributs.

Les incidences entre éléments au sein du modèle sont exprimées au travers de méta-associations. Dans la FCR celles-ci se traduisent en une collection de relations, appelées techniques car leur sémantique est indépendante du domaine modélisé. La distinction des éléments techniques de ceux ayant trait au domaine aide la reconstitution du modèle restructuré à partir des Processus de restructuration de modèles UML en utilisant l'ARC treillis finaux en indiquant les abstractions qui doivent apparaître explicitement dans celui-ci. De leur côté, les relations techniques jouent un rôle de guide pour l'abstraction sur contextes multiples : elles permettent à la découverte d'une abstraction sur son contexte d'être répercutée sur les contextes adjacents. De plus, elles permettent d'assembler le modèle définitif à partir des treillis [Rouane,07].

Construction d'une FTR :

Une FTR est dérivée à partir de la FCR. Nous construisons à présent autant de treillis que de contextes de la FCR, lesquelles représente toutes les abstractions potentielles des éléments du modèle.

La génération du modèle UML à partir d'une FTR : La méthode est basée sur un processus exploratoire et semi-automatique faisant intervenir des outils de navigation de treillis et le concepteur pour préciser des choix de modélisation selon ses objectifs. La méthode se déroule en trois étapes :

1) Parmi les treillis de la FTR on choisi celui dont les concepts sont classificateurs qui correspondent aux classes des modèles. Ces concepts seront traduits en classes du modèle globale.

2) Les concepts des treillis seront parcourus pour détecter des éventuelles abstractions significatives. Pour cela on cherche parmi les successeurs directs d'un concept classificateur les autres concepts pertinents.

Un concept pertinent est un concept dont l'intension contient des attributs formels tels que nom=..., type=..., etc., exprimant une information du domaine du modèle, ou une référence relationnelle vers un autre concept d'un autre treillis de la FTR, construit sur un autre type d'élément du modèle, qui a été reconnu comme pertinent auparavant.
Cette définition récursive de la pertinence pose un problème de circularité dans son calcul au niveau des concepts inter-reliés dont les intensions n'indiquent aucune information du domaine.

L'algorithme 1 de Rouane permet de déterminer si un concept est pertinent, il s'appuy sur la routine DOMAIN qui vérifie si un attribut formel donné désigne une information du domaine (ligne 10), et sur la structure de données Path pour détecter la présence d'un même concept plus d'une fois sur un même chemin de calcul de pertinence, afin de surveiller la circularité au niveau des concepts inter-reliés.

L'algorithme commence par le considérer un concept **c** comme étant non pertinent (ligne 4).

Ensuite, il vérifie dans Path si ce concept a été déjà visité, si oui, l'algorithme s'arrête en retournant une valeur indiquant l'impossibilité d'évaluer sa pertinence (lignes 5 et 6) , sinon le concept est rajouté à Path (ligne 8).

Ensuite, l'algorithme parcourt tous les attributs de l'intension du concept **c** (ligne 9) et pour chaque attribut vérifie s'il appartient aux attributs locaux, notés Intl(**c**) (ligne 10), ensuite s'il s'agit d'une information domaine (ligne 11). Si c'est le cas, l'algorithme s'arrête (ligne 12) en considérant le concept comme étant pertinent par rapport au domaine. Les attributs qui ne sont pas du domaine mais qui sont de type relationnel, notés Intr(**c**), subissent une évaluation de pertinence qui affecte la pertinence du concept courant **c** (ligne 14). En effet, si un concept indiqué par un attribut relationnel appartenant à Intr(**c**) est pertinent, le concept **c** est jugé pertinent par transitivité et il n'est plus utile d'examiner les autres attributs de Intr(**c**) non visités (ligne 16).

Les concepts dont l'évaluation de la pertinence est confrontée au problème de la circularité des relations sont soumis au jugement du concepteur. [Rouane,07]

```
1: Algorithm RELEVANT
2: (In : RLF a family of lattices, c a concept, Path chain of concepts) : [1..3]
3: Local  : relevant an integer variable
4: relevant ← 2  {by default classifier-concept is set to be irrelevant}
5: if c ∈ Path then
6:   return 3 {relevancy can't be determinated because of cycle in concept relations}
7: else
8:   Path ← Path ∪ {c}
9: for all a ∈ Int(c) do
10:   if a ∈ Int_l(c) then
11:     if DOMAIN(a) then
12:       return 1    {relevant classifier-concept}
13:   else
14:     relevant ← RELEVANT(RLF,cod(a),Path)    {a ∈ Int_r(c)}
15:     if relevant=1 then
16:       return 1    {relevant classifier-concept}
17: return relevant
```

Algorithme 1 : Détermination de la pertinence d'un concept

3) génération du modèle UML par la traduction en éléments UML des concepts classificateurs retenus et des concepts des autres treillis qu'ils désignent. Ainsi, les concepts-classificateurs retenus sont traduits en classes UML, les liens d'héritage sont déduits de la relation d'ordre entre les concepts-classificateurs et l'intension de chaque concept-classificateur retenu est traduite en éléments de modélisation tels que attributs, rôles, opérations, etc.

L'algorithme 2 decrit par Rouane exprime les diverses étapes de la génération du modèle UML restructuré [Rouane,07]. Un stockage séparé est utilisé pour les divers types de concepts classificateurs, à savoir, les concepts à examiner (Candidates), ceux déjà examinés et libellés pertinents (Relevant),

les non pertinents (Dummy) et finalement ceux dont la pertinence n’a pu être établie (Unknown). L’algorithme s’appuie sur les routines RELEVANT, INTERPRET et UPDATE qui permettent de calculer la pertinence d’un concept, de traduire en élément UML un concept donné et de mettre à jour le modèle UML final, respectivement. A noter que **c** désigne l’ensemble des successeurs du concept **c** dans le treillis. [Rouane,07].

1: **Algorithm** FTR2UML(**In :** RLF a family of lattices, **Out :** M an UML model)
2: Local : Candidates - a set of classifier-concepts to explore
3: Local : Relevant - a set of classifier-concepts to interpret as UML classes
4: Local : Dummy - a set of irrelevant classifier-concepts
5: Local : Unknown - a set of concepts whose relevancy can't be determinated
6:
7: Candidates $\leftarrow$ COMPUTE-STARTING-CONCEPTS($\mathcal{L}^{\infty}_{classifiers}$)
8: Relevant $\leftarrow \emptyset$, Dummy $\leftarrow \emptyset$, Unknown $\leftarrow \emptyset$
9: **for all** $c \in$ Candidates **do**
10: **switch**(RELEVANT(RLF,c,$\emptyset$))
11: **case 1 :** {relevant classifier-concept}
12: Relevant $\leftarrow$ Relevant $\cup \{c\}$
13: Candidates $\leftarrow$ Candidates $\cup\ Cov^{u}(c) \setminus$ Dummy
14: **case 2 :** {irrelevant classifier-concept}
15: Dummy $\leftarrow$ Dummy $\cup \uparrow c$
16: **case 3 :** {relevancy can't be determinated}
17: Unknown $\leftarrow$ Unknown $\cup \{c\}$
18: USER-ASSESSMENT(Unknown,Relevant)
19: **for all** $c \in$ Relevant **do**
20: UPDATE(M,INTERPRET(FTR,c))
21: **return** M

Algorithme 2 : Génération d’un modèle structuré à partir d’une FTR

Noms des éléments du modèle restructuré : La tâche de synthèse de noms est complexe car elle revient à vouloir deviner la sémantique d’un élément UML. Toutefois, un certain nombre de techniques ont été explorées. Par exemple, les éléments ayant un homologue dans le modèle de départ pourront se voir affecter le nom correspondant (ou une combinaison de noms s’il s’agit d’une fusion d’éléments nommés différemment), Pour le reste, c’est-à-dire les abstractions nouvelles, les pistes explorées incluent la composition des noms

des entités nommées ayant mené à la création de l'entité abstraite. La composition pourrait prendre la forme d'un préfixe (langues latines) ou un suffixe (langues germaniques) commun à toutes les entités nommées.

De façon plus prudente, une concaténation de tous les noms peut se substituer aux préfixe/suffixe communs. L'alternative est la recherche d'un terme, éventuellement composite, généralisant les noms des éléments en question. Celui-ci pourrait être recherché au sein des structures hiérarchiques de WORDNET (EUROWORDNET pour les langues autres que l'anglais), ou tout autre thesaurus électronique. [Rouane,07].

V.5. Résultat :

Le modèle globale du système d'information complexe, obtenue après l'application de notre approche est décrit dans la Figure V.6.

Les niveaux d'abstraction diffèrent selon le nombre de classes d'entrée, ainsi pour n classes on obtient N niveaux.

Les niveaux inférieures contiennent les attributs propre à chaque bases de données, plus on monte en abstraction, les niveaux contiendront des classes abstraites (classe ab) avec les attributs communs à deux, ou trois … classes jusqu'à l'arrivée au dernier niveau avec des classes abstraite qui regroupent le nombre maximum d'attributs communs. Si un attribut est commun à toutes les classes alors au dernier niveau, on aura une seul classe abstraite.

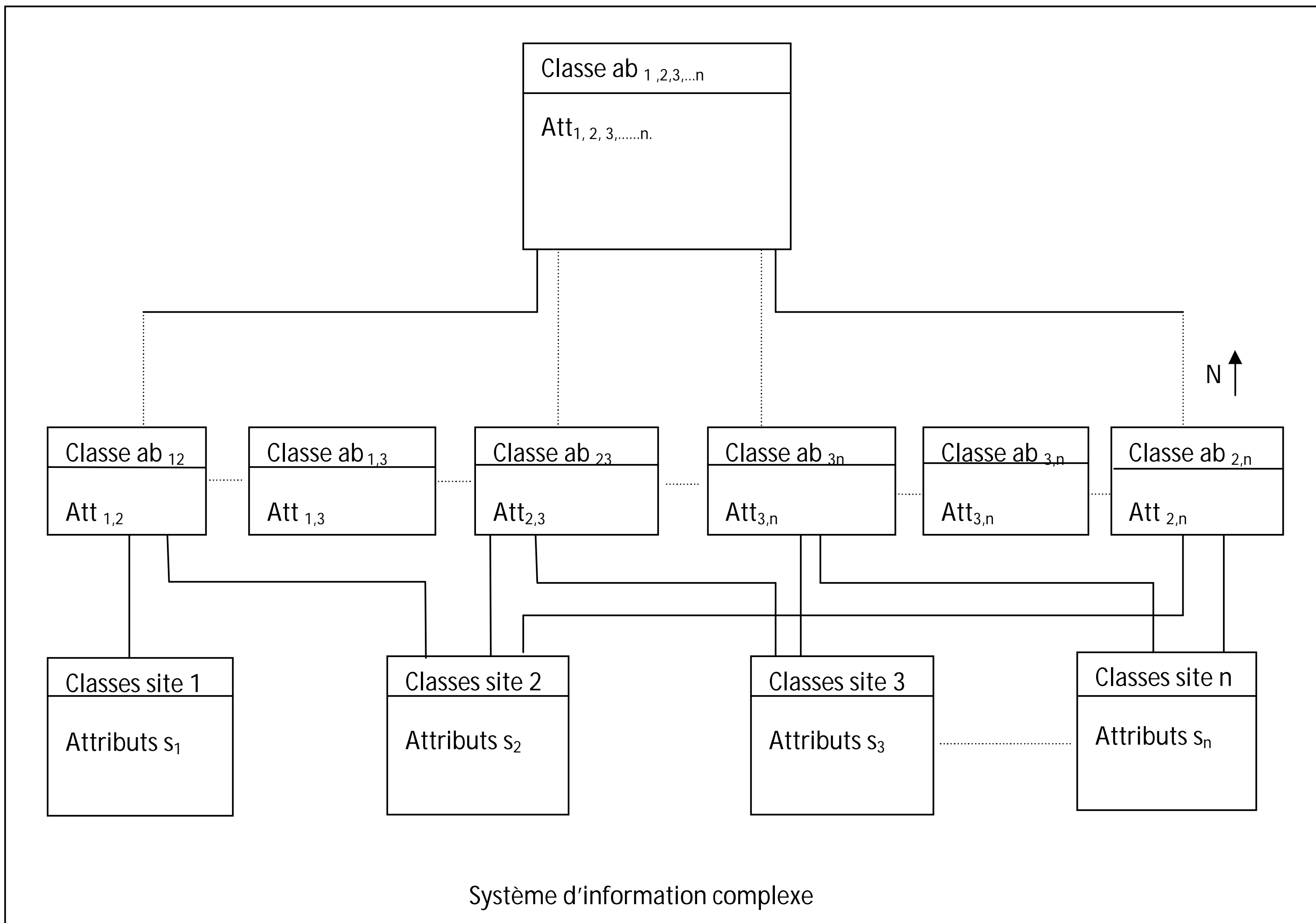


Figure V.6 : Modèle globale du système d'information complexe

V.6. Conclusion

Dans ce dernier chapitre nous avons présentée l'architecture de notre outil de maintenance des applications puis nous avons détaillé l'approche utilisée pour le processus de factorisation.

Dans ce qui suit nous finalisant notre mémoire par une conclusion générale et quelques perspectives.

Conclusion générale

Conclusion générale

Suite à une étude approfondie sur l'ensemble des modèles de représentation des données et les problèmes de maintenance dans les applications informatiques, nous avons déduits que les méthodes introduites par l'ingénierie dirigée par les modèles, à savoir les transformations des modèles et leur refactorisation, sont les plus appropriées pour maintenir les bases de données hétérogènes.

Disposer d'un modèle de classes d'un bon niveau d'abstraction est très important pour disposer d'une vision globale et claire du système modélisé afin de faciliter sa compréhension. En effet, la simplicité de compréhension et la facilité de réutilisation du modèle dépendent en grande partie de la qualité de sa hiérarchie des classes [Booch et al., 07]. Une bonne hiérarchie de classes factorise le maximum d'attributs et de méthodes dans un nombre minimal de classes. La construction d'une telle hiérarchie est une tâche difficile surtout si le nombre de classes composant le système est élevé. En effet, pour un système comportant n classes, il faut examiner les attributs et méthodes communs de tous les 2^n sous-ensembles possibles de classes.

Dans le cadre de notre proposition, notamment l'architecture de l'outil de maintenance, en voulant éliminer l'hétérogénéité des modèles de représentation des données par l'utilisation des composants qui génèrent des transformations automatiques vers des modèles de classe, nous retrouvons la difficulté pour obtenir un modèle général de classes de bonne hiérarchie . Pour palier à cette difficulté nous avons adopté la méthode de refactoring basée sur l'analyse relationnelle des concepts (ARC). Cette méthode d'analyse de données offre une solution exacte et unique pour la suppression des doublons.

Perspectives :

Comme perspectives de ce travail, nous prévoyons d'implémenter notre outil dans un environnement de transformation des modèles, et de la tester sur un système d'information complexe expérimental en premier. Suite à ces testes nous apporteront les modifications nécessaires, pour pouvoir le tester sur un système d'information complexe de taille réelle.

Références bibliographiques

[Abedmouleh, 13] Aymen ABEDMOULEH. Approche *Domain-Specific Modeling* pour l'opérationnalisation des scénarios pédagogiques sur les plateformes de formation à distance. Thèse Doctorat. Université du Maine le 11 juillet 2013.

[Ahmad, 11] Adeel Ahmad. Contribution à la Multi-modélisation des Applications Distribuées pour le Contrôle de l'Évolution des Logiciels. Thèse de doctorat. Université du Littoral Côte d'Opale.2011.

[Allilaire et al. ,06] Freddy Allilaire, Fredéric Jouault , Jean Bézévin, Ivan Kurtev et Patrick Valduriez. ATL: a QVT-like Transformation Language. OOPSLA'06, Portland, Oregon, USA, 2006,

[AZAIEZ, 07] Selma AZAIEZ. Approche dirigée par les modèles pour le développement de systèmes multi-agents. Thèse de Doctorat. Université De Savoie le 11 décembre 2007.

[Benatallah, 96] Boualem Benatallah. Object-Oriented Database Schema Evolution: A compromise. Modeling and Simulation. Thèse de Doctorat. Université Joseph-Fourier - Grenoble I, le 4 mars 1996.

[Benelellam et al. , 14] Amine Benelellam et Massimo Tisi et István Ráth et Benedek Izsó et Dimitrios S. Kolovos. Towards an Open Set of Real-World Benchmarks for Model Queries and Transformations. BigMDE (2014).

[Bex et al., 02] Geert Jan Bex, Sebastian Maneth et Frank Neven. A formal model for an expressive fragment of XSLT . *Information System,* pages 21-39. 2002.

[BEZIVIN et al. ,99] Jean Bézivin, Jean-Paul Bouchet et Erwan Breton. Correspondance structurelle entre produits et procédés : un pattern classique, analysé avec des méta-modèles explicites. Journée GRACQ 7 juin 1999.

[Bézivin et al. ,01] Jean Bézivin et Nicolas Ploquin. Tooling the MDA framework: a new software maintenance and evolution scheme proposal. JOOP 2001.

[Bézivin, 02] Jean Bézivin ET Xavier Blanc. PROMESSES ET INTERROGATIONS DE L'APPROCHE MDA. Développeur Référence, Septembre 2002.

[Bézivin et al. ,04] Rapport de Synthèse de l'AS CNRS sur le MDA (Model Driven Architecture).2004.

[Bézivin et al., 05] Jean Bézivin, Frédéric Jouault, Jean Paliès. Towards Model Transformation Design Patterns.2005.
[Biehl et al. , 10] Matthias Biehl, Chen DeJiu et Martin Törngren. Integrating safety analysis into the model-based development toolchain of automotive embedded systems. Conference on Languages, compilers, and tools for embedded systems, Stockholm, Sweden. page125-132. 2010.
[Bouarioua, 13] Mouna Bouarioua. Une approche basée transformation de graphes pour la génération de modèles de réseaux de Petri analysables à partir de diagrammes UML. Thèse de doctorat. Université Constantine 2. 2013.

[Boukhari, 14] Ilyès Boukhari. Intégration et exploitation de besoins en entreprise étendue fondées sur la sémantique. Thèse de Doctorat. Ecole Nationale Supérieure de Mécanique et d'Aérotechnique, le 14 janvier 2014.

[Boulbaba et al., 08] Boulbaba Ben Ammar, Mohamed Tahar Bhiri et Jeanine Souquières. Schéma de refactoring de diagrammes de classes basé sur la notion de délégation. Atelier ERTSI, pages 1 à 12. 2008.

[Bounif, 09] Hassina Bounif . Predictive Approach for Schema Evolution. Thèse de Doctorat. École Polytechnique Fédérale de Lausanne, le 6 février 2009.

[Bounif et al., 09] Hassina Bounif et Rachel Pottinger. Schema Repository for Database Schema Evolution. 2009.

[Bourett, 03] Ronald Bourett "XML et les bases de données", http://www.rpbourret.com , 2003.

[Brant et al. , 99] John Brant et Don Roberts. Refactoring techniques and tools (Plenary talk). In *Smalltalk Solutions*, New York, 1999.

[BRETON, 01] Erwan Breton et Jean Bézivin. Un méta-modèle de gestion par les activités : Définition et intégration.2001.

[BRETON, 02] Erwan BRETON. Contribution à la représentation de processus par des techniques de méta-modélisation. Thèse de doctorat université de Nante, 2002.

[Büttner et al. 12] Fabian Büttner et Marina Egea et Jordi Cabot et Martin Gogolla.Verification of ATL Transformations Using Transformation Models and Model Finders. ICFEM 2012: 14th International Conference on Formal Engineering Methods (2012).

[Casais, 92] E. Casais. An incremental class reorganization approach. European Conference on Object-Oriented, Springer, 1992.

[Cavarero et al., 13] A. Authosserre-Cavarero, F. Bertrand, M. Blay-Fornarino, P. Collet, H. Dubois. Ingénierie dirigée par les modèles : quels supports à l'interopérabilité des systèmes d'information ?. Revue Ingenierie Des Systemes D'information (2013)".

[Chen et al., 96] J B Chen, Lee. Generation and reorganization of subtype hierarchies, Journal of Object Oriented Programming, 1996.

[Chettaoui, 08] Hanène Chettaoui . Interopérabilité entre modèles hétérogènes en conception coopérative par des approches d'Ingénierie Dirigée par les Modèles. Thèse de Doctorat. Institut Polytechnique de Grenoble le 25 Novembre 2008.

[Chikofsky et al. ,90] E. J. Chikofsky et J. H. Cross, Reverse engineering and design recovery - a taxonomy . *IEEE Software*, pages 13–17, 1990.

[Choquet et al., 10] Rémy Choquet, Douglas Teodoro, Giovanni Mels, Ariane Assele, Emilie Pasche, Patrick Ruch, Christian Lovis et Marie-Christine Jaulent. Partage de données biomédicales sur le web sémantique. WSM 2010.

[Choquet, 11] Rémy Choquet. Partage de données biomédicales : modèles, sémantique et qualité. Thèse de Doctorat. Universite Pierre Et Marie Curie, le 16 Décembre 2011.

[Choppy Heisel,06] Christin Chopp et Maritta Heisel. Une approche à base de patrons pour la spécification et le développement de systèmes d'information. Rapport technique LIPN-2006-05, Université Paris XIII, France, 2006.

[Cicchetti et al. ,08] A. Cicchetti, D. Di Ruscio, R. Eramo, et A. Pierantonio. Automating co-evolution in model-driven engineering. *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*, pages 222–231. 2008.

[Clasen et al. , 12] Cauê Clasen, Marcos Didon, Del Fabro et Massimo Tisi. Transforming Very Large Models in the Cloud: a Research Roadmap. First International Workshop on Model-Driven Engineering on and for the Cloud (2012).

[Clavreul, 11] Mickaël Clavreul.Composition de modèles et de métamodèles : Séparation des correspondances et des interprétations pour unifier les approches de composition existantes.Thèse de Doctorat. Univérsité de Rennes 1, le 7 décembre 2011.

[Combemale, 08] Benoît Combemale. Ingénierie Dirigée par les Modèles (IDM) État de l'art. Institut de Recherche en Informatique de Toulouse (UMR CNRS 5505), 2008.

[Combemale, 08 /01] Benoît Combemale. Approche de méta modélisation pour la simulation et la vérification de modèle Application à l'ingénierie des procédés. Thèse de Doctorat. Université De Toulouse, le 11 juillet 2008.

[Cook, 92] W. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes, OOPSLA pages. 1–15, 1992.

[Cudennec, 05] Loïc Cudennec. Gestion de la cohérence des données dans les systèmes distribués Étude bibliographique. Rapport de recherche INSA, IFSIC, IRISA, Projet Paris, Février 2005.

[Czarnecki, 03] Krzysztof Czarnecki, et Simon Helsen. Classification of Model Transformation Approaches. *OOPSLA,* 2003.

[Dao et al. , 04] Michel Dao, Marianne Huchard, M. Rouane Hacène, C. Roume, et Petro Valtchev. Improving Generalization Level in UML Models Iterative Cross Generalization in Practice. 12th International Conference on Conceptual Structures, ICCS 2004,Huntsville, AL, USA, July 19-23, pages :348-360 , 2004.

[DEL FABRO et al. 07] Marcos Didonet DEL FABRO et Patrick VALDURIEZ : Semi-automatic model integration using matching transformation and weaving models. SAC'07, Seoul, Korea, March 11-15, 2007.

[Diaw et al. , 08] Samba Diaw, Rédouane Lbath et Bernard Coulette. Etat de l'art sur le développement logiciel dirigé par les modèles. TSI -X/2008. L'ingénierie dirigée par les modèles. 2008

[Djebbi, 04] Olfa Djebbi et MP Gervais. MDA : Vers l'industrialisation de construction d'applications réparties. Rapport de recherche LIP6, 2004.

[DO et al., 02] Hong-Hai DO, Sergey MELNIK et Erhard RAHM : Comparison of schema matching evaluations. *In Web, Web-Services, and Database Systems*, pages 221–237, 2002.

[Dodinet et al ., 09] Gilles Dodinet, Michel Zam et Geneviève Jomier. Evolutivité des applications Web : un métamodèle exécutable : Prototypage incrémental d'applications Web. Actes du XXVII° congrès INFORSID, Toulouse, mai 2009

[Dolques et al., 09] Xavier Dolques, Marianne Huchard et Clémentine Nebut. Génération de transformation de modèles par application de l'ARC sur des exemples.2009.

[Dumez, 10] Christophe Dumez. Approche dirigée par les modèles pour la spécification, la vérification formelle et la mise en œuvre des services Web composés. Thèse de Doctorat. Université de Technologie de Belfort-Montbéliard, le 31août 2010.

[Einarsson, 11] Hafsteinn Þór Einarsson. Refactoring Uml Diagrams And Models with Model-To-Model Transformations. Thèse de Magister scientifique. School of Engineering and Natural Sciences University of Iceland. 2011.

[Englebert, 95] V. Englebert, J. Henrard, J.-M. Hick, D. Roland, et J.-L. Hainaut. DB-MAIN : un atelier d'ingénierie de bases de données.1995

[Euzenat, 00] Jérôme Euzenat. XML est-il le langage de représentation de connaissance de l'an 2000. Actes 6e journées « langages et modèles à objets », Mont Saint-Hilaire (CA), pp59-74, 2000.

[FALLERI ET AL., 08] Jean-Rémy FALLERI, Marianne HUCHARD, Mathieu LAFOURCADE et Clémentine NEBUT : Meta-modelMatching for AutomaticModel Transformation Generation. *InMODELS'08,* Springer, 2008.

[Favre, 95] Jean-Marie Favre, Une approche pour la maintenance et la ré-ingénierie globale des logiciels. Thèse de doctorat. Université Joseph Fourier - Grenoble I. 1995.

[Fellbaum, 98] C. FELLBAUM. Wordnet : An Electronic Lexical Database, MIT Press, 1998.

[Fowler, 99] M. Fowler. *Refactoring: Improving the Design of Existing Programs.* Addison-Wesley, 1999.

[France et al.] Robert B. France, Eunjee Song, DaeKyoo Kim et Sudipto Ghosh. Using Roles for PatternBased Model Refactoring. Colorado State University, Fort Collins, CO 80523, USA.

[France et al. 03] Robert France, Sudipto Ghosh, Eunjee Song et Dae-Kyoo Kim. A Metamodeling Approach to Pattern-Based Model Refactoring. IEEE SOFTWARE. 2003.

[FREIRE JUNIOR, 97] José Celso FREIRE JUNIOR ; thèse de doctorat : Une approche de Multi-Modélisation et de Méta-Modélisation, 1997.

[Gerbé, 00] Olivier Gerbé. Un modèle uniforme pour la modélisation et la méta modélisation d'une mémoire d'entreprise. Thèse pour l'obtention du grade de Philosophiae Doctor (Ph.D.) en informatique. Université de Montréal. 2000.

[Gerbé et al. ,01] Olivier Gerbé et Jean Bézivin. Towards a Precise Definition of the OMG/MDA Framework. IEEE 2001.

[GISI, 11] Groupe Ingénierie Système et Intégration. Transformation des diagrammes d'activités SysML1.2 vers les réseaux de Petri dans un cadre MDE. Etude Université Paul Sabatier - Laboratoire d'Analyse et d'Architecture des Systèmes, 2011.

[Jézéquel et al., 06] Jean-Marc Jézéquel, Sébastien Gérard et Benoit Baudry. Le génie logiciel et l'IDM : une approche unificatrice par les modèles. J. Estublier. L'ingénierie dirigée par les modèles, Lavoisier, Hermes-science, 2006. <inria-00512541>.2006.

[Jouault et al., 06] Frédéric Jouault et Ivan Kurtev. Transforming Models with ATL. J.-M. Bruel (Ed.): MoDELS 2005 Workshops, LNCS 3844, pp. 128 – 138, 2006. Springer Berlin Heidelberg, 2006.

[KAPPEL et al., 06] Gerti KAPPEL, Elisabeth KAPSAMMER, Horst KARGL, Gerhard KRAMLER, Thomas REITER,Werner RETSCHITZEGGER, Wieland SCHWINGER et Manuel WIMMER. Lifting metamodels to ontologies : A step to the semantic integration of modeling languages. *Proceedings of MoDELS 2006*, pages 528–542, 2006.

[Kim, 08] Dae-Kyoo Kim. Software Quality Improvement via Pattern-Based Model Refactoring.11th IEEE High Assurance Systems Engineering Symposium.2008

[Laforcade, 05] Pierre Laforcade. Approche par transformation de modèles pour la conception d'EIAH. 2005.<hal-00005664>

[Lahoud, 13] Inaya Lahoud. Un système multi-agents pour la gestion des connaissances hétérogènes et distribuées. Thèse de Doctorat. Université de téchnologie Belfort-Montbéliard, 2013.

[Lajmi, 10] Amine Lajmi. Usine logicielle de composants de simulation de procédés CAPE-OPEN. Thèse de Doctorat. Université Pierre & Marie Curie. 2010.

[Landaisa et al., 02] Paul Landaisa, Ana Simonetb, Didier Guillonb, Christian Jacquelinetc, Mohamed Ben Saïda, Claude Mugniera, Michel Simonetb. SIMS@REIN : un système d'information multi-sources pour l'insuffisance rénale terminale. *C. R. Biologies 325 (2002) 515–528.* © 2002 Académie des sciences / Éditions scientifiques et médicales Elsevier SAS

[LEMESLE, 00] LEMESLE Richard. Techniques de Modélisation et de Méta-modélisation. Thèse de doctorat. Université de nantes, 2000.

[Leriche, 06] Sébastien Leriche. Architectures à composants et agents pour la conception d'applications réparties adaptables. Thèse pour obtenir le grade de Docteur. Université Toulouse III - Paul Sabatier.2006.

[Lieberherr et al. ,90] Lieberherr K., Bergstein P. et I Lepe S. . Abstraction of Object-Oriented Data Models.1990.

[Lieberherr et al. ,91] Lieberherr K., Bergstein P., Incremental Class Dictionnary Learning and Optimization , ECOOP'91, 1991, p. 371–396.

[Lopes et al., 06] Denivaldo LOPES, Slimane HAMMOUDI et Zair ABDELOUAHAB : Schema matching in the context of model driven engineering : From theory to practice. Springer, 2006.

[Lopes et al., 09] Valéry Lopes, Éric Leclercq, Marie-Noëlle Terrasse. Contrôle des évolutions d'un système d'information pour la recherche clinique. Actes du XXVII° congrès INFORSID, Toulouse, mai 2009.

[Lounis et al.] H. Lounis et M.K Abdi. Analyse de l'impact du changement : approche et étude de cas.

[Maddeh et al., 08] Maddeh Mohamed, Mohamed Romdhani, Khaled Ghedira. Classification des approches de refactorisation des modèles. Mulhouse IDM 5-6 juin 2008.

[Mallet et al. ,00] Frédéric Mallet , Charles André et François Lagarde. Un profil UML pour la modélisation multiniveau. Rapport de recherche. ISRN INRIA/RR --7287 – FR+ENG. 2000.

[Markovic,] Slavisa Markovic. Composition of UML Described Refactoring Rules. Swiss National Scientific Research , the reference number 2000-067917.

[Marticorena et al., 06] Raul Marticorena, Carlos Lopez, and Yania Crespo, Extending a Taxonomy of Bad Code Smells with Metrics, *WOOR,* Nantes, 2006.

[Menet, 10] Ludovic MENET. Formalisation d'une approche d'Ingénierie Dirigée par les Modèles appliquée au domaine de la Gestion des Données de Référence. Thèse de Doctorat. Université Paris VIII, le 24 juin 2010.

[Miralles et al., 14] André Miralles, Xavier Dolques, Marianne Huchard, Florence Le Ber, Thérèse Libourel , Clémentine Nebut et Abdoulkhader Osman-Guédi. Exploration de la factorisation d'un modèle de classes sous contrôle des acteurs.2014

[Misbhauddin et al., 13] Mohammed Misbhauddin et Mohammad Alshayeb. UML Model refactoring: a systematic literature review . Springer Science+Business Media New York 2013

[Moalla et al., 12] Néjib Moalla, Hervé Panetto et Xavier Boucher. Interopérabilité et partage de connaissances. Ingénierie des Systèmes d'Information (ISI) , pages 7-17. 2012.

[Moha et al., 09] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. Generic Model Refactorings. Springer, 2009.

[Moore et al., 96] I. Moore, T. Clement. A simple and efficient algorithm for inferring inheritance hierarchies.1996.

[MOTTU, 08] Jean-Marie MOTTU. Oracles et qualification du test de transformations de modèles .Thèse de doctorat. Université De Rennes 1. 2008.

[Panetto et al., 04] Hervé Panetto, Giuseppe Berio, Khalid Benali, Nacer Boudjlida Et Michaël Petit A Unified Enterprise Modelling Language For Enhanced Interoperability Of Enterprise Models. *Proceedings of the 11th IFAC INCOM2004 Symposium*, April 5th-7th, Bahia, Brazil. 2004.

[Pereira et al., 04] Claudia Pereira, Liliana Favre et Liliana Martinez. Refactoring UML Class Diagram. Idea Group Inc 2004. IRMA International Conference, Innovations Through Information Technology pp.506-510. 2004.

[Pollet, 05] Damien Pollet. Une architecture pour les transformations de modèles et la restructuration de modèles UML. Thèse de doctorat. Université de Rennes 1. 2005.

[Pontisso, 09] Nadège Pontisso. Association cohérente de données dans les systèmes temps réel à base de composants - Application aux logiciels spatiaux. Thèse de Doctorat. Institut National Polytechnique de Toulouse, le 16 décembre 2009.

[Rahm et al., 01]. E. RAHM. et P. BERNSTEIN. A survey of approaches to automatic schema matching. VLDB Journal : Very Large Data Bases, pages. 334–350.2001.

[RIEU, 99] Dominique RIEU. Ingénierie des Systèmes d'Information : Bases de Données, Bases de Connaissances et Méthodes de Conception. Mémoire d'habilitation à diriger les recherches. Institut National Polytechnique de Grenoble. 1999.

[Rivera et al. , 08] J.E. Rivera and A. Vallecillo. Representing and operating with model differences. *Objects, Components, Models and Patterns*, pages 141–160, 2008.

[Robbins ,98] Jason E. Robbins . Design Critiquing Systems,1998.

[Rouane et al., 07] Mohamed Hacene Rouane, Michel Dao, Marianne Huchard, Petko Valtchev. Analyse formelle de données relationnelles pour la réingénierie des modèles UML. Isabelle Borne, Xavier Crégut, Sophie Ebersold, Frédéric Migeon. LMO'07 : Langages et Modèles à Objets, Mar 2007, Toulouse, France, France. Hermés Lavoisier, pp.151-166. 2007.

[Sans, 08] Verginie SANS, Maintenance de vues XML matérialisées à partir de sources web non coopérantes. Thèse de Doctorat. Univérsité de cergy-Pontoise, le 5 decembre 2008.

[Sen, 10] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry et Jean-Marc Jézéquel**.** Reusable model transformations. Springer-Verlag 2010.

[SEN et al. 12] Sagar Sen et Jean-Marie Mottu et Massimo Tisi et Jordi Cabot. Using Models of Partial Knowledge to Test Model Transformations. International Conference on Model Transformation (2012),2012.

[Schätz et al., 08] Bernhard Schätz, Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Stefan Wagner. Clone detection in automotive model-based development. ICSE '08 Proceedings of the 30th international conference on Software engineering, Pages 603-612. 2008.

[Sheeren, 05] David Sheeren. Méthodologie d'évaluation de la cohérence inter-représentations pour l'intégration de bases de données spatiales. Une approche combinant l'utilisation de méta données et l'apprentissage automatique. Thèse de doctorat. Université Paris 6, 2005.

[Staken ,01]KimbroStaken. http://www.xml.com/pub/a/2001/10/31/nativexmldb.html. 2001.

[Störrle, 07] Harald Störrle. A PROLOG-based Approach to Representing and Querying Software EngineeringModels. Visual Languages And Logic. Coeur d'Alene, Idaho, USA ,September 23rd, 2007.

[Sunyé et al., 01] Gerson Sunyé, Damien Pollet, Yves Le Traon et Jean-Marc Jézéquel. Refactoring UML Models. M. Gogolla and C. Kobryn (Eds.): UML 2001, LNCS 2185, pp. 134–148, 2001. Springer-Verlag Berlin Heidelberg 2001

[Sun, 13] Wuliang Sun, Robert B. France et Indrakshi Ray. Analyzing Behavioral Refactoring of Class Models. Technical Report CS-13-104 Computer. Colorado State University. 2013.

[Opdyke, 92] William F. Opdyke. Refactoring Object-Oriented Frameworks. Thèse de Doctorat, University of Illinois at Urbana-Champaign, 1992.

[Van Der Straeten et al. ,06] Ragnhild Van Der Straeten ,Viviane Jonckers et Tom Mens. A formal approach to model refactoring and model refinement. Springer-Verlag. 2006.

[Vojtisek, 09] Didier Vojtisek. OpenEmbeDD: La plate-forme et ses outils d'ingénierie de modèles. Génie Logiciel - Neptune - Ingéniérie dirigée par les modèles 89, pages 31-37. 2009.

[Wiedemann, 06] Antoine Wiedemann. Approche MDA pour la transformation d'un modèle UML en schéma relationnel. Université de Rennes 1 - Laboratoire IRISA. INFORSID 2007, May 2006, Perros-Guirec, France. pp.587. <hal-00450857>.2006.

[Zekkaoui et al., 14] Mounir Zekkaoui et Abdelhadi Fennan. Unified Approach for Building Heterogeneous Artifacts and Consistency Rules. Journal Of Emerging Technologies In Web Intelligence, Vol. 6, No. 1, February 2014.

[Zerdazi ,03] Amar Zerdazi. Représentation de schémas de bases de données hétérogènes sous forme de méta-schémas XML. Christian FLUHR Gaberiella SALZANO. 2003. Hal <mem 00000130>. 2003.

[Zitouni, 96] Mohammed Zitouni. Elaboration d'un outil d'évaluation et d'amélioration du processus de maintenance des logiciels : Une étude exploratoire. Rapport final d'activité de synthèse. Université du Québec à Montréal, 1996.

[Zoukar ,04] Iyad Zoukar, Camille Salinesi et Colette Rolland. Evolution du système d'information par l'implantation d'un Progiciel de Gestion Intégrée Systématiser la mise en correspondance entre le système et l'organisation. 2004.