ACADÉMIE DE MONTPELLIER
UNIVERSITÉ **M**ONTPELLIER **II**
Sciences et **T**echniques du **L**anguedoc

THÈSE

présentée au Laboratoire d'Informatique de Robotique
et de Microélectronique de Montpellier pour
obtenir le diplôme de doctorat

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Information, Structures, Systèmes**

Génération de Transformations de Modèles : une approche guidée par les treillis de Galois

par

Xavier DOLQUES

Soutenue le 18 Novembre 2010, devant le jury composé de :

Directrice de thèse
Marianne HUCHARD, professeur ..... LIRMM, Université Montpellier II

Co-encadrante de thèse
Clémentine NEBUT, maître de conférence ..... LIRMM, Université Montpellier II

Rapporteurs
Jean-Marc JÉZÉQUEL, professeur ..... Université Rennes 1
Amedeo NAPOLI, directeur de recherche CNRS ..... LORIA, Nancy
Houari SAHRAOUI, professeur ..... Université de Montréal

Président du jury
Xavier BLANC, professeur ..... Université Bordeaux 1

Examinateurs
Éric BOURREAU, maître de conférence ..... LIRMM, Université Montpellier II

Remerciements

Une thèse se déroule sur une période de trois ans. Trois ans riches en évènements et en rencontres, et dont le résultat est fortement influencé par l'entourage. Cette section est dédiée à tous ceux qui ont contribué durant ces trois ans, directement ou indirectement, à la production de ce document. J'attire ici l'attention du lecteur sur la non-exhaustivité de l'énumération qui va suivre, et sur la profonde injustice dont ma mémoire va se rendre coupable par de nombreux oublis dont ledit lecteur sera peut être la victime. J'invite ainsi les oubliés à me faire part de leur mécontentement, et je ferai en sorte de réparer mes torts autour d'un verre. Par ailleurs il est à noter que l'ordre de cette énumération n'a pas de signification particulière et que les énumérations de personnes se font en tenant compte de l'ordre lexicographique sur les prénoms. J'invite donc les derniers à ne pas en prendre ombrage.

Pour commencer je remercie les membres du jury qui ont accepté de relire la première version du document avec un regard critique et l'on jugé suffisamment pertinent pour me permettre de soutenir la thèse qu'il présente. Je remercie par la même occasion toutes les personnes présentes lors de la soutenance.

Je remercie ma famille, en particulier mes parents, ma sœur et son mari, qui m'ont apporté leur soutien bien avant la thèse et qui continuent encore aujourd'hui. Ils ont su être là quand il fallait, et ont accepté mes choix sans condition, sans poser de questions. Ils se sont sans doute inquiétés plus que de raison et ont tout fait pour m'aider lorsqu'ils le pouvaient.

Je remercie le LIRMM de bien avoir voulu m'héberger durant 3 ans. Je remercie en particulier les membres du département informatique pour leur ouverture et leur bonne humeur. Je remercie l'équipe D'OC de m'avoir accueilli, avec une mention particulière pour Yolande qui fut une voisine de bureau très gentille et très amicale pour le thésard qui venait d'arriver.

Je remercie les étudiants auprès desquels j'ai pu enseigner pour m'avoir fait découvrir les joies et les frustrations du métier d'enseignant.

Je remercie mes camarades de master 2 avec qui j'ai passé de bons moments durant les différents enseignements et surtout durant le stage qui a suivi [1]. Ce stage engendra un channel IRC qui encore aujourd'hui nous permet de garder contact, j'en remercie donc les

1. merci AB

membres qui m'ont soutenu durant ces 3 ans et ont été une source inestimable d'informations utiles (ou pas) : Ercete, Fannoche, Fifre, Gilink, Kiki, Nomemie, Psykoo, Seb, Shura, Thuamley, Valek.

Je remercie par ailleurs les doctorants compagnons de galère du LIRMM : Abdoulkader, Antoine, Aymen, Benoît V., Benoît D., Floréal, Gilles, Guillaume, Jean, Jean-Baka, Jean-Rémy, Julien, Khalil, Lisa, Nadia, Paola, Pattaraporn, Pierre, Philippe, Raluca, Thibault, Thomas, Yuan, Zeina ... [2] Je remercie les différents doctorants que j'ai pu rencontré durant les activités de formation de la thèse, telle que les formations CIES et surtout l'organisation des DOCTISS. Je remercie tous les participants des séminaires du vendredi ainsi que les éditeurs des papiers présentés. Je remercie les participants des soirées *coin*, en particulier Chloë et Soffana qui n'ont pas encore été citées. Je remercie aussi les personnes que j'ai rencontré au cours de mes différents déplacements, en particulier Assel avec qui j'ai passé d'excellent moments. Je remercie toutes les personnes que j'ai pu croisé, avec qui j'ai pu discuter, partager une pause café, manger, boire ou qui m'aurait tout simplement gratifié d'un sourire ou d'une poignée de main et que par maladresse j'aurai oublié dans ces remerciements.

Enfin, je souhaite remercier tout particulièrement mes deux encadrantes de thèse Clémentine et Marianne pour tout ce qu'elles ont pu faire pour moi durant ces trois ans. Merci tout d'abord d'avoir fait le nécessaire pour me garder en thèse après les six mois de stage passés ensembles. Merci d'avoir été là quand j'en ai eu besoin, et d'avoir été d'une fiabilité à toute épreuve [3] dans les périodes critiques, en particulier lors de la rédaction de ce document. Merci de m'avoir laissé toute latitude dans mes choix de recherche(ou tout du moins de m'en avoir laissé l'illusion). Merci pour tous vos conseils et pour les engagements que vous avez pu prendre pour m'aider dans mon travail. Merci enfin pour toutes les qualités humaines dont vous avez fait preuve et qui font que je n'aurai pas voulu d'autres encadrants. J'espère que ce document répond à vos attentes et que j'aurai encore le plaisir de pouvoir travailler avec vous.

Pour conclure, je remercie le lecteur qui donne son sens à ce document en le lisant et j'espère que ce dernier répondra à ses attentes.

2. si vous êtiez doctorants et que vous ne vous voyez pas dans la liste, c'est que vous êtes sûrement dans les points de suspension

3. à part peut-être pour la ponctualité des débuts de réunion, mais je suis loin d'être tout blanc de ce côté là...

Table des matières

Remerciements **i**

Table des matières **iii**

Table des figures **v**

Liste des tableaux **vii**

1 Introduction **1**
1.1 Contributions 2
1.2 Plan 4

2 État de l'art **5**
2.1 Ingénierie dirigée par les modèles 5
2.2 Les Transformations de modèle 8
2.3 Génération de transformations de modèles 10
2.3.1 Génération de transformations de modèle basée sur l'alignement de Méta-modèles 10
2.3.2 Génération de transformations de modèle basée sur des modèles d'exemple 12
2.4 Conclusion 16

3 Alignement de modèles **19**
3.1 Aperçu sur les méthodes d'appariement 20
3.2 Appariement de modèles 23
3.2.1 Exemple illustratif 24
3.2.2 Appariement basé sur les valeurs d'attributs 25
3.2.3 Adaptation de l'approche AnchorPROMPT 31
3.3 Étude de cas 37
3.3.1 Outil 37
3.3.2 Sélection des exemples 39
3.3.3 Métriques 45

3.3.4 Résultats . . . 47
3.4 Conclusion . . . 51

4 Génération de transformations 53
4.1 Exemple illustratif . . . 54
4.2 Analyse Relationnelle de Concepts . . . 56
4.2.1 Analyse Formelle de concepts . . . 56
4.2.2 Analyse Relationnelle de concepts . . . 58
4.3 Spécification de l'approche de génération de transformation . . . 67
4.3.1 Classification des éléments du modèle . . . 67
4.3.2 Classification des liens d'appariements . . . 69
4.3.3 Interprétation des treillis et création de règles . . . 72
4.3.4 Gestion des liens dans le modèle cible . . . 74
4.4 Étude de cas . . . 75
4.4.1 Outil . . . 75
4.4.2 Résultats . . . 75
4.5 Conclusion . . . 80

5 Conclusion et Perspectives 87
5.1 Perspectives . . . 88
5.1.1 Alignement de modèles . . . 89
5.1.2 Génération de transformation de modèles . . . 90

Bibliographie 93

A Liste détaillée des exemples pour les études de cas 99
A.1 Transformations endogènes . . . 99
A.2 Transformations exogènes . . . 100

B Correction de problèmes de généralisation dans les diagrammes de cas d'utilisation UML 105

Table des figures

1.1 Schématisation de l'approche de développement de transformation de modèles. 3

2.1 Pile de méta-modélisation représentant les différents niveaux de modélisation. 7

3.1 Modèle d'exemple source en UML. 25
3.2 Méta-modèle UML utilisé pour définir le diagramme de la figure 3.1. 25
3.3 Représentation sous forme de diagramme d'objets du diagramme UML de la figure 3.1. 26
3.4 Modèle d'exemple cible en Entité-Association. 26
3.5 Méta-modèle Entité-Association pour définir le diagramme de la figure 3.4. . . . 27
3.6 Représentation sous forme de diagramme d'objets du diagramme Entité-Association de la figure 3.1. 27
3.7 Illustration de la méthode d'appariement utilisée pour AnchorPROMPT. 32
3.8 Illustration des chemins correspondants. 35
3.9 Alignement de deux chemins correspondants. 36
3.10 Alignement de deux chemins correspondants. 37
3.11 Schéma illustrant le fonctionnement de l'outil. 38
3.12 Méta-modèle d'alignement. 39
3.13 *delegation1* : une transformation endogène de modèles UML. Ici sont présentés deux modèles UML, à gauche un modèle source et à droite le modèle obtenu par la transformation *delegation1*. 45
3.14 Séparation des différents ensembles de paires d'appariement. 45
3.15 Calcul de la précision sur les exemples de l'étude de cas. 47
3.16 Calcul du rappel sur les exemples de l'étude de cas. 48
3.17 Calcul de l'effort sur les exemples de l'étude de cas. 49
3.18 Calcul du f-score ($\alpha = 0.5$) sur les exemples de l'étude de cas. 50

4.1 Méta-modèle source de l'exemple illustratif (en gris sont indiqués des identifiants associés à chaque élément auxquels on se référera par la suite). 54
4.2 Méta-modèle cible de l'exemple illustratif (en gris sont indiqués des identifiants associés à chaque élément auxquels on se référera par la suite). 54

4.3 Exemple illustratif : en bas à gauche un modèle *Association* et en haut à droite le modèle *Persons* correspondant (en gris sont indiqués des identifiants associés à chaque élément auxquels on se référera par la suite). 55
4.4 Treillis résultant de l'application de l'AFC au contexte de la table 4.1. 59
4.5 Treillis obtenus à l'issue de l'étape d'initialisation. 62
4.6 Treillis du contexte *modelA* à l'étape 2. 64
4.7 Treillis du contexte *modelA* à l'étape finale. 66
4.8 Treillis issus du contexte modelB de l'exemple illustratif. 70
4.9 Treillis issus du contexte metaModelB de l'exemple illustratif. 71
4.10 Treillis issus de l'application de l'ARC sur le contexte *MapLinks* décrit à la table 4.8. 82
4.11 Représentation concrète du modèle de règles correspondant à l'interprétation des treillis de notre exemple illustratif. 83
4.12 Métamodèle de la représentation des règles. 83
4.13 Illustration de l'interprétation d'un concept du treillis *AssocFC*. 84
4.14 Schéma de fonctionnement de l'outil. 84
4.15 *En haut :* Les deux méta-modèles implémentés, à gauche le méta-modèle que nous appellerons UML et à droite le méta-modèle que nous appellerons Entité-Association. *En bas :* les deux modèles d'exemple avec les liens d'appariement représentés par des lignes en pointillés. 85
4.16 Ce que génère le processus $Proc_{\exists}$ pour la règle 2. 86
4.17 Les règles générées par le processus $Proc_{\forall\exists}$ correspondant aux règles 4 et 5. . . 86

Liste des tableaux

2.1 Récapitulatif des méthodes de génération de transformation décrites. 17
2.2 Récapitulatif des méthodes de génération de transformation à partir de modèles d'exemples. 18

3.1 Instances d'attributs du modèle source (à gauche) et du modèle cible (à droite) de notre exemple. 29
3.2 Alignement obtenu à partir de l'exemple. 32
3.3 Analyse des transformations du zoo ATL (avril 2009) 40
3.4 Extrait choisi des transformations du zoo ATL. 43

4.1 Contexte Formel représentant l'appartenance de liens par des éléments. 57
4.2 Contextes formels pour l'application de l'ARC sur le modèle cible de l'exemple. Contexte décrivant les classes à gauche et contexte décrivant les éléments à droite. 60
4.3 Contextes relationnels pour l'application de l'ARC sur le modèle cible de l'exemple. 61
4.4 Contexte *modelA* enrichi à l'étape 2. 63
4.5 Contexte *modelA* enrichi à l'étape 3. 65
4.6 Contextes formels du modèle cible pour l'application de l'ARC sur l'exemple. Contexte décrivant les classes à gauche et contexte décrivant les éléments à droite. 68
4.7 Contextes relationnels du modèle cible pour l'application de l'ARC sur l'exemple. 69
4.8 Contexte formel regroupant les liens d'appariement de l'exemple illustratif. . . 71
4.9 Contextes relationnels représentant la relation entre les liens d'appariements et les éléments source et cibles qu'ils relient. 72
4.10 Métriques quantitatives sur les treillis des règles obtenus à partir de notre ensemble d'exemples avec une configuration utilisant l'opérateur $S_{\forall\exists}$ (processus $Proc_{\forall\exists}$) 78
4.11 Métriques quantitatives sur les treillis des règles obtenus à partir de notre ensemble d'exemples avec une configuration utilisant l'opérateur $S_{\exists}$ (processus $Proc_{\exists}$) 79

Chapitre 1

Introduction

Longum iter est per praecepta, breve et efficax per exempla. [1] Ce que Sénèque explique à Lucillius, de nombreux développeurs aimeraient pouvoir l'appliquer dans leurs tâches de programmation. Il serait en effet plus facile de montrer à un ordinateur deux listes de nombres, la seconde liste correspondant à la première triée, et de laisser l'ordinateur généraliser et définir les bonnes opérations pour effectuer le tri, plutôt que de développer soi-même un algorithme de tri. Plusieurs tentatives ont été effectuées dans le génie logiciel pour définir un système d'apprentissage de programmes à partir d'exemples, dont un grand nombre sont décrites dans [Lieberman, 2001]. Le potentiel d'un tel système, s'il fonctionnait, serait énorme puisqu'il permettrait à toutes les personnes n'ayant aucune connaissance en programmation de créer des programmes. Malgré tous les efforts déployés, les solutions proposées ne permettent de créer que des programmes très simples pour un coût en termes de temps relativement élevé mais avec l'avantage que les systèmes sont faciles et rapides à prendre en main. Ces systèmes ont pour la plupart été utilisés comme des plateformes d'initiation à la programmation pour les plus jeunes.

Avec l'avènement de l'ingénierie dirigée par les modèles est née la problématique de la programmation de transformations de modèles. De manière simplifiée, un modèle peut être défini comme un ensemble de données représentant un système et écrites dans un langage bien défini particulier que l'on nomme méta-modèle. On peut par exemple citer les méta-modèles UML, qui définit un ensemble de diagrammes permettant de modéliser les différents aspects d'un programme, et Entité-Relation, qui permet la modélisation d'une base de données. Une transformation de modèles peut alors être vue comme un programme modifiant un modèle pour en changer les données et/ou le réécrire dans un autre langage. Les précédents exemples cités font l'objet d'une transformation souvent utilisée comme référence dans la littérature et qui consiste à transformer un diagramme de classes UML en un modèle Entité-Relation. Cette transformation sera décrite plus en détail dans

1. « La voie du précepte est longue, celle de l'exemple courte et efficace » *Lettres à Lucilius*, Sénèque

la section 3.2.1. Un programme de transformation de modèles doit pouvoir s'appliquer sur n'importe quel modèle écrit dans un langage donné.

La création d'une transformation de modèles fait généralement intervenir différents acteurs :

- Les experts métiers, spécialistes d'un domaine métier mais n'ayant pas de connaissance des transformations de modèles. On fait appel à eux pour la réalisation de modèles en entrée d'une transformation et la vérification de modèles en sortie d'une transformation. Pour la création d'une transformation de modèles, l'expert métier spécialiste du domaine du méta-modèle source n'est pas forcément le même que l'expert métier spécialiste du domaine du méta-modèle cible. Il est par exemple rare qu'un spécialiste d'UML ait autant de compétences dans le domaine des bases de données, et inversement.
- Les experts en transformations de modèles, spécialistes des langages de transformation de modèles mais n'ayant pas d'expertise des méta-modèles sources et cibles de la transformation.

Ces deux groupes d'experts sont a priori disjoints et doivent donc collaborer pour permettre la création d'une transformation de modèles.

L'écriture de ces transformations nécessite la création de nouvelles méthodes de programmation pour pouvoir les développer efficacement en prenant en compte les compétences des différents acteurs. Le problème se situant essentiellement dans la difficulté pour les experts métiers de participer activement au développement de la transformation en utilisant leur connaissance du domaine métier.

La thèse que nous défendons est qu'il est possible d'assister efficacement le développement d'une transformation de modèles par apprentissage sur des exemples. Cette approche permet en particulier aux experts métier de prendre une part active dans le développement de la transformation car ils travaillent alors dans le langage qu'ils maîtrisent le mieux, c'est-à-dire le langage de modélisation dont ils sont experts. Leur participation assure une bonne précision au résultat.

1.1 Contributions

Dans ce manuscrit nous proposons un système de génération de transformations guidée par des exemples de modèles, c'est-à-dire que la génération du code de la transformation de modèles sera réalisée en partie grâce à des exemples de l'exécution de la transformation, permettant ainsi aux experts métiers de participer activement à la génération de la transformation sans avoir à maîtriser le langage de transformation utilisé.

Nos contributions sont les suivantes :

- Nous proposons une approche pour générer l'alignement des modèles d'exemples à partir desquels on souhaite générer une transformation. En effet une approche par l'exemple ne peut pas se contenter d'une donnée d'entrée et d'un résultat, elle

a besoin de « voir » l'exécution de la transformation sur les exemples. Cette exécution est représentée par un ensemble de liens indiquant pour un élément du modèle source ce qu'il devient dans le modèle cible. Ainsi lorsque par exemple un élément *Cl_Personne* de type *Class* avec un élément *prop_nom* de type *Property* est transformé en un élément *Ent_Personne* de type *Entity* et un élément *att_nom* de type *Attribute* nous créons les liens (*Cl_Personne,Ent_Personne*) et (*prop_nom,att_nom*) pour que le système comprenne que *Ent_Personne* a été créé à partir de *Cl_Personne* et *att_nom* a été créé à partir de *prop_nom*. L'approche que nous proposons s'inspire d'une méthode utilisée dans le domaine du web sémantique pour établir des correspondances entre ontologies.

– Nous proposons une approche pour générer une transformation de modèles à partir d'exemples accompagnés d'un alignement. À partir de l'exemple précédent la transformation attendue contiendrait les règles suivantes : « une *Class* avec une *Property* se transforme en une *Entity* avec un *Attribute* » et « une *Property* appartenant à une *Class* se transforme en un *Attribute* appartenant à une *Entity* ». Nous utilisons pour arriver à nos fins une méthode de classification à base de treillis de concepts qui nous permet d'obtenir un ensemble de règles de transformations classées dans un treillis permettant de naviguer entre les différentes solutions proposées.

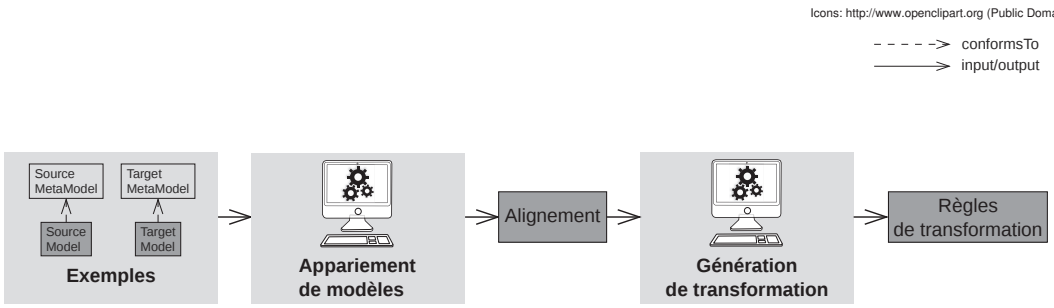


FIGURE 1.1 : Schématisation de l'approche de développement de transformation de modèles.

La combinaison de ces approches permet la définition d'une méthode de développement de transformation de modèles, schématisée par la figure 1.1, dans laquelle les experts métier ont un rôle essentiel en produisant les exemples dont dépend la définition de la transformation. Les règles obtenues permettent alors de faciliter la tâche du développeur. Ces approches ont fait l'objet des publications suivantes : [Dolques *et al.*, 2009a],[Dolques *et al.*, 2009b] et [Dolques *et al.*, 2010b].

1.2 Plan

La transformation de modèles est l'opération qui est au cœur de l'ingénierie dirigée par les modèles et qui a fait l'objet de nombreux travaux de recherche. Nous allons donc dans le chapitre 2, après une présentation de l'ingénierie dirigée par les modèles, établir un état de l'art sur la transformation de modèles et en particulier sur les différentes approches permettant de générer une transformation de modèles de manière automatique ou semi-automatique. Sont présentées notamment les méthodes de génération de transformations de modèles basées sur l'appariement des méta-modèles ainsi que les méthodes de génération de transformations de modèles basées sur l'analyse d'exemples de modèles. Ces dernières représentent le domaine dans lequel se situent les travaux de ce document.

Nous présentons ensuite notre contribution que nous avons séparée en deux chapitres : les chapitres 3 et 4.

Le chapitre 3 est dédié à une approche dont le but est d'obtenir un alignement de modèles. Après une revue des principales méthodes d'appariement de schémas et d'ontologies de la littérature, nous présentons notre adaptation de l'une d'entre elles pour le problème spécifique de l'alignement de modèles. Une implémentation de cette approche est alors évaluée par une étude de cas avec l'aide d'un ensemble de métriques que nous définissons. L'étude de cas est composée d'exemples de transformations de modèles provenant de sources diverses telles qu'un dépôt public de transformations ou des transformations de référence décrites dans la littérature. On évalue la précision, le rappel, l'effort et le f-score de chaque résultat par rapport à un alignement expert.

Le chapitre 4 présente une approche de génération de transformations de modèles à partir de modèles d'exemples. Cette approche se base sur une technique de classification appelée Analyse Relationnelle de Concepts que nous présentons suivie de son intégration dans notre approche. Nous décrivons ensuite l'implémentation de l'approche ainsi qu'une évaluation de notre méthode à partir d'une étude de cas. L'étude de cas est composée des exemples utilisés pour le chapitre précédent. On évalue la taille du treillis obtenu en résultat et la taille des différentes règles qui le composent et pour une transformation la pertinence des règles générées.

Pour finir, le chapitre 5 dresse le bilan des travaux exposés dans ce manuscrit et présente les perspectives envisagées pour la suite de ces travaux.

Nous présentons en complément dans l'annexe A la liste des différentes transformations utilisées dans les évaluations des chapitres 3 et 4 et dans l'annexe B un article en anglais concernant des travaux que nous avons effectués sur la correction de diagrammes de cas d'utilisation.

Chapitre 2

État de l'art

2.1 Ingénierie dirigée par les modèles

Le génie logiciel voit régulièrement apparaître de nouveaux paradigmes de programmation. On peut notamment citer la programmation fonctionnelle, le paradigme objet et plus récemment les composants et les aspects. Chacun ayant ses avantages et inconvénients selon les utilisations, chaque paradigme induit une vision différente de la programmation.

Par ailleurs, avec l'augmentation des besoins encouragée par l'amélioration des performances des ordinateurs, la taille des programmes n'a eu de cesse d'augmenter et le besoin d'outils pour appréhender de grandes quantités de code s'est fait ressentir. Ainsi des langages proposant des représentations des différentes étapes du développement avec un plus haut niveau d'abstraction sont apparus, tels que UML [Booch *et al.*, 1998] ou Merise [Tardieu *et al.*, 1983]. Le langage UML en particulier présente de nombreux diagrammes dont l'utilité apparaît à différentes étapes du développement, et se veut suffisamment complet pour parer à toutes les situations. Mais il s'est avéré trop limité, ne pouvant prendre en compte tous les paradigmes de programmation existant ou nouvellement créés et ne pouvant être totalement en adéquation dans toutes les situations possibles. L'ingénierie des modèles (IDM [Estublier *et al.*, 2005]) est née de ce besoin de pouvoir créer des langages de modélisation adaptés à chaque situation. Le principal objectif de l'IDM est de placer la modélisation au centre du développement en considérant les modèles comme des éléments de première classe du processus de développement.

D'après [Rothenberg, 1989], dont voici un extrait traduit : « la modélisation au sens le plus large est l'utilisation la plus rentable d'une chose à la place d'une autre pour un certain but. Cela nous permet d'utiliser quelque chose de plus simple, de plus sûr ou de moins coûteux qu'en réalité à la place de la réalité pour un certain but. Un modèle représente la réalité pour un but donné ; le modèle est une abstraction de la réalité dans le sens où il ne peut pas représenter tous les aspects de la réalité. Cela nous permet de traiter avec le

monde d'une manière simplifiée, en évitant la complexité, le danger et le caractère irréversible de la réalité. » Cette définition, bien qu'ayant été écrite en 1989, soit plus de 10 ans avant la création de l'IDM, en définit parfaitement les principes : les modèles ont été introduits pour faciliter la gestion d'un projet en offrant différentes vues avec différents buts, par exemple un diagramme de classes est un moyen rapide d'appréhender la structure d'un programme écrit dans un langage orienté objet. Cette définition nous permet de cerner l'utilité d'un modèle de manière générale, et les définitions qui suivent vont nous permettre de décrire sous quelle forme se présente un modèle dans le domaine plus restreint de l'IDM :

Définition 2.1 (Modèle) *« Un modèle est une description d' (une partie d') un système écrit dans un langage bien défini » (traduction d'un extrait de [Kleppe* et al.*, 2003]). Une carte routière par exemple peut être considérée comme un modèle puisqu'il s'agit d'une représentation d'une partie de la surface de la planète suivant une syntaxe précise, définie par la légende, focalisée sur les voies de déplacement. La définition des concepts du langage d'un modèle est définie par son méta-modèle.*

Définition 2.2 (Méta-modèle) *« Un méta-modèle est un modèle qui définit le langage pour exprimer un modèle » (traduction d'un extrait de [Kleppe* et al.*, 2003]). Les modèles écrits dans ce langage sont alors dits conformes au méta-modèle. Un méta-modèle étant aussi un modèle, il est donc conforme à un méta-modèle : le méta-méta-modèle.*

Définition 2.3 (Méta-méta-modèle) *Un méta-méta-modèle est le modèle d'un langage permettant d'écrire des langages de modélisation. Il est conforme à lui-même, devenant ainsi le sommet de la pile de méta-modélisation (voir figure 2.1). Ainsi, chaque plateforme de modélisation est basée sur un méta-méta-modèle. On peut citer par exemple MOF et sa version simplifiée EMOF définis par l'OMG [OMG, 2006], Ecore qui est le méta-méta-modèle d'Eclipse [Budinsky* et al.*, 2003] ou encore FM3 qui est le méta-méta-modèle du projet FAME [Kuhn et Verwaest, 2008].*

Un méta-modèle définit un langage, c'est-à-dire qu'il définit un ensemble de concepts et de relations entre ces concepts que l'on peut utiliser pour construire des modèles bien formés. Toutefois il ne définit pas toujours une syntaxe. La plupart des plateformes de modélisation définissent alors une syntaxe générique, dite syntaxe abstraite qui permet de stocker de manière générique un arbre syntaxique conforme à un méta-modèle. À cet effet l'OMG a défini XMI [OMG, 2007], un langage basé sur XML pour représenter les modèles. On peut voir alors pour un même modèle plusieurs représentations, la représentation en syntaxe abstraite et la ou les représentations en syntaxe concrète définies en accompagnement du méta-modèle.

L'ingénierie dirigée par les modèles consiste donc à utiliser les modèles comme éléments de base du développement. Elle est au cœur de plusieurs méthodologies de développement à base de modèles dont MDA [Soley et the OMG Staff Strategy Group, 2000] est

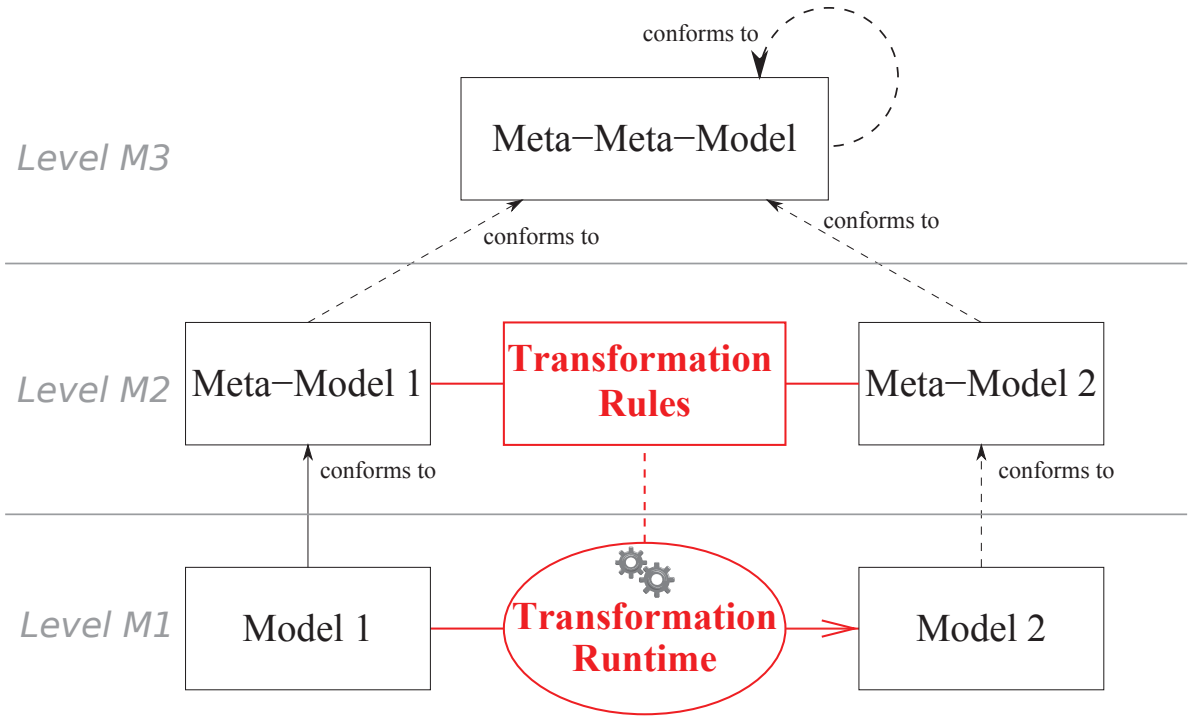


FIGURE 2.1 : Pile de méta-modélisation représentant les différents niveaux de modélisation.

sans doute la plus connue. Elle propose un découpage du processus de développement en trois étapes correspondant à trois types de modèle :

- *Computation Independent Model (CIM) :* il s'agit d'une vue du système s'intéressant essentiellement à l'environnement du système et à ses exigences. La structure du système n'est pas prise en compte.
- *Platform Independent Model (PIM) :* il s'agit d'une vue du système représentant le côté opérationnel du système mais sans prendre en compte les détails liés à la plateforme. Il s'agit d'une représentation de la partie du système commune à toutes les plateformes.
- *Platform Specific Model (PSM) :* il s'agit d'une combinaison du *Platform Independent Model* avec les détails spécifiques à la plateforme.

Dans une approche IDM, si le modèle est la donnée principale, l'opération principale est alors la transformation de modèle. La formalisation des modèles et surtout de leur méta-modèle permet d'envisager la création de programmes manipulant les modèles pour créer d'autres modèles, rendant ainsi les modèles productifs, c'est-à-dire que la création de modèles ne vient pas s'ajouter au processus de développement pour aider les développeurs à maîtriser leur projet mais en est une étape. Les informations contenues dans les domaines pourront être répercutées dans d'autres modèles par le biais d'une transforma-

tion. Un exemple courant est la génération de code à partir de diagrammes de classe : les classes sont créées une seule fois, dans un modèle, et le code de leur structure (entêtes, attributs et signatures de méthodes) est généré automatiquement. Dans les méthodes de développement traditionnelles, les modèles étaient dits contemplatifs et n'avaient qu'une fonction de documentation dans le projet.

2.2 Les Transformations de modèle

Dans [Kleppe *et al.*, 2003] on trouve pour les transformations de modèles les définitions suivantes :

- « Une *transformation* est la génération automatique d'un modèle cible à partir d'un modèle source, suivant une définition de la transformation » ;
- « Une *définition de transformation* est un ensemble de règles de transformation qui, réunies, décrivent comment un modèle dans un langage source peut être transformé dans un langage cible » ;
- « Une *règle de transformation* est une description de la manière dont une ou plusieurs structure du langage source peuvent être transformées en une ou plusieurs structures du langage cible ».

Mens et Gorp généralisent cette définition en considérant qu'une transformation peut avoir plusieurs modèles source et cible. Ils présentent dans leur état de l'art [Mens et Gorp, 2006] une classification des différentes transformations de modèles selon les critères suivants :

- *similarité des espaces techniques :* l'espace technique correspond aux langages de représentations supportés pour représenter les modèles en machine. On peut citer par exemple XMI [OMG, 2007] défini par l'OMG.
- *transformation endogène versus transformation exogène :* une transformation endogène est une transformation dont les modèles source et cible ont le même méta-modèle tandis qu'ils sont différents pour une transformation exogène.
- *transformation horizontale versus transformation verticale :* une transformation horizontale est une transformation où les modèles source et cible sont au même niveau d'abstraction, tandis qu'ils sont différents dans les transformations verticales. Le niveau d'abstraction dépend du niveau de détails dans le modèle, mais ne dépend pas forcément du méta-modèle, ainsi une transformation endogène, telle qu'un raffinement de modèles, peut-être considérée comme verticale.
- *transformation syntaxique versus transformation sémantique :* une transformation syntaxique se contente de modifier la syntaxe de représentation des modèles tandis qu'une transformation plus complexe est considérée comme sémantique. On considère notamment le passage d'un syntaxe concrète vers une syntaxe abstraite comme une transformation syntaxique.

Les transformations peuvent être écrites dans différents langages :

- des langages généralistes auxquels on ajoute un framework ou des librairies permettant la manipulation des modèles, par exemple EMF [Budinsky *et al.*, 2003] avec JAVA ou FAME [Kuhn et Verwaest, 2008] qui est disponible dans plusieurs langage ;
- des langages spécialisés tels que KERMETA [Muller *et al.*, 2005] qui est un langage orienté objet, ATL [Bézivin *et al.*, 2003] qui est un langage déclaratif ou VIATRA [Csertán *et al.*, 2002] qui est un langage basé sur les transformations de graphes.

L'écriture d'une transformation nécessite de manipuler les éléments définis dans le méta-modèle, ainsi la programmation d'une transformation nécessite une bonne connaissance du méta-modèle, ainsi que le lien entre les éléments définis dans le méta-modèle et la syntaxe concrète lorsque celle-ci existe. Par exemple dans le cas d'UML [OMG, 2010] une association reliant deux classes dans un diagramme de classe possède des rôles auxquels sont associés des cardinalités et peut être navigable dans les deux sens dans le cas où les deux extrémités de l'association ont connaissance de celle-ci ou dans un seul sens lorsqu'une seule des deux extrémités à connaissance de l'association. En syntaxe abstraite une association est un élément de type *Association* et les rôles sont des éléments de type *Property* reliés à l'association par une relation de type *MemberEnd*. Mais la difficulté se situe au niveau de la navigabilité qui n'est pas définie par un attribut dans l'association mais de la manière suivante : si la *Property* est reliée à l'*Association* par une relation de type *owningAssociation* alors l'extrémité représentée par la *Property* n'est pas navigable, si la *Property* est reliée à une *Class* par une relation de type *owningClass*, alors l'extrémité représentée par la *Property* est navigable (la figure 3.2 représente un extrait du méta-modèle UML dans lequel on peut voir la définition des *Association*). Il est à noter qu'une *Property* reliée à une *Class* par une relation de type *owningClass* peut aussi représenter un attribut si la *Property* n'est pas associée à une *Association*. Ceci montre bien qu'avoir une bonne connaissance d'un langage comme UML n'implique pas nécessairement de maîtriser son méta-modèle or ces deux compétences sont nécessaires pour écrire une transformation de modèles.

Dans [Czarnecki et Helsen, 2006] les auteurs proposent une classification des approches de transformation de modèles selon huit critères :

- *la spécification* : la spécification représente ici les mécanismes de contraintes comme des preconditions et postconditions qui peuvent être incluses dans le langage de transformation ;
- *les règles de transformation* : les règles de transformations peuvent être présentées de différentes manières : de manière classique sous forme de règles avec une prémisse et une conclusion, mais aussi des fonctions ou des templates peuvent aussi être considérés comme des formes de règles ;
- *le contrôle d'application des règles* : il s'agit ici de l'ordonnancement des règles et des parties du modèles à traiter ;
- *organisation des règles* : les structures d'organisation des règles telles que des modules ou des packages ;
- *relation source-cible* : cela concerne les relations entre les modèles source et cible :

dans certains cas le modèle cible est le modèle source sur lequel on a appliqué des modifications tandis que dans d'autres cas les deux modèles sont différents ;
– *incrémentalité* : cela décrit si l'approche de transformation permet de modifier le modèle cible lorsqu'on modifie le modèle source ;
– *directionalité* : si la transformation n'est appliquable que dans un seul sens elle est considérée comme unidirectionnelle, tandis qu'elle sera considérée comme multidirectionnelle dans le cas contraire ;
– *traçabilité* : cela concerne les mécanismes permettant d'enregistrer les différentes étapes de l'exécution d'une transformation.

2.3 Génération de transformations de modèles

Les transformations de modèle requièrent du temps et des compétences particulières : il est nécessaire pour le développeur de maîtriser à la fois un langage de transformation de modèles et les méta-modèles source et cible de la transformation. Dans certains cas la difficulté pour réaliser une transformation réside plus dans la taille de la transformation et des méta-modèles que dans sa complexité. C'est pour faciliter la création de ces transformations qu'ont été créés deux types d'approche : le premier type présenté à la section 2.3.1 se base sur les similarités entre méta-modèles et le second type présenté à la section 2.3.2 se base sur des modèles d'exemples pour générer des transformations plus complexes.

2.3.1 Génération de transformations de modèle basée sur l'alignement de Méta-modèles

La génération de transformations de modèles à partir d'alignement de méta-modèles est fortement inspirée des méthodes d'alignement de schémas, que nous aborderons plus en détails dans la section 3.1. Cela peut s'appliquer lorsque les méta-modèles source et cible sont proches à la fois sémantiquement et structurellement pour permettre la déduction de la transformation automatiquement à partir des seules informations contenues dans les méta-modèles. Ce type d'approche se trouve donc approprié dans le cadre de migrations de modèles, c'est-à-dire lorsque les méta-modèles sources et cibles sont deux langages proches dédiés à un même but.

Une première approche est présentée dans [Lopes *et al.*, 2005, 2006] où les auteurs présentent une approche de génération automatique de transformation dans laquelle nous pouvons dégager deux parties :
– Dans une première partie, les auteurs définissent un algorithme d'alignement automatique de méta-modèles appelé SAMT4MDE. Il s'agit d'établir un ensemble de correspondances entre les éléments du méta-modèle source et du méta-modèle cible. Pour cela, l'approche propose de comparer les éléments des méta-modèles de même type et d'appliquer sur chaque couple possible une fonction mesurant la simila-

rité entre les éléments. La fonction est définie différemment selon que les éléments à comparer sont des classes, des types primitifs ou des énumérations, mais pour chaque couple elle donne un résultat permettant d'évaluer son degré de similarité. L'ensemble des couples ayant un degré de similarité suffisamment élevé par rapport à une borne définie formeront alors un alignement.

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

La première partie de cette approche a été par la suite améliorée dans [Lopes *et al.*, 2009] : les modifications portent principalement sur la fonction d'évaluation du degré de similarité, qui évalue non seulement des similarités de valeur au niveau des noms ou des attributs, mais elle évalue aussi la similarité au niveau de la structure des éléments en prenant en compte les éléments qui leur sont liés.

Dans [Del Fabro et Valduriez, 2007] les auteurs présentent une approche similaire à la précédente. Il s'agit ici de générer par transformations de modèles un modèle dit de tissage (*weaving model* en anglais) similaire aux modèles d'alignement vus précédemment. Ce modèle s'obtient grâce à un processus d'alignement qui comme dans les précédentes approches définit d'abord une fonction de similarité appliquée à tous les couples d'éléments des méta-modèles source et cible. Cette fonction de similarité permet de définir un degré de similarité pour chaque couple, il est ici calculé à partir de comparaison de chaînes de caractères, de comparaison de noms à l'aide d'un dictionnaire de synonymes et d'éléments de structure. Le *similarity flooding* [Melnik *et al.*, 2002] est utilisé pour propager la similarité. Le principe du *similarity flooding* est basé sur l'hypothèse que les voisins respectifs d'éléments reconnus comme similaires ont de grandes chances d'être eux aussi similaires. Ainsi les voisins d'éléments ayant une similarité importante voient leur similarité augmenter ce qui permet, par propagation, de compléter l'alignement. L'étape finale consiste à prendre les couples dont la similarité est la plus élevée pour obtenir alors le modèle de tissage permettant la génération d'une transformation

L'approche présentée dans [Falleri *et al.*, 2008] se concentre sur la génération d'alignement et reprend la plupart des éléments de l'approche de Del Fabro et Valduriez mais de manière plus détaillée. Il y est notamment présenté les différentes possibilités pour traduire les méta-modèles dans un format de données exploitable pour l'algorithme *similarity flooding* et l'impact sur les résultats.

La dernière approche, présentée dans [Kappel *et al.*, 2006] prend le problème d'un autre point de vue : les auteurs considèrent que le problème de l'alignement de méta-modèles est difficile car les méta-modèles ne se contentent pas de décrire un ensemble de concepts mais sont aussi guidés par la manière d'implémenter ces concepts sous forme d'un langage. Ils considèrent par ailleurs que l'alignement d'ontologies est plus facile car les ontologies sont plus appropriées pour représenter l'ensemble des concepts d'un langage sans

se soucier de son implémentation. Les auteurs proposent alors un processus permettant la transposition d'un méta-modèle en une ontologie. Cette transposition permet d'utiliser les techniques d'alignement d'ontologies, ici COMA++. Une fois l'alignement obtenu, il est alors possible de remonter vers les méta-modèles pour établir un lien entre les éléments du méta-modèle source et ceux du méta-modèle cible.

2.3.2 Génération de transformations de modèle basée sur des modèles d'exemple

Bien que les travaux sur la programmation par l'exemple ou les requêtes par l'exemple soient assez anciens, nous donnons pour origine de la transformation de modèles par l'exemple les travaux de Varró, tout d'abord dans [Varró, 2006] où l'auteur propose une approche générale en s'inspirant des systèmes de génération automatique de règles XSLT. L'auteur part du constat que les langages de transformation ont tendance à monter graduellement leur niveau d'abstraction pour se conformer à la spécification QVT (Query, Views and Transformations) de l'OMG [OMG, 2008] qui encourage les langages déclaratifs basés sur les règles de transformation. Il propose alors une approche permettant la spécification semi-automatique d'une transformation, c'est-à-dire une ébauche de la transformation. Cette approche est itérative et chaque itération se découpe en quatre étapes :

- *étape 1* : création manuelle d'un alignement de modèles prototypes. Il s'agit de créer un ensemble de modèles dans le langage source de la transformation, ainsi que les modèles transformés correspondant dans le langage cible, et couvrant les différentes situations possibles de la transformation. Puis on relie les éléments sources et cibles par des liens d'appariement, typés en fonction de la règle de transformation dont ils sont l'illustration. Par exemple dans une transformation allant de UML vers un modèle Entité-Association, un type de lien d'appariement *ClassToEntity* illustrera la règle transformant une *Class* vers une *Entity*.
- *étape 2* : dérivation automatique des règles de transformation en se basant sur l'alignement de l'étape 1. Ces règles devraient permettre d'obtenir à partir des modèles sources donnés en exemple les modèles transformés correspondant, eux aussi donnés en exemple.
- *étape 3* : raffinement manuel des règles. Les règles obtenues portent sur la structure, et ne prennent pas en compte d'éventuelles modifications de valeurs, qui sont à faire par le développeur.
- *étape 4* : validation des règles obtenues sur de nouveaux cas de tests qui deviennent, lorsque les règles ne sont pas satisfaisantes, de nouveaux prototypes pour une nouvelle itération du processus.

L' intérêt d'une telle approche est l'utilisation des langages source et cible comme « langage de transformation », c'est-à-dire comme le langage de définition de la transformation,

avec l'hypothèse à l'origine de la plupart des approches « par l'exemple » qui est que la manipulation d'exemples est plus intuitive.

L'auteur définit par ailleurs les éléments à prendre en considération dans la création de transformations. C'est ainsi qu'il définit le *1-context* d'un élément du modèle source comme la description du voisinage immédiat de cet élément. Cette description consiste pour chaque association dont l'élément peut être une extrémité, à préciser l'existence ou non de cette association ainsi que le type de l'extrémité opposée. Bien que le passage au *n-context* ne pose pas de difficulté particulière, l'auteur considère que le voisinage immédiat est suffisant dans la grande majorité des cas pour découvrir une règle de transformation.

L'ensemble des *1-context* pour un type de lien d'appariement forme un *joint 1-context*. L'analyse de tous les *1-context* permet de définir pour un type d'élément donné les *voisins obligatoires* qui sont nécessaires pour qu'une règle s'applique (les voisins qui apparaissent dans tous les *1-context*), les *voisins autorisés* dont la présence n'empêche pas la règle de s'appliquer (les voisins non-obligatoires qui apparaissent dans au moins *1-context*).

Un contexte pour l'élément cible d'un lien d'appariement est aussi défini. Il est caractérisé par l'ensemble des voisins contenus dans l'élément par une relation de composition et qui ne sont pas cible d'un lien d'appariement.

Approche utilisant la logique inductive

La spécification [Varró, 2006] a été suivie par [Balogh et Varró, 2009], une proposition d'implémentation de l'approche précédente utilisant la programmation par logique inductive (ILP [Muggleton et De Raedt, 1994]). La logique inductive est une méthode de *machine learning* consistant à créer des règles logiques à partir de faits et règles existantes.

De manière informelle l'ILP est basée sur un système d'inférence qui s'apparente à de l'apprentissage par généralisation sur un ensemble de faits positifs et négatifs. À partir d'un ensemble B de règles logiques que l'on nommera les connaissances de base, d'un ensemble E^+ de faits positifs, c'est-à-dire des faits qui sont considérés comme vrais mais qui ne peuvent pas être déduits à partir de B et d'un ensemble E^- de faits négatifs, c'est-à-dire des faits qui sont considérés comme faux mais qui ne peuvent pas être déduits à partir de B, le mécanisme d'induction permet de trouver une hypothèse H telle que $B \wedge H$ permet de déduire E^+ sans en déduire E^-.

Dans cette approche, les connaissances de base sont formées par un ensemble de clauses décrivant les modèles sources et cibles alors que les faits positifs sont formés par les liens d'appariement et les faits négatifs par l'ensemble des couples d'éléments source-cible qui ne sont pas appariés entre eux.

À partir de ces données, un moteur d'inférence en logique inductive (ici [ALEPH] est utilisé) peut induire une hypothèse pour chaque type d'appariement sur les propriétés communes des éléments sources et cibles.

Approche basée sur la syntaxe concrète

Une autre approche de génération de transformations dirigée par l'exemple a été spécifiée par [Wimmer *et al.*, 2007]. Il s'agit d'une adaptation de la spécification de [Varró, 2006], en remplaçant l'étude du voisinage des éléments par l'étude du lien entre syntaxe concrète et abstraite. Ces travaux partent de l'hypothèse que la représentation en syntaxe concrète contient des informations pour créer les différentes conditions des règles d'une transformation. Notamment le phénomène de « dissimulation de concept » fait référence au cas où tous les concepts de la syntaxe concrète ne sont pas représentés de manière explicite dans le méta-modèle. Un exemple de ce phénomène est le concept d'attribut dans les diagrammes de classe UML qui n'apparaît pas explicitement dans le méta-modèle UML. En effet il n'existe pas de classe pour représenter les attributs, il sont représentés sous forme de *Property* lorsqu'elles ne sont pas une extrémité d'*Association* (la figure 3.2 représente un extrait du méta-modèle UML dans lequel on peut voir la définition des *Property*). Une *Property* peut aussi représenter le concept de rôle lorsqu'elle est extrémité d'une *Association*. On peut alors considérer que les concepts d'attributs et de rôles sont « cachés » dans le méta-modèle.

Ainsi, si on connaît pour un élément de modèle le concept qu'il représente en syntaxe concrète, il est possible de lui associer un ensemble de contraintes que l'élément doit satisfaire. Par exemple, une *Property* qui est représentée par un attribut en syntaxe concrète respectera la contrainte « n'est pas extrémité d'une *Association* ». L'approche de génération de transformation est similaire aux travaux précedents : la première étape consiste à créer des modèles d'exemple source et cible, et à créer manuellement des liens d'appariement entre les éléments des modèles. La différence avec la description de [Varró, 2006] est que cet alignement s'effectue sur les modèles en syntaxe concrète. Cela nécessite alors qu'aient été créés des éditeurs en syntaxe concrète pour les méta-modèles source et cible capables de gérer la création de liens d'appariement. Les liens d'appariement relient alors des triplets $< as_E, cs_E, const(as_E)? >$ où cs_E représente l'élément en syntaxe concrète, as_E l'élément en syntaxe abstraite et $const(as_E)?$ les contraintes associées à l'élément en syntaxe abstraite. Là encore, il est nécessaire que la transformation de la syntaxe concrète vers la syntaxe abstraite ait été implémentée de manière à fournir les différentes contraintes associées à un élément.

À partir de ces couples de triplets, le processus de génération de transformation cherche alors à définir quel type d'élément est créé pour un type d'élément donné, créant alors un ensemble de règles qui transforment un élément d'un type source donné en un élément d'un type cible particulier. Lorsqu'il arrive qu'un même type d'élément soit transformé de plusieurs manières différentes le processus de génération utilise les contraintes associées aux concepts concrets des éléments. Par exemple, dans la transformation d'un diagramme UML vers un diagramme Entité-Association, une *Property* peut être transformée en *Attribute* si elle est représentée comme un attribut en syntaxe concrète et en *Role* si elle est représentée comme un rôle en syntaxe concrète. Cette transformation est décrite

plus en détail dans la section 3.2.1. Le processus créera alors deux règles différentes pour les *Property* qui ne s'appliqueront que si l'élément sur lequel on les applique respecte les contraintes spécifiques aux concepts de la syntaxe concrète.

Le processus se poursuit en cherchant pour chaque règle des correspondances entre les valeurs des attributs. Il s'agit dans certains cas d'utiliser des heuristiques pour trouver des correspondances dans les modèles d'exemples. Il arrive aussi que certaines valeurs d'attributs soient contraintes par leur représentation en syntaxe concrète, auquel cas le processus utilise encore les informations liées au passage entre la syntaxe concrète et la syntaxe abstraite.

Enfin, la dernière étape du processus avant la génération du code est la création de liens entre les objets cibles des règles créées. La création de liens se base sur le méta-modèle, les contraintes liées au passage entre syntaxe concrète et abstraite et les liens d'appariement. Lorsque la situation apparaît trop ambiguë, le processus fait alors appel à l'utilisateur pour orienter le choix.

À la fin de toutes ces étapes l'ensemble des résultats obtenu est alors transformé de manière directe vers le langage ATL.

Application de transformation par l'exemple

La dernière approche que nous ayons identifiée est l'approche présentée dans [Kessentini *et al.*, 2008, 2009] et [Kessentini *et al.*, 2010b]. Le principe diffère des deux précédentes approches dans le sens où il n'est plus question de générer le code d'une transformation de modèles à partir d'exemples mais de faire apprendre à un système à transformer des modèles. Il s'agit pour l'utilisateur de fournir des exemples sources et cibles à un programme, qui apparaît alors comme une boite noire, durant une phase d'apprentissage. Une fois cette phase d'apprentissage passée cette même boite noire permet de transformer un modèle source en un modèle cible. Ainsi aucun code n'est fourni à l'utilisateur et il ne peut influer sur la transformation qu'en fournissant des exemples. Il s'agit aussi d'un système évolutif, qui va pouvoir prendre en compte les transformations effectuées et corrigées par l'utilisateur pour s'améliorer.

Durant la phase d'apprentissage, des modèles sources et cibles sont donnés au système avec des liens d'appariement qui, ici, ne relient pas un élément à un autre, mais un groupe d'éléments, appelé *bloc*, dans le modèle source à un *bloc* dans le modèle cible. Le système possède alors une liste d'*exemples de transformations*, c'est-à-dire d'exemples de blocs avec le *bloc* transformé correspondant.

Lorsqu'on fournit un modèle à transformer au système, chaque élément du modèle entraîne la création d'une dimension. On se retrouve alors avec un espace à n dimensions, où n correspond au nombre d'éléments dans le modèle. Chaque dimension possède un intervalle de valeurs discrètes $[0..N]$ où N est le plus grand index des exemples de transformation extraits durant la phase d'apprentissage. Il s'agit alors de trouver la position dans cet espace permettant d'associer à chaque dimension (c'est-à-dire à chaque élément du mo-

dèle) un exemple de transformation approprié, c'est-à-dire trouver un exemple de transformation tel que le bloc source contienne un élément suffisamment proche de l'élément de modèle correspondant à la dimension analysée.

Cette vision du problème permet alors de réduire le problème de la transformation d'un modèle en un problème d'optimisation. La méthode utilisée ici est l'optimisation par essaim de particules (PSO [Kennedy et Eberhart, 1995]). Il s'agit de placer de manière aléatoire un ensemble de particules dans l'espace des solutions et, à l'aide d'une fonction mesurant la qualité d'une solution, déplacer les particules de manière à converger vers une solution satisfaisante. Cela permet de trouver une solution sans avoir à parcourir tout l'espace des solutions dont la taille ne permettrait pas d'obtenir un résultat en un temps raisonnable.

Cette solution peut être améliorée dans certains cas grâce à la méta-heuristique du recuit simulé [Kirkpatrick *et al.*, 1983], qui consiste à prendre une solution et à en modifier certaines dimensions de manière aléatoire, dans le but d'améliorer cette solution en plusieurs itérations. Cette méthode fait appel à une variable que l'on nomme la température et qui va décroître à chaque itération. La température permet de définir la probabilité d'accepter une détérioration de la solution, plus elle est élevée et plus un changement diminuant la qualité de la solution sera accepté. Cela permet d'éviter les optimums locaux, ainsi on acceptera beaucoup de modifications de la solution au début du processus, mais au fur et à mesure que la température descend, la solution devient plus « rigide » et on n'accepte alors plus que des améliorations de la solution.

Cette approche a été appliquée par la suite à la transformation de diagrammes de séquence vers les réseaux de Petri colorés. Cette transformation a la particularité de s'appliquer à des diagrammes dynamiques dans lesquels la chronologie a une grande importance. L'article [Kessentini *et al.*, 2010a] présente une adaptation de l'approche dans laquelle la fonction objectif est modifiée pour prendre en compte la cohérence temporelle.

2.4 Conclusion

Nous récapitulons dans la table 2.1 les différentes approches présentées dans cet état de l'art pour la génération de transformations de modèle. Les méthodes de création de transformation de modèles représentent un enjeu important pour l'ingénierie des modèles. Parmi les méthodes de génération automatique, l'appariement de méta-modèles est une approche intéressante dans les cas les plus simples, mais les cas les plus complexes requièrent un processus différent tel que les approches par l'exemple.

Les différentes approches de génération de transformations de modèles basées sur des exemples de modèles sont présentées dans le tableau récapitulatif 2.2. Un problème commun à ces approches reste la quantité de données nécessaire à la réalisation de la transformation : la création de l'alignement de modèles est la tâche la plus fastidieuse du processus dans chaque cas, et peut être source d'erreur lorsque la transformation est complexe.

TABLE 2.1 : Récapitulatif des méthodes de génération de transformation décrites.

Références des méthodes	**Génération de transformation**	
	à partir des méta-modèles	**à partir de modèles d'exemples**
[Lopes *et al.*, 2005]	×	
[Del Fabro et Valduriez, 2007]	×	
[Falleri *et al.*, 2008]	×	
[Balogh et Varró, 2009]		×
[Kessentini *et al.*, 2008]		×
[Wimmer *et al.*, 2007]		×
Notre approche		×

Par ailleurs, l'approche de [Balogh et Varró, 2009] nécessite une première spécification de la transformation pour donner un type aux liens d'appariement. L'approche de [Wimmer *et al.*, 2007] nécessite quant à elle le développement d'outils au niveau des éditeurs de modèles rendant difficile son application sur de nouveaux langages. La principale qualité d'une approche par l'exemple est son côté intuitif, ne nécessitant pas de réflexion préalable. Il est donc nécessaire d'essayer de réduire à la simple création d'exemples les tâches nécessaires à la génération de la transformation.

Une approche de génération de modèles transformés telle que celle de [Kessentini *et al.*, 2008] peut être envisagée si le contexte permet une vérification manuelle des résultats, c'est à dire si le nombre d'application de la transformation est limité et si une spécification complète de la transformation n'est pas possible. Elle n'est cependant pas appropriée pour la génération de transformations critiques, ou devant être utilisées massivement car dans de tels cas la confiance dans le résultat est nécessaire.

Nous présentons dans les chapitres suivants une approche de génération de transformation de modèles à partir d'exemples générant la définition d'une transformation. Cette approche ne requiert pas de développements particuliers pour les méta-modèles source et cible. Les alignements passés en paramètre ne sont pas typés et peuvent être générés en partie automatiquement grâce à une méthode d'alignement. Le résultat fourni est un ensemble de règles de transformations parmi lesquelles le développeur peut choisir les plus appropriées par rapport à la transformation voulue. Ces règles sont organisées sous forme de treillis ce qui facilite la navigation entre les différentes règles proposées.

TABLE 2.2 : Récapitulatif des méthodes de génération de transformation à partir de modèles d'exemples.

Références des méthodes	**Alignement d'entrée**	**Développement spécifique à la transformation**	**Données de sortie**
[Balogh et Varró, 2009]	un ensemble de couples d'éléments classés dans différentes catégories	aucun	code de la transformation (ciblé pour le langage VIATRA)
[Kessentini *et al.*, 2008]	un ensemble de couples de blocs	aucun	après apprentissage le système fournit le résultat de la transformation
[Wimmer *et al.*, 2007]	un ensemble de couples d'éléments	les contraintes du passage de la syntaxe concrète vers la syntaxe abstraite	code de la transformation (ciblé pour le langage ATL)
Notre approche	un ensemble de couples d'éléments	aucun	treillis des règles. Les règles sélectionnées sont ensuite traduites en code de la transformation (ciblé pour le langage ATL)

Chapitre 3

Alignement de modèles : Génération semi-automatique des liens dans les exemples par des techniques d'appariement

Les approches d'apprentissage par l'exemple ont besoin de « voir » l'action qu'on souhaite leur apprendre et de « suivre » les transformations de données pour pouvoir les reproduire. Les méthodes d'intelligence artificielle se basent en général sur un grand ensemble de données pour extraire des règles de transformation, mais dans notre cas l'ensemble des exemples est limité car produit par le développeur. Il faut donc, comme dans le cas de la programmation par l'exemple [Lieberman, 2001], fournir au mécanisme d'apprentissage une traçabilité de l'exemple de transformation. Cette traçabilité peut être obtenue au moment de la création des exemples en analysant les actions du développeur, mais cela n'est possible qu'à la condition que l'environnement de création des exemples permette cette analyse. Or dans le cadre de l'ingénierie dirigée par les modèles, un méta-méta-modèle unique est défini (par exemple Ecore pour la plateforme Eclipse) et associé à un format de stockage en mémoire (par exemple XMI défini par l'OMG pour MOF mais aussi utilisé pour Ecore), mais il n'en est rien pour les éditeurs. Ainsi nous nous posons comme contrainte de ne demander aucun développement spécifique aux méta-modèles source et cible dans les éditeurs.

La création d'une trace de l'exécution de la transformation que l'on souhaite effectuer revient à apparier les éléments entre les modèles d'exemple source et cible. Dans les approches de génération de transformation de modèles basées sur les exemples présentées dans la section 2.3.2 et l'approche présentée dans le chapitre 4, la tâche la plus fastidieuse est la création manuelle de cet appariement. La contrainte que nous avons de ne pas imposer au développeur la création d'outils spécifiques aux méta-modèles intervenant dans la transformation ne nous permet pas de gérer la création des liens au moment de la créa-

tion des exemples. Il est donc nécessaire pour le développeur de créer tout d'abord ses exemples et d'ajouter ensuite tous les liens d'appariement, ce qui peut introduire de nombreuses erreurs dans le cas d'exemples un peu compliqués. Cependant, si le développeur s'impose quelques règles lors de la création de ses exemples, nous pensons qu'il est possible de l'assister en générant de manière automatique les appariements ne posant aucune ambiguïté et en lui laissant le soin de régler les cas nécessitant une intervention humaine. Dans ce chapitre nous proposons une méthode qui s'inspire des méthodes d'alignement de schémas ou d'ontologies utilisées dans les domaines de représentation des connaissances. Aussi nous présenterons dans la section 3.1 un bref aperçu des méthodes d'appariement, avant de décrire notre approche dans la section 3.2, accompagnée dans la section 3.3 par une étude de cas permettant d'évaluer sur des cas concrets les forces et les faiblesses de notre approche. Nous conclurons dans la section 3.4.

3.1 Aperçu sur les méthodes d'appariement

L'appariement d'éléments de deux entités basé sur leur sémantique se retrouve dans la littérature principalement dans le domaine des bases de données et de la représentation de connaissances sous le nom d'alignement ou d'appariement. Nous avons déjà vu dans la section 2.3.1 que les problématiques d'appariement se retrouvent dans le génie logiciel par exemple pour la génération de transformation de modèles par alignement de métamodèles.

Nous allons dans cette section aborder le problème de l'alignement de manière plus générale et présenter les principales approches existant dans la littérature. L'article [Shvaiko et Euzenat, 2005] est un état de l'art sur les différentes méthodes d'alignement qui regroupe à la fois les méthodes d'appariement d'ontologies et les méthodes d'appariement de schémas. Ontologies et schémas sont deux termes utilisés pour faire référence à des modèles de représentation de connaissance. Le terme schéma fait référence aux schémas de base de données dans le monde de l'intégration d'information ou plus généralement aux XML-schemas et catalogues du web. La principale différence entre schémas et ontologies vient du fait que les schémas n'ont pas de sémantique explicite de leurs données, tandis que les ontologies sont des systèmes logiques obéissant à certaines sémantiques formelles.

Définition 3.1 (Lien d'appariement) *D'après [Shvaiko et Euzenat, 2005], un lien d'appariement (*mapping element *en anglais) se définit par un quintuplet* (id, e, e', n, R) *où* id *est un identifiant pour le lien ; e et* e' *sont les entités mises en relation ; n est une mesure de confiance du lien et R est la relation qui relie e et* e' *(équivalence, inclusion, subsomption...)*

Définition 3.2 (Alignement) *Un alignement (*alignment *en anglais) est un ensemble de liens d'appariement.*

Définition 3.3 (Appariement) *L'appariement (*matching *en anglais) est le processus qui permet de générer un alignement.*

On peut remarquer dans la plupart des méthodes d'appariement une base commune : en dehors de l'approche QOM (voir plus loin) toutes les approches présentées effectuent un produit cartésien entre l'ensemble des éléments du schéma source et l'ensemble des éléments du schéma cible, et pour chaque couple est appliquée une fonction de calcul de similarité. La manière de calculer la similarité est propre à chaque méthode, mais on retrouve assez souvent un recours à la comparaison des noms des éléments, soit par simple comparaison de chaînes de caractères, soit en faisant appel à des outils de traitement de la langue naturelle plus évolués. Enfin, la dernière étape à effectuer est généralement un filtrage des solutions pour garder les liens d'appariement dont la similarité est la plus élevée. Là encore chaque méthode propose son propre dispositif de filtrage.

L'approche présentée par [Madhavan *et al.*, 2001] et nommée CUPID est une méthode d'appariement de schéma. Elle nécessite de réduire la donnée d'entrée à une structure d'arbre. Le coefficient de similarité utilisé ici est une combinaison d'une similarité linguistique et une similarité structurelle. La similarité linguistique correspond à une similarité calculée au niveau des noms d'éléments. La similarité structurelle est calculée selon les règles suivantes :

- deux éléments feuilles sont considérés similaires s'ils ont une similarité linguistique élevée et si leur voisins respectifs sont similaires ;
- deux éléments non feuilles sont similaires s'ils ont une similarité linguistique élevée et si les sous-arbres dont ils sont racines sont similaires, c'est-à-dire si l'on trouve une grande quantité de liens de similarité entre éléments des deux sous-arbres ;
- deux éléments non feuilles sont similaires si l'ensemble des feuilles du sous-arbre dont ils sont racines sont similaires, même si leurs enfants directs ne le sont pas.

Le filtrage est ensuite effectué en définissant une borne supérieure pour les coefficients de similarité.

Les approches [Melnik *et al.*, 2002; Noy et Musen, 2001; Euzenat *et al.*, 2004] ont la particularité de découper le calcul de similarité en deux parties : la première partie, commune aux trois approches, est la création d'un premier alignement grâce à une fonction de similarité basée sur l'analyse de chaînes de caractères. Cet alignement est ensuite complété par une approche spécifique :

- dans le cas d'AnchorPROMPT [Noy et Musen, 2001], le premier alignement permet d'établir des ancres qui correspondent aux extrémités des liens d'appariement existants. L'approche se base sur l'hypothèse que les éléments qui sont situés dans le schéma source entre deux ancres ont de fortes chances de se trouver entre les deux ancres correspondantes dans le schéma cible. Ainsi le processus fait augmenter le coefficient de similarité d'un couple d'éléments se trouvant entre les extrémités de deux liens d'appariement. Les couples possédant le plus grand coefficient de similarité deviennent alors des liens d'appariement.

– dans le cas du similarity flooding [Melnik *et al.*, 2002], les schémas sont tout d'abord transformés sous forme de graphes étiquetés dans lesquels chaque élément et attribut du schéma devient un nœud étiqueté par sa valeur et chaque relation entre éléments ou relation d'appartenance entre élément et attribut devient un arc étiqueté par son type. Le similarity flooding se base sur l'hypothèse que l'augmentation du coefficient de similarité d'un élément entraîne l'augmentation du coefficient de similarité de ses voisins. Ainsi le premier alignement applique des coefficients de similarité sur certains couples de nœuds des graphes formés et plusieurs itérations permettent de propager la similarité aux nœuds voisins. Une fois arrivé à un point fixe, lorsque la valeur de la propagation devient négligeable, il s'agit alors d'établir quels appariements conserver. Contrairement à la plupart des approches qui définissent un seuil général, les auteurs proposent ici de définir un seuil local pour chaque nœud. Cette approche de filtrage se trouve plus détaillée à la section 3.2.3 car nous l'utiliserons dans notre approche.
– l'approche OLA [Euzenat *et al.*, 2004] est très proche du similarity flooding. OLA s'applique spécifiquement sur les ontologies et cela a pour conséquence de rajouter des informations dans les graphes étiquetés générés : chaque nœud et arc appartient à une catégorie correspondant aux différents types présents dans les ontologies. Les coefficients de similarité sont calculés à partir d'une formule dépendant de la similarité au niveau des noms des éléments correspondant aux liens d'appariement déjà existants, mais dépendant aussi des coefficients de similarité des éléments voisins. L'ensemble des coefficients de similarité forme un système d'équations, dont le processus tente d'obtenir une solution approchée en propageant les modifications de coefficients de similarité sur plusieurs itérations.

L'approche COMA [Do et Rahm, 2002] et son évolution COMA++ [Aumueller *et al.*, 2005] présentent une plateforme d'appariement permettant d'utiliser différents processus d'appariement. Cette plateforme permet notamment d'appliquer simultanément plusieurs approches d'appariement et offre plusieurs méthodes pour combiner les résultats obtenus.

L'approche GLUE [Doan *et al.*, 2003] propose un système d'appariement d'ontologies basé sur des méthodes d'apprentissage. Le principe est d'appliquer une méthode d'apprentissage pour classer les éléments de l'ontologie source dans différents concepts. Il est alors possible d'établir pour un couple d'éléments des ontologies source et cible la probabilité pour qu'ils appartiennent tous les deux au même concept. GLUE peut utiliser plusieurs méthodes d'apprentissage différentes pour obtenir de meilleurs résultats. Il s'agit ensuite de trouver l'alignement respectant à la fois un ensemble de contraintes spécifiques au domaine, un ensemble d'heuristiques et les valeurs de similarité obtenues à partir des probabilités.

L'approche QOM [Ehrig et Staab, 2004] est une approche d'appariement d'ontologies dont la particularité est sa faible complexité, de l'ordre de $O(n \times log(n))$ qui permet de l'utiliser sur de grandes ontologies. Ceci est possible en diminuant le nombre de couples

dont on calcule la similarité. Dans les précédentes approches, tous les couples d'éléments source-cible sont évalués. Ici les auteurs proposent un ensemble de stratégies pour choisir les différents couples dont on calculera le coefficient de similarité, comme par exemple un choix aléatoire jusqu'à obtenir un pourcentage de couples donné, ou lorsqu'un appariement satisfaisant est trouvé, choisir des couples dans le voisinage.

L'approche S-Match [Giunchiglia *et al.*, 2007] est une approche d'appariement pouvant s'appliquer indifféremment sur toute structure que l'on peut ramener à un arbre. Il s'agit d'extraire pour tous les éléments des graphes source ou cible leur sémantique sous forme de formules logiques. Ce sont ces formules qui sont alors comparées pour établir la similarité entre deux nœuds. On trouve ici à la place du coefficient de similarité des autres approches une relation sémantique. Les relations sémantiques possibles sont l'équivalence, la généralisation, la spécialisation, le dépariement (c'est-à-dire l'absence de lien sémantique) et le recouvrement.

Les modèles d'exemples que nous souhaitons aligner contiennent des valeurs similaires. En effet lors d'une transformation certaines informations sont conservées, comme des noms ou des identifiants. Il apparaît donc pertinent d'utiliser ces valeurs pour la création d'alignements. Nous ne pouvons par contre pas nous baser sur les systèmes s'appuyant sur la comparaison de la sémantique des éléments. Il est en effet peu probable qu'une transformation transforme par exemple un mot dans une chaîne de caractères en un de ses synonymes. Nous devons par ailleurs utiliser une méthode s'appliquant sur des graphes pouvant contenir des cycles. Tous ces critères permettent de réduire les approches possibles à AnchorPROMPT, OLA et similarity flooding. Parmi ces approches, OLA et similarity flooding nécessitent de prendre en compte la similarité des types de relations entre éléments pour propager la similarité, or ces relations sont définies au niveau du méta-modèle. Nous n'avons dans notre approche aucune hypothèse concernant la similarité des relations entre les méta-modèles sources et cibles. Il apparaît donc que, parmi les approches proposées, AnchorPROMPT est l'approche la plus appropriée à la résolution du problème de l'alignement de modèles.

3.2 Appariement de modèles

Contrairement aux méthodes d'alignement d'ontologies, de schémas ou de méta-modèles présentées dans la section 3.1, l'appariement de modèles ne peut pas se baser sur un langage connu par avance pour définir les points importants sur lesquels baser la comparaison. Ainsi pour définir une méthode de comparaison générique entre deux modèles nous ne connaissons pas par avance les classes et attributs au niveau méta-modèle des éléments sur lesquels la comparaison va porter. Par contre, nous savons que l'un des modèles est le résultat d'une transformation automatique de l'autre et donc que toutes les données du modèle cible dépendent du modèle source. Dans le cas d'une transformation essentiellement portée sur la structure, les valeurs des attributs tels que les noms restent

les mêmes ou sont très proches. Nous basons donc notre méthode d'appariement sur la conservation de valeurs au cours de la transformation.

Ainsi nous présentons dans la section 3.2.2 un premier appariement qui, à défaut d'être complet, s'avère très fiable. Pour compléter l'appariement, nous proposons d'utiliser une technique de propagation d'alignement qui se base sur un alignement fiable pour trouver d'autres liens d'appariement en prenant comme hypothèse qu'une partie de la structure est conservée. Nous avons opté pour une adaptation de la méthode AnchorPROMPT [Noy et Musen, 2001] que nous présentons dans la section 3.2.3.

3.2.1 Exemple illustratif

Pour illustrer la présentation de la méthode, nous allons utiliser un exemple de transformation de UML vers Entité-Association [Chen, 1976]. Cette transformation a été choisie car elle est souvent considérée comme représentative dans la littérature, et a notamment été utilisée dans l'édition 2005 de l'atelier Model Transformation in Practice [mtip]. Notre approche nécessite en entrée deux modèles : l'un représentant un modèle que l'on souhaiterait donner en entrée à la transformation que l'on est en train de spécifier et l'autre représentant le résultat souhaité une fois notre transformation appliquée sur le premier.

Notre modèle d'entrée est représenté en syntaxe UML à la figure 3.1. Il s'agit d'une modélisation simpliste d'une classification d'œuvres littéraires où un texte identifié par son titre peut avoir un certain style. Chaque texte date d'une année donnée et est écrit par au moins un auteur identifié par un nom et un prénom. Un texte peut également être préfacé par un autre auteur. La figure 3.2 présente le méta-modèle utilisé pour écrire notre modèle et défini grâce au méta-méta-modèle Ecore. Il s'agit d'une partie du méta-modèle UML défini par l'OMG [OMG, 2010]. Nous avons conservé une version simplifiée des diagrammes de classe contenant les classes, les attributs (représentés par des *property* dans le méta-modèle), les associations et les généralisations. Par souci de simplification les *GeneralizationSet*, bien qu'implémentés dans le méta-modèle, ne seront pas utilisés dans l'exemple. La figure 3.3 est la représentation sous forme de syntaxe abstraite du modèle telle que la représente l'outil dynamicGMF [dynamicGMF].

Le modèle de sortie est représenté à la figure 3.4 dans le langage de modélisation Entité-Association [Chen, 1976]. Il s'agit de la modélisation correspondant au précédent modèle dans un autre langage. Les classes sont devenues des *Entity* et les attributs des *Attribute*. Les associations quant à elles sont transformées en *Relationship* avec les *Role* et *Cardinality* adéquates. Le méta-modèle Entité-Association simplifié utilisé pour cet exemple est présenté à la figure 3.5. Nous n'avons là encore conservé que la partie nécessaire à l'exemple où les principaux éléments sont les *Entity* qui peuvent contenir des *Attribute* ou être reliés entre eux par des *Relationship*. La figure 3.6 est la représentation en syntaxe abstraite du modèle telle que la représente l'outil dynamicGMF [dynamicGMF].

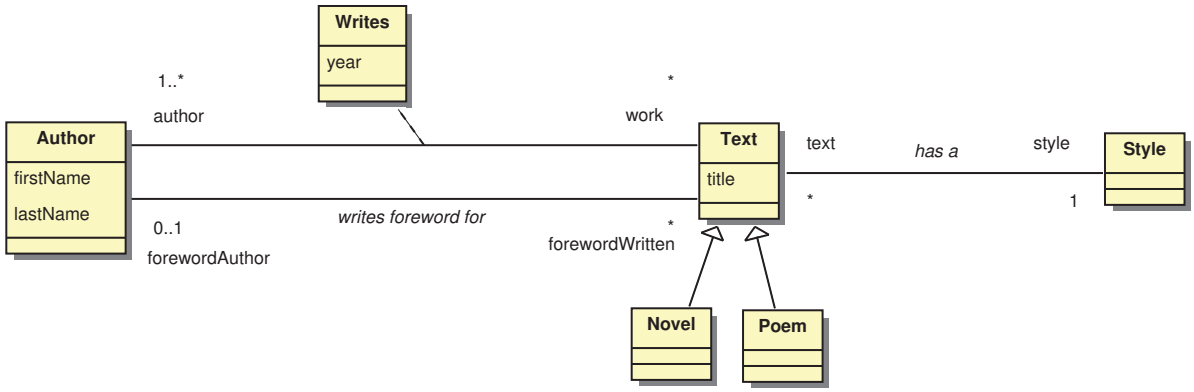


FIGURE 3.1 : Modèle d'exemple source en UML.

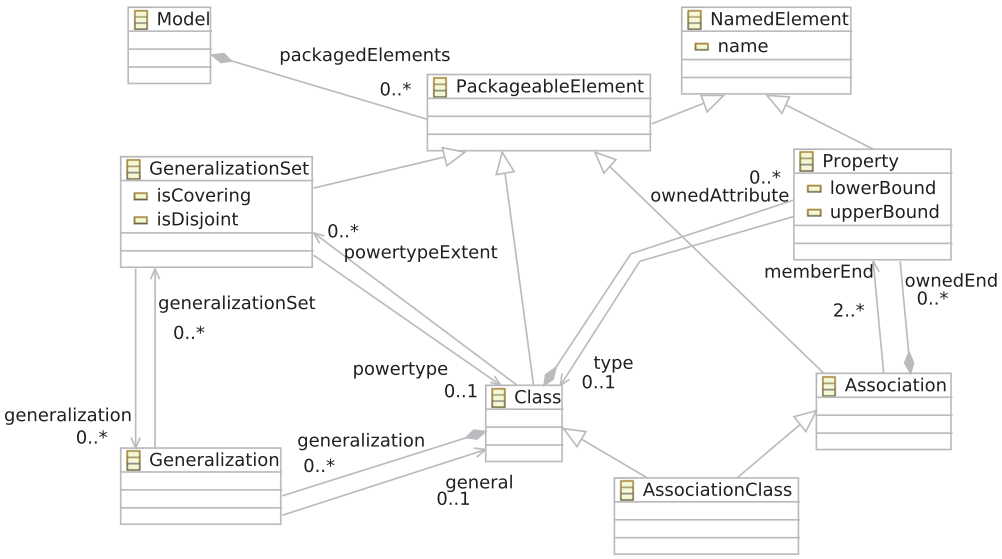


FIGURE 3.2 : Méta-modèle UML utilisé pour définir le diagramme de la figure 3.1.

3.2.2 Appariement basé sur les valeurs d'attributs

Nous allons ici présenter une technique d'alignement automatique de modèles basée sur les valeurs des propriétés des éléments des modèles. Notre but est de trouver un sous-ensemble des liens d'appariements entre les deux modèles, et ce avec le maximum de précision possible.

Les modèles que nous considérons sont des instances de méta-modèles composés essentiellement de classes, elles-mêmes contenant des attributs et des références vers d'autres classes. Ainsi un modèle est essentiellement composé d'objets reliés entre eux par

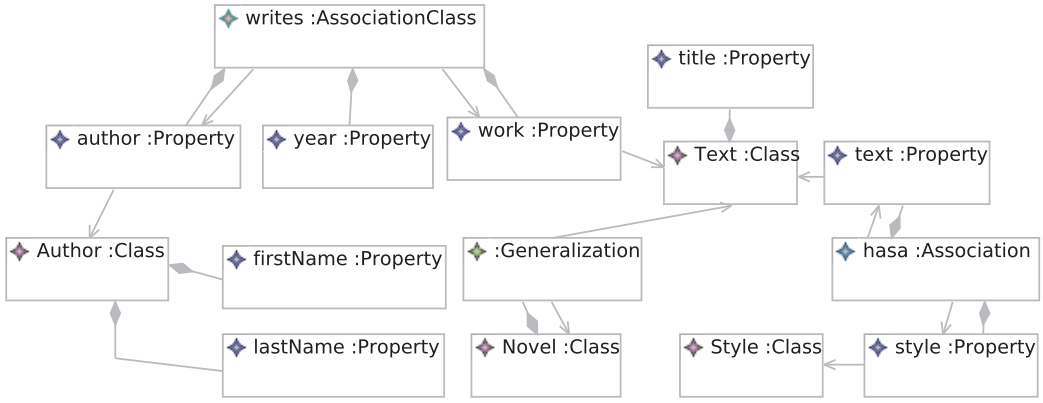


FIGURE 3.3 : Représentation sous forme de diagramme d'objets du diagramme UML de la figure 3.1.

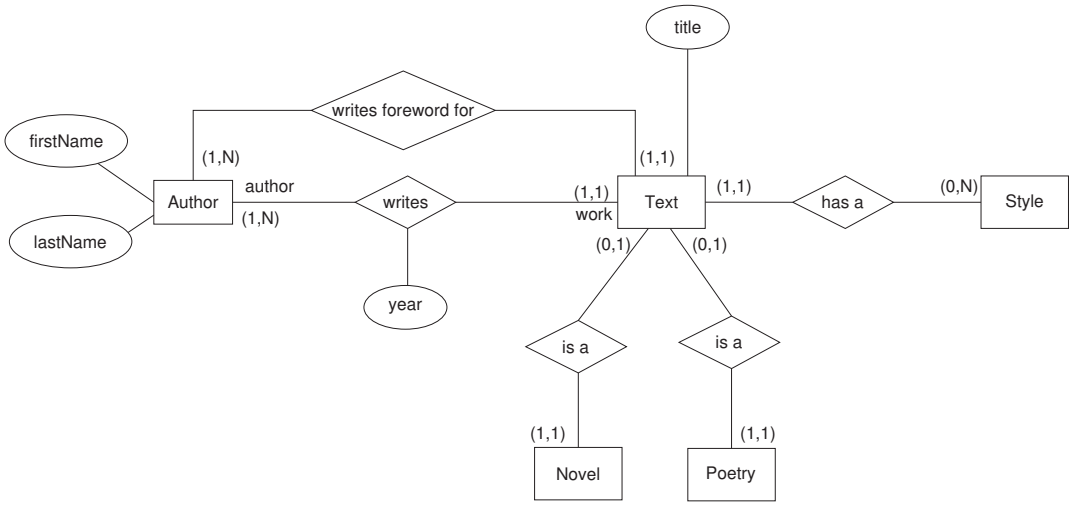


FIGURE 3.4 : Modèle d'exemple cible en Entité-Association.

des références et possédant des attributs. Les attributs ont un nom et un type définis au niveau du méta-modèle et une valeur définie au niveau du modèle. Les types des attributs pris en compte sont des types qui ne sont pas définis dans les méta-modèles source et cible. Ces types s'apparentent aux types primitifs définis dans les langages : entier, flottant, booléen, chaîne de caractères...

La méthode que nous employons doit être générique, et s'appliquer sur n'importe quel couple de modèles indépendamment de leur méta-modèle. La méthode d'appariement d'éléments se base sur les valeurs des attributs, il s'agit d'extraire des modèles l'ensemble des valeurs des attributs, et de comparer les valeurs des attributs du modèle source avec

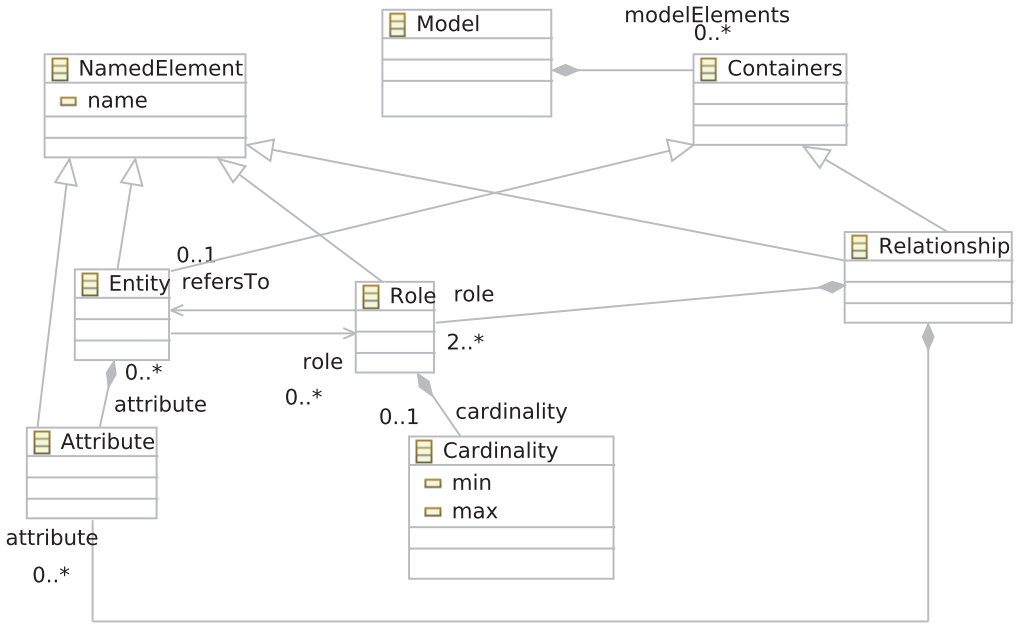


FIGURE 3.5 : Méta-modèle Entité-Association pour définir le diagramme de la figure 3.4.

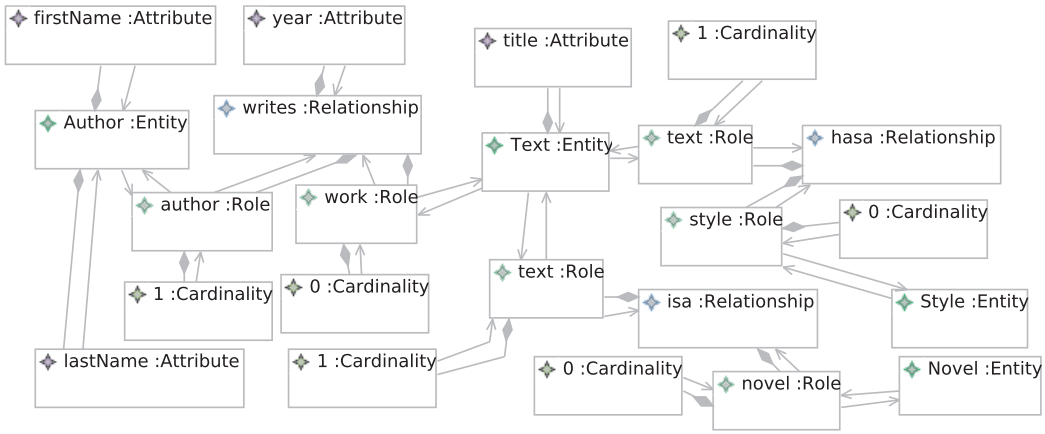


FIGURE 3.6 : Représentation sous forme de diagramme d'objets du diagramme Entité-Association de la figure 3.1.

celles du modèle cible. Dans le cas où il existe une ressemblance significative entre deux attributs, alors nous apparions les éléments qui les possèdent. Nous allons maintenant nous intéresser à la manière de juger de manière automatique de cette ressemblance.

Nous considérons ici les valeurs données aux attributs au niveau modèle. Nous appellerons ici « instance d'attribut » d'un modèle M un triplet $i = (o, a, v)$ où o est un élément de M, a est un attribut défini dans la classe de o, c'est-à-dire au niveau du méta-modèle de M et v est la valeur associée à a dans o. Ainsi le triplet $(author, name, "Author")$ dans le modèle de la figure 3.1 fait référence à la valeur de l'attribut *name* pour l'objet de type *Class* nommé *"Author"* (ici *author* est une référence vers l'objet). Pour une instance d'attribut i les fonctions $object(i)$, $attribute(i)$ et $value(i)$ permettent respectivement de récupérer l'objet, l'attribut et la valeur de i. Soit $I(M)$ l'ensemble des instances d'attributs du modèle M. Les instances d'attributs de notre exemple sont représentées dans la table 3.1 : la colonne objet contient un identifiant de l'objet contenant la valeur (obtenu à partir du hashcode de l'objet), la colonne *Attribut* contient le nom de l'attribut ainsi que celui de la classe qui le contient sous la forme *Classe::Attribut*, la colonne *Valeur* contient les valeurs données aux attributs.

Notre approche consiste pour deux modèles M_{src} et M_{tgt} à garder les couples d'éléments respectant une fonction de filtrage nommée `match`. Cela revient à obtenir un ensemble $A = \{(o_1, o_2) | \exists a_1, v_1, (o_1, a_1, v_1) \in I(M_{src}) \wedge \exists a_2, v_2, (o_2, a_2, v_2) \in I(M_{tgt}) \wedge match(v_1, v_2)\}$.

La construction de cet ensemble A est décrite par la procédure 1 où la fonction de filtrage `match` s'applique sur tous les éléments de $I(M_{src}) \times I(M_{tgt})$.

Procédure 1 : discoverAnchorPairs(M_{src}, M_{tgt})

Data : M_{src} : source model, M_{tgt} : target model
Result : A : set of anchors pair
$A \leftarrow \emptyset$;
foreach $x \in I(M_{src})$ *and* $y \in I(M_{tgt})$ **do**
  if `match`($value(x)$,$value(y)$,M_{src},M_{tgt}) **then**
    $A \leftarrow A \cup \{(object(x), object(y))\}$;

La fonction $match$ est une fonction vérifiant si les objets de deux instances d'attributs forment un lien d'appariement. Elle se base essentiellement sur la similarité des valeurs.

Nous proposons ici deux méthodes qui nous semblent pertinentes pour tester la similarité. La connaissance de cette méthode peut guider l'utilisateur pour créer des exemples qui pourront l'utiliser de manière efficace, c'est-à-dire en gardant exactement la même valeur d'un modèle à l'autre et en n'utilisant une valeur dans un modèle qu'une seule fois. Il est aussi possible de créer facilement une fonction de test de similarité spécifique à une transformation donnée en s'adaptant par exemple aux conventions de nommage pour supprimer les préfixes avant la comparaison.

TABLE 3.1 : Instances d'attributs du modèle source (à gauche) et du modèle cible (à droite) de notre exemple.

Modèle Source

Objet	Attribut	Valeur
26049230	Class::name	Author
24745276	Property::name	firstName
24745276	Property::lowerBound	0
24745276	Property::upperBound	1
20237898	Property::name	lastName
20237898	Property::lowerBound	0
20237898	Property::upperBound	1
30638546	Class::name	Text
15734641	Property::name	title
15734641	Property::lowerBound	0
15734641	Property::upperBound	1
9649099	Class::name	Novel
3852606	Class::name	Poetry
24257622	Class::name	Style
11511434	AssociationClass::name	writes
31375837	Property::name	work
31375837	Property::lowerBound	0
31375837	Property::upperBound	1
25804854	Property::name	author
25804854	Property::lowerBound	1
25804854	Property::upperBound	-1
11150143	Property::name	year
11150143	Property::lowerBound	0
11150143	Property::upperBound	1
32512553	Association::name	hasa
4558657	Property::name	style
4558657	Property::lowerBound	0
4558657	Property::upperBound	1
12590745	Property::name	text
12590745	Property::lowerBound	0
12590745	Property::upperBound	1
18414151	Association::name	writes foreword for
14111765	Property::name	forewordAuthor
14111765	Property::lowerBound	0
14111765	Property::upperBound	1
13725633	Property::name	forewordWritten
13725633	Property::lowerBound	0
13725633	Property::upperBound	1

Modèle cible

Objet	Attribut	Valeur
30969271	Entity::name	Author
4205299	Attribute::name	firstName
12839401	Attribute::name	lastName
3273383	Entity::name	Text
20039836	Attribute::name	title
23690087	Entity::name	Novel
2409635	Entity::name	Poetry
32098350	Entity::name	Style
22367538	Relationship::name	writes
20910958	Role::name	author
1352077	Cardinality::min	1
1352077	Cardinality::max	-1
8012937	Role::name	work
19509473	Cardinality::min	0
19509473	Cardinality::max	-1
1812813	Attribute::name	year
534353	Relationship::name	isa
21846985	Role::name	text
29683960	Cardinality::min	1
29683960	Cardinality::max	1
11024915	Role::name	novel
8180602	Cardinality::min	0
8180602	Cardinality::max	1
18885993	Relationship::name	isa
25515362	Role::name	text
11050211	Cardinality::min	1
11050211	Cardinality::max	1
30685694	Role::name	poetry
2850225	Cardinality::min	0
2850225	Cardinality::max	1
21559496	Relationship::name	writes foreword for
29705835	Role::name	forewordAuthor
9578500	Cardinality::min	1
9578500	Cardinality::max	0
25252664	Role::name	forewordWritten
25068634	Cardinality::min	0
25068634	Cardinality::max	-1
19097823	Relationship::name	hasa
28970806	Role::name	text
3975755	Cardinality::min	1
3975755	Cardinality::max	1
13640204	Role::name	style
7031149	Cardinality::min	0
7031149	Cardinality::max	-1

Fonction d'appariement basée sur l'égalité

Lors de la définition d'une telle fonction nous gardons toujours à l'esprit la nécessité d'une précision maximale, c'est pourquoi notre première fonction, décrite par la fonction 2, vérifie l'égalité parfaite entre deux valeurs. Cette égalité est à pondérer par le fait que certaines valeurs telles que des stéréotypes, des cardinalités, des booléens, etc. reviennent fréquemment dans les modèles, et s'appuyer sur ces valeurs serait source d'erreurs. Nous nous limitons donc aux chaînes de caractères et aux nombres, mais surtout deux instances d'attributs créent un couple si et seulement si leur valeur n'apparaît qu'une fois dans chaque modèle. Ainsi la fonction `frequency`($valeur, modèle$) retourne le nombre de fois où $valeur$ apparaît dans $modèle$.

Fonction 2 : match(v_{src}, v_{tgt},M_{src},M_{tgt})

Data : v_{src}, v_{tgt} : attribute values,
M_{src},M_{tgt} : models
Result : $match?$: boolean
$match? \leftarrow (\texttt{frequency}(v_{src}, M_{src}) = 1) \wedge$ **(a)**
$(\texttt{frequency}(v_{tgt}, M_{tgt}) = 1) \wedge$
$(v_{src} = v_{tgt})$;

Fonction d'appariement basée sur les sous-chaînes

Il arrive que la transformation implique un changement de convention de nommage : les classes peuvent par exemple avoir un préfixe « Cl_ ». Nous proposons un second test de similarité décrit par la fonction 3 qui permet de traiter le cas de l'ajout ou la suppression d'un préfixe et/ou d'un suffixe. Dans un tel cas la chaîne de caractères initiale est une sous-chaîne de la chaîne de caractères transformée ou inversement la chaîne de caractères transformée est une sous-chaîne de la chaîne de caractères initiale. La condition au repère (a) de la fonction d'appariement basées sur les sous-chaînes présentée par la fonction 2 vérifie si les valeurs que nous considérons apparaissent plusieurs fois dans leurs modèles respectifs. Si cette condition est vérifiée, alors les valeurs ne peuvent pas être utilisées de manière fiable. La condition au repère (b) combinée avec la précédente correspond à la condition de la fonction décrite par l'algorithme 2, nous avons donc la garantie d'obtenir un alignement incluant l'alignement obtenu par la première méthode $match$. Dans le cas d'une inégalité de valeur, il faut alors vérifier si une des deux valeurs est sous-chaîne de l'autre (repère (c)). La fonction $substring(s_1, s_2)$ retourne vrai si s_2 est sous-chaîne de s_1. Si une des valeurs est une sous-chaîne de l'autre, alors nous vérifions si la sous-chaîne

existe déjà dans le modèle de la plus grande chaîne. Cette vérification est nécessaire car dans le cas où elle existe, c'est avec elle que la sous-chaîne doit être appariée.

Fonction 3 : match(v_{src}, v_{tgt},M_{src},M_{tgt}) (sous-chaîne)

Data : v_{src}, v_{tgt} : attribute values,
M_{src},M_{tgt} : models
Result : $match?$: boolean
if (frequency(v_{src}, M_{src})$\neq 1$) $\wedge$ (frequency(v_{tgt}, M_{tgt}) $\neq 1$)) **then** (a)
  $match? \leftarrow false$;
else if $v_{src} = v_{tgt}$ **then** (b)
  $match? \leftarrow true$;
else if $\neg$substring(v_{src}, v_{tgt}) $\wedge$ $\neg$substring(v_{tgt}, v_{src}) **then** (c)
  $match? \leftarrow false$;
else if substring(v_{src}, v_{tgt}) **then** (d)
  $match? \leftarrow$ (frequency(v_{tgt}, M_{src}) = 0);
else if substring(v_{tgt}, v_{src}) **then** (e)
  $match? \leftarrow$ (frequency(v_{src}, M_{tgt}) = 0);

La table 3.2 représente le résultat obtenu en appliquant notre approche sur les données de la table 3.1. Le résultat est un ensemble de couples d'objets dont sont donnés les identifiants. Pour faciliter la lecture, la classe de l'objet et la valeur de l'attribut permettant l'appariement sont ajoutées entre parenthèses. Dans notre exemple l'utilisation des fonctions `match` basées sur l'égalité (fonction 2) ou les sous-chaînes de caractères (fonction 3) donnent le même résultat. On voit l'intérêt d'utiliser la fréquence d'apparition des valeurs sur les cardinalités ou les noms de rôle qui reviennent souvent tels que "text". La fréquence d'apparition pour les sous-chaînes est aussi utile car les associations "writes" et "writes foreword for" n'ont aucun lien bien qu'une valeur soit sous-chaîne de l'autre.

3.2.3 Adaptation de l'approche AnchorPROMPT

Lorsque toutes les classes d'un méta-modèle possèdent un attribut identifiant, la méthode d'appariement décrite dans la section 3.2.2 est suffisante. Mais il arrive bien souvent que les liens de l'alignement obtenu ne couvrent pas toute la transformation. Il est donc nécessaire d'envisager une approche complémentaire pour essayer d'obtenir les liens restant. Nous avons vu dans la section 3.1 que certaines méthodes d'alignement prennent en paramètre un alignement existant pour le compléter. Celle qui nous intéresse ici est la méthode AnchorPROMPT que nous allons adapter pour les besoins de notre approche.

Le principe général d'AnchorPROMPT est présenté par la figure 3.7. AnchorPROMPT prend en paramètre un alignement considéré comme précis et qui sera conservé à la fin du processus. Chaque élément appartenant à un couple de cet alignement est appelé une *ancre*. Prenons deux ancres a et b dans un modèle source séparé par un ensemble E d'éléments. Les transformés a' et b' de ces deux éléments a et b sont aussi des ancres et les

TABLE 3.2 : Alignement obtenu à partir de l'exemple.

Source	Cible
26049230 (Class "Author")	30969271 (Entity "Author")
24745276 (Property "firstName")	4205299 (Attribute "firstName")
20237898 (Property "lastName")	12839401 (Attribute "lastName")
30638546 (Class "Text")	3273383 (Entity "Text")
15734641 (Property "title")	20039836 (Attribute "title")
9649099 (Class "Novel")	23690087 (Entity "Novel")
3852606 (Class "Poetry")	2409635(Entity "Poetry")
24257622 (Class "Style")	32098350(Entity "Style")
11511434 (AssociationClass "writes")	22367538 (Relationship "writes")
31375837 (Property "work")	8012937 (Role "work")
25804854 (Property "author")	20910958 (Role "author")
11150143 (Property "year")	1812813 (Attribute "year")
32512553 (Association "hasa")	19097823 (Relationship "hasa")
4558657 (Property "style")	13640204(Role "style")
18414151 (Association "writes foreword for")	21559496 (Relationship "writes foreword for")
14111765 (Property "forewordAuthor")	29705835 (Role "forewordAuthor")
13725633 (Property "forewordWritten")	25252664 (Role "forewordWritten")

transformés des éléments de E ont de grandes chances de se trouver entre a' et b' et si un élément e de E est à une distance d de a alors son transformé e' a de grandes chances de se trouver à distance d de a'.

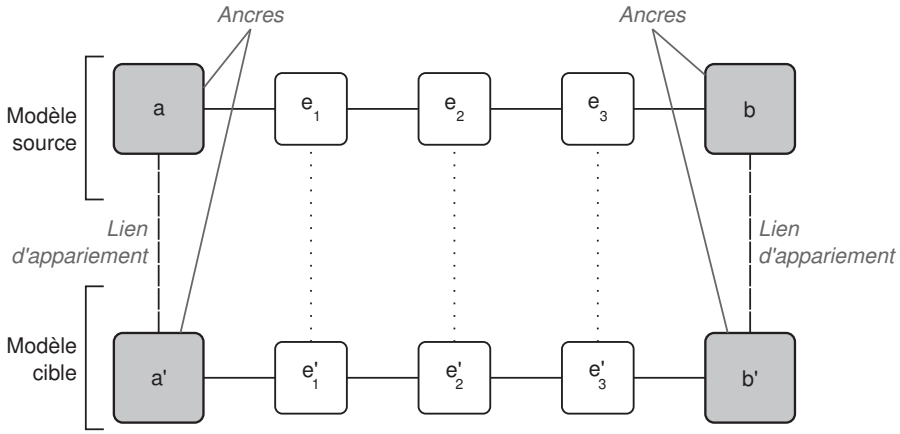


FIGURE 3.7 : Illustration de la méthode d'appariement utilisée pour AnchorPROMPT.

AnchorPROMPT a été défini pour un contexte différent du nôtre, et nous y avons apporté les adaptations suivantes :

– AnchorPROMPT s'applique sur des ontologies et de ce fait les éléments des modèles pris en entrée peuvent être classés par une relation de généralisation. An-

chorPROMPT utilise ce fait pour introduire dans son processus des regroupements d'éléments partageant la même généralisation. Nous ne pouvons pas procéder de la même manière pour nos modèles car nous n'avons pas de moyen de savoir si les méta-modèles utilisés définissent une relation de généralisation.
– AnchorPROMPT ne compare que des chemins de longueurs identiques entre deux ancres or l'expérience nous a montré que dans de nombreux cas sur les données que nous avons, les chemins n'étaient pas de longueur identique. Nous avons donc proposé une généralisation de la méthode aux chemins de longueurs proches.

Le processus de notre approche peut se découper en 3 étapes : il s'agit d'abord d'énumérer les chemins entre deux ancres et les transformés de ces deux ancres ; ensuite il faut calculer les indicateurs de correspondance pour chaque élément dans chaque couple de chemins ; enfin à partir des indicateurs de correspondance nous pouvons appliquer un filtrage.

Énumération des chemins entre les ancres

Définition 3.4 Chemin *: Nous considérons ici la définition propre à la théorie des graphes, c'est-à-dire une séquence d'éléments du modèle telle que deux éléments consécutifs sont reliés par une arête. Nous pouvons considérer les chemins simples où un élément peut apparaître deux fois dans un chemin, mais pas une arête, et les chemins élémentaires où un élément ne peut apparaître qu'une fois dans un chemin.*

La première étape est de trouver l'ensemble des chemins de longueur inférieure à une borne β reliant deux ancres à la fois dans le modèle source et le modèle cible. Nous considérons un modèle comme un graphe $G = (E, V)$ où l'ensemble des arêtes E est l'ensemble des références entre les éléments et l'ensemble des nœuds V est l'ensemble des instances des classes du méta-modèle. L'ensemble des ancres dans un modèle est donné par l'ensemble $A \subset V$.

Les procédures 4 et 5 décrivent la manière dont nous énumérons l'ensemble des chemins entre tous les couples d'ancres. Il s'agit essentiellement d'un parcours en profondeur du graphe à partir de chaque ancre. Contrairement à un parcours en profondeur classique nous ne nous contentons donc pas de passer une fois par nœud mais nous parcourons tous les chemins en limitant la profondeur du parcours à β.

Il faut ensuite trouver, à partir de ces deux ensembles de chemins, les couples de chemins du modèle source et du modèle cible dont les extrémités correspondent. Nous appellerons par la suite ces chemins des *chemins correspondants*.

Calcul de l'indicateur de correspondance entre éléments

Pour chaque couple de chemins correspondants nous prenons l'ensemble des couples d'éléments formé par le produit cartésien de l'ensemble des nœuds d'un chemin avec

Procédure 4 : enumeratePaths(M,A,β)

Data : M : model,
A : set of anchors in M,
β : depth max
Result : P : set of paths
$P \leftarrow \emptyset$;
foreach $a \in A$ **do**
  $P \leftarrow P \cup$ `searchPaths`(a, a, β, A);

Procédure 5 : searchPaths($path$,e,$depth$,A)

Data : $path$: path already visited,
e : current node,
$depth$: maximum depth from current node,
A : set of anchors
Result : P : set of paths
$P \leftarrow \emptyset$;
`mark`(e);
if $e \in A$ **then**
  $P \leftarrow P \cup$ `append`$(path, e)$;
if $depth \neq 0$ **then**
  foreach $x \in neighbors(e)$ **do**
    if $\neg$`isMarked?`(x) **then**
      $P \leftarrow P \cup$ `searchPaths`(`append`$(path, e), e, depth - 1, A$);
`unmark`(e);

l'ensemble des nœuds de l'autre chemin. Nous calculons un indicateur de correspondance pour chaque couple d'éléments, cet indicateur est un nombre rationnel supérieur ou égal à 0. Il vaut 0 par défaut et est augmenté à chaque fois que le couple d'éléments appartient à un couple de chemins correspondants. Par exemple dans la figure 3.8 le couple de nœuds (x_1, y_1) apparaît dans les couples de chemins correspondants suivants : $((a, x_1, x_2, c), (a', y_1, y_2, c'))$, $((a, x_1, x_2, c), (a', y_1, y_3, y_4, c'))$ et $((b, x_1, x_3, d), (b', y_1, d'))$. Pour chaque couple de chemins correspondants nous allons calculer un indicateur de correspondance local de (x, y) défini par la somme de tous ces indicateurs et qui formera l'indicateur de correspondance du couple de nœuds.

Contrairement à AnchorPROMPT il n'est pas requis de prendre des chemins de longueurs identiques et nous augmentons l'indicateur de correspondance du couple de manière plus forte si les éléments se retrouvent à la même position relativement à la longueur

des chemins considérés. Cette situation est décrite par la figure 3.9 : (x_3, y_2) a un indicateur de correspondance maximal car x_3 et y_2 sont au milieu de leurs chemins respectifs, par contre (y_1, x_5) a un indicateur de correspondance très faible car y_1 et x_5 ont des positions très différentes dans leurs chemins respectifs.

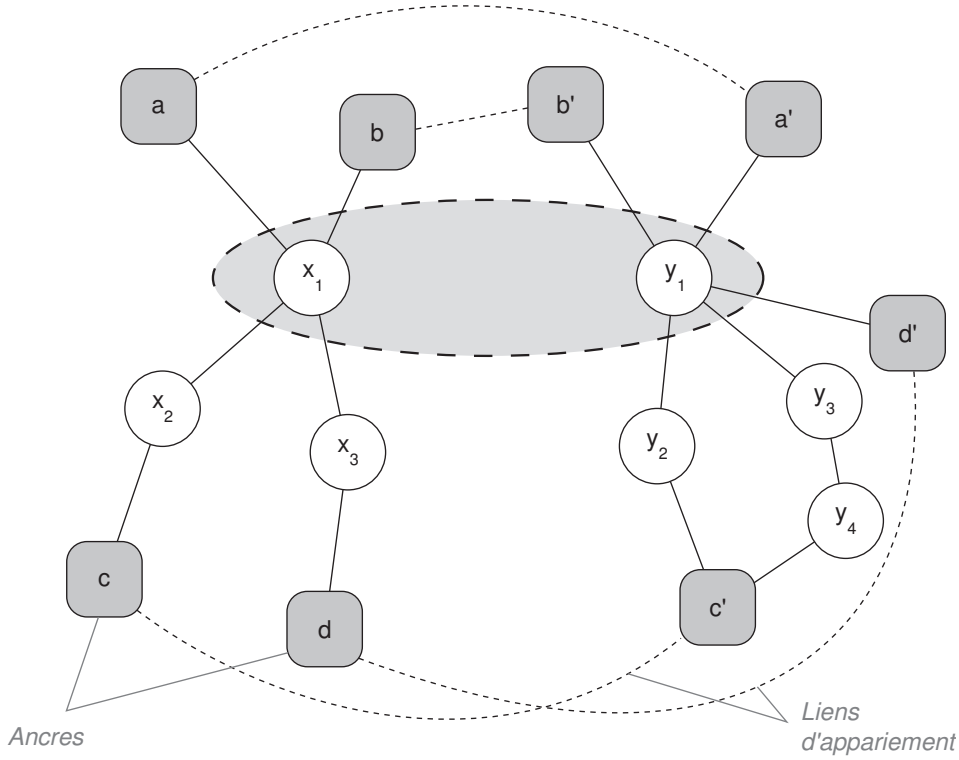


FIGURE 3.8 : Illustration des chemins correspondants.

Plus précisément, soient X et Y deux chemins correspondants. X et Y commencent et finissent donc par deux ancres qui forment un lien d'appariement dans l'alignement A passé en paramètre. Soient $x \in X$ et $y \in Y$ où x et y ne sont pas des extrémités de X et Y. La fonction $index(x)$ retourne la position de x. La formule permettant de déterminer l'indicateur de correspondance du couple (x, y) est :

$$W(x,y) = 1 - \left| \frac{index(x)}{length(X)-1} - \frac{index(y)}{length(Y)-1} \right|$$

Ainsi dans la figure 3.9, $W(x_1, y_1) = 1 - \left|\frac{1}{6} - \frac{1}{4}\right| = 0.92$ et $W(x_1, y_2) = 1 - \left|\frac{1}{6} - \frac{2}{4}\right| = 0.67$, ce qui nous montre que x_1 a un indicateur de correspondance plus élevé avec y_1 qu'avec y_2.

Dans le cas où un couple d'éléments se trouve sur plusieurs couples de chemins correspondants, nous faisons la somme des indicateurs de correspondance.

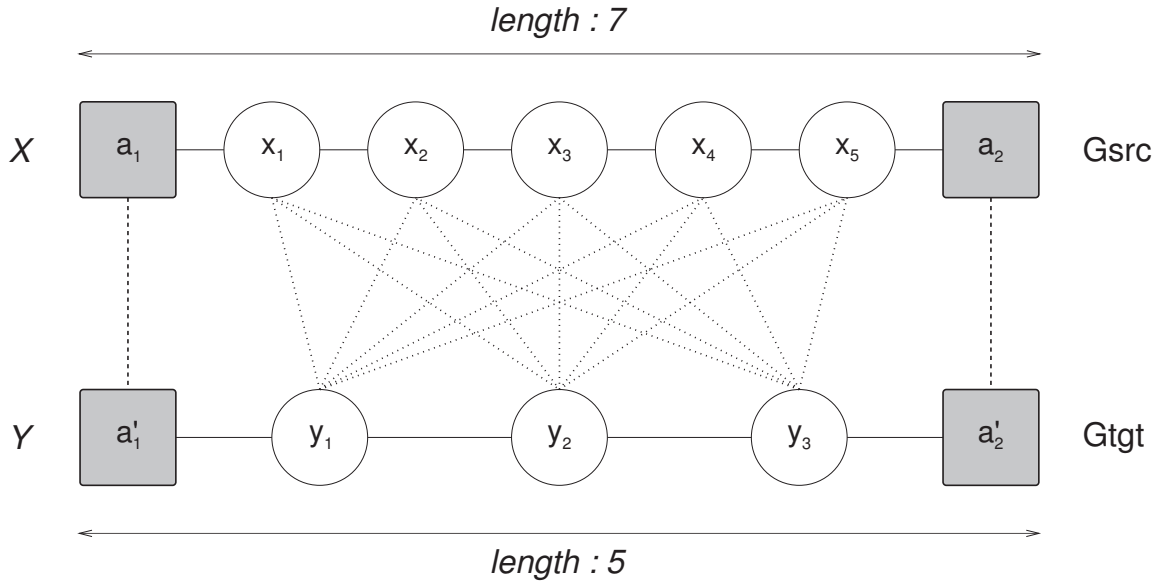


FIGURE 3.9 : Alignement de deux chemins correspondants.

Filtrage des correspondances

À l'issue de l'étape précédente de calcul d'indicateurs de correspondance, nous avons un grand nombre de couples avec un indicateur de correspondance. Si l'indicateur est élevé, alors nous pouvons considérer qu'il s'agit d'un lien d'appariement tandis que les couples avec les indicateurs les plus faibles sont à éliminer. Le problème qui se pose est qu'il n'est pas possible de déterminer un plafond global pour les indicateurs. En effet un couple apparaissant dans plusieurs chemins correspondants mais à des positions différentes peut avoir un indicateur plus faible qu'un couple placé à la même position mais dans un seul couple de chemins correspondants. Nous proposons donc de déterminer le plafond de manière locale pour chaque nœud. Nous considérons que chaque nœud possédant au moins un indicateur de correspondance avec un autre nœud supérieur à 0 possède au moins un lien de correspondance. Nous pouvons donc considérer un plafond local à partir de l'indicateur maximum.

Pour chaque élément du modèle source (resp. cible), nous prenons l'indicateur de correspondance le plus élevé et nous gardons uniquement les liens avec les éléments du modèle cible (resp. source) qui ont un indicateur dont la valeur est supérieure à un plafond dépendant de la valeur la plus élevée (dans notre étude présentée à la section 3.3 nous fixons ce plafond à 80% de l'indicateur maximal). Un lien est considéré comme un lien d'appariement s'il a été conservé pour ses deux extrémités.

La méthode de filtrage est illustrée par la figure 3.10. Si nous prenons en compte l'élément *text :Property* faisant partie du modèle source, il est associé à plusieurs éléments du

modèle cible. Le lien ayant le plus fort indicateur de correspondance est celui avec l'élément *text :Role*. Ce dernier est relié à plusieurs éléments du modèle source et parmi ces liens, celui avec l'élément *text :Property* a l'indicateur de correspondance le plus élevé. Ce lien, avec l'indicateur le plus fort à la fois pour la source et la cible peut être considéré comme lien d'appariement.

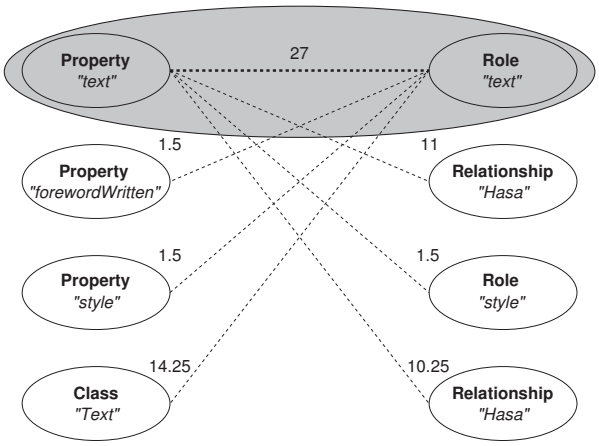


FIGURE 3.10 : Alignement de deux chemins correspondants.

3.3 Étude de cas

Nous proposons dans cette section une étude de cas pour valider nos hypothèses. Nous décrirons tout d'abord notre implémentation dans la section 3.3.1. Le choix des exemples influe grandement sur les résultats : la technique d'appariement fonctionnera d'autant mieux si l'auteur des exemples prend en considération les mécanismes d'appariement automatiques. Un autre problème est de trouver un ensemble de transformations qui puisse être considéré comme représentatif. Nous expliquerons dans la section 3.3.2 comment nous sélectionnons les transformations et les exemples de transformations sur lesquels nous appliquerons ce protocole. Nous décrirons dans la section 3.3.3 un ensemble de métriques adaptées à la comparaison d'un modèle produit de manière automatique par rapport à un modèle de référence. Nous préciserons le protocole de test et nous analyserons enfin les résultats dans la section 3.3.4.

3.3.1 Outil

La figure 3.11 illustre de manière schématique le processus de notre approche. Nous avons basé l'implémentation de notre outil sur la plate-forme de développement Eclipse et son framework de modélisation EMF [Budinsky *et al.*, 2003]. Ainsi, les méta-modèles

que nous manipulons sont tous conformes au méta-méta-modèle Ecore qui est au cœur de EMF. L'étape d'appariement décrite par la section 3.2.2 est présentée ici sous le nom de *Anchor Discovery*. Elle est liée à l'étape de propagation décrite à la section 3.2.3 à laquelle elle fournit un premier alignement. L'association des deux méthodes forme notre outil d'appariement.

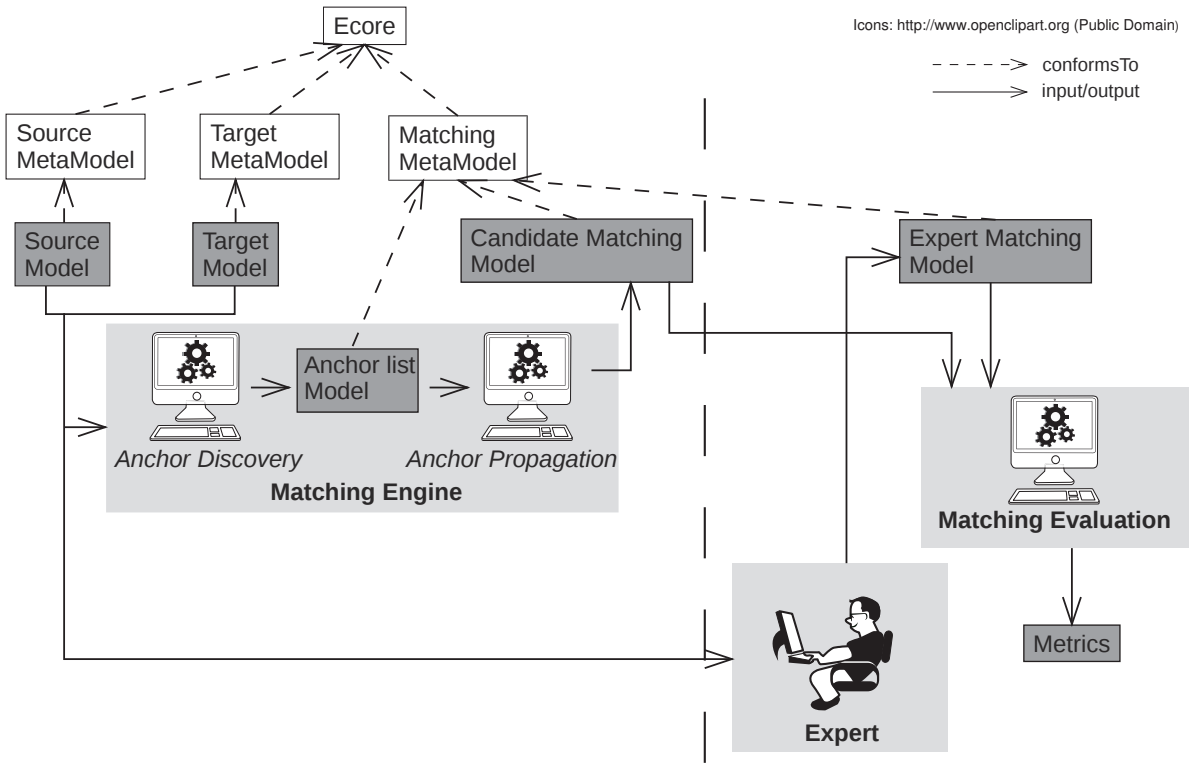


FIGURE 3.11 : Schéma illustrant le fonctionnement de l'outil.

La description des alignements se fait conformément au méta-modèle que nous avons développé et qui est décrit par la figure 3.12. Un alignement (*MatchingModel*) est décrit par un ensemble de couples d'éléments (*MatchingLink*). Les éléments sont ici des *EObject* ce qui permet à ce méta-modèle de s'adapter à tous les modèles au format EMF.

La figure 3.11 décrit aussi la manière dont s'organise notre méthode d'évaluation de l'outil. La partie droite de la figure montre que nous avons développé un outil prenant en paramètre l'alignement résultat du *Matching Engine* pour le comparer à un alignement créé à la main par un expert. Le résultat alors obtenu est un ensemble de valeur permettant d'évaluer de manière automatique la qualité de l'alignement obtenu grâce aux métriques définies.

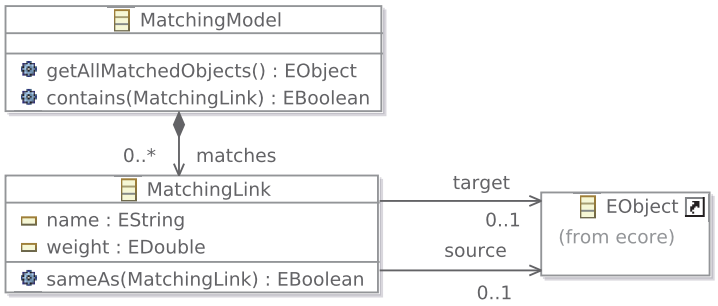


FIGURE 3.12 : Méta-modèle d'alignement.

3.3.2 Sélection des exemples

Le zoo ATL

La principale source d'exemples que nous utilisons ici est le zoo des transformations ATL [Zoo ATL]. Il s'agit d'un dépôt public où l'on trouve des transformations de modèles écrites en ATL. Pour la plupart de ces transformations sont fournis non seulement le code et les méta-modèles mais aussi un exemple couvrant une grande partie de la transformation.

Le langage ATL est un langage de transformation largement utilisé. Il s'agit d'un langage principalement déclaratif, permettant l'implémentation de transformations de modèles à partir de règles, mais proposant, lorsqu'une implémentation purement déclarative s'avère difficile, la possibilité d'utiliser des procédures impératives nommées *helper*.

Les travaux de [Mens et Gorp, 2006] et [Czarnecki et Helsen, 2006] classent les transformations d'après différents critères : la modification ou non du degré d'abstraction ainsi que le sens de cette modification (augmentation ou diminution de ce degré) ; la différence ou non des métamodèles source et cible ou la nature des modèles transformés (modèles ou programmes). Cette première classification a été utilisée afin de sélectionner des transformations de différentes catégories. Nous avons également défini des métriques qui visent à capturer le profil de ces transformations.

Nombre de Règles de la transformation (Number of Transformation Rules, NOR). Ce nombre inclut le nombre de règles déclaratives et impératives. Son interprétation doit être prudente puisqu'une valeur élevée peut malgré tout traduire un simple alignement du modèle source sur le modèle cible, donc être relativement simple. Un nombre de règles supérieur au nombre d'éléments du modèle source peut indiquer que la construction de certains éléments du modèle cible est composite et requiert l'utilisation de *helpers*. Regarder le détail des règles permet de s'en assurer.

Nombre de Helpers (Number Of Helpers, NOH). La présence de *helpers* indique généralement qu'une navigation complexe est nécessaire : dans le modèle source, soit pour collecter des informations, soit pour effectuer des opérations complexes ; ou dans le modèle cible, notamment pour initialiser des valeurs ou effectuer des opérations complexes. Dans certains cas, certaines règles sont conditionnées par des valeurs d'attribut ou par la présence ou l'absence de certains éléments dans l'environnement de l'élément à transformer.

Nombre de Lignes de Code (Number of Lines Of Code, NLOC). Le nombre de lignes de code complète la description donnée par le nombre de règles. Le nombre moyen de lignes de code par règle indique la complexité moyenne de l'initialisation des éléments dans le modèle cible ou la complexité moyenne du filtre de la règle. Les lignes blanches et les commentaires ne sont pas comptabilisés.

Nombre de Conditions (Number Of Conditions, NOC). Le nombre de conditions correspond au nombre d'instructions if(condition)then instruction endif qui apparaissent dans une transformation. Cette valeur doit être rapprochée du nombre de règles de transformations. Une valeur proche de 1 indique que de nombreuses règles sont conditionnées par des valeurs d'attribut ou par la présence ou non de certains éléments dans l'environnement de l'élément à transformer.

Nombre de Règles Filtrées (Number of Filtered Rules, NOFR). Le nombre de règles filtrées correspond au nombre de règles qui disposent d'un filtre. Cette valeur discrimine les règles de transformation qui n'effectuent un traitement que sous certaines conditions qui doivent être réalisées dans le modèle source.

Nombre de Règles Déclaratives (Number Of Declarative Rules, NODR). Ce nombre correspond au nombre de règles déclaratives (avec ou sans filtre). Cette valeur est connectée au nombre d'alignements effectués à travers la transformation. Comme une règle peut, notamment par des instructions de navigation, traiter plusieurs éléments du modèle cible ou source, cela rend la valeur de cette métrique approximative.

Nombre de Règles Impératives (Number Of Imperative Rules, NOIR). Ce nombre correspond au nombre de règles impératives. Elles sont utilisées quand une règle de transformation est difficilement exprimable de manière déclarative par exemple lorsque des calculs (opérations sur des collections ou calculs complexes) doivent être effectués.

TABLE 3.3 : Analyse des transformations du zoo ATL (avril 2009)

Transformation	NOR	NOH	NLOC	NOC	NOFR	NODR	NOIR

TABLE 3.3 : Analyse des transformations du zoo ATL (avril 2009) – suite–

Transformation	NOR	NOH	NLOC	NOC	NOFR	NODR	NOIR
Accessors	22	21	868	0	0	22	0
Ant2Maven	29	0	256	0	0	29	0
Applet	25	13	641	0	0	25	0
AssertionModification	13	7	220	0	0	13	0
AsyncMethods	3	3	53	0	1	3	0
ATL_WFR	18	21	591	0	18	18	0
ATL2BindingDebugger	2	0	33	0	0	2	0
ATL2Tracer	2	0	86	0	0	2	0
BibTeX2DocBook	9	4	174	0	8	9	0
Book2Publication	1	3	35	0	1	1	0
Class2Relational	6	1	89	0	4	6	0
CodeClone2SVG	3	1	126	0	0	3	0
CopyModel	2	0	43	0	2	2	0
CPL2SPL	15	7	272	0	2	15	0
DataTypes	23	15	559	0	2	23	0
Disaggregation	7	2	133	0	0	7	0
DSLModel2KM2	10	7	210	2	9	10	0
EliminateRedundantInheritance	8	3	120	0	0	8	0
emf2km3	11	1	103	0	1	9	2
EquivalenceAssoc.Attributes	20	12	506	0	0	20	0
Families2Persons	2	2	40	0	2	2	0
ForeignKey	10	0	118	0	0	8	2
GeometricalTransformations	3	3	109	5	0	3	0
Grafcet2PetriNet	5	0	61	0	0	5	0
IEEE1471_2_MoDAF	13	1	203	0	0	13	0
IntroducePrimaryKey	10	0	119	0	0	8	2
IntroducingInterface	9	1	144	0	0	9	0
Java2DataTypes	23	19	571	0	2	23	0
JavaSource2Table	2	2	57	0	0	2	0
KM32ATLCopier	2	4	131	0	1	2	0
KM32CONFATL	10	14	956	0	0	10	0
KM32DOT	7	18	334	0	0	7	0
KM32EMF	10	1	117	0	2	10	0
KM32Measure	10	3	265	6	2	2	8
KM32Metrics	2	15	195	1	0	2	0
KM32OWL	14	4	249	11	1	11	3
KM32Problems	18	7	326	0	18	18	0
km32xml	2	0	117	0	0	2	0
Make2Ant	5	0	64	0	0	5	0

TABLE 3.3 : Analyse des transformations du zoo ATL (avril 2009) – suite–

Transformation	**NOR**	**NOH**	**NLOC**	**NOC**	**NOFR**	**NODR**	**NOIR**
MDL2GMF	3	12	225	0	0	3	0
Measure2Table	7	2	119	5	0	0	7
Measure2XHTML	23	8	726	30	0	5	18
MergeModel	20	3	434	0	0	20	0
MergingPartialClasses	11	2	171	4	0	9	2
ModelCopy	105	0	1827	0	105	105	0
module	10	14	956	0	0	10	0
Monitor2Semaphore	24	10	661	0	15	24	0
MySQL2KM3	11	17	459	0	10	11	0
OCL2R2ML	37	11	757	1	11	37	0
PathExp2PetriNet	3	1	65	0	0	3	0
Public2Private	2	1	77	0	2	2	0
R2ML2OCL	37	8	507	0	9	37	0
R2ML2RDM	58	31	976	0	10	58	0
RaiseSupplier	8	1	148	0	0	8	0
RedundantClassRemovable	12	2	199	4	0	10	2
Removing	11	2	207	4	0	9	2
Replace	12	0	146	0	0	10	2
RSS2ATOM	3	0	57	0	0	3	0
SD2flatSTMD	11	33	264	0	0	11	0
SimplePDL2PetriNet	5	0	230	2	2	5	0
Singleton	3	2	107	0	2	3	0
SpreadsheetMLSimplified2SoftwareQ.C.	4	4	116	1	2	4	0
SpreadsheetMLSimplified2XML	12	1	179	0	0	12	0
SSL2SDL	12	15	260	0	1	12	0
Table2SpreadsheetMLSimplified	3	1	93	0	0	3	0
Table2SVGBarChart	4	15	264	1	0	2	2
Table2SVGPieChart	8	17	442	1	0	6	2
Table2TabularHTML	8	1	101	0	1	6	2
Tree2List	1	0	16	0	1	1	0
TypeA2TypeB	3	0	24	0	2	3	0
UML22Measure	10	6	318	7	2	2	8
UML2AnyLogic	2	19	729	21	1	2	0
UML2ClassDiagramToKM3	4	6	143	3	3	2	2
UML2JAVA	6	4	67	0	6	6	0
UML2MOF	11	8	363	0	5	11	0
UML2OWL	25	7	515	24	11	17	8
UML2Transformations	6	1	85	0	0	6	0
UmlActivityDiagram2MSProject	3	3	70	0	2	3	0

TABLE 3.3 : Analyse des transformations du zoo ATL (avril 2009) – suite–

Transformation	NOR	NOH	NLOC	NOC	NOFR	NODR	NOIR
UMLCD2UMLProfile	4	0	109	0	3	3	1
UMLDI2SVG	14	27	1473	0	0	14	0
WSDL2R2ML	17	3	248	5	2	17	0
XML2KML	84	5	922	0	0	84	0
XML2SpreadsheetMLSimplified	11	10	261	0	11	11	0
XSLT2XQuery	7	0	145	0	3	7	0
Moyenne	12,45	6,17	298,59	1,59	3,46	11,59	0,86
Min	1	0	16	0	0	0	0
Max	105	33	1827	30	105	105	18
Ecart type	15,75	7,45	320,45	4,85	11,65	15,85	2,52

Dans cette liste de 84 transformations (table 3.3), nous avons sélectionné quelques transformations selon plusieurs critères :

- faisabilité technique : les données qui nous sont nécessaires sont les méta-modèles source et cible de la transformation, des modèles sources et cibles d'exemple et une spécification précise de la transformation. De nombreuses transformations ont été écartées simplement parce que l'une de ces données manquait ou n'était pas exploitable, notamment quand les modèles ne sont pas au format XMI.
- complexité de la transformation : nous avons essayé de ne pas prendre que des transformations simples, mais qui ne soient pas non plus trop complexes pour pouvoir analyser facilement les résultats obtenus.
- la transformation est constituée principalement de règles déclaratives, qui correspondent à ce que l'on peut espérer découvrir avec les techniques utilisées.

TABLE 3.4 : Extrait choisi des transformations du zoo ATL.

Transformation	NOR	NOH	NLOC	NOC	NOFR	NODR	NOIR
Book2Publication	1	3	35	0	1	1	0
Class2Relational	6	1	89	0	4	6	0
Disaggregation	7	2	133	0	0	7	0
EliminateRedundantInheritance	8	3	120	0	0	8	0
emf2km3	11	1	103	0	1	9	2
EquivalenceAssoc.Attributes	20	12	506	0	0	20	0
Families2Persons	2	2	40	0	2	2	0
JavaSource2Table	2	2	57	0	0	2	0
KM32EMF	10	1	117	0	2	10	0
KM32Problems	18	7	326	0	18	18	0
UML2ClassDiagramToKM3	4	6	143	3	3	2	2
IntroducingInterface	9	1	144	0	0	9	0

La table 3.4 représente la liste des transformations que nous avons extraites du zoo ATL en suivant les précédents critères.

Les transformations choisies

Aux transformations choisies dans le zoo ATL se rajoutent celles créées dans notre équipe pour tester des cas particuliers de l'approche et d'autres transformations connues que nous avons implémentées. On trouvera dans l'annexe A la liste détaillée des transformations.

De ces différentes sources nous avons réuni 22 couples d'exemples source et cible. Neuf de ces couples spécifient des transformations endogènes tandis que les 13 couples restants spécifient des transformations exogènes. Les transformations endogènes sont des refactorings et les transformations exogènes sont essentiellement des traductions des données de modèles dans d'autres formalismes ayant une sémantique similaire. On peut malgré tout remarquer un sous-ensemble de ces transformations, dont la définition est informelle, où seule une partie des données sont extraites pour les représenter dans un formalisme avec un but différent.

Les transformations endogènes sélectionnées sont les suivantes (présentées par ordre alphabétique) :

- delegation1 (inspiré de [Fowler *et al.*, 2000])
- delegation2 (inspiré de [Fowler *et al.*, 2000])
- Disaggregation ([Zoo ATL])
- EliminateRedundantInheritance ([Zoo ATL])
- EquivalenceAttributesAssociations ([Zoo ATL])
- extractClass (inspiré de [Fowler *et al.*, 2000])
- hidedelegate (inspiré de [Fowler *et al.*, 2000])
- IntroducingInterface ([Zoo ATL])
- IntroducePrimaryKey ([Zoo ATL])

La figure 3.13 présente les modèles source et cible utilisés pour la transformation endogène *delegation1*.

Les transformations exogènes sélectionnées sont les suivantes (présentées par ordre alphabétique) :

- associations-persons (originale)
- Book2Publication ([Zoo ATL])
- emf2km3 ([Zoo ATL])
- Ecore2Class ([Zoo ATL])
- Families2Persons ([Zoo ATL])
- JavaSource2Table ([Zoo ATL])
- KM32EMF ([Zoo ATL])
- KM32problems ([Zoo ATL])
- uml-er3 (originale)

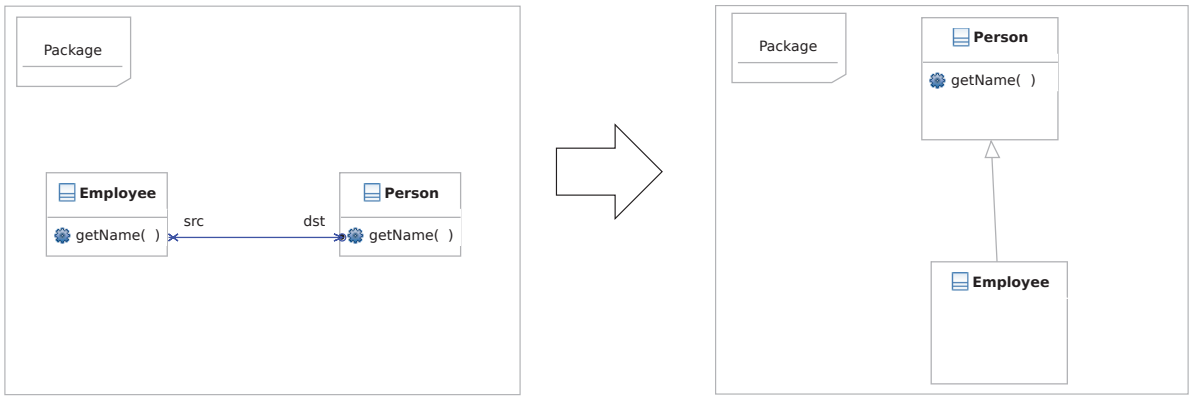


FIGURE 3.13 : *delegation1* : une transformation endogène de modèles UML. Ici sont présentés deux modèles UML, à gauche un modèle source et à droite le modèle obtenu par la transformation *delegation1*.

– uml-er (originale)
– uml-er2 (originale)
– uml-er-iccs (originale)
– UML2ClassDiagramToKM3 ([Zoo ATL])

Des exemples de transformations exogènes sont présentés dans les sections 4.3 et 4.4.

3.3.3 Métriques

Notre problème est similaire aux problèmes d'alignements de schémas ou d'ontologies. Or la littérature fournit pour ces derniers un ensemble de métriques d'évaluation [Do *et al.*, 2002] éprouvées et reconnues par la communauté. Ces métriques nous permettront d'évaluer la qualité de l'alignement A_{auto} obtenu par un outil en comparaison de celui créé par un expert A_{expert}. Nous appellerons $A_{positifs} = A_{auto} \cap A_{expert}$ l'ensemble des liens d'appariement présents à la fois dans l'alignement obtenu par l'outil et celui créé par l'expert (voir figure 3.14).

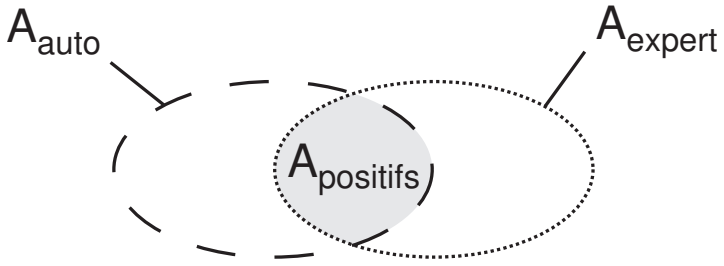


FIGURE 3.14 : Séparation des différents ensembles de paires d'appariement.

Précision. La précision (*precision* en anglais) est une métrique permettant de juger la qualité des couples d'appariements obtenus dans A_{auto}, c'est-à-dire le ratio des couples donnés par l'outil qui appartiennent à l'alignement créé par l'expert. Cette métrique est donc sensible à la quantité de mauvais couples qu'une méthode d'alignement produit. La précision est un nombre rationnel entre 0 et 1 ; la valeur 0 est obtenue si $A_{positifs} = \emptyset$ et augmente avec le nombre d'appariements corrects dans A_{auto} pour atteindre dans le meilleur cas la valeur 1 si tous les appariements de A_{auto} sont corrects, c'est-à-dire si $A_{auto} \subseteq A_{expert}$. Elle se calcule grâce à la formule :

$$précision = \left| \frac{A_{positifs}}{A_{auto}} \right|$$

Rappel. Le rappel (*recall* en anglais) est une métrique permettant de juger dans quelle proportion l'appariement A_{auto} est complet, c'est-à-dire le ratio des couples donnés par l'expert qui appartiennent à l'alignement fourni par l'outil. Cette métrique est donc sensible à la quantité de couples non produits par une méthode d'alignement. Le rappel est un nombre rationnel entre 0 et 1 ; la valeur 0 est obtenue si $A_{positifs} = \emptyset$ et augmente avec le nombre d'appariements de A_{expert} obtenus dans A_{auto} pour atteindre dans le meilleur cas la valeur 1 si tous les appariements experts sont découverts de manière automatique, c'est-à-dire si $A_{expert} \subseteq A_{auto}$. Elle se calcule grâce à la formule :

$$rappel = \left| \frac{A_{positifs}}{A_{expert}} \right|$$

Ces deux métriques sont complémentaires et une méthode d'alignement idéale doit obtenir $précision = rappel = 1$. Il arrive cependant que l'on veuille privilégier une métrique par rapport à l'autre. Par exemple l'approche de découverte d'ancres décrite dans la section 3.2.2 privilégie la précision par rapport au rappel car le résultat obtenu sert de donnée d'entrée à la méthode décrite dans la section 3.2.3 et doit contenir le moins de données erronées possible.

F-score. F-score (*F-score* ou *F-measure* en anglais) est une métrique combinant la précision et le rappel. La précision et le rappel sont pondérés dans cette métrique par une variable $\alpha \in \mathbb{Q}$ entre 0 et 1. Plus α est proche de 0 et plus le rappel est privilégié par rapport à la précision. La précision est privilégiée lorsque α s'approche de 1. Ainsi, pour $\alpha = 0$ la métrique correspond au rappel et pour $\alpha = 1$ elle correspond à la précision. Dans le cas où $\alpha = 0.5$ le résultat est la moyenne harmonique du rappel et de la précision. Dans tous les cas le F-score est un nombre rationnel entre 0 et 1. La valeur 0 est obtenue lorsque $A_{positifs} = \emptyset$ et augmente avec la précision et le rappel pour obtenir la valeur 1 dans le meilleur cas. On obtient le F-score par la formule suivante :

$$\text{F-score}(\alpha) = \frac{|A_{positifs}|}{(1-\alpha) \times |A_{expert}| + \alpha \times |A_{auto}|}$$

Effort. L'effort (*Overall* ou *Accuracy* en anglais) est une métrique combinant la précision et le rappel, mais contrairement au F-score qui mesure la qualité des couples d'un alignement, son but est de quantifier l'effort nécessaire pour corriger l'alignement fourni par un outil. Contrairement aux précédentes métriques, l'effort est un nombre rationnel compris entre $-\infty$ et 1. Il est égal dans le meilleur cas à 1 si $precision = rappel = 1$ et diminue lorsque le nombre d'appariements de A_{auto} n'appartenant pas à $A_{positifs}$ augmente et lorsque le nombre d'appariements de A_{expert} n'appartenant pas à $A_{positifs}$ augmente. C'est-à-dire que l'effort dépend du nombre d'appariements à enlever ou à rajouter à A_{auto} pour obtenir A_{expert} et s'approche de 1 lorsque ce nombre est faible par rapport à la taille de A_{expert}. L'effort s'obtient grâce à la formule suivante :

$$effort = 1 - \frac{(|A_{auto}| - |A_{positifs}|) + (|A_{expert}| - |A_{positifs}|)}{|A_{expert}|}$$

3.3.4 Résultats

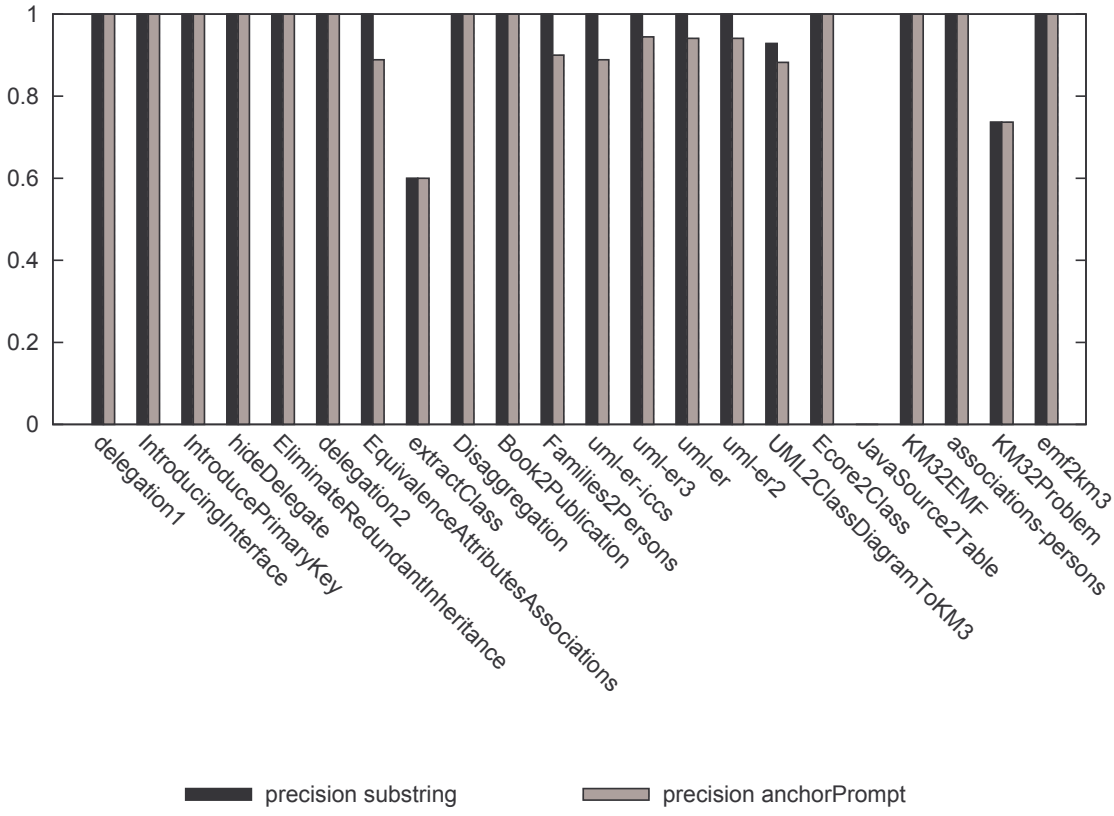


FIGURE 3.15 : Calcul de la précision sur les exemples de l'étude de cas.

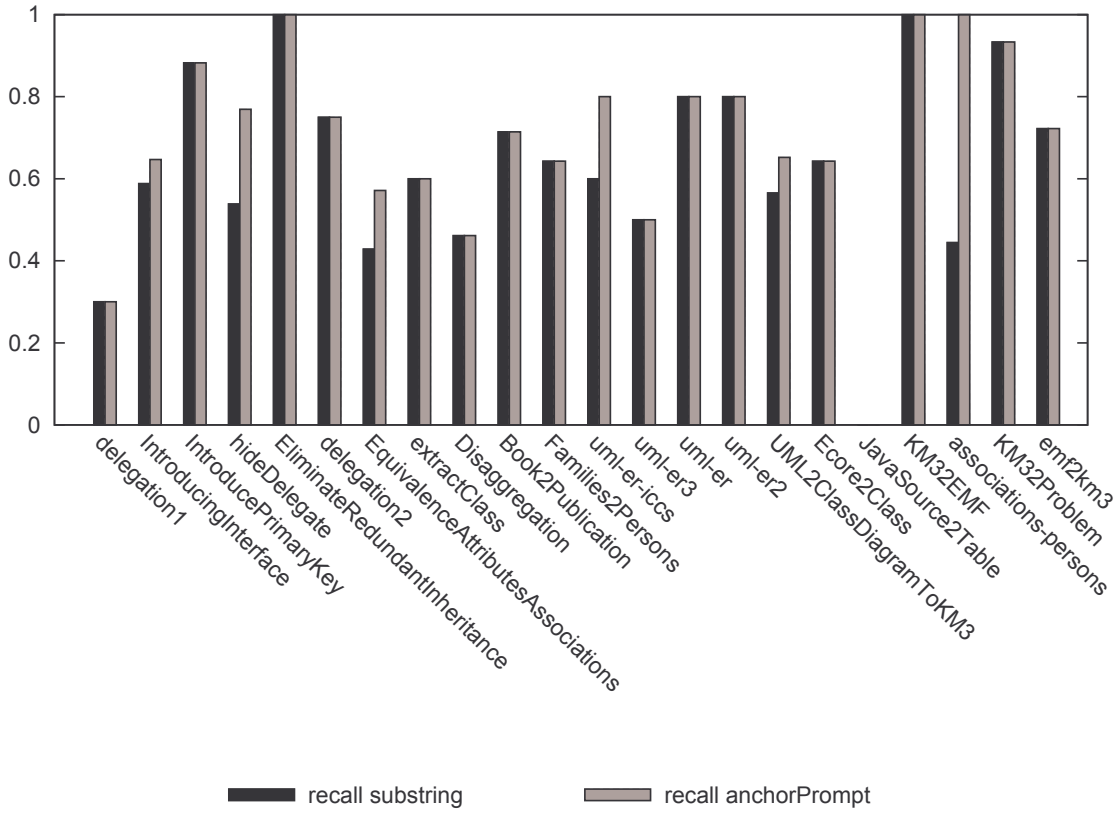


FIGURE 3.16 : Calcul du rappel sur les exemples de l'étude de cas.

Protocole expérimental

Pour tous les exemples, un alignement que nous considérons correct a été réalisé par nos soins. Cet alignement est l'*alignement expert*. Nous lançons ensuite la procédure automatique d'alignement sur chaque exemple et nous récupérons deux alignements : le premier alignement est l'alignement obtenu uniquement par la technique d'appariement basée sur les valeurs d'attributs (voir section 3.2.2) en utilisant la technique de comparaison des sous-chaînes, tandis que le second est le résultat de l'application de notre adaptation de l'approche AnchorPROMPT sur le premier alignement. Ces deux appariements sont comparés automatiquement à l'alignement expert et les figures 3.15,3.16, 3.17 et 3.18 montrent les valeurs obtenues pour les différentes métriques étudiées.

Analyse des résultats

La figure 3.15 présente la précision pour chaque alignement c'est-à-dire la qualité des éléments trouvés sans prendre en compte la proportion des appariements de l'alignement

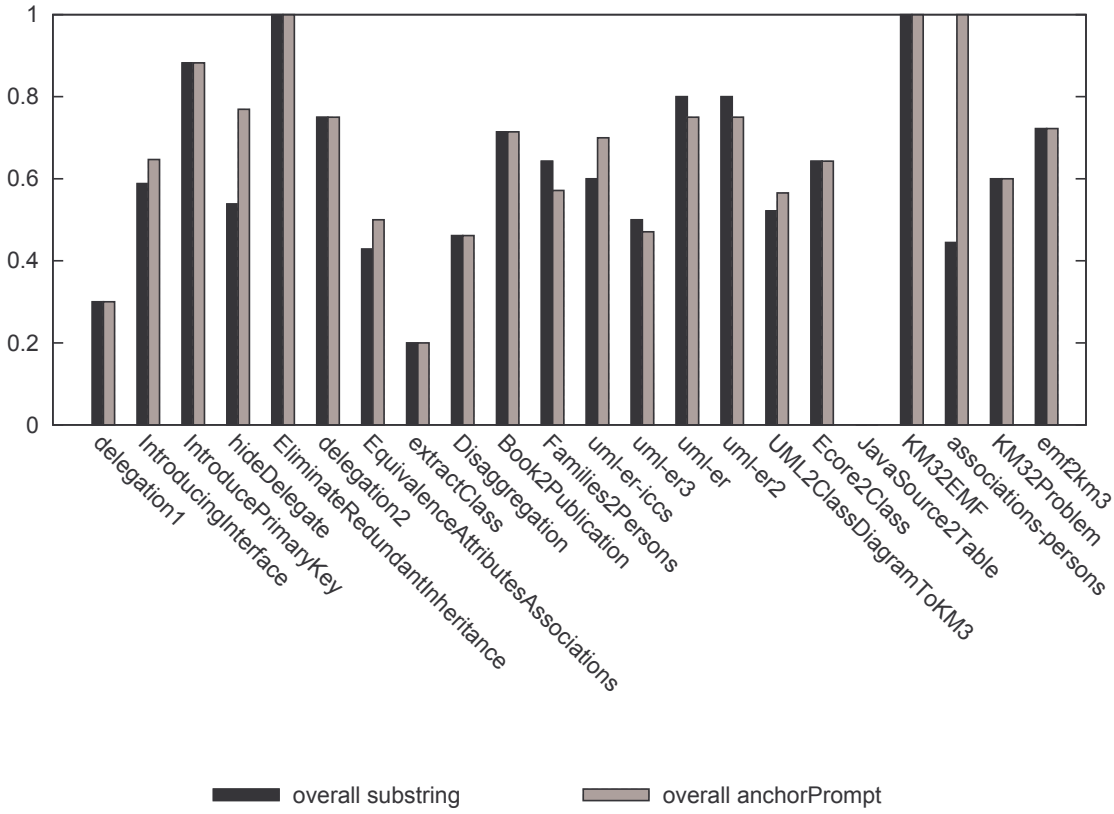


FIGURE 3.17 : Calcul de l'effort sur les exemples de l'étude de cas.

expert absents. Nous voyons que dans la globalité, notre approche fournit des alignements avec peu de mauvais appariements.

Dans 18 cas sur 22, la technique d'appariement basée sur les valeurs d'attributs donne une précision parfaite. Lorsqu'on complète les alignements obtenus avec notre adaptation d'AnchorPROMPT la précision baisse dans un certain nombre de cas. Ces résultats correspondent à nos attentes concernant la précision de l'appariement basé sur les valeurs d'attributs que nous voulions la plus haute possible pour pouvoir utiliser le résultat de cette méthode comme base à notre adaptation d'AnchorPROMPT. La perte de précision observée lors de l'utilisation de cette dernière apparaît assez raisonnable pour être corrigée manuellement à la fin du processus d'appariement.

La figure 3.16 présente les valeurs du rappel pour les alignements de notre étude de cas. La valeur n'atteint la valeur 1 que dans 3 cas, les autres alignements sont tous incomplets. Cela montre bien que notre méthode, bien qu'ayant pour but d'aider la création d'alignement, reste une méthode semi-automatique nécessitant une intervention humaine. Les résultats de la méthode AnchorPROMPT apparaissent ici en dessous de ce qui était at-

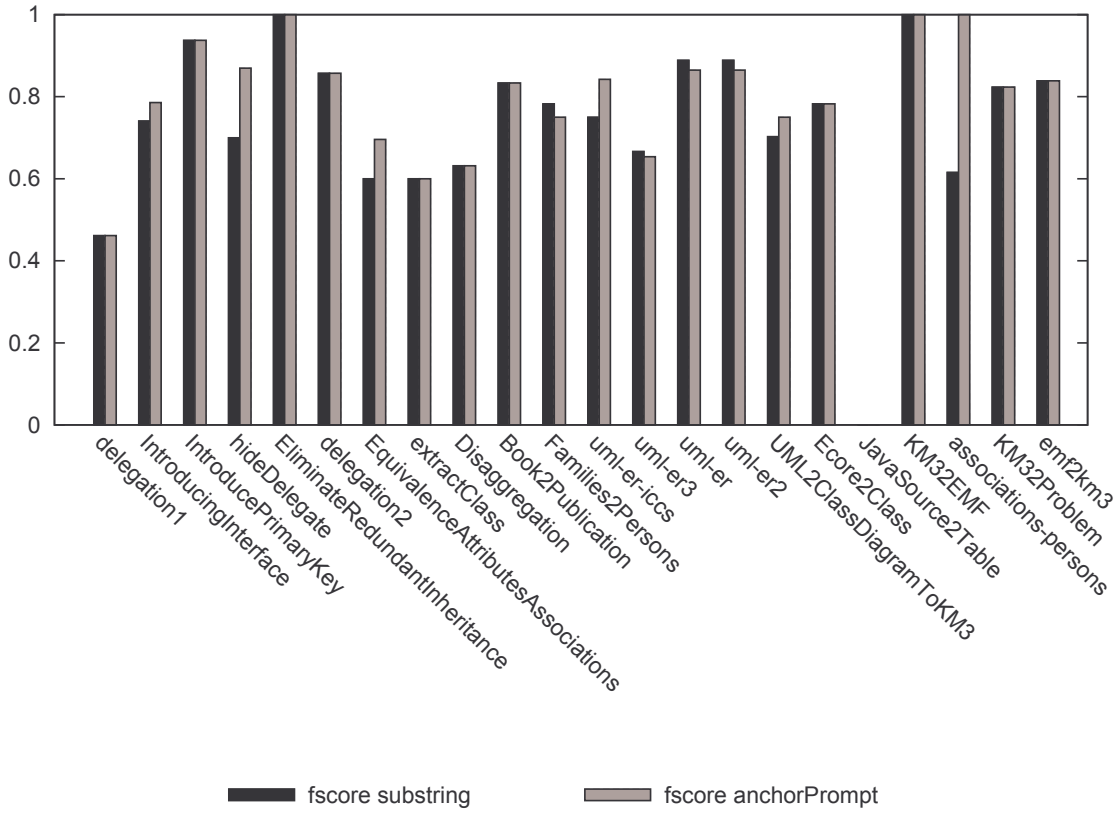


FIGURE 3.18 : Calcul du f-score ($\alpha = 0.5$) sur les exemples de l'étude de cas.

tendu car elle n'améliore la valeur du rappel que dans 6 cas. Cette impression se confirme avec les calculs d'effort où l'on voit qu'AnchorPROMPT ne diminue pas l'effort nécessaire pour rendre les alignements corrects dans beaucoup de cas, et l'augmente dans quelques cas. Ce constat nous encourage à envisager de nouvelles pistes pour améliorer encore l'approche AnchorPROMPT de manière à limiter l'introduction de mauvais appariements et à améliorer la déduction de nouveaux appariements.

Le f-score présenté à la figure 3.18 est un peu redondant avec le calcul d'effort dont les valeurs sont assez proches lorsque le paramètre du f-score est 0.5. On remarque pour toutes les métriques qu'une des transformations donne des résultats nuls. Cette transformation est une illustration des limites de notre approche. Il s'agit de la transformation nommée *JavaSource2Table* qui prend un modèle de code Java en paramètre et retourne pour chaque définition de méthode combien de fois est appelée chaque méthode du programme. Ce résultat est présenté par un modèle instance du méta-modèle *Table* qui modélise un tableau comme un ensemble de lignes, chaque ligne contient un ensemble de cellules et chaque cellule contient une valeur. Au niveau de l'alignement le problème qui

se pose est que chaque ligne et chaque colonne représentent une méthode, et la valeur dans une cellule correspond au nombre d'appel de la méthode de la colonne à laquelle elle appartient dans la définition de la méthode de la ligne à laquelle est appartient. Ainsi le premier obstacle à la création d'un alignement automatique vient du fait que chaque nom de méthode apparaît deux fois, or notre approche basée sur les valeurs d'attributs nécessite qu'une valeur n'apparaisse qu'une fois dans chaque modèle. Le second obstacle vient du fait qu'il n'est pas possible de relier de manière automatique les valeurs des cellules correspondant aux nombres d'appels de méthodes, avec les appels de méthode dans le modèle Java. Ainsi la méthode d'alignement utilisée ne permet pas d'obtenir des liens d'appariement corrects sur cet exemple. De manière plus générale on remarquera que dans cet exemple la transformation perd une très grande partie de la sémantique du modèle de départ. Il n'y a par exemple aucun lien entre une ligne et une colonne représentant la même méthode et la sémantique des valeurs de cellule n'est pas définie par le méta-modèle *Table*. Il n'y a rien non plus dans le méta-modèle qui permette de différencier une cellule contenant un nom de méthode avec une cellule contenant un nombre d'appels. Cette perte de sémantique explique les résultats obtenus sur cet exemple.

3.4 Conclusion

Le problème de l'alignement de modèles est très important dans le cadre d'une approche de génération de transformations de modèles car de la qualité de l'alignement de modèles dépend l'efficacité de l'approche de génération de transformation. En effet, un alignement incomplet ne permettra pas de générer toutes les règles de la transformation et un alignement erroné engendrera la création de règles au comportement incorrect. Cette tâche est cependant fastidieuse et de ce fait propice à l'introduction d'erreurs dans le processus.

Nous avons donc proposé une approche de génération automatique de l'alignement de modèles, basée sur l'approche d'alignement d'ontologies AnchorPROMPT, et permettant de faciliter le processus d'alignement. Cette approche a été découpée en deux parties : une première partie génère des liens d'appariement entre éléments à partir de mesures de similarités dans les valeurs des attributs de chaque élément ; une deuxième partie se base sur ce premier alignement et génère de nouveaux liens d'appariement en considérant les éléments se trouvant sur des chemins entre des éléments déjà appariés.

L'approche a été implémentée dans un outil que nous avons utilisé pour l'évaluer. L'évaluation de l'approche a été effectuée sur un ensemble d'exemples de transformations provenant de source diverses. Nous avons calculé pour chaque exemple de transformation la précision, le rappel, l'effort et le f-score du résultat de notre approche par rapport à un alignement expert. Les résultats sont encourageants, particulièrement pour la génération des liens d'appariement à partir des valeurs d'attributs qui donne de très bons résultats en termes de précision. On notera toutefois que la seconde étape de l'approche a quelque-

fois tendance à faire baisser la précision. Ce constat nous encourage à prévoir de nouvelles améliorations de l'approche, comme par exemple ajouter des tests de similarité sur les valeurs d'attribut dans le processus d'AnchorPROMPT.

Chapitre 4

Génération de transformations à partir d'exemples

La transformation de modèles est un aspect très important dans le développement dirigé par les modèles. Les langages et outils de transformations sont nombreux mais leur usage est réservé aux experts de la transformation ayant une bonne maîtrise du langage de transformation et du méta-méta-modèle sous-jacent, ainsi qu'une bonne connaissance des méta-modèles impliqués dans la transformation c'est-à-dire du domaine. Malheureusement les experts du domaine n'ont généralement pas les compétences suffisantes pour maîtriser les technologies dirigées par les modèles et les développeurs de transformations de modèles ne sont bien souvent pas experts du domaine.

Model Transformation By Example (MTBE) [Varró, 2006] est une proposition pour laisser la spécification de la transformation aux experts du domaine en leur demandant de réaliser des exemples représentatifs de modèles sources et les modèles transformés correspondants ainsi que des liens de transformation indiquant pour chaque élément d'un modèle transformé quels sont les éléments qui ont permis de le construire. En utilisant une approche d'apprentissage il est alors possible de créer la transformation de modèles correspondant aux exemples. De cette manière, les experts du domaine participent activement au développement sans qu'ils aient besoin d'interagir avec le code de la transformation.

Nous allons présenter dans ce chapitre une approche de génération de transformation basée sur MTBE. Nous allons tout d'abord présenter dans la section 4.1 un exemple illustratif. Nous nous appuierons sur cet exemple tout d'abord pour définir les concepts et présenter le processus de l'Analyse Relationnelle de Concepts dans la section 4.2. Ensuite nous présenterons dans la section 4.3 le fonctionnement de notre approche. À l'issue de cette présentation nous décrirons dans la section 4.4.1 l'implémentation de cette approche que nous avons mise en œuvre pour réaliser les expérimentations de notre étude de cas décrite dans la section 4.4.

4.1 Exemple illustratif

Pour faciliter la compréhension, nous allons présenter dans cette section l'exemple sur lequel nous nous appuierons pour décrire notre approche. Nous avons souhaité ici créer nos propres méta-modèles pour que le lecteur garde à l'esprit que notre approche n'est pas liée à un méta-modèle particulier comme UML.

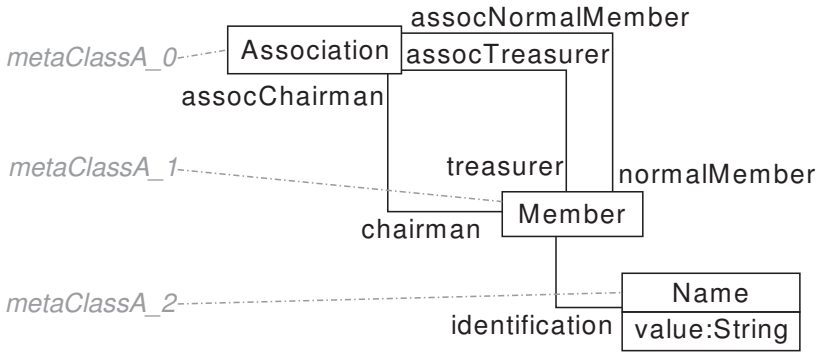


FIGURE 4.1 : Méta-modèle source de l'exemple illustratif (en gris sont indiqués des identifiants associés à chaque élément auxquels on se référera par la suite).

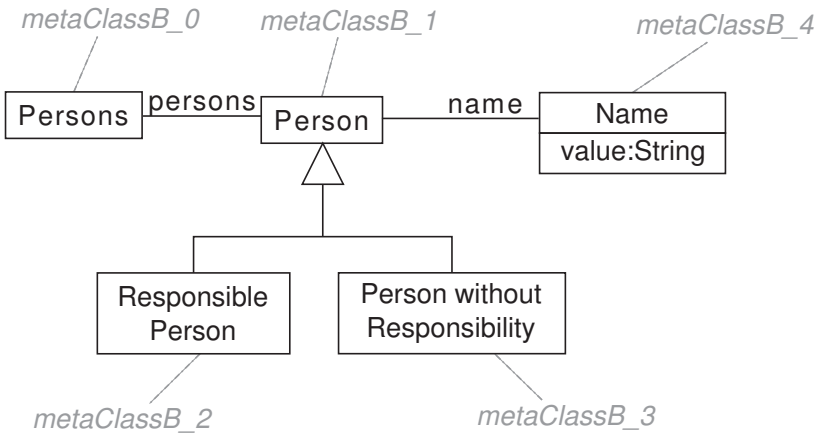


FIGURE 4.2 : Méta-modèle cible de l'exemple illustratif (en gris sont indiqués des identifiants associés à chaque élément auxquels on se référera par la suite).

Notre approche a pour but de générer une transformation de modèles, il nous faut donc définir un méta-modèle source et un méta-modèle cible. Le méta-modèle source que nous considérons ici est le méta-modèle *Association* présenté par la figure 4.1. Ce méta-modèle présente un ensemble de personnes possédant un nom (*Name*) et réunies au sein d'une

association. Ainsi chaque membre (*Member*) possède un rôle différent dans l'association : il peut être membre simple (*normalMember*), trésorier (*treasurer*) ou président de l'association (*chairman*).

Le méta-modèle cible considéré est le méta-modèle *Persons* présenté par la figure 4.2. Ce méta-modèle présente un ensemble de personnes possédant un nom (*Name*) et classées en deux catégories, les personnes avec responsabilité (*Responsible Person*) et les personnes sans responsabilité (*Person without Responsibility*).

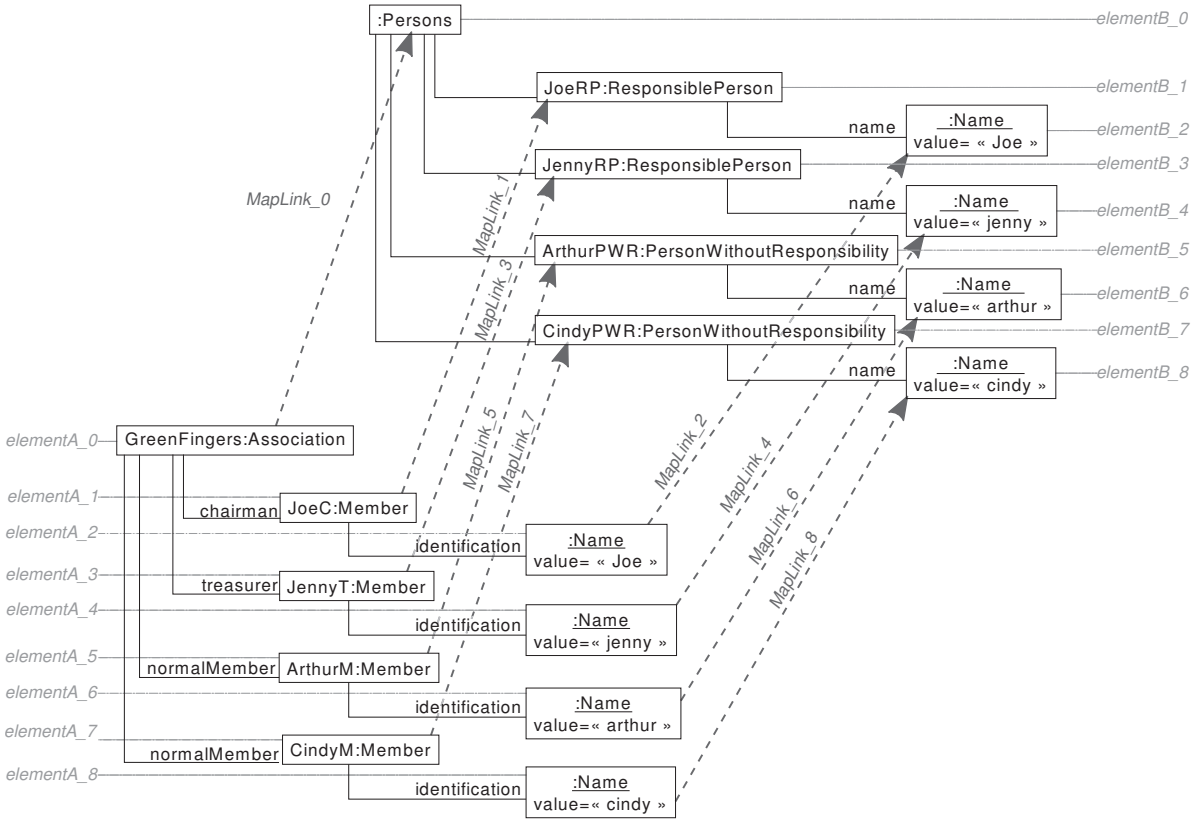


FIGURE 4.3 : Exemple illustratif : en bas à gauche un modèle *Association* et en haut à droite le modèle *Persons* correspondant (en gris sont indiqués des identifiants associés à chaque élément auxquels on se référera par la suite).

La transformation que l'on souhaite obtenir transforme les membres « normaux » en personnes sans responsabilité et les autres membres en personnes avec responsabilité.

La figure 4.3 présente un exemple de modèle conforme au méta-modèle source et son modèle correspondant conforme au méta-modèle cible. Les modèles sont présentés en utilisant la syntaxe UML des diagrammes d'instances. Des liens d'appariement entre les différents éléments des deux modèles ont été ajoutés, ils sont considérés corrects, qu'ils aient été obtenus manuellement ou automatiquement comme décrit dans le chapitre 3.

4.2 Analyse Relationnelle de Concepts

L'Analyse Formelle de Concepts (AFC [Ganter et Wille, 1999]) est une technique de classification d'objets en regroupements pertinents basés sur des caractéristiques communes. Cette technique ainsi que l'extension qui nous intéresse, l'Analyse Relationnelle de Concepts (ARC [Huchard *et al.*, 2007a]), font partie de la théorie des treillis et sont dans la continuité des travaux de [Birkhoff, 1940] et [Barbut et Monjardet, 1970]. Nous allons dans la section 4.2.1 décrire le fonctionnement de l'AFC, car l'ARC que nous décrirons dans la section 4.2.2 se base sur l'AFC.

4.2.1 Analyse Formelle de concepts

Les données d'entrée de l'AFC est un *contexte formel*, qui représente la relation binaire entre un ensemble d'objets et un ensemble d'attributs.

Définition 4.1 (Contexte Formel) *Un contexte formel est un triplet $K = (O, A, I)$, où O et A sont respectivement l'ensemble des objets et celui des attributs (caractéristiques), et $I \subseteq O \times A$ contient un couple (o, a) si et seulement si o possède a.*

Par exemple, en utilisant les données de l'exemple de la section 4.1, le contexte formel de la table 4.1 représente une classification des éléments du modèle *Association* présenté à la figure 4.3 en fonction du type de relation qui les relie à d'autres éléments. L'ensemble des objets du contexte est formé par l'ensemble des *éléments* du modèle. Nous appelons *éléments* les instances de classes du méta-modèle. Chaque élément est représenté par un identifiant de la forme *ElementA_x*. L'ensemble des attributs du contexte est formé par l'ensemble des rôles d'associations dans le méta-modèle (ou *EReference* dans EMF). Un couple objet-attribut appartient à la relation d'incidence si l'élément représentant l'objet est source d'au moins une association dont le type correspond à l'attribut du couple. Par exemple, *ElementA_0* représente l'*Association* « GreenFingers » et il existe deux *normalMember* dans cette *Association*. Donc, la relation d'incidence du contexte contient le couple $(ElementA_0, normalMember)$. Par contre, l'*Association* « GreenFinger » ne possède pas d'*identification*, la relation d'incidence ne contiendra donc pas le couple $(ElementA_0, identification)$.

Définition 4.2 (Concept) *Soient $X \subseteq O$ et $Y \subseteq A$. Le couple (X, Y) est un concept si et seulement si $X = \{o \in O | \forall y \in Y, (o, y) \in I\}$ et $Y = \{a \in A | \forall x \in X, (x, a) \in I\}$.*

X est l'extension du concept (objets couverts) et Y son intension (attributs partagés).

*La paire $(\{ElementA_5, ElementA_7\}, \{identification, assocNormalMember\})$, par exemple, forme un concept du contexte de la table 4.1 (*Concept_3 *de la figure 4.4). Les croix correspondant à toutes les paires objets-attributs du concept forment dans la table représentant le contexte un pavé maximal. C'est-à-dire que, aux permutations de lignes et*

TABLE 4.1 : Contexte Formel représentant l'appartenance de liens par des éléments.

	normalMember	chairman	treasurer	assocNormalMember	assocChairman	assocTreasurer	identification
ElementA_0	x	x	x				
ElementA_1					x		x
ElementA_2							
ElementA_3						x	x
ElementA_4							
ElementA_5				x			x
ElementA_6							
ElementA_7				x			x
ElementA_8							

colonnes près, elles forment un rectangle plein qu'il n'est pas possible d'agrandir en rajoutant un objet ou un attribut du contexte. Ainsi d'après la définition l'ensemble des concepts du contexte correspond à l'ensemble des pavés maximaux.

Définition 4.3 (Treillis de concepts) *L'ensemble des concepts est un ensemble partiellement ordonné par la relation d'ordre*

$$(X_1, Y_1) \leq (X_2, Y_2) \Longleftrightarrow X_1 \subseteq X_2$$

Dans tout contexte $K = (O, A, I)$, tout couple de concepts possède un unique supremum (plus petit majorant). Il existe donc un supremum global, c'est-à-dire un concept (X, Y) supérieur à tous les autres, dans le cas où $X = O$. Tout couple de concepts possède aussi un infimum (plus grand minorant) unique. Il existe donc un infimum global, c'est-à-dire un concept (X, Y) inférieur à tous les autres, dans le cas où $Y = A$. Le fait que chaque couple de concepts ait une borne inférieure et une borne supérieure unique implique donc que le diagramme de Hasse représentant l'ensemble des concepts est un treillis, que l'on nomme treillis des concepts.

L'AFC est un processus consistant à générer à partir d'un contexte formel l'ensemble de tous ses concepts ordonnés sous forme d'un treillis. La représentation classique des treillis

de concepts affiche pour chaque concept la totalité de son extension et de son intension. On voit alors un ensemble d'informations redondantes qu'il est possible de simplifier pour faciliter la lecture des concepts sans diminuer la quantité d'information présente.

Soit un concept (X_1, Y_1). Pour tout concept (X_2, Y_2) tel que $(X_1, Y_1) \leq (X_2, Y_2)$ on a $X_1 \subseteq X_2$. Ainsi lorsqu'un objet appartient à un concept donné, il appartient aussi à tous les concepts supérieurs. L'extension simplifiée d'un concept ne possède que les objets qui n'appartiennent à aucun des concepts inférieurs, permettant ainsi d'alléger l'affichage d'un treillis de concepts et d'en améliorer sa lisibilité. Un raisonnement similaire peut être fait sur les intensions car lorsqu'un attribut appartient à un concept donné, il appartient aussi à tous les concepts inférieurs. On peut donc définir l'intension simplifiée d'un concept comme l'ensemble des attributs de l'intension qui n'apparaissent pas dans les concepts supérieurs.

Nous utilisons par la suite une représentation des treillis simplifiée en affichant pour chaque concept l'extension simplifiée et l'intension simplifiée. La figure 4.4 représente donc le treillis simplifié généré à partir du contexte de la table 4.1. Chaque boite représente un concept. Les boites sont partagées en trois compartiments : le premier est un identifiant du concept généré automatiquement par l'outil de génération de treillis ; le second contient l'intension simplifiée du concept et le troisième son extension simplifiée. Ainsi le concept 4 est défini par la paire $(\{ElementA_1\}, \{identification, assocChairman\})$ mais comme $identification$ appartient déjà au concept 6 qui est supérieur au concept 4 on affiche seulement la paire simplifiée $(\{ElementA_1\}, \{assocChairman\})$.

Dans la figure 4.4 le concept 0 correspond au concept dont l'extension contient tous les objets du contexte avec l'ensemble des attributs qu'ils partagent souvent appelé le concept *top*. Le concept 2 est le concept opposé dont l'extension contient tous les attributs avec l'ensemble des objets qui les partagent et généralement appelé *bottom*. Le concept 6 est le concept dont l'extension contient tous les *Member*, car ils partagent tous la caractéristique d'avoir un nom et sont ainsi regroupés dans un même concept. Ce concept est partagé en trois sous-concepts : le concept 3 dont l'extension contient les *normalMember*, le concept 4 dont l'extension contient le *chairman* et le concept 5 dont l'extension contient le *treasurer*.

4.2.2 Analyse Relationnelle de concepts

Nous avons présenté dans la section précédente une classification des éléments d'un modèle. Cette classification est représentée par le treillis de la figure 4.4. Nous verrons par la suite que notre approche se base sur une classification similaire des modèles, mais prenant en compte plus de paramètres. Il s'agit par exemple de prendre en compte pour un élément donné non seulement le type d'association dont il est source, mais aussi la description des éléments cible de cette association. Par description nous entendons non seulement le type de l'élément, mais aussi le concept auquel il appartient. C'est-à-dire que l'on souhaite par exemple exprimer le fait que l'*Association* « GreenFingers » contient par l'association *chairman* un *Member* contenant un *Name* par l'association *identifica-*

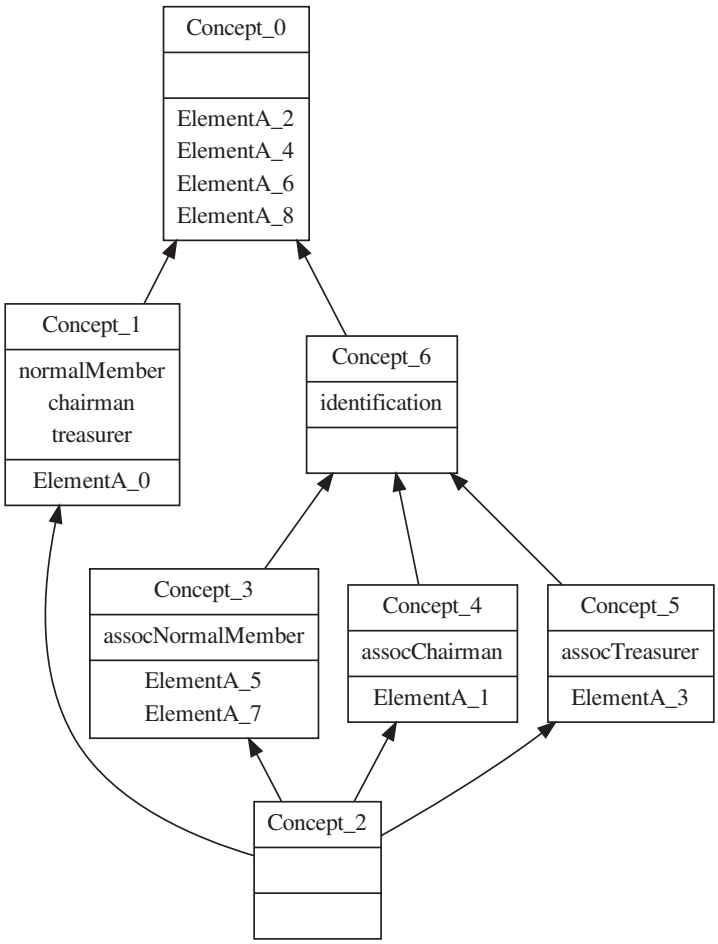


FIGURE 4.4 : Treillis résultant de l'application de l'AFC au contexte de la table 4.1.

tion. L'ARC nous permet de créer ce type de classification en étendant l'AFC : tout d'abord en prenant plusieurs contextes formels en paramètres, permettant la création de plusieurs treillis, mais surtout en introduisant les contextes relationnels. Ces derniers permettent de lier deux contextes formels, permettant au premier contexte formel de considérer les concepts du second contexte comme attributs. On nomme l'ensemble des contextes formels et relationnels une *famille de contextes relationnels.*

Définition 4.4 (Famille de Contextes Relationnels) *Une famille de contextes relationnels $\mathscr{R}$ est un couple (K,R). K est un ensemble de contextes formels $K_i = (O_i, A_i, I_i)$, R est un ensemble de contextes relationnels $R_j = (O_k, O_l, I_j)$ (O_k et O_l sont les ensembles d'objets des contextes K_k et K_l de K).*

Pour une classification plus complète de notre exemple nous créons deux contextes formels décrits par la table 4.2. Un contexte *metaModelA* décrivant les classes du méta-

TABLE 4.2 : Contextes formels pour l'application de l'ARC sur le modèle cible de l'exemple. Contexte décrivant les classes à gauche et contexte décrivant les éléments à droite.

metaModelA	(name,Association)	(name,Member)	(name,Name)
metaClassA_0	×		
metaClassA_1		×	
metaClassA_2			×

modelA
elementA_0
elementA_1
elementA_2
elementA_3
elementA_4
elementA_5
elementA_6
elementA_7
elementA_8

modèle et un contexte *modelA* décrivant les éléments du modèle. Le contexte *modelA* ne contient aucun attribut, car les caractéristiques de classification des éléments sont uniquement relationnelles. Les objets du contexte *metaModelA* ont pour attributs un identifiant pour chaque classe exprimé par l'attribut *name* de la classe (on considère que dans un méta-modèle il n'existe pas deux classes avec le même nom) pour faciliter la lecture des treillis. Les attributs *name* sont exprimés de la manière suivante : $(name, < nomdelaclasse >)$. Les attributs pris en compte pour les éléments du modèle sont représentés par les contextes relationnels de la table 4.3. Le premier contexte nommé *metaClassA* modélise la relation entre les éléments du modèle et les classes qu'ils instancient. C'est-à-dire que la paire $(ElementA_0, metaClassA_0)$ exprime le fait que « GreenFinger », identifié par *ElementA_0* est instance de *Association*, identifié par *metaClassA_0*. Les autres contextes modélisent les différentes associations qui relient les éléments du modèle. Par exemple le contexte *assocChairman* est la représentation de toutes les instances de l'association *assocChairman* qui relient un *Member* à l'*Association* à laquelle il appartient. Nous pouvons voir ici qu'il existe une telle association entre « JoeC » identifié par *ElementA_1* et« GreenFinger », identifié par *ElementA_0*. La différence avec des contextes formels est qu'ici les ensembles d'attributs sont des objets d'autres contextes formels. Rien n'empêche d'avoir le même ensemble comme objets et comme attributs : c'est le cas dans notre exemple pour tous les contextes relationnels à l'exception de *metaClassA*.

L'ARC permet l'émergence de nouvelles abstractions par itération d'une étape de construction des treillis de concepts suivie d'une étape de concaténation des contextes formels avec les contextes relationnels correspondant enrichis par les concepts créés à l'étape précédente.

Étape d'initialisation.

Les treillis sont construits à cette étape en utilisant l'AFC classique : pour chaque contexte formel K_i, un treillis $\mathscr{L}_i^0$ est créé.

L'étape d'initialisation de notre exemple est présentée à la figure 4.5. Il s'agit des treillis obtenus en appliquant l'AFC sur les contextes de la table 4.2. Le treillis obtenu à partir de

TABLE 4.3 : Contextes relationnels pour l'application de l'ARC sur le modèle cible de l'exemple.

metaClassA	metaClassA_0	metaClassA_1	metaClassA_2
elementA_0	×		
elementA_1		×	
elementA_2			×
elementA_3		×	
elementA_4			×
elementA_5		×	
elementA_6			×
elementA_7		×	
elementA_8			×

assocChairman	elementA_0	elementA_1	elementA_2	elementA_3	elementA_4	elementA_5	elementA_6	elementA_7	elementA_8
elementA_0									
elementA_1	×								
elementA_2									
elementA_3									
elementA_4									
elementA_5									
elementA_6									
elementA_7									
elementA_8									

assocTreasurer	elementA_0	elementA_1	elementA_2	elementA_3	elementA_4	elementA_5	elementA_6	elementA_7	elementA_8
elementA_0									
elementA_1									
elementA_2									
elementA_3	×								
elementA_4									
elementA_5									
elementA_6									
elementA_7									
elementA_8									

chairman	elementA_0	elementA_1	elementA_2	elementA_3	elementA_4	elementA_5	elementA_6	elementA_7	elementA_8
elementA_0		×							
elementA_1									
elementA_2									
elementA_3									
elementA_4									
elementA_5									
elementA_6									
elementA_7									
elementA_8									

identification	elementA_0	elementA_1	elementA_2	elementA_3	elementA_4	elementA_5	elementA_6	elementA_7	elementA_8
elementA_0									
elementA_1			×						
elementA_2									
elementA_3					×				
elementA_4									
elementA_5							×		
elementA_6									
elementA_7									×
elementA_8									

assocNormalMember	elementA_0	elementA_1	elementA_2	elementA_3	elementA_4	elementA_5	elementA_6	elementA_7	elementA_8
elementA_0									
elementA_1									
elementA_2									
elementA_3									
elementA_4									
elementA_5	×								
elementA_6									
elementA_7	×								
elementA_8									

treasurer	elementA_0	elementA_1	elementA_2	elementA_3	elementA_4	elementA_5	elementA_6	elementA_7	elementA_8
elementA_0				×					
elementA_1									
elementA_2									
elementA_3									
elementA_4									
elementA_5									
elementA_6									
elementA_7									
elementA_8									

normalMember	elementA_0	elementA_1	elementA_2	elementA_3	elementA_4	elementA_5	elementA_6	elementA_7	elementA_8
elementA_0						×		×	
elementA_1									
elementA_2									
elementA_3									
elementA_4									
elementA_5									
elementA_6									
elementA_7									
elementA_8									

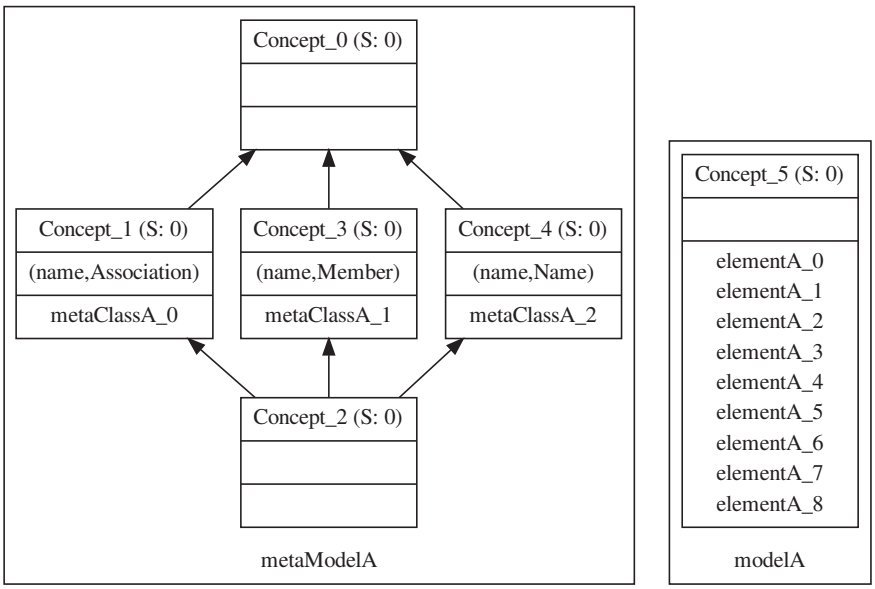


FIGURE 4.5 : Treillis obtenus à l'issue de l'étape d'initialisation.

metaModelA est un treillis où chaque objet appartient au top et à un concept où l'extension ne possède qu'un élément. Cela est la conséquence d'un contexte où chaque objet possède un seul attribut identifiant. Le treillis obtenu à partir de *modelA* contient un seul concept, car le contexte formel à l'origine de ce treillis ne contient aucun attribut.

Étape n+1.

Pour chaque contexte relationnel $R_j = (O_k, O_l, I_j)$, un contexte enrichi $R_j^{n+1} = (O_k, A, I_j^{n+1})$ est créé. A correspond aux concepts du treillis $\mathscr{L}_l^n$ (treillis sur les objets de O_l de l'étape n).

Soit $R_j(o) = \{o' \in O_l | (o, o') \in I_j\}$. Soient $o \in O_k$ et $a \in A$. La relation d'incidence I_j^{n+1} contient l'élément (o, a) si $S(R_j(o), Extension(a))$ est $vrai$.

La fonction S est un *opérateur de scaling* permettant de transformer la relation d'incidence entre deux ensembles d'objets en une relation d'incidence entre un ensemble d'objets et un ensemble de concepts. Deux opérateurs de scaling ont été introduits par [Huchard *et al.*, 2007b] :

- $S_\exists(R(o), Extension(a))$ est $vrai$ si et seulement si $\exists x \in R(o), x \in Extension(a)$
- $S_{\forall\exists}(R(o), Extension(a))$ est $vrai$ si et seulement si $\forall x \in R(o), x \in Extension(a) \wedge \exists x \in R(o), x \in Extension(a)$

L'opérateur que nous utiliserons principalement par la suite, sauf indication du contraire, et qui est habituellement utilisé est l'opérateur $S_\exists$, appelé *wide scaling operator* dans [Huchard *et al.*, 2007b].

TABLE 4.4 : Contexte *modelA* enrichi à l'étape 2.

	metaClassA					chairman	treasurer	normalMember
	c0	c1	c2	c3	c4	c5	c5	c5
ElementA_0	x	x				x	x	x
ElementA_1	x			x				
ElementA_2	x				x			
ElementA_3	x			x				
ElementA_4	x				x			
ElementA_5	x			x				
ElementA_6	x				x			
ElementA_7	x			x				
ElementA_8	x				x			

	identification	assocChairman	assocTreasurer	assocNormalMember
	c5	c5	c5	c5
ElementA_0				
ElementA_1	x	x		
ElementA_2				
ElementA_3	x		x	
ElementA_4				
ElementA_5	x			x
ElementA_6				
ElementA_7	x			x
ElementA_8				

L'application de l'AFC sur $K_k \cup \{R_j^{n+1} = (O_k, A, I)\}$ crée de nouveaux concepts qui sont ajoutés à $\mathscr{L}_k^n$ pour obtenir $\mathscr{L}_k^{n+1}$. Le processus s'arrête quand les treillis d'une étape n sont équivalents (aux étiquettes près) à ceux de l'étape $n-1$: en effet, si tous les concepts déjà créés sont pris en compte et qu'aucun nouveau concept n'est créé, alors il n'y aura plus d'enrichissement possible. Le nombre de concepts pour chaque concept formel $K(O, A, I)$ étant borné par l'ensemble des parties de O, il est garanti que le processus se termine.

Le contexte présenté à la table 4.4 est le contexte formel *modelA* de la table 4.2 enrichi à partir des contextes relationnels de la table 4.3 et des treillis de l'étape d'initialisation de la figure 4.5. Nous considérons que cette étape, qui suit l'étape d'initialisation est l'étape 2. Le contexte formel *metamodelA* ne peut pas être enrichi, car son ensemble d'objets n'est source d'aucun contexte relationnel, il restera donc identique durant tout le processus ainsi que son treillis associé. À cause de problèmes de taille, le contexte est présenté en deux parties, mais il s'agit bien d'un seul contexte. En appliquant l'AFC sur ce contexte on obtient le treillis présenté à la figure 4.6. On notera qu'à ce stade du processus, le treillis contient une classification dépendant de la classe des éléments et des associations dont ils

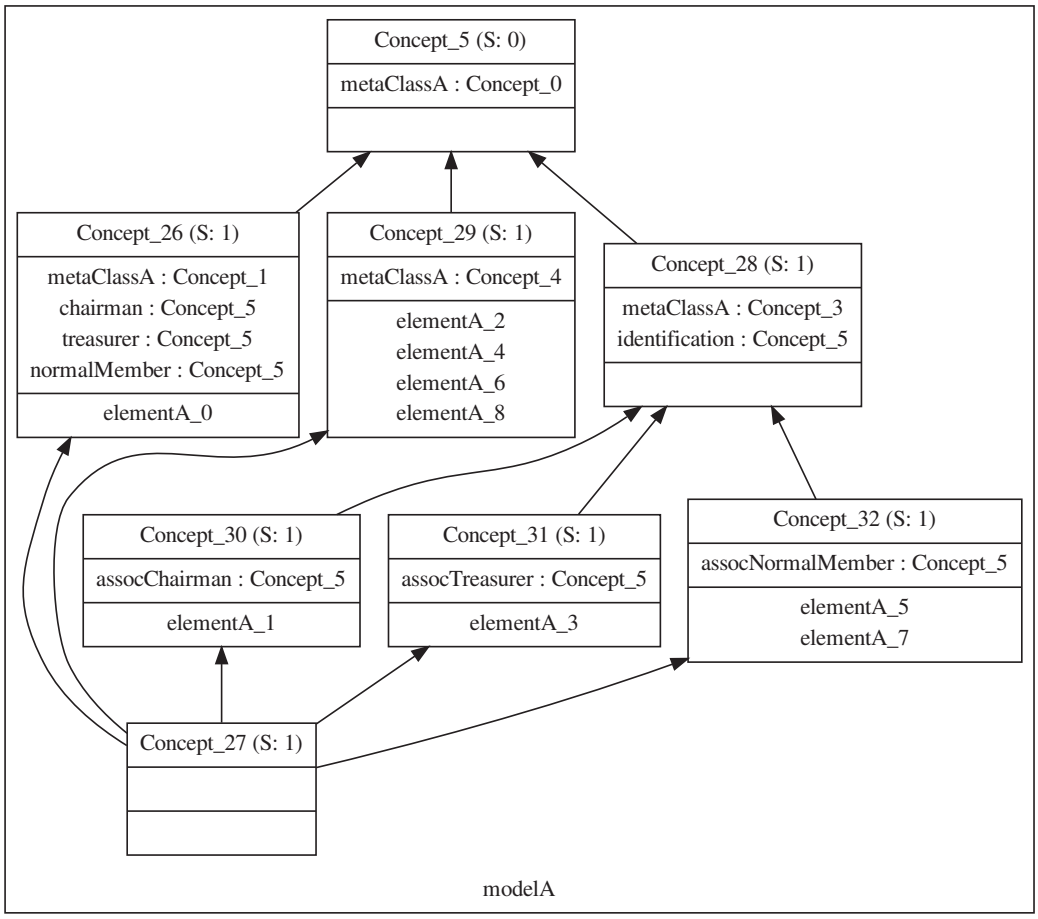


FIGURE 4.6 : Treillis du contexte *modelA* à l'étape 2.

TABLE 4.5 : Contexte *modelA* enrichi à l'étape 3.

	metaClassA					chairman			treasurer			normalMember		
	c0	c1	c2	c3	c4	c5	c28	c30	c5	c28	c31	c5	c28	c32
ElementA_0	x	x				x	x	x	x	x	x	x	x	x
ElementA_1	x			x										
ElementA_2	x				x									
ElementA_3	x			x										
ElementA_4	x				x									
ElementA_5	x			x										
ElementA_6	x				x									
ElementA_7	x			x										
ElementA_8	x				x									

	identification		assocChairman		assocTreasurer		assocNormalMember	
	c5	c29	c5	c26	c5	c26	c5	c26
ElementA_0								
ElementA_1	x	x	x	x				
ElementA_2								
ElementA_3	x	x			x	x		
ElementA_4								
ElementA_5	x	x					x	x
ElementA_6								
ElementA_7	x	x					x	x
ElementA_8								

sont source. Ces dernières caractéristiques sont exactement celles utilisées pour obtenir le treillis de la figure 4.4. Ainsi le *Concept_28* représente l'ensemble des *Member*. Cela s'exprime par l'attribut relationnel *metaClassA : Concept_3* qui signifie que tous les éléments de l'extension du concept sont instances des éléments du *Concept_3* (voir figure 4.5), or le *Concept_3* représente la classe *Member*. L'attribut *identification : Concept_5* indique que tous ces éléments contiennent par l'association *identification* des éléments du *Concept_5* c'est-à-dire le concept dont l'extension contient tous les éléments du modèle. À ce stade du processus il n'est pas encore possible de voir les spécificités des éléments contenus par l'association *identification*.

Les concepts obtenus nous permettent d'enrichir à nouveau le contexte formel *modelA* pour obtenir la table 4.5 de l'étape 3 du processus. Pour chaque contexte relationnel, nous prenons l'ensemble des concepts du treillis du contexte formel cible, mais pour des problèmes de place les colonnes vides du contexte formel n'ont pas été reproduites. Le treillis obtenu à partir de ce contexte est représenté à la figure 4.7. Les intensions des concepts ont pour la plupart été enrichies, ainsi le concept 28 représente l'ensemble des *Member*

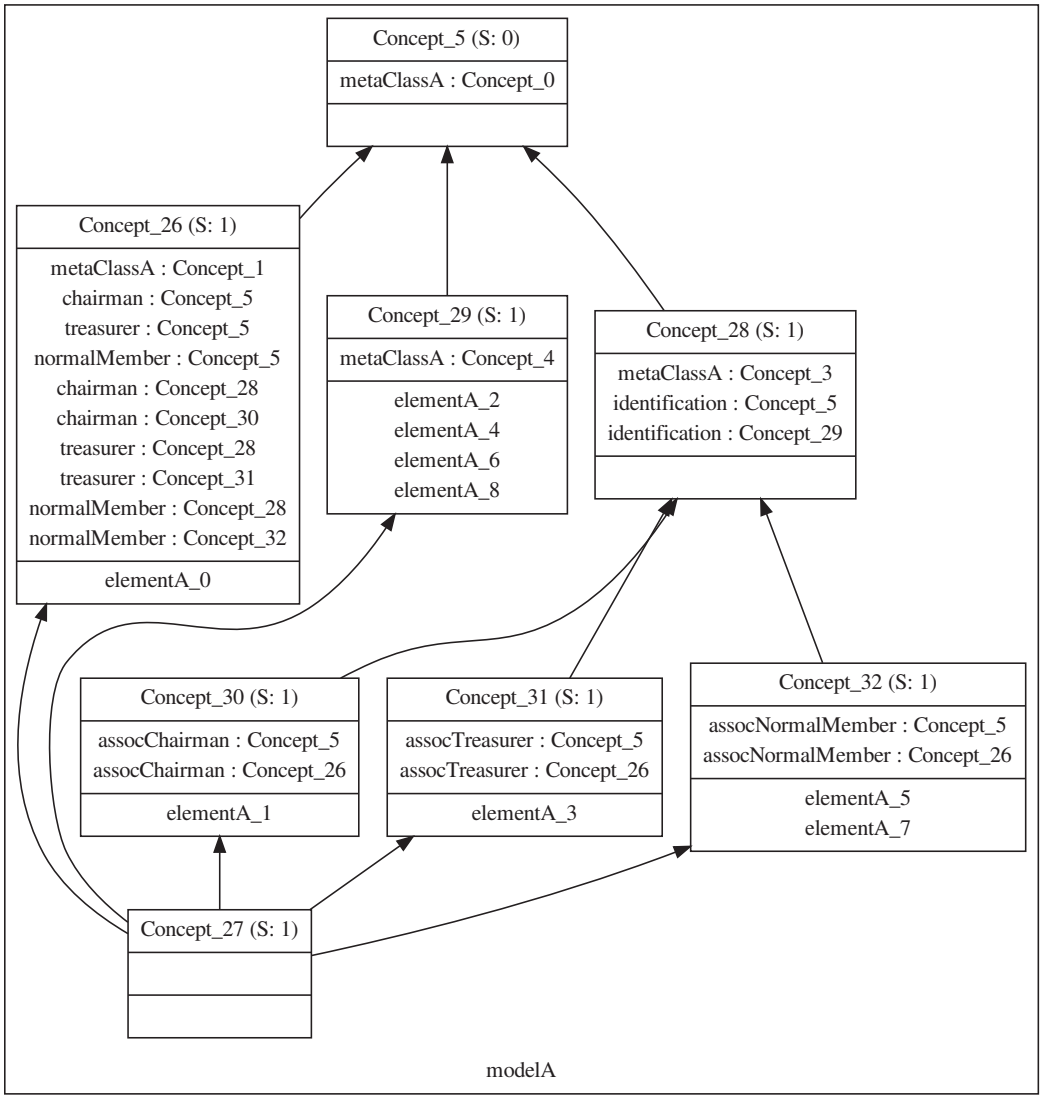


FIGURE 4.7 : Treillis du contexte *modelA* à l'étape finale.

(concept 3) sources d'au moins une association *identification* dont la cible est représentée par le concept 29 qui est l'ensemble des éléments de classe *Name*. Nous noterons qu'aucun concept n'a été introduit dans le processus, ainsi le contexte formel *modelA* ne peut plus être enrichi. Il s'agit donc du treillis final.

4.3 Spécification de l'approche de génération de transformation

Ce chapitre présente une approche de génération de définitions de transformations, c'est-à-dire d'un ensemble de règles de transformations permettant de transformer un modèle décrit dans un langage source en un autre modèle décrit dans un langage cible. Cette génération se fait à partir d'exemples, c'est-à-dire un modèle décrit dans un langage source et un autre modèle qui est le résultat de la transformation à développer.

Les données d'entrée sont un modèle d'exemple source, le modèle cible correspondant, et un alignement entre les deux modèles tel que présenté dans le chapitre 3. Nous prendrons pour convention par la suite de rajouter le suffixe *A* pour tout élément faisant référence au modèle source et le suffixe *B* pour les références au modèle cible.

L'idée générale de l'approche est de créer des regroupements de liens d'appariement de l'alignement. Ces regroupements se font à partir des caractéristiques communes des éléments sources et cibles. Ainsi nous obtenons pour un ensemble de liens d'appariement une correspondance entre des caractéristiques dans le langage source et les caractéristiques équivalentes dans le langage cible. Nous présenterons dans la section 4.3.1 comment créer la famille de contextes relationnels nécessaire pour classifier les éléments des modèles en utilisant l'ARC et nous verrons dans la section 4.3.2 comment étendre cette famille pour obtenir une classification des liens d'appariement. Nous montrerons alors dans la section 4.3.3 comment passer de cette classification sous forme de treillis à des règles de transformation.

4.3.1 Classification des éléments du modèle

La classification des éléments du modèle a déjà été évoquée sur un exemple dans la section 4.2.2. Nous avons vu alors que l'ARC nous permettait de classer des objets en fonction de différentes caractéristiques. Nous allons voir quelles sont les caractéristiques à extraire de manière automatique pour le problème particulier de la génération de transformations. Ces caractéristiques portent essentiellement sur le voisinage et le type de chaque élément.

Les modèles sources et cibles de l'exemple sont traités de manière identique. Le principal ensemble d'objets que nous cherchons à classifier est l'ensemble des éléments du modèle. Le contexte formel principal que nous considérons a donc pour ensemble d'objets l'ensemble des éléments du modèle : nous nommerons ce contexte *model*. Le contexte *modelA* de notre exemple illustratif est présenté à la table 4.2 et le contexte *modelB* est pré-

TABLE 4.6 : Contextes formels du modèle cible pour l'application de l'ARC sur l'exemple. Contexte décrivant les classes à gauche et contexte décrivant les éléments à droite.

metaModelB	(name,Persons)	(name,Person)	(name,ResponsiblePerson)	(name,PersonWithoutResponsibility)	(name,Name)
metaClassB_0	×				
metaClassB_1		×			
metaClassB_2			×		
metaClassB_3				×	
metaClassB_4					×

modelB
elementB_0
elementB_1
elementB_2
elementB_3
elementB_4
elementB_5
elementB_6
elementB_7
elementB_8

senté à la table 4.6. Les deux contextes ne possèdent pas d'attributs, la classification se fait uniquement sur les relations avec d'autres objets.

La première caractéristique de classification sur laquelle se base notre approche est la classe dont les éléments sont instances. Il s'agit donc de traduire la relation d'instanciation d'une classe du métamodèle par un élément du modèle sous forme de contexte relationnel. Pour cela nous considérons un second contexte formel, que nous nommerons *metamodel*, ayant pour objets l'ensemble des classes définies dans le méta-modèle. Il est important que chaque classe soit représentée par un concept différent, chaque classe possède donc comme attribut un identifiant unique. La relation d'instanciation est ensuite représentée par un contexte relationnel nommé *metaClass* prenant pour source le contexte *model* et pour cible le contexte *metamodel*. Les contextes *metamodelA* et *metamodelB* de notre exemple sont présentés aux tables 4.2 et 4.6 et les contextes relationnels *metaClassA* et *metaClassB* sont présentés aux tables 4.3 et 4.7.

La seconde caractéristique de classification pour un élément est l'ensemble des caractéristiques de ses voisins. Par exemple dans le modèle source de la figure 4.3, les *Member* sont différenciés par l'association qui les relie à l'*Association* (*assocNormalMember*, *assocTreasurer* ou *assocChairman*) plutôt que leur classe. Ainsi pour établir une règle de transformation qui transformerait un *Member* en *Person without Responsability* il est nécessaire de vérifier que le *Member* est associé par *normalMember* à une *Association*. Pour que ces informations puissent apparaître dans la classification, nous allons utiliser des contextes relationnels, un pour chaque association dans le modèle, dont la relation d'incidence va du contexte formel *model* vers lui-même. Ainsi les contextes relationnels présentés par la

TABLE 4.7 : Contextes relationnels du modèle cible pour l'application de l'ARC sur l'exemple.

persons	elementB_0	elementB_1	elementB_2	elementB_3	elementB_4	elementB_5	elementB_6	elementB_7	elementB_8
elementB_0		×		×		×		×	
elementB_1									
elementB_2									
elementB_3									
elementB_4									
elementB_5									
elementB_6									
elementB_7									
elementB_8									

metaClassB	metaClassB_0	metaClassB_1	metaClassB_2	metaClassB_3	metaClassB_4
elementB_0	×				
elementB_1			×		
elementB_2					×
elementB_3			×		
elementB_4					×
elementB_5				×	
elementB_6					×
elementB_7				×	
elementB_8					×

name	elementB_0	elementB_1	elementB_2	elementB_3	elementB_4	elementB_5	elementB_6	elementB_7	elementB_8
elementB_0									
elementB_1			×						
elementB_2									
elementB_3					×				
elementB_4									
elementB_5							×		
elementB_6									
elementB_7									×
elementB_8									

table 4.3, à l'exception du contexte *metaClassA*, sont la traduction des différentes associations du modèle d'exemple.

Le résultat de tout cela est un treillis issu du contexte formel *model* où les éléments du modèle sont regroupés par type et par caractéristiques du voisinage. Le treillis de la figure 4.7 est le treillis classifiant tous les éléments du modèle source de l'exemple illustratif. Chaque concept peut être vu comme un motif décrivant le voisinage autour d'un type d'élément. Ces motifs peuvent éventuellement être enrichis en modifiant les opérateurs de scaling (voir section 4.2.2). Les treillis issus des contextes du modèle cible sont présentés aux figures 4.8 et 4.9.

4.3.2 Classification des liens d'appariements

Dans la section précédente, nous avons créé un ensemble de motifs dans les modèles source et cible à partir d'exemples. Une règle de transformation peut être considérée comme l'association entre un motif source et un motif cible. Il nous reste donc à

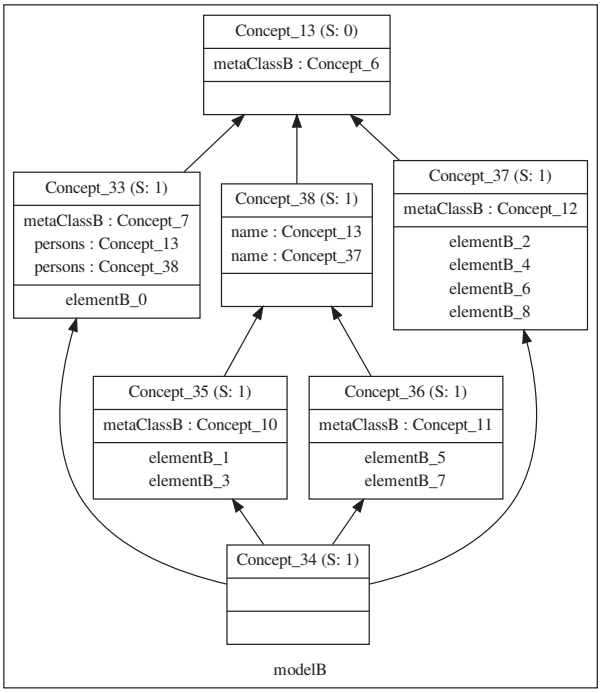


FIGURE 4.8 : Treillis issus du contexte modelB de l'exemple illustratif.

« apprendre » quels motifs sont à associer. Pour cela nous allons utiliser les liens d'appariement. Chaque lien relie un élément du modèle source à un élément du modèle cible. Il s'agit maintenant de classifier les liens en fonction des caractéristiques des éléments sources et cibles. Cette classification passe tout d'abord par la création d'un contexte formel qui se rajoute à la famille de contextes définie précédemment ; nous le nommerons *MapLinks*. Ce contexte a pour ensemble d'objets l'ensemble des liens d'appariement de l'alignement. Le contexte ne possède pas d'attributs, car la classification ne prend en compte que les relations entre les liens et les éléments sources ou cibles. Il est présenté à la table 4.8. À ce contexte s'ajoutent deux contextes relationnels, le premier modélise la relation entre les objets de *MapLinks* et les éléments du modèle source. Ce contexte sera nommé *MapLinkA*. Le second contexte prend en compte la relation entre les objets de *MapLinks* et les éléments du modèle cible et sera nommé *MapLinkB*. Ces deux contextes relationnels sont représentés par la table 4.9. Le treillis du contexte *MapLinks*, construit à partir de la famille de contexte contenant les contextes représentés par les tables 4.2, 4.3, 4.6, 4.7, 4.8 et 4.9, est présenté à la figure 4.10.

Les concepts de ce treillis sont formés par des ensembles de liens d'appariement dont

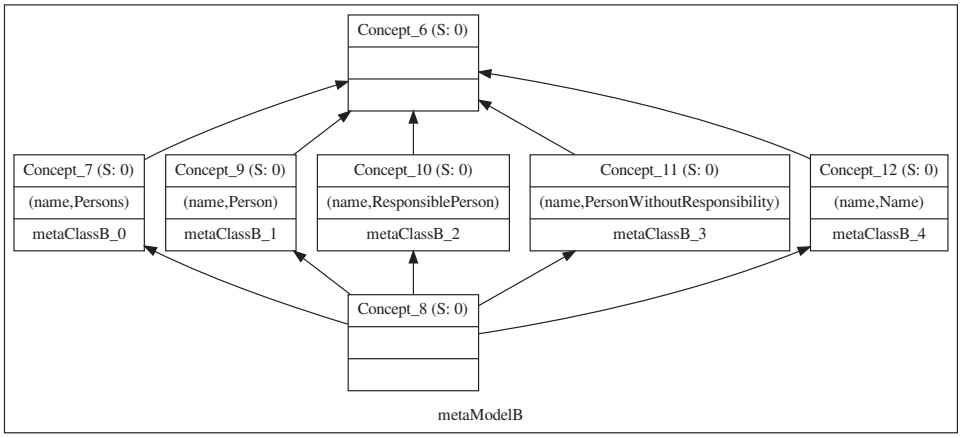


FIGURE 4.9 : Treillis issus du contexte metaModelB de l'exemple illustratif.

TABLE 4.8 : Contexte formel regroupant les liens d'appariement de l'exemple illustratif.

MapLinks
MapLink_0
MapLink_1
MapLink_2
MapLink_3
MapLink_4
MapLink_5
MapLink_6
MapLink_7
MapLink_8

les extrémités partagent des caractéristiques communes. De manière informelle, les extrémités des liens des concepts les plus hauts dans le treillis partagent des caractéristiques « générales » (*i.e.* ces extrémités appartiennent à un concept haut dans le treillis des éléments du modèle) tandis que plus on descend dans le treillis plus ces caractéristiques sont « détaillées ». Par exemple le concept 47 associe les éléments de type *Member* (concept 28) du modèle source aux éléments de type *Responsible Person* (concept 35) du modèle cible. Cette règle générale est spécialisée par le concept 44 qui associe les éléments de type *Member* avec une association *assocChairman* (concept 30) aux éléments de type *Responsible Person* et par le concept 45 qui associe les éléments de type *Member* avec une association *assocTreasurer* (concept 31) aux éléments de type *Responsible Person*.

TABLE 4.9 : Contextes relationnels représentant la relation entre les liens d'appariements et les éléments source et cibles qu'ils relient.

MapLinkA	elementA_0	elementA_1	elementA_2	elementA_3	elementA_4	elementA_5	elementA_6	elementA_7	elementA_8
MapLink_0	×								
MapLink_1		×							
MapLink_2			×						
MapLink_3				×					
MapLink_4					×				
MapLink_5						×			
MapLink_6							×		
MapLink_7								×	
MapLink_8									×

MapLinkB	elementB_0	elementB_1	elementB_2	elementB_3	elementB_4	elementB_5	elementB_6	elementB_7	elementB_8
MapLink_0	×								
MapLink_1		×							
MapLink_2			×						
MapLink_3				×					
MapLink_4					×				
MapLink_5						×			
MapLink_6							×		
MapLink_7								×	
MapLink_8									×

4.3.3 Interprétation des treillis et création de règles

La famille de contextes définie précédemment nous permet d'obtenir cinq treillis reliés entre eux :

- *metaModelA* : une classification du méta-modèle source ;
- *modelA* : une classification du modèle source ;
- *metaModelA* : une classification du méta-modèle cible ;
- *modelA* : une classification du modèle cible ;
- *MapLinks* : une classification des liens d'appariement.

Les concepts du treillis *MapLinks* regroupent des liens d'appariement ayant des sources ou des cibles possédant des caractéristiques communes. Vu que chaque lien ne possède qu'une seule source et une seule cible, on peut considérer les concepts du treillis comme des règles de transformation : en effet chaque concept aura dans ses caractéristiques au moins une référence vers un concept de *modelA* et au moins une référence vers un concept de *modelB*. Malgré tout, nous considérons qu'il est nécessaire pour créer une règle que les concepts référencés aient un type comme caractéristique, car dans la plupart des approches de transformation la condition minimale pour l'application d'une transformation est une condition de typage, et la création d'un nouvel élément dans le modèle cible nécessite aussi d'en connaître son type. Ainsi dans le treillis de la figure 4.10 le concept 14 ne sera pas retenu comme une règle potentielle car les concepts 5 (voir figure 4.7) et 13 (voir figure 4.8) sont des concepts dont la référence de type *metaClass* dans l'intension pointe vers un concept ne représentant pas une classe du méta-modèle : les concepts 0 et 6 sont les concepts top des treillis *metaModelA* (figure 4.5) et *metaModelB* (figure 4.9).

De la même manière le concept 42 ne sera pas retenu comme une règle potentielle car le concept 38 (voir figure 4.8) n'introduit pas de nouvelle référence de type *metaClass* par rapport au concept 13 vu précédemment.

Pour faciliter la lecture nous utiliserons pas la suite la notation *Treillis.c* pour faire référence au concept *c* du treillis *Treillis*.

À partir d'un concept *Maplink.c* représentant une règle, les propriétés requises pour la prémisse (resp. la conclusion) de la règle sont obtenues par l'analyse du concept *modelA.c'* (resp. *modelB.c'*), qui correspond au concept le plus spécialisé de *modelA* (resp. *modelB*) appartenant à l'intension de *Maplink.c*. Nous définissons trois catégories différentes de caractéristiques :

- *Les caractéristiques nécessaires* : elles sont décrites par l'intension de *modelA.c'* (resp. *modelB.c'*). Ce sont les caractéristiques qui sont communes à tous les liens d'appariement de l'extension de *Maplink.c*.
- *Les caractéristiques autorisées* : elles sont décrites par l'intension des concepts spécialisant *modelA.c'* (resp. *modelB.c'*), à condition que l'extension de ce concept contienne une extrémité d'au moins un lien de *Maplink.c*. Ce sont des caractéristiques qui n'apparaissent pas pour tous les liens d'appariement du concept, nous les considérons alors comme optionnelles.
- *Les caractéristiques interdites* : elles sont décrites par les intensions des concepts n'incluant pas dans leur extension une extrémité d'un lien de *Maplink.c*. Ces caractéristiques sont particulièrement importantes si elles appartiennent à des concepts spécialisant *modelA.c'* (resp. *modelB.c'*) car elles sont alors discriminantes pour le choix d'une règle à appliquer.

Les définitions de ces caractéristiques sont basées sur une hypothèse de monde clos, or le contexte de ce travail est un monde ouvert, il est donc important que les exemples utilisés soient suffisamment significatifs et couvrent un maximum de possibilités pour que le résultat puisse être considéré comme correct.

L'interprétation des treillis par l'utilisateur n'est pas chose aisée, en effet l'interprétation d'un concept du treillis *Maplink* nécessite de naviguer dans les cinq treillis et le nommage automatique des noms de concepts ou des identifiants pour les objets ne facilite pas la tâche. Nous proposons donc une représentation des règles dans un format plus adapté et créé spécialement à cet effet. L'idée est de garder la structure du treillis *Maplink*, car le treillis permet une navigation parmi les règles et d'intégrer l'interprétation des concepts des treillis *modelA* et *modelB* pour que toutes les informations nécessaires apparaissent dans un seul treillis.

Le méta-modèle de cette structure de données est présenté à la figure 4.12. Chaque règle est composée d'une prémisse et d'une conclusion. Une prémisse ou une conclusion sont des motifs.

La figure 4.11 présente une vue concrète de l'interprétation du treillis *Maplink* de notre exemple (voir figure 4.10). Les différents types de caractéristiques sont renseignés à l'aide de cardinalités : [1..*] pour les caractéristiques obligatoires ; [0..*] pour les caractéristiques

autorisées et [0..0] pour les caractéristiques interdites. Les caractéristiques autorisées et interdites ne sont ici représentées que pour les éléments sources des motifs. Plus on s'éloigne de la source dans un motif et moins ces caractéristiques ont de sens.

4.3.4 Gestion des liens dans le modèle cible

Nous nous sommes concentrés tout d'abord dans notre approche sur la création des bons types d'éléments dans le modèle cible par les transformations générées. Il est ensuite nécessaire d'avoir les informations permettant de relier entre eux les éléments du modèle cible par les bonnes relations. Nous présentons ici une approche permettant de déduire les différentes relations pour relier les éléments en utilisant le processus ARC et la famille de contextes définie précédemment. Cette approche n'est à l'heure actuelle pas implémentée dans l'outil que nous présentons par la suite.

Nous utilisons une approche pour les liens similaire à celle utilisée pour les éléments. Nous créons tout d'abord un contexte formel *AssocFC* contenant toutes les relations des modèles et où les relations sont classées en fonction de leur type. Nous identifions ensuite dans un premier contexte relationnel nommé *AssocFromRC*, pour chaque relation, les liens d'appariement dont la source de la relation est une extrémité et dans un second contexte relationnel nommé *AssocToRC*, toujours pour chaque relation, les liens d'appariement dont la cible de la relation est une extrémité. Par exemple dans la figure 4.3, pour la relation *chairman* à laquelle on donne l'identifiant *Assoc_0* entre « GreenFingers » et « JoeC » on trouvera dans la relation d'incidence de *AssocFromRC* le couple ($Assoc_0, MapLink_0$) et dans la relation d'incidence de *AssocToRC* le couple ($Assoc_0, MapLink_1$).

Lorsqu'on applique l'ARC sur la famille de contextes augmenté des contextes décrits précédemment, on obtient un nouveau treillis dans lequel les concepts contiendront des règles et des types de relations dans l'intension et des associations dans l'extension. Ainsi pour un concept donné, si on a un type de relation provenant du modèle source *(modelA,r1)*, un type de relation provenant du modèle cible *(modelA,r2)*, un concept du treillis *Maplink* par le contexte relationnel *AssocFromRC* nommé *ML1* et un concept du treillis *Maplink* par le contexte relationnel *AssocToRC* nommé *ML2* alors on se retrouve dans la configuration illustrée par la figure 4.13. Nous pouvons en déduire pour la définition de la règle associée au concept *ML1* qui associe un type *x1* à un type *y1* que l'élément *y2* voisin de *y1* par la relation *(modelA,r2)* est le résultat de la transformation de *x2*, voisin de *x1* par la relation *(modelA,r1)*.

L'approche présentée ici permet de résoudre le cas où le voisin d'un élément du modèle cible est le résultat d'une transformation d'un voisin d'un élément du modèle source. Cela correspond à une partie assez importante des cas lorsque le modèle transformé garde une structure similaire du modèle source, mais n'est pas satisfaisant pour des cas plus complexes et ce problème nécessite donc la création de nouvelles techniques, que nous explorerons dans nos travaux futurs.

4.4 Étude de cas

4.4.1 Outil

L'approche présentée ici a été implémentée en utilisant les outils de modélisation proposés par Eclipse. Eclipse est un environnement de développement à l'origine développé pour JAVA. Cet environnement est extrêmement modulable et propose un framework de modélisation basé sur le méta-méta-modèle Ecore qui s'inspire fortement de EMOF défini par l'OMG.

Nous utilisons dans notre outil le plugin eRCA [eRCA] qui est une implémentation de l'ARC utilisant le framework EMF pour définir les structures de données nécessaires à l'Analyse Relationnelle de Concepts. De cette manière il nous reste à définir la transformation qui permet de transformer les exemples en une famille de contextes relationnels et la transformation qui interprète les treillis en règles de transformation. L'outil n'intègre pas encore la création de liens entre les éléments du modèle cible. Nous avons aussi implémenté une troisième étape qui consiste à traduire les modèles de règles en code dans un langage textuel grâce au langage de transformation modèle-vers-texte Acceleo. Cette troisième étape nous permet de générer du code dans le langage ATL ([Bézivin *et al.*, 2003]) qui, par son côté déclaratif, s'avère proche de l'expressivité des règles générées et nous apparaît donc comme le langage le plus approprié.

4.4.2 Résultats

La validation d'une telle approche par expérimentation pose de nombreux problèmes car bien que s'appuyant sur les treillis de concepts, qui sont des outils mathématiques à la sémantique bien définie, la pertinence du résultat reste à l'appréciation de l'utilisateur. En effet notre approche se veut être une aide à la définition de transformations de modèles et ne prétend en aucun cas générer une transformation complète et utilisable sans modifications. Il est difficile de définir si le résultat obtenu, n'étant pas une transformation parfaite, reste malgré tout un artefact utile au développement d'une définition de la transformation.

Les règles de transformation que nous générons se présentent sous forme d'un treillis dont certains concepts représentent une règle. Ces règles sont représentées comme l'association d'un motif source avec un motif cible.

Nous avons donc représenté dans les tableaux 4.10 et 4.11 un ensemble de métriques quantitatives sur les treillis générés par notre outil dans deux configurations différentes : dans les deux cas la création de propriétés autorisées est calculée pour le premier élément des motifs, la génération de propriétés interdites est désactivée, car incompatible avec une des deux configurations. Dans un cas l'opérateur de scaling $S_{\forall\exists}$ est utilisé sur les contextes relationnels décrivant les associations reliant les éléments des modèles sources et cibles, tandis que dans l'autre cas l'opérateur de scaling utilisé reste l'opérateur $S_{\exists}$ pour tous les

contextes relationnels. Nous appellerons ces deux processus respectivement les processus $Proc_{\forall\exists}$ et $Proc_{\exists}$.

Les métriques présentées sont les suivantes :

- taille du treillis *MapLinks* : les treillis des règles est une interprétation du treillis *MapLinks*. Chacun de ses concepts est conservé dans le treillis des règles. La taille du treillis donne donc un ordre de grandeur sur la quantité de données générée par le processus
- nombre de règles extraites : seulement une partie des concepts du treillis est interprétable sous forme de règles. Cette valeur correspond au nombre de règles générées.
- taille de prémisse : Le motif source d'une règle est considéré comme la prémisse d'une règle, c'est-à-dire la condition nécessaire à l'exécution de la règle. Les motifs se présentent sous forme de graphes enracinés où les nœuds et les arêtes sont étiquetés. La taille d'un motif correspond au nombre de nœuds dans un motif.
- profondeur de prémisse : Nous appelons profondeur la distance du nœud le plus éloigné de la racine.
- taille de conclusion : Le motif cible d'une règle est appelé la conclusion de la règle. Cette métrique est similaire à la taille de prémisse
- profondeur de conclusion : Le motif cible d'une règle est appelé la conclusion de la règle. Cette métrique est similaire à la profondeur de prémisse

Nous présentons pour les tailles et profondeurs de motifs les différentes variations que peuvent avoir celles-ci dans les treillis de règles en calculant les valeurs minimales, maximales, moyennes et les écarts type.

La taille des treillis pour le processus $Proc_{\forall\exists}$ reste dans l'ensemble raisonnable, à l'exception de la transformation *UML2ClassDiagram_to_KM3* qui génère 104 concepts. Alors que l'opérateur $S_{\forall\exists}$ est plus restrictif que l'opérateur $S_{\exists}$ la taille des treillis n'augmente pas forcément pour le processus $Proc_{\exists}$. Elle augmente malgré tout de manière significative pour *UML2ClassDiagram_to_KM3*, passant de 104 à 200 concepts avec $Proc_{\exists}$. Il est malgré tout intéressant de noter que le nombre de règles extraites n'augmente pas forcément avec la taille des treillis. Ainsi pour le processus $Proc_{\forall\exists}$ le nombre de règles extraites pour la transformation *UML2ClassDiagram_to_KM3* est de 29, très proche de la transformation *uml-er3* à partir de laquelle le processus génère 28 règles alors que le treillis contient 53 concepts. Le nombre de règles extraites reste inférieur ou égal à 29 pour le processus $Proc_{\forall\exists}$, ce qui peut être considéré comme un nombre de règles à étudier raisonnable. Pour le processus $Proc_{\exists}$le nombre maximal de règles monte à 40 et peut alors être considéré comme limite, mais il faut toutefois noter qu'en dehors de la transformation à 40 règles, toutes les autres transformations génèrent moins de 30 règles. Le nombre de règles est fortement dépendant des choix effectués lors de la création des modèles d'exemple, et un certain nombre d'entre eux n'ont pas été développés dans le but d'être utilisés par un processus de génération par l'exemple. La taille des premisses, comme celle des conclusions est parfois très importante. Un motif de plus de 20 nœuds devient difficile à interpréter. Le facteur responsable de la taille des prémisses est bien souvent la taille des mo-

dèles d'exemple et le nombre de relations pour chaque élément. La profondeur du motif est aussi un frein à la facilité d'interprétation des motifs, alors que bien souvent les éléments les plus importants se trouvent à une profondeur inférieure à 3. Il serait possible de modifier la profondeur des motifs et par la même occasion leur taille en limitant le nombre d'itérations dans le processus ARC, au prix d'une perte de détails dans la règle.

Nous allons aborder dans le reste de cette section une évaluation qualitative sur un exemple de transformation exogène. L'exemple exogène que nous présentons est l'exemple nommé *uml-er-iccs*. La transformation en question est une transformation d'un diagramme de classes similaire aux diagrammes de classes UML (nous les appellerons donc par la suite des modèles UML) vers un modèle Entité-Association. Les méta-modèles et prototypes de modèles utilisés sont présentés à la figure 4.15. De manière informelle les différentes règles de transformations nécessaires à la définition de la transformation sont les suivantes :

1 – Un élément de type *RootUML* contenant des *Association* et des *Class* sera transformé en un élément de type *RootER* contenant des *RelationShip* et des *Entity*. *RootUML* et *RootER* sont respectivement les éléments racine des modèles UML et des modèles Entité-Association.

2 – Un élément de type *Class* se transforme en un élément de type *Entity*

3 – Un élément de type *Association* se transforme en un élément de type *Relationship*

4 – Un élément de type *Property* associé à une *Class* par l'association *owningClass* mais n'étant pas extrémité d'une *Association* se transforme en élément de type *Attribute*

5 – Un élément de type *Property* étant une extrémité d'une *Association* se transforme en *Role*

6 – Un élément de type *Property* étant une extrémité d'une *Association* se transforme en *Cardinality*

Nous cherchons donc à obtenir 6 règles. Les tables 4.10 et 4.11 montrent que le processus $Proc_{\forall\exists}$ produit 10 règles et le processus $Proc_{\exists}$ 12 règles.

Les trois premières règles sont des règles simples et le processus $Proc_{\forall\exists}$ génère une règle pour chacune d'elles tandis que le processus $Proc_{\exists}$ en génère deux pour la règle 2 comme le montre l'extrait du treillis des règles de la figure 4.16. La génération de ces deux règles s'explique par le fait que dans l'exemple utilisé comme base à la génération des règles la *Class Account* possède par l'association *ownedAttribute* deux types de *Property* différentes : la *Property owner* qui est extrémité d'une *Association* et la *Property number* qui ne l'est pas. Tandis que la *Class Client* ne possède que la *Property name* qui n'est pas reliée à une *Association*. Cette différence s'exprime durant le processus $Proc_{\exists}$ sous la forme de deux règles reliées par un lien de spécialisation entre les deux. Dans un tel cas, la règle la plus générale est la plus indiquée. Le processus $Proc_{\forall\exists}$ quant à lui ne prend en compte que les aspects communs à toutes les instances d'un type d'association, c'est-à-dire pour la *Class Account* seulement les caractéristiques communes de tous les éléments auxquels

TABLE 4.10 : Métriques quantitatives sur les treillis des règles obtenus à partir de notre ensemble d'exemples avec une configuration utilisant l'opérateur $S_{\forall\exists}$ (processus $Proc_{\forall\exists}$)

	taille du	nombre de	taille premisse				profondeur premisse			
nom	treillis MapLinks	règles extraites	min	max	moy.	écart type	min	max	moy.	écart type
associations-persons	9	6	1	5	4	1.53	1	3	2	0.91
Book2Publication	4	2	2	2	2	0	2	2	2	0
delegation1	47	14	6	45	23	11.6	4	8	6	1.13
delegation2	11	6	8	16	9	3	3	5	4	0.82
Disaggregation	22	10	3	8	6	1.9	2	6	4	1.45
Ecore2Class	14	8	3	10	6	2.12	3	5	3	0.94
EliminateRedundantInheritance	7	5	3	4	3	0.89	2	4	3	0.63
emf2km3	26	19	2	33	11	8.19	2	5	3	1.19
EquivalenceAttributesAssociations	51	17	4	28	11	5.41	2	5	3	1.21
extractClass	9	4	8	8	8	0	3	4	3	0.87
Families2Persons_final	11	8	1	7	5	1.8	1	4	2	1.06
hideDelegate	49	22	6	95	30	23.37	4	8	5	1.11
IntroducePrimaryKey	24	14	3	10	6	1.95	2	5	3	0.96
IntroducingInterface	40	11	5	17	9	3.81	3	5	4	0.8
JavaSource2Table	17	10	4	7	5	1.18	2	6	3	1.61
KM32EMF	18	15	3	11	7	2.53	2	5	3	0.89
KM32Problem	29	24	8	241	23	45.84	4	6	5	0.82
uml-er	36	21	2	8	4	1.69	2	6	3	1.09
uml-er-iccs	18	10	2	6	4	1.76	2	4	2	1.05
uml-er2	34	20	2	7	3	1.63	2	6	2	1.43
uml-er3	53	28	2	4	3	0.73	2	4	2	0.94
UML2ClassDiagram_to_KM3	104	29	32	71	46	10.81	8	12	10	1.38

	taille conclusion				profondeur conclusion			
nom	min	max	moyenne	écart type	min	max	moyenne	écart type
associations-persons	1	3	2	0.58	1	3	2	0.58
Book2Publication	1	1	1	0	1	1	1	0
delegation1	8	16	9	2.1	3	6	4	1.07
delegation2	9	32	21	9.56	5	7	5	1.08
Disaggregation	3	9	5	1.76	2	5	4	0.95
Ecore2Class	1	5	4	1.32	1	4	3	1.12
EliminateRedundantInheritance	3	6	5	1.26	2	4	3	0.63
emf2km3	3	11	7	2.35	2	5	3	0.86
EquivalenceAttributesAssociations	5	19	10	4.38	3	6	4	0.84
extractClass	10	13	11	1.32	4	5	4	0.87
Families2Persons_final	1	3	2	0.5	1	3	2	0.5
hideDelegate	4	109	31	28.96	2	8	4	1.49
IntroducePrimaryKey	3	8	5	1.58	2	5	3	0.96
IntroducingInterface	2	14	7	3.75	2	5	3	1.48
JavaSource2Table	1	2	1	0.63	1	2	1	0.63
KM32EMF	2	33	12	8.88	2	4	3	0.89
KM32Problem	1	1	1	0	1	1	1	0
uml-er	3	21	10	4.85	2	10	5	2.5
uml-er-iccs	5	6	5	0.32	3	4	3	0.77
uml-er2	3	21	10	4.89	2	10	5	2.52
uml-er3	3	21	10	4.92	2	10	5	2.48
UML2ClassDiagram_to_KM3	4	13	8	2.46	2	6	4	1.13

TABLE 4.11 : Métriques quantitatives sur les treillis des règles obtenus à partir de notre ensemble d'exemples avec une configuration utilisant l'opérateur $S_\exists$ (processus $Proc_\exists$)

	taille du	nombre de	taille premisse				profondeur premisse			
nom	treillis MapLinks	règles extraites	min	max	moy.	écart type	min	max	moy.	écart type
associations-persons	9	6	1	5	4	1.53	1	3	5	0.82
Book2Publication	4	2	2	2	2	0	2	2	4	1.21
delegation1	47	12	15	33	18	5.89	4	8	5	1.22
delegation2	8	5	7	17	9	4	3	5	3	0.87
Disaggregation	22	10	6	10	6	1.26	4	6	5	0.8
Ecore2Class	14	8	7	15	8	3.16	3	6	5	0.95
EliminateRedundantInheritance	8	6	7	11	8	1.63	4	6	4	1.27
emf2km3	26	19	14	52	20	9.48	5	7	4	0.77
EquivalenceAttributesAssociations	47	18	11	36	15	7.96	3	6	4	0.77
extractClass	10	4	6	6	6	0	3	4	4	1.15
Families2Persons_final	11	8	1	7	5	1.8	1	3	4	1.22
hideDelegate	53	22	17	117	35	28.7	4	7	2	0
IntroducePrimaryKey	31	17	11	18	12	2.5	4	7	2	0.91
IntroducingInterface	38	11	10	20	10	3.02	3	6	4	1.61
JavaSource2Table	17	10	5	9	7	1.3	3	7	2	0.87
KM32EMF	18	15	11	20	14	3.74	4	6	3	1.35
KM32Problem	37	28	28	378	52	65.36	7	9	8	0.65
uml-er	35	24	2	11	6	2.75	2	9	5	1.15
uml-er-iccs	21	12	3	9	6	1.78	2	5	7	1.54
uml-er2	25	15	2	9	4	2.19	2	6	3	1.41
uml-er3	40	19	2	4	3	0.83	2	4	2	0.97
UML2ClassDiagram_to_KM3	200	40	28	187	58	40.51	5	10	5	2.72

	taille conclusion				profondeur conclusion			
nom	min	max	moyenne	écart type	min	max	moyenne	écart type
associations-persons	1	4	2	0.91	1	3	2	0.58
Book2Publication	1	1	1	0	1	1	1	0
delegation1	7	17	7	2.89	3	6	4	1.04
delegation2	15	26	20	4.96	5	6	5	0.63
Disaggregation	7	11	7	1.26	4	6	4	0.89
Ecore2Class	1	8	4	2.26	1	4	3	1
EliminateRedundantInheritance	7	11	8	1.63	4	6	5	0.82
emf2km3	11	20	14	3.59	4	6	4	1.17
EquivalenceAttributesAssociations	9	38	14	8.37	4	7	5	1.2
extractClass	16	20	18	2	5	6	5	0.87
Families2Persons_final	1	4	2	0.79	1	3	2	0.5
hideDelegate	17	186	39	37.75	4	7	5	0.88
IntroducePrimaryKey	9	14	9	1.21	4	6	4	1.03
IntroducingInterface	2	20	9	4.53	2	6	4	1.48
JavaSource2Table	1	2	1	0.63	1	2	1	0.63
KM32EMF	14	52	21	10.23	4	7	5	1.1
KM32Problem	1	1	1	0	1	1	1	0
uml-er	2	15	9	3.61	2	8	5	1.88
uml-er-iccs	5	6	5	0.29	3	4	3	0.76
uml-er2	2	15	9	3.71	2	8	5	1.88
uml-er3	2	15	9	3.47	2	8	5	1.84
UML2ClassDiagram_to_KM3	18	45	22	6.06	4	7	5	0.74

elle est reliée par *ownedAttribute*. Dans ce cas là il n'y a alors plus de différence avec la *Class Client*.

Les règles 4, 5 et 6 ont une *Property* pour racine de la prémisse, les règles 5 et 6 partagent la même prémisse. Là encore le processus $Proc_{\forall\exists}$ limite le nombre de règles créées par rapport au processus $Proc_{\exists}$. On retrouve ainsi pour la règle 4 deux règles dans le treillis des règles obtenu par le processus $Proc_{\exists}$ conte une seule pour le processus $Proc_{\forall\exists}$. On retrouve par contre dans les deux cas trois règles pour représenter la règle 5, ce qui s'explique par le fait que dans l'exemple la *Property ownership* est reliée à une *Association* par l'association *owning Association* tandis que la *Property owner* est reliée à une *Class* par l'association *owning Class*. Comme le montre l'extrait du treillis des règles généré par le processus $Proc_{\forall\exists}$ présenté à la figure 4.17 : *rules_6* est la règle générée à partir de *owner* et *rules_11* est la règle générée à partir de *ownership*. *rules_13* est une règle plus générale ne prenant pas en compte ces différences.

rules_8 est ce qui se rapproche le plus de la règle 4, mais il manque par rapport à la définition l'interdiction pour la *Property* d'être reliée à une *Association*. Le processus $Proc_{\forall\exists}$ utilise l'opérateur de scaling $S_{\forall\exists}$ qui empêche la création de propriétés interdites sans risque de changer la sémantique du concept à l'origine de la règle. Rien ne permet en effet de dire dans *rules_8* que, parmi tous les liens d'appariement formant l'extension du concept dans le treillis *MapLinks*, aucun des éléments sources n'est relié à une *Association*. Le processus $Proc_{\exists}$ permet par contre de déduire à partir du treillis un ensemble de propriétés interdites en regardant les propriétés qui n'apparaissent ni dans le concept de *MapLinks* analysé ni dans les concepts qui partagent une partie de son extension.

4.5 Conclusion

La création de transformation est une opération complexe nécessitant la connaissance des méta-modèles, détenue par les experts métier, et la maîtrise du langage de transformation, apportée par le développeur. Une approche de transformation basée sur les exemples permet aux experts métier de participer à la création d'une transformation. Nous avons proposé dans ce chapitre une approche de transformation à partir d'exemples basée sur l'Analyse Relationnelle de Concepts qui est un processus de classification appartenant au domaine de la théorie des treillis. Le processus prend en paramètres un exemple de modèle source et le modèle transformé correspondant, ainsi que l'alignement de ces deux modèles.

Nous générons une classification des éléments de chaque modèle de sorte qu'à chaque élément soit associé un motif décrivant son environnement. La classification des liens d'appariement permet alors d'associer un motif source avec un motif cible pour créer une règle. Cette approche est implémentée dans un outil que nous utilisons pour évaluer notre approche sur une étude de cas. Nous évaluons pour chacune des transformations de notre étude de cas un ensemble de métriques concernant la taille du treillis obtenu et des règles

qui le composent. Nous évaluons par ailleurs la pertinence des résultats obtenus sur une transformation particulière. Les résultats obtenus sont satisfaisants mais mériteraient une évaluation à plus grande échelle. La taille et le nombre des règles restent raisonnables pour une interprétation par le développeur, bien que dans certains cas les plus grandes règles apparaissent un peu importantes. Ces résultats nous incitent donc à étudier des méthodes pour réduire la taille des règles tout en maintenant une expressivité suffisante. Une méthode pour atteindre ce but consisterait à réduire le nombre d'itérations du processus ARC.

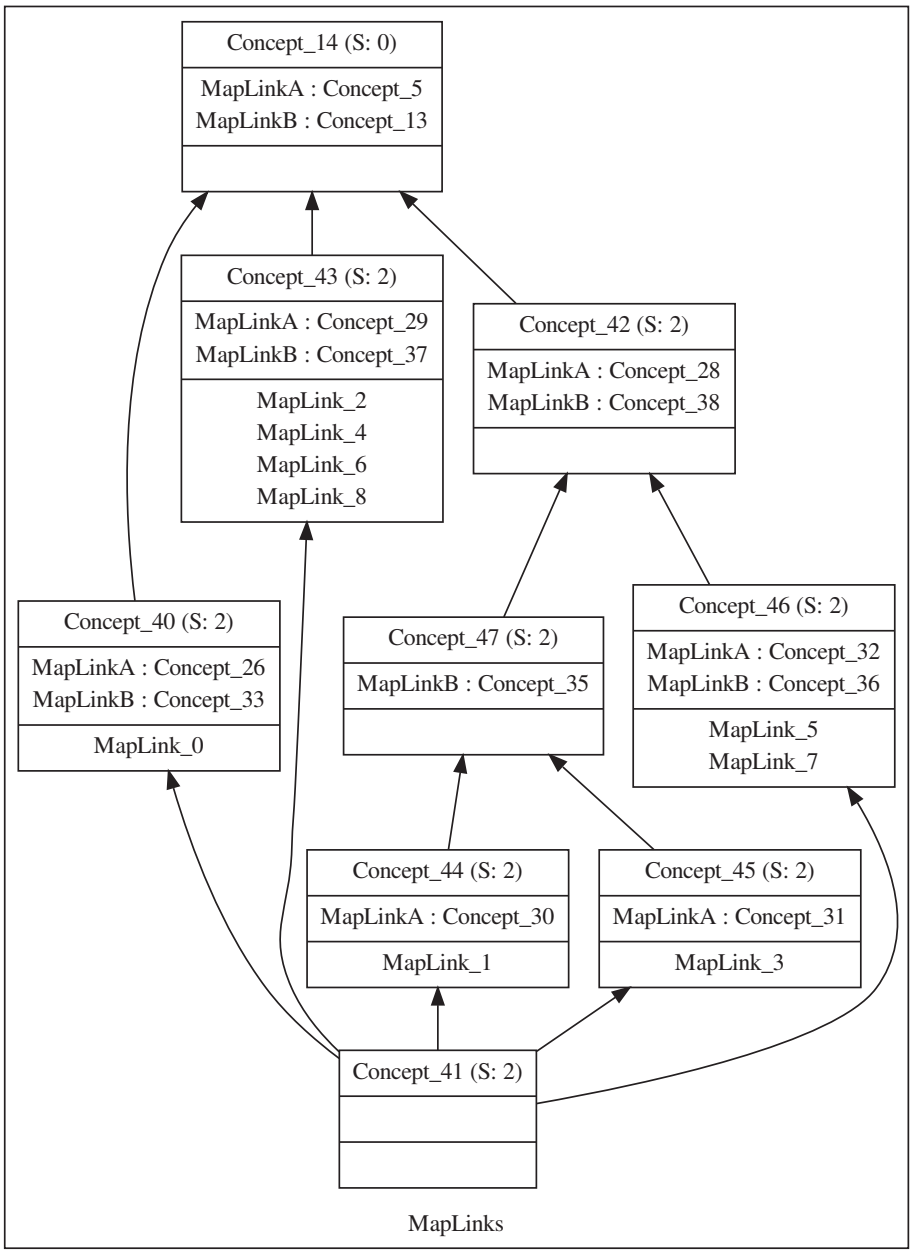


FIGURE 4.10 : Treillis issus de l'application de l'ARC sur le contexte *MapLinks* décrit à la table 4.8.

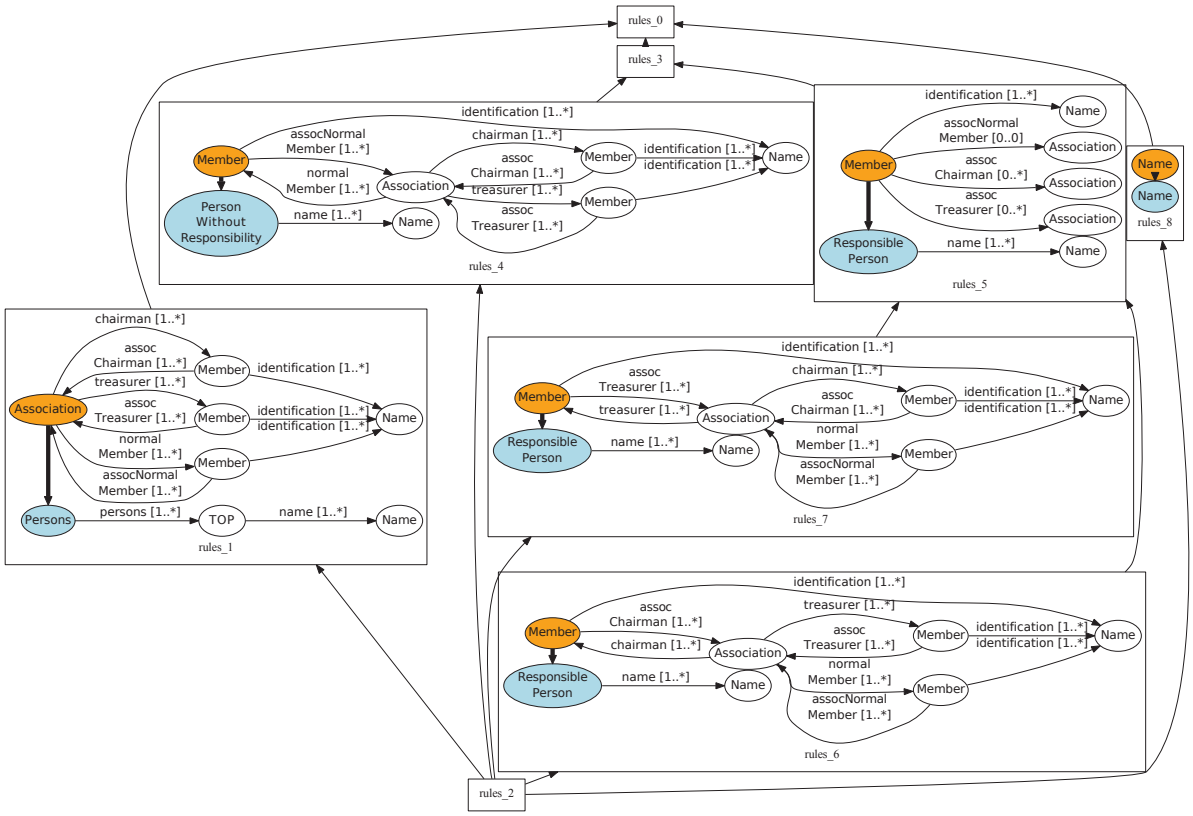


FIGURE 4.11 : Représentation concrète du modèle de règles correspondant à l'interprétation des treillis de notre exemple illustratif.

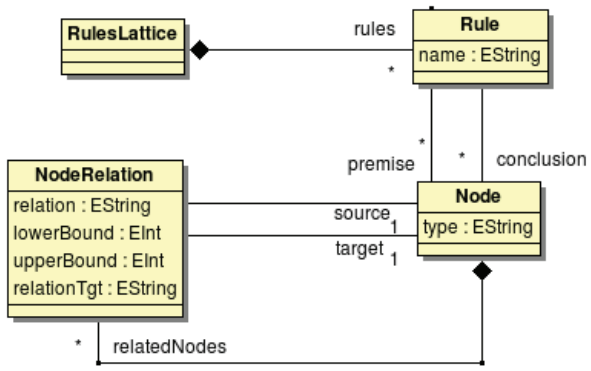


FIGURE 4.12 : Métamodèle de la représentation des règles.

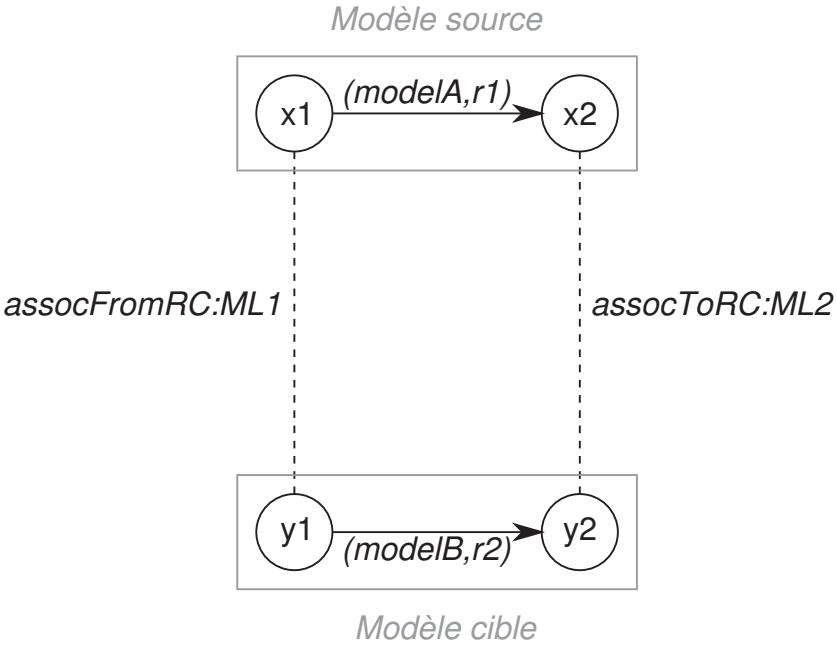


FIGURE 4.13 : Illustration de l'interprétation d'un concept du treillis *AssocFC*.

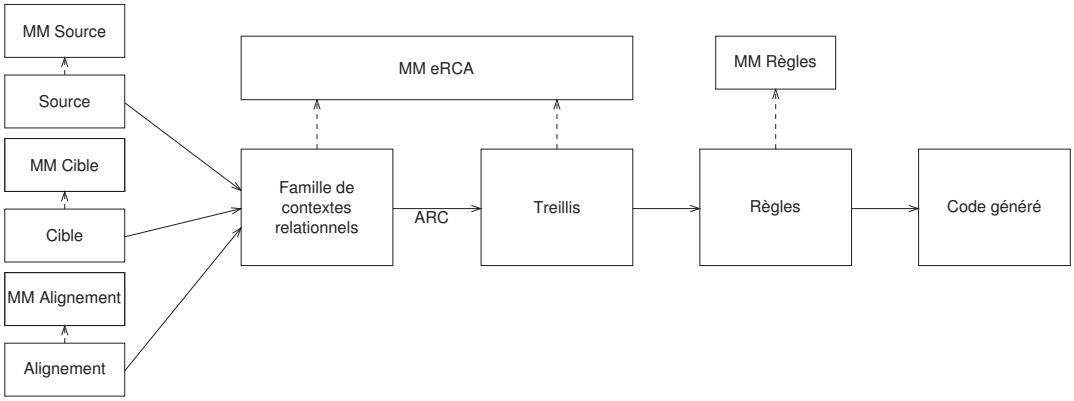


FIGURE 4.14 : Schéma de fonctionnement de l'outil.

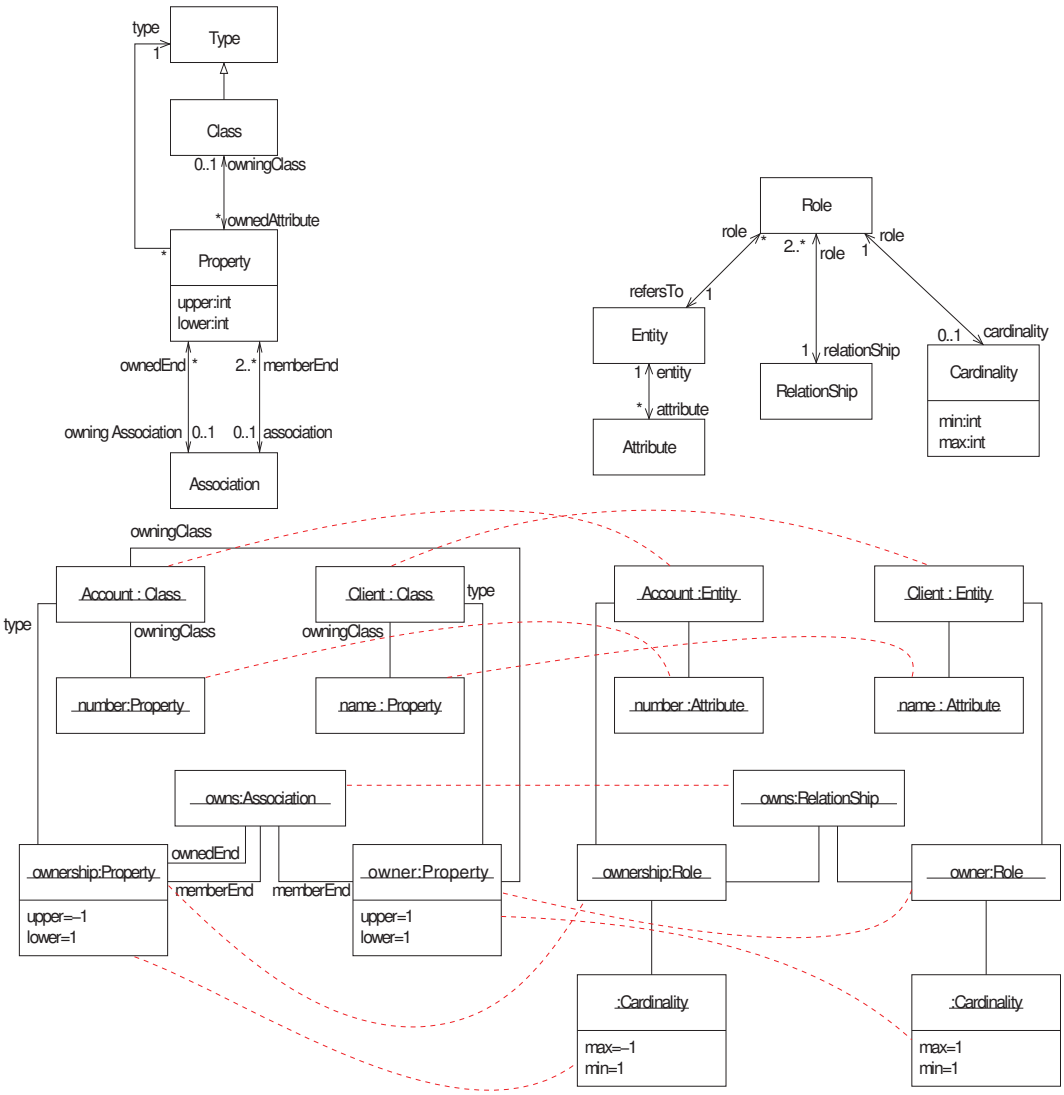


FIGURE 4.15 : *En haut :* Les deux méta-modèles implémentés, à gauche le méta-modèle que nous appellerons UML et à droite le méta-modèle que nous appellerons Entité-Association. *En bas :* les deux modèles d'exemple avec les liens d'appariement représentés par des lignes en pointillés.

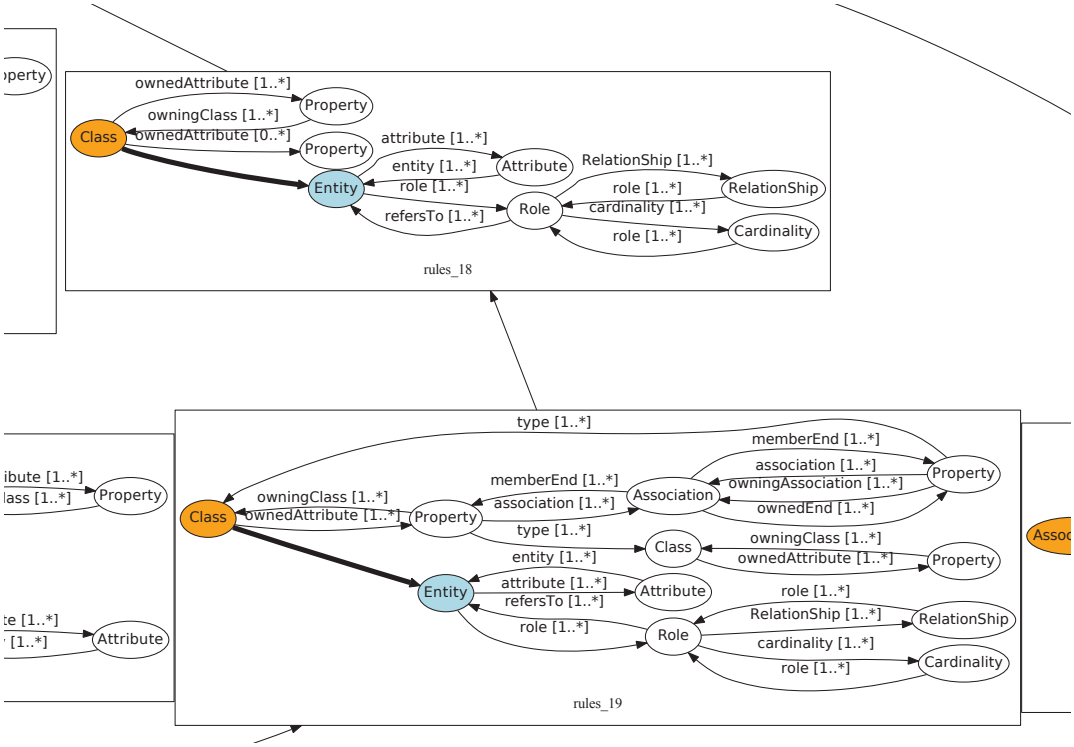


FIGURE 4.16 : Ce que génère le processus $Proc_{\exists}$ pour la règle 2.

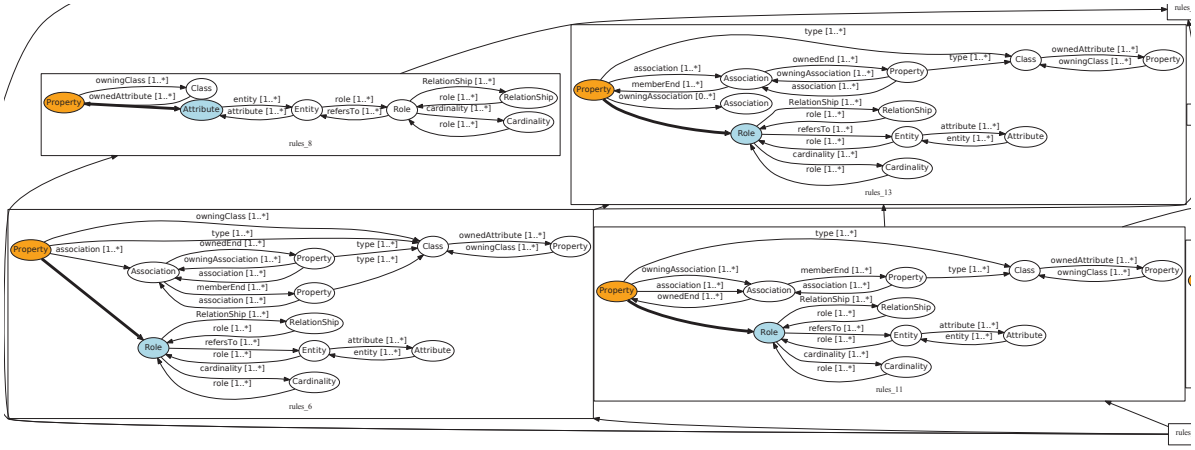


FIGURE 4.17 : Les règles générées par le processus $Proc_{\forall\exists}$ correspondant aux règles 4 et 5.

Chapitre 5

Conclusion et Perspectives

La création de transformation de modèles est un processus difficile car il nécessite différentes compétences réparties entre plusieurs acteurs. Il est nécessaire de maîtriser un langage de transformation de modèles, ce qui est normalement le rôle du développeur, et de connaître les méta-modèles sources et cibles de la transformation. Cette dernière compétence appartient généralement à des experts métiers qui n'ont pas de compétences particulières dans le développement de transformations. L'interaction entre les acteurs est primordiale puisqu'il s'agit pour les experts métiers de transmettre au développeur les informations nécessaires à la spécification d'une transformation.

Une solution proposée dans la littérature est de générer la transformation à partir d'exemples de modèles. Cela permet aux experts métier de participer activement à la création de la transformation sans qu'ils aient besoin d'acquérir des compétence de programmation de transformations de modèles. Ce type d'approche n'est apparu que récemment et de nombreux problèmes n'ont pas encore trouvé de solution satisfaisante. Par exemple l'exploitation des exemples nécessite qu'ils soient alignés, c'est-à-dire qu'il faut établir des liens entre les éléments du modèle cible et les éléments du modèle source qui ont permis de les créer. Les autres problèmes se situent au niveau des pré-requis de chaque approche et de la forme des résultats que l'on obtient.

Nous présentons dans cette thèse une nouvelle approche de génération de transformation de modèles par l'exemple. Cette approche a été séparée en deux parties : une première partie dédiée à la génération d'alignements de modèles et une seconde partie dédiée à la génération de transformations de modèles proprement dite. Chacune de ces parties a fait l'objet d'une recherche dans la littérature pour appréhender l'existant et situer nos travaux par rapport à l'état de l'art.

Nous nous sommes intéressés dans une premier temps à simplifier le processus de création des données d'entrée nécessaires à l'application de notre approche. En effet la création de l'alignement des modèles d'exemples est une tâche fastidieuse et propice à l'apparition d'erreurs. Les alignements de modèles ont une grande importance dans les approches de transformation de modèles à partir d'exemples car ils sont la représentation

de la trace d'exécution de la transformation à générer sur un exemple. Nous avons donc proposé d'aligner le modèle source et le modèle transformé d'un exemple, afin de déduire le plus automatiquement possible les liens de transformation entre la source et la cible. Pour développer notre approche nous nous sommes inspirés des travaux existants sur les alignements de schéma et d'ontologies. Nous avons adapté une de ces approches au problème particulier de l'alignement de modèles pour l'intégrer dans un outil. Cet outil nous a permis de réaliser une étude de cas montrant sur un ensemble d'exemples de transformations provenant de sources diverses l'efficacité de notre approche. Les résultats sont très encourageants mais montrent que l'approche d'alignement nécessite des améliorations notamment au niveau de la précision.

Notre seconde contribution est une nouvelle approche de génération de transformations de modèles, qui permet de pallier les limitations des approches existantes décrites précédemment. L'approche de génération de transformation de modèles à partir d'exemples s'apparente aux approches existantes tout en proposant une amélioration sur certains points que nous considérons essentiels. Notre approche utilise une méthode de classification appartenant au domaine de la théorie des treillis. Nous l'utilisons ici comme un système d'apprentissage sur des exemples. Nous extrayons de chaque modèle une classification des éléments en associant à chaque élément un motif dépendant de son type et son voisinage. Les règles sont ensuite déduites à partir de l'association d'un motif source avec un motif cible obtenu grâce aux liens d'appariement et à la classification précédente. Nous avons fait en sorte de limiter les pré-requis techniques nécessaires à l'application de notre approche pour en permettre son usage dans la majorité des cas. Ainsi l'implémentation actuelle de l'approche nécessite seulement que les modèles soient reconnus par le framework de modélisation utilisé. Cela signifie qu'aucun développement particulier n'est nécessaire pour un méta-modèle donné. Nous avons par ailleurs limité la quantité d'information demandée à l'utilisateur en entrée du processus de génération en plus des modèles d'exemples. Ainsi nous ne demandons pas de classification préalable des liens d'appariement et pour ces derniers notre approche de génération d'alignements permet de réduire l'effort nécessaire à leur création. L'utilisation d'une méthode de classification nous permet d'obtenir en sortie un ensemble de solutions organisées sous forme de treillis ce qui en permet une navigation facile. Cette approche a été implémentée dans un outil qui nous a permis de réaliser une étude de cas sur les exemples utilisés précédemment. Les résultats montrent que l'approche retourne des résultats satisfaisants mais soulève des réserves quant à la taille des règles obtenues, ce qui nous pousse à envisager la modification de certains paramètres.

5.1 Perspectives

Les travaux présentés dans ce manuscrit ont permis d'ouvrir de nouvelles perspectives pour l'amélioration de notre approche ou l'exploration de nouvelles pistes. Nous présen-

tons ici les suites que nous souhaitons donner à nos travaux. Nous avons partagé nos perspectives en deux sections correspondant aux deux parties de notre contribution.

5.1.1 Alignement de modèles

Amélioration de l'approche. L'appariement guidé par les valeurs d'attributs est fondé essentiellement sur l'analyse des chaînes de caractères. Une évolution du système pourrait prendre en compte d'autre types d'attributs et dans certains cas considérer plusieurs attributs à la fois : par exemple un intervalle est souvent représenté par deux attributs, on pourrait donc faire l'analyse sur deux attributs en même temps plutôt qu'un seul. Cela ne pourrait pas se faire sur tous les types d'attributs au risque de faire exploser la complexité déjà relativement importante. Nous nous sommes aperçus durant l'évaluation que l'application d'AnchorPROMPT fait baisser la précision dans de nombreux cas par rapport aux résultats donnés par une simple approche d'appariement basée sur les valeurs d'attributs. Pour pallier cela on pourrait rajouter des contraintes sur les valeurs d'attributs en plus du système d'augmentation de la similarité. Ces contraintes sur les valeurs d'attributs seraient bien évidemment moins fortes que celles utilisées pour la création des ancres : on pourrait par exemple supprimer l'impact de la fréquence d'apparition de ces valeurs dans les modèles. Si l'évaluation montre que l'impact sur la précision disparaît tout en gardant un certain bénéfice sur le rappel, cela rendrait peut-être possible en utilisant l'alignement obtenu de faire une seconde itération d'AnchorPROMPT sans conditions (telle qu'elle existe dans l'approche actuelle) afin d'obtenir de meilleurs résultats.

Alternative à AnchorPROMPT. Nous avons choisi pour générer des alignements de modèles d'adapter l'approche d'alignement d'ontologies AnchorPROMPT car elle nous semble la plus appropriée. Malgré tout, d'autres méthodes d'alignement pourraient servir d'inspiration, notamment le *similarity flooding* [Melnik *et al.*, 2002] ou OLA [Euzenat *et al.*, 2004]. Ces deux méthodes similaires posent de nombreux problèmes pour l'étiquetage des relations entre éléments qui sont nécessaires pour la propagation de similarité. Mais s'il s'avère possible de contourner ces problèmes, elles permettraient sûrement d'obtenir des résultats différents de ceux obtenus avec AnchorPROMPT. Il faudrait alors les comparer et suivant les résultats, choisir la meilleure alternative ou combiner deux méthodes.

Amélioration de l'implémentation. L'implémentation de l'approche contient une interface de création manuelle des liens, pour analyser les résultats obtenus et permettre de les modifier ou les compléter. L'interface actuelle est relativement simple et peu ergonomique, il serait intéressant d'explorer les solutions possibles dans le domaine de la création d'interfaces graphiques. Il faudrait notamment prévoir une plus grande interaction avec l'utilisateur, par exemple pour lui signaler les appariements créés ayant un indice de similarité faible et donc sur lesquels l'utilisateur devrait porter son attention en priorité.

Les résultats obtenus lors de la validation se sont concentrés essentiellement sur les résultats objectifs, mais il faudrait aussi pouvoir prendre en compte la réaction des développeurs face à ce type d'approche.

5.1.2 Génération de transformation de modèles

Amélioration de la génération de règles. La principale piste à explorer en vue d'améliorer l'approche de génération de transformation est l'étape de génération de liens entre éléments dans le modèle cible. L'approche présentée dans ce manuscrit étend la famille de contextes passée en paramètre de l'Analyse Relationnelle de Concepts pour analyser pour chaque voisin d'un élément du modèle source s'il est aussi voisin de l'élément transformé dans le modèle cible. D'autres approches peuvent être explorées, par exemple [Wimmer *et al.*, 2007] propose une méthode simple basée sur les voisins possibles d'un élément par analyse du méta-modèle qui pourrait s'avérer suffisamment efficace dans la majorité des cas. Le processus doit aussi pouvoir prendre en compte les valeurs d'attributs à modifier, il faudrait sans doute pour cela modifier la définition des alignements pour étudier sur les exemples d'où proviennent les valeurs dans les modèles cibles. La génération d'alignement automatique se basant sur les valeurs d'attributs, il serait sans doute pertinent d'utiliser les résultats du processus d'appariement de modèles.

Modifier la représentation des alignements. La description des alignements apparaît à l'usage un peu limitée, et de ce fait la définition d'un bon alignement est souvent sujette à discussion. Ainsi il faudrait pouvoir rajouter de nouveaux type d'appariements ou de marqueurs pour signaler par exemple les éléments créés par défaut (par exemple la racine d'un modèle doit souvent être créée quoi qu'il arrive et n'a pas forcément besoin d'informations provenant de la racine du modèle source). La possibilité d'avoir plusieurs éléments dans une extrémité d'appariement apparaît aussi comme nécessaire, mais peut poser problème lors de la traduction vers des contextes pour l'ARC. Nous avons dans notre expérimentation présenté des refactorings, mais beaucoup d'entre eux sont des refactorings locaux prenant des éléments d'un modèle en paramètre pour les modifier. Il serait intéressant d'étudier si la distinction des éléments paramètres de la transformation permet de générer des transformations paramétrées. Tous ces changements seraient évidemment à répercuter sur le processus de génération d'alignements.

Adaptation de l'Analyse Relationnelle de concepts. Au niveau de l'ARC, les opérateurs de scaling ont un impact important sur la sémantique des résultats. En utiliser plusieurs à la fois pourrait améliorer la précision des résultats, mais se poseraient alors les problèmes de performance et de quantité des résultats. Il est aussi possible de créer de nouveaux opérateurs de scaling, par exemple un opérateur dont le sens serait « il existe un unique ». La taille des motifs obtenus dans les règles pose aussi un problème : les résultats montrent

que cette taille est souvent grande ce qui rend difficile l'interprétation des règles, et l'expérience montre que dans beaucoup de cas, seule une partie des motifs est intéressante. La taille des motifs pourrait être diminuée en limitant le nombre d'itérations du processus ARC, mais il faudrait dans ce cas établir le critère justifiant cette limite.

Nous avons durant nos expériences rencontré quelques problèmes de performances dans le cas de gros exemples. Il existe de nombreux algorithmes pour implémenter l'ARC dont l'efficacité dépend du type de données évaluées. Il est possible que l'un d'entre eux soit plus approprié pour le cas particulier de la génération de transformation de modèles. Par ailleurs l'approche MTBE dont nous nous inspirons est itérative, et nécessiterait la spécification et l'utilisation d'algorithmes incrémentaux pour ne prendre en compte que les modifications apportées sur les exemples entre deux itérations.

Pour remplacer ou compléter l'utilisation de l'ARC, nous envisageons d'étudier les techniques utilisées dans les systèmes de raisonnement à base de cas qui sont des systèmes d'apprentissage qui nous apparaissent appropriés à notre problématique.

Évaluation des capacités de l'approche. Nous avons vu par l'expérience que notre approche ne permettait pas de générer n'importe quelle transformation. Il serait intéressant d'étudier en termes de calculabilité les limites de notre approche. Une fois ces limites définies il deviendra alors possible d'imaginer de nouvelles approches permettant de dépasser ces limites.

Amélioration de l'implémentation. Dans l'implémentation actuelle, le treillis des règles est affiché grâce à l'outil de visualisation de graphes Graphviz ([Graphviz]) : mais cet outil ne nous permet pas de modifier les règles ou la manière de les afficher ni de naviguer entre elles. De la même manière que dans la section précédente, il est nécessaire d'améliorer l'interface avec l'utilisateur. L'approche propose un méta-modèle de règles pour décrire le résultat de notre approche. Les modèles conformes à ce méta-modèle peuvent ensuite être traduits dans un langage de transformation. À l'heure actuelle seule la traduction vers ATL est développée, et des traductions vers d'autres langages sont à envisager.

Améliorer la validation. La validation du processus reste une tâche difficile. Il est important de pouvoir étudier l'impact de notre approche sur le développement en terme de rapidité de développement mais aussi en terme de qualité du résultat. Une bonne validation nécessiterait par exemple l'utilisation de l'approche par un ensemble de développeurs en situation réelle. La qualité du résultat pourrait alors se mesurer sur la quantité et le type de modifications effectuées par ces développeurs. En effet si le développeur ne fait que rajouter des éléments aux règles obtenues, cela signifie que le résultat bien qu'incomplet a fourni une base de départ saine sur laquelle s'appuyer, tandis qu'un grand nombre de suppressions montre que le résultat obtenu introduit des erreurs dans le processus de développement. Il serait aussi important d'analyser le ressenti des développeurs car il ar-

rive que des approches de développement théoriquement efficaces soient en réalité trop contraignantes pour être adoptées.

Adaptation de notre travail à d'autres problématiques. Nous nous sommes focalisés pour cette thèse sur l'utilisation de notre approche comme un moyen de développer une nouvelle transformation de modèles. Cependant les outils de traçabilité que l'on peut trouver dans certains systèmes de transformation de modèles peuvent représenter un autre moyen d'obtenir un alignement entre deux modèles. Ainsi, l'utilisation de ces traces nous permettrait d'utiliser notre approche pour faciliter la ré-ingénierie de transformations dans d'autres langages de programmation. De nombreuses raisons peuvent pousser à changer d'environnement de travail, rendant les applications développées dans l'environnement précédent obsolètes. Il est alors nécessaire de les redévelopper dans le nouvel environnement. Le problème se pose de la même manière pour les transformations de modèles et une approche par l'exemple, si une trace d'exécution de l'ancien système est disponible, pourrait faciliter grandement la migration.

Bibliographie

ALEPH. site web du projet ALEPH, 2005. http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph. Cité page 13.

David AUMUELLER, Hong-Hai DO, Sabine MASSMANN et Erhard RAHM : Schema and ontology matching with COMA++. *In SIGMOD 2005*, pages 906–908. ACM, 2005. Cité page 22.

Zoltán BALOGH et Dániel VARRÓ : Model transformation by example using inductive logic programming. *Software and Systems Modeling*, 8(3):347–364, 2009. Cité pages 13, 17 et 18.

Marc BARBUT et Bernard MONJARDET : *Ordre et Classification : Algèbre et Combinatoire*, volume 2. Hachette, 1970. Cité page 56.

Garrett BIRKHOFF : *Lattice theory*. American Mathematical Society, Providence, RI, 1st édition, 1940. Cité page 56.

Grady BOOCH, James RUMBAUGH et Ivar JACOBSON : *The Unified Modeling Language User Guide*. Addison-Wesley, 1998. Cité page 5.

Frank BUDINSKY, Stephen A. BRODSKY et Ed MERKS : *Eclipse Modeling Framework*. The Eclipse series. Pearson Education, 2003. Cité pages 6, 9 et 37.

Jean BÉZIVIN, Grégoire DUPÉ, Fredéric JOUAULT, Gilles PITETTE et Jamal Eddine ROUGUI : First experiments with the atl model transformation language : Transforming xslt into xquery. *In OOPSLA 2003 Workshop*, 2003. Cité pages 9 et 75.

Peter P. CHEN : The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976. Cité page 24.

György CSERTÁN, Gábor HUSZERL, István MAJZIK, Zsigmond PAP, András PATARICZA et Dániel VARRÓ : Viatra : Visual automated transformations for formal verification and validation of uml models. *In Proceedings of the 17th IEEE international conference on Automated software engineering*. IEEE Computer Society, 2002. Cité page 9.

Krzysztof CZARNECKI et Simon HELSEN : Feature-based survey of model transformation approaches. *IBM SYSTEMS JOURNAL*, 45(3):621–645, 2006. Cité pages 9 et 39.

Marcos Didonet DEL FABRO et Patrick VALDURIEZ : Semi-automatic model integration using matching transformation and weaving models. *In International Conference SAC'07*, pages 963–970. ACM, 2007. Cité pages 11 et 17.

Hong-Hai DO, Sergey MELNIK et Erhard RAHM : Comparison of schema matching evaluations. *In Web, Web-Services, and Database Systems*, pages 221–237, 2002. Cité page 45.

Hong-Hai DO et Erhard RAHM : COMA - a system for flexible combination of schema matching approaches. *In VLDB*, pages 610–621. Morgan Kaufmann, 2002. Cité page 22.

AnHai DOAN, Jayant MADHAVAN, Robin DHAMANKAR, Pedro DOMINGOS et Halevy ALON : Learning to match ontologies on the semantic web. *VLDB J.*, 12(4):303–319, 2003. Cité page 22.

Xavier DOLQUES, Marianne HUCHARD et Clémentine NEBUT : From transformation traces to transformation rules : Assisting model driven engineering approach with formal concept analysis. *In Supplementary Proceedings of ICCS'09*, pages 15–29, 2009a. Cité page 3.

Xavier DOLQUES, Marianne HUCHARD et Clémentine NEBUT : Génération de transformation de modèles par application de l'ARC sur des exemples. *In LMO'09 : Langages et Modèles à Objets*, pages 61–75, 2009b. Cité page 3.

Xavier DOLQUES, Marianne HUCHARD, Clémentine NEBUT et Philippe REITZ : Fixing generalization defects in UML use case diagrams. *In CLA10*, 2010a. à paraître. Cité page 105.

Xavier DOLQUES, Marianne HUCHARD, Clémentine NEBUT et Philippe REITZ : Learning transformation rules from transformation examples : An approach based on relational concept analysis. *In Companion proceedings of EDOC'10*. IEEE Computer Society Press, 2010b. à paraître. Cité page 3.

Xavier DOLQUES, Lala MADIHA HAKIK, Marianne HUCHARD, Clémentine NEBUT et Philippe REITZ : Correction des défauts de généralisation dans les diagrammes de cas d'utilisation UML. *In LMO'10 : Langages et Modèles à Objets*, pages 51–66, 2010c. Cité page 105.

dynamicGMF. site web du projet reflective ecore model diagram editor, 2009. http://dynamicgmf.sourceforge.net/. Cité page 24.

Marc EHRIG et Steffen STAAB : QOM - quick ontology mapping. *In International Semantic Web Conference*, pages 683–697. Springer, 2004. Cité page 22.

eRCA. site web de l'outil eRCA.
http://erca.googlecode.com. Cité page 75.

Jacky ESTUBLIER, Jean-Marie FAVRE, Jean BÉZIVIN, Laurence DUCHIEN, Raphael MARVIE, Sébastien GÉRARD, Benoit BAUDRY, Mokrane BOUZHEGOUB, Jean Marc JÉZÉQUEL, Mireille BLAY et Michel RIVEIL : Rapport de synthèse. Rapport technique, Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles, 2005. Cité page 5.

Jérôme EUZENAT, David LOUP, Mohamed TOUZANI et Petko VALTCHEV : Ontology alignment with ola. *In Proceedings of the 3rd EON Workshop, 3rd International Semantic Web Conference*, pages 333–337, 2004. Cité pages 21, 22 et 89.

Jean-Rémy FALLERI, Marianne HUCHARD, Mathieu LAFOURCADE et Clémentine NEBUT : Meta-model Matching for Automatic Model Transformation Generation. *In MODELS'08, LNCS 5301*, pages 326–340. Springer, 2008. Cité pages 11 et 17.

Martin FOWLER, Kent BECK, John BRANT, William OPDYKE et Don ROBERTS : *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 2000. Cité page 44.

Bernhard GANTER et Rudolf WILLE : *Formal Concept Analysis, Mathematical Foundations*. Springer, 1999. Cité page 56.

Fausto GIUNCHIGLIA, Mikalai YATSKEVICH et Pavel SHVAIKO : Semantic matching : Algorithms and implementation. 9:1–38, 2007. Cité page 23.

Graphviz. site web de l'outil Graphviz.
http://www.graphviz.org. Cité page 91.

Marianne HUCHARD, Mohamed Rouane HACÈNE, Cyril ROUME et Petko VALTCHEV : Relational concept discovery in structured datasets. *Ann. Math. Artif. Intell.*, 49(1-4):39–76, 2007a. Cité page 56.

Marianne HUCHARD, Amedeo NAPOLI, Mohamed Rouane HACÈNE et Petko VALTCHEV : Mining description logics concepts with relational concept analysis. *In* Paula BRITO, Patrice BERTRAND, Guy CUCUMEL et Francisco de CARVALHO, éditeurs : *Selected Contributions in Data Analysis and Classification*, volume 49, pages 39–76. 2007b. Cité page 62.

Gerti KAPPEL, Elisabeth KAPSAMMER, Horst KARGL, Gerhard KRAMLER, Thomas REITER, Werner RETSCHITZEGGER, Wieland SCHWINGER et Manuel WIMMER : Lifting metamodels to ontologies : A step to the semantic integration of modeling languages. *In Proceedings of MoDELS 2006*, pages 528–542, 2006. Cité page 11.

James KENNEDY et Russel C. EBERHART : Particle swarm optimization. *In Proceedings IEEE International Conference on Neural Networks*, pages 1942–1948, 1995. Cité page 16.

Marouane KESSENTINI, Arbi BOUCHOUCHA, Houari SAHRAOUI et Mounir BOUKADOUM : Example-based Sequence Diagrams to Colored Petri Nets Transformation using Heuristic Search. *In Modelling Foundations and Applications*, pages 156–172. Springer, 2010a. Cité page 16.

Marouane KESSENTINI, Houari SAHRAOUI et Mounir BOUKADOUM : Model Transformation as an Optimization Problem. *In MODELS'08, LNCS 5301*, pages 159–173. Springer, 2008. Cité pages 15, 17 et 18.

Marouane KESSENTINI, Houari SAHRAOUI et Mounir BOUKADOUM : Méta-modélisation de la transformation de modèles par l'exemple : approche méta-heuristiques. *In* Bernard CARRÉ et Olivier ZENDRA, éditeurs : *LMO'09 : Langages et Modèles à Objets*, pages 75–90, Nancy, mars 2009. Cepaduès. Cité page 15.

Marouane KESSENTINI, Houari SAHRAOUI, Mounir BOUKADOUM et Omar BEN OMAR : Model transformation by example : a search-based approach. *Software and Systems Modeling Journal*, 2010b. (à paraître). Cité page 15.

Scott KIRKPATRICK, Charles Daniel GELATT et Mario P. VECCHI : Optimization by simulated annealing. *Science*, 220:671–680, 1983. Cité page 16.

Anneke KLEPPE, Jos WARMER et Wim BAST : *MDA Explained. The Model Driven Architecture : Practice and Promise.* Addison-Wesley, 2003. Cité pages 6 et 8.

Adrian KUHN et Toon VERWAEST : Fame – a polyglot library for metamodeling at runtime. *In Third International Workshop on Models@run.time*, 2008. Cité pages 6 et 9.

Henry LIEBERMAN, éditeur. *Your Wish Is My Command : Programming by Example.* Interactive Technologies. Morgan Kaufmann Publishers, 2001. Cité pages 1 et 19.

Denivaldo LOPES, Slimane HAMMOUDI et Zair ABDELOUAHAB : Schema matching in the context of model driven engineering : From theory to practice. *In* Tarek SOBH et Khaled ELLEITHY, éditeurs : *Advances in Systems, Computing Sciences and Software Engineering*, pages 219–227. Springer, 2006. Cité page 10.

Denivaldo LOPES, Slimane HAMMOUDI et Zair ABDELOUAHAB : A step forward in semi-automatic metamodel matching : Algorithms and tool. *In* Joaquim FILIPE et José CORDEIRO, éditeurs : *Proceeding of ICEIS 2009*, pages 137–148. Springer, 2009. Cité page 11.

Denivaldo LOPES, Slimane HAMMOUDI, Jean BÉZIVIN et Frédéric JOUAULT : Generating transformation definition from mapping specification : Application to web service platform. *In CAiSE'05, LNCS 3520*, pages 309–325, 2005. Cité pages 10 et 17.

Jayant MADHAVAN, Philip A. BERNSTEIN et Erhard RAHM : Generic schema matching with cupid. *In VLDB*, pages 49–58. Morgan Kaufmann, 2001. Cité page 21.

Sergey MELNIK, Hector GARCIA-MOLINA et Erhard RAHM : Similarity flooding : A versatile graph matching algorithm and its application to schema matching. *In ICDE*, pages 117–128. IEEE Computer Society, 2002. Cité pages 11, 21, 22 et 89.

Tom MENS et Pieter Van GORP : A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006. Cité pages 8 et 39.

mtip. Model transformation in practice workshop 2005, 2005. http://sosym.dcs.kcl.ac.uk/events/mtip05/. Cité page 24.

Stephen MUGGLETON et Luc DE RAEDT : Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994. Cité page 13.

Pierre-Alain MULLER, Franck FLEUREY et Jean-Marc JÉZÉQUEL : Weaving executability into object-oriented meta-languages. *In* L. BRIAND et S. KENT, éditeurs : *Proceedings of MODELS/UML'2005*, 2005. Cité page 9.

Natalya F. NOY et Mark A. MUSEN : Anchor-prompt : Using non-local context for semantic matching. *In Proc. of the Workshop on Ontologies and Information Sharing at IJCAI-2001*, pages 63–70, Seattle (USA), 2001. Cité pages 21 et 24.

OMG : Meta object facility (MOF) core specification. Rapport technique, Object Management Group, 2006. Cité page 6.

OMG : MOF 2.0/XMI mapping, version 2.1.1. Rapport technique, Object Management Group, 2007. Cité pages 6 et 8.

OMG : MOF QVT final adopted specification. Rapport technique, Object Management Group, 2008. Cité page 12.

OMG : UML 2.3 superstructure. Rapport technique, Object Management Group, 2010. Cité pages 9 et 24.

Jeff ROTHENBERG : *AI, Simulation & Modeling*, chapitre The Nature of Modeling, pages 75–92. John Wiley & Sons, Inc., 1989. Cité page 5.

Pavel SHVAIKO et Jérôme EUZENAT : A survey of schema-based matching approaches. *Journal on Data Semantics IV*, pages 146–171, 2005. Cité page 20.

Richard SOLEY et THE OMG STAFF STRATEGY GROUP : Model driven architecture. Rapport technique, Object Management Group, 2000. Cité page 6.

Hubert TARDIEU, Arnold ROCHFELD et René COLLETTI : *La méthode Merise - Tome 1 Principes et outils*. Editions d'organisation, 1983. Cité page 5.

Dániel VARRÓ : Model transformation by example. *In Proc. MODELS 2006, LNCS 4199*, pages 410–424. Springer, 2006. Cité pages 12, 13, 14 et 53.

Manuel WIMMER, Michael STROMMER, Horst KARGL et Gerhard KRAMLER : Towards model transformation generation by-example. *In HICSS '07 : Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 285b, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2755-8. Cité pages 14, 17, 18 et 90.

Zoo ATL. site web du zoo de transformations ATL. http://www.eclipse.org/m2m/atl/atlTransformations/. Cité pages 39, 44 et 45.

Annexe A

Liste détaillée des exemples pour les études de cas

A.1 Transformations endogènes

delegation1.
- *méta-modèle :* UML
- *source :* refactoring Fowler
- *description :* refactoring sur des diagrammes UML remplaçant une relation de délégation, c'est-à-dire une association dirigée entre deux classes partageant des méthodes communes, par une relation de généralisation.

delegation2.
- *méta-modèle :* UML
- *source :* refactoring Fowler
- *description :* refactoring sur des diagrammmes UML remplaçant une relation de spécialisation entre deux classes par une association. La classe la plus spécifique dans le modèle source reprend alors les méthodes de la classe la plus générale. Il s'agit de l'opération inverse du refactoring précédent.

Disaggregation.
- *méta-modèle :* KM3
- *source :* zoo ATL
- *description :* refactoring sur des diagrammes KM3 (langage proche d'ecore) partageant une classe contenant de nombreux attributs en plusieurs classes dans lesquelles sont réparties les attributs en fonction de commentaires sur les attributs. La classe d'origine contient alors des références vers les nouvelles classes créées.

EliminateRedundantInheritance.
- *méta-modèle :* KM3
- *source :* zoo ATL
- *description :* refactoring sur des diagrammes KM3 visant à supprimer les arcs de transitivité sur les relations de généralisation d'une hiérarchie de classes.

EquivalenceAttributesAssociations.
- *méta-modèle :* UML
- *source :* zoo ATL
- *description :* refactoring sur des diagrammes UML visant à remplacer les associations par des attributs équivalents.

extractClass.
- *méta-modèle :* UML
- *source :* refactoring Fowler
- *description :* refactoring sur des diagrammes UML visant à extraire les attributs d'une classe formant un concept pour les remplacer par une classe contenant les attributs. Similaire à la transformation *Disaggregation*.

hideDelegate.

IntroducingInterface.
- *méta-modèle :* UML
- *source :* zoo ATL
- *description :* refactoring sur des diagrammes UML visant à créer des interfaces à partir des classes existantes.

IntroducePrimaryKey.
- *méta-modèle :* KM3
- *source :* zoo ATL
- *description :* refactoring sur des diagrammes KM3 ajoutant un attribut *clé primaire* dans les classes.

A.2 Transformations exogènes

associations-persons.
- *méta-modèle source :* Association
- *méta-modèle cible :* Persons
- *source :* original

– *description :* transformation partant d'un modèle d'associations composées de membres vers un modèle représentant des groupes de personnes.

Book2Publication.
– *méta-modèle source :* Book
– *méta-modèle cible :* Publication
– *source :* zoo ATL
– *description :* transformation partant d'une représentation de livres où chaque chapitre est détaillé vers une représentation plus condensée regroupant en une entrée les informations d'un livre.

Ecore2Class.
– *méta-modèle source :* Ecore
– *méta-modèle cible :* Class
– *source :* zoo ATL (extrait de la transformation Class2Relational)
– *description :* transformation partant de modèles Ecore vers des modèles de diagramme de classes. Cette transformation se trouvait dans l'archive contenant la transformation Class2Relational que nous n'avons pas pu traiter pour des raisons techniques.

emf2km3.
– *méta-modèle source :* Ecore
– *méta-modèle cible :* KM3
– *source :* zoo ATL
– *description :* transformation partant de modèles Ecore vers des modèles KM3. Le langage KM3 est un langage de modélisation très similaire au langage défini par Ecore.

Families2Persons.
– *méta-modèle source :* Families
– *méta-modèle cible :* Persons
– *source :* zoo ATL (méta-modèles modifiés)
– *description :* transformation partant d'un modèle représentant une famille de personnes vers un modèle représentant les personnes sans tenir compte de leurs liens familiaux.

JavaSource2Table.
– *méta-modèle source :* JavaSource
– *méta-modèle cible :* Table
– *source :* zoo ATL

– *description :* transformation créant à partir de code Java un tableau de statistiques permettant de voir quelles sont les méthodes invoquées dans chaque définition de méthode.

KM32EMF.
– *méta-modèle source :* KM3
– *méta-modèle cible :* Ecore
– *source :* zoo ATL
– *description :* transformation partant de modèles KM3 vers des modèles Ecore. Il s'agit de la transformation inverse à *emf2km3*.

KM32Problem.
– *méta-modèle source :* KM3
– *méta-modèle cible :* Problem
– *source :* zoo ATL
– *description :* transformation ayant pour but de détecter des situations problématiques dans les modèles KM3. Le résultat est un rapport d'erreur sous forme d'un modèle conforme à un méta-modèle Problem.

uml-er.
– *méta-modèle source :* extrait UML
– *méta-modèle cible :* extrait Entité-Association
– *source :* original
– *description :* transformation d'un modèle de classes UML vers un modèle Entité-Association. Les modèles d'exemples présentés font intervenir des powertype et GeneralizationSet rendant la transformation plus complexe.

uml-er2.
– *méta-modèle source :* extrait UML
– *méta-modèle cible :* extrait Entité-Association
– *source :* original
– *description :* transformation similaire à la précédente mais les exemples présentés sont plus petits.

uml-er3.
– *méta-modèle source :* extrait UML
– *méta-modèle cible :* extrait Entité-Association
– *source :* original
– *description :* transformation similaire aux deux précédentes mais sans powertype et GeneralizationSet.

uml-er-iccs.
- *méta-modèle source :* extrait UML
- *méta-modèle cible :* extrait Entité-Association
- *source :* original inspiré des articles de Varro et du MTIP05
- *description :* transformation d'un modèle de classes UML vers un modèle Entité-Association. Les méta-modèles et les exemples sont différents des transformations précédentes et présentent moins de particularités du langage UML.

UML2ClassDiagramToKM3.
- *méta-modèle source :* UML
- *méta-modèle cible :* KM3
- *source :* zoo ATL
- *description :* transformation d'un diagramme de classes UML vers un modèle KM3.

Annexe B

Correction de problèmes de généralisation dans les diagrammes de cas d'utilisation UML

Les travaux présentés dans cette annexe concernent l'utilisation de l'ARC pour appliquer des patrons de refactorisation de manière globale sur les diagrammes de cas d'utilisation UML dans le but d'améliorer leur lisibilité. Les différents patrons portent essentiellement sur des défauts de généralisation des diagrammes et sont appliqués de telle sorte que la sémantique des diagrammes visés reste inchangée. Une première spécification de ces travaux a été présentée dans [Dolques *et al.*, 2010c] et la version présentée ici provient de [Dolques *et al.*, 2010a]

Fixing generalization defects in UML use case diagrams

Xavier Dolques, Marianne Huchard, Clémentine Nebut, and Philippe Reitz

LIRMM, CNRS and Université Montpellier II, Montpellier
{dolques,huchard,nebut,reitz}@lirmm.fr

Abstract. Use case diagrams appear in early steps of a UML-based development. They capture user requirements, structured by the concepts of use cases and actors. An admitted good modeling practice is to design simple, easy-to-read use case diagrams. That can be achieved by introducing relevant generalizations of actors and use cases. The paper proposes an approach based on Formal Concept Analysis and one of its variants, Relational Concept Analysis, to refactor a use case diagram as a whole in order to make it clearer. We developed a tool and studied the results on about twenty examples, concluding on the relevancy of the approach on our benchmark.

Keywords: Formal Concept Analysis, UML use case diagrams

1 Introduction

Within a UML-based development, the very first model to be designed is the use case diagram, that is later used all over the modeling process. For example, in the Rational Unified Process [11], the development process is driven by the use cases: the use cases are continuously used to inject further functionalities in static models, they guide the testing plan, etc. Thus even if the use case diagram is probably the furthest one from implementation, it takes a large place in the design of many artifacts. Use case diagrams model the boundary of a system, the actors that interact with the system, and the use cases (main functionalities offered to actors by the system). Use cases are linked to the actors interacting with them, and use cases may be organized using three types of relation: inclusion, extension (that can be seen as conditional inclusion), and generalization.

As mentioned in the UML user guide [2], "Organizing your use cases by extracting common behavior (through include relationships) and distinguishing variants (through extend relationships) is an important part of creating a simple, balanced, and understandable set of use cases for your system." A good practice for use case diagrams is to make them simple to read [1], so as to catch at first glance the boundaries of the system, its main actors and main functionalities. Thus a tangled use case diagram (*e.g.* with numerous and tangled relations linking use cases to actors and to other use cases) will regularly brake the development since a designer, each time he or she will consult it, will have a delay to understand (again) the use case diagram.

In this paper, we investigate an approach to make use case diagrams clearer by introducing generalized actors and use cases to factorize relations. Our approach is based on several refactoring patterns (*e.g.* when two use cases include the same third one, they can be generalized by a new use case including this third use case), that are globally combined using Formal Concept Analysis (FCA) and one of its variant Relational Concept Analysis (RCA). The approach is implemented in an Eclipse-integrated tool that takes as input a UML diagram, encodes it into FCA (or RCA) contexts, generates the corresponding concept lattices, and finally produces as output the refactored use case diagram. We studied the results of this approach on several examples, comparing the pairs of input diagram/output diagram, and FCA versus RCA.

The following of the paper is structured as follows. We first illustrate in Section 2 the use case diagram refactoring with an example that will later be used to illustrate our approach. Section 3 details the used refactoring patterns, and Section 4 explains how they are globally applied using FCA or RCA. Results of our experiments on examples are given in Section 5. Section 6 compares our approach w.r.t. related work.

2 A motivating example

This section presents a motivating example. We briefly recall the main elements involved in UML use case diagrams, and then illustrate the use case generalization defects. The application we are interested in should allow various users to buy products or apply for interships. Figure 1 shows a use case diagram for this application. Actors represent the role(s) played by external persons or systems

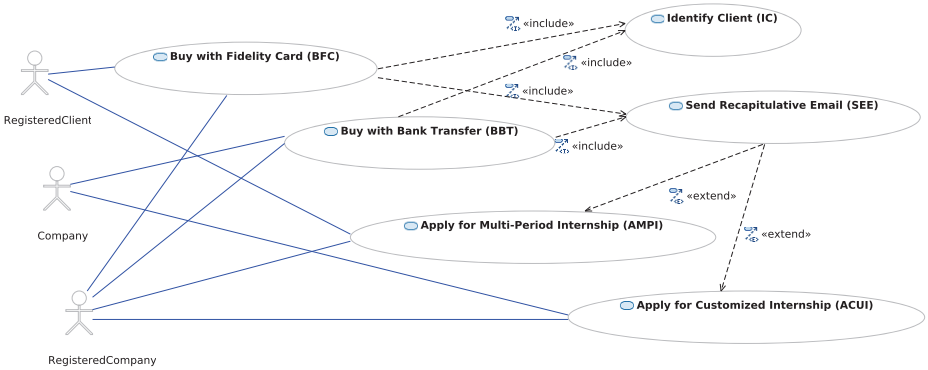


Fig. 1. A use case diagram for the internship example

that interact with the modelled system. They are depicted using a "stick man" icon (or a box stereotyped <<actor>> with the name of the actor). Here, three roles are identified: a company, a registered client, and a registered company.

Use cases, depicted by named ellipses, model large functionalities. In our example, we find the use case `Apply for Multi-Period Internship`, modelling the functionality corresponding to the possibility to apply for internships organized over several periods (for example a day per month). Use cases are associated (graphically by a continuous line) to the actors involved in them: actors to which the functionality is dedicated or secondary actors that interact with the system during the achievement of the functionality. In our example, the use case `Apply for Multi-Period Internship` is linked to the `Registered Client` and the `Registered Company` actors, since both of those actors can apply for such internships (whereas non-registered companies cannot). A use case may include another one. Graphically, it is shown by a dashed oriented arrow stereotyped by <<include>>. Semantically, that means that the including use case explicitly incorporates the behaviour of the included use case. For example, the use case `Buy with Bank Transfer` includes the use case `Send Recapitulative Email`. A use case may be extended by another one. Graphically, it is shown by a dashed oriented arrow stereotyped by <<extend>>. Semantically, that means that the extended use case *may* include the behaviour of the extending use case. In our example, a recapitulative email may be sent while realizing an appliance for a multi-period internship. Conditions for the extension and extension points can complete the extension relationship, as discussed later in the paper. UML provides two generalization mechanisms in use case diagrams: for actors and for use cases. Figure 2 shows a refactored use case diagram for the internship example, in which those two mechanisms appear. A use case may generalize another one: the

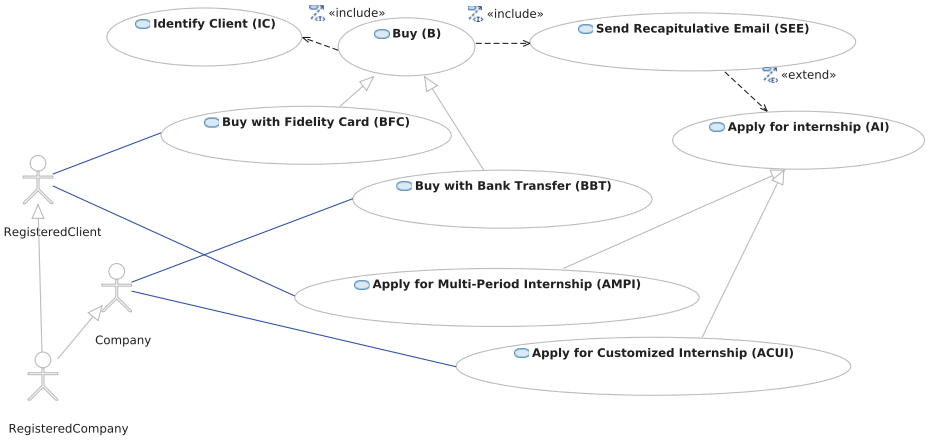


Fig. 2. A refactored use case diagram for the internship application using FCA

child use case inherits the behaviour of the parent use case, as well as its meaning. It is graphically represented by an edge ended by a triangle, like for class inheritance. In the refactored example of Figure 2, the use case `Buy` generalizes the use cases `Buy with Bank Transfer` and `Buy with Fidelity Card`. An actor may

generalize another one: the child actor inherits the roles of the parent actor, *i.e.* its interactions with use cases. In Figure 2, the actor `RegisteredCompany` specializes the actor `Company`.

The initial diagram of Figure 1 suffers from the large number of crossed links from actors to use cases, and to a lesser extent, from the large number of links from use cases to use cases. The actor `RegisteredCompany` is linked to four use cases. Introducing a generalization from this actor to the actors `Company` and `RegisteredClient` is semantically correct and factorizes the four links: in the refactored diagram proposed in Figure 2, the actor `RegisteredCompany` inherits its links to the use cases from its two parent actors. Similarly, the two inclusions of the use case `Client Identification` (resp. `Send Recapitulative Email`) by the use cases `Buy with Fidelity Card` and `Buy with Bank Transfer` can be factorized introducing the sound use case `Buy`. Last, the two extensions can be factorized introducing the sound `Apply For Internship` use case.

The approach we propose aims at detecting the possible relevant generalizations in a use case diagram, and introduces them in order to obtain a simpler use case diagram. For that, we first identify (in refactoring patterns) situations for which introducing a generalization may be useful. Then to automatically apply these patterns, we use Formal Concept Analysis (FCA) or one of its variants Relational Concept Analysis (RCA). The refactoring patterns are presented in the next section while the FCA/RCA application is detailed in Section 4.

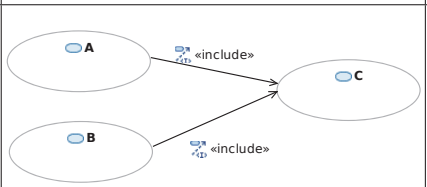
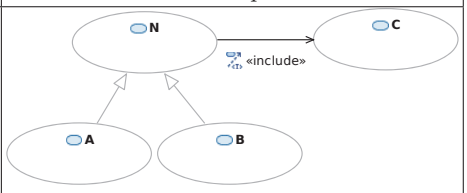
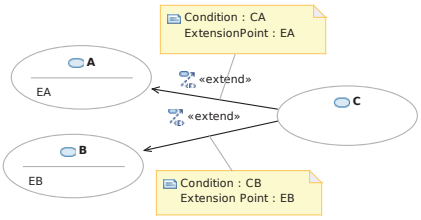
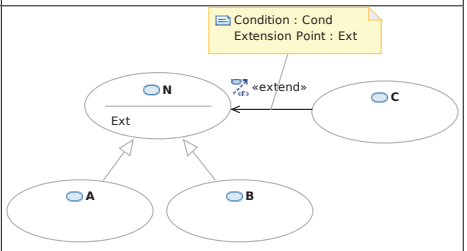
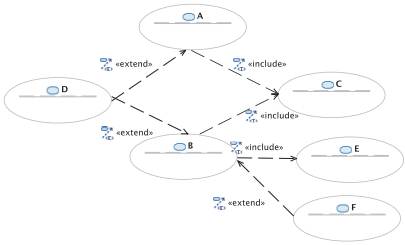
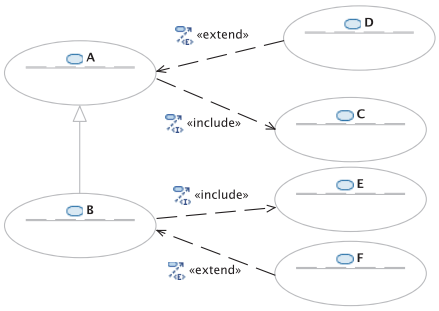
3 Local refactoring patterns

To correct generalization defaults, we propose a systematic approach that introduces more general elements (actors or use cases) based on the detection of shared relations. However, this does not guarantee the relevance of factorizing the relations using more general use cases or actors. In this section, we study several refactoring proposals. We consider here a simple set-based semantics, associating to a use case diagram the set of its actors and for each actor the feasible sequences of leaf use cases. The use cases having specializations are supposed to be abstract, *i.e.* replaced by one of their specializations during a concrete usage of the system.

Refactorings between use cases. Table 1 presents the refactoring patterns for use cases. The pattern I considers the situation of two use cases `A` and `B` including the same use case `C`. In this situation, all the sequences that can be performed by an actor and that contain `A` (resp. `B`) will also contain `C`. We propose to refactor the two inclusions as seen in the refactored pattern: a sequence that can be performed by an actor and containing `A` will still contain `C`, since `A` specializes `N` that contains `C`. In situation II, when the two conditions of extension `CA` and `CB` are identical and that the extension points (that roughly specify when the extension should occur) are also identical, then a generalization can be introduced. It frequently occurs that extension points and conditions are not specified for the extensions, in this case we propose to apply the refactoring, whereas if the extension conditions or the extension points are different, this refactoring does

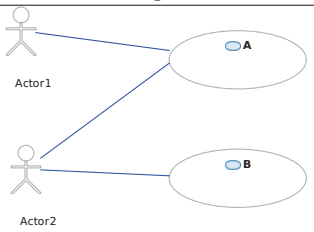
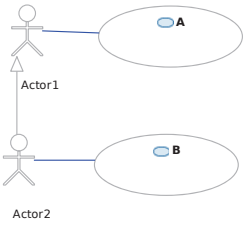
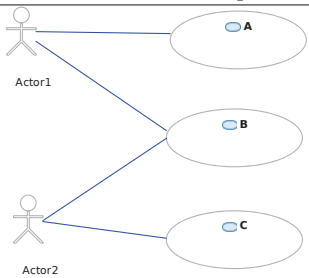
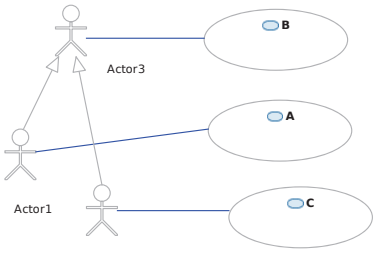
not apply. The situation III introduces a specialization refactoring. The source pattern owns a use case `A` for which the sets of outgoing `include` relations and incoming `extend` relations are strictly included in those of another use case `B`. The proposed refactoring derives `B` from `A` and removes from `B` the inherited relations. Situations to factorize incoming include relations or outgoing extend relations were studied but rejected since they were not pertinent.

Table 1. Refactoring patterns between use cases

	source pattern	refactored pattern
I	 outgoing include	
II	 incoming extend	
III	 shared extend and include	

Refactorings between actors. Table 2 presents two refactoring patterns between use cases and actors. The first one (I) introduces a specialization relation between actors when the set of associations of one actor is strictly included in the associations of another actor, and then removes the inherited associations. The second one (II) adds an abstract actor when the sets of associations of two actors have a non-empty intersection but each actor has associations that the other one does not have.

Table 2. Refactoring patterns between actors and use cases

	source pattern	refactored pattern
I	 Actors and use cases with specialization	
II	 Actors and use cases with factoring	

4 Global refactoring through Formal Concept Analysis

The refactoring schemas presented in the previous section do not give a unique solution to deal with a use case diagram as a whole. For that, we use Formal Concept Analysis [7] which systematically groups objects owning common characteristics in a minimal generalization structure.

Refactoring with FCA. We build two formal contexts (Table 3). In one context, we associate an actor with a use case when the actor is involved in the use case. In the second context, use cases are described by inclusion of another use case, extension by another use case and names. Figures 3 and 4 present the lattices. In the lattice of actors (Fig. 3), a generalization relationship is introduced: `C2`, which represents `RegisteredCompany`, is indeed a subconcept of `C1` (`RegisteredClient`) and `C3` (`Company`). This is an application of the first actor-use case refactoring. The lattice of use cases (Fig. 4) highlights two refactorings. The outgoing-include refactoring appears on the left: `C5` generalizes the concepts that introduce `Buy with Fidelity Card` (`C8`) and `Buy with Bank Transfer` (`C11`) because these two use cases include `Send Recapitulative Email` and `Identify Client`. `C7` shows an opportunity to apply the incoming extend refactoring (II): it factorizes the characteristic of being extended by `Send Recapitulative Email`, which is shared by `Apply for Multi-Period Internship` (`C10`) and `Apply for Customized Internship` (`C12`). The final use case diagram (Fig. 2) is deduced from the two lattices. The (non-trivial) concepts are interpreted as actors or use cases and the lattice partial order as generalization/specialization as illustrated just before. In our example, no new actor is created, but it hap-

Table 3. Contexts for FCA refactoring (use case names have been shorten)

	IsInvolvedIn			
	BFC	AMPI	BBT	ACUI
RegisteredClient	×	×		
Company			×	×
RegisteredCompany	×	×	×	×

	Includes		IsExtendedBy	Name					
	SEE	IC	SEE	BFC	SEE	AMPI	BBT	ACUI	IC
BFC	×	×		×					
SEE					×				
AMPI			×			×			
BBT	×	×					×		
ACUI			×					×	
IC									×

pens in other diagrams of our benchmark. Two new use case factorizations are introduced: `Buy` (from `C5`) and `Apply for internship` (from `C7`) that appear in the final diagram and increase its abstraction level and reusability.

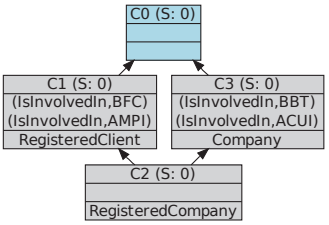


Fig. 3. The lattice of actors built using FCA

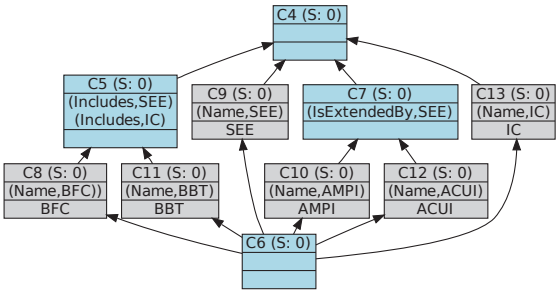


Fig. 4. The lattice of use cases built using FCA

Refining the refactoring with RCA For further refining of the result, we applied Relational Concept Analysis (RCA) [9], one of the extensions of Formal Concept Analysis that takes into account relationships between formal objects in the classification process. The input of Relational Concept Analysis is a set of tables (a relational context family or RCF) such that some tables represent object-attribute relations and some tables capture object-object relations. In the encoding we choose (Table 4), the non-relational part of the RCF is composed of two object-attribute tables: actors with no description (empty attribute set), and use cases described by their names. The relational part of the RCF is composed of three object-object tables: `Includes` relation, `IsExtendedBy` relation and `IsInvolvedIn` relation. The choice of these relations is guided by the refactoring patterns that we have identified as relevant, and by preliminary experiments that highlighted cases where combinatorial explosion occurred.

One lattice is built for each object-attribute table, in our case a lattice for actors, and a lattice for use cases. Those two lattices integrate the attributes and the relations. RCA iterates on two successive steps: building of lattices with classical FCA framework (on each object-attribute table concatenated with the

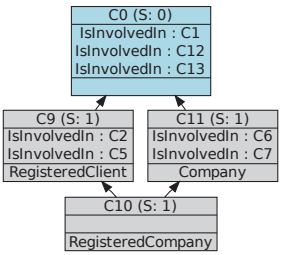


Fig. 5. The lattice of actors built using RCA

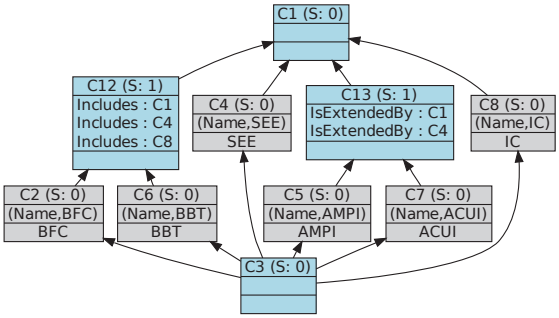


Fig. 6. The lattice of use cases built using RCA

corresponding object-object tables) and integration of the concepts discovered at the current iteration in the relational part (to be used at the next iteration). The integration of the concepts discovered at the current iteration uses scaling operators, here the existential operator has been used. For the integration, an object-object relation (*e.g.* `Includes`) is transformed by replacing the objects which are in the columns by the concepts of the lattice built at the current iteration on these objects. Then a relation is established in the new, existentially scaled, table, between an object and a concept, when the object is in relation with at least one object in the extent of the concept. For example, `RegisteredClient IsInvolvedIn BFC` in the original relation. Then, when at an iteration `BFC` is in the extent of a concept, say `C12`, in the scaled `IsInvolvedIn` relation, `RegisteredClient` is considered as participating in one of the use cases grouped by `C12`. A pair (`RegisteredClient`, `C12`) is added in the scaled table `IsInvolvedIn`. This pair enriches the description of the three actors and at the next FCA step, it is factored out in `C0`, top of the actor lattice. That highlights the opportunity to define a more generalized actor, this opportunity did not appear in the one-step FCA building.

Table 4. Relational Context Family for RCA refactoring

use case	Name BFC	SEE	AMPI	BBT	ACUI	IC
BFC	×					
SEE		×				
AMPI			×			
BBT				×		
ACUI					×	
IC						×

Includes	BFC	SEE	AMPI	BBT	ACUI	IC
BFC		×				×
SEE						
AMPI						
BBT		×				×
ACUI						
IC						

IsExtendedBy	BFC	SEE	AMPI	BBT	ACUI	IC
BFC						
SEE						
AMPI		×				
BBT						
ACUI		×				
IC						

IsInvolvedIn	BFC	SEE	AMPI	BBT	ACUI	IC
RegisteredClient	×		×			
Company				×	×	
RegisteredCompany	×		×	×	×	

actor
RegisteredClient
Company
RegisteredCompany

At each step, new concepts can appear, as well as new pairs relating these new concepts. The process iterates until no new concept or description emerge. Figures 5 and 6 show the final lattices. Lattices are interpreted as in the FCA

analysis, adding an interpretation of the references between the two lattices. The result of RCA improves the result of FCA on this example by adding the new, more general actor (Fig. 7). Furthermore, the interpretation of the concepts is used to interpret the references between lattices. For example, the characteristic `isInvolvedIn:C12` which appears in `C0` of the actor lattice signals that the actor which will be created to interpret `C0` will participate in the use case resulting from the interpretation of the use case `C12`, that is the general use case `Buy`.

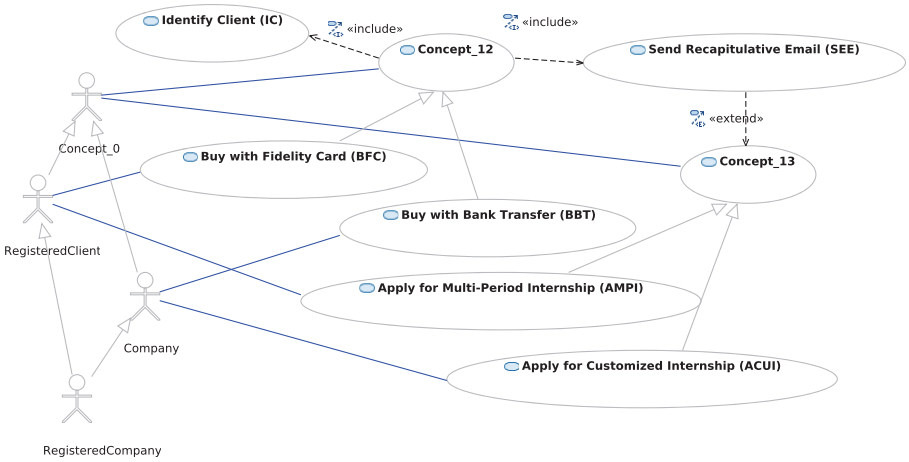


Fig. 7. The internship example refactored using RCA

5 Case Study

We present in this section the current implementation of our approach and the results obtained by running it on different kinds of data.

The Eclipse Modelling Framework (EMF) is a facility from the Eclipse IDE to implement modelling languages and generate tools to manipulate instances of those languages from programs. We use in our tool two Eclipse plugins based on EMF: UML2[1] and eRCA[2]. UML2 is an implementation of UML and therefore allows us to load, modify and save use case diagrams. eRCA is an implementation of FCA and RCA algorithms which uses EMF to implement contexts and lattices. We have developed a tool implementing both the FCA and RCA approaches for use case refactoring. For both of them we developed the same three steps described in Section 4: first the tool loads a use case diagram using UML2 plugin and encodes it as contexts in eRCA format; second it uses the eRCA tool

[1] http://www.eclipse.org/uml2/
[2] http://code.google.com/p/erca/

to generate lattices in eRCA format; finally lattices are automatically analyzed to create back a use case diagram in the UML2 plugin format and to save it.

We present here the results obtained using data gathered from different sources such as student projects or UML courses taken from the internet[3]. We encoded these diagrams using the UML2 plugin in Eclipse. Then we applied our two processes in all of them and automatically processed the results to compare them with the original diagrams. We computed the density of the diagram which is, if we consider the diagram as a graph $G = (V, E)$, the number of edges over the number of possible edges $|E|/|V|^2$. We assume that the lower density we have the clearer the diagram is. Of course, this is just indicative. We present in Figure 8 how the density can be changed by our approach on all our data. The last five diagrams considered (`test1` to `test5`) are used to test specific situations. We see that in most of the diagrams the density is lowered by using FCA and RCA, but with better results for FCA. Density goes down due to the creation of new use cases or actors and to the factorization of the relationships. An inverse situation can occur if the number of generalization links created is higher than the number of links removed by factorization. This is why FCA seems to be better here than RCA, because RCA creates more concepts. Nevertheless, the density for RCA usually remains better than the original density. We also see that density is not improved for the diagram `Invoice`. Indeed, we assume that each actor involved in a use case has the same role, so they can be factored into the same concept. But in the case of this diagram, many actors are involved in the same use cases but with different roles. This may be allowed depending on the interpretation of the UML specification, but results in the merging of many actors into one. By removing those actors and thus not adding new abstractions among actors, the density is then increased. This can be partly solved by adding an identifier for each actor when creating the contexts, but this prevents us to find new generalizations of actors since none of them would share in their intent the same identifiers.

6 Related Work

Literature dealing with designing use cases is prolific, but usually concentrates on textual use cases [13, 3], and not on UML use case diagrams. It is generally admitted that use cases should be described by quite a lot of information that does not appear in use case diagrams, such as the nominal and exceptional flows of events or scenarios corresponding to a use case, and many research activities studied how to structure each use case, in particular with dedicated templates [3] or natural language techniques [6]. Fewer work [14, 10] deal with the way to structure the relations linking uses cases to actors, or use cases to other use cases. In [14] a metamodel is proposed for use cases, extending and grouping existing metamodels, and detailing relations between use cases: composition, dependency, precedence, extension, generalization, etc. Our work focuses on the relations introduced in the use case part of the UML specification, however it should be

[3] all data available at http://www.lirmm.fr/%7edolques/publications/data/cla10

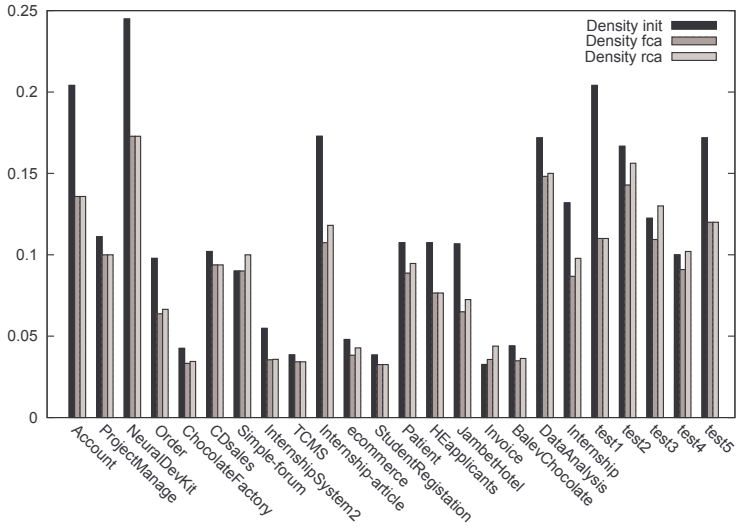


Fig. 8. Experimental results on the evolution of the density

interesting to investigate how use case models conform to the metamodel of [14] could be refactored. Issa [10] proposes refactoring the scenario part of a use case, for example by separating a use case into several use cases. Henderson-Sellers et al. [8] introduce metrics to compare various versions of a use case, mainly consisting of graph metrics.

FCA has already been applied to use cases [12, 5] but not with the objective to refactor use case diagrams. In [12] use cases written in a controlled natural language are classified into a lattice using the terms contained in the use cases, with a visualization objective. In [5] use cases are classified using their important terms, in order to detect main notions that can become classes. A preliminary version of this work can be found in the french-written reference [4]. The new contributions of this paper consist in the comparison of results obtained with RCA and FCA (while reference [4] focuses only on FCA), in the case study, and in a new tool integrating RCA and FCA.

7 Conclusion

UML use case diagrams can be seen as the functional cornerstone of UML modeling. In this paper, we proposed an approach to refactor such use case diagrams in order to make them clearer. The principle is to use FCA or RCA in order to detect new abstractions that, when introduced in the diagram, can reduce its visual complexity. Using the tool we developed to implement this approach, we carried on experiments using a set of about 20 use case diagrams to measure its efficiency in terms of density of relations in the diagram. It results that both FCA and RCA processes decrease the density of realistic-size use case diagrams,

the density obtained with FCA being lower than with RCA. However, RCA discovers more abstract use cases or actors that can be relevant. To go further in this study, we are studying accurately other metrics, both quantitative as average degree, and qualitative, as precision.

Acknowledgments. Authors would like to thank L. M. Hakik for a preliminary study, K. Bouzroud, I. Dagha, H. El Assam, M. El Asri, and J. Ruiz-Simari for their help in the implementation of the current tool, and the anonymous reviewers for their suggestions on evaluation metrics.

References

1. Ambler, S.W.: The Object Primer: Agile Model-Driven Development with UML 2.0. Cambridge University Press, New York, NY, USA (2004)
2. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, chap. 16. Addison-Wesley Prof. (2005)
3. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley Professional (2000)
4. Dolques, X., Madiha Hakik, L., Huchard, M., Nebut, C., Reitz, P.: Correction des défauts de généralisation dans les diagrammes de cas d'utilisation UML. In: Proc. of LMO'10 (Langages et Modèles à Objets). pp. 51–66 (2010)
5. Düwel, S., Hesse, W.: Bridging the gap between use case analysis and class structure design by formal concept analysis. In: Proceedings of Modellierung 2000. pp. 27–40. Fölbach-Verlag (2000)
6. Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Application of linguistic techniques for use case analysis. In: IEEE International Conference on Requirements Engineering. p. 157. IEEE Computer Society, Los Alamitos, CA, USA (2002)
7. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer (1999)
8. Henderson-Sellers, B., Zowghi, D., Klemola, T., Parasuram, S.: Sizing use cases: How to create a standard metrical approach. In: OOIS. Lecture Notes in Computer Science, vol. 2425, pp. 409–421. Springer (2002)
9. Huchard, M., Hacene, M.R., Roume, C., Valtchev, P.: Relational concept discovery in structured datasets. Ann. Math. Artif. Intell. 49(1-4), 39–76 (2007)
10. Issa, A.: Utilising refactoring to restructure use-case models. In: Proc. of the World Congress on Engineering, 2007. pp. 523–527. LNCS (2007)
11. Kruchten, P.: The Rational Unified Process: An Introduction, Second Edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
12. Richards, D., Böttger, K., Aguilera, O.: A controlled language to assist conversion of use case descriptions into concept lattices. In: Australian Joint Conference on Artificial Intelligence. pp. 1–11. LNCS, Springer (2002)
13. Rolland, C., Achour, C.B.: Guiding the construction of textual use case specifications. Data Knowl. Eng. 25(1-2), 125–160 (1998)
14. Rui, K., Butler, G.: Refactoring use case models : The metamodel. In: ACSC'03. CRPIT, vol. 16, pp. 301–308. ACS (2003)

Abstract

Model transformation is a fundamental operation for Model Driven Engineering. It can be performed manually or automatically, but in the later cas the developper needs to master all the meta-models involved. Model Transformation generation from examples allows to create a model transformation based on source models examples and target models exemples. Working at the model level allows the use of concrete syntaxes defined for the meta-models so there is no more need for the developper to perfectly know them. We propose a method to generate model transformations from examples using Relational Concept Analysis (RCA) which provides a set of transformation rules ordered under the structure of a lattice. RCA is a classification method based on matching links between models to extract rules. Those matching are a common feature between the model transformation generation from examples methods, so we propose a method based on an ontology matching approach to generate them.

Keywords:

Résumé

La transformation de modèles est une opération fondamentale dans l'ingénierie dirigée par les modèles. Elle peut être manuelle ou automatisée, mais dans ce dernier cas elle nécessite de la part du développeur qui la conçoit la maîtrise des méta-modèles impliqués dans la transformation. La génération de transformations de modèles à partir d'exemples permet la création d'une transformation de modèle en se basant sur des exemples de modèles sources et cibles. Le fait de travailler au niveau modèle permet d'utiliser les syntaxes concrètes définies pour les méta-modèles et ne nécessite plus une maîtrise parfaite de ces derniers. Nous proposons une méthode de génération de transformations de modèles à partir d'exemples basée sur l'Analyse Relationnelle de Concepts (ARC) permettant d'obtenir un ensemble de règles de transformations ordonnées sous forme de treillis. L'ARC est une méthode de classification qui se base sur des liens de correspondances entre les modèles pour faire émerger des règles. Ces liens étant un problème commun à toute les méthodes de génération de transformation de modèles à partir d'exemples, nous proposons une méthode basée sur des méthodes d'alignement d'ontologie permettant de les générer.

Mots clefs :

LIRMM — 161, rue Ada — 34095 Montpellier cedex 5 — France