MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
وزارة التعليـــم العـالـي والبـحـث العلمـي

BADJI MOKHTAR-ANNABA UNIVERSITY
UNIVERSITÉ BADJI MOKHTAR-ANNABA

جامعـة باجي مختار – عنابـــــــــة

Faculté des Sciences de L'Ingéniorat **Année 2015-2016**

Département d'Informatique

THESE

Présentée en vue de l'obtention

du diplôme de Doctorat en Sciences

Préservation de la traçabilité des préoccupations dans un contexte d'ingénierie basée sur les modèles

Option

Génie Logiciel

Par

Mr Mohamed Yassine HAOUAM

Directeur de Thèse :

Mr Djamel MESLATI Professeur Université Badji Mokhtar-Annaba

DEVANT LE JURY

Président :

Fadila ATIL	Professeur	Université Badji Mokhtar-Annaba

Examinateurs :

Allaoua CHAOUI	Professeur	Université Abdelhamid MEHRI-Constantine 2
Zineddine BOURAS	MCA	Ecole Préparatoire aux Sciences et Techniques, Annaba
Hakim BENDJENNA	MCA	Université Larbi Tébessi-Tébessa

Résumé

L'ingénierie dirigée par les modèles (IDM) représente une approche aujourd'hui reconnue pour maîtriser la complexité croissante des architectures logicielles et matérielles. En effet, L'IDM permet, entre autre, de représenter ces architectures de façon abstraite puis de générer, grâce à des transformations successives, le code source correspondant.

Dans un processus MDE, les relations de traçabilité sont étroitement liées aux artefacts logiciels (modèles). Les liens de traces sont généralement stockés dans des modèles qui peuvent être produits par des transformations. Lors de l'évolution d'un modèle, il est nécessaire d'analyser l'impact du changement de modèles sur les liens de traces et d'effectuer des opérations de maintenance permettant d'assurer la consistance entre liens de traces et modèles. Vu le nombre important de relations de traces générées lors du développent d'un logiciel, de petite taille, la gestion manuelle de ces liens constitue, en effet, une tâche fastidieuse et source d'erreurs.

Dans cette thèse, nous proposons une approche pour la maintenance des liens de traces lorsqu'une transformation est totalement ou partiellement achevée.

Cette thèse traite aussi de l'utilisation des liens de trace pour assurer la synchronisation des artefacts lorsqu'un modèle évolue.

Abstract

Model Driven Engineering (MDE) enable the control of the increasing complexity of software and hardware architectures. Indeed, the MDE allows, among others, to represent these architectures in an abstract way and then to generate, through successive transformations, the corresponding source code.

In an MDE process, the traceability relations are tightly coupled with software artefacts (models). Trace links are usually stored in the form of tracing models produced by transformations. When changes occur to the artifacts, it is necessary to understand the impact of the development activity on the traceability links and attempts to keep these links in synch with the models. Manual maintenance of traceability can be time-consuming and error-prone due to the large number of potential relationships that exist even for small software systems.

In this thesis, we provide an approach for the maintenance of trace links when a transformation was completely or partially achieved.

This thesis is also concerned with the use of trace links to ensure the artefacts synchronization when a model evolves.

ملخص

تعد الهندسة بواسطة النماذج من بين المناهج التي تساهم حاليا بالتحكم في التعقيدات المتزايدة لهندسة البرمجيات و العتاد. في الواقع، من بين ما توفره لنا الهندسة بواسطة النماذج، امكانية تمثيل مختلف البنيات بطريقة مجردة و من ثم التحويل المتتالي للنماذج الى غاية الحصول على البرامج الموافقة لها.

اثناء معالجة النماذج، تتشكل مجموعة من الروابط التي تسمح لنا بمتابعة مختلف النماذج المتحصل عليها. يتم عادة تخزين هذه الروابط في نماذج خاصة يمكن الحصول عليها كنتائج فرعية لعملية التحويل. في حالة اجراء تعديلات على احد النماذج، يعد من الضروري تحليل مدى تأثير هذا التغير على روابط المتابعة وبالتالي انجاز عمليات الصيانة اللازمة لضمان التناسق بين النماذج و روابط المتابعة. ونظرا للعدد الكبير من الروابط التي يتم إنشاؤها اثناء تطوير البرمجيات (ذات الحجم الصغير)، فان التسيير اليدوي لهذه الروابط يعد عملية شاقة و عادة ما تنتج عنه الكثير من الاخطاء.

في هذه الاطروحة، نقترح منهاج لصيانة روابط المتابعة بعد الانجاز الكلي او الجزئي لعملية التحويل.

كما تم التطرق ايضا الى امكانية استعمال روابط المتابعة لضمان التناسق بين النماذج المرتبطة ببعضها في حالة اجراء تعديلات على احداها.

Remerciements

Je tiens en tout premier lieu à remercier les nombreuses personnes qui m'ont aidées à mener à bien cette thèse, sur le plan professionnel comme sur le plan personnel.

J'adresse mes premiers remerciements à Djamel MESLATI , Professeur à l'Université Badji Mokhtar de Annaba, pour m'avoir proposé ce sujet et pour m'avoir encadrer et conseiller.

Je tiens à remercier tout particulièrement les membres du jury pour l'honneur qu'ils me font d'avoir accepté de rapporter ce mémoire.

Je tiens à exprimer mes sincères remerciement à Jean-Pierre Giraudin, qui m'a accueilli dans son équipe de recherche SIGMA à Grenoble.

Merci à tous les membres de l'équipe IRIT-MACAO pour leur soutien, nos discussions et tous les bons moments que nous avons passés ensemble.

Je remercie chaleureusement les membres de l'équipe Dart (Equipe conjointe LIFL, INRIA), et plus particulièrement Anne Etien, pour tous les conseils qu'elle a pu me donner et les discussions que nous avons eu ensemble.

Je tiens aussi à remercier Abdelhak-Djamel Serai, de m'avoir accueilli dans son équipe MAREL du LIRMM, et de m'avoir encouragé et conseillé.

Je remercie ma famille pour leur soutien indéfectible, leur patience et leurs encouragements. Je remercie du fond du cœur ma mère, mes frères, mes sœurs, mes beaux-frères, mes belles-sœurs, ma femme et mes trois belles filles *Chahd*, *Sirine* et *Missene*.

J'ai encore un merci particulier à tous mes amis pour leur prévenance et leurs encouragements.

Dédicaces

A l'âme de mon père.

Table des matières

Introduction générale **7**

0.1 Contexte et problématique 8

0.2 Motivations 9

0.3 Objectifs de la thèse 10

0.4 Organisation de ce document 10

1 État de l'art sur l'ingénierie dirigée par les modèles **13**

1.1 Introduction 14

1.2 Les principes généraux de l'IDM 15

1.2.1 Modèle 15

1.2.2 Métamodèle : langage de modélisation 16

1.3 L'approche MDA 19

1.3.1 Le modèle CIM 21

1.3.2 Le modèle PIM 21

1.3.3 Le modèle PDM 21

1.3.4 Le modèle PSM 22

1.4 Transformation de modèles 22

1.4.1 Approches de transformation 22

1.4.2 Principes de la transformation de modèles 25

1.4.3 Propriétés des transformations 26

1.4.4 Langages de transformation 27

1.5 Conclusion 32

2 Séparation des préoccupations dans l'IDM 33

2.1 Introduction . . . 34
2.2 Modélisation des préoccupations . . . 35
2.2.1 Modélisation Orientée-Aspect . . . 35
2.2.2 Multi-modélisation . . . 37
2.3 Composition des préoccupations . . . 37
2.3.1 Composition dans les approches de modélisation orientées aspect 38
2.3.2 Composition dans les approches de transformation de modèles 44
2.4 Conclusion . . . 51

3 Traçabilité dans les transformations de modèles 53

3.1 Introduction . . . 55
3.2 Utilisations des traces . . . 56
3.3 Stockage des informations de traçabilité . . . 57
3.3.1 Liens de traçabilité intégrés dans les modèles sources/cibles . 57
3.3.2 Liens de traçabilité stockés dans un modèle externe . . . 58
3.4 La capture des liens de traçabilité . . . 58
3.4.1 Liens de traçabilité explicite . . . 58
3.4.2 Traçabilité implicite . . . 60
3.5 traçabilité de modèles vers texte . . . 60
3.5.1 Méta-modèle de trace de modèles vers texte . . . 61
3.5.2 Génération d'une trace de modèles vers texte . . . 62
3.6 Approches de traçabilité . . . 62
3.6.1 Traçabilité pour l'ingénierie des exigences . . . 62
3.6.2 Traçabilité pour les modèles . . . 70
3.6.3 Traçabilité pour les transformations . . . 73
3.7 Etude comparative des approches de traçabilité . . . 75
3.7.1 Représentation . . . 75
3.7.2 Outillage . . . 76
3.7.3 Analyse d'impact . . . 77
3.7.4 Evolutivité (Scalability) . . . 78

3.7.5 Synthèse . . . 79
3.8 Conclusion . . . 81

4 Maintenance de la traçabilité 83
4.1 Introduction . . . 84
4.2 Un scénario d'évolution de modèles . . . 85
4.2.1 Description des modèles . . . 85
4.2.2 Chaîne de transformation . . . 87
4.2.3 Scénario d'évolution . . . 101
4.3 Processus d'évolution des liens de traces . . . 103
4.3.1 Comparaison de modèles . . . 104
4.3.2 Détection et classification des changements . . . 108
4.3.3 Evolution des liens de traces . . . 109
4.4 Validation de l'approche . . . 110
4.4.1 Objectifs et questions de recherche . . . 111
4.4.2 Ressource expérimentale . . . 112
4.4.3 Résultats et discussion . . . 113
4.5 Conclusion . . . 114

5 Utilisation de traces pour la synchronisation de modèles 117
5.1 Introduction . . . 118
5.2 Évolution de Logiciels . . . 119
5.2.1 Catégories de l'évolution de logiciels . . . 120
5.2.2 Activités de développement évolutionnaire . . . 121
5.3 Evolution dans l'IDM . . . 122
5.3.1 Model-metamodel Co-Evolution . . . 123
5.3.2 Synchronisation de modèles . . . 129
5.3.3 Evolution de Transformation . . . 132
5.4 Présentation de l'approche . . . 133
5.4.1 Détection et classification des changements . . . 133
5.4.2 Analyse d'impact du changement . . . 136
5.4.3 Synchronisation d'artéfacts . . . 137

5.5 Conclusion 138

6 Conclusion générale **139**

6.1 Rappel du cadre et des objectifs de la thèse 140

6.2 Bilan 140

6.3 Perspectives 142

Bibliographie **143**

Contributions Scientifiques **154**

Liste des figures

1.1 Architecture à 4 couches de méta-modélisation 17
1.2 Extrait du métamodèle ECore [101] 18
1.3 Edition des modèles Java dans l'éditeur générique de EMF. 19
1.4 Editeur de méta-modèles EMF sous forme arborescente 20
1.5 Les différents modèles dans MDA 20
1.6 Schéma de base d'une transformation 26
1.7 Architecture du standard QVT. 28
1.8 Extrait du métamodèle des règles ATL. 29
1.9 Composition d'un métamodèle d'action dans le métamodèle EMOF. 31

2.1 Représentation de la fonctionnalité Logged dans Theme/UML [7] . . 39
2.2 Composition des Themes Logger et CMS 40
2.3 Le modèle d'aspect "Observer" en utilisant la notation France et al. [96] . 41
2.4 Le modèle d'aspect spécifique à un contexte en utilisant la notation France et al. [96] 42
2.5 Exemple de règle de transformation d'un modèle de classe UML [106] 44
2.6 Processus de génération de transformations orientées composition [31] 45
2.7 Métamodèle de tissage de base [31] 46
2.8 Exemple de fusion avec EML [93] 49

3.1 Méta-modèle de trace de modèles vers texte [84] 61
3.2 Types de liens de traçabilité 63
3.3 Métamodèle de traçabilité [88] 65
3.4 Architecture de l'approche EBT 68

3.5 Méta-modèle de traçabilité [64] . . . 73
3.6 Méta-modèle de trace des transformations [36] . . . 74
3.7 Exemple de trace pour une chaîne de transformations [36] . . . 75

4.1 Transformations d'un système bancaire sécurisé . . . 85
4.2 Bank model . . . 86
4.3 Security model . . . 87
4.4 Secure bank model . . . 87
4.5 Extrait du méta-modèle UML concernant le diagramme de classes. . . 89
4.6 Métamodèle de tissage (Weaving metamodel) . . . 90
4.7 Modèle de tissage (Weaving model) . . . 91
4.8 Modèle UML résultat de la composition *(Secure bank model)*. . . . 94
4.9 Extrait du méta-modèle Java. . . . 95
4.10 Méta-modèle de traçabilité . . . 98
4.11 Exemple des liens de traces . . . 99
4.12 Exemple de modèle de traces sous forme arborescente . . . 100
4.13 Modèles de la transformation T2 en XMI . . . 102
4.14 Exemple d'évolution du modèle "Secure Bank" . . . 103
4.15 Processus d'évolution des liens de traces [56] . . . 104
4.16 Extrait du métamodèle de différence d'EMF Compare. . . . 105
4.17 Différences entre les deux versions de modèle . . . 108
4.18 Analyse quantitative . . . 114

5.1 Processus de co-évolution [52] . . . 126
5.2 Structure générale de l'approche de représentation de modèle de différence [20] . . . 128
5.3 Incremental transformation [57] . . . 130
5.4 Schéma de notre approche de synchronisation [54] . . . 133
5.5 Opération *RenameElement* . . . 134
5.6 Opération *DeleteElement* . . . 135
5.7 Opération *AddElement* . . . 135
5.8 Détection des éléments impactés . . . 137

Liste des tableaux

3.1 Synthèse de l'étude des travaux existants 80

4.1 Evolution des relations de traçabilité 113

Listings

2.1 Extrait des spécifications EML de fusion de deux modèles [93] 48
3.1 Production d'un lien de trace ATL 59
4.1 Règle de création de classes en ATL 91
4.2 Règle de création de propriétés en ATL 92
4.3 Règle de création des opérations en ATL 93
4.4 Extrait du code de transformation ATL 96
4.5 Règle ATL tracée (C2C) 100
4.6 Comparaison de modèles avec EMF Compare 107
4.7 Structure de base d'une règle d'évolution 110

Introduction générale

Sommaire

0.1 Contexte et problématique 8
0.2 Motivations 9
0.3 Objectifs de la thèse 10
0.4 Organisation de ce document 10

0.1 Contexte et problématique

Depuis les débuts du génie logiciel, la taille et la complexité des logiciels développés augmentent de plus en plus rapidement alors que les contraintes de temps de développement, de qualité, de maintenance et d'évolution sont toujours plus fortes. Dans ce contexte, les techniques de génie logiciel sont contraintes à évoluer sans cesse pour permettre de gérer la complexité et d'assurer la qualité du logiciel produit. Cette évolution se fait de manière continue depuis la création d'assembleurs et de compilateurs dans les années 1950 jusqu'à l'apparition de plateformes de programmation orienté-objets à la fin des années 1990. Cette évolution se traduit par une augmentation progressive du niveau d'abstraction auquel sont développés les systèmes logiciels.

Aux vues de ces complexités, de nouvelles techniques et paradigmes de programmations sont sans cesse recherchés. Les travaux proposés par l'IDM (Ingénierie Dirigée par les Modèles) apparaissent comme un moyen de se défaire des détails d'implémentation liés à l'utilisation, par exemple, d'une bibliothèque ou d'un langage de programmation particulier. La spécification du système peut alors se faire grâce à un modèle de conception, exprimant les différentes fonctionnalités du système modélisé. Comme complément à l'utilisation de modèles pour abstraire les détails d'implémentation, l'IDM propose l'utilisation de transformations de modèles à modèles et de modèles vers texte permettant de passer automatiquement d'un niveau d'abstraction à un autre. Ainsi, les détails d'implémentation liés à l'application peuvent être masqués au concepteur et le code source d'une application modélisée peut être directement généré. L'utilisation conjointe de la modélisation et des transformations de modèles offrent un support attractif pour le développement rapide d'applications en permettant d'éviter les détails et spécificités introduits par un langage de programmation.

En IDM, à chaque fois qu'une transformation de modèles a lieu, des éléments sont consommés pour en produire de nouveau. Pour garder des informations sur les éléments qui en ont produit d'autres, il est important de conserver des liens entre les éléments utilisés et ceux produits par une transformation, appelés liens

de traçabilité. Par exemple, la traçabilité est utilisée dans le cadre de l'ingénierie des exigences afin de s'assurer que les exigences d'un projet se retrouvent dans les modèles, le logiciel et les cas de tests. La traçabilité peut également être utilisée pour déterminer les différents éléments impactés par chaque modification dans la spécification ou dans les besoins d'une application informatique. Ces changement doivent donc être identifiés pour pouvoir co-évoluer les artéfacts impactés avec moins d'efforts.

La mise en oeuvre d'une solution de traçabilité n'est pas une tâche aisée, il peut être difficile d'identifier le niveau de traçabilité nécessaire dans un contexte particulier. Plusieurs techniques peuvent être utilisées pour récupérer et maintenir les liens de trace. Nous nous proposons de travailler sur cette problématique, d'une part en maintenant les liens de trace entre les différents éléments impliqués dans une chaîne de transformation, d'autre part, en maintenant la cohérence entre les modèles d'entrée et les modèles de sortie, lorsque un modèle d'entrée évolue.

0.2 Motivations

La traçabilité a longtemps été reconnue comme l'une des exigences les plus importantes du processus de développement de logiciels [51]. Dans une approche basée sur les modèles, deux niveaux de traçabilité sont nécessaires : il faut d'une part établir les liens de traçabilité entre les différents artefacts créés ou générés, et d'autre part suivre l'évolution de chacun de ces éléments et la propager sur les artefacts impactés par ce changement [3].

Malgré l'importance de la traçabilité dans le développement logiciel, sa pratique n'est pas encore très répandue. Le coût, l'effort et la discipline nécessaires pour créer et maintenir des liens de trace dans un système logiciel en évolution rapide peut être extrêmement élevé [25].

Très souvent, au cours de la maintenance et de l'évolution d'un logiciel, les développeurs se concentrent seulement sur la production de code. Les liens de traçabilité deviennent alors obsolètes car les développeurs n'ont pas de temps ou ne peuvent pas en consacrer à la mise à jour de ces liens. Ceci est généralement dû soit au

processus de traçabilité qui peut être mal définis et ad-hoc soit en raison du manque d'outils de traçabilité efficaces . Ceci nous à motivé a chercher une solution permettant d'améliorer le processus de traçabilité et par conséquent réduire le coût et l'effort nécessaires pour la création et la maintenance des liens de trace.

La deuxième motivation de cette thèse est d'améliorer le processus de développement de logiciels, et d'intervenir plus précisément dans la tâche de l'analyse de l'impact de changement. Nous visons principalement la réduction de l'effort ainsi que le coût de l'évolution d'un artefact en introduisant une méthode de propagation du changement, entre l'élément modifié et les artéfacts impactés par ce changement, fondée sur la synchronisation incrémentales de modèles.

0.3 Objectifs de la thèse

Cette thèse tente de répondre à certaines interrogations liées à la préservation de la traçabilité des préoccupations dans le cadre de l'IDM. Par conséquent, Cette thèse vise à proposer une approche basée sur l'IDM pour la maintenance des liens de traces produits par une chaîne de transformations lors de l'évolution de modèles. Notre approche est fondée sur la définition rigoureuse de (méta)modèles et de processus de transformation afin de pouvoir :

— définir un processus d'ingénierie basé sur l'approche IDM, permettant de prendre en compte à la fois l'aspect fonctionnel et l'aspect non-fonctionnel,

— capturer et conserver les liens de traçabilité entre les modèles d'entrée et de sortie dans un processus de transformation de modèles basé sur la séparation de préoccupations,

— Proposer une approche permettant l'utilisation des liens de traces produits par la chaîne de transformations pour la synchronisation de modèles.

0.4 Organisation de ce document

Le chapitre 1 a pour objectif de présenter les principes de l'ingénierie Dirigée par les Modèles (IDM) et plus particulièrement le concept de chaîne de transformations,

notion clef qui est utilisée dans l'intégralité de cette thèse.

Le chapitre 2 présente les principales approches basées sur la séparation des préoccupations.

Le chapitre 3 présente un état de l'art sur l'utilisation de la traçabilité dans l'IDM. Nous étudions tout d'abord des travaux issus du domaine de l'ingénierie des exigences. En second lieu, nous étudions des travaux plus spécifiques au domaine de l'IDM. Une synthèse est proposée et un ensemble de problématiques encore ouvertes est déduit de cette étude.

Le chapitre 4 présente notre approche pour la maintenance des liens de traces. La première partie sert á présenter notre technique de traçabilité permettant la capture et la conservation des liens de traçabilité entre les modèles d'entrée et les modèles de sortie dans un processus de transformation de modèles basé sur la séparation de préoccupations. La deuxième partie décrit notre approche permettant d'assurer l'intégrité des modèles de traces produits par la chaîne de transformations.

Le chapitre 5 s'attache à répondre à une problématique de synchronisation de modèles au moyen d'une approche basée sur les liens de trace. l'approche proposée permet d'assurer la cohérence entre les modèles consommés et les modèles produits par la chaîne de transformations lors de l'évolution de modèles.

Finalement, le chapitre 6 donne une conclusion et dresse un bilan récapitulant les idées principales de notre thèse et nos contributions. On y dresse également les extensions possibles et les principales perspectives dégagées à l'issue de nos travaux.

Chapitre 1

État de l'art sur l'ingénierie dirigée par les modèles

Sommaire

1.1 Introduction 14

1.2 Les principes généraux de l'IDM 15

1.2.1 Modèle 15

1.2.2 Métamodèle : langage de modélisation 16

1.3 L'approche MDA 19

1.3.1 Le modèle CIM 21

1.3.2 Le modèle PIM 21

1.3.3 Le modèle PDM 21

1.3.4 Le modèle PSM 22

1.4 Transformation de modèles 22

1.4.1 Approches de transformation 22

1.4.2 Principes de la transformation de modèles 25

1.4.3 Propriétés des transformations 26

1.4.4 Langages de transformation 27

1.5 Conclusion 32

1.1 Introduction

L'ingénierie Dirigée par les Modèles (IDM), ou Model Driven Engineering (MDE) en anglais, est une branche de l'ingénierie du logiciel. Selon [10], l'IDM est une généralisation de la programmation orientée objet (OOP). Les concepts principaux de l'OOP sont les *classes* et les *instances* et deux relations *instance de* et *hérite de*. Un objet est une instance d'une classe et une classe peut étendre une autre classe. Pour l'IDM, le terme fondamental est celui de *modèle*. Un modèle représente un point de vue d'un système et il est défini par le langage de son *métamodèle*. Autrement dit, un modèle contient des *éléments* conforment aux *concepts* et aux *relations* exprimées dans le métamodèle. Les deux relations de base entre un *modèle* et son *métamodèle* sont *représenté par* et *conforme à*. Un modèle représente une partie d'un système et il est conforme à un métamodèle. De même un métamodèle est conforme à un autre méta-métamodèle, habituellement cette régression est stoppée en considérant que le méta-métamodèle est conforme à lui même.

L'IDM a pour but d'apporter une nouvelle vision unifiée permettant de concevoir des applications en séparant la logique métier de l'entreprise, de toute plateforme technique. En effet, la logique métier est stable et subit peu de modifications au cours du temps, contrairement à l'architecture technique. Il est donc évident de séparer les deux pour faire face à la complexité des systèmes d'information et aux coûts excessifs de migration technologique.

En novembre 2000, l'Object Management Group (OMG) a proposé l'approche nommée Model Driven Architecture (MDA) pour le développement et la maintenance des systèmes à prépondérance logicielle [99]. MDA applique la séparation des préoccupations entre la logique métier des systèmes informatiques et les plates-formes utilisées et se fonde sur l'utilisation massive des modèles. Ainsi, la pérennité est l'objectif principal de MDA. Il s'agit de faire en sorte que la logique métier des applications ne soit plus mêlée aux considérations techniques de mise en production. Il devient dès lors possible de capitaliser les savoir-faire et d'être beaucoup plus réactif aux changements technologiques. Pour atteindre cet objectif, MDA vise à représenter sous forme de modèles toute l'information utile et nécessaire à la construction

et à l'évolution des systèmes d'information. Les modèles sont au centre du cycle de vie des logiciels et des systèmes d'information.

L'objectif de ce chapitre est d'étudier les concepts clés de l'ingénierie des modèles que nous allons utiliser fréquemment tout au long du reste du document. La section 1.2 définit les notions de modèle, métamodèle, méta-métamodèle et les relations qui existent entre ces concepts. La section 1.3 présente l'approche de modélisation de l'OMG : MDA *(Model Driven Architecture)*. La section 1.4 définit la notion de transformation de modèle et présente les différentes techniques de transformations existantes.

1.2 Les principes généraux de l'IDM

l'ingénierie dirigée par les modèles (IDM) est basée sur le principe du "tout est modèle" [13]. L'idée centrale est d'utiliser autant de modèles, à différents niveaux d'abstraction, que la description d'un système le nécessite. Le code réel de l'application n'est plus considéré comme artefact de première importance car il est obtenu par transformation d'un modèle plus abstrait représentant le processus métier de l'application vers un autre plus concret représentant le système selon une technologie donnée.

Cette section définit les concepts de base de l'IDM et les relations qui existent entre ces concepts.

1.2.1 Modèle

Il existe plusieurs définitions de ce qu'est un modèle. Généralement, les modèles sont utilisés pour analyser le système spécifié ou pour générer l'implémentation du système requis. Il est presque impossible d'avoir une définition universellement acceptée [70]

L'OMG définit le modèle par *"A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text"* [63] qu'on peut traduire par : "Le modèle d'un système est une description ou une spécification de ce système et de son

environnement pour un objectif donné. Un modèle est souvent présenté comme une combinaison de diagrammes et de texte". Cette définition présente l'idée générale de ce qu'est un modèle et comment il est représenté.

Bézivin définit le modèle par *"A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system"* [12]. Dans cette définition il est important de noter qu'un modèle est une abstraction d'un système. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé. On dit alors que le modèle représente le système.

1.2.2 Métamodèle : langage de modélisation

Un *métamodèle* est un modèle qui définit le langage utilisé pour décrire des modèles [29]. Un métamodèle représente les concepts du langage de méta-modélisation utilisé et la sémantique qui leur est associée. En d'autres termes, le métamodèle décrit le lien existant entre un modèle et le système qu'il représente. On dit qu'un modèle est conforme à un métamodèle si l'ensemble des éléments du modèle sont définis par le métamodèle.

Un métamodèle devrait faire partie d'une architecture de méta-modélisation, qui permet à un métamodèle à être considérée comme un modèle, et d'une manière similaire, il est lui-même décrit par un autre métamodèle. On désigne ce métamodèle particulier par le terme *méta-métamodèle* [22]. Pour supporter cette approche l'OMG a introduit une architecture à 4 couches de méta-modélisation. Ces couches sont représentées dans la figure 1.1 :

Dans les sous-sections suivantes, nous présenterons le Meta Object Facility (MOF)[80] qui a été défini par l'OMG comme standard pour la définition de méta-modèles et son implémentation par l'Eclipse Modelling Framework (EMF) [76].

1.2.2.1 Meta Object Facility (MOF)

Le Meta Object Facility (MOF) est la technologie proposée par l'OMG pour définir des méta-données. Nous pouvons définir une méta-donnée comme étant une

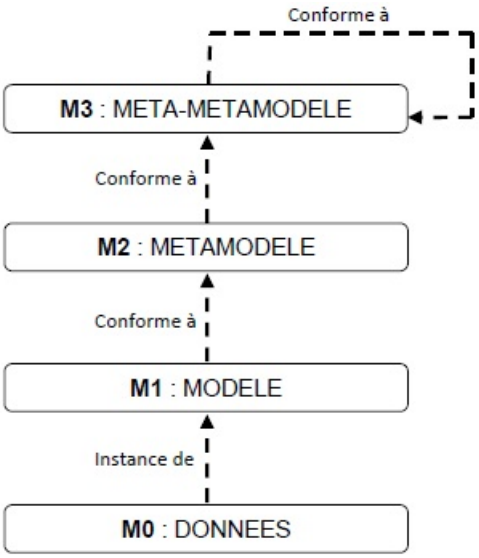


FIGURE 1.1 – Architecture à 4 couches de méta-modélisation

donnée sur les données. Le but du MOF est de fournir un cadre de travail (framework) supportant tout type de méta-donnée et permettant la définition de nouveaux types au fur et à mesure de l'évolution des besoins. Le MOF spécifie la structure et la syntaxe de nombreux métamodèles comme UML (Unified Modeling Language), CWM (Common Warehouse Metamodel) et SPEM (Software Process Engineering Metamodel). Le MOF spécifie aussi des mécanismes d'interopérabilité entre ces métamodèles [80].

1.2.2.2 Eclipse Modeling Framework (EMF)

L'Eclipse Modeling Framework (EMF) est un framework de modélisation et une infrastructure de génération de code pour la construction d'outils et des applications basées sur des modèles de données structurées [76]. Depuis un modèle décrit généralement en XMI (XML Metadata Interchange) [79], EMF fournit des outils permettant de produire des classes Java représentant le modèle avec un ensemble de classes pour adapter les éléments du modèle afin de pouvoir les visualiser, les éditer avec un système de commandes et les manipuler dans un éditeur.

Le plus important est qu'EMF fournit les fondements à l'interopérabilité avec

d'autres outils ou applications basés sur EMF. EMF propose un langage de méta-modélisation appelé Ecore qui diffère sensiblement de celui défini par le MOF. La figure 1.2 présente un extrait du métamodèle Ecore.

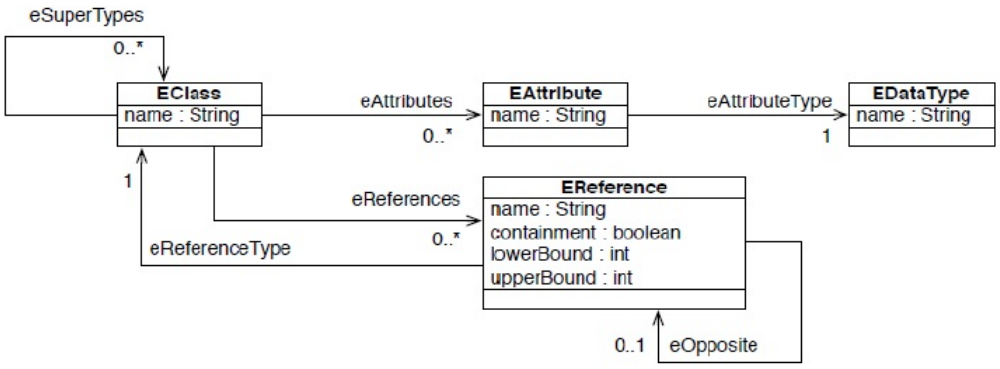


FIGURE 1.2 – Extrait du métamodèle ECore [101]

Le métamodèle *Ecore* est basé sur quatre méta classes : *EClass*, *EAttribute*, *EDataType* et *EReference*. La méta-classe EClass est utilisée pour modéliser une classe. Une classe possède un nom, des attributs et des références. Un attribut est caractérisé par un nom et un type. La méta-classe *EReference* représente la référence vers une classe utilisée par une des extrémités d'une association. Une référence possède un nom, et un type de référence représentant un lien vers une classe. La méta-classe *EDataType* représente le type d'un attribut (ex. entier, booléen, etc.).

EMF permet de créer deux types de modèles, d'un côté des modèles définissant des concepts (*méta-modèle*) et de l'autre des modèles instanciant ces concepts. Tout modèle EMF est une instance d'un modèle EMF avec pour racine commune le modèle Ecore fournit par EMF. EMF permet non seulement de créer un méta-modèle représentant les concepts désirés par l'utilisateur mais il permet ensuite à l'utilisateur de créer des modèles issus de ce méta-modèle et de les manipuler en permettant la génération des éditeurs spécifiques au méta-modèle. La figure 1.3 présente une capture d'écran de l'éditeur générique d'Eclipse/EMF. Cet éditeur présente les modèles sous la forme d'une arborescence d'objets et permet d'éditer les propriétés de chaque objet. Nous avons créé un modèle Java conformément au métamodèle Java donné dans la figure 1.4. L'éditeur produit un modèle au format

XMI qui peut être ensuite utilisé par une transformation.

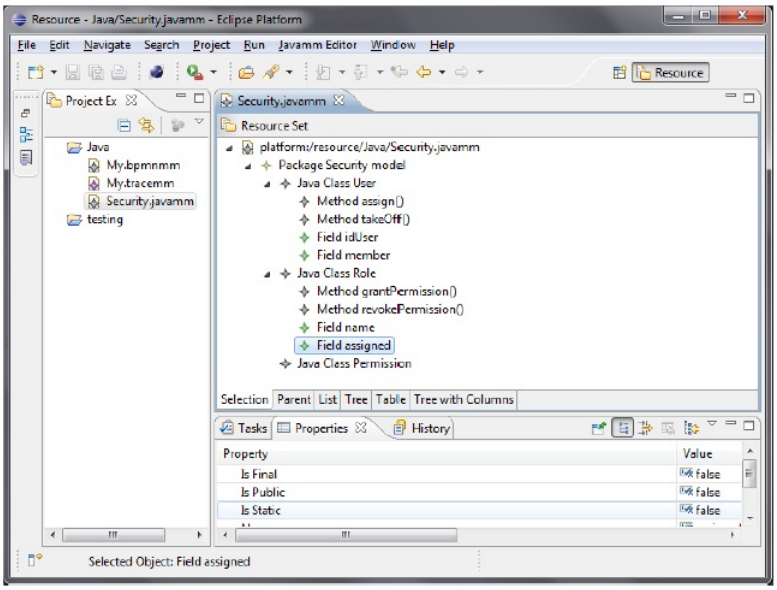


FIGURE 1.3 – Edition des modèles Java dans l'éditeur générique de EMF.

1.3 L'approche MDA

Le MDA (Model Driven Architecture) est une initiative de l'OMG rendue publique en 2000. C'est une proposition à la fois d'une architecture et d'une démarche de développement [85]. L'idée de base du MDA est de séparer les spécifications fonctionnelles d'un système des spécifications de son implémentation sur une plate-forme donnée.

MDA est basé sur plusieurs formalismes standards de modélisation, notament UML, MOF et XMI, afin de promouvoir les qualités intrinsèques des modèles, telles que pérennité, productivité et prise en compte des plateformes d'exécution. Le principe clé de MDA consiste en l'utilisation de modèles aux différentes phases de développement d'une application. Plus précisément, MDA préconise l'élaboration de quatre types de modèles : CIM (Computation Independent Model), PIM (Platform Independent Model), PDM (Platform Description Model) et PSM (Platform Specific Model). Ces modèles de base seront présentés plus en détail dans les sous sections

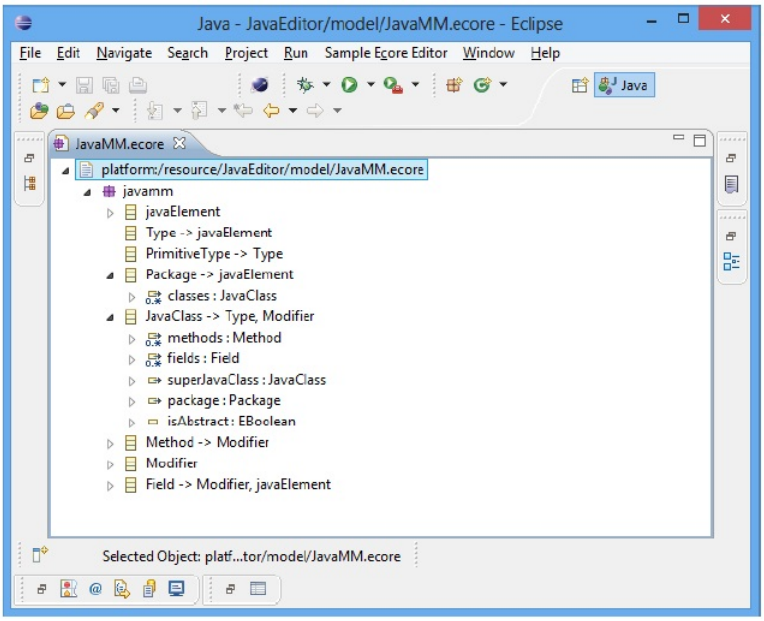


FIGURE 1.4 – Editeur de méta-modèles EMF sous forme arborescente

suivantes.

Une méthode de développement de logiciels selon l'approche MDA doit décrire comment construire des systèmes logiciels de manière fiable et reproductible en utilisant les différents types de modèles. La figure 1.5 explicite les différents types de modèles élaborés dans un cycle de développement en Y, ainsi que les relations qui existent entre eux.

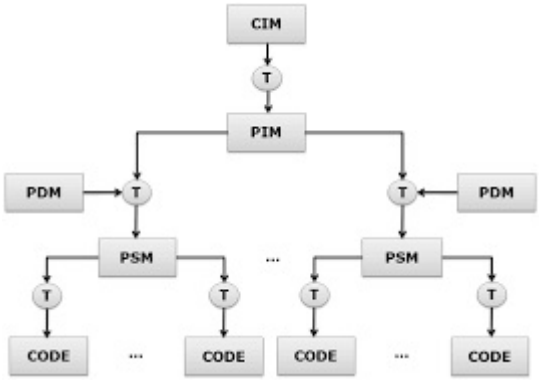


FIGURE 1.5 – Les différents modèles dans MDA

1.3.1 Le modèle CIM

C'est le modèle métier ou le modèle du domaine d'application. Le CIM (*Computation Independent Model*) permet la vision du système dans l'environnement où il opèrera, mais sans rentrer dans le détail de la structure du système ni de son implémentation. Dans ce modèle, les exigences sont exprimées en utilisant le vocabulaire du domaine [33]. L'objectif du CIM est l'établissement d'un consensus de compréhension entre les experts du domaine métier et les concepteurs du système. Les modèles d'exigences peuvent être considérés comme des éléments contractuels, destinés à servir de référence lorsqu'on voudra s'assurer qu'une application est conforme aux demandes du client.

1.3.2 Le modèle PIM

Un modèle PIM (*Platform Independent Model*) présente une indépendance vis-à-vis de toute plateforme technique (EJB, CORBA, .NET, etc.). Le PIM représente la logique métier spécifique au système ou le modèle de conception. Il décrit le système, mais ne montre pas les détails de son utilisation sur la plateforme. Le PIM de base représente uniquement les capacités fonctionnelles métier et le comportement de l'application. Les PIM suivants ajoutent des aspects technologiques et architecturaux mais toujours sans détails spécifiques d'une plate-forme. Par exemple, ces modèles peuvent intégrer des informations sur la persistance, les transactions, la sécurité, etc. Un PIM doit être raffiné avec les détails d'une ou plusieurs architecture(s) particulière(s) pour produire un PSM.

1.3.3 Le modèle PDM

Un modèle PDM (*Platform Description Model*) décrit la plate-forme sur laquelle le système va être exécuté (EJB, .NET, etc.). Actuellement, il est souvent défini informellement sous forme de manuels de logiciels et de matériels. Dans une démarche MDA, on se base sur les PDM pour générer les PSM à partir des PIM. Une telle approche est appelée cycle de développement en Y.

1.3.4 Le modèle PSM

Dans le PSM (*Platform Specific Model*), le système est spécifié en combinant tous les éléments spécifiés dans le PIM avec l'ajout des détails de la plateforme technique spécifiée par les architectes d'un système. Les PSM servent principalement à faciliter la génération de code à partir d'un modèle d'analyse et de conception. Ces modèles contiennent les informations nécessaires à l'exploitation d'une plateforme d'exécution. Comme pour les PIM, il existe plusieurs niveaux de PSM. Le premier est obtenu directement à partir du modèle PIM, les autres sont obtenus par transformations successives jusqu'à l'obtention d'un système exécutable.

1.4 Transformation de modèles

Les deux principaux artefacts de l'ingénierie des modèles sont les modèles et les transformations de modèles. D'un point de vue général, on appelle *transformation de modèles* tout programme dont les entrées et les sorties sont des modèles. La définition et l'automatisation des transformations a pour objectif de rendre les modèles plus opérationnels et d'augmenter la productivité du développement dans une approche IDM.

Cette section propose un tour d'horizon rapide des techniques et approches permettant la mise en œuvre des transformations de modèles dans l'IDM.

1.4.1 Approches de transformation

De nombreuses techniques sont utilisées pour élaborer des transformations de modèle. Selon la classification proposée dans [27] [28], les approches de transformations peuvent être classées en deux grandes catégories : les approches de transformations "modèle vers modèle" et les approches de transformations "modèle vers texte".

1.4.1.1 Approches "Modèle vers Modèle"

Une transformation *Modèle vers modèle* se définit par l'opération de génération d'un ou plusieurs modèles cible à partir d'un ou de plusieurs modèles source [97]. Les modèles source et cible peuvent être conformes au même métamodèle, la transformation est dite *endogène*, ou bien conformes aux métamodèles différents , la transformation est dite *exogène*. Une classification des différents types de transformation peut aussi être faite selon le changement de niveau d'abstraction. Une transformation est dite *horizontale* lorsque le modèle produit par la transformation a le même niveau d'abstraction que celui d'entrée et *verticale* lorsque le modèle produit introduit un changement de niveau d'abstraction [75].

Dans cette catégorie on trouve les approches de manipulation directe de modèle, les approches relationnelles, les approches basées sur la transformation de graphes, les approches guidées par la structure des modèles et les approches hybrides.

Les approches de *manipulation directe de modèle* offrent des mécanismes de représentation interne et un ensemble d'API pour la manipulation directe de modèles. Elles sont généralement implémentées comme un framework orienté objet. Cependant, ces API sont souvent couplées avec un langage de programmation impératif (tel que Java). Les contraintes relatives à l'implémentation telles que la gestion des éléments cibles, la sélection des éléments source, l'ordre d'exécution des règles, sont à la charge du développeur. C'est le cas, par exemple de Java Metadata Interface (JMI) [32].

L'idée de base des approches *relationnelles* consiste à établir une relation déclarative entre les éléments source et cible. Des contraintes peuvent s'ajouter pour définir cette relation. La programmation logique avec ses concepts d'unification et de résolution semble un choix naturel pour implémenter une telle approche, les prédicats peuvent être utilisés pour décrire les relations. C'est le cas, par exemple, de Mercury et F-Logic [49].

Les approches *basées sur la transformation de graphes* s'appuient sur les notions théoriques de transformation de graphes. Elles permettent de filtrer en premier lieu le patron (LHS pour left-hand side) de graphe source et le remplacer par le patron

(RHS pour right-hand side) dans le graphe cible. Dans cette catégorie on peut citer GreAT [2], UMLX [107], et BOTL [74].

Les approches *guidées par la structure* sont caractérisées par un processus de transformation de modèle à deux phases : La première phase permet de créer la structure hiérarchique du modèle cible, alors que durant la deuxième phase, les attributs et les références sont mis en place afin de compléter le modèle produit. Dans cette approche, l'environnement de transformation définit l'ordonnancement et la stratégie d'application des règles. L'utilisateur fournit seulement les règles de transformation. Un exemple type de cette approche est le framework fourni par OptimalJ.

Enfin, les approches *hybrides* combinent les différentes techniques décrites précédemment. Le langage TRL *(Transformation Rule Language)* est la composition des approches déclaratives et impératives. Une règle de mapping spécifie la relation entre les éléments source et cible. Les règles opérationnelles dans TRL représentent les règles exécutables de transformation. Le langage ATL *(*Atlas Transformation Language) [65] est aussi une approche hybride qui présente certaines similitudes avec le langage TRL.

1.4.1.2 Approches "modèle vers texte"

La transformation *modèle vers code* est généralement considérée comme un cas particulier des transformations modèle vers modèle [28]. Pour des raisons pratiques liées à la réutilisation des compilateurs existants, le code produit est souvent généré sous format texte. Dans cette catégorie On distingue deux types d'approches : les approches basées sur un mécanisme de "visiteur" et les approches basées sur des canevas ou "templates"

Le principe des approches *basées sur "visiteur"* est de fournir des mécanismes permettant de parcourir la représentation interne du modèle et d'écrire le code dans un fichier sous format texte. Le framework orienté objet Jamda est un exemple de cette approche ; il comprend un ensemble de classes Java associées aux modèles UML, une API pour manipuler les éléments de modèles, et un mécanisme "visiteur" pour générer le code.

Le deuxième type d'approche, *basées sur des canevas ou "templates"*. Dans cette catégorie on trouve plusieurs outils MDA tels que JET, Codagen Architect et AndroMDA,

1.4.2 Principes de la transformation de modèles

Le processus de transformation est composé de trois étapes :

— La définition des règles de transformation,

— L'expression des règles de transformation,

— L'exécution des règles de transformation

1.4.2.1 Définition des règles de transformation

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

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

1.4.2.2 Expression des règles de transformation

Pour exprimer les règles de transformation, un langage de spécification de règles est nécessaire. Un langage de transformation peut être déclaratif, impératif ou hybride [27]. Dans la programmation déclarative, on décrit d'une part les données du problème à traiter et d'autre part les contraintes sur ces données. Par opposition à un programme déclaratif, un programme impératif décrit comment le résultat devrait être obtenu en imposant une suite d'actions que la machine doit effectuer. Un

langage hybride regroupe à la fois les paradigmes de programmation déclarative et impérative.

1.4.2.3 Exécution des règles de transformation

Une fois spécifiées et exprimées, les règles requièrent un moteur d'exécution pour être exécutées. Ce moteur prend comme entrée le modèle et le métamodèle source, le métamodèle cible, ainsi que les règles de transformation. Il produit en sortie le modèle cible. La figure 1.6 illustre le processus de transformation de modèles.

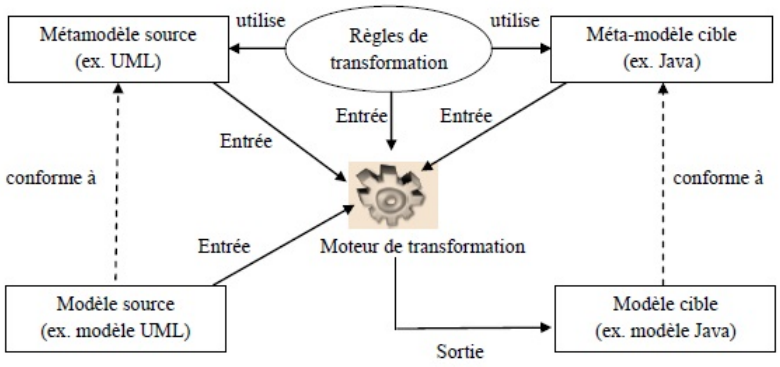


FIGURE 1.6 – Schéma de base d'une transformation

1.4.3 Propriétés des transformations

Les principales propriétés [103] qui caractérisent les transformations de modèles sont : la réversibilité, la traçabilité, la réutilisabilité, l'ordonnancement et la modularité.

— *Réversibilité* : Les transformations peuvent être unidirectionnelles ou bidirectionnelles. Une transformation est dite unidirectionnelle si elle s'exécutent dans un seul sens. Une transformation est dite réversible si elle se fait dans les deux sens.

— *Traçabilité* : La traçabilité dans les transformations consiste à créer et enregistrer des liens entre les éléments des modèles cibles et ceux des modèles sources. Certaines approches de transformations n'offrent pas de mécanisme de traçabilité ; il appartient alors à l'utilisateur de trouver un moyen pour

créer et gérer les liens de trace.

— *Réutilisabilité* : la réutilisabilité permet de réutiliser des règles de transformation dans d'autres transformations de modèles.

— *Ordonnancement* : l'ordonnancement consiste à représenter les niveaux d'imbrication des règles de transformation. En effet, les règles de transformations peuvent déclencher d'autres règles.

— *Modularité* : une transformation modulaire permet de mieux modéliser les règles de transformation en faisant un découpage du problème. Un langage de transformation de modèles supportant la modularité facilite la réutilisation des règles de transformation.

1.4.4 Langages de transformation

L'IDM se focalise sur les spécifications du MOF pour définir des langages de transformation de modèles. Dans cette section, nous présentons de façon non exhaustive les langages de transformation qui ont été définis à partir des recommandations du MOF : le standard MOF QVT [81], le langage ATL [65], le langage Kermeta [77] et le MOFScript [83].

1.4.4.1 MOF Query/View/Transformation (QVT)

Query/View/Transformation (QVT) [81] est le langage de transformation normalisé par l'OMG pour les modèles basés sur une architecture MOF. Ce standard a été proposé suite à un Request For Proposal (RFP) lancé par l'OMG en 2002 afin de chercher une réponse compatible avec sa vision MDA et ses fondations. QVT est basé sur une architecture de langages déclaratifs et impératifs. Parmi les langages déclaratifs, QVT propose un langage de relations *(QVT-Relations)* et un langage noyau *(QVT-Core)* permettant de spécifier des correspondances sur les éléments des modèles manipulés. Pour les langages impératifs, QVT définit un langage d'opérations de correspondances *(Operational Mappings)* et un langage d'implémentation interne d'opérations *(Black box)*. L'architecture proposée par QVT est présentée par la figure 1.7.

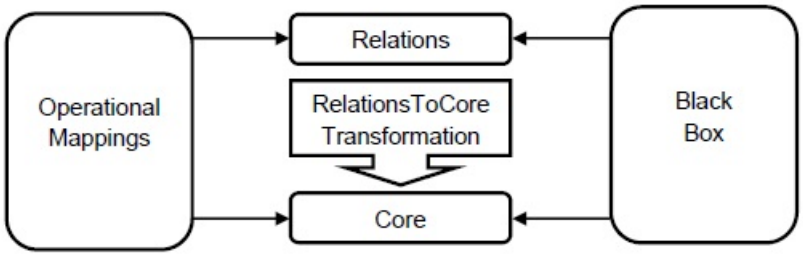


FIGURE 1.7 – Architecture du standard QVT.

Le langage *QVT-Relations* est un langage orienté utilisateur permettant de définir des transformations à un niveau d'abstraction élevé. Il a une syntaxe textuelle et graphique.

Le langage *QVT-Core* forme l'infrastructure de base pour la partie déclarative, c'est un langage de bas niveau défini par une syntaxe assez simple. Ce langage sert à spécifier la sémantique du langage *QVT-Relations*, donnée sous la forme d'une transformation *Relations2Core*.

Le langage *Operational Mappings* est une extension impérative permettant d'implémenter des règles de transformations. Il étend les deux langages déclaratifs de QVT en ajoutant des constructions impératives ainsi que des constructions OCL *(Object Constraint Language)* [82] à effet de bord afin d'assurer une certaine facilité d'utilisation aux développeurs peu habitués aux langages déclaratifs. Les opérations de correspondance *(Mappings Operations)* peuvent être utilisées pour implémenter une ou plusieurs relations quand la spécification déclarative de ces relations est complexe.

Enfin, QVT propose un deuxième mécanisme d'extension pour spécifier des transformations, en permettant d'invoquer des fonctionnalités de transformations implémentées dans un langage externe *(Black Box)*.

Le standard QVT propose une définition syntaxique et sémantique des trois sous-langages, mais ne donne aucune direction concernant l'implémentation. Ainsi, plusieurs implémentations existent actuellement, à savoir, QVTo [16] proposé par l'éditeur Borland et SmartQVT [8] proposé par France Télécom.

QVT à pour principal avantage de reposer sur des standards préexistants à savoir

le MOF et OCL, permettant ainsi de développer des outils qui seront d'une part simples à utiliser pour un grand nombre d'utilisateurs, d'autre part compatibles avec un grand nombre d'outils de modélisation existants.

1.4.4.2 ATLAS Transformation Language (ATL)

ATL [65] est un langage de transformation de modèles proposé par le groupe ATLAS de l'Université de Nantes. ATL est un langage hybride (déclaratif et impératif). L'approche déclarative d'ATL est basée sur OCL. Une règle déclarative d'ATL, appelée *Matched Rule*, est spécifiée par un nom, un ensemble de patrons sources *(InPattern)* mappés avec les éléments sources, et un ensemble de patrons cibles *(OutPattern)* représentant les éléments créés dans le modèle cible. L'approche impérative d'ATL est basée sur deux concepts : *Called Rule* et *Action Block*. l'appel d'une règle *Called Rule* est similaire à l'appel d'une procédure quand à *Action Block* est une séquence d'instructions impératives. La figure 1.8 présente un extrait du métamodèle des règles de transformation ATL.

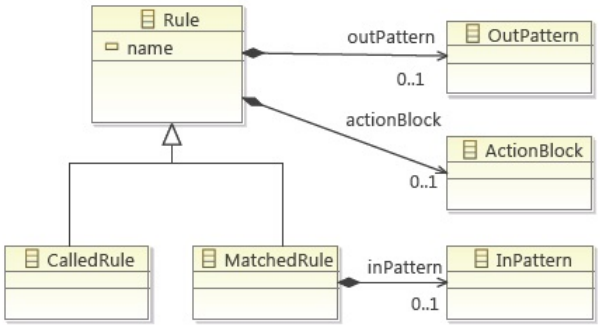


FIGURE 1.8 – Extrait du métamodèle des règles ATL.

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

Les instructions impératives ATL peuvent être regroupées en :

— Expressions : basées sur OCL.

— Variables : Une déclaration d'une variable est faite par le mot clé *let*.

Par exemple : *let VarName : varType = initialValue ;*

— Attribution : L'opérateur d'affectation $<-$ peut être utilisé à cette fin. De plus, ATL fournit l'opérateur := pouvant être utilisé quand une résolution automatique n'est pas nécessaire. Ceci permet à l'opérande de gauche de prendre la valeur de celle de droite.
— Manipulation d'instances : Les instances peuvent être créées de manière implicite grâce à l'approche déclarative. De plus, il est possible d'expliciter la création ou la destruction d'une instance en utilisant les opérateurs *new* et *delete*.
— Déclarations conditionnelles *if, else*.
— Déclaration de boucles : ATL permet la déclaration de boucles *while*, *do...while(condition)* et *foreach*.

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

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

ATL supporte deux modes d'exécution : *standard et raffinement*. Dans le mode standard, les éléments sont créés seulement quand les patrons source définis dans les règles déclaratives ont été reconnus dans le modèle source. Le système instancie alors les éléments des patrons cible. Une fois l'étape d'instanciation passée, un lien de traçabilité est créé, qui associe chaque élément reconnu du modèle source à un élément créé du modèle cible. Finalement, le système évalue ces liens de traçabilité afin de déterminer les valeurs des propriétés des éléments instanciés. Dans le mode raffinement, les éléments dont les patrons source n'ont pas été appariés par les règles sont automatiquement copiés dans le modèle cible par le moteur d'exécution. Ceci réduit considérablement le développement de transformations destinées à ne modifier qu'une petite partie d'un modèle.

1.4.4.3 Kermeta

Kermeta est un langage permettant de décrire à la fois la structure et le comportement de modèles. Il a été conçu pour être compatible avec le langage de métamodélisation EMOF (Essential MOF), un sous ensemble de MOF [101], et le langage Ecore défini par l'environnement Eclipse [101]. Il offre un langage d'action pour spécifier le comportement des modèles. Kermeta est destiné à être le langage d'un noyau de plate-forme orientée modèle. Il a été conçu pour être une base commune pour implémenter différents langages de méta-données : langages d'action, langages de contraintes ou langages de transformation. Comme les concepts de MOF sont des concepts orientés objet, le langage Kermeta inclut la plupart des mécanismes orientés objet classiques.

Au niveau architecture, Kermeta est conçu en utilisant les principes de la modélisation orientée aspect. Le métamodèle de Kermeta est construit comme un tissage entre un métamodèle d'action (aspect comportemental) et un métalangage de type EMOF (aspect structurel)(figure 1.9) . Le but est d'avoir un métalangage exécutable capable de décrire la sémantique opérationnelle des opérations.

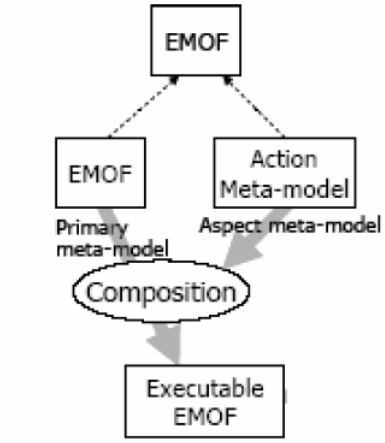


FIGURE 1.9 – Composition d'un métamodèle d'action dans le métamodèle EMOF.

1.4.4.4 MOFScript

Les langages de transformation tels que QVT et Atlas sont essentiellement focalisés sur des transformations de type « modèle vers modèle ». Cette catégorie de transformation consiste à prendre en entrée un modèle source et des règles de transformation et de produire un modèle cible. Un besoin naturel est apparu par la suite, celui de pouvoir simplement générer du code à partir d'un modèle et d'effectuer des transformations de type « modèle vers texte ». Dans cette optique, l'OMG a proposé MOFScript [83] comme Framework de transformation de modèles MOF vers du code.

1.5 Conclusion

Dans ce chapitre nous avons présenté les concepts importants liés à l'ingénierie dirigée par les modèles (IDM) qui propose de travailler à un niveau d'abstraction élevé via des modèles pour faciliter la conception des systèmes complexes. Les modèles manipulés sont construits grâce à des métamodèles (section 1.2) décrivant un ensemble de concepts spécifiques à un domaine. Nous avons positionné l'IDM comme étant une approche permettant de concevoir des applications en séparant la logique métier de l'entreprise, de toute plateforme technique. Cela est possible grâce à des transformations automatiques de la description de plateformes abstraites et indépendantes (PIM) vers la description de plateformes concrètes et spécifique (PSM) (section 1.3).

Nous avons ensuite présenté les transformations de modèles (section 1.4) qui fournissent un moyen efficace pour manipuler automatiquement les modèles et passer d'un formalisme à un autre. Dans ce contexte, nous avons identifié deux types de transformations : les transformations de modèles à modèles, manipulant et produisant des modèles, et les transformations de modèles vers texte, manipulant des modèles, mais produisant du code en sortie (*du texte*).

Chapitre 2

Séparation des préoccupations dans l'IDM

Sommaire

2.1 Introduction 34

2.2 Modélisation des préoccupations 35

2.2.1 Modélisation Orientée-Aspect 35

2.2.2 Multi-modélisation 37

2.3 Composition des préoccupations 37

2.3.1 Composition dans les approches de modélisation orientées aspect 38

2.3.2 Composition dans les approches de transformation de modèles 44

2.4 Conclusion 51

2.1 Introduction

La séparation des préoccupations (SoC, Separation of Concerns) est un concept présent depuis de nombreuses années dans l'ingénierie des logiciels. La séparation en préoccupations apparaît dans les différentes étapes du cycle de vie du logiciel et sont donc de différents ordres. Il peut s'agir de préoccupations d'ordre fonctionnel (séparations des fonctions de l'application), technique (séparation des propriétés du logiciel système), ou encore liées aux rôles des acteurs du processus logiciel (séparation des actions de manipulation du logiciel).

Par ces séparations, le logiciel n'est plus abordé dans sa globalité, mais par parties. Cette approche réduit la complexité de conception, de réalisation, mais aussi de maintenance d'un logiciel et en améliore le compréhension, la réutilisation et l'évolution.

Les exemples suivants illustrent quelques grandes catégories de préoccupations :

— préoccupations concernant les données, avec par exemple la persistance ou le contrôle d'accès ;

— préoccupations architecturales, avec par exemple l'interopérabilité entre applications hétérogènes ;

— préoccupations système, avec par exemple l'intégration du logiciel métier dans la plateforme d'exécution, la persistance, la sécurité, etc.

Le principe de la séparation des préoccupations au niveau modèle est identique à celui au niveau code (l'approche AOP typique [66]). L'objectif est de gérer la complexité du système. Il est important cependant de noter que les modèles eux-mêmes visent aussi le même but par l'augmentation du niveau d'abstraction. La modélisation par aspect fournit donc une haute capacité à gérer la complexité (par la séparation des préoccupations et par abstraction).

La séparation des préoccupations conduit naturellement au besoin de l'intégration des modèles. Dans un processus de développement dirigé par les modèles, l'intégration des modèles se fonde généralement sur la *composition de modèles*. La composition apparaît sous divers termes selon le contexte d'application. Il s'agit globalement d'une opération qui consiste à combiner un certain nombre de modèles

pour en créer un ou plusieurs.

Dans le développement de logiciels par aspects *AOSD : Aspect Oriented Software Development* [66] [23] [42], on parle souvent d'opération de tissage (*weaving*). Le *tissage* consiste à composer des modèles d'aspects - représentant des préoccupations non fonctionnelles appelées aussi préoccupations transverses - avec le modèle modélisant le cœur fonctionnel de l'application.

Dans le domaine des bases de données, la composition se réalise par l'opération d'intégration d'un ensemble de vues d'une base de données, ou de plusieurs schémas de bases de données hétérogènes et réparties.

En ingénierie des exigences, les fonctionnalités du système sont souvent décrites selon des points de vue différents. Il en résulte un ensemble de modèles appelés aussi vues. L'obtention d'une vue globale sur le système est réalisée à travers la *fusion* des vues partielles [95].

Dans la suite de ce chapitre, Nous introduisons les principales approches basées sur la séparation des préoccupations. Nous présentons tout d'abord les travaux issus du domaine de la modélisation par aspects (*Aspect Oriented Modelling*). Nous présentons ensuite les travaux issus de la communauté IDM sur la composition de modèles ; ces travaux réutilisent les connaissances en transformation de modèle pour définir et implémenter les opérations de composition de modèles.

2.2 Modélisation des préoccupations

Dans l'IDM, il existe plusieurs approches basées sur la séparation des préoccupations. Dans cette section nous présentons les principales approches : La Modélisation Orientée-Aspect (*Aspect Oriented Modeling*) et la multi-modélisation (*multi-modelling*).

2.2.1 Modélisation Orientée-Aspect

L'idée de la séparation des préoccupations transversales (*crosscutting concern* en anglais), c'est-à-dire l'identification de différentes préoccupations dans le développement logiciel et leur séparation en les encapsulant dans des modules appropriés,

a été appliquée par la programmation Orientée Aspects (AOP pour Aspect Oriented Programming)[66]. Une application orientée aspect se fonde sur un programme principal, appelée le *programme de base*, et un ensemble d'*aspects*. Le programme de base détermine la sémantique métier de l'application. Les aspects sont ceux qui capturent des fonctionnalités transverses de l'application. Un aspect est caractérisé par deux éléments : (1) le code ajouté et (2) la localisation. Les codes à insérer sont définis dans des gréffons (*advice* en anglais). La localisation est l'endroit dans le programme de référence où l'aspect doit être attaché "tissé". L'une des expérimentations les plus abouties de langage orienté aspect est AspectJ [68] développé par l'équipe à l'origine de l'AOP.

Dans le langage orienté aspect AspectJ [68], un processus de tissage est décomposé en deux phases. Une phase de détection qui détermine toutes les zones (appelées *points de jonction*, *joint point* en anglais) où l'aspect doit être tissé. Les point de jonction possibles sont spécifiés par une *expression de coupe* (*pointcut* en anglais). Une deuxième phase consiste à composer l'*advice* avec le programme de base aux endroits préalablement détectés, c'est-à-dire au niveau des points de jonctions. .

Avec l'arrivée de l'IDM, le paradigme orienté-aspect ne s'est plus restreint au niveau de la programmation, et il s'étend maintenant aux phases amonts du développement logiciel, par exemple au niveau des exigences, de l'analyse et de la conception [96]. L'objectif de la modélisation orientée-aspect*(*Aspect Oriented Modelling) et de décrire un système complexe par la modélisation de plusieurs préoccupations à un niveau d'abstraction plus élevé, et en les composant à travers un processus de *tissage* qui peut être vu comme une transformation "horizontales" de modèles.

On peut distinguer deux familles d'approches de modélisation par aspects [43]. La première famille s'intéresse à la modélisation des concepts propres à la programmation par aspects, comme les points de jonction, en se basant sur des extensions des langages de modélisation tel que UML. La seconde famille d'approches propose par contre de structurer les aspects à un niveau d'abstraction plus élevé (analyse/-conception).

2.2.2 Multi-modélisation

La multi-modélisation est basée sur l'utilisation des DSMLs [60], langages de modélisation spécifiques au domaine(*Domain Specific Modeling Languages*). Pour chaque préoccupation, l'approche consiste à rassembler la connaissance du domaine d'application dans un langage de modélisation dédié. Chaque DSML fournit les abstractions les plus appropriées pour chaque préoccupation impliquée. Les modèles construits à partir de ce langage ne doivent plus être uniquement contemplatifs, mais productifs (exécutables) pour permettre de générer le code d'une application en partie ou de manière complète. Du fait de sa portée restreinte, il est beaucoup plus facile de définir un langage de modélisation spécifique qu'un langage généraliste comme UML.

Pour spécifier un système complexe, les approches de multi-modélisation utilisent plusieurs modèles qui sont spécifiés en utilisant différents DSMLs. Par conséquent, chaque modèle est conforme à un méta-modèle différent. Bien que l'utilisation de différents méta-modèles permet de spécifier le système avec des concepts spécifiques au domaine, elle entraîne également une plus grande complexité pour la combinaison, la fusion et l'intégration des modèles qui sont spécifiés à différents niveaux d'abstraction et conformes à des méta-modèles différents.

2.3 Composition des préoccupations

La séparation des préoccupations conduit naturellement au besoin de l'intégration des modèles. Quand plusieurs modèles sont utilisés pour décrire un système, ils doivent être intégrés afin de produire un modèle global du système. L'intégration des modèles se fonde sur la *composition de modèles*.

Dans [31] la composition de modèles est définie par : *"Model composition is an operation that combines two or more models into a single one."*

Conformément à cette définition, on peut dire que la composition de modèle est un processus qui prend deux ou plusieurs modèles en entrée, les intègre au travers d'une opération de composition et produit un modèle composite en sortie.

Dans la suite de cette section, nous proposons un panorama des techniques de composition de modèles. Nous introduisons tout d'abord les travaux issus du domaine de la modélisation par aspects (AOM). Nous présentons ensuite les travaux issus de la communauté IDM sur la composition de modèles ; ces travaux réutilisent les connaissances en transformation de modèle pour définir et implémenter les opérations de composition de modèles.

2.3.1 Composition dans les approches de modélisation orientées aspect

Comme nous l'avons vu dans la sous-section 2.2.1, La modélisation par aspects propose de considérer les aspects au niveau conception, ce qui revient à représenter les aspects par des modèles. Dans cette perspective, deux familles d'approches sont à distinguer. La première famille propose de modéliser les programmes intégrant des aspects, ce qui revient à faire apparaître les constructions propres à la programmation par aspects comme les points de jonction dans les modèles de conception. La seconde famille d'approches propose par contre de structurer les aspects au niveau des modèles. C'est à cette seconde famille que nous nous intéressons dans la suite de cette section. Nous décrivons ainsi les approches de Clarke et al. [23], de France et al. [42] et de Whittle et al. (MATA) [106] .

2.3.1.1 Composition de modèles dans l'approche Theme

L'approche Theme [23] porte sur le domaine de l'AOSD au niveau de la phase d'analyse des exigences avec *Theme/Doc*, et au niveau de la phase de conception avec *Theme/UML*.

Theme/UML est utilisé pour produire une décomposition d'un système en *Themes*. Un *Theme* correspond à un élément de conception permettant l'encapsulation d'une fonctionnalité ou d'une préoccupation transverse. Il est représenté par un paquetage stéréotypé par « theme ». Il peut être combiné avec d'autres *Themes* à laide d'une relation de substitution appelée 'bind' qui exprime la composition entre deux Themes. Un Theme est représenté par un paquetage paramétré. Chaque

paramètre correspond à une classe et à l'ensemble des méthodes sur lesquelles la fonctionnalité doit être tissée. La partie structurelle du système est souvent représentée dans Theme/UML par des diagrammes de classes, tandis que les diagrammes de séquence sont utilisés pour décrire le comportement.

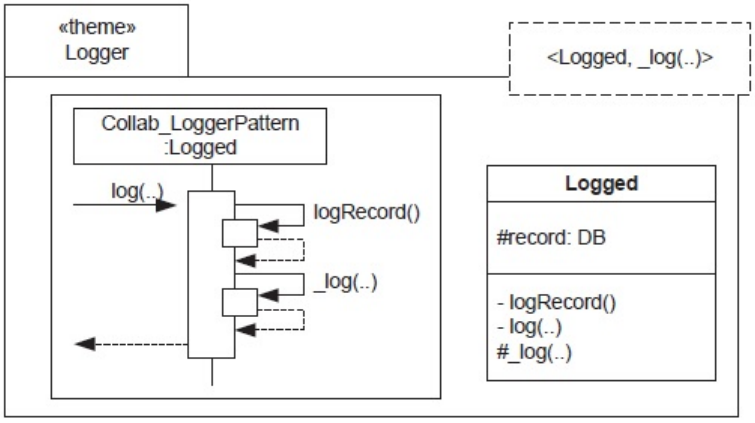


FIGURE 2.1 – Représentation de la fonctionnalité Logged dans Theme/UML [7]

La figure 2.1 représente le Theme *Logger*, exemple classique de préoccupation transverse (traçe). Étant donné que toutes les opérations du système doivent être tracées, toutes les opérations des Themes doivent être complétées par un comportement d'authentification (*logging*). Cet aspect a été défini comme une fonctionnalité générique pour être en mesure de concevoir le comportement de *logging* séparément des opérations qui nécessitent une étape d'authentification. L'aspect *Logger* comporte deux paramètres de template : la classe *Logged* et la méthode *_log*. Le tissage du Theme de base CMS *(Course Management System)* avec le Theme *Logger* utilise une relation de composition de type 'bind' (Figure 2.2). Cette relation permet de substituer le paramètre *Logged* par les classes *Person*, *Student* et *Professor* définies dans CMS, et de substituer la méthode *_log* par leurs méthodes respectives *register*, *unregister* et *giveMark*.

La composition de modèles dans Theme/UML est spécifiée par une relation appelée *relation de composition* qui permet d'identifier les parties identiques dans les Themes à composer, et de spécifier comment ces parties doivent être composées. Trois types de composition sont possibles : (i) la fusion, (ii) la substitution

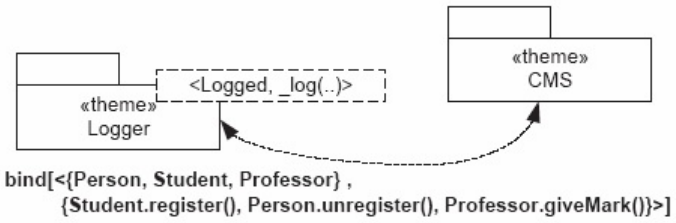


FIGURE 2.2 – Composition des Themes Logger et CMS

'bind', (iii) le remplacement 'override'. La fusion est utilisée pour la composition de deux Themes de base. La substitution est surtout utilisée pour la composition d'un Theme de base avec un Theme d'aspect. La relation de remplacement est utilisée quand le comportement décrit par un Theme doit être remplacé par un nouveau comportement décrit par un autre Theme.

L'intérêt majeur de l'approche Theme porte sur la méthodologie qu'elle propose. En effet, la décomposition du modèle de conception en 'Themes' facilite la compréhension du système, et permet de résoudre les problèmes de conception tels que la dispersion de la modélisation d'une exigence dans plusieurs éléments de modélisation, ainsi que l'enchevêtrement de plusieurs exigences dans une même unité de conception. Cependant, un inconvénient de cette approche est le manque d'outils et de formalisation des mécanismes de composition.

2.3.1.2 Composition de modèles dans l'approche de France et al.

L'approche de modélisation par aspects AAM (Aspect-Oriented Architecture Models) [42] permet de gérer la composition de *modèles d'aspect* (les préoccupations transverses modélisées), avec un modèle de base appelé *modèle primaire* (*primary model* en anglais) qui représente le cœur fonctionnel de l'application. Les modèles d'aspect sont modélisés en utilisant des *diagrammes templates*, qui sont décrits par des paquetages paramétrés inspirés d'UML 2.0.

La composition d'un modèle d'aspect avec un modèle de base exige en premier lieu une étape d'instanciation du modèle d'aspect. Ceci est fait en substituant les paramètres définis dans l'aspect générique par les valeurs spécifiques de l'application. Le modèle d'aspect instancié est appelé *modèle d'aspect spécifique à un contexte*.

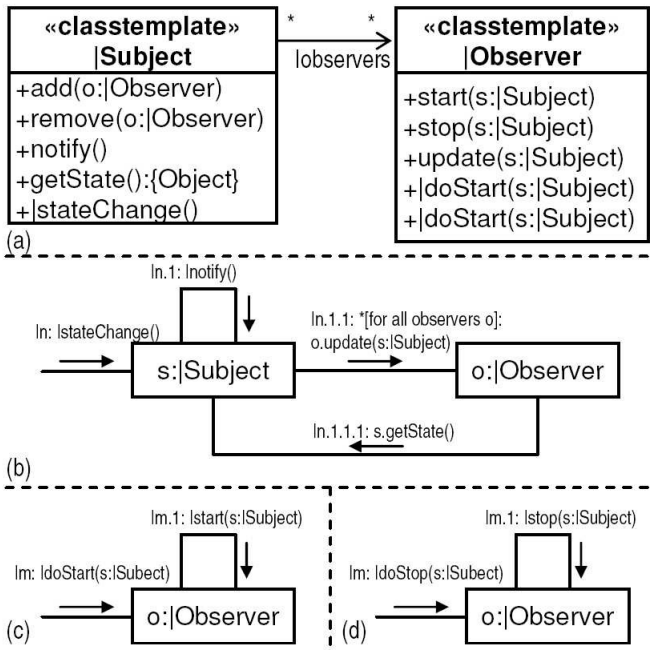


FIGURE 2.3 – Le modèle d'aspect "Observer" en utilisant la notation France et al. [96]

L'association suivante instancie le modèle d'aspect "Observer" de la figure 2.3 et produit un modèle d'aspect spécifique à un contexte, représenté sur la figure 2.4.

```
(|Subject,BookCopy); (|Observer, BookManager);
(|stateChange(),borrowCopy()); (|doStart(s:|Subject),buyBook());
(|stateChange(),returnCopy()); (|doStop(s:|Subject),discardBook());
(|observers, bookManagers);
```

Le modèle d'aspect spécifique à un contexte est ensuite composé avec un modèle de base. Dans [90], les auteurs introduisent une technique de composition basée sur un algorithme de composition et un ensemble d'actions élémentaires appelées *directives de composition*. Ces directives servent à la résolution de conflits entre éléments (par exemple, lorsque deux classes ont le même nom). Ils permettent également d'effectuer des opérations, soit sur les éléments des modèles telles que la création, la suppression d'un élément, soit sur les modèles d'aspects, en spécifiant par exemple l'ordre de composition des modèles d'aspects avec le modèle de base.

L'approche AAM est une des approches existantes les plus avancées. Le point

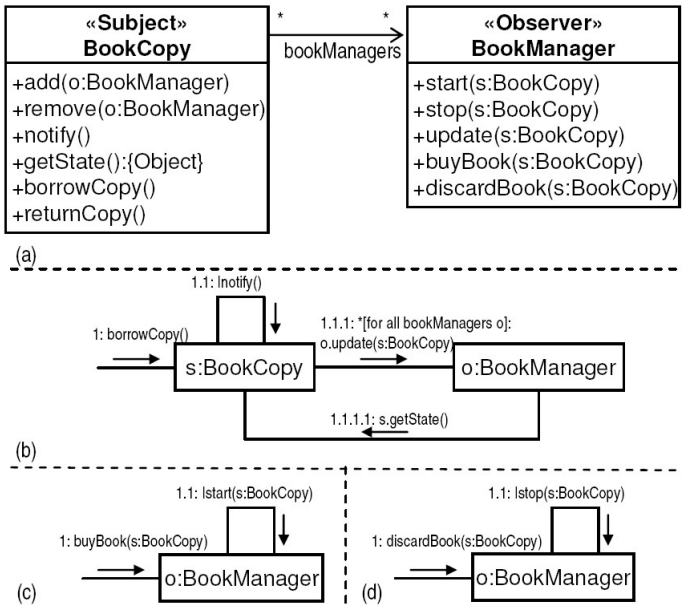


FIGURE 2.4 – Le modèle d'aspect spécifique à un contexte en utilisant la notation France et al. [96]

fort de cette approche est de proposer des mécanismes de composition bien définis et implémentés avec le langage Kermeta [96]. Nous pouvons tout de même signaler que les mécanismes de composition sont restreints aux modèles statiques.

2.3.1.3 Composition de modèles dans l'approche MATA

L'approche MATA (*Modeling Aspects using a Transformation Approach*) [106] est une approche de composition de modèles d'aspect basée sur les techniques de transformation de modèle. Plus précisément, elle utilise un langage de transformation à base de règles de réécriture de graphes. L'opération de composition nécessite l'utilisation de deux types de modèles, un modèle de *base* et un modèle *aspect*. un modèle d'aspect est défini comme une combinaison de deux parties dépendantes : un patron et une spécification de composition. Le patron est utilisé pour détecter un emplacement dans le modèle de base où les spécifications de composition seront appliquées.

Les spécifications de composition sont définies par l'utilisation de trois types d'annotations représentés par les stéréotypes suivants : *create*, *delete* et *context*. Le

stéréotype *create* est utilisé pour annoter les éléments qui vont être ajoutés dans le modèle de base, alors que les éléments marqués par le stéréotype *delete* vont être supprimés du modèle de base. Le stéréotype *context* est utilisé pour éviter d'appliquer un stéréotype à plusieurs éléments dans le cas où un élément est annoté par un de ces stéréotypes et contient d'autres éléments.

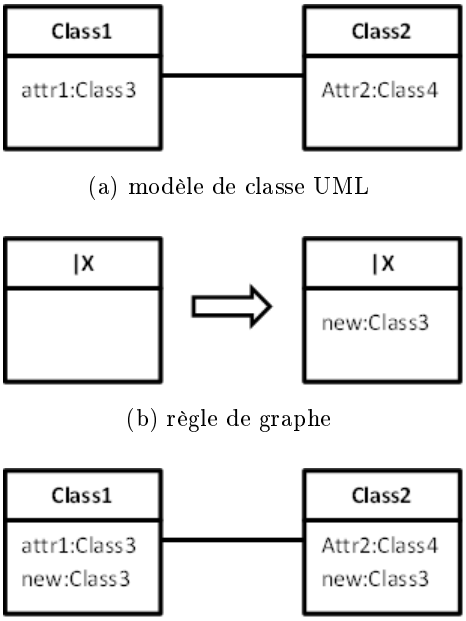
Le processus de composition avec MATA se fait en deux étapes : d'abord un motif décrit par le patron de l'aspect est recherché dans le modèle de base, puis on procède à la modification de ce motif selon la spécification de composition.

L'outil support de MATA est développé sous l'environnement RSM (Rational Software Modeler) d'IBM. Un prototype a été mis en application et un certain nombre de cas d'études ont été modélisés avec ce prototype. MATA utilise l'outil AGG pour l'exécution des règles de transformation de graphe. Les utilisateurs peuvent définir un ensemble d'aspects et l'outil produit un modèle composé de ces aspects avec le modèle de base. L'utilisateur peut également définir un ordre de composition des aspects dans le cas où un aspect doit se composer avant les autres. Si un ordre n'est pas défini, l'outil choisit un ordre par défaut.

MATA considère la composition d'aspects comme un cas particulier de transformation de graphes. Cette transformation est définie par des règles de réécriture de graphe. Une règle prend en entrée un modèle de base M_b et un modèle d'aspect M_a et produit par fusion le modèle composéM_{ab}. Elle se compose de deux parties : la première partie définit un patron qui spécifie un ensemble de points de coupure définissant des emplacements dans le modèle M_b où les nouveaux éléments doivent être ajoutés. La deuxième partie spécifie les éléments et la manière dont ces éléments doivent être ajoutés au modèle M_b. La figure 2.5 décrit un exemple simple d'application d'une règle définie par un patron.

La figure 2.5a représente le modèle de classe UML à transformer, la figure 2.5b décrit la règle de transformation et la figure 2.5c présente le résultat d'application de cette règle sur le modèle initial. Selon cette règle, le patron correspond à toute classe UML. Pour chaque classe, l'attribut new est ajouté.

La particularité de cette approche est que les règles de transformation de graphe sont définies en utilisant la syntaxe concrète du langage de modélisation. Cette pro-



(a) modèle de classe UML

(b) règle de graphe

(c) modèle UML transformé

FIGURE 2.5 – Exemple de règle de transformation d'un modèle de classe UML [106]

priété distingue cette approche des approches de transformations les plus connues, par le fait que ces approches définissent la transformation au niveau du métamodèle, en utilisant la syntaxe abstraite du langage de modélisation. Ce qui facilite la tâche du modélisateur, mais ne favorise pas la réutilisation des transformations qui sont spécifiées explicitement dans les modèles d'aspect.

2.3.2 Composition dans les approches de transformation de modèles

La relation entre la composition de modèles et la transformation de modèles a fait l'objet de plusieurs travaux et projets de recherche. Nous allons présenter trois de ces approches qui nous apparaissent les plus représentatives. AMW [31], EML [93] et Kompose [37].

2.3.2.1 Composition de modèles dans AMW

L'outil AMW (*ATLAS Model Weaver*) est un framework générique de composition de modèles. Il est basé sur les principes de tissage et de transformation de

modèles pour produire et implémenter l'opération de composition [11]. AMW considère que la composition est un cas particulier de la transformation qui prend deux modèlesM_a et M_b en entrée et combine leurs contenus dans un modèle M_{ab}.

L'opération de composition dans AMW comprend deux étapes. La première étape consiste à spécifier dans un modèle de tissage (*weaving model*) les liens de composition entre les éléments des modèles d'entrée. Un lien de composition indique une relation entre deux éléments. La deuxième étape consiste à transformer les modèles de tissage en programmes ATL. Le processus de transformation d'un modèle de tissage vers le programme ATL est réalisé par une autre transformation écrite aussi en ATL. Ce qu'on appelle une transformation d'ordre supérieur HOT (*Higher Order Transformation*). La transformation générée permet à partir de deux modèles en entrée de produire le modèle composé en sortie. La figure 2.6 ci-dessous décrit le processus d'utilisation de la transformation HOT pour générer des transformations orientées composition.

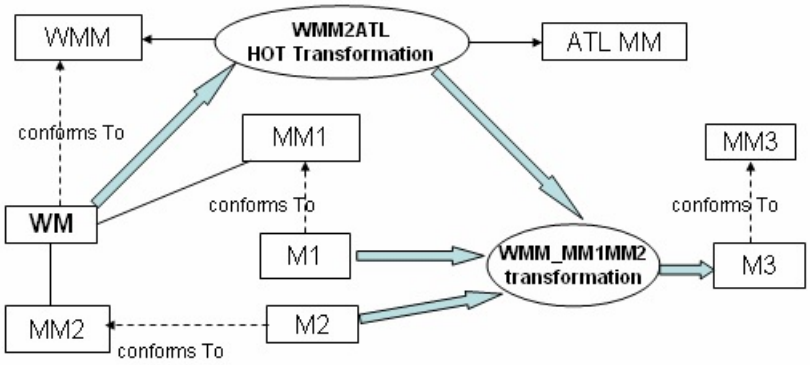


FIGURE 2.6 – Processus de génération de transformations orientées composition [31]

Le modèle de tissage a pour objectif principal la représentation des différents types de liens entre les éléments des modèles. Il n'existe pas un standard qui définisse un métamodèle de tissage capable d'exprimer toutes les sémantiques des liens de composition. Cependant, l'approche AMW a défini un métamodèle de tissage de base (*core weaving metamodel*) qui répond aux exigences communes pour la gestion des liens entre les modèles (figure 2.7). Le métamodèle de tissage de base de AMW contient les concepts de tissage de base. *WElement* est l'élément basique de tous

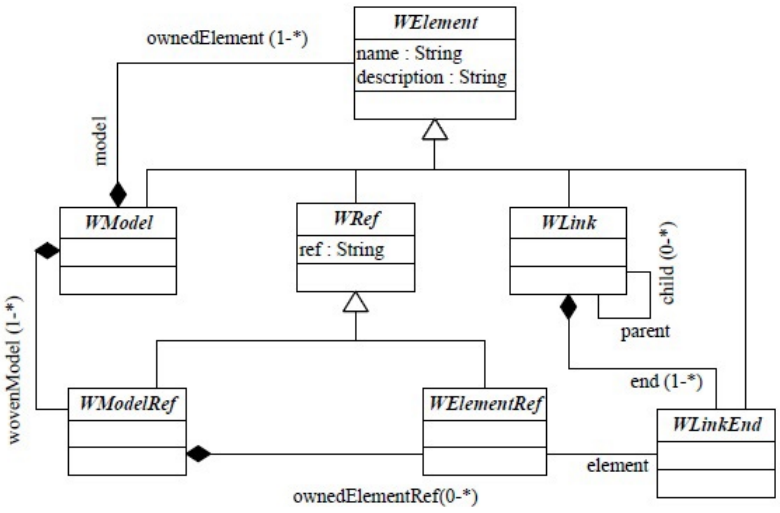


FIGURE 2.7 – Métamodèle de tissage de base [31]

les éléments du métamodèle de tissage ; *WModel* représente la racine du modèle de tissage. *WLink* représente des liens entre les éléments des modèles. *WLinkEnd* indique le type des éléments pouvant être composés.

De plus, cette approche supporte un mécanisme d'extension capable d'étendre le métamodèle de tissage de base afin de construire les nouveaux concepts de tissage spécifiques au domaine de composition de modèle (i.e. fusion, remplacement, union, etc.). C'est les concepts *WLink* et *WLinkEnd* que l'on peut étendre pour ajouter les nouveaux types de relation de composition (par exemple fusion, remplacement etc.) et également définir les types d'éléments que les nouvelles relations peuvent relier (par exemple au lieu de relier des *EObjets* de *EClass*, ce sera peut être relier les objets des classes de l'utilisateur *Livre* et *Publication*). Cette extension permet d'avoir un langage de tissage dédié à la composition de modèles.

L'approche AMW présente l'avantage d'être générique et favorise la réutilisation grâce au mécanisme d'extension du métamodèle de tissage proposé. Cependant la définition manuelle des liens entre les éléments des modèles est une tâche fastidieuse dans les applications de grande taille.

2.3.2.2 Le langage EML

Le langage EML (*Epsilon Merging Language*) proposé par [93] est un langage à base de règles *(rule-based language)* dédié à la fusion des modèles de divers métamodèles ou de différentes technologies. EML fait partie de la plateforme Epsilon (*Extensible Platform for Specification of Integrated Languages for mOdel maNagement*) supportée par l'environnement Eclipse.

Une spécification de composition en EML est définie par trois types de règles : comparaison *(match rules)*, fusion *(merge rule)* et transformation *(transform rules)*.

Une règle de comparaison permet de comparer deux instances de deux métaclasses et de contrôler leurs conformité. Le corps de la règle est divisée en deux parties : une partie comparaison *(compare)*, et une partie qui se charge du contrôle de conformité *(conform)*. A l'exécution, les *match rules* s'exécutent sur toutes les paires d'instances de métaclasses apparaissant dans les modèles source. La partie comparaison détermine si deux instances correspondent en utilisant un ensemble de critères de comparaison. La partie contrôle de conformité s'applique uniquement aux instances qui ont satisfait la condition spécifiée par la partie comparaison.

Une règle de fusion *(merge rule)* spécifie le comportement de fusion de deux instances d'éléments de modèles, ces instances ayant été déterminées comme correspondantes et conformes á la fois. Autrement dit, l'entrée de l'exécution des règles de fusion est les paires d'instances catégorisées par l'exécution des règles de comparaison. Une règle de fusion est définie par un nom, les types des métaclasses source, et une liste d'éléments que la règle devra produire dans le modèle cible. Le corps de la règle définit la spécification du contenu de l'élément cible créé.

Une règle de transformation *(transform rule)* spécifie la manière de transformer les éléments pour lesquels aucun élément correspondant n'a été identifié dans le modèle opposé. La structure d'une règle de transformation est similaire à celle d'une règle de fusion.

Nous illustrons cette approche par un exemple décrivant un scénario de fusion de deux diagrammes de classes conformes au métamodèle UML1.4. (figures 2.8a et 2.8b). Le modèle composé obtenu par la fusion des deux modèles source est présenté

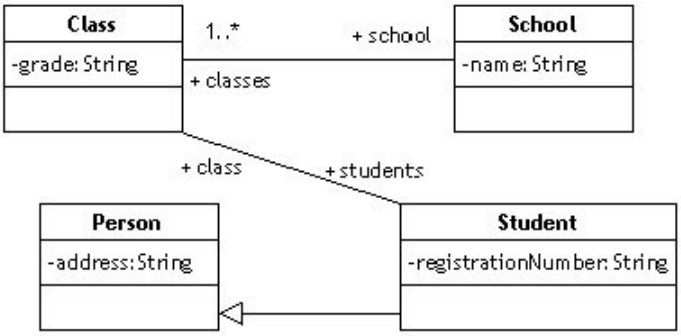
dans la figure 2.8c. Dans le modèle composé, certaines classes sont spécifiées par le stéréotype *« merged »* ce qui signifie que ces classes ont été obtenues par la fusion des deux classes correspondantes dans les modèles source. Alors que d'autres classes, spécifiées par les stéréotypes *« right »* ou *« left »*, ont par contre été obtenues par transformation de classes définies uniquement dans l'un des modèles source.

Le listing 2.1 présente un extrait en EML du programme de fusion.

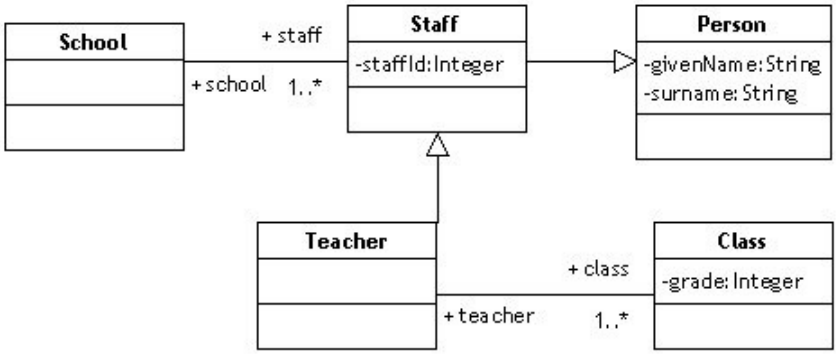
Listing 2.1 – Extrait des spécifications EML de fusion de deux modèles [93]

```
rule MatchClasses
match l: Left!Class
with r: Right!Class
compare
        return l.name = r.name and l.namespace.matches(r.namespace)↩
            ;

conform
        return l.isAbstract = r.isAbstract;

rule ClassWithClass
merge l: Left!Class
with r: Right!Class
into m: Merged!Class
        m.name := l.name;
        m.namespace := l.namespace.equivalent();
        m.feature := l.feature.includeAll(r.feature).equivalent();


rule ClassToClass
transform source:Left!Class
to target: Right!Class
        target.name := source.name;
        target.namespace := source.namespace.equivalent();
        target.feature := source.feature.equivalent();
```

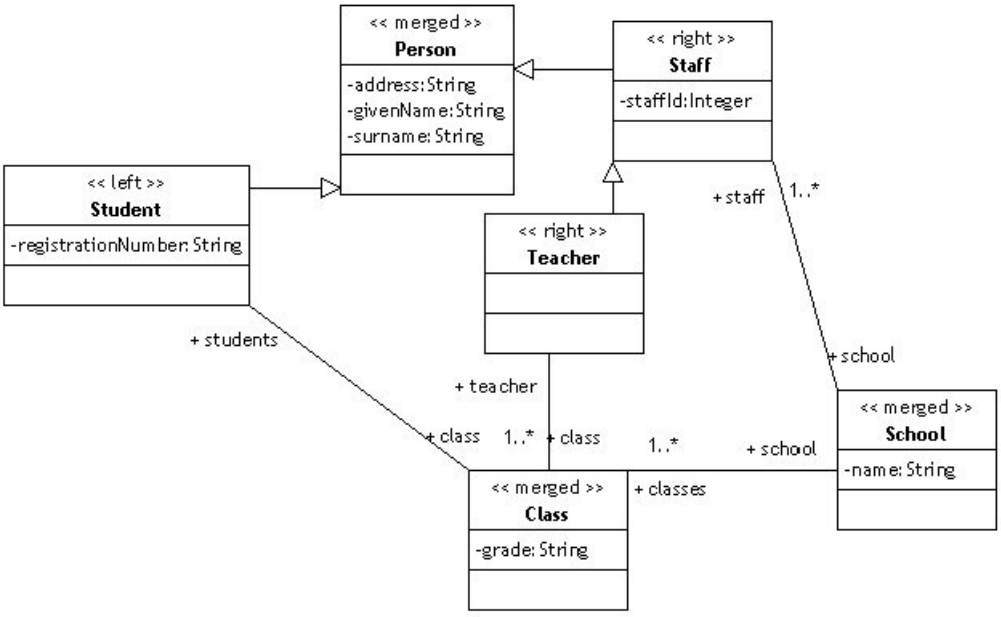
La règle de comparaison *MatchClasses* retourne vrai si et seulement si les deux classes sont soit concrètes, soit abstraites à la fois et leurs nom et leurs espaces



(a) modèle source 1



(b) modèle source 2



(c) le modèle composé

FIGURE 2.8 – Exemple de fusion avec EML [93]

de noms sont les mêmes. La règle de fusion *ClassWithClass* crée, dans le modèle composite résultat, une classe dont le nom et l'espace de noms viennent de la classe de gauche, les propriétés sont l'union des propriétés de deux classes en entrée. La règle de transformation *ClassToClass* crée l'objet destination dont le nom, l'espace de nom et les propriétés proviennent de l'objet source.

Dans EML, les règles ont la capacité d'étendre les autres règles par la déclaration extends. Cette capacité permet à l'utilisateur de réutiliser des librairies de règles déjà construites et de ce fait, d'être capable d'ajouter ses sémantiques de composition supplémentaires sans réécrire tous les codes de sa règle.

2.3.2.3 Kompose

Kompose est un outil de composition implémentant l'approche de composition de modèles basé sur les signatures [38] [37]. Les modèles composés sont en Ecore. Le type de composition est la fusion. Notons que Kompose a commencé historiquement par la perspective de modélisation orientée aspect, donc il a distingué les modèles primaires aux modèles d'aspect. Cependant, selon ses auteurs, cette distinction n'a pas de signification spéciale sauf pour avoir une bonne vision conceptuelle de la séparation des préoccupations.

Kompose a un langage de composition de modèle permettant de décrire des spécifications de composition appelées directives de composition. Le concept central de ce langage est le Composeur *(Composer)*. Un composeur représente une opération de composition effectuée sur deux métamodèles.

La structure d'un composeur comprends : un métamodèle primaire en entrée, un métamodèle aspect en entrée, le nom du métamodèle composite en sortie et l'ensemble des directives de composition.

Kompose fournit deux types de directives de composition : *pre-merge directive* et *pos-merge directive*. Les *pre-merge directives* sont appliquées sur les métamodèles avant de les composer, les *post-premege directives* sont appliquées sur le métamodèle composite avant de le produire. Les premières spécifient les modifications simples sur les modèles en entrée telles que le renommage, la suppression ou l'ajout d'éléments

afin de forcer ou empêcher la fusion. Les deuxièmes réconcilient le modèle fusionné pour qu'il soit fiable et cohérent.

2.4 Conclusion

Dans ce chapitre nous avons présenté une étude des approches de composition de modèles représentatives du domaine. Il n'y a pas encore de standard relatif à la composition de modèles, mais la composition de modèles a été abordée dans de nombreux travaux de recherche.

Chapitre 3

Traçabilité dans les transformations de modèles

Sommaire

3.1 Introduction 55

3.2 Utilisations des traces 56

3.3 Stockage des informations de traçabilité 57

3.3.1 Liens de traçabilité intégrés dans les modèles sources/cibles . 57

3.3.2 Liens de traçabilité stockés dans un modèle externe 58

3.4 La capture des liens de traçabilité 58

3.4.1 Liens de traçabilité explicite 58

3.4.2 Traçabilité implicite 60

3.5 traçabilité de modèles vers texte 60

3.5.1 Méta-modèle de trace de modèles vers texte 61

3.5.2 Génération d'une trace de modèles vers texte 62

3.6 Approches de traçabilité 62

3.6.1 Traçabilité pour l'ingénierie des exigences 62

3.6.2 Traçabilité pour les modèles 70

3.6.3 Traçabilité pour les transformations 73

3.7 Etude comparative des approches de traçabilité 75

3.7.1 Représentation 75

3.7.2 Outillage 76

3.7.3 Analyse d'impact 77

3.7.4 Evolutivité (Scalability) 78

3.7.5 Synthèse 79

3.8 Conclusion 81

3.1 Introduction

La traçabilité dans la conception de logiciels a longtemps été réduite à la traçabilité des besoins des utilisateurs. Elle consistait à mettre en œuvre des sortes d'associations entre les besoins (requirements) des projets et les artefacts de conception et de réalisation des logiciels.

Il n'existe pas une définition universelle de la traçabilité. Dans [51] la traçabilité est définie comme : *« l'effort de description et de suivi de la vie d'une exigence, dans les deux directions en amont et en aval (c.-à-d. depuis ses origines, à travers son développement et sa spécification, jusqu'à son déploiement et son utilisation, et aussi à travers toutes les périodes d'évolution et d'itération) »*.

Le glossaire standard de l'IEEE de la terminologie d'ingénierie logicielle [61] définit la traçabilité par : *« La traçabilité permet d'établir les degrés de parentés entre les produits d'un processus de développement, notamment les produits liés par une relation de prédécesseur-successeur ou de maître-subordonné »*.

Cette forme de traçabilité appelée souvent ingénierie des besoins [3] sert principalement à documenter la satisfaction de l'implémentation des systèmes vis-à-vis des besoins des utilisateurs. Cela se fait au moyen de liens définis entre les besoins et les artefacts d'implémentation.

Dans le domaine de l'ingénierie des modèles, plusieurs définitions ont été utilisées dans la littérature. [86], Paige et al. décrivent la traçabilité comme : *" the ability to chronologically interrelate uniquely identifiable entities in a way that matters. [...] Traceability refers to the capability for tracing artifacts along a set of chained [manual or automated] operations."*.

Aizenbud-Reshef et al. définissent la traçabilité par *"any relationship that exists between artifacts involved in the software-engineering life cycle."* [3].

En IDM, à chaque fois qu'une transformation de modèles a lieu, des éléments sont consommés pour en produire de nouveau. Pour garder des informations sur les éléments qui en ont produit d'autres, il est important de conserver des liens entre les éléments utilisés et ceux produits par une transformation, appelés lien de traçabilités. Évidemment, En IDM, plusieurs relations de traçabilités existent et

selon les domaines, ces relations peuvent prendre plusieurs sens.

Dans cette thèse, nous nous concentrons uniquement sur les chaînes de transformations de modèles. La section 3.2 présente les applications pour lesquelles les traces de transformations de modèles sont utilisées. les techniques généralement employées pour stocker les informations de traçabilité sont montrées dans la section 3.3. Nous présentons dans la section 3.6 un ensemble représentatif des approches de traçabilité. A travers les sections 3.4 et 3.5, nous nous attachons à présenter, les procédés mis en oeuvre pour la génération des traces pour les transformations de modèles à modèles et de modèles vers texte.

3.2 Utilisations des traces

La génération et la conservation des liens de trace est une étape primordiale mais elle n'est pas une finalité en soi. Ces informations doivent être inspectées et exploitées. Nous présentons ici les applications pour lesquelles les traces de transformations de modèles sont utilisées.

— *Analyse d'impact* : L'analyse d'impact est une des utilisations de la trace de transformations de modèles la plus classique. L'analyse d'impact est une technique utilisée pour identifier les parties d'un programme qui doivent être modifiée si un concept est modifié ou supprimé [9]. En effet, chaque modification dans la spécification ou dans les besoins d'un projet informatique existant peut impacter différentes parties du programme déjà produit. Ces parties doivent donc être identifiées pour pouvoir modifier le programme avec moins d'efforts. Dans ce contexte, la trace est représentée par un lien reliant les différents éléments de modèles les uns aux autres. Il devient alors plutôt aisé de regarder quels sont les éléments qui vont être impactés par une modification.

— *Analyse de couverture* : Elle permet de déterminer lesquelles des exigences sont totalement implémentées dans le système et lesquelles ne le sont pas.

— *Protéger contre "Gold plating"* : Elle fournit un mécanisme qui permet de s'assurer que chaque artéfact dans le système correspond en fait à une exi-

gence. Autrement, les artéfacts supplémentaires peuvent augmenter le risque et le coût du projet.

— *Analyse des compromis* : Quand plusieurs choix d'implémentation existent, la traçabilité devra faciliter l'analyse de compromis *(trade-off analysis)* en permettant la comparaison entre les différentes répercussions de chaque option.

3.3 Stockage des informations de traçabilité

Dans [94], Kolovos et al. présentent les deux approches généralement employées pour la gestion de la traçabilité dans un environnement basé sur les modèles. la première approche consiste à stocker les informations de traçabilité dans le modèle lui même, en utilisant de nouveaux éléments, comme par exemple des stéréotypes ou des attributs. L'autre consiste à stocker les informations de traçabilité dans un modèle externe. Pour profiter de leurs avantages, les auteurs proposent d'utiliser les deux approches présentées. Les liens de traçabilités sont donc conservés dans un modèle externe, mais peuvent être composés à la demande avec les modèles d'entrée et de sortie de la transformation.

3.3.1 Liens de traçabilité intégrés dans les modèles sources/cibles

Cette approche propose d'annoter les modèles d'entrée et/ou de sorties pour profiter des informations de trace directement sur les artefacts manipulés par la transformation. Par exemple, pour représenter un raffinement dans un modèle UML, une association stéréotypée « refines » lie les deux éléments concernés [58]. Cette approche est populaire auprès des modeleurs parce qu'elle représente les liens de traçabilité par des éléments de modèle visuels et donc facilement compréhensible par les humains. Cependant, cette approche peut uniquement représenter des liens de traçabilité intra-modèles. En effet, il est difficile de tisser un lien entre deux éléments de modèles conformes à des méta-modèles différents en embarquant le lien de trace dans l'un ou l'autre des modèles. En plus, si un modèle est impliqué dans plusieurs transformations, l'ajout systématique de liens de traçabilité entraîne sa

pollution, ce qui peut nuire à sa lisibilité et à sa compréhension.

3.3.2 Liens de traçabilité stockés dans un modèle externe

Dans cette approche, les liens de traçabilité sont stockées dans un modèle dédié, *modèle de trace*, référençant les éléments des modèles d'entrée et de sortie. Cela nécessite que chaque élément possède un identifiant unique et persistant, tel que l'identifiant xmi.id du MOF [80], afin d'éviter les ambiguïtés. Cependant, sans outils dédiés permettant une interrogation ou une visualisation de la trace, celle-ci est assez peu compréhensible par un humain. Cette façon de procéder a pour avantage d'être plus simple à gérer tout en laissant les modèles sources et destination de la transformation inchangés (*clean*).

Dans cette perspective, tout outil de transformation qui supporte des transformations avec plusieurs modèles en sortie sera capable de fournir de la traçabilité sans qu'il soit au préalable nécessaire de définir ou d'étendre les constructions du langage ou moteur de transformation. Ces approches supposent l'existence d'un méta-modèle de traçabilité sur la base à la quelle sera généré le modèle de trace.

3.4 La capture des liens de traçabilité

Le méta-modèle de trace utilisé est le point central d'un mécanisme de traçabilité, mais il est également important de s'intéresser à la façon dont la trace sera produite durant l'exécution de la transformation. Dans cette section, nous présentons les différentes techniques permettant d'obtenir des informations de traçabilité. Ces techniques reposent sur l'utilisation des liens de traçabilité explicite et des liens de traçabilité implicite d'une transformation.

3.4.1 Liens de traçabilité explicite

On parle de lien de traçabilité explicite lorsque l'on spécifie explicitement les liens de traçabilité qui seront construits pendant la transformation. Les règles de transformation sont donc modifiées pour gérer la trace au moment de leurs exécutions. Cette technique est notamment mise en œuvre dans [64].

Le listing 3.1 présente un exemple de règle de transformation ATL dans laquelle la production d'un lien de traçabilité a été insérée (ligne 11 à 18). La ligne 2 montre que le modèle de trace est explicitement défini comme modèle de sortie de la transformation. La règle *A2BPlusTrace* (ligne 4 à 19) produit un élément t de type B (ligne 8) à partir d'un élément s de type A (ligne 6). L'élément t porte le même nom que l'élément s (ligne 9). Dans cette règle de transformation, les instructions permettant la construction d'un lien de trace *traceLink* ont été insérées aux lignes 11 à 14 et aux lignes 16 à 17. Les lignes 11 à 14 construisent un lien de trace de type *TraceLink* en précisant le nom de la règle tracée *A2BPlusTrace* et en pointant t comme élément de destination. Les lignes 16 et 17, quant à elles, pointent l'élément s comme élément source.

Listing 3.1 – Production d'un lien de trace ATL

```
1  module Src2DstPlusTrace;
2  create OUT : Dst, trace : Trace from IN : Src;
3
4   rule A2BPlusTrace {
5                   from
6                           s : Src!A
7                   to
8                           t : Dst!B (
9                                   name <- s.name
10                          ),
11                          traceLink : Trace!TraceLink (
12                          ruleName <- 'A2BPlusTrace',
13                          targetElements <- Sequence {t}
14                          )
15                  do {
16                          traceLink.refSetValue('sourceElements',
17                          Sequence{s});
18                  }
19          }
```

Les lignes de code insérées dans les transformations pour produire des liens de trace sont ajoutées manuellement, ce qui rend l'opération fastidieuse et limite la

maintenance et l'évolution de la transformation. Pour automatiser l'insertion de ces fragments de code, Jouault propose dans [64] l'utilisation de transformation d'ordre supérieur (HOTs-High Order Transformation). Ces transformations considèrent le langage de transformation de modèle comme un simple modèle (conforme au méta-modèle du langage de transformation) qu'il est possible de manipuler. Ainsi, il est possible d'ajouter, supprimer ou modifier les concepts d'une transformation en utilisant une autre transformation de modèle.

L'ajout de la construction de la trace directement dans le code des transformations permet d'ajouter les fragments de code uniquement pour certaines règles de transformation en créant un type de lien particulier et ainsi orienter la production de la trace. Cependant, cette technique est particulièrement longue à mettre en œuvre manuellement et requiert une bonne connaissance de la transformation à tracer.

3.4.2 Traçabilité implicite

Par opposition à la traçabilité explicite où la production de la trace est directement ajoutée dans la transformation à tracer, la traçabilité implicite permet la production des liens de trace sans intrusion dans la transformation. Ainsi, la transformation ne requiert pas de modification particulière pour pouvoir bénéficier du support de la trace, celle-ci étant produite directement par le moteur de transformation.

Dans [108], Yie et al. ont proposé une manière alternative pour produire la trace durant l'exécution d'une transformation ATL. Cette fois, ce sont les liens de traçabilités implicite qui sont visés. Les informations nécessaires à la construction de la trace sont récupérées directement à partir du moteur de transformation à partir du bytecode ATL produit.

3.5 traçabilité de modèles vers texte

Les traces produites par les transformations de modèles vers texte permettent de tisser un lien entre des modèles et une structure textuelle. Les méta-modèles de trace utilisés pour les transformations de modèles à modèles ne fonctionnent plus.

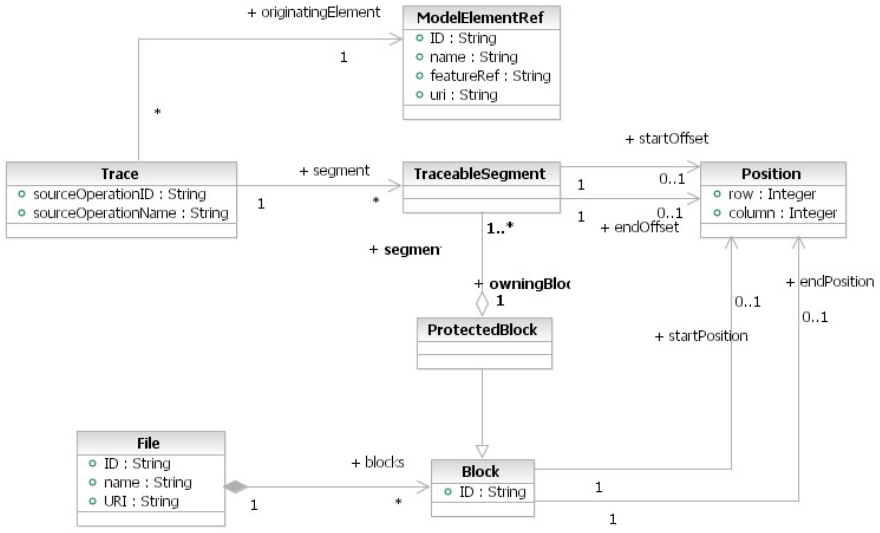


FIGURE 3.1 – Méta-modèle de trace de modèles vers texte [84]

Il faut donc utiliser un formalisme dédié pour pouvoir gérer ces liens de traces.

3.5.1 Méta-modèle de trace de modèles vers texte

Une transformation de modèles vers texte consiste à associer à un concept d'un méta-modèle d'entrée un bloc de lignes de codes. À titre d'exemple, la figure 3.1 présente le méta-modèle de trace de modèles vers texte proposé dans [84].

Dans ce méta-modèle, les deux structures, modèles et textes, sont clairement distinguées. La méta-classe *ModelElementRef* représente une référence à un élément de modèle, alors que la méta-classe *File* représente une référence au fichier texte généré. Un fichier est composé de plusieurs *Blocks* de texte commençant et se terminant à des *Positions* particulières spécifiées par leur ligne (*row*) et leur colonne (*collumn*). Un lien de trace *Trace* relie à un élément de modèle *ModelElementRef* un segment de texte *TraceableSegment* tout en conservant le nom de la règle de transformation exécutée. Les blocs de lignes peuvent être protégés, *ProtectedBlock*, c'est-à-dire, que certains segments tracés sont protégés des changements éventuels et ne peuvent pas être modifiés par l'utilisateur.

3.5.2 Génération d'une trace de modèles vers texte

Dans le contexte de transformations de modèles à modèles, des lignes de codes sont insérées (manuellement ou automatiquement) pour générer le modèle de trace à l'exécution de la transformation. Cependant, dans le contexte de transformations de modèles vers texte, les langages de transformation ne permettent pas de générer des modèles, ce qui empêche l'utilisation de cette technique.

La méthode employée pour la construction de modèles de trace consiste à récupérer directement dans le moteur de transformation les informations dont la trace à besoin. Cependant, l'efficacité de cette technique dépend étroitement de la structuration (*correct*) de la transformation en règles. Si la transformation n'a pas été correctement "découpée" en règles, on peut se retrouver, dans des cas extrêmes, avec une trace ne comportant qu'un seul lien de traçabilité entre tous les éléments des modèles et tous les blocs d'un fichier.

3.6 Approches de traçabilité

Dans [44], Ismênia et al. ont proposé une classification des approches de traçabilité. Dans cette section, nous allons présenter un ensemble représentatif des approches de traçabilité les plus discutées dans la littérature. Ces approches sont classées en trois catégories : *La traçabilité pour l'ingénierie des exigences*, *La traçabilité dans l'approche modèle* et *La traçabilité pour les transformations*.

3.6.1 Traçabilité pour l'ingénierie des exigences

Dans [51], Gotel et Finkelstein définissent la traçabilité comme la capacité à décrire et à suivre une exigence tout le long du cycle de vie du processus de développement : de sa spécification jusqu'à son déploiement et son utilisation. Dans les sous-sections suivantes, nous présentons quatre approches de cette catégorie.

3.6.1.1 Les modèles de référence

Ramesh et Jarke [88] ont conduit une étude empirique dans plusieurs entreprises afin d'étudier les meilleures pratiques dans le domaine de la traçabilité des exigences. Le résultat de cette étude empirique est l'obtention d'un ensemble de *modèles de référence* décrivant l'aspect type de lien de traçabilité entre différents artéfact. Les participants de l'étude ont été classés en deux catégories distinctes suivant leur niveau de pratique en matière de traçabilité. Ils sont désignés par des *simples utilisateurs (low-end user)* ou des *utilisateurs avancés (high-end user)*.

Les auteurs de cette étude ont adopté un système simple de quatre types de liens de traçabilité pour classifier leurs observations empiriques (figure 3.2).

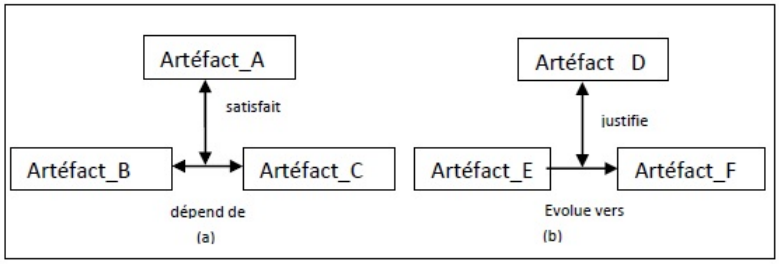


FIGURE 3.2 – Types de liens de traçabilité

Le premier groupe de liens est relatif aux livrables des processus de développement. En d'autres termes, ils décrivent la nature des relations entre les artéfacts.

Dans la figure 3.2a, l'artéfact A définit une contrainte ou un but qui devrait être *satisfait* par un ou plusieurs artéfacts plus bas (Artéfacts B, C). L'artéfact à satisfaire implique une *dépendance* entre les artéfacts B, C. Ainsi, il y a deux types de liens relatifs aux artéfacts de la figure 3.2a : les liens de *satisfaction* et de *dépendance*. Ils sont souvent utilisés par les simples utilisateurs de traçabilité.

Le deuxième groupe de liens de traçabilité est relatif aux processus de développement (qui produisent les artéfacts). Ils sont capturés en regardant l'historique et l'évolution du processus lui-même et ne peuvent pas être capturés à travers les artéfacts.

Dans la figure 3.2b, les types de lien d'évolution ont une direction temporelle :

l'artéfact E à gauche *évolue* vers l'artéfact F de droite par une action dont le raisonnement (*justification*) est capturé dans l'artéfact D. Ainsi, les deux types de liens de processus sont des liens *d'évolution* et de *raisonnement*. Les utilisateurs avancés de traçabilité utilisent régulièrement les types de lien appartenant à ces deux types.

Pour réaliser leurs modèles de référence, un métamodèle est présenté. Il a été validé par les participants à l'étude qui ont confirmé que la plupart des informations relatives à la traçabilité peuvent être capturées dans un simple métamodèle, qui peut être considéré comme les primitives d'un langage basique pour catégoriser et décrire des modèles de traçabilité. Il comporte trois méta-classes :

— *« Stakeholder »* représentant les acteurs impliqués dans les activités de développement et de la maintenance des logiciels ;

— *« Source »* représentant la source d'information : document, réunion, norme, etc. ;

— *« Object »* représentant les objets à tracer, c'est-à-dire toutes les entrées et sorties du processus du développement. Il peut s'agir de besoins, modèles, composants, décisions, etc..

Ce métamodèle est présenté par la figure 3.3. Les intervenants « Stakeholder » peuvent jouer un rôle sur les artefacts de modélisation ainsi que sur les traces elles-mêmes. Les sources documentent les artefacts de modélisation, et les intervenants gèrent les sources.

Chaque entité et lien du métamodèle peut être instancié et spécialisé , afin de créer des modèles de traces spécifiques pour les organisations ou les projets qui veulent mettre en place un suivi des évolutions des besoins. Ces modèles doivent répondre à six questions élémentaires, représentant six dimensions des informations de traçabilité :

1. *Quelle information est tracée (What)?* Dans le modèle, les *Objects* représentent les entrées et les sorties du processus de développement. Ils peuvent être des Besoins, Composants Systèmes, Décisions, etc. La traçabilité entre ces éléments est représentée par le lien *Traces-To*. Par exemple, un lien de dépendance entre deux éléments (un Besoin et une Décision) peut être représenté par une instanciation du lien *Traces-To*.

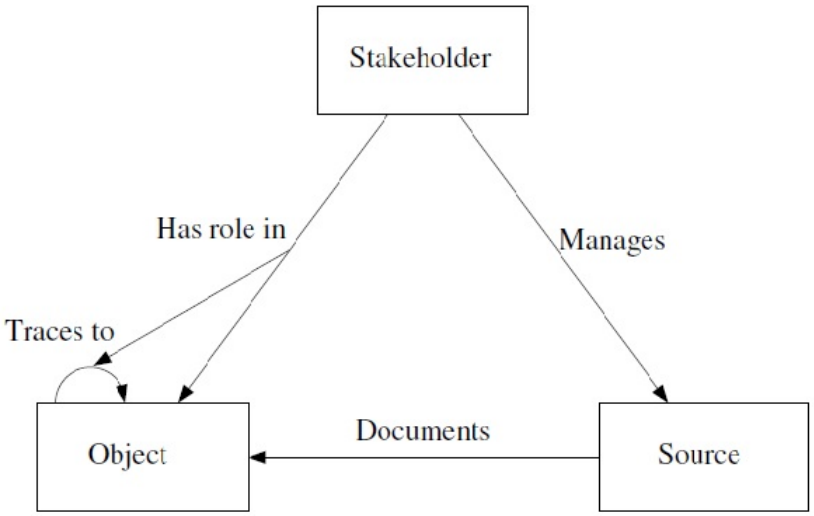


FIGURE 3.3 – Métamodèle de traçabilité [88]

2. *Qui sont les intervenants qui jouent différents rôles dans la création, la maintenance et l'utilisation des divers Objets et des liens de traçabilité les reliant (Who) ?* Dans le modèle, les *Stakeholders* représentent les acteurs impliqués dans le processus de développement. Ils peuvent être : Directeurs, Analystes, Concepteurs, etc. Ils jouent différents rôles dans la création et l'utilisation des divers Objets et les liens de traçabilité.

3. *Où l'information est-elle représentée (Where) ?* Tous les objets sont documentés par des *Sources*, qui peuvent être soit des média physiques comme un document, soit des abstractions plus ou moins intangibles comme des personnes ou des procédures non-documentées. Les *Sources* peuvent être des documents (Spécification de besoins, par exemple), des Réunions, Appels Téléphoniques, etc. Les intervenants *(Stakeholder)* gèrent les sources ; ils les créent, les maintiennent et les utilisent.

4. *Comment l'information est-elle représentée (How) ?* Comme nous l'avons déjà évoqué, les *Sources* peuvent être physiques ou intangibles. Elles peuvent aussi être plus ou moins formelles. Certaines sources, comme un document de spécification des besoins, peuvent être présentées sous forme textuelle, alors

que d'autres telles que des documents de conception peuvent être représentées de différentes manières (graphique et/ou textuelle).

5. *Pourquoi tel objet a été créé, modifié, ou pourquoi a-t-il évolué (Why) ?* La raison de la création, ou de la modification des différents objets exprimés par la spécialisation et/ou l'instanciation de la méta-classe *Object*. Chaque instance doit être ensuite liée à l'objet dont elle justifie la présence par une spécialisation d'un lien *Traces-to*.

6. *Quand l'information a-t-elle été capturée, modifiée, ou quand a-t-elle évoluée (When)* ? Toutes les informations temporelles concernant les éléments du modèle doivent être stockées comme des attributs. Par exemple, la fréquence, la durée ou la date doivent être représentées.

A partir de ce méta-modèle, plusieurs modèles de référence sont présentés. Le modèle de référence destiné aux simples utilisateurs est composé de quatre éléments : les exigences, les procédures de vérification, les composants du système et les systèmes externes. Ils sont liés par des liens représentant la satisfaction, dérivation, dépendance, etc. Les modèles définis destinés aux utilisateurs avancés sont beaucoup plus riches. Il en existe quatre :

— Le *Requirements management submodel* permet la gestion des exigences, facilite la compréhension, la capture, le suivi et la vérification de celles-ci.
— Le *Rationale subModel* permet de décrire la spécification, l'élaboration, la décomposition et la modification des différents artefacts (exigences, modèles,...). C'est là où les problèmes et les conflits dûs aux interprétations différentes, aux intérêts contradictoires et aux points de vue différents vont être détectés.
— Le *Design/Allocation submodel* est utilisé pour désigner toute activité impliquée dans la création des artéfacts, y compris la réalisation.
— Le *Compliance verification SubModel* permet de s'assurer que chaque exigence est correctement implémentée dans le système développé.

Les travaux présentés dans [88] proposent donc un noyau de méta-modélisation permettant de créer plusieurs modèles de référence adaptés aux types d'utilisateurs. De cette étude, les auteurs concluent que les modèles de traçabilité ne peuvent pas

être totalement génériques, et que chaque entreprise et projet a des besoins tellement spécifiques en terme de traçabilité qu'il est souvent nécessaire de créer ses propres modèles.

3.6.1.2 Une approche de traçabilité à base d'évènement (EBT)

Dans [24], les auteurs proposent une approche basée sur la notification d'évènements, *Event-Based Traceability (EBT)*, afin de résoudre les problèmes liés à l'évolution des informations de traçabilité. Les changements des besoins sont classés en sept catégories (*create*, *inactivate*, *modify*, *merge*, *refine*, *decompose* and *replace*) et des événements sont relevés selon le type.

Dans cette approche les exigences et les artéfacts sont reliés via un mécanisme de publication d'évènements «publisher-subscriber». Au départ, tous les artéfacts doivent s'enregistrer sur le serveur d'évènements à travers leur gestionnaire d'abonnement. Si une exigence est modifiée, les évènements de notification sont publiés dans un serveur d'évènements et les notifications sont envoyées à tous les abonnés reliés à cette exigence.

L'architecture de l'approche EBT est donnée à la Figure 3.4. Elle est composée d'un gestionnaire d'exigences «Requirements Manager», un serveur d'évènement «Event Server» et un gestionnaire d'abonnement «Subscriber Manager». Ces trois composants sont connectés en utilisant un mécanisme standard de communication.

— Le gestionnaire d'exigences traite les exigences et est responsable du déclenchement des évènements de changement.
— Le gestionnaire d'abonnement est chargé de recevoir les notifications d'évènement et de les traiter d'une manière appropriée à l'artéfact géré et le type de message reçu.
— Le serveur d'évènement est principalement responsable de la gestion des abonnements, de recevoir les notifications de changement et la transmission de messages d'évènement personnalisés au gestionnaire d'abonnement des artéfacts à charge.

En résumé, l'approche EBT propose une méthode de traçabilité basée sur la notification d'évènements. Cette technique est applicable même dans un environne-

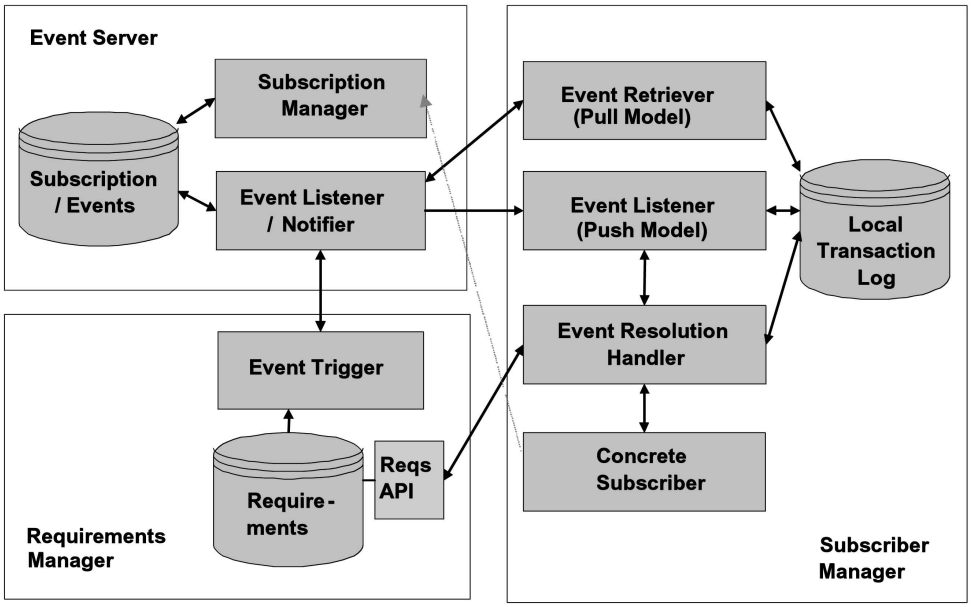


FIGURE 3.4 – Architecture de l'approche EBT

ment de développement hétérogène et distribué. La méthode prend également en charge la gestion de projet par la mise à jour et la maintenance des artéfacts du système. Cependant, cette approche se concentre sur la création manuelle des liens de traçabilité plutôt que sur leur maintenance.

3.6.1.3 Liens entre modèles et exigences

Dans [5], Almeida et al. proposent un *framework* pour gérer les relations entre les exigences et les modèles qui les réalisent. Puisque les techniques dirigées par les modèles définissent différents niveaux d'abstraction associés aux modèles tel que les modèles indépendants des plates-formes (PIM) et les modèles dépendants des plates-formes(PSM), Les exigences sont tracées tout au long de ces niveaux. Une notion de conformité *(conformance)* est proposée entre les modèles et les exigences, qui réifie la réalisation des exigences dans un modèle. Un modèle M1 est conforme à un modèle M2 si l'ensemble des réalisations de M1 est contenu dans l'ensemble des réalisations de M2. Concrètement, des matrices, appelées *"traceability cross-table"* sont utilisées pour représenter les relations entre les exigences et tout type d'artefact

de tout niveau d'abstraction : les exigences sont en ligne et les artefacts en colonne.

3.6.1.4 Une approche orientée but (GCT)

L'approche proposée par Cleland-Huang et al. [26] vise à apporter une contribution à la gestion des besoins non-fonctionnels lors du développement d'une application.

Les auteurs présentent une approche orientée but, appelée *Goal Centric Traceability (GCT)*, s'articulant autour d'un graphe orienté, le *Softgoal Interdependency Gragh (SIG)*. Le SIG est utilisé pour modéliser les besoins non-fonctionnels et leurs relations sous la forme de buts à atteindre*(Goals)*. Les sommets de ce graphe sont donc des buts inter-connectés, par exemple : le but "bonne performance" qui veuille sur la performance du système. Les buts sont décomposés en sous buts, *subgoals*, décrivant les qualités du système désirées par les décideurs. Par exemple, le but *"bonne performance"* peut avoir comme sous-but "effectuer la recherche d'information en moins de trois secondes". Les sous-buts sont décomposés en opérationalisations, qui sont des moyens à mettre en œuvre permettant de contribuer positivement ou négativement à ses buts parents. Par exemple l'opérationalisation "indexer les données" contribue positivement aux buts précédemment cités.

le processus se décompose en quatre phases : la modélisation des buts, la détection d'impacts, l'analyse des buts et la prise de décision.

Durant la phase de *modélisation des buts*, ceux-ci sont identifiés, validés par l'ensemble des décideurs, et modélisés dans le SIG. Durant la phase de *détection d'impacts*, des liens sont établis entre les modèles fonctionnels et les éléments du SIG pouvant être impactés. Lorsqu'un changement survient, l'ensemble des éléments impactés est retourné par un algorithme exécuté sur le SIG. L'algorithme étant basé sur une approche probabiliste lui permettant d'inclure des paramètres fixés par les décideurs (hiérarchie entre les buts, etc.), cet ensemble doit être validé par le développeur. Vient ensuite la phase *d'analyse des buts*, où l'impact du changement effectué est propagé dans le SIG dans le but d'évaluer son effet global. Il s'agit d'une ré-évaluation des buts, qui produit un rapport d'analyse d'impact où les buts qui sont impactés (positivement ou négativement) sont répertoriés. Finalement, les

décideurs examinent ce rapport durant la phase de *prise de décision* et déterminent si le changement doit être réalisé ou non. A ce stade, des solutions alternatives peuvent être proposées, afin de réduire ou de supprimer des impacts négatifs dûs au changement.

Afin de valider leur approche, les métriques *"rappel"* et *"précision"* issues du domaine de la recherche d'information sont utilisés. Elles permettent d'évaluer si des traces ont été oubliées, et si de mauvaises traces ont été récupérées. L'approche a été implémentée et testée, montrant la faisabilité d'utiliser une approche probabiliste pour retrouver des liens de traçabilité dans le cadre de besoins non-fonctionnels.

Cette approche présente donc une manière de gérer les besoins non-fonctionnels, et propose de représenter leurs liaisons sous la forme d'un graphe. Ce graphe est ensuite utilisé pour effectuer de l'analyse d'impact. Afin d'évaluer la pertinence des résultats obtenus, les auteurs ont eu l'idée d'adapter les mesures issues du domaine de la recherche d'informations.

3.6.2 Traçabilité pour les modèles

Les travaux présentés dans cette approche s'attachent aux problématiques de la traçabilité dans un environnement de développement orienté modèle. Ils ont été sélectionnés car ils couvrent et apportent des contributions dans la gestion et l'utilisation des traces durant un développement orienté modèle.

3.6.2.1 Une approche de traçabilité à base de scénarios (SBT)

L'approche SBT [34] est une technique utilisée dans le domaine d'ingénierie logicielle pour la création des liens de traçabilité entre un code source et les différents artéfacts associés à un système finalisé.

Le but de l'approche SBT est de tracer la couverture des liens entre les spécifications et l'implémentation durant la phase de réingénierie ou de maintenance des systèmes. L'approche génère des liens de traçabilité par observation des scénarios de test appliqués à un système en exécution.

l'approche SBT s'appuie sur trois artefacts principaux : des scénarios de test, un

modèle décrivant le système (cas d'utilisation, diagramme de classes) et des classes d'implémentation (système ou une partie du système qui fonctionne). Ces éléments peuvent être reliés par quatre types de liens de traçabilité :

— lien de trace entre les scénarios de test et le système,

— lien de trace entre les éléments des modèles et le système,

— lien de trace entre les éléments des modèles et le scénario de test,

— lien de trace entre les éléments des modèles.

3.6.2.2 Sémantique opérationnelle pour la traçabilité (OST)

Dans [4], Aizenbud-Reshef et al. ont introduit la définition d'une sémantique opérationnelle pour les liens de traçabilité *(OST :Operational Semantics for Traceability)*. La relation de traçabilité utilise la relation de dépendance définie par le méta-modèle d'UML : la métaclasse *dependency*. L'approche définit trois problématiques auxquelles la mise en place de la traçabilité peut répondre :

— interrogation de traces par des requêtes (par exemple l'analyse d'impact ou de couverture) ;

— pouvoir suivre les liens tout au long du cycle de développement d'un projet ;

— mise à jour du système et de sa documentation.

Les auteurs définissent deux types de sémantique : la *sémantique préventive* et la *sémantique réactive*. La sémantique préventive permet de décrire ce qui ne doit pas arriver, la sémantique réactive décrit les actions qui doivent être effectuées lorsqu'un changement est effectué sur un ou plusieurs éléments, ou sur la relation de dépendance elle-même.

La sémantique opérationnelle est définie par une ou plusieurs propriétés sémantiques. Une propriété sémantique d'une relation est un triplet *(event, condition, action)*.

— *event* représente une action utilisateur sur les éléments d'une relation ;

— *condition* est une contrainte logique ;

— *action* peut être de deux types :

  — *preventive* définit une contrainte avant l'application de l'action ;

  — *reactive* définit l'action devant être effectuée après l'action, de manière

similaire aux *trigger* utilisé dans bases de données.

3.6.2.3 Maintenance basée sur des règles (traceMaintainer)

Dans [72], les auteurs présentent une approche semi(automatique) pour assurer la maintenance des liens de trace entre des modèles UML qui expriment les besoins, l'analyse et la conception des systèmes orientés objet. Les auteurs supposent que le développement se fait sous l'environnement de développement CASE (Computer Aided Software Engineering). L'approche est concernée par des changements progressifs d'un ensemble évolutif de relations de traçabilité. Elle n'est pas chargée de la création d'une série initiale de relations de traçabilité, mais elle doit assurer la maintenance des relations déjà établies. Cette approche se compose de deux phases :

1. Reconnaissance : consiste à capturer les changements élémentaires effectuées sur les éléments du modèle *(elementary changes)* et de décrire l'activité de développement *(development activity)* appliquée à chacun de ces élément, une activité de développement se compose d'une sequence de changements élémentaires.

2. Maintenance : mise à jour des relations de traçabilité associées à l'élément de modèle modifié. Cette approche est basée sur l'utilisation d'un modèle d'information de traçabilité *(TIM : traceability information model)* qui spécifie les types de relations de traçabilité qu'on peut définir sur les différents types d'artefacts du projet.

L'approche de Mäder et al. est basée sur l'utilisation d'un ensemble de règles de maintenance définissant les mises à jour de traçabilité à effectuer si une activité de développement est reconnue. Ces règles sont exprimées en XML Schema Definition (XSD). Chaque règle se compose d'un identifiant <Rule id>, un nom <Name>, une description de l'activité de développement <Description> et une ou plusieurs sections <Alternative>.

3.6.3 Traçabilité pour les transformations

3.6.3.1 Approche "Loosely Coupled Traceability (LCT)"

L'approche présentée dans [64] propose l'ajout du code à une transformation écrite en ATL pour générer automatiquement des liens de traçabilité. L'auteur considère l'information de traçabilité comme un modèle, plus précisément comme un modèle cible supplémentaire dans une transformation. Cependant, cela exige l'existence d'un méta-modèle qui décrit le modèle externe de trace, de la même manière qu'on a un besoin d'un méta-modèle qui décrit le modèle source et cible d'une transformation. Les développeurs doivent donc créer manuellement un supplément "des modèles de sortie" dans les règles de mapping, qui décrivent les éléments du modèle de traçabilité à être créé.

La Figure 3.5 montre un métamodèle simplifié de la traçabilité. Il est composé de deux méta-classes. La première représente le lien entre les éléments *(TraceLink)* tandis que la seconde représente un élément quelconque du modèle d'entrée ou de sortie *(AnyModelElement)*. Un lien de trace, définit une relation entre des éléments sources et cibles des modèles. Les liens de traces portent un attribut *ruleName* permettant de préciser quelle est la règle qui a consommé les éléments pointés par *sourceElements* et créé ceux pointés par *targetElements*.

FIGURE 3.5 – Méta-modèle de traçabilité [64]

Comme les programmes de transformation sont en réalité des modèles, une transformation ATL peut être transformée en une autre transformation ATL, transformation d'ordre supérieur ou High Order Transformation (HOT)[102]. Pour éviter l'intervention manuelle des développeurs, cette transformation peut insérer automatiquement le code qui permet de produire le modèle de traçabilité.

3.6.3.2 Un framework de traçabilité pour les transformations de modèles (FTTM)

Dans [36], Les auteurs proposent un framework implémenté dans le langage Kermeta permettant de faciliter la trace des transformations de modèles. Le méta-modèle de trace proposé est représenté en figure 3.6.

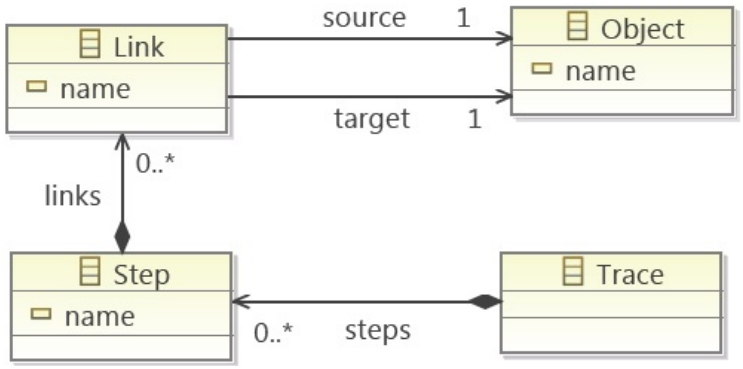


FIGURE 3.6 – Méta-modèle de trace des transformations [36]

Ce méta-modèle permet de stocker la trace de plusieurs transformations (une chaîne de transformations). La méta-classe *Trace* permet le stockage de plusieurs *Step*. Un *Step*, quant à lui, permet de conserver plusieurs liens *Link* créés pour une transformation. Un lien de trace *Link* permet d'associer seulement deux éléments *Object*, un élément source à un élément destination. Un *Object* représente un modèle ou un élément de modèle.

La figure 3.7 présente un exemple de trace pour une chaîne composée de deux transformations. Les transformations sont représentées par deux instances de la méta-classe *Step*, respectivement Step1 et Step2. La première transformation prend les éléments A_i en entrée et produit les éléments B_j en sortie. La deuxième transformation utilise les éléments B_k en entrée et génère les éléments C_l en sortie. La trace ainsi produite garde les liens entre les trois modèles de la chaîne de transformations et permet de naviguer du premier au dernier. Par exemple, il est possible de savoir que l'élément C_2 du dernier modèle produit a été créé à partir des éléments A_1 et A_2 dans le premier modèle.

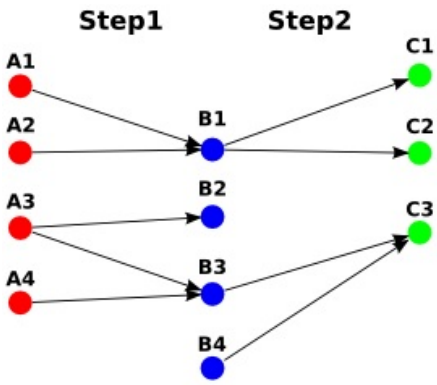


FIGURE 3.7 – Exemple de trace pour une chaîne de transformations [36]

3.7 Etude comparative des approches de traçabilité

Dans cette section nous allons présenter une étude comparative des approches de traçabilité présentées dans la section précédente. Nous commençons d'abord par décrire quelques critères qui nous ont permis d'effectuer cette comparaison.

3.7.1 Représentation

Ce critère permet de comparer les approches en terme de capacité de représentation des informations de traçabilité. Les différentes approches étudiées proposent plusieurs façons pour représenter et stocker les relations de traçabilité.

Dans [88], Ramesh et Jarke proposent des modèles de référence pour représenter différents niveaux d'informations de traçabilité et différents types de liens. ils proposent un métamodèle qui décrit la structure des modèles de référence.

Dans EBT [24], Cleland-Huang et al. proposent l'utilisation des souscriptions basées sur les événements pour représenter les informations de traçabilité.

Dans [5], Almeida et al. utilisent de simples tableaux à deux dimensions pour représenter les relations entre les besoins et les éléments de modèle.

Les approches GCT [26] et SBT [34] quant à elles utilisent une structure de graphes pour représenter les liens de trace. La première approche utilise un graphe (SIG) pour représenter les relations entre les besoins (buts) et leurs artefacts de

développement(opérationalisations). La seconde approche utilise un graphe, appelé *footprint graph*, pour représenter les informations de traçabilité.

L'approche OST [4] définit une sémantique opérationnelle de relations de traçabilité qui capture et représente les informations de traçabilité en utilisant un ensemble de propriétés sémantiques de la forme "Evénement-Condition-Action".

Dans [72], les différents types de relations de traçabilité qu'on peut définir sont spécifiée dans le modèle d'information de traçabilité *(TIM : traceability information model)*. Les relations de traçabilité établies entre les éléments du modèle conformément au modèle d'information de traçabilité sont stockées dans un référentiel qui fournit des fonctionnalités pour stocker, modifier et interroger les relations de traçabilité.

Dans [64], des modèles de traces sont générés par des transformations écrites en ATL. L'auteur considère les informations de traçabilité comme un modèle, il est donc possible d'étendre des programmes ATL pour permettre la génération de traces lors de transformations de modèles.

Dans [36], Falleri et al. proposent un framework pour la traçabilité des transformations de modèles écrites dans la langue Kermeta. Ils considèrent le modèle de traces comme un graphe biparti où les nœuds sont de deux types : des nœuds sources et des nœuds cibles.

3.7.2 Outillage

L'outillage est un critère fondamental pour la mise en œuvre d'une approche de traçabilité, non seulement, pour la gestion et la visualisation des liens de trace, mais aussi pour pouvoir raisonner sur ces informations.

La plupart des approches de traçabilité des exigences sont outillées. L'approche EBT [24] est implémenté sur le système de gestion des exigences DOORS. Les métamodèles de référence proposés par Ramesh et Jarke sont mis en œuvre en utilisant le système de gestion de base de données déductive et orientée objet ConceptBase. Ils ont été par la suite intégrés dans des outils commerciaux de gestion de la traçabilité, tels que SLATE et Tracenet [44]. Les liens de traces dans l'approche GCT [26] quant a eux sont générés en utilisant un modèle de réseau probabiliste.

Dans l'approche SBT [34], l'analyse de traces et l'interprétation des résultats sont automatisées. L'approche proposée par Mäder et al. un prototype appelé *traceMAINTAINER* est implémenté. traceMAINTAINER peut être déployé avec n'importe quel outil CASE(Computer-Aided Software Engineering) permettant la capture des événements de changement élémentaires. L'approche proposée par Jouault est implémentée en langage ATL [64]. Falleri et al. [36] ont implémenté leurs transformations ainsi que la génération de leurs traces sous l'environnement Kermeta et ont visualisé les traces grâce à l'outil de visualisation de graphes Graphviz [45]. Par contre, dans l'approche OST aucune implémentation n'est proposée.

3.7.3 Analyse d'impact

L'analyse d'impact est un critère qui vérifie si une approche permet d'identifier toutes les conséquences d'un changement sur les différents artefacts du système tout au long de son cycle de vie.

La plupart des travaux issus de l'ingénierie des besoins ont fourni des mécanismes pour l'analyse d'impact de changement. Ramesh et Jarke [88] fournissent des moyens d'analyse de l'impact des changements selon la description du "rationale submodel". Dans l'approche EBT [24], la notification d'évènements permet de maitriser le processus de gestion des changements de données. Dans l'approche GCT [26], l'analyse de l'impact des changements entre les exigences fonctionnelles et non-fonctionnelles est assurée à l'aide d'un graphe d'interdépendance *(Softgoal Interdependency Gragh, SIG)*. Dans l'approche de Mäder et al. traceMAINTAINER [72] est un outil qui fonctionne en arrière-plan pendant qu'un développeur manipule des digrammes UML (diagrammes structurels). Une fois une activité de développement est reconnue, traceMAINTAINER effectue les opérations de maintenance nécessaires aux relations de traçabilité impactées par ces changements.

Les autres approches [5, 34, 4, 64, 36], n'ont pas fourni de mécanismes pour effectuer une analyse de l'impact des changements.

3.7.4 Evolutivité (Scalability)

La taille et la complexité des projets logiciels ne cessent d'augmenter. La scalability est donc un critère important à prendre en compte : une approche de traçabilité peut conduire a la création d'un nombre très important de liens qui rend la gestion et l'exploitation de cette quantité d'informations plus difficile.

Ramesh et Jarke proposent des modèles de référence qui peuvent être utilisés dans différents niveaux de complexité. Ces modèles sont plus ou moins complexes suivant les besoins des concepteurs et la nature de l'application développée. Par conséquent, l'approche a été conçue pour être applicable sur tout type de projet.

Almeida et al. affirment qu'il est possible de fragmenter les différents modèles afin d'utiliser plusieurs tables de traçabilité [5]. Néanmoins, l'absence d'outillage pour l'approche présentée rend son application sur des projets de taille moyenne difficile [44]. Dans l'approche EBT, lorsque le projet évolue, il est difficile de maintenir une bonne performance du serveur d'événements.

L'approche GCT [26], présentée par Cleland-Huang et al. peut supporter des projets plus importants. Une manière possible de réaliser avec des systèmes plus importants, est d'explorer la possibilité de décomposer le graphe d'interdépendance (SIG) en sous-graphes.

Egyed [34] a appliqué son approche basée sur des scénarios sur plusieurs projets à grande échelle. L'auteur a pu observer que La complexité de l'analyse de traces paraît acceptable pour des projets de taille importante.

Dans [72], les auteurs considèrent que l'application de l'approche dans les grands projets n'est pas évidente. Cette catégorie de projets est généralement caractérisés par la manipulation de différents types d'artéfacts et, par conséquent, l'utilisation de plusieurs outils qui ne supportent pas la traçabilité de la même façon.

Dans [64], Jouault affirme que son approche est évolutive pour différents projets de transformation en ATL. Par contre, les auteurs de FTTM [36] n'ont pas fourni de preuve sur l'efficacité de leur approche pour des grands projets.

3.7.5 Synthèse

Nous avons vu que la génération et l'exploitation des liens de trace a fait l'objet de nombreux travaux de recherche. Afin de comparer les différents approches présentés dans les sections précédentes à travers les 4 critères que nous avons définis, nous les avons regroupées dans la table 3.1.

Nous pouvons remarquer que les travaux issus de l'ingénierie des besoins sont les plus avancés. Cette communauté est l'un des pionniers dans ce domaine, car la traçabilité constitue le moyen permettant de s'assurer que les besoins sont bien pris en compte dans le système en cours de développement. Les travaux proposés dans cette catégorie apportent en réalité une contribution sur une problématique bien précise de l'utilisation de la traçabilité, main une solution globale n'est pas proposée. Cependant, de nombreux problèmes rencontrés lors de la mise en place de la traçabilité sont soulevés : la sémantique des liens de traçabilité, les artéfacts tracés, la structure de représentation des liens, la taille des données, la maintenance des liens au cours des évolutions, le degré d'automatisation, etc.

La communauté modèle a pu exploiter les techniques présentées dans le cadre de l'ingénierie des besoins pour la gestion de traçabilité dans un contexte de développement basé sur des modèles. Les approches orientées transformation s'intéressent quant à elles uniquement à la structure (méta-modèle de traçabilité) et à la génération des liens.

Que ce soit dans le cadre de l'ingénierie des besoins ou celle des modèles, des travaux de nature différente et pour des utilisations différentes ont été proposés à travers la littérature. Cependant, très peu de travaux proposent des mécanismes permettant l'évolution automatique des liens de trace. Une approche de traçabilité doit assurer la cohérence de l'information de traçabilité en tenant en compte la mise à jour automatique de modèles de traces face à l'évolution de modèles ou de transformations.

		Représentation	Analyse d'impact	Outillage	Evolutivité
Exigences	Modèles de référence	Modèle et Métamodèle	Oui	Oui	Oui
	EBT	Souscriptions basées sur les événements	Oui	Oui	Non
	Liens entre modèles et exigences	Tableau	Non	Non	Non
	GCT	Graphe (SIG)	Oui	Oui	Oui
Modèles	SBT	Graphe (footprint graph)	Non	Oui	Oui
	OST	Evènement, conditions et actions	Non	Non	-
	traceMaintainer	Traceability Information Model (TIM)	Oui	Oui	Non
Transformations	LCT	Modèle	Non	Oui	Oui
	FTTM	Modèle	Non	Oui	Non

TABLE 3.1 – Synthèse de l'étude des travaux existants

3.8 Conclusion

Dans ce chapitre nous avons vu comment les transformations de modèles à modèles et de modèles vers texte sont tracés. Nous avons montré quelques applications classiques pour lesquelles les traces de transformations de modèles sont utilisées Nous avons discuté les différentes techniques utilisées pour capturer les liens de traçabilité lors de l'exécution d'une transformation avant de présenter quelques méta-modèles de trace utilisés par les langages de transformations les plus utilisés dans le monde de l'IDM, à savoir, ATL et Kermeta. Finalement, Nous avons présenté les approches de traçabilité les plus connues dans le domaine et nous avons donné une étude comparative succincte.

Chapitre 4

Maintenance de la traçabilité

Sommaire

4.1 Introduction 84

4.2 Un scénario d'évolution de modèles 85

4.2.1 Description des modèles 85

4.2.2 Chaîne de transformation 87

4.2.3 Scénario d'évolution 101

4.3 Processus d'évolution des liens de traces 103

4.3.1 Comparaison de modèles 104

4.3.2 Détection et classification des changements 108

4.3.3 Evolution des liens de traces 109

4.4 Validation de l'approche 110

4.4.1 Objectifs et questions de recherche 111

4.4.2 Ressource expérimentale 112

4.4.3 Résultats et discussion 113

4.5 Conclusion 114

4.1 Introduction

Le contrôle de l'évolution des logiciels exige une compréhension profonde des changements et leur impact sur les différents artefacts du système. Il est raisonnable de penser que ce phénomène s'applique aussi pour les logiciels développés en suivant une approche IDM. Avec une telle démarche se pose le problème de la cohérence entre les différents modèles impliqués dans le cycle de développement.

Dans ce chapitre, nous nous intéressons particulièrement à l'impact du changement de modèles sur les liens de traces produits par la chaîne de transformations. L'objectif de ce chapitre est de présenter dans un premier temps notre technique de traçabilité permettant la capture et la conservation des liens de traçabilité entre les modèles d'entrée et de sortie dans un processus de transformation de modèles basé sur la séparation de préoccupations [53]. Par la suite nous présentons notre approche pour la maintenance des relations de traçabilité générées au départ par la chaîne de transformations [55, 56].

Pour l'implémentation du prototype, nous avons décidé d'utiliser la plate-forme de développement open source Eclipse. Notre choix est justifié par l'existence d'un grand nombre d'outils et de technologies autour de l'IDM tels que les API graphiques comme GEF (Graphical Editing Framework) [48], les technologies de gestion de modèles comme EMF (Eclipse Modeling Framework) [101] et GMF (Graphical Modeling Framework) [40]. Au niveau du langage de transformation, le choix s'est porté sur le langage de transformation ATL [65] pour son caractère hybride. De plus, ATL est considéré maintenant comme un standard de transformation dans Eclipse et est intégré depuis 2007 dans le projet M2M [39].

Ce chapitre est structuré de la manière suivante. Dans un premier temps, nous présentons dans la section 4.2 un exemple concret permettant d'illustrer l'évolution des modèles dans un processus de transformation de modèles. Dans la sous-section 4.2.2.3, nous présentons le mécanisme que nous utilisons pour générer et conserver les liens de traces. Par la suite, en section 4.3, nous présentons en détail notre approche de maintenance de la traçabilité. Finalement, nous validons cette nouvelle approche en section 4.4 avant de conclure en section 4.5.

4.2 Un scénario d'évolution de modèles

Le scénario utilisé est un cas simplifié d'un *système bancaire sécurisé*. La figure 4.1 présente une chaine de transformations permettant de produire un *code Java* à partir de deux modèles d'entrée *Bank model* et*Security model*.

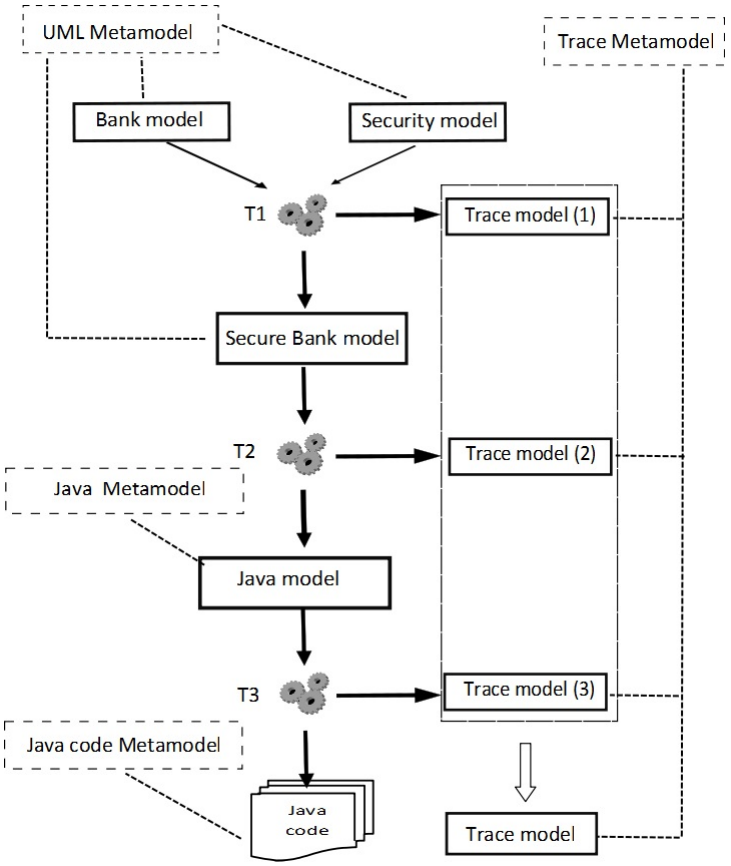


FIGURE 4.1 – Transformations d'un système bancaire sécurisé

4.2.1 Description des modèles

Nous présentons dans cette section les modèles proposés par le concepteur pour décrire un système bancaire sécurisé, *Bank model* et *Security model*. Le premier modèle représente les fonctionnalités métiers du système*(core model)*. Le deuxième modèle *Security model* représente des préoccupations non fonctionnelles, il définit une politique de sécurité permettant de contrôler l'application d'un ensemble d'opé-

rations sur un ensemble d'objets *(advice model)*.

4.2.1.1 Bank model

La figure 4.2 présente un diagramme de classes qui exprime l'aspect fonctionnel du système. Le modèle montre que nous pouvons effectuer des transactions sur des comptes clients. Deux types de transactions sont définies (1) retirer un montant d'un compte *Withdrawal*, et (2) déposer un montant dans un compte *Deposit*. Nous remarquons qu'en raison de la séparation des préoccupations, aucune politique de contrôle d'accès n'a été spécifiée dans ce modèle.

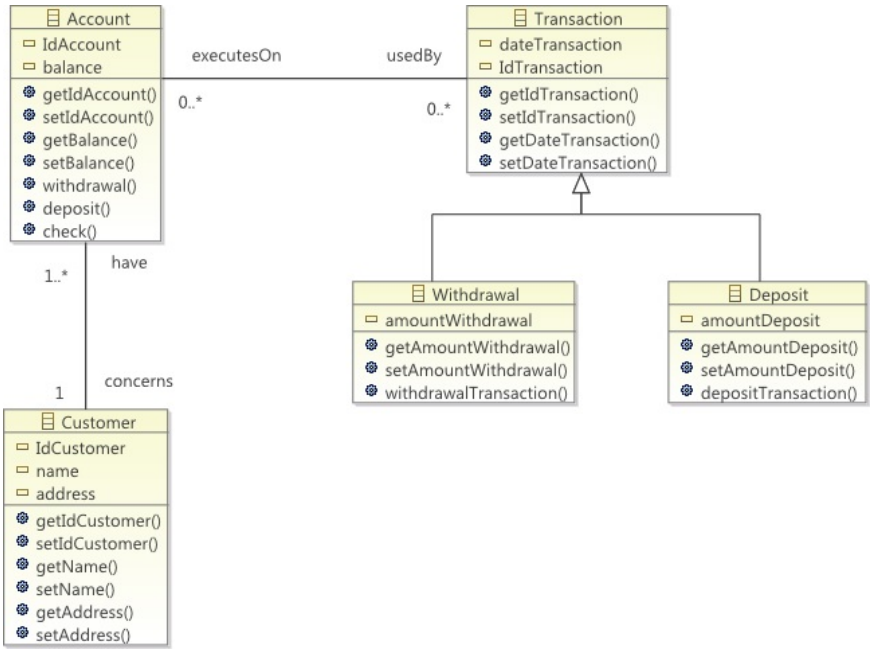


FIGURE 4.2 – Bank model

4.2.1.2 Security model

Dans ce scénario, le contrôle d'accès est également modélisée par un diagramme de classes. Comme le montre la figure 4.3, le modèle de sécurité décrit un ensemble d'objets *Object* et les autorisations *Permission* requises pour effectuer des opérations sur ces objets. Les autorisations sont affectées à des rôles *Role* et les utilisateurs du système acquièrent des autorisations en étant membres de ces rôles.

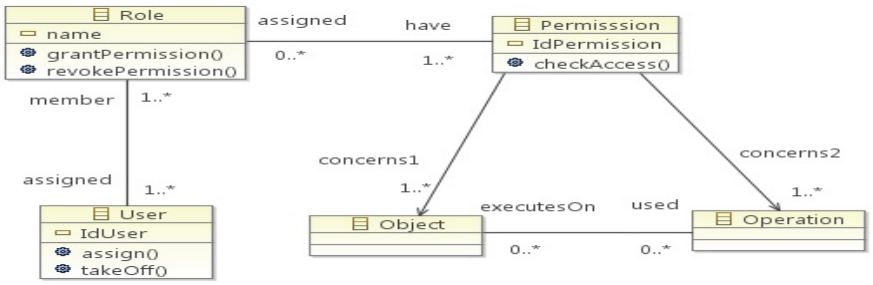


FIGURE 4.3 – Security model

4.2.1.3 Secure bank model

Ce modèle est obtenu par composition du modèle représentant les fonctionnalités métiers que le système doit assurer *Bank model* et le modèle qui présente des services de sécurité dont le système a besoin pour effectuer ses fonctionnalités métiers *Security model*. Comme le montre la figure 4.4, le modèle contient des entités métiers et des entités de sécurité qui sont impliquées dans une version simplifiée d'une application bancaire sécurisée.

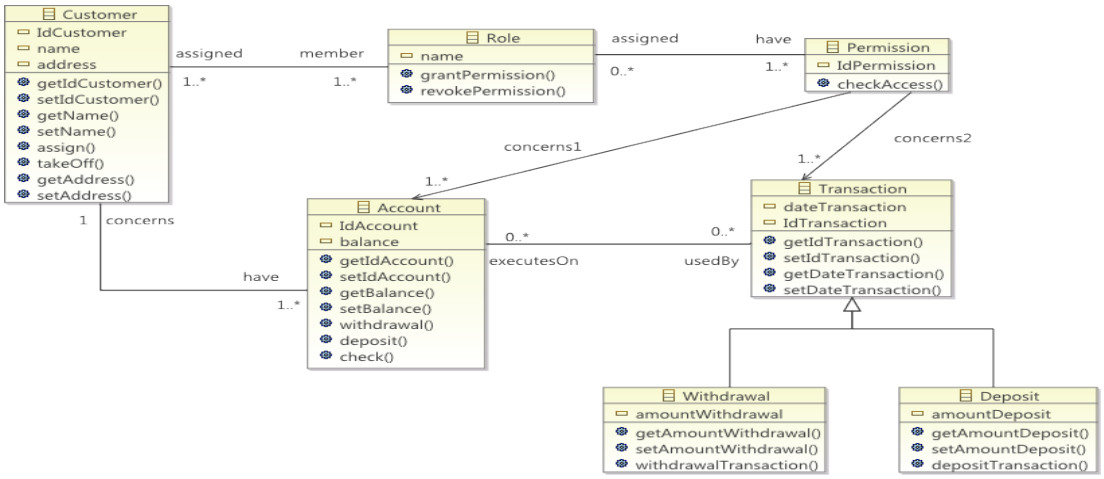


FIGURE 4.4 – Secure bank model

4.2.2 Chaîne de transformation

La chaîne de transformations comprend trois transformations :

1. La première transformation *T1* permet la composition de deux modèles UML : *Bank* et *Security*. Cette première transformation est implémentée par AMW (Atlas Model Weaver) et produit le modèle *Secure bank*.
2. La seconde transformation *T2*, est une transformation de modèles à modèles qui génère un modèle *Java* à partir du modèle *Secure bank*. Cette transformation est implémentée par ATL (Atlas Transformation Language).
3. Finalement, la troisième et dernière transformation *T3*, est une transformation de modèle vers texte qui génère le code *Java*.

4.2.2.1 Composition des modèles UML

Cette transformation permet la composition de deux modèles UML : *Bank* et *Security*. Le modèle produit par cette transformation est un modèle UML (*Secure bank model*) qui décrit le système global en représentant les fonctionnalités métiers (l'aspect fonctionnel) ainsi que les services de sécurités (l'aspect non-fonctionnel).

Les modèles source utilisés ont été conçus avec l'éditeur générique d'Eclipse/EMF. Nous avons crée des modèles UML conformément au métamodèle UML présenté dans la figure 4.5.

Une classe (*Class*) est caractérisée par un ensemble d'attributs (*Property*) et des opérations (*Operation*). Ces propriétés possèdent un type : dans le cas des attributs, il s'agit de leur type, dans le cas d'une opération, il s'agit du type de l'élément retourné. Les opérations possèdent une liste de paramètres, eux-aussi typés.

Implémentation du métamodèle de tissage

Cette étape de la réalisation consiste à définir un métamodèle de tissage. La figure 4.6 présente un extrait du métamodèle de tissage permettant la composition de deux modèles UML. Pour introduire la sémantique de tissage entre les éléments, nous avons étendu le métamodèle de AMW (Atlas Model Weaver) défini dans [30], le but étant de fournir des modèles appelés *Modèles de tissage* « *weaving models* ». Ces modèles de tissage peuvent être utilisés par la suite par une transformation d'ordre supérieur HOT (Higher Order Transformation) pour générer des transformations

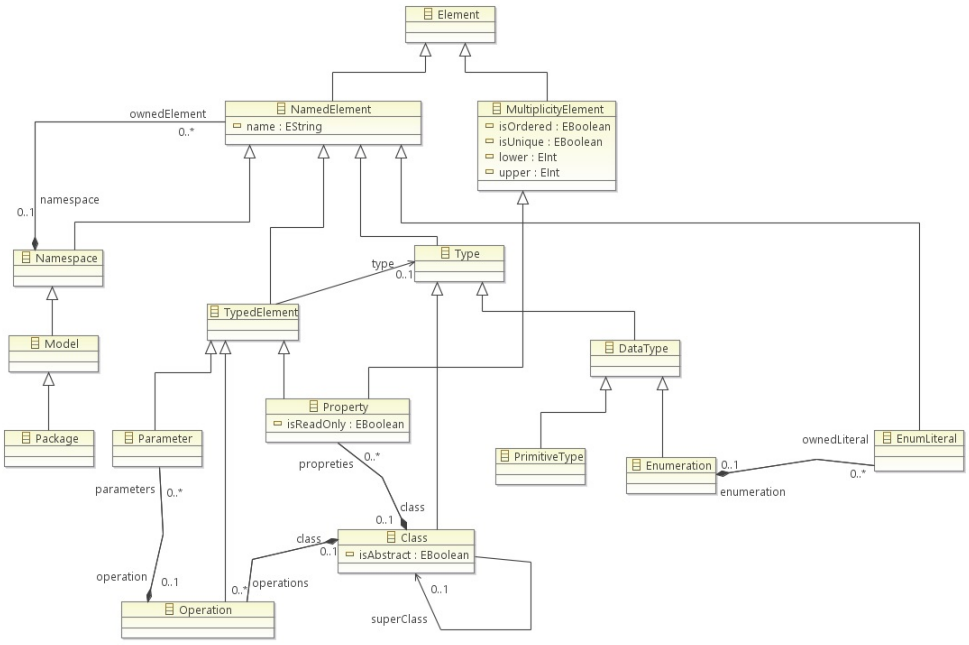


FIGURE 4.5 – Extrait du méta-modèle UML concernant le diagramme de classes.

écrites en ATL.

Nous présentons dans ce qui suit les éléments clés du métamodèle :

— La métaclasse *WeavingModel* représente le modèle de tissage qui contient la description des relations entre les deux modèles concernés par la composition *coreModel* et *adviceModel*.

— Les éléments *Class*, *Method* et *Attribute* représentent les éléments d'extrémité d'une relation de composition.

— Les références *createClass*, *createAttribute* et *createMethod* permettent de créer respectivement une classe, une propriété ou une opération.

Comme nous l'avons expliqué dans l'approche de composition de modèles AMW, l'opération de composition comprend deux étapes. La première étape consiste à spécifier les liens de composition entre les éléments des modèles d'entrée. Ces liens sont stockés dans un modèle de tissage *weaving model*.

La figure 4.7 présente un extrait du modèle de tissage qui spécifie les liens de composition entre les éléments des modèles d'entrée *bankModel (core model)* et *securityModel (advice model)*. Par exemple :

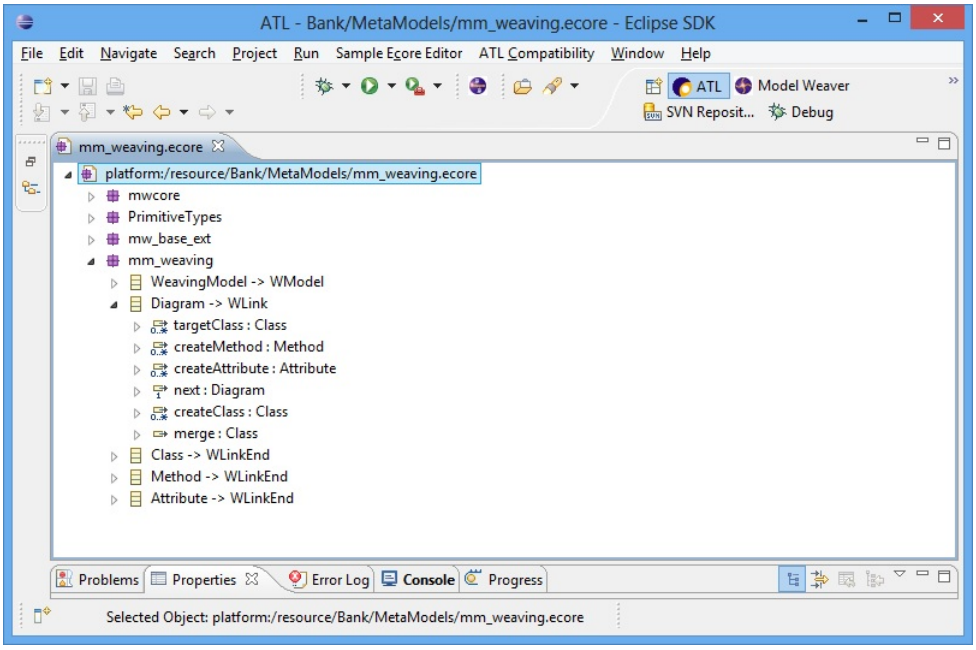


FIGURE 4.6 – Métamodèle de tissage (Weaving metamodel)

1. le lien de composition {*« createClass » Class Permission*} permet la création d'une copie de la classe *Permission* dans le modèle cible.
2. le lien de composition {*« createAttribute » Attribute member*, *« createMethod »Method assign()*, *« targetClass » Class Customer*} permet l'ajout de la propriété *member* et de l'opération *assign()* à la classe cible *Customer*.
3. le lien de composition {*« merge » Class Operation*, *« targetClass » Class Transaction*} permet de mettre en correspondance les deux classes référencées (*Transaction* et *Operation*). Ce type de lien est utilisé pour la transformation des associations entre les classes correspondantes.

Spécification des règles de composition

Une fois le modèle de tissage établi, la deuxième étape du processus consiste à appliquer des règles de composition selon la nature des relations définies entre les éléments des modèles d'entrée. Nous présentons dans cette section quelques règles définies dans le module de composition.

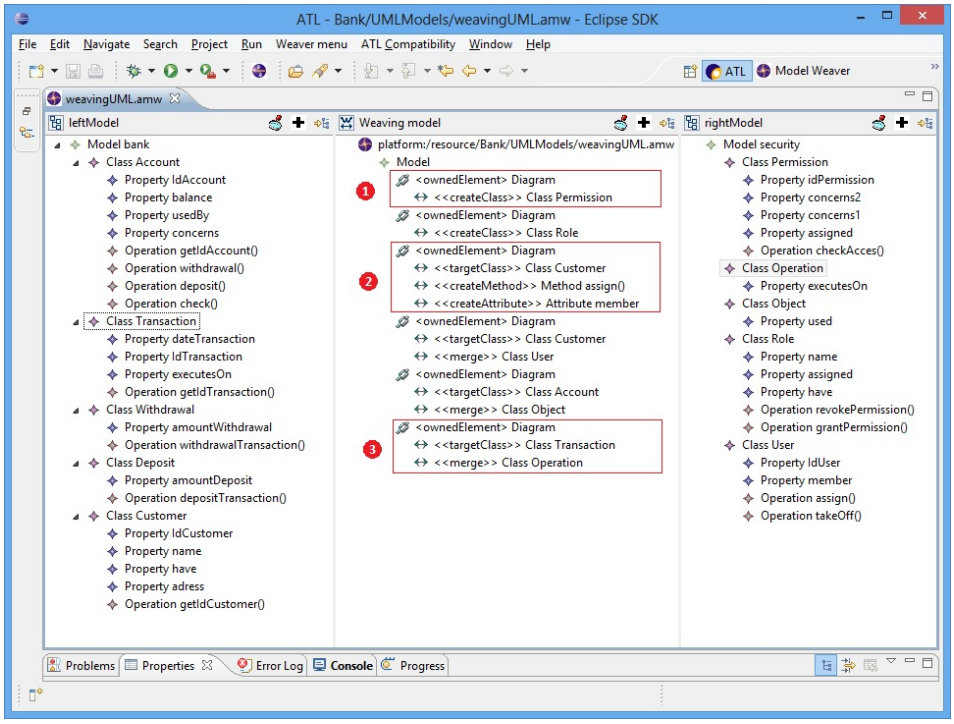


FIGURE 4.7 – Modèle de tissage (Weaving model)

Règle de création de classes

— Pour chaque classe définie dans le modèle de base (*Bank model*), on crée dans le modèle de sortie une classe :
  — avec le même nom,
  — avec les même propriétés et les même opérations,
  — avec les même relations d'association et de généralisation.

— Pour chaque relation de composition { *« createClass »*} référençant une classe définie dans le modèle d'aspect (*Security model*), on crée dans le modèle de sortie une classe :
  — avec le même nom,
  — avec les même propriétés et les même opérations,

Le code présenté au listing 4.1 ci-dessous illustre la règle de création de classes en ATL.

Listing 4.1 – Règle de création de classes en ATL

```
1  module weavingTrans; -- Module Template
2  create secureBank : UML from IN : AMW, bank : UML, security : UML;
3
4  helper context UML!Element def: isLeft : Boolean =
5    UML!Element.allInstancesFrom('bank')->exists(e | e = self );
6
7  helper context UML!Element def: isRight : Boolean =
8  AMW!Diagram.allInstancesFrom ('IN')  -> collect (e | e.createClass) -> ↩
       flatten()-> exists
9  (e| UML!Element.getInstanceById('security', e.element.ref)=self) ;
10
11 rule Class {
12          from
13                  com : UML!Class (com.isLeft or com.isRight)
14          to
15              out : UML!Class (
16                          name <- com.name,
17                          superClass <- com.superClass,
18                          isAbstract <- com.isAbstract
19                  )
20 }
```

Règle de création de propriétés

— Pour chaque lien de composition {*« createAttribute »*, *« targetClass »*} reliant une propriété du modèle d'aspect (*Security model*) et une classe cible du modèle de base (*Bank model*), on crée dans la classe référencée par {*« targetClass »*} une propriété :

— avec le même nom,

— avec le même type,

— avec la même multiplicité.

Le code présenté au listing 4.2 ci-dessous illustre la règle de création de propriétés en ATL.

Listing 4.2 – Règle de création de propriétés en ATL

```
1  rule newProperty {
2   from
3    attr : AMW!Attribute
```

```
4   to
5    out : UML!Property (
6     name <- UML!Property.getInstanceById('security', attr.element.ref).name,
7     type <-  UML!Property.getInstanceById('security', attr.element.ref).type↩
          ,
8     lower <- UML!Property.getInstanceById('security', attr.element.ref).↩
          lower,
9     upper <-UML!Property.getInstanceById('security', attr.element.ref).upper↩
          ,
10    class <-  UML!Class.getInstanceById('bank', attr.refImmediateComposite()↩
          .targetClass->first().element.ref)
11                )
12  }
```

Règle de création des opérations

— Pour chaque lien de composition {*« createMethod »*, *« targetClass »*} reliant une opération du modèle d'aspect (*Security model*) et une classe cible du modèle de base (*Bank model*), on crée dans la classe référencée par {*« targetClass »*} une opération :

— avec le même nom,

— avec le même type,

— avec les mêmes paramètres.

Le code présenté au listing 4.3 ci-dessous illustre la règle de création d'opération en ATL.

Listing 4.3 – Règle de création des opérations en ATL

```
1   rule newOperation {
2    from
3     met : AMW!Method
4    to
5     out : UML!Operation (
6      name <- UML!Operation.getInstanceById('security', met.element.ref).name,
7      class <-   UML!Class.getInstanceById('bank',met.refImmediateComposite().↩
           targetClass->first().element.ref),
8      parameters <- UML!Operation.getInstanceById('security', met.element.ref)↩
           .parameters,
9      type <- UML!Operation.getInstanceById('security', met.element.ref).type
10                )
11  }
```

Modèle UML résultat de la composition

La figure 4.8 présente le modèle UML résultant de la composition des deux modèles d'entrée, modèle de base *(Bank model)* et modèle d'aspect *(Security model)*. Ce modèle est construit par application des règles de composition décrites dans les sections précédentes. Pour des raisons de lisibilité, nous avons présenté le modèle résultat à partir de l'éditeur générique d'Eclipse/EMF.

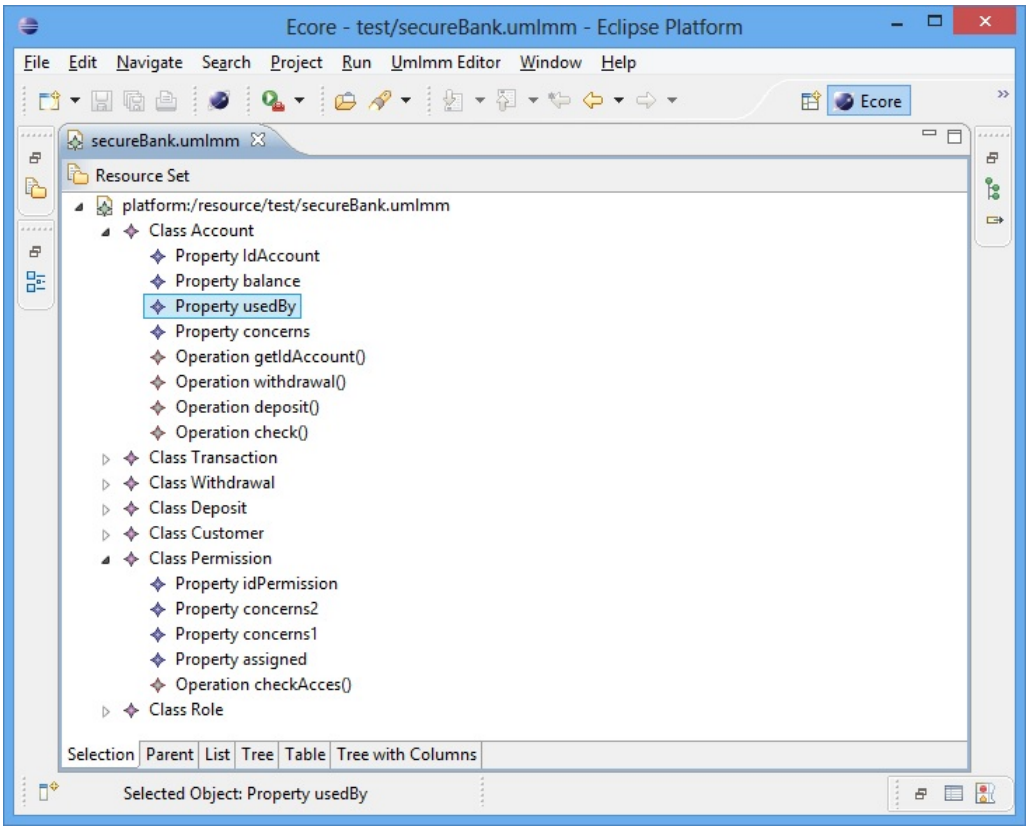


FIGURE 4.8 – Modèle UML résultat de la composition *(Secure bank model)*.

4.2.2.2 Transformation modèles à modèles

Comme cela a été présenté dans la description du langage ATL, les règles de transformation ATL sont décrites dans un programme appelé *module*. Un module de transformation est défini par un ensemble de règles destinées à transformer un ou plusieurs modèles source pour en créer un ou plusieurs en sortie. Par conséquent,

les règles de transformation sont décrites dans un module, qui, une fois exécuté, produit un modèle conforme au métamodèle JAVA à partir d'un modèle UML.

Un extrait du méta-modèle Java que nous avons utilisé est présenté dans la figure 4.9.

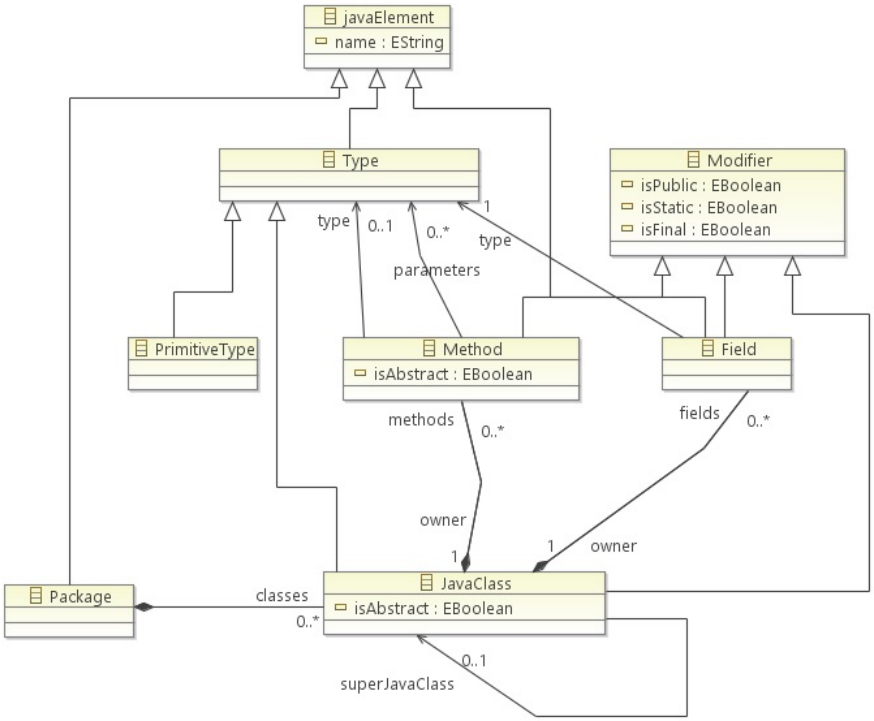


FIGURE 4.9 – Extrait du méta-modèle Java.

Une classe java peut posséder une seule super-classe. On peut y déclarer des méthodes (*Method*) et des attributs (*Field*). Les méthodes sont associées à un type de retour, ainsi qu'à une liste de paramètres typés. Les attributs quant à eux référencent un Type.

La transformation de notre modèle UML vers un modèle Java consiste en ce qui suit :

— Chaque paquetage UML donne un paquetage Java : les noms sont les mêmes mais en Java, le nom est complètement qualifié.

— Chaque classe UML donne une classe Java, de même nom, dans le paquetage correspondant et avec les mêmes *modifiers*.

— Chaque DataType UML donne un type primitif correspondant en Java de même nom, dans le paquetage correspondant ;

— Chaque attribut UML donne un attribut Java respectant le nom, le type, la classe d'appartenance et les *modifiers*.

— Chaque opération UML donne un attribut Java respectant le nom, le type, la classe d'appartenance et les *modifiers*.

Le listing 4.4 décrit un extrait du code de transformation ATL. La règle A2F (ligne 45 à 59) produit un élément *b* de type *JAVA !Field* (ligne 49) à partir d'un élément *a* de type *UML !Property* (ligne 47). L'élément *b* porte le même nom que l'élément *a* (ligne 50). L'élément *b* est déclaré dans la classe produite à partir de la classe de l'élément *a* (ligne 51). Le type de l'élément *b* est le type correspondant au type de l'élément *a* (ligne 5).

Listing 4.4 – Extrait du code de transformation ATL

```
1  module UML2JAVA;
2  create OUT : JAVA, trace: TRACE from IN : UML;
3
4  --helper context UML!Namespace
5  helper context UML!Namespace def: getExtendedName() : String =
6  if self.namespace.oclIsUndefined() then  ''
7  else if self.namespace.oclIsKindOf(UML!Model) then  ''
8  else
9  self.namespace.getExtendedName() + '.'
10 endif endif + self.name;
11
12 rule P2P {
13         from
14                 a:UML!Package (a.oclIsTypeOf(UML!Package))
15         to
16             b:JAVA!Package (
17                 name<-a.getExtendedName()),
18                         traceLink:TRACE!TraceLink (
19                         ruleName <- 'P2P',
20                         targetElts <- Sequence{b})
21                          do {
```

```
22      traceLink.refSetValue('sourceElts', Sequence{a});
23    }
24  }
25
26  rule C2C {
27          from
28                  a:UML!Class
29          to
30                  b:JAVA!JavaClass(
31                          name<-a.name,
32                          isAbstract<-a.isAbstract,
33                          superJavaClass<-a.superClass,
34                          package<-a.namespace),
35                          traceLink:TRACE!TraceLink (
36                          ruleName <- 'C2C',
37                          targetElts <- Sequence{b})
38                           do {
39      traceLink.refSetValue('sourceElts', Sequence {a});
40    }
41  }
42  ...
43  rule A2F{
44          from
45                  a:UML!Property
46          to
47                  b: JAVA!Field(
48                          name<-a.name,
49                          owner<-a.class,
50                          type<-a.type),
51                          traceLink:TRACE!TraceLink (
52                          ruleName <- 'A2F',
53                          targetElts <- Sequence{b})
54                           do {
55      traceLink.refSetValue('sourceElts', Sequence {a});
56    }
57  }
```

4.2.2.3 Modèle de traces

Pour pouvoir tracer les préoccupations fonctionnelles et non-fonctionnelles, nous avons proposé dans [53] un méta-modèle de traçabilité. Le méta-modèle de traçabilité est présenté entièrement en figure 4.10. Il est composé de huit méta-classes.

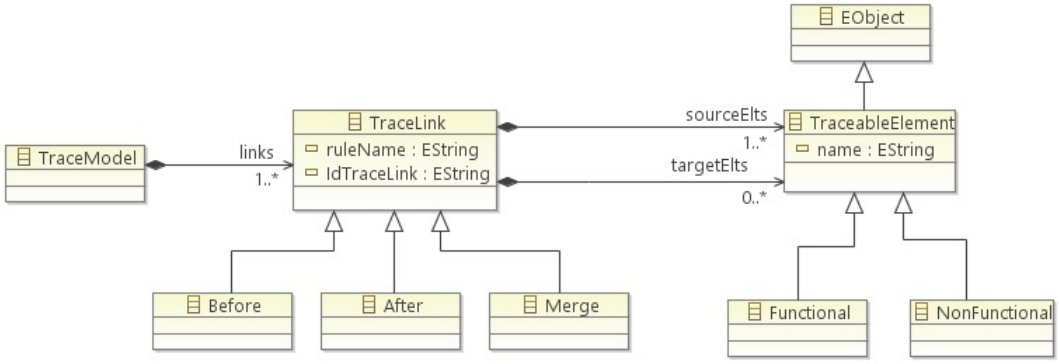


FIGURE 4.10 – Méta-modèle de traçabilité

Le méta-modèle de Jouault [64] a été pris comme base pour la construction de notre méta-modèle. Ainsi, on retrouve le même noyau structurel que pour le méta-modèle de trace présenté dans la section 3.6.3.1. La méta-classe *TraceModel* représente la racine du modèle de traces. La méta-classe *TraceLink* représente le lien entre les éléments tandis que la méta-classe *TraceableElement* représente un élément quelconque utilisé ou produit par une transformation. Les liens de traces portent deux attributs *IdTraceLink* et *ruleName*. Le premier permet de différencier les différents liens créés lors de la construction de la trace et le second permet de préciser quelle est la règle qui a consommé les éléments pointés par *source* et créé ceux pointés par *target*. L'utilisation d'une référence vers l'élément *EObject* permet de tracer n'importe quel élément de modèle sans en préciser les types. La figure 4.11 montre un exemple de liens de traces en tenant compte uniquement les méta-classes de base (*TraceLink* et *TraceableElement*).

Sur l'exemple, les classes A, B et C appartiennent au modèle d'entrée alors que les classes D et E appartiennent au modèle de sortie. Les liens de traces sont eux

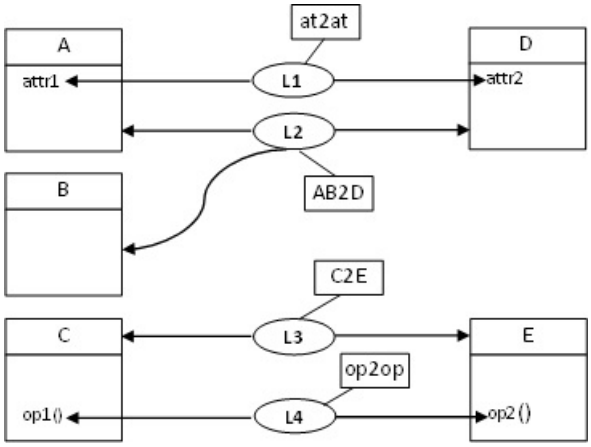


FIGURE 4.11 – Exemple des liens de traces

représentés par les éléments notés *L1*, *L2*, *L3* et *L4*. En regardant les informations contenues dans ce modèle de traces, il est possible d'affirmer à partir du lien *L1* que l'attribut *attr1* a été utilisé par la règle *at2at* pour créer l'attribut *attr2*. De même, le lien *L4* montre que l'opération *op1()* a servi à créer l'opération *op2()* à partir de la règle *op2op* . Le lien *L2* permet de dire que l'élément *D* du modèle de sortie est créé à partir des éléments *A* et *B* du modèle d'entrée et par application de la règle *AB2D*. Finalement, le lien *L3* nous montre que la classe *C* a été consommée par la règle *C2E* qui a produit la classe *E*.

Dans notre méta-modèle de traces, nous proposons de séparer les préoccupations, le concept *TraceableElement* est raffiné par l'introduction de deux nouveaux concepts : *Functional* et *Non-functional*. Ces concept sont utilisés pour capturer respectivement les éléments générés pour les préoccupations fonctionnelles et ceux générés pour les préoccupations non-fonctionnelles.

Afin d'enrichir la sémantique des liens de traces, nous définissons trois types de liens : *Before*, *After* et *Merge*. Les liens de traces *Before* et *After* indiquent que l'élément produit en sortie est obtenu par l'insertion d'une ou plusieurs instances de la méta-classe *Non-Functional* avant, respectivement après, une ou plusieurs instances de la méta-classe *Functional*. Finalement, le lien de trace *Merge* indique que l'élément produit en sortie est obtenu par la fusion d'une ou plusieurs instances

de la méta-classe *Non-Functional* avec une ou plusieurs instances de la méta-classe *Functional*.

La figure 4.12 présente une capture d'écran du modèle de traces sous forme arborescente. Par exemple, le lien de trace *merge* signifie que l'élément de sortie *Customer* est obtenu par la fusion de l'élément non-fonctionnel *User* et l'élément fonctionnel *Customer*.

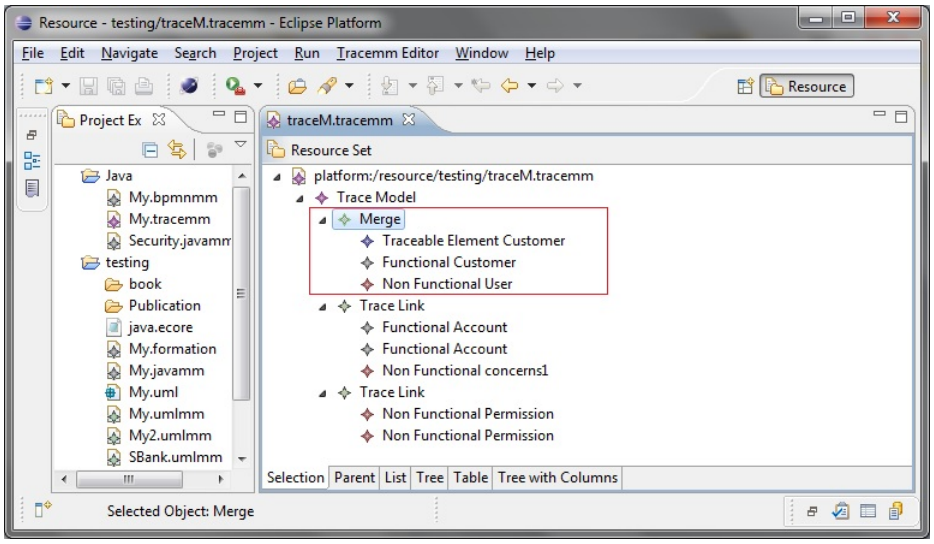


FIGURE 4.12 – Exemple de modèle de traces sous forme arborescente

Durant les transformations, des liens de traces ont été tissés et sont stockés dans des modèles de traces qui sont conformes à leur méta-modèle présenté à la figure 4.10.

Pour illustrer comment ces traces sont générées, nous considérons la transformation T2 qui produit un modèle spécifique à la plateforme (Java model) à partir d'un modèle indépendant de la plate-forme (Secure bank model). Un exemple de règle ATL tracée associée à la transformation T2 est présenté par le listing 4.5.

Listing 4.5 – Règle ATL tracée (C2C)

```
1 rule C2C {
2         from
3                 a:UML!Class
4         to
```

```
5                   b:JAVA!JavaClass(
6                           name<-a.name,
7                           isAbstract<-a.isAbstract,
8                           superJavaClass<-a.superClass,
9                           package<-a.namespace),
10                          traceLink:TRACE!TraceLink (
11                          ruleName <- 'C2C',
12                          targetElts <- Sequence{b})
13                           do {
14      traceLink.refSetValue('sourceElts', Sequence {a});
15    }
16  }
```

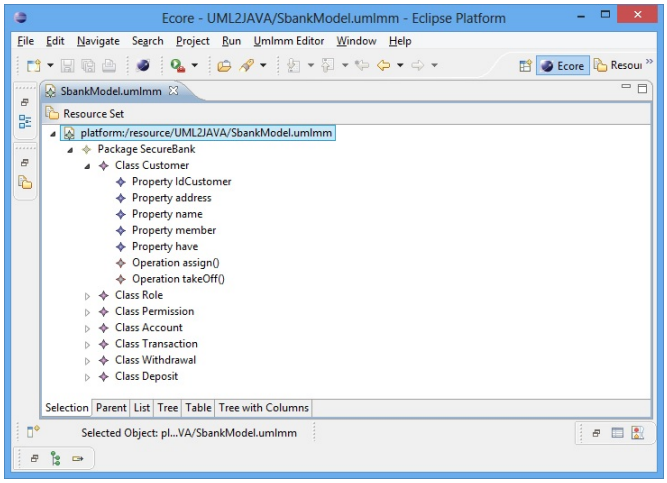
Cette règle ATL peut générer un lien de trace *traceLink*, conforme au méta-modèle *TRACE*, permettant de lier chaque élément impliqué *a* à l'élément créé dans *b*. Les lignes 10 à 12 construisent un lien de trace de type TraceLink en précisant le nom de la règle tracée (C2C) grâce à son attribut ruleName (ligne 11) et en pointant b comme élément de destination (ligne 12). Les lignes 13 à 15, quant à elles, pointent l'élément a comme élément source.

La figure 4.13 présente un extrait des modèles d'entrée et de sortie de la transformation *UML2JAVA*. Les dépendances entre les éléments d'entrée (figure 4.13a) et les éléments de sortie (figure 4.13b) sont spécifiées par les liens de traces présentés dans la figure 4.13c. Par exemple, le premier lien de trace, montre que la classe java *(Java Class Customer)* est obtenue à partir de la classe UML *Class Customer* en appliquant la règle de transformation appelée *C2C*. Le deuxième lien indique que la propriété *IdCustomer* a été consommée par la règle *A2F* qui a produit le champ *Field IdCustomer*. Finalement, le dernier lien nous montre que l'opération *assign()* a servi à créer la méthode Java *assign()* suite à l'application de la règle *OP2M*.

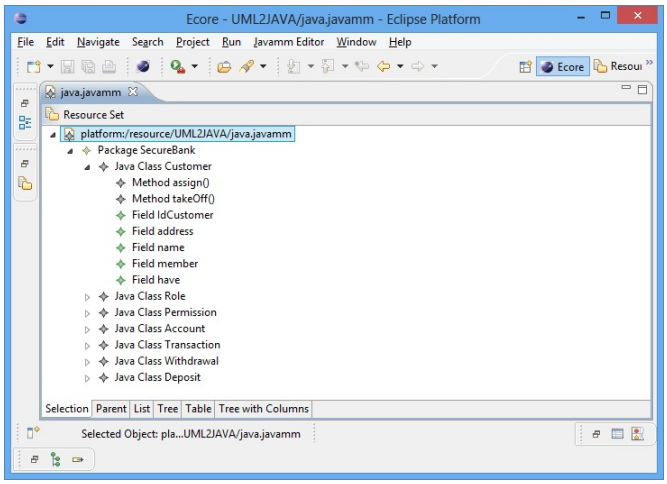
4.2.3 Scénario d'évolution

La figure 4.14 présente une version évoluée du modèle "Secure Bank" décrit dans la section 4.2.1. Les changements introduits sont :

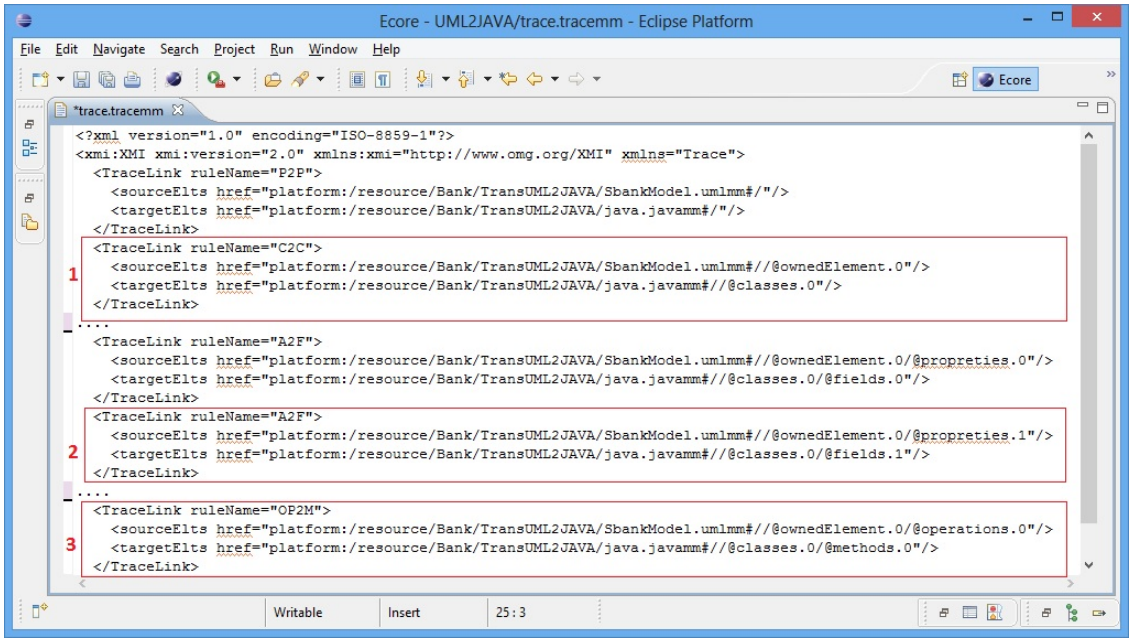
1. Ajout d'une classe *"Bank"*, d'une méthode *"newAccount"* et d'une associa-



(a) Secure Bank model



(b) Java model



(c) Trace model

FIGURE 4.13 – Modèles de la transformation T2 en XMI

tion *"manage"* entre la classe *"Bank"* et la classe *"Account"*

2. Changement du nom de la classe *"Transaction"* en *"BasicTransaction"*
3. Suppression de l'attribut *"balance"* de la classe *"Account"*
4. Ajout de l'attribut *"amount"* à la classe *"Account"*

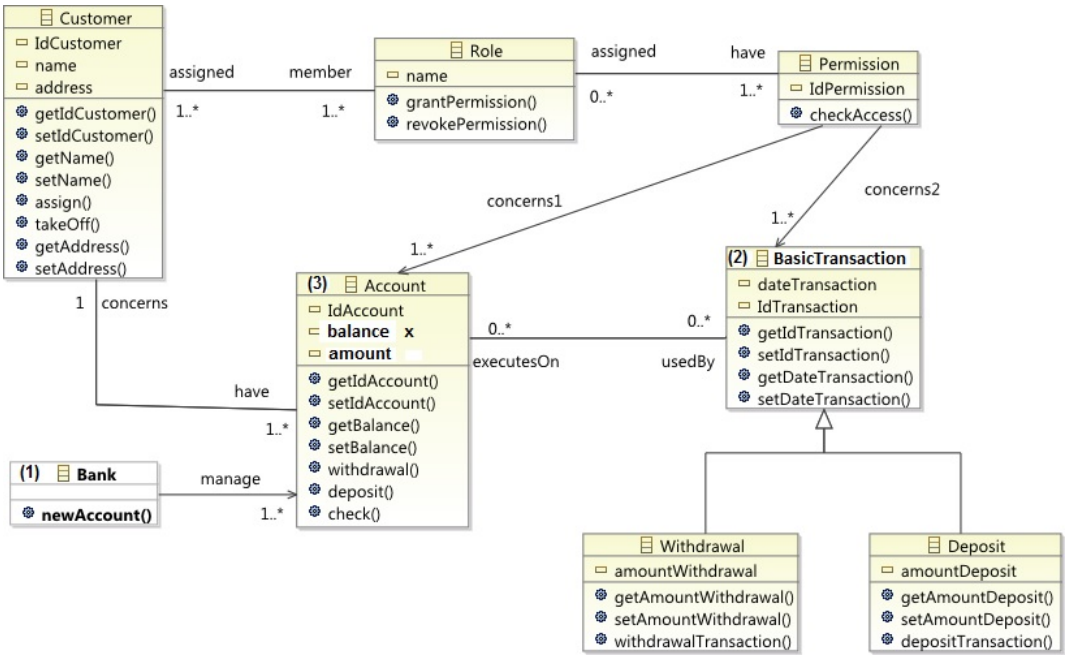


FIGURE 4.14 – Exemple d'évolution du modèle "Secure Bank"

4.3 Processus d'évolution des liens de traces

le processus d'évolution des liens de traces se déroule en trois étapes importantes : une étape de comparaison de modèles *(models comparison)*, une étape de détection et classification des changements *(Changes detection and classification)* et une étape d'évolution des liens *(Evolve links)*. La première étape consiste à détecter les changements élémentaires entre deux versions d'un modèle. La seconde étape fournit des informations sur toutes les modifications nécessaires aux relations de traçabilité. La troisième étape consiste à mettre à jour les liens associés aux éléments modifiés. La figure 4.15 illustre ce processus [56].

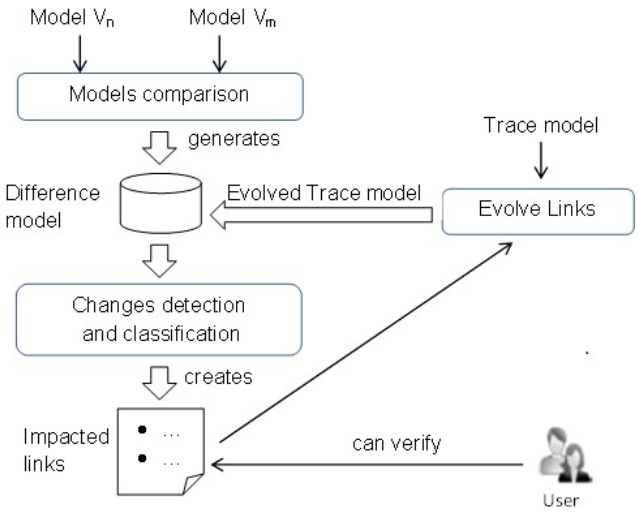


FIGURE 4.15 – Processus d'évolution des liens de traces [56]

4.3.1 Comparaison de modèles

Cette étape consiste à détecter des changements entre la version initiale et la version évoluée du modèle. La gestion des versions est accomplie par une représentation basée sur les modèles permettant la détection, l'enregistrement et la visualisation des différences entre deux versions du modèle. Cette représentation est basée sur le travail de Cicchetti et al. [21], qui a introduit une technique pour la modélisation des changements en définissant trois opérations élémentaires : ajout, suppression et mise-à-jour.

Pour mettre en œuvre notre approche, nous avons utilisé l'environnement EMF d'Eclipse. Nous avons fait ce choix car cet environnement est couramment utilisé dans le monde académique et il offre un grand nombre d'outils. Parmi les outils de comparaison de modèles disponibles, nous avons utilisé *EMF Compare* [17], qui est intégré à l'environnement EMF, et correspond aux principes exposés par Cicchetti et al. [21] Il permet de calculer le modèle de différence « delta » entre deux versions du même modèle et de fournir le résultat de la comparaison sous la forme d'un modèle que nous pouvons exploiter selon nos besoins.

EMF Compare fournit le résultat d'une comparaison de modèles sous la forme

de deux modèles, un modèle de correspondances (*MatchModel*) et un modèle de différences (*DiffModel*). Dans le cadre de notre utilisation, nous n'avons pas besoin de connaître le détail des points communs entre les modèles comparés, nous nous intéressons au détail des différences observées, s'il y en a.

Les différences sont toujours interprétées comme des changements sur l'élément de droite. Un élément de différence est ajouté au modèle de différence qui, en fin d'exécution de l'algorithme, contient toutes les différences entre les modèles avec une description de ce qui est différent.

La figure 4.16 présente un extrait du métamodèle de différence d'EMF Compare. Un modèle de différences est constitué d'un ensemble d'éléments de différence (*DiffElement*). Un élément de différence peut simplement regrouper plusieurs éléments correspondant à une même unité (même package, même modèle).

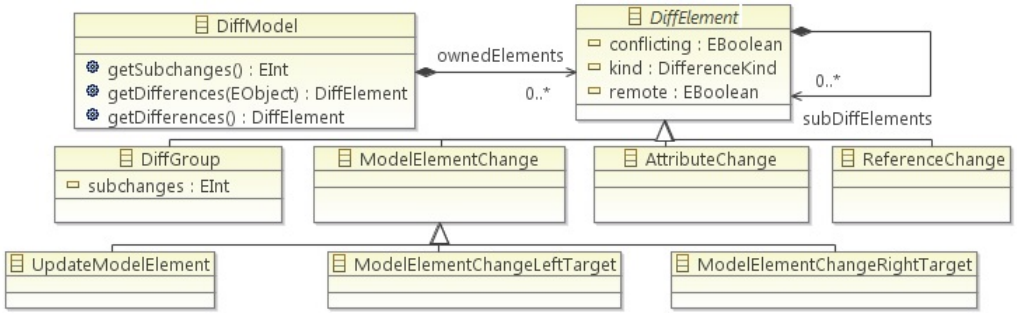


FIGURE 4.16 – Extrait du métamodèle de différence d'EMF Compare.

EMF Compare prend en compte trois types de changements :

— *Attribute changes* : Ce type de modification décrit la différence entre les valeurs d'un attribut donné.

— *Reference changes* : Ce type de changement indique la modification d'une référence d'un élément à un autre.

— *Model Element changes* : Ce type de changement spécifie les éléments ayant été ajoutés ou supprimés.

Dans notre approche, les changements de modèle sont définis par trois (03) opérateurs principaux :

1. **Ajouter un élément *(add)*** : Pour modifier un modèle, ce type de chan-

gement nous permet d'ajouter de nouveaux éléments. Une fois que la modification est effectuée, des transformations sont relancées et de nouveaux artefacts peuvent être générés. L'évolution des modèles d'entrée ou de sortie nécessite la mise à jour des modèles de traces en ajoutant de nouveaux liens de traces produits par la chaine de transformations.

Mise à jour des liens de traces : L'ajout d'un nouvel élément à un modèle nécessite la ré-exécution des transformations afin de générer les artefacts correspondants aux éléments nouvellement crées. Les nouvelles relations générées implicitement ou explicitement par les transformations spécifient quels éléments sont liés au élément inséré, il est donc nécessaire de faire évoluer le modèle de trace en créant une ou plusieurs relations de traçabilité sur l'élément nouvellement créé.

Impact de l'ajout sur les liens de traces : Ce type de changement n'a aucun impact sur les relations de traçabilité existantes, mais de nouvelles relations doivent être ajoutées au modèle de trace pour assurer l'intégrité des liens de traces.

2. **Supprimer un élément *(delete)*** : Ce type de modification vous permet de supprimer un élément du modèle. La suppression d'un élément consiste en la suppression de toutes ses relations. Il est donc nécessaire de supprimer les relations de traçabilité devenues inutiles.

   Mise à jour des liens de traces : Lorsqu'un élément est supprimé d'un modèle, il est nécessaire d'enlever toutes les relations de traçabilité associées à cet élément à partir du modèle de trace.

   Impact de la suppression sur les liens de traces : La suppression des relations de traçabilité n'a pas d'impact si elles ne relient que des éléments indépendants. Si les éléments cibles sont liées à d'autres éléments, il est donc nécessaire de vérifier s'ils sont toujours valides et nécessaires ou non.

3. **Modifier un élément *(modify)*** : L'opérateur de modification permet aux utilisateurs de modifier le nom d'un élément (par exemple, renommer un package, une classe, un attribut, une association ou une opération) ou de

modifier ses propriétés (par exemple, la multiplicité minimale d'une association).

Mise à jour des liens de traces : Lorsque un élément est renommé par l'utilisateur, Il est nécessaire de maintenir, dans le modèle de traces, l'ensemble des relations de traçabilité associées à l'élément modifié.

Impact de la modification sur les liens de traces : Ce type de changement n'a aucun effet sur les relations de traçabilité déjà existantes.

Le code présenté au listing 4.6 permet de calculer les différences entre la version initiale et la version évoluée du modèle.

Listing 4.6 – Comparaison de modèles avec EMF Compare

```
1 import org.eclipse.emf.compare.diff.metamodel.DiffElement;
2 import org.eclipse.emf.compare.diff.metamodel.DiffModel;
3 import org.eclipse.emf.compare.diff.service.DiffService;
4 import org.eclipse.emf.compare.match.metamodel.MatchModel;
5 import org.eclipse.emf.compare.match.service.MatchService;
6 import org.eclipse.emf.compare.util.ModelUtils;
7 import org.eclipse.emf.ecore.EObject;
8 import org.eclipse.emf.ecore.resource.ResourceSet;
9 import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
10 public class compare {
11 public static void main (String[] args) throws IOException,
12 // Load the two input models
13 ResourceSet resourceSet1=new ResourceSetImpl();
14 ResourceSet resourceSet2=new ResourceSetImpl();
15 URI uri1=URI.createFileURI("model/BankV1.ecore");
16 URI uri2=URI.createFileURI("model/BankV0.ecore");
17 EObject model1=ModelUtils.load(uri1,resourceSet1);
18 EObject model2=ModelUtils.load(uri2,resourceSet2);
19 // Matching
20 MatchModel match=MatchService.doMatch(model1,model2,Collections<String,↩
      Object>emptyMap());
21 // Compare the two models
22 DiffModel diff=DiffService.doDiff(match,false);
23 ModelUtils.save (diff ,"model/diffxmi");
24 // List of differences
25 List <DiffElement>differences=diff.getDifferences();
26 for ( DiffElement diffElement:differences ) {
27    System.out.println("Change type : "+ diffElement.getKind());
```

```
28     System.out.println(diffElement.toString()); }
29 }
30 }
```

La figure 4.17 présente le résultat de la comparaison de deux versions du modèle *Secure bank* que nous avons présenté en section 4.2.3. les changements générés par *EMF compare* et représentés par le modèle de différence sont : (1) l'élément *Transaction* est renommé (2) l'ajout des éléments *Bank* et *amount*, et (3) suppression de l'élément *balance*,

```
Change type: Change
Attribute name in BasicTransaction has changed from Transaction to
BasicTransaction

Change type: Addition
org.eclipse.emf.ecore.impl.EClassImpl@e49d67c (name: Bank)
(instanceClassName: null) (abstract: false, interface: false) has been added

Change type: Addition
org.eclipse.emf.ecore.impl.EAttributeImpl@97494c8 (name: amount) (ordered:
true, unique: true, lowerBound: 0, upperBound: 1) (changeable: true,
volatile: false, transient: false, defaultValueLiteral: null, unsettable:
false, derived: false) (iD: false) has been added

Change type: Deletion
org.eclipse.emf.ecore.impl.EAttributeImpl@203fa5ac (name: balance) (ordered:
true, unique: true, lowerBound: 0, upperBound: 1) (changeable: true,
volatile: false, transient: false, defaultValueLiteral: null, unsettable:
false, derived: false) (iD: false) has been removed
```

FIGURE 4.17 – Différences entre les deux versions de modèle

4.3.2 Détection et classification des changements

Les changements détectés et représentés à la phase de *comparaison* sont ajoutés à une liste de mise à jour qui fournit les informations nécessaires à l'évolution des relations de traçabilité. Les changements de base qu'il est possible d'effectuer grâce à notre approche sont présentées et classées en 2 grandes catégories :

— **CATEGORIE 1 : *Changements avec impact sur les liens de traces.*** Dans cette catégorie, une opération de mise à jour est nécessaire pour maintenir l'intégrité des relations de traçabilité stockées dans le modèle des traces. L'ajout de la classe "Bank" et la suppression de l'attribut "balance", illustrés en Figure 4.14, sont des exemples de ce type de modifications.

— **CATEGORIE 2 :** ***Changements sans impact sur les liens de traces.*** Cette catégorie correspond aux évolutions dont l'impact sur les relations de traçabilité est nul. Nous considérons que ce type de changement est bien sans impact sur l'intégrité du modèle de traces. L'exemple d'une évolution sans impact est la modification du nom de la classe "Transaction" comme illustré en Figure 4.14.

4.3.3 Evolution des liens de traces

Les changements obtenus lors de la phase de détection et classification des changements sont utilisés pour déterminer les opérations nécessaires pour faire évoluer les liens de traces face à l'évolution de modèles. Selon les types de changements présentés en section 4.3.2, les différentes opérations qui doivent être effectuées pour faire évoluer les modèles de traces sont les suivantes :

— *Un nouvel élément est ajouté au modèle source.* La maintenance de la traçabilité nécessite la création de nouvelles relations sur les éléments nouvellement créés.

— *Un élément est supprimé du modèle source.* La mise à jour de la traçabilité consiste à enlever tous les liens de traces qui font référence à l'élément supprimé.

— *Un élément est remplacé par un autre élément.* La maintenance des relations de traçabilité nécessite de restaurer tous les liens faisant référence à l'élément remplacé et les mettre à jour en remplaçant l'élément référencé par le nouvel élément.

Les actions de mise à jour à entreprendre pour chacune des modifications enregistrées dans la liste de mise à jour se reposent sur l'ensemble de directives suivant :

1. L'action de mise à jour concernant les relations de traçabilité associées aux éléments modifiés consiste à préserver les liens de traces, correspondants à chaque élément modifié, lorsque l'élément est maintenu dans la nouvelle version du modèle source.

2. Si un élément est supprimé dans le modèle évolué, alors tous les liens de

traces le reliant avec n'importe quel autre élément doivent être supprimés.

3. SI un nouvel élément est ajouté au modèle, alors de nouveaux liens doivent être créés et ajoutés au modèle de traces.

4. Toutes les relations associées aux éléments modifiés doivent être maintenues. Ces relations ne sont pas impactées par cette activité d'évolution.

Le listing 4.7 présente la structure de base d'une règle d'évolution. Une règle est caractérisée par un *nom* (ligne 1), une *description* (ligne 2) donnant une information textuelle et générale sur la règle. La ligne 4 exprime une garde qui spécifie à quelle condition la règle s'exécute. La partie *condition* est similaire à une expression booléenne en Java. Une condition peut contenir deux ou plusieurs expressions Booléennes simples connectées par des opérateurs logiques. Nous fournissons une condition prédéfinie à chaque type de changements générés par l'opération de comparaison de modèles. Chaque condition exprime qu'un type particulier de changement de modèle est produit. Par exemple, la condition *"addAssociation"* vérifie si le modèle de différence a généré un changement suite à l'ajout d'une nouvelle association.

Listing 4.7 – Structure de base d'une règle d'évolution

```
1 evolveRule "<name>" {
2    description = "..."
3    evolve {
4       (<condition part> => <action part>)
5  }
6 }
```

4.4 Validation de l'approche

Cette section présente la validation de notre approche. Afin de s'assurer que les modifications effectuées sont complètes et correctes, nous avons utilisé deux mesures qui sont généralement utilisés pour évaluer les approches de reconnaissance tenant compte de l'incertitude et de l'imprécision : « la precision » et le « rappel » [6].

4.4.1 Objectifs et questions de recherche

Afin d'éprouver notre approche, nous l'avons utilisé sur le cas d'étude présenté en section 4.2. En utilisant ce cas d'étude, nous pouvons évaluer et comparer notre approche à celle manuelle en terme de temps passé et la qualité des modifications effectuées pour la maintenance des liens de traçabilité lors de l'évolution des modèles. Pour cela, nous nous sommes posé les questions suivantes :

— *Question de recherche 1 (Effort manuel)* : Est-ce que notre approche est capable de réduire le temps et l'effort nécessaires pour la génération et la maintenance des relations de traçabilité ? L'évolution manuelle des liens de traces impose au concepteur de parcourir les modèles évolués et d'effectuer lui même les opérations requises pour gérer l'impact de ces changements sur les liens de traces.

— *Question de recherche 2 (Qualité de la maintenance)* : Est-ce que l'approche proposée peut améliorer la qualité des liens de traces par rapport à celle obtenue avec l'approche manuelle ? Pour mesurer, d'une part le taux de modifications correctement effectuées, et d'autre part le nombre de modifications oubliées, nous allons utiliser le rappel et la précision. Pour calculer ces métriques, nous rappelons ces définitions :

Δ_c l'ensemble des changements correctement effectués.

Δ_i l'ensemble des changements erronés.

Δ_m l'ensemble des changements qui ont été oubliés.

La *précision* est la proportion des changements correctement effectués parmi tous les changements établis. Elle permet de juger de la qualité des changements effectués par notre approche. La précision P est définie par :

$$P = \triangle_c / (\triangle_c + \triangle_i).$$

Plus la précision est proche de 1, plus le nombre de modifications correctement effectuées est important.

Le *rappel* est le rapport du nombre des changements correctement établis au nombre de tous les changements devant être effectués. Il nous permet de savoir si des changements ont été oubliés. Le rappel R est défini par :

$$R = \triangle_c / (\triangle_c + \triangle_m).$$

Plus ce ratio est proche de 1, moins le nombre d'opérations oubliées est élevé.

4.4.2 Ressource expérimentale

Dans le but de valider notre approche, nous avons utilisé le scénario défini en section 4.2. Les artefacts utilisés sont des modèles UML décrivant un système bancaire simplifié. Les artefacts du système sont définis à deux niveaux d'abstraction : un digramme de classe UML et un modèle Java. Lors de l'exécution de la chaine de transformations, un modèle de traces conservant les liens entre les éléments des modèles source et cible est produit. Dans notre cas, le modèle généré se compose de 47 liens de traçabilité.

Afin de répondre à la première question, nous avons mesuré le temps d'exécution nécessaire à l'application de notre approche sur le diagramme de classe (présenté en section précédente). Pour cela, 23 opérations de modifications ont été appliquées sur le modèle source. Les modifications effectuées couvrent les majeurs types de changements souvent appliqués sur un diagramme de classes. Pour l'expérimentation, la maintenance manuelle des relations de traçabilité a pris, environ, 2 heures.

Pour répondre à la deuxième question, nous avons demandé aux concepteurs d'effectuer une série de modifications sur le diagramme de classe. Suite à cette opération, 60 éléments ont été ajoutés, renommés ou supprimés. Pour évaluer la performance de notre approche, deux métriques ont été calculé, le rappel et la précision. Ces paramètres mesurent la qualité des liens de traces générés. Autrement dit, ils nous donnent des informations sur la complétude et la correction des liens de traces produits par notre approche.

TABLE 4.1 – Evolution des relations de traçabilité

	Correctement effectués	Incorrectement effectués	Oubliées	Total
Liens de traçabilité	48	8	4	60
%	80%	13%	7%	100%

4.4.3 Résultats et discussion

L'application de notre approche a pris, environ, 40 minutes. Le temps pris en compte comprend la génération des relations de traçabilité, la comparaison des modèles et l'évolution des liens. Pour les changements non identifiés et ceux détectés avec ambiguïté, l'intervention de l'utilisateur est nécessaire et par conséquent l'évolution de ces traces est gérée manuellement. En tenant compte du temps de traitement des modifications requises,non prises en compte par l'approche, le résultat obtenu est plus important que celui obtenu manuellement.

La table 4.1 montre que 80% des modifications ont été correctement détectées et traitées alors que 13% ont été incorrectement effectuées. Les 7% restantes n'ont pas été prises en compte par l'approche. Ces résultats sont représentés graphiquement sur la figure 4.18.

Les modifications qui n'ont pas été effectuées correctement sont liées au fait qu'une séquence d'opérations élémentaires peut avoir plusieurs interprétations. Par exemple, l'ajout d'un nouvel élément avec les mêmes propriétés de ce qui a été supprimé, génère généralement un changement de type "renommer" au lieu des deux modifications : "supprimer" et "ajouter".

Dans le but de faciliter l'interprétation des résultats obtenus par notre approche, nous avons utilisé deux mesures reprises de la recherche d'information (IR). Les deux informations largement utilisée sont le rappel et la précision. Ce sont des mesures

faciles à calculer, qui ont aussi le mérite d'être faciles à interpréter quel que soit le domaine de recherche où elles sont appliquées. Dans notre cas, le taux de précision obtenu est 86%. Cela signifie que 86% des modifications ont été correctement détectée et interprétée par l'approche alors que 14% des liens qui ont été trouvés sont incorrects. Le taux de rappel, quant a lui , est de 92%. Ceci montre que seulement 8% des liens de traces, qui auraient dû être trouvés, ont été manqués.

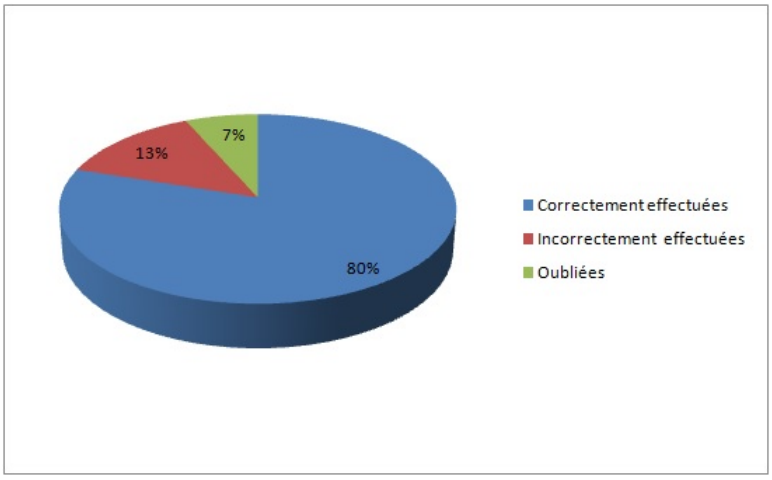


FIGURE 4.18 – Analyse quantitative

4.5 Conclusion

Dans ce chapitre, nous avons commencé par la présentation de notre technique de traçabilité permettant la capture et la conservation des liens de traçabilité entre les modèles utilisés dans un processus de transformation de modèles basé sur la séparation de préoccupations. Ensuite, nous avons présenté une approche pour la maintenance des liens de traces générés par la chaine de transformations de modèles lors de l'évolution des modèles.

Nous avons validé notre approche sur un cas d'étude développé dans l'environnement Eclipse Modeling Framework (EMF). Cette validation nous a amené à développer une transformation des diagrammes de classes UML vers des modèles Java. La validation, via les métriques rappel et précision, nous montre que notre

approche a pu effectuer la quasi-totalité des modifications requises pour maintenir l'intégrité des relations de traçabilité stockées dans le modèle des traces.

Nous considérons une exploitation possible des liens produits par une chaine de transformations. Dans le chapitre suivant, nous utilisons ces traces afin de répondre à une problématique de co-évolution de modèles. Dans cette thèse nous préférons utiliser le terme synchronisation de modèles pour désigner ce type d'évolution.

Chapitre 5

Utilisation de traces pour la synchronisation de modèles

Sommaire

5.1 Introduction 118

5.2 Évolution de Logiciels 119

5.2.1 Catégories de l'évolution de logiciels 120

5.2.2 Activités de développement évolutionnaire 121

5.3 Evolution dans l'IDM 122

5.3.1 Model-metamodel Co-Evolution 123

5.3.2 Synchronisation de modèles 129

5.3.3 Evolution de Transformation 132

5.4 Présentation de l'approche 133

5.4.1 Détection et classification des changements 133

5.4.2 Analyse d'impact du changement 136

5.4.3 Synchronisation d'artéfacts 137

5.5 Conclusion 138

5.1 Introduction

L'ingénierie dirigée par les modèles que nous avons présentée dans le chapitre 1 est une approche de conception qui vise à conserver les modèles comme point de référence dans un processus de développement logiciel. L'un des points clé d'une approche IDM est la possibilité de transformer des modèles d'un espace de modélisation ou d'un niveau d'abstraction vers un autre.

La spécification du système peut alors se faire grâce à un modèle de conception, exprimant les différentes fonctionnalités du système modélisé. Ce modèle est utilisé comme entrée dans une chaîne de transformation de modèles(MTC). Une MTC est une séquence de transformations de modèles à modèles et de modèles vers texte permettant, entre autre, de passer automatiquement d'un niveau d'abstraction à un autre. Ainsi, les détails d'implémentation liés à l'application peuvent être masqués au concepteur et le code source d'une application modélisée peut être directement généré.

Lors de l'utilisation d'une démarche dirigée par les modèles pour le développement d'une application, il arrive fréquemment que des modèles évoluent au fil des versions de cette application et ces évolutions peuvent avoir des répercussions importantes sur les artefacts étroitement liés à ces modèles. Dans l'IDM, un artefact peut être un métamodèle, un modèle (modèle instance), ou une transformation. Les modifications effectuées sur un modèle instance peuvent entrainer des modifications sur tous les modèles générés par la chaîne de transformation.

Le but du travail présenté dans ce chapitre est de proposer une approche de synchronisation de modèles afin d'aider à maintenir automatiquement la consistance entre le modèle modifié et les modèles qui lui sont associés. L'objectif étant que l'intervention manuelle du concepteur soit la plus réduite possible. La raison qui nous a poussée à investiguer cette idée est que la plupart des travaux présentés sur l'évolution des modèles reposent : soit sur la re-génération entière des modèles cibles *(re-transformation)*, soit à faire la transformation seulement sur les parties modifiées *(live transformation)*. Bien que cette solution semble évidente, le problème est plus important dans le cas où certaines transformations sont effectuées de manière semi-

automatique ou manuelle.

Notre approche de synchronisation de modèles est basée sur les liens de traçabilité capturés pendant l'exécution de la transformation. Pour mieux gérer l'évolution de modèles, nous proposons une classification des changements en deux catégories : (1) des changements qui nécessitent la ré-exécution des transformations et (2) des changements qui ne l'exigent pas.

La suite de ce chapitre est organisée comme suit : La section 5.2 décrit brièvement la notion d'évolution de logiciels. Nous y introduisons une classification d'évolution du logiciel et présentons quelques activités liées à cette évolution. La section 5.3 dresse un état de l'art des principaux travaux concernant l'évolution dans l'ingénierie des modèles selon l'évolution de l'un des artéfacts : métamodèle, modèle et transformation. Dans la section 5.4, nous présentons notre approche de synchronisation de modèles basée sur les liens de trace générés par la chaine de transformations de modèles [54].

5.2 Évolution de Logiciels

L'évolution du logiciel est un champ relevant du domaine de l'ingénierie du logiciel et ayant comme but l'étude des moyens d'adaptation du logiciel aux changements continus qui affectent aussi bien les exigences ou besoins des utilisateurs que les environnements opérationnels supports[69]. Le génie ou ingénierie du logiciel est un terme issu de la première conférence consacrée à ce domaine et organisé par le comité scientifique de l'OTAN en Allemagne en 1968 [78]. Cette conférence a posé les principes et les fondements permettant de faire en sorte que la production du logiciel soit semblable à toutes les disciplines relevant des différents domaines de l'ingénierie. Le but étant de produire des logiciels opérationnels et fiables et cela dans le respect des coûts et des délais.

Durant les quatre dernières décennies, l'évolution du logiciel a connu une importance de plus en plus grandissante faisant d'elle un concept clé du génie logiciel. Le standard IEEE 1219 de la maintenance du logiciel [62] définit la maintenance comme « *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 ». Le terme évolution du logiciel manque de définition standard. Généralement, évolution et maintenance du logiciel sont utilisées comme synonyme l'une de l'autre.

5.2.1 Catégories de l'évolution de logiciels

L'évolution des besoins des utilisateurs, l'adaptation aux nouvelles technologies, et la restructuration, ont été identifiées comme les principales causes de l'évolution des logiciels [98]. Ces activités sont les motivations pour plusieurs types d'évolution [100] :

- *L'évolution de la spécification des besoins* : dans un projet de développement, les besoins à satisfaire sont exprimés dans le cahier des charges de l'application. Le contenu de ce cahier, satisfaisant pour une certaine durée, peut progressivement accuser une différence avec l'expression des besoins engendrés par un monde évolutif. Il arrive souvent que ces besoins soient effectifs dès le début du projet, mais ils sont constatés tardivement et ne sont donc pas spécifiés dans le cahier des charges (signé auparavant). Dans un autre cas, faute d'une bonne estimation, ces besoins qui ont été jugés secondaires en début du projet, ont pu regagner de l'importance au cours du développement. Dans les deux cas, une réactualisation de l'expression des besoins peut devenir nécessaire et provoquer une renégociation du cahier des charges entre les différentes parties du développement. La décision de faire évoluer l'expression des besoins rend particulièrement nécessaire la mise à jour des composants déjà développés.
- *L'évolution ou maintenance corrective* : il s'agit principalement de modifications pour la correction d'anomalies ou de défauts.
- *L'évolution ou maintenance adaptative* : ce sont des modifications qui consistent à migrer le logiciel, vers un nouvel environnement matériel ou logiciel.
- *L'évolution ou maintenance perfective* : ces changements désignent toutes les actions ayant pour but d'améliorer des critères de qualité telles que les

performances, la facilité d'utilisation, etc.

5.2.2 Activités de développement évolutionnaire

De nombreuses activités sont liées à l'évolution du logiciel. Certaines de ces activités sont considérées comme des thèmes de recherche cruciaux. Parmi ces activités nous pouvons citer : la rétro-ingénierie, le refactoring, l'analyse d'impact et propagation du changement, etc. Dans le cadre de cette thèse nous nous intéressons particulièrement à l'analyse de l'impact du changement *impact analysis* et à la propagation du changement *change propagation*.

— La rétro-ingénierie est une activité qui consiste à produire des artefacts de niveau d'abstraction plus élevés à partir des éléments ou artefacts de plus bas niveau d'abstraction tels que les code sources. Cette activité (retro-enginierie) se justifie par le fait que dans la majorité des cas, les documents ou artefacts de haut niveau d'abstraction sont obsolètes voir même inexistants. Il est donc nécessaire de produire des abstractions de haut niveau qui correspondent à l'état actuel du système à faire évoluer pour mieux le refaire ou le faire autrement. Dans [18], Chikofsky et al. ont décrit une taxinomie des activités rentrant dans le cadre de la rétro-ingénierie ou ingénierie inverse.

— le refactoring est une activité de maintenance d'un genre particulier. Il s'agit non pas d'une tâche de correction des anomalies ou de réalisation de petites évolutions pour les utilisateurs mais d'une activité plus proche de la maintenance préventive et adaptative dans le sens où elle n'est pas directement visible de l'utilisateur.

  Le refactoring est une activité d'ingénierie logicielle consistant à modifier le code source d'une application de manière à améliorer sa qualité sans altérer son comportement du point de vue de ses utilisateurs [41]. Son rôle concerne essentiellement la pérennisation de l'existant, la réduction des coûts de maintenance et l'amélioration de la qualité de service au sens technique du terme (performance et fiabilité). Le refactoring est en ce sens différent de la maintenance corrective et évolutive, qui modifie directement le comportement du logiciel, respectivement pour corriger un bogue ou ajouter ou améliorer des

fonctionnalités.

— Le but de l'analyse d'impact consiste à déterminer les effets directs et indirects d'une opération de changement. En général, Cela est possible en exploitant l'ensemble des liens ou relations qui existent entre les différents artefacts du logiciels. Ces liens incluent des relations de mise en correspondance entre artefacts du logiciel [1]. La propagation du changement consiste à effectuer les changements requis aux entités du logiciel après modification d'une entité particulière.

5.3 Evolution dans l'IDM

L'évolution des logiciels est le processus majeur assurant l'innovation et l'amélioration continues des méthodes, des techniques et des outils de traitement de l'information. De façon similaire, l'utilisation d'une démarche dirigée par les modèles est sensible à l'évolution. Dans l'IDM, l'évolution se confronte à plusieurs problèmes principalement liés aux étroites dépendances entre métamodèles et modèles, métamodèles et transformations, et entre chaque phase de transformation et celles qui la suivent. En se basant sur le classement proposé par Van Deursen et al. [104], nous pouvons diviser l'évolution dans l'IDM en quatre catégories :

— *Evolution régulière* : ce type d'évolution affecte les modèles existants quand l'application concernée à besoin de répondre aux nouvelles exigences et que le langage de modélisation du système peut les spécifier. Par conséquent, les modèles sont modifiés, mais les métamodèles et les transformations restent les mêmes.

— *Evolution de métamodèle* : ce type d'évolution est nécessaire lorsque le langage de modélisation doit être modifié. cette évolution à un impact important sur les modèles conformes au métamodèle d'origine. Dans la littérature, il existe un terme assez répandu pour désigner ce problème et qui est *la co-évolution de modèles* ou en anglais *metamodel and model coevolution.*

— *Evolution de la plateforme* : Cette évolution est nécessaire dans le cas où de nouvelles exigences liées à la plate-forme cible sont ajoutées. Dans ce cas les

transformations modèle-modèle et modèle-texte doivent être modifiées. Les modèles et les métamodèles restent inchangés.

— *Evolution d'abstraction* : ce type d'évolution apparaît lorsque de nouveaux langages de modélisation sont ajoutés, pour permettre une meilleure compréhension d'une technique ou d'un domaine particulier. Ce cas nécessite la modification des modèles, des métamodèles et des transformations.

Plusieurs travaux de recherche se sont intéressés à l'évolution dans l'IDM. Dans les sous-sections suivantes, nous allons introduire l'essentiel de ces travaux selon une classification basée sur l'évolution de l'un des artéfacts : métamodèle, modèle et transformation.

5.3.1 Model-metamodel Co-Evolution

Un métamodèle sert à exprimer les concepts communs à l'ensemble des modèles d'un même domaine. Pendant la phase de modélisation ou de maintenance, lorsque le domaine évolue, les concepteurs devraient alors faire évoluer les différents métamodèles sur lesquels ils travaillent afin d'ajouter de nouvelles fonctionnalités , modifier des éléments existants, ou supprimer des éléments devenus inutiles. Ceci risque d'engendrer des problèmes à partir du moment où les métamodèles sont liés avec d'autres artefacts 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 qui seront impactés par les changements. Lorsque de nouveaux concepts sont ajoutés au métamodèle, la cohérence entre les modèles et le métamodèle est maintenue. Cependant, si des concepts sont modifiés ou supprimés, les modèles pourraient ne plus être conformes à la nouvelle version du métamodèle et ils devront être ajustés afin de préserver la cohérence entre les modèles et le métamodèle évolué. Par exemple, si un attribut est supprimé d'une méta-classe, il sera donc nécessaire de corriger les modèles en éliminant chaque instance de cet attribut. Dans la littérature, ce processus est désigné par le terme *Model-metamodel Co-Evolution*. Cependant, si chaque activité est considérée séparément, l'évolution au niveau modèle est dénotée par le terme *Migration*.

De plus, lorsque le métamodèle change, la transformation qui l'utilise comme

source ou cible doit être aussi adaptée. Le problème de *l'évolution de transformation* sera présenté en sous-section 5.3.3.

La co-évolution de modèles a fait l'objet d'un grand nombre de travaux proposés dans la littérature. Dans [71], Rose et .al proposent une classification des approches de métamodèle et modèle co-évolution en trois catégories : *spécification manuelle*, *Co-évolution basée opérateur* et *le matching de métamodèle*.

5.3.1.1 Spécification manuelle

Dans cette catégorie, la stratégie de migration est implémentée manuellement par le développeur. La spécification peut être exprimée dans un langage de programmation tel que Java ou dans un langage de transformation de modèles. L'adaptation manuelle de modèles est une tâche fastidieuse et source d'erreurs, elle pourrait introduire des incohérences entre modèles et métamodèles.

Rose et al. [91] se sont inspirés des langages de transformation de modèles (ATL et QVT) pour proposer un langage hybride nommé Epsilon Flock. Contrairement à ATL, ce langage permet de copier automatiquement les éléments du modèle qui sont toujours conformes au méta-modèle évolué. Ces éléments sont copiés à l'aide d'un algorithme nommé « Conservative copy ». Pour chaque élément du modèle initial, l'algorithme vérifie sa conformité au méta-modèle évolué. Si l'élément est toujours conforme (l'élément n'est pas impacté par l'évolution), alors l'algorithme procède à la copie automatique de cet élément dans le modèle co-évolué. Des règles de transformation Flock sont écrites manuellement pour faire co-évoluer les éléments impactés par l'évolution du méta-modèle.

5.3.1.2 Co-évolution basée opérateur

Dans les approches *operator-based co-evolution*, l'évolution de métamodèles doit être effectuée à travers un ensemble d'opérateurs *(operators)* fournis aux développeurs. Chaque opérateur spécifie un type d'évolution du métamodèle avec une stratégie de migration de modèle correspondante. Par exemple, l'opérateur *"Make Reference Containment"* peut faire évoluer le métamodèle tels que chaque référence d'un type *"non-containment"* devient une référence *"containment"*. De plus, il doit

assurer la migration de modèles en remplaçant les valeurs de la référence évoluée par des copies.

Dans [105], l'auteur propose une approche transformationnelle pour assister l'évolution de métamodèle par une adaptation progressive. Ceci signifie que l'évolution du métamodèle est assurée par des transformations permettant d'effectuer trois types d'opérations : refactoring, construction, et destruction. Cette approche offre de nombreux avantages par rapport à l'adaptation manuelle : (1) les changements sont explicites et pourraient être utilisés comme moyen de documentation et de traçabilité, (2) plusieurs propriétés de préservation des transformations sont présentés ce qui permet de qualifier une adaptation en fonction de la préservation sémantique ou d'instance, (3) La co-adaptation de modèles est réalisée automatiquement par des co-transformations, et (4) les scripts d'adaptation sont des éléments logiciels qui peuvent être réutilisés dans des scénarios d'adaptation similaires ou être modifiés pour altérer les décisions d'adaptation.

Dans [73], Markus et .al proposent une approche de migration de modèles à travers le langage *COPE (Coupled Evolution of Metamodels and Model)*. COPE est un langage intégrée dans Eclipse Modeling Framework (EMF) qui fournit un mécanisme de co-évolution. Tous les changements détectés entre deux versions d'un métamodèle sont enregistrés de façon automatique afin d'y associer des opérations de migration. Il fournit des primitives pour faciliter l'expression des opérations d'adaptation et de migration qui doivent être complétées par la suite par le concepteur.

L'efficacité d'une approche de co-évolution basée opérateur dépend fortement de la richesse de la librairie d'opérateurs de co-évolution fournie. Dans le cas où aucun opérateur décrivant l'adaptation requise n'est trouvé, le développeur de métamodèle devrait utiliser une autre approche pour effectuer la migration du modèle. Dans COPE, par exemple, si aucun opérateur de co-évolution n'est approprié, la migration est spécifiée manuellement dans un langage de programmation général.

5.3.1.3 Le matching de métamodèles

Les différents approches proposées dans cette catégorie sont basées sur la génération d'un modèle de différences à partir du métamodèle d'origine et de la nouvelle

version. Ensuite, des opérations de migration peuvent être appliquées afin de propager les changements sur les éléments impactés.

Gruschko et al. proposent une stratégie d'évolution de modèles basée soit sur un *modèle de différence* obtenu à partir des deux versions du métamodèle évolué (comparaison directe) ou sur une liste de modifications enregistrées au cours de l'évolution du métamodèle (comparaison indirecte). Le processus de co-évolution proposé dans [52] est présenté par la figure 5.1.

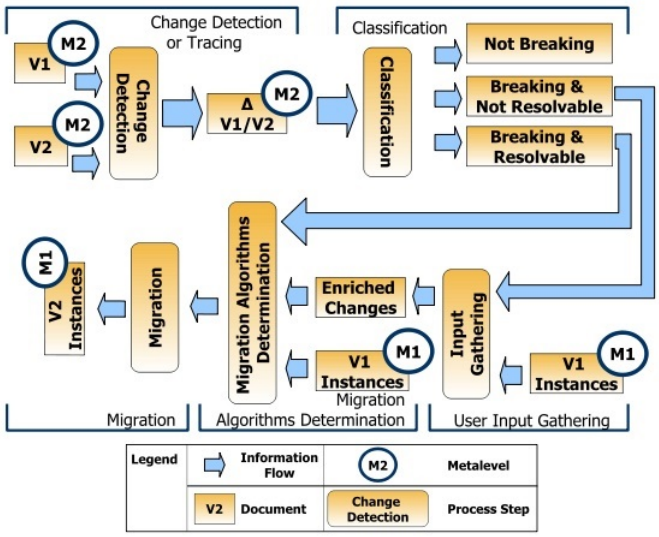


FIGURE 5.1 – Processus de co-évolution [52]

Selon Gruschko et al., le mécanisme d'évolution de modèles suit trois phases : (1) détection des changements, (2) classification des changements selon l'impact, et (3) migration des modèles et impacts des changements.

La phase *détection des changements* consiste à spécifier des différences *(delta)* lors de l'évolution d'un métamodèle. Le delta construit permettra par la suite d'identifier les éléments de modèles impactés par le changement et les modifications possibles qui peuvent être appliquées sur les modèles pour que le domaine maintienne sa cohérence.

La *classification des changements* permet de mieux gérer les impacts en affectant à chaque cas un traitement particulier. Gruschko et al. ont proposé une classification en trois catégories :

— *Sans effet de bord (Non breaking changes)* : Cette catégorie correspond aux évolutions dont les changements ne mettent pas en cause la cohérence du système. Par exemple, l'ajout d'une nouvelle méta-classe ou d'une nouvelle méta-propriété.

— *Avec effet de bord et soluble (Breaking and resolvable)* : Les évolutions de ce type reflètent des changements qui peuvent être résolus automatiquement sans intervention humaine. Par exemple, la modification d'un élément de modèle.

— *Avec effet de bord et non soluble (Breaking and non resolvable)* : Cette catégorie fait référence à des changements qui ne peuvent pas être résolus automatiquement. Une intervention humaine est donc nécessaire afin d'apporter des informations supplémentaires pour assurer l'opération d'adaptation. Par exemple, la suppression d'un élément du méta-modèle signifie que l'instance concernée doit être supprimée alors que cette dernière est reliée à d'autres éléments.

Lors de la phase *migration des modèles et impacts des changements*, des opérations de migration peuvent être appliquées afin de propager les changements aux modèles . L'opération de migration est une opération de transformation qui concerne seulement les éléments nécessitant un ajustement afin de maintenir la cohérence entre les modèles du système lors de leurs évolutions.

Deux approches, inspirées des travaux de Gruschko et al., sont présentées dans la littérature [20, 47]. Les deux approches utilisent un processus de co-évolution similaire à celui présenté dans la figure 5.1. De plus, une stratégie de co-évolution est déterminée en utilisant une transformation d'ordre supérieur (HOT) qui prend le modèle de différence en entrée et produit un modèle de transformation permettant de migrer les modèles associés.

Par exemple, l'approche de Cicchetti et al. consiste à obtenir en premier lieu un métamodèle de différence (MMD) à partir duquel le modèle de différence contenant les changements sera instancié. Ensuite, le modèle de différence est utilisé pour décrire une stratégie de migration en utilisant généralement une transformation de modèles d'ordre supérieur (HOT) qui permet de générer automatiquement une trans-

formation pour la migration. La structure générale du mécanisme de représentation de changement est représentée dans la figure 5.2.

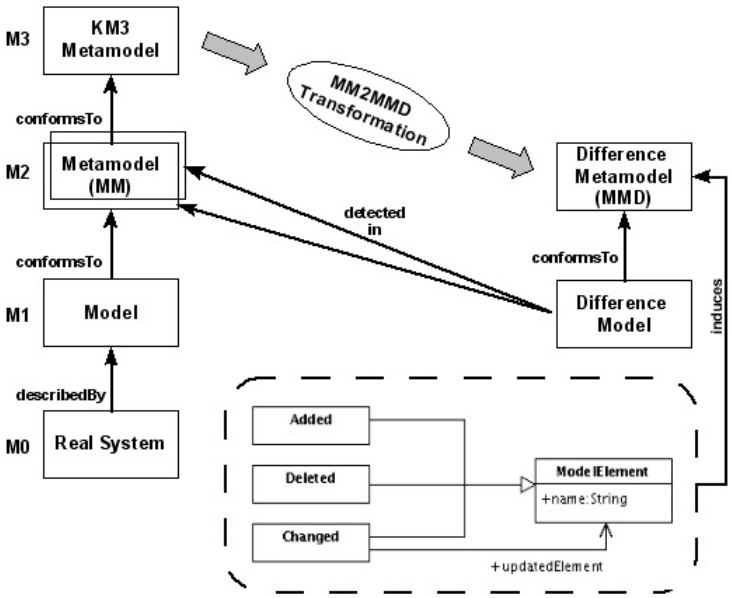


FIGURE 5.2 – Structure générale de l'approche de représentation de modèle de différence [20]

Par rapport à la spécification manuelle et à la co-évolution basée opérateurs, les approches de matching nécessitent moins d'effort de la part du développeur des métamodèles puisque il n'a besoin que de fournir le métamodèle initial et le métamodèle évolué. Cependant, pour certains types de changement, plusieurs stratégies de migration peuvent être utilisées. Par exemple, lorsqu'une méta-classe est supprimée, une des stratégies de migration possibles est la suppression de toutes ses instances. Une deuxième possibilité est de modifier le type de chaque instance de la méta-classe supprimée par une autre méta-classe qui spécifie des caractéristiques structurelles équivalentes. Les informations spécifiées par le métamodèle de changement ne sont pas suffisantes pour distinguer correctement entre les différentes stratégies de migration possibles. Ces approches doivent donc fournir aux développeurs des directives leur permettant de sélectionner la stratégie de migration la plus appropriée parmi toutes les alternatives possibles.

5.3.2 Synchronisation de modèles

Dans la littérature, le terme de *"co-évolution"* fait référence à la co-évolution entre un méta-modèle et des modèles conformes [19] [105]. Cette problématique ressemble fortement aux évolutions de schémas dans le domaine des bases de données. Pour assurer la relation de conformité entre le schéma et les données dans un contexte de bases de données orientées objet [89], les changements de schéma sont répercutés sur les instances existantes. Nous précisions que les co-évolutions des méta-modèles [59], [19] sont hors du cadre de notre thèse.

La co-évolution peut aussi être considérée comme la propagation des changements entre des modèles, mais c'est le terme *synchronisation de modèles* qui est généralement utilisé pour designer l'adaptation des modèles cibles au fur et à mesure que des modifications sont appliquées sur les modèles sources.

La transformation est l'action générative de modèles, alors la synchronisation est l'action de coordonner plusieurs modifications entre elles, en fonction des niveaux d'abstraction de modèles, précédemment générés. Il convient donc de parler de la "propagation du changement entre modèles".

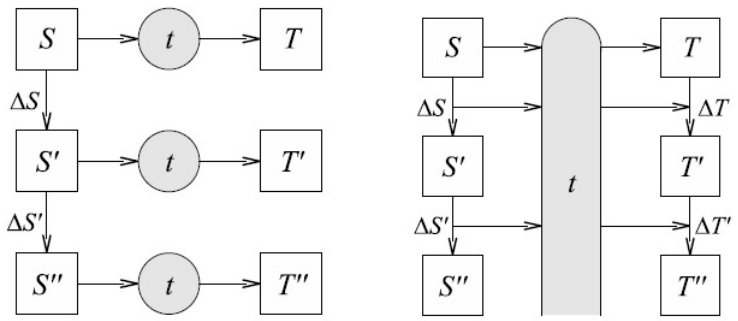
Comme discuté dans le chapitre 3, la traçabilité facilite l'automatisation de certaines activités telles que l'analyse d'impact et la propagation du changement. Nous discutons dans cette section les techniques d'analyse d'impact et de propagation du changement les plus utilisées dans la littérature.

5.3.2.1 Analyse d'impact du changement

En général, l'analyse de l'impact du changement est définie comme l'identification des conséquences potentielles d'un changement et l'estimation de ce qui doit être modifié afin d'accomplir le changement [14]. Elle se focalise sur les détails de l'ingénierie selon deux aspects : la traçabilité et la dépendance [15]. La traçabilité s'intéresse aux liens entre les exigences, les spécification, les artéfacts de modélisation, les artéfacts d'implémentation afin de déterminer la portée nécessaire pour initier un changement. La dépendance s'intéresse plutôt aux relations entre les constituants du système afin de déterminer les conséquences de ce changement [87].

5.3.2.2 Transformation incrémentale

La transformation des modèles consiste à passer d'un modèle source écrit dans un langage donné à un modèle cible écrit dans un langage identique ou distinct. A chaque modification du modèle source, le modèle cible doit être cohérent avec la transformation. Pour garder le modèle cible toujours cohérent avec la transformation du modèle source, deux approches sont possibles : la première repose sur la re-génération entière des modèles cibles. Cette technique est désignée par le terme *re-transformation*. L'autre, appelée *live transformation* [57], consiste à faire la transformation seulement sur les parties modifiées. La figure 5.3 illustre la différence entre ces deux approches.



(a) Re-transformation. (b) Live transformation.

FIGURE 5.3 – Incremental transformation [57]

Dans cette figure, les modèles source et cible sont présentés respectivement sur les parties gauche et droite, et le contexte de transformation dans le milieu. L'approche *Live transformation* facilite la propagation des changements des modèles sources vers les modèles cibles sans la re-génération entière des modèles cibles et est donc plus efficace lorsque les modifications apportées aux modèles sources affectent seulement une partie des modèles cibles.

La transformation incrémentale a été utilisé essentiellement pour traiter la scalabilité de transformations de modèles. Pour les modèles de grande taille, le temps d'exécution de la transformation peut être considérablement réduit en utilisant la transformation incrémentale [57]. Cependant, il a été suggéré que la scalabilité doit

être traitée non seulement en essayant de développer des techniques pour rendre la transformation des modèles plus rapide, mais aussi en fournissant des principes, des pratiques et des outils pour construire des modèles qui sont moins monolithique et plus modulaire. [67]. Pour ce faire, la recherche dans la synchronisation de modèles devrait se concentrer sur la maintenabilité ainsi que la scalabilité.

Plusieurs approches de synchronisation de modèles proposées dans la littérature sont basées sur la transformation incrémentale.

5.3.2.3 Vers une synchronisation automatique de modèles

Plusieurs travaux fournissant des fondements pour l'automatisation de synchronisation de modèles ont été proposés à travers la littérature. Les concepts de traçabilité présentés au chapitre 3 représentent l'une des solutions qui peut faciliter l'automatisation de synchronisation de modèles. Dans la plupart des approches proposées pour la synchronisation de modèles, les auteurs s'intéressent à la synchronisation de modèles avec d'autres modèles. Cependant, très peu de travaux travaillent sur la synchronisation entre modèles et autres types d'artéfacts.

La synchronisation entre modèles et texte propose de modifier automatiquement un modèle en fonction des modifications effectuées sur le code généré ou de modifier un texte généré à partir des modifications effectuées sur le modèle. La trace de modèles vers texte peut être utilisée pour identifier, soit les éléments relatifs aux blocs de code modifiés, soit les blocs de code relatifs à l'élément modifié.

Giese et Wagner [50] propose une solution qui synchronise les modèles de manière bidirectionnelle, i.e. chaque modification du modèle source entraîne une mise à jour du modèle cible et vice versa. La transformation est ici considérée comme un méta-modèle de correspondance qui mappe les deux méta-modèles. L'approche est basée sur TGG (triple graphe grammar) composé d'un modèle source, une instanciation du méta-modèle de transformation et le modèle cible, avec des liens de mapping entre les classes de correspondance et les objets des deux modèles. L'algorithme exploite les nœuds de correspondance qui sont concernés par les modifications. Il les projette sur l'autre modèle afin de le synchroniser avec le modèle source.

5.3.3 Evolution de Transformation

Cette évolution peut être conséquente à une évolution du domaine, de la plate-forme, etc. L'évolution des transformations de modèles doit être effectuée dans l'un des deux cas suivants :

1. Lorsque les métamodèles source ou cible sont modifiés, les transformations doivent être adaptés afin de prendre en considération les concepts ajoutés ou modifiés.
2. Lorsque la plate-forme technique a évolué suite à l'apparition de nouvelles exigences.

L'évolution des transformations de modèles n'a fait l'objet que de très peu de travaux de recherche. Généralement, l'évolution est gérée par des techniques de composition de transformation de modèles.

Dans [92], les auteurs proposent une approche de transformation de modèles basée sur les ontologies. Cette approche intègre les ontologies pour permettre la génération et l'évolution automatiques des transformations de modèles. Leur proposition décrit des méthodes pour générer et adapter les transformations de modèles en dépit des différences structurelles et sémantiques des métamodèles. Différents formats de représentation et différentes sémantiques sont résolus par l'annotation des métamodèles en utilisant les concepts d'une ontologie de référence.

Dans [35], Etien et al. proposent un mécanisme permettant la réutilisation des transformations dans une chaine de transformation de modèles. Ce mécanisme permet l'enchaînement de deux transformations de modèles ayant des métamodèles d'entrée et de sortie incompatibles et qui doivent travailler conjointement pour construire une seule sortie. L'idée de base est d'offrir la possibilité d'étendre une transformation existante afin d'obtenir une nouvelle transformation ayant comme métamodèle source une version étendue du métamodèle d'origine.

Dans [46], les auteurs proposent une approche comportant deux étapes :

— *Étape de détection :* elle consiste à détecter des changements entre la version initiale et la version évoluée du méta-modèle. Les changements peuvent être « simples » ou « complexes ». Les changements simples sont calculés à l'aide

d'un modèle de différences. Des formules de la logique des prédicats sont appliquées sur les changements « simples » pour déterminer ceux qui sont « complexes ».

— *Étape de co-évolution* : cette étape prend en entrée la version initiale et la version évoluée du méta-modèle, ainsi que les changements calculés lors de l'étape précédente. Les changements sont classées dans trois catégories (*sans effet de bord, avec effet de bord et soluble, avec effet de bord et non soluble*) selon leur impact sur la transformation. Des transformations de type HOT sont utilisés pour faire évoluer les modèles de transformation.

5.4 Présentation de l'approche

Le schéma 5.4 donne une vue globale de notre approche de synchronisation de modèles. L'approche se base sur un processus de trois phases : (1) détection et classification des changements *(changes detection and classification)*, (2) analyse d'impact du changement *(perform impact analysis)*, et (3) synchronisation d'artéfacts *(artefacts synchronisation)* [54].

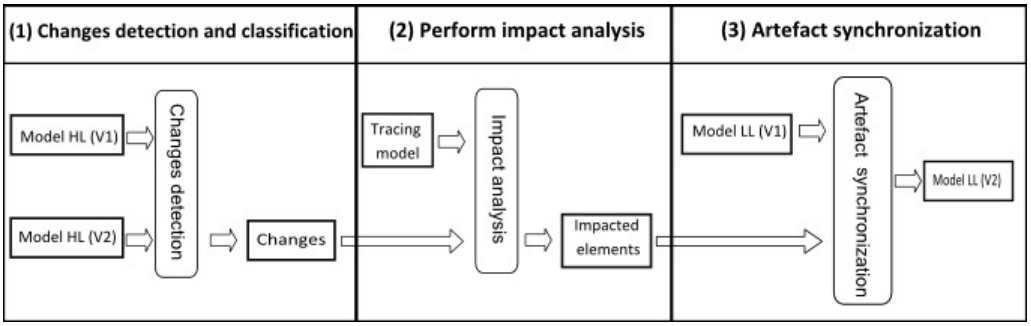


FIGURE 5.4 – Schéma de notre approche de synchronisation [54]

5.4.1 Détection et classification des changements

La première phase consiste à déterminer les différences entre deux versions de modèle pour établir un modèle de différence. De ce modèle, nous effectuons une classification des changements afin de mieux gérer leurs impacts sur les modèles de

sortie.

5.4.1.1 Détection des changements

EMF Compare prend en compte trois types de changements :

— **Attribute changes** : Ce type de modification décrit la différence entre les valeurs d'un attribut donné.

— **Reference changes** : Ce type de changement indique la modification d'une référence d'un élément à un autre.

— **Model Element changes** : Ce type de changement spécifie les éléments ayant été ajoutés ou supprimés.

Dans notre cas, les changements de modèle sont définis par l'ensemble d'opérateurs suivants :

RenameElement

Cette opération consiste à renommer un package, une classe, un attribut, une opération, une association , ou bien une extrémité d'association. L'opération *RenameElement* n'exige pas la ré-exécution des transformations. Car renommer un élément nécessite le changement du nom de chaque artéfact généré à partit de cet élément. La figure 5.5 présente un exemple d'application de l'opération *RenamaAttribute*.

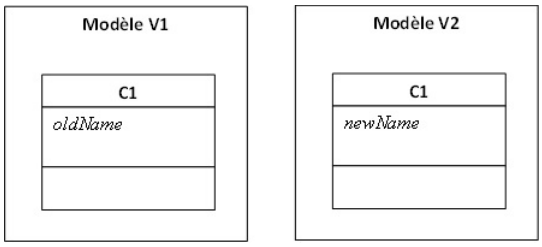


FIGURE 5.5 – Opération *RenameElement*

DeleteElement

Cet opérateur permet de supprimer un package, une classe, un attribut, une opération, une association, ou une généralisation du modèle. Les opérations de sup-

pression sont des opérations qui peuvent intervenir dans un cadre d'évolution modèle. Afin d'assurer la consistance entre le modèle source et le modèle cible, nous proposons de supprimer tous les éléments générés à partir des éléments supprimés. La figure 5.6 présente un exemple d'application de l'opération *DeleteAttribute*.

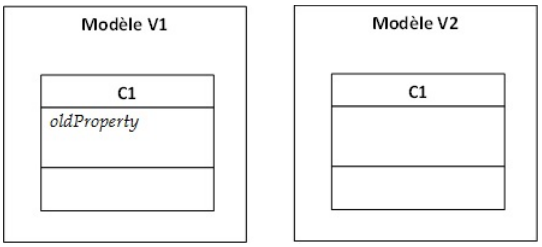


FIGURE 5.6 – Opération *DeleteElement*

AddElement

L'opérateur *AddElement* permet d'ajouter un package, une classe, un attribut, une association, une opération ou une généralisation du modèle. La figure 5.7 présente un exemple d'application de l'opération *AddAttribute*.

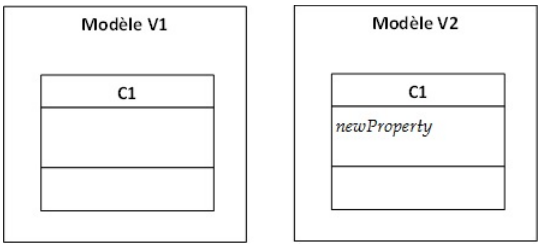


FIGURE 5.7 – Opération *AddElement*

5.4.1.2 Classification des changements

La classification des évolutions est une étape clé pour gérer au mieux leurs impacts sur les modèles. Nous avons donc commencé par établir les évolutions possible d'un modèle d'entrée. Ensuite, Nous avons défini deux catégories d'évolutions.

— **CATEGORIE 1 : *Evolution based on the re-execution processing.***
Dans cette catégorie, une re-exécution de la transformation est nécessaire

pour maintenir la conformité des modéles de sortie avec le modèle évolué. Prenons l'exemple de l'ajout d'une propriété, la règle de transformation correspondante doit être exécutée afin de générer l'élément de sortie associé.

— **CATEGORIE 2 :** ***Evolution without re-execution processing.*** Dans cette catégorie, la co-évolution du modèle de sortie peut s'effectuer sans la re-exécution de la transformation. Prenons l'exemple de la modification du nom d'une propriété, la synchronisation peut s'effectuer en modifiant tout élément avec le même nom et ayant été généré à partir de la propriété renommée.

L'étape de classification consiste à réorganiser les modèles de différence selon les catégories ci-dessus. Le modèle de différence (figure 4.17) montre que trois opérations d'évolution ont été effectuées sur le modèle d'entrée (*Secure bank model*). L'opération de suppression n'exige pas la re-exécution de la transformation, l'opération de modification n'exige pas également la re-exécution de la transformation (cf catégorie 2). Par contre, l'opération d'addition nécessite la re-exécution de la transformation (cf catégorie 1).

5.4.2 Analyse d'impact du changement

L'analyse d'impact du changement est effectuée en se basant sur les modifications détectées pendant la phase de détection et classification des changements, ainsi que sur les liens de trace générés par les transformations et stockées dans le modèle de traces. Les liens de trace sont utilisés afin d'identifier les éléments qui sont touchés par chaque changement des éléments de modèle.

Pour notre exemple, la figure 5.8 montre les éléments impactés par le changement *RenameClass* qui a permis de modifier le nom de la classe UML "Transaction" en "BasicTransaction". Le lien de trace exprime une relation entre la classe UML modifiée *Transaction* et la classe Java générée *Transaction*.

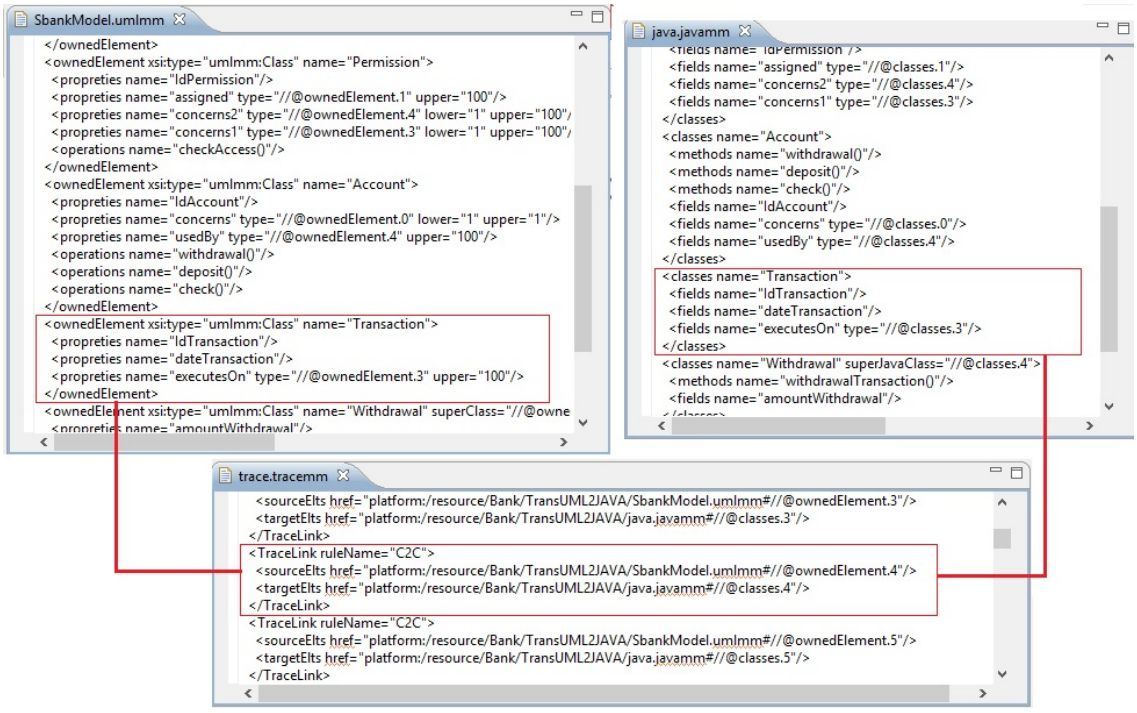


FIGURE 5.8 – Détection des éléments impactés

5.4.3 Synchronisation d'artéfacts

L'étape de synchronisation consiste à adapter les éléments impactés par les changements à la nouvelle version du modèle source.

Pour les évolutions de la catégorie 2, la migration est assurée par une règle de transformation spécifique au patron d'évolution détecté.

Pour les évolution de la catégorie 1, la synchronisation nécessite l'exécution de la transformation définie par la chaine de transformation de modèles.

Dans notre cas, le modèle Java produit à partir de la transformation T2 doit être modifié en réponse aux changements effectués sur le modèle *Secure Bank*.

— L'introduction de la classe *Bank* consiste à ré-exécuter la transformation de manière à générer les éléments correspondants dans le modèle Java.

— La modification appliquée sur la classe *Transaction* doit être propagée en renommant tout élément impacté par cette modification.

— Chaque élément lié à l'élément *balance* doit être supprimé.

5.5 Conclusion

Dans ce chapitre, nous avons introduit l'évolution dans l'IDM en détaillant les problèmes principalement liés aux étroites dépendances entre métamodèles et modèles, métamodèles et transformations, et entre chaque phase de transformation et celles qui la suivent. Le chapitre a également présenté les approches et les techniques les plus connues selon une classification basée sur l'évolution de l'un des artéfacts clés de l'IDM.

Dans ce chapitre, nous avons aussi présenté une approche de synchronisation de modèles permettant d'assurer la consistance entre les modèles suite à l'évolution d'un ou plusieurs éléments des modèles impliqués par la chaîne de transformations. Le travail proposé repose sur les liens de traces générés et maintenus par notre approche de maintenance de la traçabilité présentée dans le chapitre précédent.

Chapitre 6

Conclusion générale

Sommaire

6.1 Rappel du cadre et des objectifs de la thèse 140
6.2 Bilan 140
6.3 Perspectives 142

6.1 Rappel du cadre et des objectifs de la thèse

Dans cette thèse, nous nous sommes concentrés sur les techniques permettant la récupération et la maintenance des liens de trace dans le cadre d'une démarche de développement basée IDM.

L'état de l'art a été divisé en deux parties. Dans la première partie, nous avons présenté les principaux concepts de l'ingénierie dirigée par les modèles (IDM). Nous nous sommes concentrés sur la séparation des préoccupations, plus précisément sur l'intégration des modèles qui est généralement basée sur la *composition de modèles*.

La deuxième partie porte sur la traçabilité dans les transformations de modèles. Dans une première étape, nous avons présenté les techniques les plus représentatives permettant de capturer et de maintenir les relations de traçabilité, après quoi , nous avons effectué une étude comparatives entre ces différentes techniques.

L'objectif principal de notre travail est la proposition d'une approche basée IDM, permettant de prendre en compte à la fois l'aspect fonctionnel et l'aspect non-fonctionnel, pour la maintenance des liens de trace produits par une chaîne de transformations lors de l'évolution de modèles.

La génération et la préservation des liens de traces n'est pas une finalité en soi, ces relations doivent être inspectées et exploitées. Pour cela nous avons proposé une approche de synchronisation de modèles basée sur l'utilisation des liens de traces produits par la chaîne de transformations.

6.2 Bilan

Notre état de l'art nous a conduit à définir deux grands axes de recherche à suivre et nous a permis d'établir notre problématique de recherche. Afin de répondre à cette problématique de recherche, nous avons apporté une proposition à chaque axe de recherche. Nous récapitulons les principales contributions comme suit :

La première contribution de cette thèse est la proposition d'une approche permettant d'assurer l'intégrité des relations de traçabilité générées par une chaîne de transformations et conservées dans un modèle de traces (conforme à un méta-modèle

de traçabilité). Notre approche est basée sur l'analyse de l'impact du changement d'un artefact sur les liens de traces initialement produits par la chaîne de transformations et la propagation des modifications nécessaires afin d'assurer l'intégrité des modèles de trace. Pour l'implémentation du prototype, nous avons utilisé la plate-forme de développement open source Eclipse [55, 56].

Pour pouvoir tracer les préoccupations fonctionnelles et non-fonctionnelles, nous avons proposé un méta-modèle de traçabilité qui nous a permis d'introduire une sémantique sur les liens générés automatiquement au cours de l'exécution de la chaîne des transformations [53].

Nous avons validé notre approche sur une transformation de modèles réalisant le passage d'un diagramme de classes UML vers un modèle Java. La validation est établie en fonction de deux métriques : le rappel et la précision. Le rappel nous a permis de savoir si des modifications ont été oubliées et la précision, quant à elles, nous a permis de juger la qualité des changements effectués par notre approche.

La seconde contribution est une solution permettant d'assurer la synchronisation de modèles en exploitant les liens de trace produits par les transformations [54]. Notre solution est basée sur l'analyse d'impact et la propagation des changements. Les différences entre la version initiale et la version évoluée du modèle sont classés en deux catégories : (1) des changements qui nécessitent la ré-exécution des transformations et (2) des changements qui ne l'exigent pas. Dans le premier cas, la conformité entre les modèles de sortie et le modèle évolué est assurée par la re-exécution de la transformation définie par la chaine de transformation. Dans le deuxième cas, une analyse d'impact du changement est effectuée en se basant sur les liens de trace générés par les transformations et stockées dans le modèle de traces. Les liens de trace sont utilisés afin d'identifier les éléments qui ont été touchés par cette modification. La co-évolution des artéfacts impactés par ce changement est assurée par l'exécution d'une règle de transformation spécifique au type de changement détecté. .

6.3 Perspectives

Notre travail de thèse ouvre des perspectives scientifiques à court et à long terme. Nous soulignons à présent les perspectives qui nous semblent pertinentes pour l'évolution des propositions réalisées.

Nous envisageons améliorer le taux de rappel en appliquant notre approche à plusieurs scénarios d'évolution plus complexes afin de détecter les séquences d'opérations pouvant contribuer à sa dégradation. Les résultats obtenus peuvent être exploités afin d'ajouter une phase de validation du modèle de différences basée sur l'exploitation de l'historique des opérations de changement effectuées sur le modèle initial.

La définition d'un méta-modèle est une tache complexe car elle nécessite d'établir des liens de différents types entre différentes catégories d'objets, et qu'on ne peut proposer un seul modèle pour les prendre en compte. L'extension du méta-modèle de trace en introduisant d'autres types de liens spécifiques à un domaine de recherche particulier constitue une autre perspective à nos travaux.

Bibliographie

[1] A. Adeel and H. Basson. Software evolution modelling : an approach for change impact analysis. In *Proceedings of the 7th International Conference on Frontiers of Information Technology*, New York, NY, USA, 2009. 122

[2] A. Agrawal, G Karsai, S. Neema, F. Shi, and A. Vizhanyo. The design of a language for model transformations. *Software and System Modeling*, 5(3) :261–288, 2006. 23

[3] N. Aizenbud-Reshef, B.T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Syst.*, page http ://www.omg.org/mda/, 2006. 9, 55

[4] N. Aizenbud-Reshef, R.F. Paige, J. Rubin, Y. Shaham-Gafni, and D.S. Kolovos. Operational semantics for traceability. In *ECMDA-TW : Traceability Workshop, at European Conference on Model Driven Architecture*, Nuremberg, Germany, 2005. 71, 76, 77

[5] J. P. Almeida, P. van Eck, and M.-E. Iacob. Requirements traceability in model-driven development : Applying model and transformation conformance. *Springer Science + Business Media, LLC 2007*, 9(2007) :327–342, 2007. 68, 75, 77, 78

[6] R. Baeza-Yates and B. Ribeiro-Neto. Modern information retrieval. *Addison-Wesley Professional*, 1999. 110

[7] E. Baniassad and S. Clarke. Theme : An approach for aspect-oriented analysis and design. In *Proceedings of the International Conference on Software Engineering*, 2004. 39

[8] M. Belaunde and G. Dupé. Smartqvt. *http ://sourceforge.net/projects/smartqvt/.*, 2006. 28

[9] K.G. Berg van den, B. Tekinerdogan, and H. Nguyen. Analysis of crosscutting in model transformations. In *ECMDA Traceability Workshop, SINTEF Report*, 2006. 56

[10] J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2) :171–188, 2005. 14

[11] J. Bézivin, S. Bouzitouna, M.D. Del Fabro, M. Gervais, F. Jouault, D. Kolovos, Kurtev I., and R. Paige. A canonical scheme for model composition. In *Proc. of ECMDA-FA, LNCS 4066*. Springer-Verlag, 2006. 44

[12] J. Bézivin and O. Gerbe. Towards a precise definition of the omg/mda framework. *Los Alamitos, CA, USA*, pages 273–280, 2005. 16

[13] J. Bézivin, F.c Jouault, P. Rosenthal, and P. Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture, European MDA Workshops : Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers*, pages 33–46, 2004. 15

[14] S. A. Bohner. Impact analysis in the software change process : a year 2000 perspective. In *ICSM*, Washington, DC, USA, 1996. IEEE Computer Society. 129

[15] S. A. Bohner. Extending software change impact analysis into cots components. In *SEW*, Washington, DC, USA, 2002. IEEE Computer Society. 129

[16] Borland. Qvt-o. *http ://www.eclipse.org/m2m/qvto/doc.*, 2007. 28

[17] C. Brun and A. Pierantonio. Model differences in the eclipse modelling framework. *UPGRADE : The Europ. Journal for the Informatics Professional*, 2008. 104

[18] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery : A taxonomy. *IEEE Softw.*, 7 :13–17, 1990. 121

[19] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. *In 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231, 2008. 129

[20] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Meta-model differences for supporting model co-evolution. In *Proc. of the 2nd Int. Workshop on Model-Driven Software Evolution, MoDSE '08, Athens (Greece)*, 2008. 127, 128

[21] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9) :165–185, 2007. 104

[22] T. Clark, P. Sammut, and J. Willans. Applied Metamodelling – A Foundation for Language Driven Development. Second Edition, 2008. 16

[23] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design : The Theme Approach*. The Addison-Wesley Object Technology Series. Addison Wesley Professional, 2005. 35, 38

[24] J. Cleland-Huang, C.K. Chang, and M. Christensen. Event-Based traceability for managing evolutionary change. 29(9) :796–810, 2003. 67, 75, 76, 77

[25] J. Cleland-Huang, O. Gotel, J.H Hayes, P. Mäder, and A. Zisman. Software traceability : trends and future directions. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 55–69, 2014. 9

[26] J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhan-skaya, and S. Christina. Goal-centric traceability for managing non-functional requirements. In *Proc. of the 27th International Conference on Software Engineering*, 2005. 69, 75, 76, 77, 78

[27] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*, Anaheim, California, USA, 2003. 22, 25

[28] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal - Model-driven software development*, 45(3) :621–645, 2006. 22, 24

[29] S. Diaw, R. Lbath, and B. Coulette. Etat de l'art sur le développement logiciel basé sur les transformations de modèles. *Technique et Science Informatiques, Ingénierie Dirigée par les Modéles*, 29(4-5/2010) :505–536, juin 2010. 16

[30] M. Didonet Del Fabro, J. Bézivin, E. Jouault, F. Breton, and G. Gueltas. Amw : a generic model weaver. In *Actes IDM'05. Paris, France*, 2005. 88

[31] M. Didonet Del Fabro, J. Bézivin, and P. Valduriez. Weaving models with the eclipse amw plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006*, 2006. 37, 44, 45, 46

[32] R. Dirckze. Java(TM) metadata interface (JMI) specification. *Version 1.0. Unisys, 1.0 edition*, 2002. 23

[33] O. Djebbi and M. P. Gervais. Mda :vers l'industrialisation de la construction d'application réparties. *Rapport interne DEA SIR. Lip6*, 2004. 21

[34] A. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 29(2) :116–132, 2003. 70, 75, 77, 78

[35] A. Etien, A. Muller, T. Legrand, and X. Blanc. Combining independent model transformations. In *Proceedings of the 2010 ACM Symposium on Applied Computing, SAC'10*, New York, USA, 2010. 132

[36] J. Falleri, M. Huchard, and C. Nebut. Towards a traceability framework for model transformations in kermeta. In *ECMDA Traceability Workshop*, 2006. 74, 75, 76, 77, 78

[37] F. Fleurey. Kompose : a generic model composition tool. *Available from :http ://www.kermeta.org/kompose/*, 2007. 44, 50

[38] F. Fleurey, B. Baudry, R. B. France, and S. Ghosh. A generic approach for automatic model composition. In *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, pages 7–15, 2007. 50

[39] Eclipse Foundation. Atlas transformation language. *http ://www.eclipse.org/atl/*, 2013. 84

[40] Eclipse Foundation. Graphical modeling framework project. *Disponible sur http ://www.eclipse.org/gmf/*, 2013. 84

[41] M. Fowler. Refactoring : improving the design of existing code. *Addison-Wesley, Upper Saddle River, New Jersey*, 1999. 121

[42] R. France, I. Ray, G. Georg, and S. Ghosh. Aspect-oriented approach to early design modelling. *IEE Proceedings Software*, 151(4) :173–185, 2004. 35, 38, 40

[43] R. France and B. Rumpe. Model-driven development of complex software : A research roadmap. *2007 Future of Software Engineering, FOSE '07*, 2007. 36

[44] I. Galvão and A. Goknil. Survey of traceability approaches in model-driven engineering. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, 2007. 62, 76, 78

[45] E. R. Gansner and S. C. North. An open graph visualization system and its applications. *Software - Practice and Experience*, 30 :1203–1233, 1999. 77

[46] J. Garcea, O. Diaz, and M. Azanza. Model transformation co-evolution : a semi-automatic approach. *In Software Language Engineering, K. Czarnecki and S. Hedin, Eds.*, vol. 7745 of Lecture Notes in Computer Science :144–163, Springer Berlin Heidelberg, 2012. 132

[47] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA '09, Berlin, Heidelberg*. Springer-Verlag, 2009. 127

[48] GEF. The graphical editing framework (gef). In *Phttp ://www.eclipse.org/*. 84

[49] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation : The missing link of MDA. In *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings. LNCS vol. 2505, Springer Verlag, 2002*, pages 90–105, 2002. 23

[50] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. In *Software and Systems Modeling*, volume 8, pages 21–43, 2008. 131

[51] O.C.Z. Gotel and A.C.W. Finkelstein. An analysis of the requirements traceability problem. *In Proc. 1st International Conf. on Req. Eng.*, pages 49–101, 1994. 9, 55, 62

[52] B. Gruschko, D.S. Kolovos, and R.F. Paige. Towards synchronizing models with evolving metamodelsn. In *Proc. Workshop on Model-Driven Software Evolution, co-located with the European Conference on Software Maintenance and Reengineering (CSMR)*, 2007. 126

[53] M. Y. Haouam and D. Meslati. A metamodel for concerns traceability. In *Proceeding of the 2nd International Conference on Information Systems and Technologies (ICIST'12)*, Sousse, Tunisia, 2012. 84, 98, 141

[54] M. Y. Haouam and D. Meslati. A traceability based approach for model synchronization. *International Journal of Software Engineering and Its Applications*, 7(6) :305–318, 2013. 119, 133, 141

[55] M. Y. Haouam and D. Meslati. Approach to traceability maintenance. In *Proceeding of the International Conference on Advanced Technology and Sciences (ICAT'14)*, Antalya, Turkey, 2014. 84, 141

[56] M. Y. Haouam and D. Meslati. Towards automated traceability maintenance in model driven engineering. *IAENG International Journal of Computer Science*, 43(2) :147–155, 2016. 84, 103, 104, 141

[57] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In *O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, Proc. International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of Lecture Notes in Computer Science, pages 321–335. Springer, 2006. 130

[58] W. Heaven and A. Finkelstein. UML profile to support requirements engineering with KAOS. *IEE Proceedings - Software*, 151(1) :10–28, 2004. 57

[59] M. Herrmannsdoerfer1, S. D. D. Vermolen, and G. Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. *SLE*, Springer-Verlag. :163–182, 2011. 129

[60] A. Hessellund, K. Czarnecki, and A. Wasowski. Guided development with multiple domain-specific languages. In *Proceedings of the ACM/IEEE 10th International Conference On Model Driven Engineering Languages and Systems*, 2007. 37

[61] IEEE. IEEE standard computer dictionary : a compilation of IEEE standard computer glossaries. *IEEE Computer Society Press, New York, NY, USA*, 1991. 55

[62] IEEE. IEEE standard 1219-1999 for software maintenance. In *IEEE Press*, 1999. 119

[63] M. Joaquin and M. Jishnu. MDA guide version 1.0.1. *omg/03-06-01. Object Management Group*, 2003. 15

[64] F. Jouault. Loosely coupled traceability for ATL. *In ECMDA Workshop on Traceability*, 2005. 58, 60, 73, 76, 77, 78, 98

[65] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl : A model transformation tool. *Science of Computer Programming*, pages 31–39, 2008. 24, 27, 29, 84

[66] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingier, and J. Irwin. Aspect oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, May 1997. 34, 35, 36

[67] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. The grand challenge of scalability for model driven engineering. *M.R. Chaudron, editor, Models in Software Engineering : Workshops and Symposia at MoDELS 2008, Reports and Revised Selected Papers*, 5421 of Lecture Notes in Computer Science :48–53, 2008. 131

[68] R. Laddad. *AspectJ in Action*. Practical Aspect-Oriented Programming. Manning Publications, Greenwich, UK, 2003. 36

[69] B. P. Lientz and E. B. Swanson. Software maintenance management. *Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA*, 1980. 119

[70] J. Ludewig. Modeling in the large and modeling in the small. *Models in software engineering - an introduction*, 2003. 15

[71] L. M. Rose, M. Herrmannsdoerfer, J. R. Williams, D. S. Kolovos, K. Garces, R. F. Paige, and F. A.C.Polack. An analysis of approaches to model migration. In *Proceedings of the Joint MoDSE-MCCM Workshop*, 2009. 124

[72] P. Mäder and O. Gotel. Towards automated traceability maintenance. *The Journal of Systems and Software*, 4(85) :2205–2227, 2012. 72, 76, 77, 78

[73] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceeding of the International Conference on Software Engineering*, pages 125–135, Portland, Oregon, USA, 2003. 125

[74] F. Marschall and P. Braun. Model transformations for the MDA with BOTL. 2003. 23

[75] T. Mens and P. Van Gorp. A taxonomy of model transformation. In *Proc. International Workshop on Graph and Model Transformation*, 2006. 23

[76] B. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. International Technical Support Organization. IBM Redbooks, 2004. 16, 17

[77] P.A. Muller, F. Fleury, and J.M. Jezequel. Weaving executability into objectoriented meta-languages. In *Proceedings of MODELS/UML 2005*, Montego Bay, Jamaica, 2005. 27

[78] P. Naur and B. Randell. Software engineering. *NATO, Scientific Affairs Division, Brussels*, 1969. 119

[79] OMG Object Management Group. XML metadata interchange specification. *Version 2.0.1, formal/05-05-06*, 2005. 17

[80] OMG Object Management Group. Meta object facility (mof) 2.0 core specification. *Available Specification*, 2006. 16, 17, 58

[81] OMG Object Management Group. Meta object facility (mof) 2.0 query/-view/transformation (QVT) specification. *OMG*, 2008. 27

[82] OMG Object Management Group. Object constraint language. *http ://www.omg.org/spec/OCL/2.2/*, 2010. 28

[83] J. Oldevik. Mofscript user guide. 2011. 27, 31

[84] G. K. Olsen and J. Oldevik. Scenarios of traceability in model to text transformations. In *Model Driven Architecture- Foundations and Applications, Third European Conference, ECMDA-FA 2007, Proccedings*, pages 144–156, 2007. 61

[85] OMG. OMG model driven architecture. *page http ://www.omg.org/mda/*, 2012. 19

[86] R.F. Paige, G.K. Olsen, D.S Kolovos, S. Zschaler, and C. Power. Building model-driven engineering traceability classifications. In *ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceedings*, pages 49–58, Sintef, Trondheim, 2008. 55

[87] V. Rajlich. A model and a tool for change propagation in software. *SSEN*, 25(1), 2000. 129

[88] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1) :58–93, 2001. 63, 65, 66, 75, 77

[89] A. Rashid and P. Sawyer. A database evolution taxonomy for object oriented databases. *JSME : Research and Practice*, 17(2) :93–141, 2005. 129

[90] R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development*, LNCS 3880 :75–105, 2006. 41

[91] L. Rose, D. Kolovos, R. Paige, F. Polack, and S. Poulding. Epsilon flock : a model migration language. *Software and Systems Modeling*, 2012. 124

[92] S. Roser and B. Bauer. Automatic generation and evolution of model transformations using ontology engineering space. *Journal on Data Semantics XI, Springer-Verlag, Berlin, Heidelberg*, pages 32–68, 2008. 132

[93] D. S. Kolovos, R. F. Paige, and F. A.C.Polack. Merging models with the Epsilon Merging Language (EML). In *Second European Conference on*

Model-Driven Architecture Foundations and Applications (ECMDA'06)., Bilbao, Spain, 2006. 44, 47, 48, 49

[94] D. S. Kolovos, R. F. Paige, and F. A.C.Polack. On-demand merging of traceability links with models. In *Second European Conference on Model-Driven Architecture Foundations and Applications (ECMDA'06).*, Bilbao, Spain, 2006. 57

[95] M. Sabetzadeh and S. Easterbrook. An algebraic framework for merging incomplete and inconsistent views. In *Proceedings of the 13th IEEE International Requirements Engineering Conference*, September, 2005. 35

[96] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel. A survey on aspect-oriented modeling approaches. *Relatorio tecnico, Vienna University of Technology*, 2007. 36, 41, 42

[97] S. Sendall and W. Kozaczynski. Model transformation : The heart and soul of model-driven software development. *IEEE Software*, 20(5) :42–45, 2003. 23

[98] D.I.K. Sjoberg. Quantifying schema evolution. *Information and Software Technology*, 35(1) :35–44, 1993. 120

[99] R. M. Soley. Model driven architecture : Three years on. In *On The Move to Meaningful Internet Systems 2003 : CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003*, pages 1048–1049, 2003. 14

[100] I. Sommerville. Software engineering. *Addison-Wesley, Boston, Massachusetts, 9th edition*, 2006. 120

[101] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. Emf : Eclipse modeling framework. *Addison-Wesley, 2. edition*, 2009. 18, 30, 84

[102] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. *In Model Driven Architecture - Foundations and Applications.*, Springer Berlin / Heidelberg, 2009. 73

[103] TOPCASED-WP5. Guide méthodologique pour les transformations de modèles. *Rapport de recherche, version 0.1, 18 Novembre 2008, IRIT/MACAO*, 2008. 26

[104] A. Van Deursen, E. Visser, and J. Warmer. Model-driven software evolution : A research agenda. In *Proceedings of ECMDA Traceability Workshop (ECMDA-TW) '08*, Int. Workshop on Model-Driven Software Evolution held with the ECSMR '07, 2007. 122

[105] G. Wachsmuth. Metamodel adaptation and model co-adaptation. *Erik Ernst, editor, ECOOP 2007 Object-Oriented Programming, Springer Berlin*, 4609 of Lecture Notes in Computer Science :600–624, 2007. 125, 129

[106] J. Whittle, P. K. Jayaraman, A. M. Elkhodary, A. Moreira, and J. Araújo. MATA : A unified approach for composing UML aspect models based on graph transformation. *T. Aspect-Oriented Software Development*, 6 :191–237, 2009. 38, 42, 44

[107] E. D. Willink. On challenges for a graphical transformation notation and the UMLX approach. *Electr. Notes Theor. Comput. Sci.*, 211 :171–179, 2008. 23

[108] Andrés Yie and Dennis Wagelaar. Advanced traceability for atl. In *International Workshop on Model Transformation with ATL*, France, 2009. 60

Contributions Scientifiques

Publications dans des revues internationales avec comité de lecture

— M. Y. Haouam and D. Meslati. *"A traceability based approach for model synchronization"*. International Journal of Software Engineering and Its Applications, 7(6) :305-318, 2013.

— M. Y. Haouam and D. Meslati. *"Towards automated traceability maintenance in model driven engineering"*. IAENG International Journal of Computer Science, 43(2) :147-155, 2016.

Conférences internationales avec comité de lecture

— M. Y. Haouam and D. Meslati. *"A metamodel for concerns traceability"*. In Proceeding of the 2nd International Conference on Information Systems and Technologies (ICIST'12), Sousse, Tunisia, 2012.

— M. Y. Haouam and D. Meslati. *"Approach to traceability maintenance"*. In Proceeding of the International Conference on Advanced Technology and Sciences (ICAT'14), Antalya, Turkey, 2014.