Réutilisation de composants logiciels pour l'outillage de DSML dans le contexte des MPSoC

Paola Vallejo

THÈSE / UNIVERSITÉ DE BRETAGNE OCCIDENTALE
sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE
Mention : Informatique
École Doctorale SICMA

présentée par

Paola Vallejo

Préparée au département informatique de
l'université de Brest
Laboratoire Lab-STICC

Réutilisation de composants logiciels pour l'outillage de DSML dans le contexte des MPSoC

Thèse soutenue le 15 décembre 2015
devant le jury composé de :

Jean-Michel BRUEL
Professeur, Université de Toulouse / *rapporteur*

Antoine BEUGNARD
Professeur, Télécom Bretagne / *rapporteur*

Steven DERRIEN
Professeur, Université de Rennes 1 / *examinateur*

Benoit COMBEMALE
Maître de conférences, Université de Rennes 1 / *examinateur*

Isabelle BORNE
Professeur, Université de Bretagne-Sud / président du jury

Jean-Philippe BABAU
Professeur, Université de Bretagne Occidentale / *directeur de thèse*

Mickaël KERBOEUF
Maître de conférences, Université de Bretagne Occidentale / encadrant

Kevin MARTIN
Maître de conférences, Université de Bretagne-Sud / encadrant

Remerciements

Un travail de thèse ne s'accomplit jamais de manière individuelle, mais grâce à l'aide et aux inestimables soutiens de plusieurs personnes. Chacune contribue à l'aboutissement de ce projet. C'est à toutes ces personnes à qui ces remerciements sont destinés.

Je tiens à remercier l'ensemble des membres du jury qui m'ont fait l'honneur de participer à la soutenance de thèse. Ils m'ont transmis une partie de leurs connaissances à travers leurs remarques. Je remercie particulièrement Jean-Michel BRUEL et Antoine BEUGNARD pour avoir accepté de rapporter ma thèse. J'ai beaucoup apprécié leurs conseils et leurs remarques inestimables. Je remercie également Steven DERRIEN, Benoit COMBEMALE et Isabelle BORNE pour avoir accepté d'évaluer mon travail.

Un grand merci à mon directeur de thèse Jean-Philippe BABAU pour ses multiples conseils, pour la gentillesse et la patience qu'il a manifestées à mon égard durant cette thèse.

Je tiens à exprimer ma gratitude et remerciements spéciaux à mes encadrants. Leurs remarques ont été primordiaux durant le déroulement de cette thèse. Merci pour leur aide et soutien illimité tout au long de ces trois années. Je remercie Mickaël KERBOEUF pour m'avoir supporté et encouragé. Je remercie Kevin MARTIN pour sa collaboration et pour ses conseils.

Je tiens aussi à mentionner le plaisir que j'ai eu à travailler au sein du Lab-STICC, et j'en remercie tous mes collègues.

Enfin, mais ce qui n'est pas le moins important, je remercie ma famille et mes amis, toujours présents (malgré la distance) pour leur soutien inconditionnel.

Résumé

La conception d'un langage de modélisation spécifique à un domaine (*DSML*) implique la conception d'un outillage dédié qui met en œuvre des fonctionnalités de traitement et d'analyse pour ce langage. Dans bien des cas, les fonctionnalités à mettre en œuvre existent déjà, mais elles s'appliquent à des portions ou à des variantes du DSML que le concepteur manipule. Réutiliser ces fonctionnalités existantes est un moyen de simplifier la production de l'outillage d'un nouveau DSML. La réutilisation implique que les données du DSML soient adaptées afin de les rendre valides du point de vue de la fonctionnalité à réutiliser.

Si l'adaptation est faite et les données sont placées dans le contexte de la fonctionnalité, elle peut être réutilisée. Le résultat produit par l'outil reste dans le contexte de l'outil et il doit être adapté afin de le placer dans le contexte du DSML (migration inverse).

Dans ce cadre, la réutilisation n'a de sens que si les deux adaptations de données sont peu coûteuses. L'objectif de cette thèse est de proposer un mécanisme qui intègre la migration, la réutilisation et la migration inverse. La principale contribution est une approche qui facilite la réutilisation de fonctionnalités existantes via des migrations de modèles. Cette approche facilite la production de l'outillage d'un DSML. Elle permet de faire des migrations réversibles entre deux DSMLs sémantiquement proches. L'utilisateur est guidé lors du processus de réutilisation pour fournir rapidement l'outillage complet et efficace d'un DSML.

L'approche a été formalisée et appliquée à un DSML (Orcc) dans le contexte des compilateurs pour les systèmes multiprocesseur intégrés sur puce (*MPSoC*).

Mots-clés

Co-évolution, Langage de modélisation spécifique à un domaine (DSML), Migration de modèles, Réutilisation.

Abstract

Designers of domain specific modeling languages (*DSML*) must provide all the tooling of these languages. In many cases, the features to be developped already exist, but it applies to portions or variants of the DSML. One way to simplify the implementation of these features is by reusing the existing functionalities. Reuse means that DSML data must be adapted to be valid according to the functionality to be reused.

If the adaptation is done and the data are placed in the context of the functionality, it can be reused. The result produced by the tool remains in the context of the tool and it must be adapted to be placed in the context of the DSML (reverse migration).

In this context, reuse makes sense only if the migration and the reverse migration are not very expensive. The main objective of this thesis is to provide a mechanism to integrate the migration, the reuse and the reverse migration and apply them efficiently. The main contribution is an approach that facilitates the reuse of existing functionalities by means of model migrations. This approach facilitates the production of the tooling for a DSML. It allows reversible migration between two DSMLs semantically close. The user is guided during the reuse process to quickly provide the tooling of his DSML.

The approach has been formalised et applied to a DSML (Orcc) in the context of the compilers for multiprocessor System-on-Chip (*MPSoC*).

Keywords

Coevolution, Domain Specific Modeling Language (DSML), Model migration, Reuse.

Sommaire

Liste des algorithmes

Liste des tableaux

Liste des figures

Liste des listings

Introduction générale

Contexte

La complexité des systèmes informatiques est en constante augmentation. Dans ce contexte, un des défis pour l'outillage de logiciels dédiés est la proposition de solutions efficaces pour prendre en compte les changements de l'environnement. Refaire un logiciel à chaque fois qu'un changement est fait devient coûteux, prend beaucoup de temps et est sujet à erreurs. Une des solutions qui permet de gérer cette complexité est de l'aborder par la *modélisation*. Cette approche a pour première vertu de fournir une représentation simplifiée d'un aspect de la réalité pour un objectif spécifique.

L'Ingénierie Dirigée par les Modèles (IDM) [Kad05] propose des outils, des concepts et des langages pour la création et la transformation de modèles. Elle promeut l'usage intensif de modèles pour implémenter les principes de l'ingénierie du logiciel. Pour les activités de modélisation pour l'informatique, un langage de modélisation généraliste (GPML), en anglais *General Purpose Modeling Language*, comme UML [UML14], fournit beaucoup de concepts de modélisation riches. Si UML répond aux besoins de modélisation du logiciel, pour les considérations spécifiques il doit être étendu via la création de *profiles*. Une autre approche pour la prise en compte de considérations spécifiques est la mise en place d'un langage de modélisation spécifique à un domaine (DSML), en anglais *Domain Specific Modeling Language*. L'intérêt des DSMLs réside dans leur simplicité et leur relation forte à un domaine spécifique. Lors de la conception de DSMLs, on s'inspire généralement d'autres DSMLs du même domaine. En effet, la plupart d'entre eux partagent de nombreux concepts communs parce qu'ils expriment des besoins similaires sur un champ sémantique identique [AD09]. Ils ne varient que pour un ensemble limité de concepts et des considérations structurelles (comme le renommage de concepts). Un ensemble de DSMLs proches avec des caractéristiques similaires constituent une *famille* de langages [OADFP08].

Une des préoccupations des concepteurs de DSMLs, est la production rapide de l'outillage de traitement et d'analyse pour leur langages. Ceci est particulièrement important dans le cas d'un langage (ou de manière plus restrictive d'un *méta-modèle*) spécifique pour lequel l'outillage complet doit être conçu (*i.e.* des outils d'analyse, des éditeurs dédiés, ou des générateurs de code). L'utilisation d'environnements de méta-modélisation tels qu'*Eclipse Modeling Framework* (EMF) [EMF15a] et *Kevoree Modeling Framework* (KMF) [FNM+14] est recommandée pour réduire le coût lié à l'obtention de ces facilités grâce à leurs capacités génératives. Ces environnements peuvent produire des éditeurs spécifiques dédiés à un domaine ou des navigateurs pour les instances du méta-modèle. Dans certains cas, ils peuvent également offrir un environnement pour des générateurs de code spécifiques au domaine [Pla15].

Malgré les facilités disponibles, un concepteur de DSML doit encore ajouter des fonctionnalités spécifiques au domaine. Il peut le faire en les spécifiant complètement, en les codant directement, ou en essayant de *réutiliser* des *portions* de code existantes. La réutilisation de code est une méthode de base pour réduire le coût d'obtention de tout l'outillage d'un DSML [FF95].

Problématiques

Les DSMLs couvrent généralement une vaste gamme de niveaux d'abstraction pour un domaine particulier. Par exemple, un DSML peut permettre de spécifier des abstractions de haut niveau pour l'interface avec les utilisateurs, ainsi que des abstractions de plus bas niveau pour gérer le stockage de données et des algorithmes spécifiques. L'intégration de plusieurs préoccupations, rend les DSMLs complexes.

Dans ce contexte, l'ajout d'une fonctionnalité implique la compréhension et la gestion de l'ensemble des concepts du DSML. De plus, afin de faciliter l'implémentation des outils, les concepteurs ont tendance à rajouter des caractéristiques supplémentaires (*i.e.* des attributs dérivés, des références opposées). Ces caractéristiques supplémentaires complexifient encore plus le DSML en ajoutant des spécificités de mise en œuvre. Au final, un DSML intègre des préoccupations inhérentes à un domaine spécifique, mais aussi des préoccupations de mise en œuvre qui le polluent. De ce fait, la compréhension et gestion des concepts du DSML deviennent de plus en plus complexes. Et l'ajout des nouvelles fonctionnalités est de plus en plus difficile.

Sachant qu'il y a souvent plusieurs DSMLs qui appartiennent à la même famille, il est fréquent de trouver des fonctionnalités dont le concepteur a besoin, déjà implantées sur d'autres DSMLs de la même famille. Si l'ajout d'une fonctionnalité est difficile et si elle existe déjà, le concepteur doit pouvoir la réutiliser. Un autre scénario classique de réutilisation consiste à trouver un DSML dédié à l'outil qui implémente des fonctionnalités génériques.

Concernant la réutilisation, nous listons deux façons de faire : par migration de la fonctionnalité et par migration de données. La migration de la fonctionnalité implique que l'outil à réutiliser est migré pour le placer dans le DSML du concepteur (la fonctionnalité est adaptée aux données). La fonctionnalité existante doit être adaptée pour qu'elle puisse gérer les concepts du DSML dans lequel elle a été placée. La figure 1 illustre la réutilisation d'une fonctionnalité Tool par migration de la fonctionnalité.

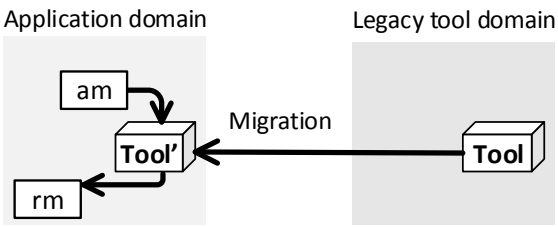


FIGURE 1 – Réutilisation par migration de la fonctionnalité

La migration de la fonctionnalité implique, d'un côté la compréhension et la manipulation de la fonctionnalité (le code source de la fonctionnalité est disponible et il peut être modifié). D'un autre côté, elle implique la compréhension du DSML afin

d'adapter l'outil en fonction de lui. De plus, de test de validation peuvent être requis pour vérifier que la migration n'a pas affecté le cœur de la fonctionnalité. L'effort nécessaire pour faire cette adaptation, n'est pas négligeable et il se rapproche de l'effort nécessaire pour mettre la fonctionnalité en œuvre directement dans le DSML. Puisque notre intérêt est de réduire l'effort lié à la mise en œuvre des fonctionnalités, dans cette thèse nous ne considérons pas cette façon de faire la réutilisation.

Dans la migration de données, la fonctionnalité à réutiliser reste inchangée, par contre, les données du DSML migrent pour devenir des données valides par rapport à la fonctionnalité (les données s'adaptent en fonction de la fonctionnalité). Dans ce cas, la migration consiste à extraire à partir du DSML, les données qui sont vraiment utiles à la fonctionnalité. La figure 2 illustre la migration de am pour réutiliser la fonctionnalité Tool existante. Dans cet exemple, la fonctionnalité a été effectivement réutilisée et elle produit le résultat rm. Mais ce résultat reste dans le contexte de la fonctionnalité et non pas dans le contexte du DSML dans lequel il est requis.

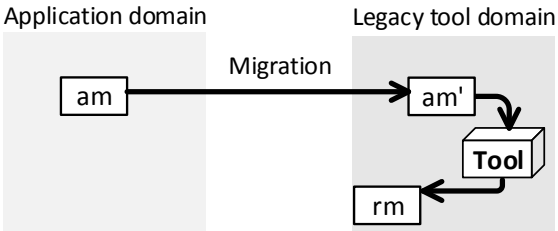


FIGURE 2 – Réutilisation par migration des données

La réutilisation par migration de données implique alors une migration supplémentaire pour migrer le résultat de la fonctionnalité et le placer dans le contexte du DSML dans lequel la fonctionnalité est réutilisée. La figure 3 illustre les deux migrations. La réutilisation implique des migrations de données vers la fonctionnalité et à partir de la fonctionnalité.

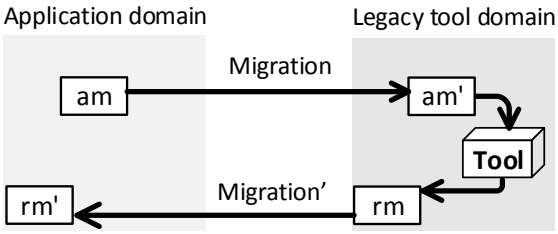


FIGURE 3 – Réutilisation par migration des données avec migration du résultat de l'outil

La réutilisation des fonctionnalités existantes facilite la production de l'outillage de DSML. Il est nécessaire d'assurer que ces migrations sont faites de manière adéquate et qu'elles ne demandent pas plus d'effort que de faire la réutilisation par migration de la fonctionnalité. Cela implique que Migration, Tool et Migration' produisent le même résultat que Tool' ($Tool' = Migration \circ Tool \circ Migration'$). Effectuer une seule migration (figure 1) pourrait paraitre plus simple qu'en faire deux, cependant ces deux migrations ne demandent pas de connaissances spécifiques par rapport à la fonctionnalité. Elles ne se concentrent que sur les données manipulées par l'outil. Dans ce contexte, l'objectif de cette thèse est de faciliter la réutilisation de fonctionnalités via

des migrations de données sans modifier le code de la fonctionnalité et en automatisant le processus de réutilisation.

Contributions

Pour répondre aux défis de la réutilisation par migration de données, nous proposons dans un premier temps de mettre les données du DSML au service de la fonctionnalité. Puis, le concepteur est guidé pour garantir que la réutilisation de la fonctionnalité dont il a besoin est effectuée. Les contributions de cette thèse sont :

La définition d'une approche pour faciliter la réutilisation de code. Elle intègre la migration des données d'un DSML et la migration du résultat de la fonctionnalité réutilisée. Elle fournit un langage de transformations dédié à la réutilisation de code, le langage qui s'appuie sur un méta-modèle appelé Modif permet de spécifier les différences entre deux langages sémantiquement proches. Le langage comporte un ensemble d'*opérateurs d'édition basiques* (rename, remove) et un ensemble d'*opérateurs d'édition complexes* (hide, flatten) qui s'appliquent à un méta-modèle (source). À partir des opérateurs appliqués, des *moteurs de migration* génériques permettent de mettre les données conformes au domaine spécifique dans le domaine d'application d'un outil existant. Dans le cas d'un outil de réécriture, ils permettent de mettre le résultat de l'outil dans le contexte spécifique du DSML.

L'outillage de l'approche. Les principes de l'approche pour faciliter la réutilisation sont développés et présentés dans un logiciel libre pour gérer la réutilisation d'outils. La majorité des étapes de la réutilisation sont automatisées. Celles qui ne sont pas automatisables parce qu'elles nécessitent une information dont l'utilisateur dispose, proposent des guides pour orienter l'utilisateur lors de la réutilisation.

La consolidation des fondements sémantiques de l'approche pour faciliter la réutilisation. L'approche propose un ensemble d'opérateurs de migration de modèles pour automatiser la réutilisation d'outils. La formalisation des migrations est basée sur une sémantique de graphes (un modèle est vu comme un graphe d'objets). Les opérateurs de migration deviennent des opérateurs de graphes. Dans le cadre d'une migration simple (*i.e.* par application d'opérateurs d'édition basiques), on peut prouver que les actions de l'outil ne sont pas défaites. Dans le cas d'une migration complexe, on met à jour les éléments du résultat de l'outil. La mise à jour est faite pour mettre le résultat de l'outil dans le contexte spécifique du DSML, mais ces modifications ne mettent pas en cause les actions de l'outil.

Plan

Cette thèse comprend trois parties. La première partie présente les travaux liés aux problématiques étudiées à travers un état de l'art sur la réutilisation d'outils, l'extraction et la migration de données.

La seconde partie présente le processus pour faciliter la réutilisation d'outils, et détaille ses objectifs. Les étapes qui composent le processus sont abordées et formalisées. Nous y détaillons leur comportement et les services offerts.

La troisième partie décrit la mise en œuvre du processus de réutilisation proposé par l'approche. Il en résulte un prototype de framework de réutilisation. Nous détaillons un scénario type d'utilisation de ce framework. Sur deux scénarios concrets (Mapviewer et Orcc) nous présentons l'utilisation de l'approche. Ces scénarios sont un moyen pour valider l'approche.

Enfin, une conclusion générale du travail présenté est dressée ainsi que des perspectives d'évolution par rapport aux contributions apportées.

Première partie

Contexte et état de l'art

Chapitre 1

Contexte

Sommaire

1.1 Introduction

Dans ce chapitre, nous présentons les concepts qui vont être employés tout au long du document.

Le sujet de la thèse porte sur l'automatisation du processus de réutilisation d'outils génériques pour des langages spécifiques, via la transformation de modèles. Nous traitons d'abord les concepts liés à la réutilisation (section 1.2). Puis, nous présentons le domaine de l'Ingénierie Dirigée par les Modèles (section 1.3). Ensuite, nous présentons les principes des langages de modélisation spécifiques à un domaine (section 1.4). Pour terminer, nous introduisons la notion d'évolution de méta-modèle et de co-évolution de modèles (section 1.5).

1.2 Réutilisation

Les principes de la réutilisation sont exposés avant de présenter les bénéfices qu'elle apporte. Puis le processus typique de réutilisation est présenté.

1.2.1 Principes

Dans cette thèse nos reprenons la définition de Prieto [PD89] : « La réutilisation consiste à utiliser des concepts ou des objets pré-existants dans une situation nouvelle, à apparier les situations nouvelles et anciennes, et à les adapter pour répondre à des nouvelles exigences ».

Dans le *domaine de l'informatique*, la réutilisation est le processus qui consiste à utiliser des artefacts existants au lieu de les construire à partir de zéro. Un artefact réutilisable, peut être n'importe quelle information dont un développeur peut avoir besoin dans le processus de création de logiciels [Fre83]. Dans cette thèse nous nous concentrons sur la réutilisation de *code source*, plus particulièrement d'algorithmes développés dans un contexte orienté-objet (méthodes de classes).

1.2.2 Les bénéfices de la réutilisation

Le développement de logiciel basé sur la réutilisation représente un certain nombre de bénéfices, dont les suivants [Cyb96], [Pae15] :

Des économies de temps et de coûts aboutissent à l'augmentation de la productivité. Les activités associées à la conception et à l'implémentation de logiciels sont remplacées par l'adaptation de logiciels. La réutilisation réduit le temps de développement et coûte donc moins. Radding [Rad99] montre dans une étude que si l'on réutilise des composants logiciels, le temps de développement des systèmes peut être réduit de 50 %. D'après une étude présentée dans [BBM96], le fait de réutiliser un composant (une classe) légèrement modifié ou *verbatim* (sans modifications) réduit la densité de défauts.

Un ensemble d'artefacts réutilisables peut souvent être considéré comme un langage de haut niveau basé sur les concepts issus d'un domaine donné. Ainsi, un développeur a la possibilité de travailler avec des notions plus abstraites liées directement

au problème et peut ignorer les détails de mise en œuvre. Le travail effectué à un niveau d'abstraction supérieur conduit à une augmentation de la productivité du développement [Cyb96].

Augmentation de la fiabilité. Par hypothèse, la durée de vie d'un artefact réutilisable est supérieure à celle de n'importe quel produit individuel. Ainsi, la fiabilité de cet artefact est également augmentée. Cela conduit à l'amélioration du système intégré de composants réutilisables plutôt que de ceux entièrement construits à partir de zéro. La réutilisation peut améliorer la qualité des systèmes parce qu'ils sont construits à partir d'artefacts existants qui ont été préalablement testés [Gra01], [Lim94] et bien documentés.

Augmentation de la facilité de maintenance. Les systèmes qui réutilisent des parties existantes sont, en général, plus simples et plus abstraits. Leur conception est proche du domaine du problème, ils peuvent facilement être modifiés pour faire face à l'évolution rapide des besoins. Cela a un impact positif sur la qualité de la maintenance de ces systèmes [Cyb96].

1.2.3 Le processus de réutilisation

La réutilisation n'a d'intérêt que si elle nécessite moins d'effort qu'un développement à partir de zéro [Kru89]. Cela arrive typiquement quand les artefacts à réutiliser concentrent une expertise métier difficile à reproduire. Ces artefacts sont paradoxalement, souvent réécrits afin de les rendre applicables aux données sémantiquement équivalentes, mais structurellement incompatibles. Dans ce contexte, le processus de réutilisation comprend généralement trois étapes [BBM96] :

1. La sélection d'un artefact réutilisable.
2. L'adaptation de l'artefact aux besoins de son application.
3. L'intégration de l'artefact adapté dans le logiciel en cours de développement.

Ce processus doit réduire le temps et l'effort humain nécessaires pour produire des produits logiciels. Cependant, il peut entrainer certains inconvénients. Parmi eux figurent la nécessité de comprendre le code et d'avoir les connaissances métiers permettant de faire les adaptations requises [Pae15], [Cyb96], [Tra88], [Hem92]. De plus, le processus de réutilisation peut devenir assez complexe et il n'existe pas d'outils d'aide qui permettent de gérer facilement cette complexité.

Dans cette thèse, nous explorons une solution alternative à cette forme de réutilisation. Elle consiste à ne pas modifier l'outil, mais à préciser son *contexte d'utilisation* via les migrations de modèles. On pourrait par exemple, prendre un outil qui fait le tri de nombres et le réutiliser pour trier des triangles ou des voitures, en adaptant le contexte en non pas l'outil. Cette thèse se situe clairement dans le domaine de l'IDM.

1.3 L'Ingénierie Dirigée par les Modèles

L'Ingénierie Dirigée par les Modèles (IDM) propose un usage intensif de modèles pour implémenter les principes du génie logiciel. Dans cette section nous présentons

les principes de l'IDM, (section 1.3.1). Ensuite, nous présentons les différents niveaux d'abstraction des modèles gérés par l'IDM (section 1.3.2). Ensuite, nous introduisons les notions de la transformation de modèles (section 1.3.3). Enfin, nous présentons une façon de mettre en œuvre les principes de l'IDM (section 1.3.4).

1.3.1 Principes

L'IDM propose un cadre de développement de logiciels avec pour objectif de découpler les spécificités métiers des contraintes technologiques liées à l'implantation de la solution [Kad05].

Parmi les défis de l'IDM, on peut trouver : concevoir une application en s'abstrayant des technologies ciblées ; assurer la pérennité des applications conçues en termes de maintenance et d'adaptation aux changements ; augmenter la productivité ; cibler plusieurs plateformes d'exécution à partir d'une seule conception ; automatiser la génération de code ; contrôler, simuler et tester le développement à différents niveaux d'abstraction.

Pour répondre à ces défis, l'IDM propose de modéliser les applications à un niveau plus abstrait que la programmation classique. Dans une approche IDM, tout est centré sur un élément fondamental : le *modèle* (il sera défini dans la section 1.3.2). Il est au cœur du processus de conception [BJRV04]. Ceci fait que les systèmes ne sont plus vus comme une suite de lignes de code, mais comme un ensemble de modèles.

L'IDM augmente la productivité et la qualité des systèmes, en augmentant le niveau d'abstraction auquel les ingénieurs travaillent. L'IDM concerne plusieurs aspects tels que les langages de modélisation dédiés, le développement de logiciel orienté par les modèles et la gestion de modèles. L'IDM permet d'éclaircir l'architecture logicielle, de maitriser efficacement la mise en œuvre et d'aboutir à une grande évolutivité et une bonne flexibilité des logiciels [BJRV04].

Pour obtenir ces résultats, l'IDM s'appuie sur la séparation des préoccupations via des modèles dédiés : chaque modèle est exprimé dans un langage de modélisation, spécifique à la préoccupation qu'il adresse. Cette approche entraîne la multiplication des langages de modélisation spécifiques, alors que leur définition et surtout leur outillage restent des tâches longues et coûteuses. Maîtriser la définition et l'outillage de ces langages est un enjeu fort de l'IDM.

Dans la section qui suit nous présentons les différents niveaux d'abstraction de l'IDM.

1.3.2 Modèles et méta-modèles

De façon générale, un modèle est une abstraction qui représente une partie de la réalité. Le modèle est construit à partir des éléments qui répondent à une syntaxe et à une sémantique qui forment un système de représentation des connaissances. Selon [JCV12], un modèle représente un système selon un certain point de vue, au niveau d'abstraction facilitant par exemple, la conception et la validation d'un aspect particulier du système.

Pour être pertinent, un modèle doit présenter les avantages suivants :

– Être abstrait : le modèle ne se concentre que sur les points importants et met de côté les détails qui ne sont pas nécessaires à l'aspect considéré.

– Être compréhensible : les aspects complexes doivent être présentés d'une façon simple.
– Être précis : le modèle représente fidèlement les aspects considérés du système modélisé.
– Être prédictif : il permet de faire des prévisions correctes sur le système modélisé.
– Être peu coûteux : il est moins coûteux de construire et d'analyser un modèle que de construire et d'analyser le système lui-même.

Pour aider à construire des modèles, l'OMG (*Object Management Group*) [OMG15] propose une spécification d'un framework de méta-modélisation, appelé MOF (*Meta-Object Facility*) [Met14]. MOF est un standard pour développer des systèmes orientés par les modèles. La spécification MOF adopte une architecture de méta-modélisation à quatre niveaux, tel que représenté sur la figure 1.1. D'autres discussions concernant les niveaux de méta-modélisation peuvent être trouvées dans [Béz04], [Kuh06] et [Sei03].

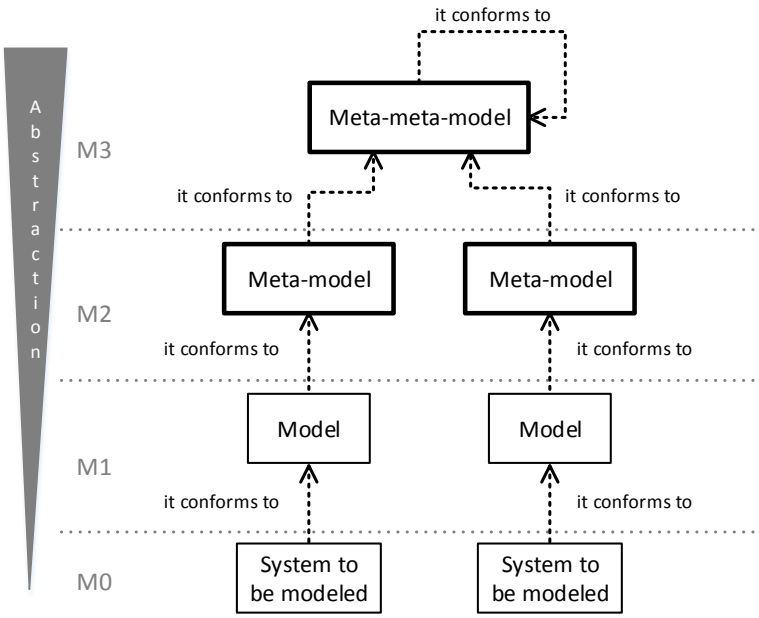


FIGURE 1.1 – Niveaux d'abstraction

M0 : Correspond aux éléments concrets modélisés.
M1 : Correspond aux modèles.
M2 : Un méta-modèle est un formalisme contenant tous les concepts nécessaires pour créer des modèles dans un domaine donné. Il décrit les types d'éléments contenus dans le modèle et la façon dont ils sont organisés et associés entre eux. Il constitue les langages par lesquels une réalité donnée peut être décrite au sens abstrait.
On peut trouver des méta-modèles généraux (GPML *General Purpose Modeling Language*) et des méta-modèles spécifiques (DSML *Domain Specific Modeling Language*). Les GPMLs ne sont pas liés à un domaine particulier *i.e.* UML (*Unified Modeling Language*), et fournissent des concepts riches d'usage générique. Ils expriment plusieurs concepts et propriétés en utilisant le même langage. En

particulier, ils permettent de représenter la structure et le comportement d'un système. Ils sont largement utilisés pour modéliser les systèmes complexes. Néanmoins, leur généricité peut devenir un inconvénient quand il s'agit d'exprimer des concepts très spécifiques d'un domaine particulier. Pour faire face à cette problématique deux solutions sont proposées : spécialiser UML via les *profiles* (ajout des contraintes et des concepts, *i.e.* le Profile UML pour MARTE : *Modeling and Analysis of Real-Time and Embedded Systems*[1]) ou s'appuyer sur un nouveau langage (qui implique la définition d'un nouveau méta-modèle).

M3 : Un méta-méta-modèle est utilisé pour décrire les méta-modèles. Un méta-méta-modèle a tous les concepts nécessaires pour se décrire lui-même. Quelques méta-méta-modèles (aussi connus sous le nom de méta-langages) les plus répandus et utilisés sont EMOF (*Essential MOF*) et Ecore [SBPM09].

Dans cette hiérarchie de modèles, la *conformité* établit la validité d'un modèle vis-à-vis du modèle de niveau supérieur. Un modèle est conforme à un méta-modèle, un méta-modèle est conforme à son méta-méta-modèle et un méta-méta-modèle est conforme à lui-même (il est auto-descriptif).

La hiérarchie à 4 niveaux existe en dehors de MOF, dans d'autres espaces technologiques que celui de l'OMG. Par exemple : XML (*eXtensible Markup Language*) [XML15], EBNF (*Extended Backus–Naur Form*) une méta-notation qui permet de définir des langages de programmation textuels [XG03], lamda calcul [Man09], etc.

Dans le contexte de l'IDM, les modèles sont manipulés par des *transformations de modèles*. Elles jouent un rôle important dans l'IDM [SK03] en permettant de relier les modèles entre eux, en explicitant le passage d'un contexte à l'autre dans le même niveau d'abstraction.

Le passage de modèle contemplatif à modèle productif [Bé05], est un passage d'une représentation fixe d'un système (sans interaction directe avec le cycle de développement logiciel), vers une représentation qui envisage un système composé de modèles utilisés directement comme source de données dans le processus de développement. Une des conséquences de ce passage est la génération semi-automatique ou automatique de code. Dans ce contexte, les modèles sont rendus *productifs* en automatisant la génération de code et de modèles. Pour qu'un modèle soit productif, il doit pouvoir être manipulé par une machine. Le langage dans lequel ce modèle est exprimé doit donc être explicitement défini [JCV12].

La section suivante, donne un aperçu sur les transformations de modèles.

1.3.3 Transformation de modèles

D'après Kleppe, *une transformation est la génération automatique d'un modèle cible à partir d'un modèle source, selon une définition de transformation. La définition de la transformation est un ensemble de règles, qui ensemble, décrivent comment un modèle dans le langage source peut être transformé en un modèle dans le langage cible* [KWB03]. Un *moteur de transformation* interprète la définition de la transformation et l'applique. Une transformation de modèle est elle-même un modèle qui est conforme à un méta-modèle de transformation.

1. http://www.omgmarte.org/

Selon Czarnecki [CH06], les transformations sont utilisées dans différents scénarios d'application, y compris :

- La génération de modèles de bas niveau (et du code éventuellement) à partir de modèles de haut niveau [KWB03], [MVG06], [SK03].
- Le mapping et la synchronisation de modèles du même niveau ou de différents niveaux d'abstraction [IK04].
- La création de vues basées sur de requêtes d'un système [BF05], [SF05].
- Les tâches d'évolution de modèles tels que le *refactoring* de modèles [SPLTJ01], [ZLG05].
- L'ingénierie inverse de modèles de haut niveau à partir de modèles de bas niveau [Fav04].

La figure 1.2 donne un aperçu des principaux concepts qui participent à la transformation de modèles. Elle présente le cas le plus simple, un modèle d'entrée (*modèle source*) et un modèle de sortie (*modèle cible*). Les deux modèles sont conformes à leur méta-modèles respectifs, (méta-modèle source et méta-modèle cible). Par ailleurs, le méta-modèle source et le méta-modèle cible peuvent coïncider dans certains cas. Si les deux modèles sont conformes au même méta-modèle, la transformation est dite *endogène*. Si les méta-modèles sont différents, la transformation est dite *exogène* [CH06]. Des exemples typiques de transformations endogènes sont : l'optimisation, le refactoring et la simplification. Des exemples de transformations exogènes sont : la génération de code et l'ingénierie inverse. Une transformation endogène particulière est une transformation *in situ*, celle-ci s'applique au modèle source pour le mettre à jour sans générer un modèle cible [MVG06].

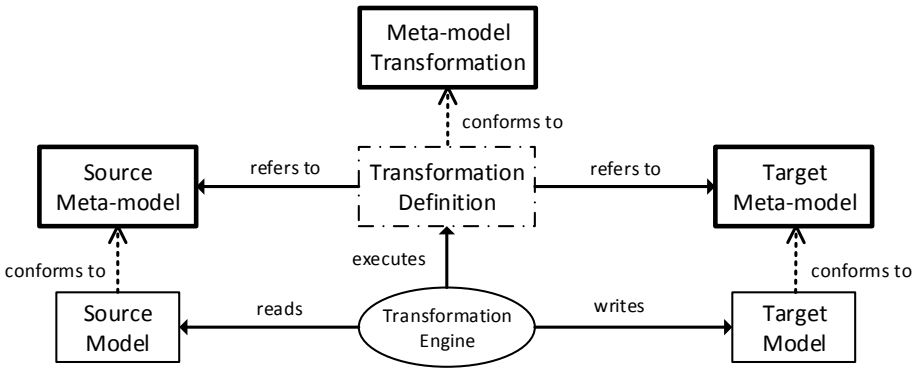


FIGURE 1.2 – Concepts basiques de la transformation de modèles d'après [CH06]

Dans les transformations exogènes, vu que le méta-modèle de sortie est différent de celui d'entrée, il est nécessaire de spécifier les relations de chacun des éléments de la source vers la cible.

Par défaut, une transformation de modèle est *unidirectionnelle*. Une transformation *bidirectionnelle* est celle qui peut être exécutée dans les deux sens, cela veut dire que chaque méta-modèle joue à la fois le rôle de source et de cible. Une transformation bidirectionnelle peut être définie en utilisant une seule règle bidirectionnelle ou à l'aide de deux règles unidirectionnelles [AKP03]. Une transformation est dite réver-

sible s'il existe une transformation permettant de retrouver le modèle source à partir du modèle cible.

Les approches de transformation de modèles sont classifiées dans deux grandes familles : M2M (*Model-To-Model*) et M2T (*Model-To-Text*). Une transformation M2T lit un modèle et produit du texte. Tandis qu'une transformation M2M lit un modèle et produit un autre modèle.

Dans la famille M2T, on trouve des approches de parcours de modèle en utilisant des langages de programmation classiques (le patron de conception visiteur peut être utilisé pour M2T [CH06]), les approches basées sur des parseurs et les approches basées sur des templates [CH03]. Dans la famille M2M, on trouve des approches dirigées par :

- La structure (*i.e.* RFP-QVT [ACE+03]).
- Manipulation directe du modèle (*i.e.* Jamda [Jam15]).
- Des relations entre les éléments du modèle (relationnelles) (*i.e.* MOF QVT Relations [QVT15], MTF (*Model Transformation Framework*) [MTF15], Tefkat [Tef15]).
- La transformation de graphes.
- Des templates.
- Des parseurs et hybrides [CH06], par exemple, Standard QVT (QVT Core, QVT Relations, QVT Operationnal) [GGKH03].

1.3.4 Mise en œuvre

Les métalangages de niveau M3 permettent de définir la syntaxe abstraite d'un langage de modélisation spécifique. Dans le paradigme de programmation orienté-objet, les concepts de la syntaxe abstraite sont représentés au travers de classes contenues dans des paquetages et reliées par différentes relations (association, composition, spécialisation). Chaque classe représente un concept du langage de modélisation, c'est-à-dire, un concept du domaine pour lequel le langage est conçu.

Parmi les divers formalismes outillés, Ecore est le formalisme de méta-modélisation sous-jacent d'*Eclipse Modeling Framework* (EMF [EMF15a]). Il coïncide en grande partie avec EMOF, une partie de la spécification MOF. EMF est une plateforme qui facilite la modélisation et la génération de code pour la construction d'outils basés sur un modèle de données structuré. Elle fournit des fonctionnalités pour la production de classes Java pour manipuler les modèles et un éditeur de modèles basique. De plus, EMF facilite la production d'éditeurs graphiques et textuels de modèles, et la définition de transformations de modèles.

La figure 1.3 donne la représentation graphique de l'extrait d'Ecore qui sera utilisé dans cette étude. Avec Ecore, un méta-modèle est organisé en paquetages (*EPackage*) lesquels peuvent eux-mêmes être constitués de sous-*EPackage*s. Chaque *EPackage* définit un ensemble de *EClassifier*s, lesquels peuvent être primitifs (*EDataType*) ou complexes (*EClass*). Les *EClass*es sont constituées d'un ensemble de propriétés (*EStructuralFeature*). Une *EStructuralFeature* est, soit un attribut de type primitif (*EAttribute*), soit une *EReference* vers une *EClass* [HVW10].

De nombreux outils et environnements de méta-modélisation s'appuient sur Ecore (*i.e.* eMoflon [eMo14], Kermeta [Ker14], ATL [ATL14], Henshin [Hen14], Epsilon [Eps14], Xtext [Xte15]) pour outiller les langages de modélisation.

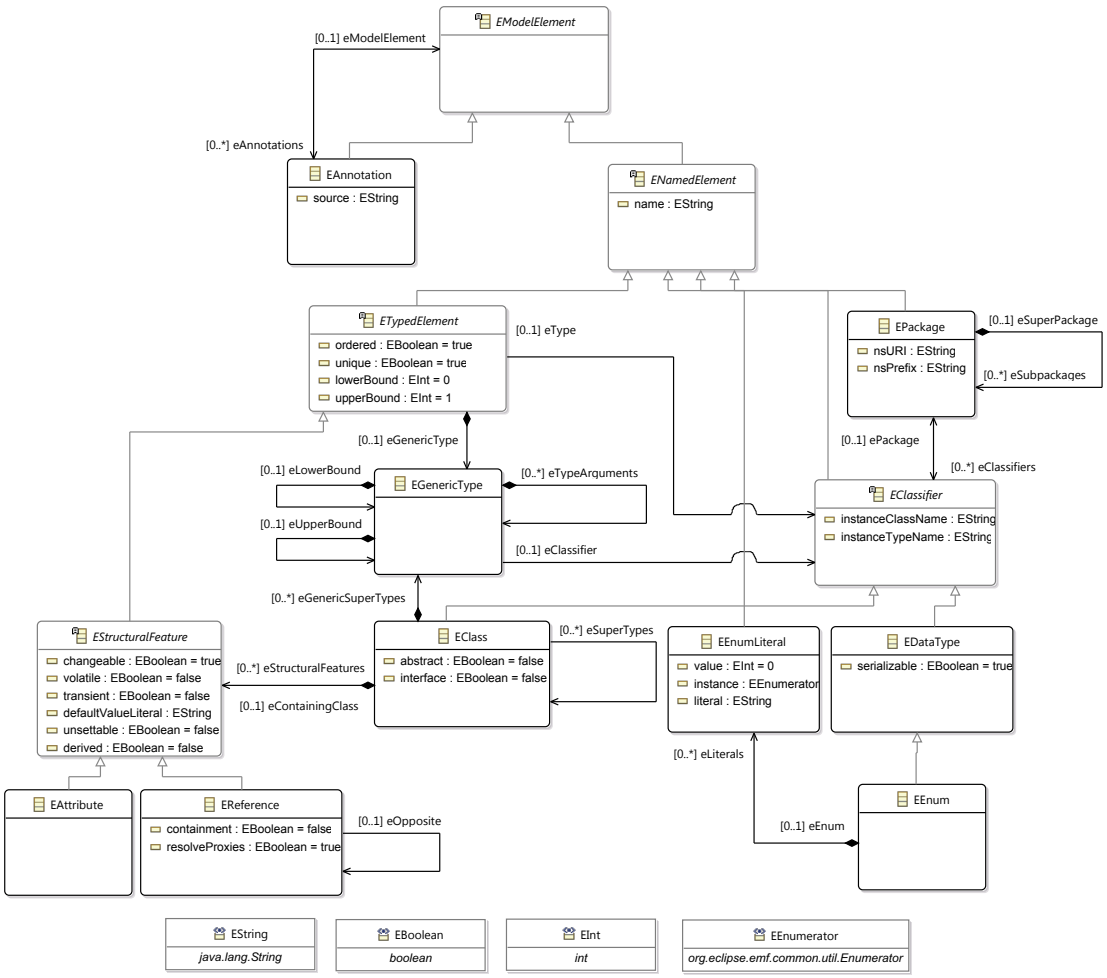
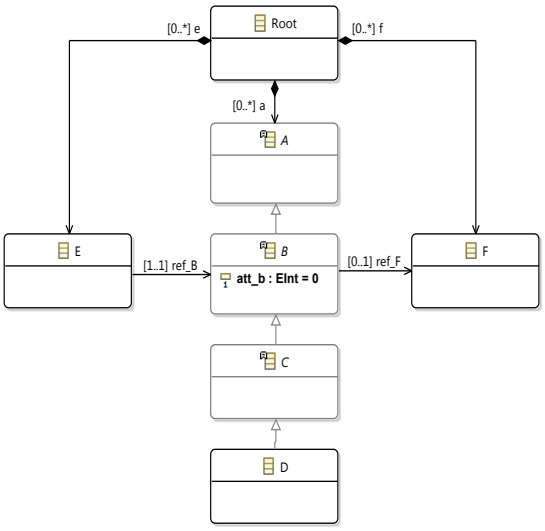


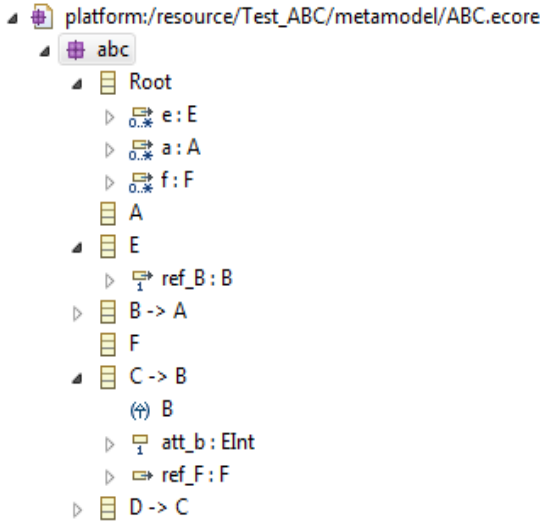
FIGURE 1.3 – Formalisme de méta-modélisation avec les concepts de base de la méta-modélisation

La figure 1.4 illustre la représentation graphique, la représentation arborescente et la représentation textuelle d'un méta-modèle fait à l'aide d'Ecore. Il s'agit d'un méta-modèle constitué des classes *Root*, *A*, *B*, *C*, *D*, *E*, et *F*. *A*, *B* et *C* étant des classes abstraites. *Root* peut contenir plusieurs *A* (référence *a*), plusieurs *E* (référence *e*) et plusieurs *F* (référence *f*). *A* possède l'attribut *att_b*. Les références *ref_B* et *ref_F* relient les classes *E*, *B* et *F*.

La figure 1.5 illustre un modèle conforme au méta-modèle de la figure 1.4. Les éditeurs arborescent et textuel des modèles sont également illustrés. Il s'agit d'un modèle constitué d'une racine, une instance de *E*, une instance de *D* dont l'attribut *att_b* vaut 15 et une instance de *F*.



(a) Représentation graphique

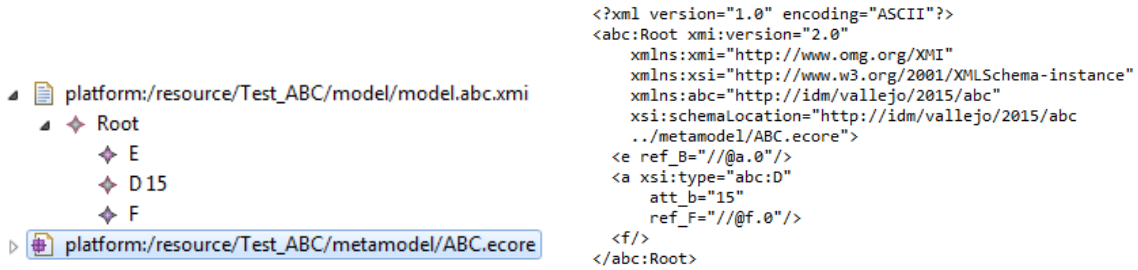


(b) Représentation arborescente

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="abc"
  nsURI="http://idm/vallejo/2015/abc" nsPrefix="abc">
  <eClassifiers xsi:type="ecore:EClass" name="Root">
   <eStructuralFeatures xsi:type="ecore:EReference" name="e"
    upperBound="-1" eType="#//E" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="a"
    upperBound="-1" eType="#//A" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="f"
    upperBound="-1" eType="#//F" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="A" abstract="true"/>
  <eClassifiers xsi:type="ecore:EClass" name="E">
    <eStructuralFeatures xsi:type="ecore:EReference" name="ref_B"
    lowerBound="1" eType="#//B"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="B" abstract="true"
    eSuperTypes="#//A"/>
  <eClassifiers xsi:type="ecore:EClass" name="F"/>
  <eClassifiers xsi:type="ecore:EClass" name="C" abstract="true"
    eSuperTypes="#//B">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="att_b"
      lowerBound="1" eType="ecore:EDataType
      http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="ref_F"
      eType="#//F"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="D" eSuperTypes="#//C"/>
</ecore:EPackage>
```

(c) Représentation textuelle

FIGURE 1.4 – Méta-modèle fait avec Ecore



(a) Représentation arborescente (b) Représentation textuelle

FIGURE 1.5 – Modèle fait avec Ecore

1.4 Langages de modélisation spécifiques à un domaine

Les langages de modélisation spécialisés permettent de modéliser une préoccupation (ou aspect particulier) d'un système dans un langage dédié fournissant les concepts spécifiques au domaine de cette préoccupation (ou de cet aspect). Un DSML est destiné à définir les concepts et les principes d'un domaine. En particulier, il définit le vocabulaire de base du domaine et la relation entre ses principaux concepts. Il ne manipule que les concepts qui sont propres au métier, il est plus simple à comprendre, à transformer et à manipuler. Les DSMLs fournissent généralement un nombre réduit de concepts, qui sont familiers aux experts du domaine, ce qui facilite leur apprentissage. Comme dans les langages de programmation, un DSML est défini selon :

- Sa syntaxe abstraite : elle décrit les concepts manipulables par un ordinateur.
- Sa syntaxe concrète : elle offre une représentation des concepts compréhensible et manipulable par l'utilisateur.
- Sa sémantique : elle définit la signification des différentes constructions du langage et donne un sens aux modèles ou programmes de ce langage.

L'expressivité des langages dédiés permet d'exprimer les mêmes solutions qu'avec un langage généraliste, mais souvent d'une manière plus simple et plus concise. Les langages de modélisation spécifiques à un domaine constituent une approche prometteuse pour la réduction des coûts de développement de logiciel et pour augmenter la productivité dans le développement. Mais du fait de leur aspect spécifique, le nombre de langages de modélisation est en constante augmentation [HWRK11], [Kle08] principalement pour deux raisons :

1. La gamme de domaines abordés par des systèmes logiciels de plus en plus large (domotique, avionique, etc.) et de plus en plus spécifique (voiture électrique, données océaniques, etc.). Aborder ces domaines demande de prendre en compte les spécificités de chaque domaine dans de nouveaux langages.
2. La taille et la complexité des systèmes logiciels. Pour conserver la maîtrise du développement de tels systèmes, ils sont décomposés en modules inter-connectés. Ces modules permettent aux développeurs d'encapsuler une préoccupation au sein d'un module d'une part et de raisonner au niveau du système en abstrayant les détails internes aux autres modules d'autre part.

Ces deux raisons amènent à la création continue de nouveaux langages (avec l'objectif d'aborder un nouveau domaine ou de proposer une nouvelle manière de gérer la complexité), ainsi que l'évolution des langages existants. Parmi tous ces langages logiciels, il n'est pas rare que certains langages partagent des caractéristiques communes (concepts, structure, outils). De la même manière, l'évolution d'un langage conduit à un ensemble de versions partageant une base commune. Ces langages, qui partagent des caractéristiques communes, forment des *familles* de langages.

Introduire un nouveau DSML conduit à la production d'encore un autre langage de modélisation. Ce qui accentue encore le problème de *la tour de Babel* (*the Babel Tower problem* [Hud97]), c'est-à-dire un grand nombre de modèles qui sont difficiles à gérer, et la nécessité de construire des ponts d'interopérabilité entre eux. Ce qui induit le besoin de la transformation de modèles. Pour s'attaquer à ce problème, nous avons besoin de faciliter la définition de DSMLs et des migrations de modèles entre DSMLs.

Ces transformations sont nombreuses et difficiles à écrire. L'écriture des transformations représente une grande charge de travail [FHLN08]. Dans ce contexte, une problématique récurrente est la transformation de modèles pour les rendre conformes à des méta-modèles similaires.

La section suivante présente un mécanisme qui facilite la transformation des méta-modèles et la migration des modèles.

1.5 Co-évolution

Dans cette section, nous introduisons les principes de la co-évolution, (section 1.5.1). Puis, nous nous concentrons sur l'évolution de méta-modèle et la co-évolution de modèles (section 1.5.2).

1.5.1 Principes

Toute évolution d'un méta-modèle impacte les artefacts (modèles, transformations, contraintes OCL, etc.) qui lui sont associés. La co-évolution consiste à faire évoluer un artefact automatiquement suite à l'évolution du méta-modèle auquel il est lié. Dans certains cas, les évolutions sont indépendantes des autres éléments du méta-modèle (révision mineure pour la migration des instances). Dans d'autres cas, les évolutions du méta-modèle introduisent des inconsistances qui ne peuvent pas être résolues automatiquement [Wac07], [BP07]. Il est important de signaler que dans le cas général, une co-évolution entièrement automatique n'est pas possible, à cause des modifications non calculables automatiquement. Les changements nécessitent parfois l'intervention du concepteur du méta-modèle [RKP+14].

Les co-évolutions les plus fréquentes sont les suivantes [Has11] : méta-modèle et transformation [MEMC10], [LBNK10], méta-modèle et contrainte, méta-modèle et documentation et méta-modèle et modèle. Plus précisément, nous allons étudier le dernier cas, soit l'évolution de méta-modèles et la co-évolution de modèles. Les transformations faites au niveau méta-modèle seront appelées *Évolutions ou Refactoring* et les transformations faites au niveau modèle seront appelées *Co-évolutions ou Migrations*. Une migration est donc l'adaptation d'un modèle comme conséquence d'une évolution du méta-modèle auquel le modèle est conforme. La section suivante présente, avec plus de détails cette co-évolution.

1.5.2 Évolution de méta-modèle et co-évolution de modèles

L'évolution de méta-modèle et co-évolution de modèles a été étudiée par différents travaux : [Wac07], [CREP08], [Ber03]. Ses principes sont illustrés par la figure 1.6. Les transformations à appliquer au méta-modèle source sont représentées dans un document de règles de transformation.

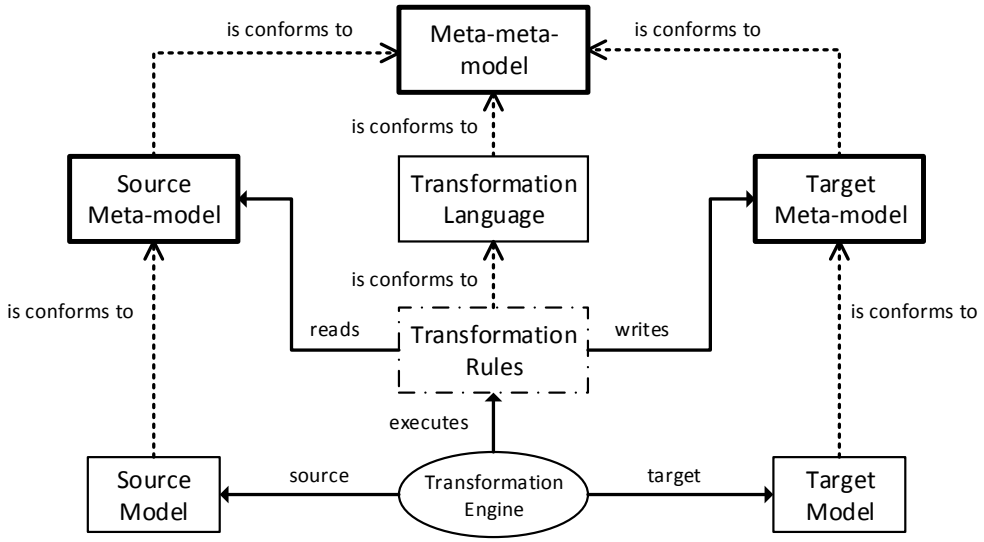


FIGURE 1.6 – Co-évolution, d'après [CREP08]

Les changements qui se produisent sur un méta-modèle peuvent avoir différents effets sur les modèles correspondants. En général, les modifications sont classées comme [BP07], [CDRP09] :

- Sans effet sur les instances qui existent (*non-breaking changes*). Elles ne cassent pas la conformité entre les modèles et leur méta-modèle.
- Avec des effets secondaires simples sur les modèles (*breaking and resolvable changes*). Elles cassent la conformité, mais les modèles peuvent être adaptés automatiquement. Dans cette catégorie se situe l'addition de nouveaux éléments au méta-modèle.
- Avec effets secondaires qui exigent une gestion supplémentaire (*breaking and unresolvable changes*). Elles cassent la conformité, elles ne peuvent pas être adaptées automatiquement et nécessitent une modification faite à la main qui ne peut pas être automatisée [SK04]. Il faut des informations complémentaires de la part de l'utilisateur.

Selon Wachsmuth [Wac07], les transformations d'un méta-modèle peuvent donner lieu à des modèles migrés invalides, c'est pourquoi, chaque évolution du méta-modèle nécessite une analyse préalable. Les interventions manuelles sont souvent nécessaires pour faire face aux *breaking changes* dans lesquels les modèles conformes ne peuvent pas être migrés automatiquement. Une intervention humaine est nécessaire pour introduire les valeurs requises, autrement des valeurs par défaut sont considérées. Les adaptations manuelles sont sujettes aux erreurs et peuvent donner lieu à

des incohérences entre le méta-modèle et les modèles liés. On peut identifier ces évolutions en fonctions des évolutions d'un méta-modèle. Par exemple, l'ajout d'un élément non obligatoire est un *non-breaking change*, le renommage est un *breaking and resolvable change* et l'ajout d'un élément obligatoire est un *breaking and unresolvable change*. Cette classification est présentée par Cicchetti dans [CDRP09]. En ce qui nous concerne, nous nous intéressons aux transformations de type *non-breaking changes* et *breaking and resolvable changes*.

La définition de la migration s'appuie généralement sur l'un des principes suivants [RPKP09] :

Spécification manuelle des migrations. La stratégie de migration est codée à la main par le concepteur du méta-modèle. Rose [RKPP10] indique que ce type de spécification est fastidieux et source d'erreurs et ralentit fortement la pratique de développement. L'automatisation de la migration en réponse à des adaptations du méta-modèle promet de réduire considérablement l'effort. Mais il est souvent difficile d'identifier les modifications appliquées au méta-modèle, et jusqu'à quel point il est possible d'automatiser la migration.

Relations explicites d'appariement (*matching*) entre concepts. Ce principe utilise des techniques d'appariement pour déterminer la stratégie de migration à partir des différences entre le méta-modèle et le méta-modèle évolué. D'après Didonet [DV09], les approches d'appariement essaient de détecter un modèle de différences à partir de la comparaison entre deux versions d'un méta-modèle. Le modèle de différences entre les deux méta-modèles est de fait un modèle de *refactoring*.

Ces approches comportent deux étapes, le *matching* et le *weaving*. La première consiste à trouver des liens entre les éléments d'un ensemble de méta-modèles d'entrée (*matching*). La deuxième consiste à créer un modèle qui capte différents types de liens entre les modèles [DV09] (*weaving model*) avec les liens de la première étape, le *weaving model* est traduit en des modèles de transformation. Le *matching* est bien connu dans plusieurs domaines d'application, comme par exemple la web sémantique, l'intégration de schémas et ontologies, les *data warehouses* et l'*e-commerce* [RB01], [SE05].

Del Fabro [DFV07] présente une approche qui utilise les *matching* et les *weaving models* pour semi-automatiser le développement des migrations. D'après lui, un processus semi-automatique accélère le temps de développement de la transformation, diminue les erreurs qui peuvent se produire dans le codage à la main (spécification manuelle des migrations) et augmente la qualité de la transformation.

Opérateurs de co-évolution. Dans ces approches, l'évolution du méta-modèle est spécifiée au moyen d'une séquence d'opérateurs. Au niveau méta-modèle, les opérateurs définissent une évolution. Au niveau modèle, les opérateurs définissent la migration correspondante.

L'application d'un opérateur de co-évolution conserve la conformité du modèle migré avec la version évoluée du méta-modèle. Par exemple l'opérateur *add* ajoute des éléments au méta-modèle et l'opérateur *remove* supprime des éléments du méta-

modèle [GJCB09], [HBJ09], [Wac07], [CREP08]. Ces opérateurs sont typiquement implémentés dans des langages adaptés, *e.g.* Kermeta et Henshin [ABJ+10].

Pour automatiser la migration de modèles, les approches basées sur des opérateurs fournissent des opérateurs de co-évolution réutilisables. Ils intègrent l'évolution du méta-modèle et la migration du modèle.

Wachsmuth propose une classification des opérateurs de co-évolution en fonction de leur sémantique et les propriétés de préservation des instances [Wac07]. COPE étend l'approche en permettant la spécification manuelle d'opérateurs personnalisés et la définition de nouveaux opérateurs [HK10]. Ces deux travaux montrent que certaines évolutions au niveau méta-modèle ont besoin d'informations lors de la migration, qui ne sont pas disponibles dans le modèle. En conséquence, ces évolutions empêchent intrinsèquement la migration automatique des modèles. Pour pallier cet inconvénient, les concepteurs de langages doivent pouvoir prendre des décisions pour faire des migrations manuelles [HR10].

Les opérateurs peuvent être classifiés en fonction de certaines propriétés [HVW10] :

Préservation du langage. L'extension d'un méta-modèle est un ensemble de modèles conformes à lui-même. Quand un opérateur est appliqué, ceci à un impact sur l'extension et par conséquent sur l'expressivité du langage. Dans [HVW10], les opérateurs sont classifiés selon leur impact : un opérateur est *refactoring* s'il existe toujours un mapping bijective entre le méta-modèle évolué et le méta-modèle original. Un opérateur est *constructeur* s'il y a une injection dans le méta-modèle évolué. Un opérateur est *destructeur* s'il supprime des éléments du méta-modèle original pour produire le méta-modèle évolué.

Préservation du modèle. Un opérateur est *model-preserving* si tous les modèles conformes au méta-modèle d'origine sont aussi conformes au méta-modèle évolué. Donc, ces opérateurs ne nécessitent pas de migration de modèles. Un opérateur est *model-migrating* si les modèles conformes au méta-modèle d'origine doivent être migrés pour les rendre conformes au méta-modèle évolué. Les opérateurs *model-preserving* effectuent des *non-breaking changes*, tandis que les *model-migrating* effectuent des *breaking* et *resolvable changes*. Pour les *breaking* et *unresolvable changes*, les opérateurs fournissent toujours une migration capable de résoudre le *breaking change* [HVW10].

Bidirectionnalité. La bidirectionnalité est la propriété qui indique que l'action d'un opérateur peut être défaite [HVW10]. Un opérateur est involutif (*self-inverse*) si et seulement si une deuxième application du même opérateur donne toujours le méta-modèle original. Un opérateur est l'*inverse* d'un autre opérateur si et seulement s'il y a toujours une composition séquentielle des deux opérateurs qui est un refactoring. Un opérateur est *safe inverse* si et seulement s'il y a toujours une composition séquentielle des deux opérateurs qui est *model-preserving* [HVW10].

1.6 Conclusion

Ce chapitre établit les concepts de base que nous utiliserons tout au long de ce document. La réutilisation est l'objectif principal de cette recherche. L'approche proposée s'appuie sur les principes de l'IDM et en particulier sur les transformations de modèles, et plus particulièrement des opérateurs de co-évolution.

Les langages de modélisation spécifiques à un domaine permettent de capitaliser le savoir-faire lié à une préoccupation spécifique. Ils offrent un moyen d'exprimer des solutions dans un langage proche du domaine de cette préoccupation, ce qui facilite leur apprentissage et leur utilisation aux experts et aux utilisateurs du domaine. L'IDM offre des environnements pour la définition et l'outillage de ces langages. Cependant, leur développement peut rester coûteux, entre autres à cause de la prise en compte des spécificités sémantiques du langage qui restent en dehors du champs des outils traditionnels de l'IDM.

Le chapitre suivant présente l'état de l'art concernant les approches de réutilisation des transformations de modèles, la différentiation et la réduction de méta-modèles, la migration de modèles et l'inversion de la migration. Ces approches proposent des solutions au problème de la réutilisation.

Chapitre 2

État de l'art

Sommaire

2.1 Réutilisation des transformations de modèles

Dans le chapitre précédent, nous avons introduit les concepts utiles pour cette thèse : la réutilisation, l'IDM et la co-évolution. Dans ce contexte, nous présentons maintenant des approches qui proposent des solutions permettant de résoudre le problème posé de la réutilisation de transformations de modèles. Avant d'étudier les approches similaires et complémentaires à la thèse, nous étudions quelques approches qui abordent différemment la problématique de la réutilisation de transformations de modèles.

2.1.1 Migration des transformations

Mendez [MEMC10] présente une approche de migration des transformations, dans laquelle si un méta-modèle évolue, les transformations associées sont mises à jour. L'objectif de cette mise à jour est de rétablir la cohérence de la transformation par rapport au méta-modèle évolué. Ainsi la transformation peut être réutilisée sur les modèles conformes au méta-modèle évolué. La migration de la transformation est effectuée en trois étapes : la détection de l'impact, l'analyse de l'impact et l'adaptation de la transformation.

La détection de l'impact permet d'identifier les inconsistances de la transformation, produites par l'évolution du méta-modèle. Concrètement, cette étape détecte les transformations qui ne respectent pas la *conformité du domaine*. C'est-à-dire la relation entre une transformation et le méta-modèle sur lequel elle s'applique. La conformité du domaine précise que les éléments de la source de la transformation doivent correspondre à une classe dans le méta-modèle source (et de même pour les éléments du méta-modèle cible). Parmi les évolutions qui peuvent être identifiées dans l'étape de détection, nous listons quelques-unes : le renommage d'une classe, le déplacement d'une propriété, la modification d'une propriété, l'ajout d'une classe ou d'une propriété (dans le méta-modèle source) ; le replacement d'une classe ; la suppression d'une classe ou d'une propriété.

L'analyse de l'impact détermine quelles sont les mises à jour à appliquer sur la transformation. Selon les évolutions détectées dans l'étape précédente, des suggestions pour rétablir la cohérence de la transformation sont proposées. Par exemple, en cas de renommage d'une classe, la transformation qui utilise la classe, doit mettre à jour le nom de la classe. Si un élément d'une classe est déplacé vers une autre classe, le chemin pour accéder à l'élément doit être mis à jour dans la transformation.

En cas de modification d'une propriété (*i.e* multiplicité), la mise à jour ne peut pas être automatisée. Prenons par exemple, une transformation qui s'applique à une propriété dont la multiplicité est [*0..**]. La multiplicité de cette propriété change et devient [*1..1*]. La transformation doit maintenant s'appliquer sur un seul élément, le choix de l'élément à transformer doit être fait manuellement, il ne peut être pas généralisé.

L'ajout d'une classe ou d'une propriété n'affecte pas la mise en place de la transformation. Mais cela a pour conséquence que la transformation ne s'applique pas sur l'élément ajouté. Afin de permettre à la transformation de s'appliquer sur cet élément ajouté, elle doit être adaptée manuellement.

Dans le cas du remplacement d'une classe, toutes les occurrences de la classe dans la transformation doivent être changées. Et les dépendances vers les éléments de la

classe doivent être fixées manuellement afin de trouver les équivalences dans la nouvelle classe. La suppression d'une classe ou d'une propriété implique que toutes les règles de transformation qui s'appliquent sur la classe ou la propriété doivent être supprimées.

Finalement, l'étape d'adaptation de la transformation applique les mises à jour déjà définies lors de l'analyse de l'impact.

Il est à noter que dans cette approche, les évolutions du méta-modèle qui ne prévoient pas de mise à jour automatique, nécessitent une intervention humaine. La migration de la transformation, implique alors la réécriture sous assistance de la transformation. De plus, cela nécessite que la transformation soit décrite en *boîte blanche* (le comportement de la transformation est connu) et que l'utilisateur de la transformation ait les connaissances nécessaires pour la modifier.

Pour ce qui nous concerne, nous partons des principes, que la transformation à réutiliser est de *boîte noire* (la description des actions de l'outil n'est pas disponible) ou au mieux, en *boîte grise* (on dispose d'une spécification de haut niveau de l'outil, mais dont le comportement n'est pas précisé). Dans le contexte de la thèse, la transformation ne doit pas être modifiée. Pour autant , de cette étude nous conservons l'idée d'une assistance dans l'adaptation de la transformation pour des situations spécifiques.

2.1.2 Sous-typage de modèles

La réutilisation peut être fondée sur les notions de *type de modèle* [GCD+12] et de *sous-typage* [SJ07]. Un type de modèle est un ensemble de classes MOF et les références qu'elles contiennent. Un sous-type, est un sous-ensemble d'un type de modèle qui possède les concepts nécessaires pour appliquer une transformation spécifique.

Le sous-typage détermine la *substituabilité* de modèles, c'est-à-dire qu'il permet qu'un modèle typé B, puisse être utilisé là où un modèle typé A est attendu (A et B étant des types de modèles et B étant un sous-type de A). Cette notion de substituabilité est fondamentale pour la réutilisation d'une transformation, car elle permet de changer son contexte d'utilisation et donc *de facto* de la réutiliser.

Guy présente quatre mécanismes pour atteindre la *substituabilité* de modèles. Chacun de ces mécanismes (isomorphique, non isomorphique, total et partiel) est défini formellement au moyen d'une *relation de sous-typage* [Guy13].

Une relation de sous-typage isomorphique, est celle où chaque classe de B, trouve une classe de A qui possède le même nom (B étant un sous-type de A). Les deux classes doivent avoir exactement le même nombre d'attributs et de références. Les attributs et les références doivent être identiques (même nom, même multiplicité, mêmes propriétés obligatoires, etc.).

Une relation de sous-typage non isomorphique, accepte des éléments représentés sous une autre forme que prévu (par exemple, les classes sont nommées différemment). Mais elle doit toujours être ramenée à une relation de sous-typage isomorphique à l'aide d'une adaptation simple de modèles. L'adaptation inverse doit aussi être spécifiée. Les adaptations sont effectuées via l'implémentation de transformations de modèles supplémentaires ou via l'addition des types manquants et de propriétés dérivées. L'utilisateur prend en charge la définition des adaptations et la vérification de la bidirectionnalité entre elles.

À titre d'illustration, nous supposons qu'un attribut est ajouté à une des classes du type de modèle de la transformation. Par défaut, la transformation ne s'applique pas sur le nouvel attribut. Et comme dans le cas de la migration des transformations (section 2.1.1), il est nécessaire de modifier la transformation pour l'inclure. La fusion de classes est un autre cas qui n'est pas traité automatiquement et qui demande l'intervention d'un expert pour déterminer comment sont reliés les éléments. Cette tâche d'adaptation manuelle peut se révéler complexe et source d'erreurs. Guy évoque en perspective à son étude, le besoin d'outils qui facilitent la mise en place des adaptations complexes.

Si tous les concepts d'un type de modèle sont utilisés par une transformation, le type de modèle est donc, considéré un *type exact*. Dans les cas d'un type de modèle exact, la réutilisation n'est pas exploitée, parce qu'il n'y a pas de sous-types sur lesquels on puisse appliquer la même transformation. La réutilisation par sous-typage de modèles est particulièrement utile pour réutiliser des transformations dont le type de modèle est plus large que le sous-ensemble des concepts utilisés par la transformation. Cependant, il n'est pas toujours possible de trouver un type de modèle commun à plusieurs types de modèles. Cela est dû à des hétérogénéités structurelles entre les différents méta-modèles.

Concernant ces travaux, nous reprenons l'idée de faciliter la réutilisation des transformations sans les modifier. Comme eux, nous portons intérêt à la manipulation des concepts qui sont manipulés par la transformation (type exact). Et nous confirmons le besoin d'inclure les adaptations des méta-modèles complexes dans la réutilisation.

2.1.3 Transformation de modèles incrémentale

Kusel [EKK+13], présente un cadre d'évaluation dédié pour les approches de transformation de modèles incrémentales [CH06]. De façon générale l'étude indique qu'un modèle peut être dérivé d'un modèle initial au moyen d'une transformation de modèles. Et que les modèles dérivés doivent être mis à jour si le modèle initial évolué. Grâce aux transformations incrémentales, il est possible de réutiliser la transformation uniquement sur la partie évoluée du modèle.

Dans l'approche de réutilisation par transformations incrémentales, la transformation réutilisée reste inchangée. Néanmoins, il est indispensable que la transformation soit en boîte blanche. Parmi les différentes approches étudiées par Kusel, seulement une accepte des transformations en boîte noire, mais ce fait peut altérer l'exactitude du résultat. Il est également nécessaire d'avoir des connaissances précises sur ce que fait la transformation et sur les données manipulées.

Dans cette étude, l'auteur évoque le besoin de permettre à l'utilisateur de définir comment et sur quelle partie du modèle doit être appliquée la transformation. Toutes les approches évaluées ont des limites, spécifiquement quand les modèles changent considérablement. Comme il n'est pas évident de déterminer automatiquement toutes les adaptations appliquées à un modèle. En effet, elles sont utilisées pour des modèles dont les évolutions sont mineures.

L'approche de réutilisation à laquelle nous nous intéressons dans cette thèse a pour points communs avec ces approches :

- Pouvoir appliquer une transformation sur une partie spécifique d'un modèle.
- Permettre à l'utilisateur de définir sur quelles parties elle doit être appliquée.

– Laisser la transformation inchangée.

Par contre, nous considérons que l'outil est en boîte grise.

2.1.4 Typage de modèles basé sur les graphes

Pham [Pha12] propose une approche qui permet de migrer les transformations existantes pour les appliquer à un méta-modèle évolué. Ce dernier a été légèrement modifié par rapport au méta-modèle d'origine. La solution utilise une relation de typage de modèles basée sur de graphes.

L'étude propose un langage qui permet à un utilisateur de définir les correspondances sémantiques entre les éléments du méta-modèle d'origine et le méta-modèle évolué. Ensuite, les transformations sont migrées en utilisant les correspondances définies. L'automatisation de la migration est de grande importance pour promouvoir la réutilisation de la transformation et pour éviter une adaptation manuelle fastidieuse et source d'erreurs. Cependant, les correspondances doivent être écrites manuellement, parce que des modifications comme la généralisation ou la réification de concepts ne sont pas faciles à identifier de façon automatique.

Concrètement, la réutilisation est effectuée en quatre étapes :

– Construction des points de vue graphiques (topologie de graphe avec multiplicité) pour le méta-modèle d'origine et pour le méta-modèle évolué.
– Déduction de la partie effective (éléments qui sont utilisés par la transformation) du méta-modèle d'origine.
– Description des correspondances entre les éléments des deux méta-modèles.
– Exécution de l'adaptation de la transformation.

De cette étude, nous portons notre attention sur l'idée de permettre aux utilisateurs de définir les correspondances entre les méta-modèles à l'aide d'un langage dédié. Même si nous voulons pouvoir exprimer des différences plus complexes entre les méta-modèles. Nous gardons également l'idée de la formalisation de la réutilisation des transformations via une sémantique de graphes. Par contre, nous considérons que les transformations sont en boîte noire ou en boîte grise.

2.1.5 Bilan

Une transformation définie pour un méta-modèle particulier ne peut pas être utilisée *stricto sensu* pour un autre méta-modèle. Comme nous l'avons vu dans les approches citées précédemment, il y existe des moyens pour faciliter la réutilisation, soit par modification directe de la transformation, soit au moyen des relations de sous-typage, soit en modifiant le contexte de la transformation. Cependant, ces approches ont des limites pour effectuer la réutilisation quand les différences entre les méta-modèles sont complexes à identifier ou à mettre en place.

L'adaptation de la transformation elle-même ou l'adaptation de son contexte implique la prise en compte de la conformité entre la transformation et le méta-modèle sur lequel elle s'applique. Elle est également important de considérer l'intelligence de la transformation. En général, ces approches sont conçus pour traiter des cas dans lesquels les méta-modèles sont structurellement proches. Et seulement les cas les plus simples comme le renommage et la suppression sont traités automatiquement.

La responsabilité pour gérer les adaptations plus complexes est déléguée à l'utilisateur. Des cas particuliers comme l'ajout et la fusion d'éléments ne peuvent pas être traités de façon automatique. Il devrait cependant y avoir des facilités pour détecter et mettre en place ces adaptations complexes de modèles, afin d'éviter que la mise en place de l'adaptation devienne aussi complexe que la mise en place de la transformation.

Notre idée c'est de proposer une approche de réutilisation de transformations, dans laquelle les transformations restent inchangées et les adaptations complexes sont traitées de façon automatique. On doit pouvoir proposer des facilités pour exprimer les différences entre les méta-modèles (de la transformation et initial). Puis, la gestion des adaptations complexes doit permettre de faire la réutilisation de transformations dans les cas où les méta-modèles sont hétérogènes. Une formalisation doit être proposée pour évaluer l'impact des adaptations.

Nous allons par la suite, explorer les approches existantes qui abordent les problématiques liées à : l'expression des différences entre les méta-modèles, l'extraction des données sur lesquelles s'applique la transformation, l'adaptation et l'adaptation inverse pour garantir la bidirectionnalité.

2.2 Exemple introductif

Nous présentons maintenant des approches qui proposent des solutions qui peuvent permettre de résoudre le problème posé. Afin de positionner ces approches, nous proposons un exemple simple mais illustratif du problème que nous traitons.

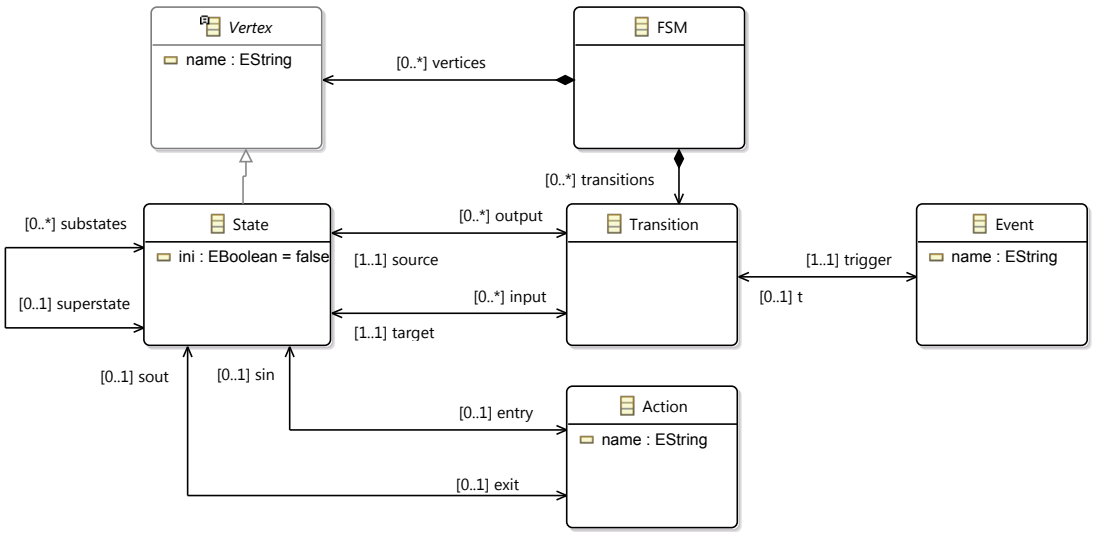


FIGURE 2.1 – Meta-modèle d'une machine à états finis

DSML. Nous considérons le méta-modèle de la figure 2.1 comme le méta-modèle qui définit un DSML pour lequel l'outillage doit être développé. Il est constitué d'états (State) hiérarchiques (*superstate* et *substates*) liés via les transitions (Transition). Une

transition est associée à un événement (Event). Les états et les transitions peuvent avoir des actions (Action) d'entrée (*entry*) et de sortie (*exit*) associées.

La figure 2.2 représente un modèle conforme au méta-modèle de la figure 2.1 avec la syntaxe concrète usuelle des diagrammes d'état. Dans ce modèle A, B et C sont des états initiaux. A est un état hiérarchique composé de l'état B et d'une action d'entrée a1. B est un état composé de l'état C et d'une action d'entrée a2. C et D sont des états atomiques. Une transition qui est associée à l'événement *ok* relie les états A et D.

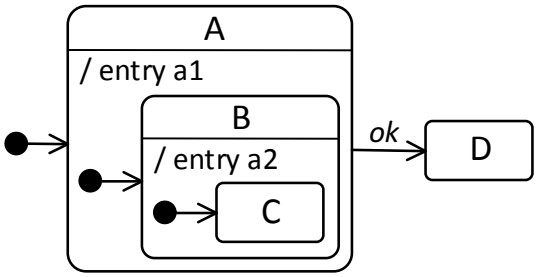


FIGURE 2.2 – Modèle conforme au méta-modèle de machine à états finis

Besoin. Nous supposons qu'il existe un besoin d'*aplatir* les machines à états *i.e.* de déterminer un diagramme équivalent sans états hiérarchiques. Un outil d'aplatissement de machines à états finis est donc requis. Au lieu de développer cet outil, nous proposons de réutiliser un outil existant.

Outil existant. Il s'agit d'un outil (Flatten) qui aplatit des machines à états conformes au méta-modèle de la figure 2.3. Les machines à états aplaties sont conformes au même méta-modèle. Ce méta-modèle modélise des états (State) hiérarchiques qui sont reliés via des transitions (Transition). Ces dernières peuvent être associées à des événements (Event).

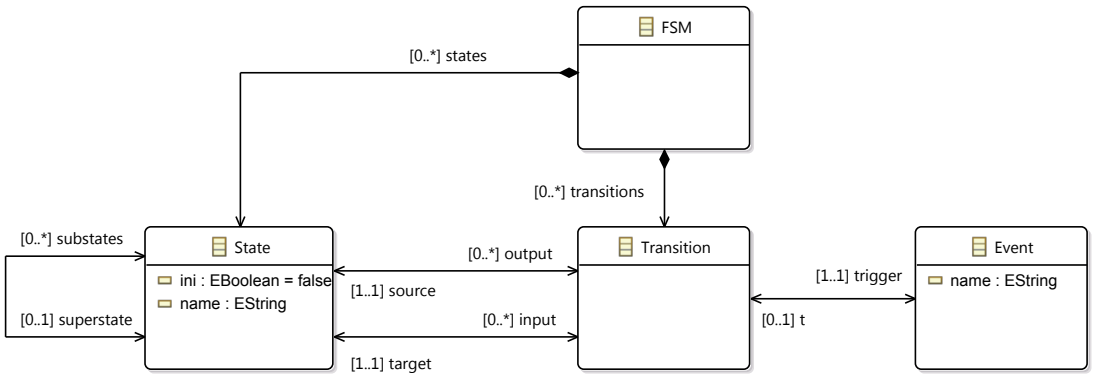


FIGURE 2.3 – Méta-modèle de l'outil qui aplati des machines à états

Contrairement au méta-modèle du DSML, ce méta-modèle ne tient pas compte d'actions ; State n'est pas une sous-classe de Vertex et est liée à FSM directement via la référence *states* ; le nom *name* de State est un attribut de lui-même et non de Vertex.

Quand cet outil est appliqué, chaque état hiérarchique est supprimé. Les états internes obtenus restent et sont renommés en fonction du nom de l'état hiérarchique et de leur propre nom.

Pour l'exemple, la figure 2.4 illustre un modèle à aplatir (il correspond au modèle de la figure 2.2, mais sans actions).

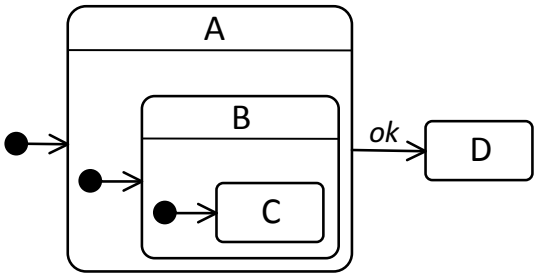


FIGURE 2.4 – Modèle de machine à états sans actions, conforme à l'outil qui aplati des machines à états

Réutilisation et problématiques. L'objectif dans cet exemple est de réutiliser l'outil existant pour aplatir le modèle de la figure 2.2. Le modèle aplati doit correspondre avec le modèle de la figure 2.5.

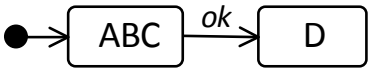


FIGURE 2.5 – Modèle aplati

Étant donné que ce modèle (figure 2.2) n'est pas conforme au méta-modèle de l'outil existant, certaines adaptations doivent être appliquées au modèle avant de l'aplatir. Les questions suivantes se posent alors :

1. Comment trouver les différences entre les méta-modèles ? (niveau méta-modèle).
2. Comment récupérer le sous-ensemble de données du DSML qui sont pertinentes pour l'outil ? (niveau méta-modèle).
3. Comment adapter les données du DSML pour qu'elles soient conformes aux données attendues par l'outil ? (niveau modèle).
4. Comment re-adapter les données produites par l'outil pour qu'elles soient à nouveau conformes au DSML ? (niveau modèle).

1. Différentiation. Pour résoudre le problème, il faut dans un premier temps pouvoir établir le degré de différentiation/correspondance entre deux méta-modèles. Dans la figure 2.6, SMM est le méta-modèle source qui correspond au domaine d'application de l'outil. TMM est le méta-modèle cible qui correspond au domaine de définition de l'outil. Dans notre exemple SMM est le méta-modèle de la figure 2.1 et TMM est le méta-modèle de la figure 2.3.

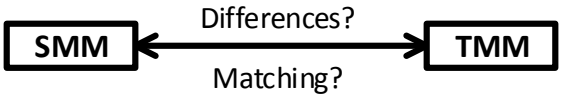


FIGURE 2.6 – Différences entre deux méta-modèles

La figure 2.7 illustre les différences entre les deux méta-modèles de l'exemple. Action, *entry*, *exit*, *sin*, *sout* ne font pas partie du méta-modèle de l'outil. Vertex et la relation d'héritage entre State et Vertex ne font pas partie du méta-modèle de l'outil, cependant, *name* est déplacé vers State et *vertices* pointe vers State. Dans le méta-modèle de l'outil *vertices* prend le nom *states*.

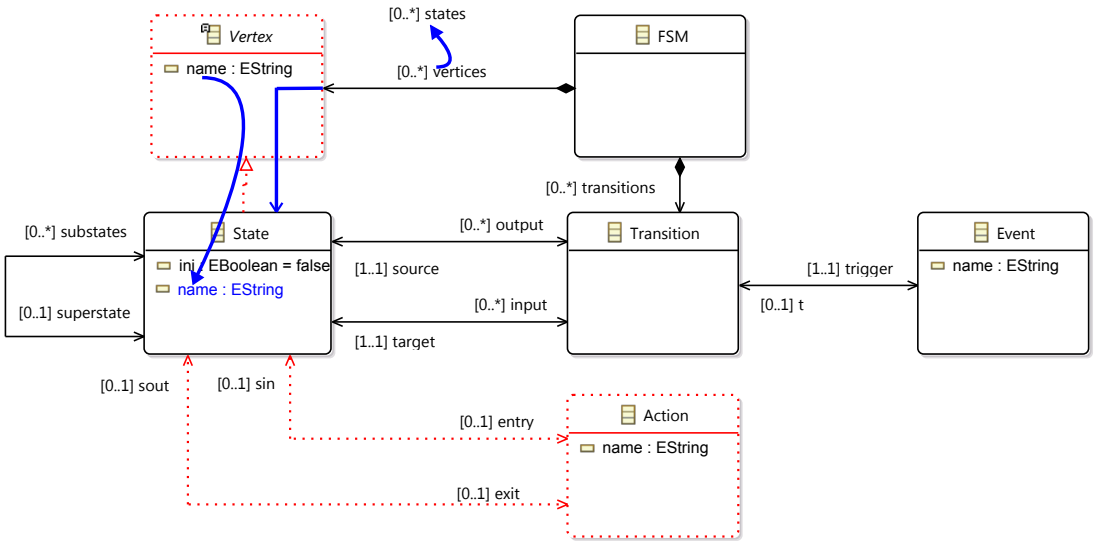


FIGURE 2.7 – Différences entre le DSML et l'outil existant

2. Réduction. Nous supposons que s'il est pertinent de vouloir réutiliser un outil qui s'applique à TMM sur des données conformes à SMM, alors il est pertinent de faire l'hypothèse que toutes les données nécessaires à l'outil sont dans SMM sous une forme éventuellement différente. Un sous-ensemble de SMM est censé correspondre au TMM (figure 2.8). La *réduction* est le mécanisme qui permet de prendre un sous-ensemble d'un méta-modèle ou d'un modèle.

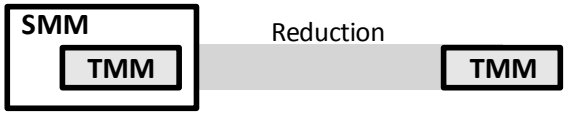


FIGURE 2.8 – Réduction d'un méta-modèle

Concernant l'exemple, la figure 2.9 donne le sous-ensemble du DSML qui est censé correspondre au méta-modèle de l'outil. Afin de transformer SMM en TMM, le sous-ensemble produit de la réduction, doit subir quelques modifications : *name* est déplacé de Vertex vers State, *vertices* est remplacé par *states*.

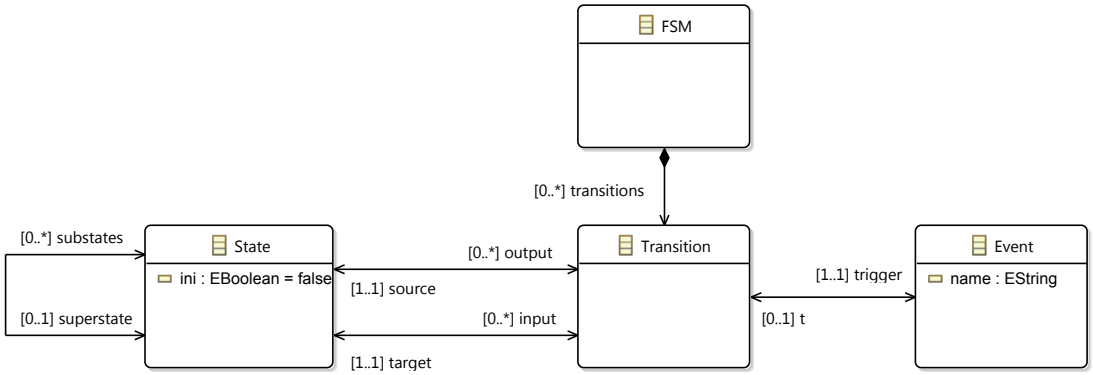


FIGURE 2.9 – Sous-ensemble du DSML

3. Co-évolution. La mise en relation de SMM avec TMM est préalable à la migration de données conformes à SMM vers TMM. À partir des différences entre les méta-modèles, on peut déduire les opérateurs de migration à effectuer sur les méta-modèles (principes de la co-évolution illustrés par la figure 2.10).

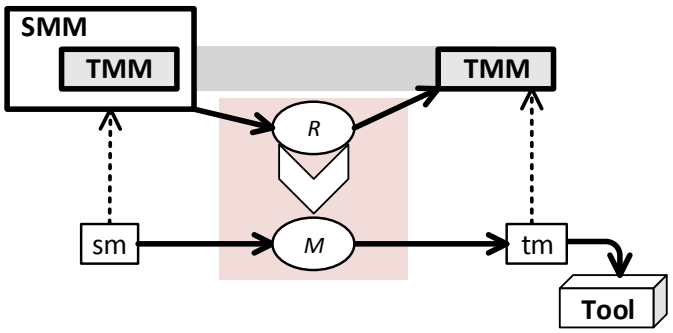


FIGURE 2.10 – Évolution de méta-modèle et co-évolution de modèles

Une fois la co-évolution appliquée au modèle de la figure 2.2 (sm), le résultat obtenu est celui illustré par la figure 2.4 (tm). L'outil peut être appliqué sur ce dernier et il donne le résultat présenté par la figure 2.5. tm est un résultat intermédiaire du processus de réutilisation, parce que ce modèle doit être rendu conforme au méta-modèle de machine à états finis et les éléments supprimés doivent être ré-injectés.

4. Inversion de migration. On s'intéresse ici au cas complexe d'un *outil de réécriture*. Il s'agit d'un outil qui fait des modifications sur un modèle pour produire en sortie, le modèle d'entrée raffiné. La réutilisation de ce type d'outils pose un problème difficile. Comme illustré par la figure 2.11, les données (sm) qui ont été traduites (tm) et postérieurement modifiées (pm) par l'outil de réécriture (Flatten) doivent être traduites dans leur contexte d'origine. Pour aborder ce problème, la migration doit être inversable. Ceci implique que la co-évolution doit être étendue (un opérateur doit permettre la migration et la migration inverse). En particulier, les données supprimées lors de la migration doivent être récupérées et remises (recontextualisées) dans le modèle inversé (rm).

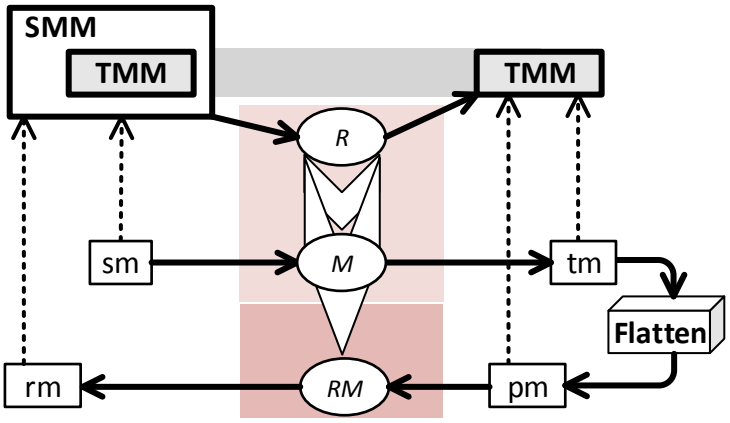


FIGURE 2.11 – Inversion de migration

La figure 2.12, illustre l'inversion de la migration dans notre exemple. Les actions a1 et a2 sont récupérées. Il y a différentes possibilités de les recontextualiser :

- a1 et a2 deviennent des actions associées à ABC.
- a1 devient une action associée à ABC, et a2 devient une action associée à D.
- a1 devient une action associée à D, et a2 devient une action associée à ABC.
- a1 et a2 deviennent des actions associées à D.

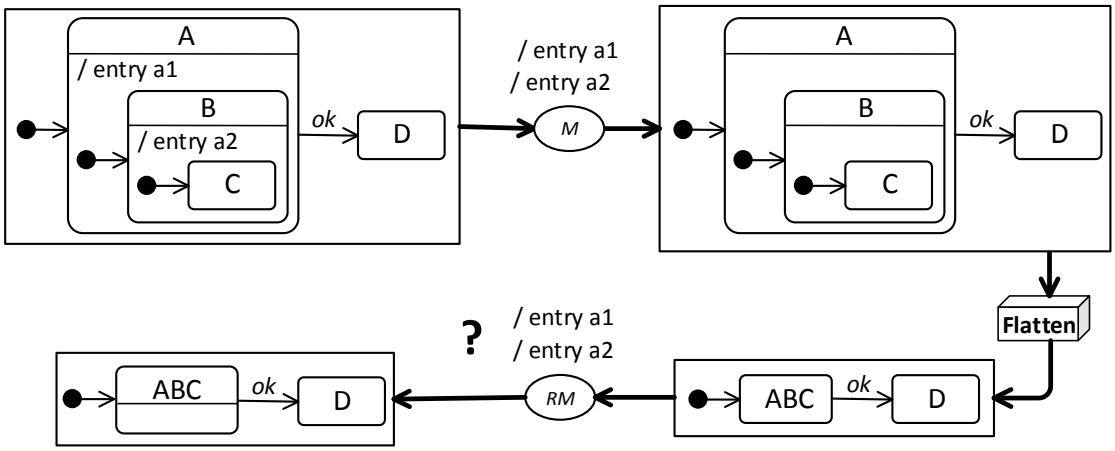


FIGURE 2.12 – Exemple d'inversion de migration

Quelle est donc la recontextualisation correcte ? Peut-on garantir qu'aucune action récupérée n'est perdue ? Pour répondre à ces questions, des approches qui permettent la différentiation, la réduction, la co-évolution et la migration inverse sont présentées dans la suite et sont évaluées du point de vue de notre objectif de réutilisation.

2.3 Différenciation de méta-modèles

Certaines des approches qui permettent de faire des comparaisons de méta-modèles seront étudiés par la suite.

EMF Compare utilise la différentiation par *similarité* [BP08]. Selon leur type, les différences sont classifiées en additions (*Additions*), mises à jour (*Changes*), déplace-

ments (*Moves*) et suppressions (*Deletions*). À titre d'illustration, la figure 2.13 présente les différences du méta-modèle du domaine d'application (figure 2.1) par rapport au méta-modèle du domaine de définition de l'aplatisseur (figure 2.3) que nous allons réutiliser. EMF Compare a identifié la présence de deux classes (Vertex et Action) et une relation d'héritage (Vertex est un *SuperType* de State) de plus. Il a également identifié une modification dans le nom de la référence *vertices* (illustré dans la partie inférieure de la figure). Enfin, il a identifié que l'attribut *name* de Action a été déplacé vers State. Les différences trouvées coïncident presque entièrement avec les différences décrites et présentées par la figure 2.7. Cependant, l'attribut *name* de State a été déplacé depuis Vertex et non depuis Action.

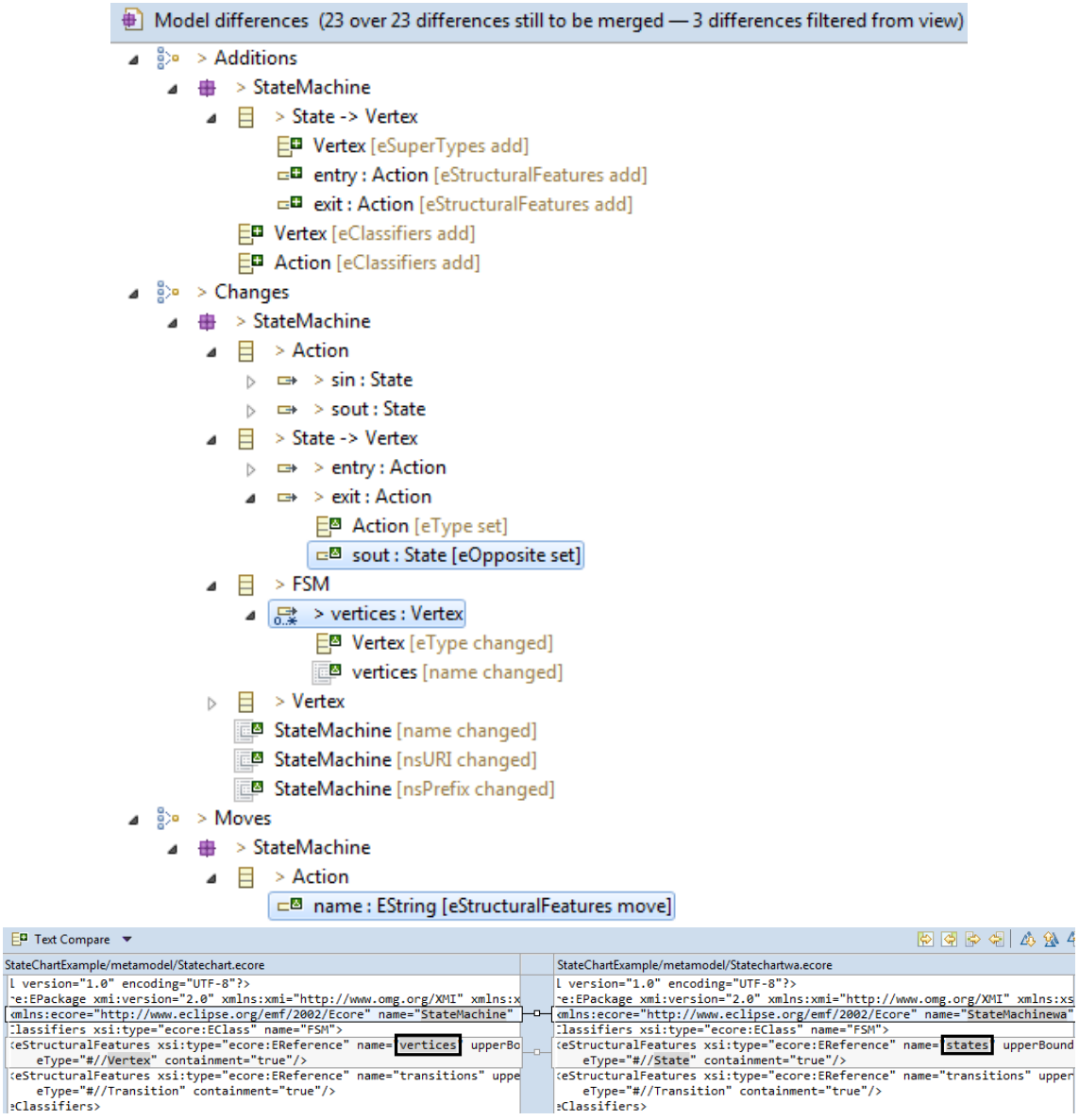


FIGURE 2.13 – Différences entre deux méta-modèles avec EMF Compare

UMLDiff [XS05] et SiDiff [SG08] sont des outils qui utilisent une approche basée sur des similarités définies dans le *model differencing*. Appliqué à notre exemple, la figure 2.14 illustre les différences entre le méta-modèle de la figure 2.1 et le méta-modèle

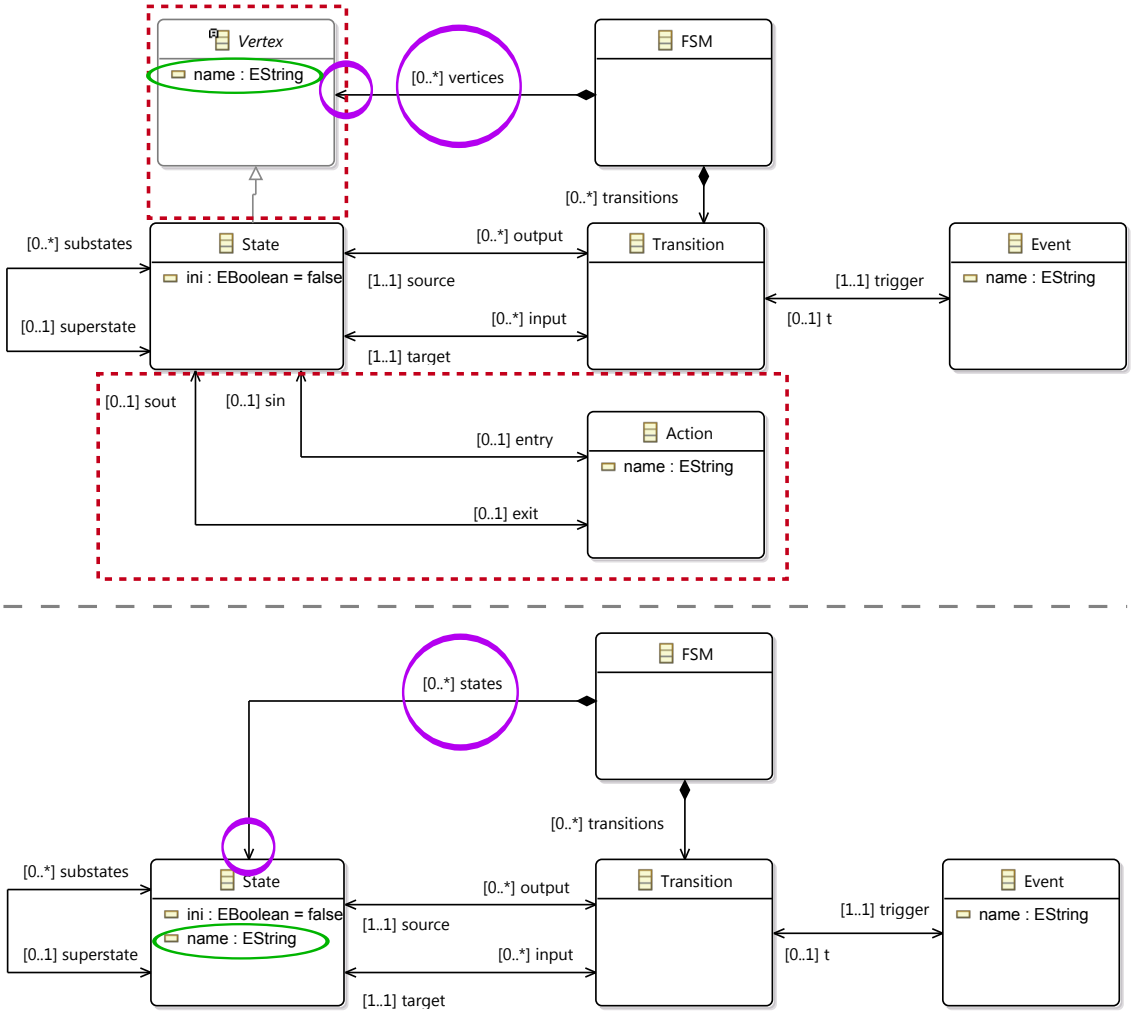


FIGURE 2.14 – Différences entre deux méta-modèles

de la figure 2.3. Un cercle indique une différence détectée. Une ellipse représente le déplacement d'un élément. Un carré pointillé indique des éléments qui sont présents dans un méta-modèle mais pas dans l'autre.

Cicchetti [CRP07] propose une approche indépendante du méta-modèle pour représenter les différences entre deux méta-modèles. Les modifications sont classifiées selon trois catégories : *addition*, *suppression* et *mise à jour*. Elles sont définies dans un modèle de différences. Ce dernier peut être ensuite utilisé pour *reconstruire* le méta-modèle cible à partir du méta-modèle source. Dans notre exemple, l'attribut *name* de State est identifié comme une addition au lieu d'un déplacement.

Les approches de différentiation offrent des solutions importantes pour l'identification des différences et des ressemblances lorsqu'elles sont simples. Cependant, les différences résultantes des opérations plus complexes comme le déplacement ne peuvent pas être déduites automatiquement [RKPP10]. Dans ces cas la déduction doit s'appuyer sur des décisions humaines basées sur le savoir faire ou certaines heuristiques. Heuvel [Heu04] propose une approche pour intégrer une aide à l'utilisateur et des heuristiques dépendantes du modèle. Nous devons considérer la différentiation automatique, mais la possibilité d'intégrer des décisions humaines ne doit pas être exclue. Car ces décisions tiennent compte de la sémantique des domaines représentés

par les méta-modèles. Enfin, la différentiation ne propose des solutions qu'à une des problématiques de la réutilisation, les sections suivantes abordent les autres problématiques.

2.4 Réduction de méta-modèles et de modèles

Comme précisé dans la section 2.2, nous supposons que toutes les données nécessaires à l'outil sont disponibles dans le domaine d'application. L'extraction de la partie pertinente d'un méta-modèle peut être obtenue grâce au *pruning* [SMBJ09] tandis que l'extraction d'une partie d'un modèle peut être obtenue grâce au *slicing* [BCBB11].

Le *pruning* désigne la suppression des classes et des propriétés qui ne sont pas nécessaires dans le méta-modèle (*meta-muddle*) [SF05] afin de ne conserver que les concepts nécessaires pour répondre à un problème d'un domaine spécifique à un moment donné [SMBJ09]. La partie extraite est appelée *méta-modèle effectif*. Le concept de *sous-typage* garanti que tous les modèles conformes au méta-muddle sont également conformes au méta-modèle effectif.

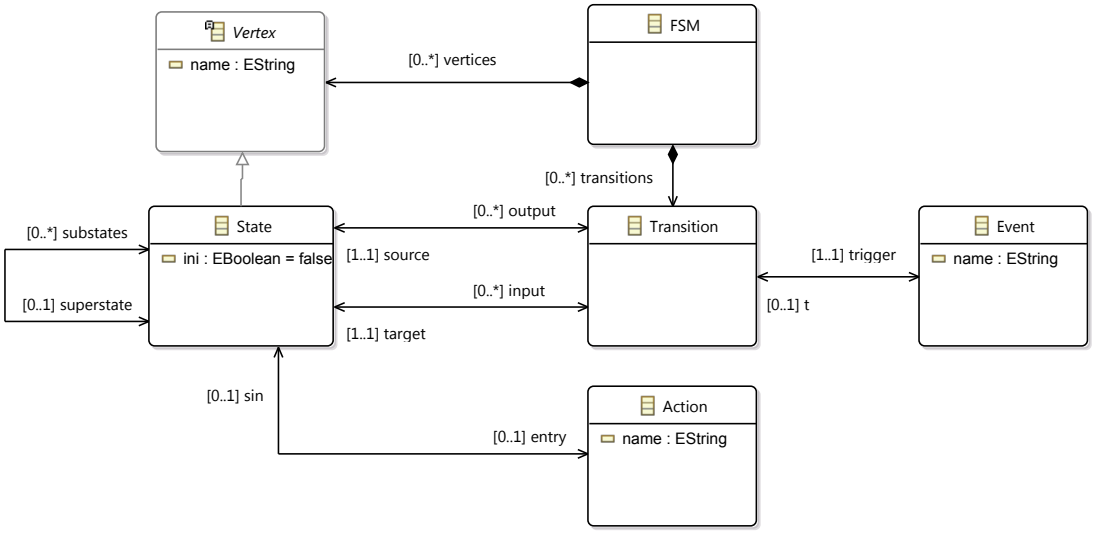


FIGURE 2.15 – Méta-modèle effectif du méta-modèle de la figure 2.1

Suivant notre exemple, la figure 2.15 illustre une partie extraite du méta-modèle de la figure 2.1. *sout* et *exit* n'en font plus partie. Dans cet exemple, le pruning ne suffit pas pour obtenir la partie du méta-modèle qui coïncide avec le méta-modèle de l'aplatisseur parce que :

1. Le pruning préserve les noms des éléments du méta-muddle, ceci empêche de renommer *vertices* en *states*.
2. Le prunnig préserve les supertypes (*superClasses*) des classes qui doivent faire partie du méta-modèle effectif, raison pour laquelle la classe Vertex est nécessaire.
3. D'après le principe du pruning, le modèle de la figure 2.2 doit être conforme au méta-modèle de la figure 2.1 et au méta-modèle effectif, alors Action, *entry* et *sin*

doivent être préservées. Les seules références qui ne sont pas utilisées sont *exit* (et sont opposée *sout*).

L'approche proposé par Pham [Pha12] permet de déterminer le méta-modèle effectif qui est utilisé par une transformation de méta-modèles. Ceci permet d'adapter la transformation pour qu'elle n'utilise que les éléments pertinents du méta-modèle.

Les *slicers* sont des outils qui extraient un sous-ensemble d'un modèle, pour un objectif donné. Ils permettent aux concepteurs de modèles de recueillir rapidement les connaissances pertinentes à partir d'un modèle, qui est en général de grande taille. Un *slice* est donc, un sous-modèle qui respecte par défaut toutes les contraintes structurelles imposées par le méta-modèle (mode *strict*). Le mode *soft* consiste à ajouter des *opposite*s sur les relations du méta-modèle, à ajouter des contraintes pour filtrer les éléments tranchés ou à élargir le format de sortie du *slice*.

Blouin [BCBB11] propose une approche générative pour produire automatiquement des *model slicers*. Le slice produit par défaut contient les classes et les propriétés préalablement choisies, de plus, il contient tous les éléments qui sont nécessaires pour que le slice soit un modèle valide par rapport au méta-modèle cible. Le framework *Kompren*[1] permet de définir des paramètres pour établir les *model slicers*. Les objectifs de cette approche sont de construire automatiquement des *model slicers* pour n'importe quel DSML et de spécialiser les *model slicers* capables d'extraire différentes formes de slice. Un exemple d'application à grande échelle du slicing est la définition d'un aplatisseur de *State Machine* UML. Seul le diagramme de classes UML et les éléments de *State Machine* constituent le slice. Tous les autres éléments sont écartés.

Appliqué à l'exemple introductif, la figure 2.16 illustre un slice du modèle de la figure 2.2, il s'agit d'un sous-modèle qui ne contient pas d'actions. Ce modèle semble être conforme au méta-modèle de l'aplatisseur mais ce n'est pas le cas parce que le modèle reste conforme à un modèle constitué de *vertices* (référence *vertices*) et il nous faut un modèle constitué de *states* (référence *states*). De ce fait, il n'est pas une entrée valide pour l'aplatisseur.

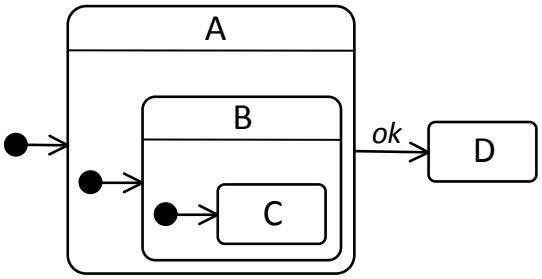


FIGURE 2.16 – Slice du modèle de la figure 2.2

Le *slicing* et le *pruning* sont adaptées pour faire des réductions. Il s'agit d'approches prometteuses pour répondre à la question 1. Pour notre objectif de réduction, nous conservons l'idée de deux aspects : (1) la réduction se fait au niveau méta-modèle et au niveau modèle. Nous envisageons une approche qui intègre la réduction aux deux niveaux. (2) dans certains cas, comme dans notre exemple (*i.e.* renommer *vertices*), les données peuvent être structurées différemment. Dans ce cas, il faut réduire le méta-modèle source mais aussi l'adapter. Nous cherchons donc une solution qui intègre la

1. https://people.irisa.fr/Arnaud.Blouin/software_kompren.html

réduction et des transformations éventuelles de refactoring. Nous considérons que la création de données n'est pas nécessaire puisque toutes les données nécessaires sont présentes dans le domaine d'application.

2.5 Évolution de méta-modèle et co-évolution de modèles

La co-évolution est un principe mis en œuvre dans plusieurs domaines de l'informatique, *e.g.* évolution de schémas, évolution de grammaires et évolution de formats de données [Lä04]. La co-évolution utilise les *mapping* et les *différences* (*c.f.* section 2.3) pour déduire les actions à appliquer pour faire co-évoluer correctement les modèles conformes à la version initiale d'un méta-modèle.

2.5.1 Spécification manuelle des migrations

MCL (*Model Change Language*) [NLBK09], [LBNK10] est un langage pour spécifier les modifications à appliquer à un méta-modèle source pour qu'il devienne un méta-modèle évolué. Les auteurs de cette approche font l'hypothèse que les modifications à appliquer sont mineures en taille et en complexité et que l'addition des nouveaux éléments nécessite d'une intervention humaine au moment de la migration. La spécification des modifications est utilisée pour migrer automatiquement les modèles conformes au méta-modèle source. MCL ne permet que la définition des correspondances entre les éléments du méta-modèle qui ont changé. Dans notre exemple, MCL peut être utilisé (figure 2.17) :

- Pour renommer *vertices* en *states* (① *MapsTo states*).
- Pour supprimer la classe Vertex (② *MapsTo NULL*).
- Pour supprimer Action (③ *MapsTo NULL*).
- Pour ajouter l'attribut *name* à la classe State (④ *MapsTo* State).

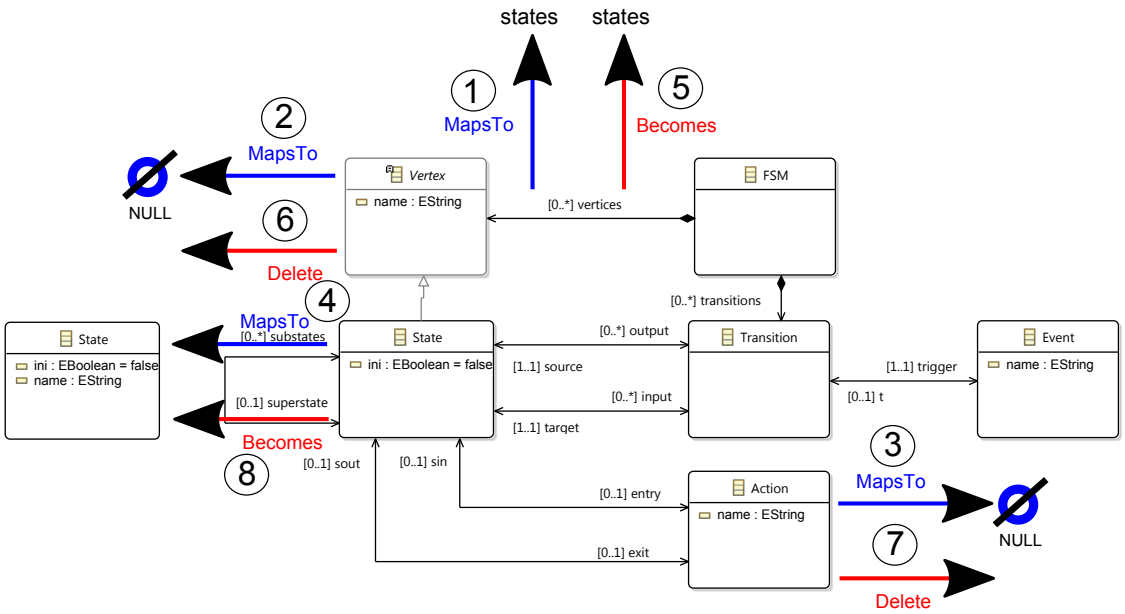


FIGURE 2.17 – Spécification manuelle des migrations avec MCL

Encore une fois *name* de Vertex n'est pas déplacé mais il est supprimé et un attribut *name* est ajouté à State. Ceci implique que l'utilisateur doit définir certaines règles pour permettre que la valeur de *name* soit conservée ou il doit intervenir pour indiquer à la main la valeur de l'attribut *name*. Le modèle migré obtenu est celui de la figure 2.16.

Sprinkle [SK04] présente un langage de transformation de graphes qui ne nécessite que la spécification des différences. Les éléments qui ne sont pas notifiés, sont implicitement copiés. Dans notre exemple, cette approche est utilisée :

- Pour renommer *vertices* (⑤ *Becomes states*).
- Pour supprimer *Vertex* (⑥ *Delete*).
- Pour supprimer Action (⑦ *Delete*).
- Pour ajouter l'attribut *name* (⑧ *CreateWithin*).

Cette approche ne facilite pas le déplacement de *name* et la cible de *states* doit être modifiée manuellement. La spécification manuelle des migrations offre une certaine souplesse et facilite la personnalisation des migrations. Par contre, l'utilisation de cette approche est sujette aux erreurs et peut donner lieu à des incohérences entre le méta-modèle et les modèles [Wac07]. Nous avons besoin d'une approche de co-évolution qui donne la priorité aux opérateurs de réduction, mais qui inclut en même temps des opérateurs plus complexes.

2.5.2 Relations explicites d'appariement entre concepts

Cicchetti [CREP08] détecte les changements complexes dans la différence entre deux versions d'un méta-modèle. Il génère une transformation de haut niveau (*Higher-Order model Transformation*) qui s'applique au méta-modèle source pour produire ensuite une transformation ATL. La transformation est elle-même appliquée au modèle source. ATL (*Atlas Transformation Language*) est un langage qui génère des migrations adaptables [JK06b], [JK06a].

Garcés [GJCB09] présente un langage de *matching* qui facilite la personnalisation du processus de *matching* et permet de rajouter de nouveaux patrons. Falleri [FHLN08] propose une approche qui détecte le *mapping* entre deux méta-modèles et l'utilise pour générer une concordance entre les méta-modèles. La concordance doit être vérifiée manuellement et peut être utilisée pour générer des modèles de transformation. L'approche transforme les méta-modèles en graphes étiquetés, puis elle utilise l'algorithme de *similarity flooding* pour déterminer les liens entre les deux méta-modèles.

Appliquées à notre exemple, les approches de Cicchetti et Garcés produisent un code de migration, dont un extrait est présenté par le listing 2.1. Il permet de produire un modèle migré comme celui de la figure 2.16 à partir du modèle de la figure 2.2.

Listing 2.1 – Migration par relation explicite entre concepts

```
rule State2State {
 from s : MM1!State ...
 to t : MM2!State (
    name <- s.name, ... ) }

rule Action2Drop {
 from s : MM1!Action ( ... )
 to drop }
```

Un des bénéfices des travaux réalisés dans le cadre de cette approche est que les différences entre les méta-modèles sont identifiées de manière automatique. Cependant, ces liens peuvent être difficiles à identifier, surtout quand un élément de la cible expérimente plusieurs transformations ou des transformations complexes (*e.g.* déplacement d'un attribut d'une classe vers une autre). La migration des modèles n'est pas unique parce qu'elle change en fonction du *mapping* déduit.

2.5.3 Opérateurs de co-évolution

Cicchetti [CDRP09] propose une approche basée sur des opérateurs réutilisables (*i.e.* qui peuvent être appliqués à n'importe quel méta-modèle) et automatiques. Elle réduit l'effort associé à la construction d'un modèle de migration. Par contre, elle ne permet d'adapter la migration en fonction des besoins spécifiques de l'utilisateur.

Rüeeg [RMvH11] et Agrawal [AKS03] évoquent le besoin d'avoir des outils pour automatiser et accélérer le processus de migration de modèles. Cependant, certaines modifications au niveau méta-modèle demandent des informations additionnelles pendant la migration. Ces informations ne sont pas disponibles dans le modèle. Par conséquent, elles ne peuvent pas être entièrement automatisées. L'interaction de l'utilisateur est permise seulement lorsque l'information manquante doit être fournie.

La proposition de Herrmannsdöerfer [HK10] est une extension d'une approche à base d'opérateurs. Elle consiste à automatiser la migration de modèles pour assurer la préservation de la sémantique à condition qu'elle soit explicitée dans le modèle source. Edapt [Eda15] s'appuie sur des opérateurs de co-évolution pour générer automatiquement des migrations adaptables à partir de la mise à jour directe de code Java.

EMFMigrate [EMF15b] est un langage visant à soutenir la co-évolution de manière générale, il ne se limite pas à des types spécifiques d'artefacts [WIRP12]. Il ne propose pas d'opérateurs de co-évolution, mais des constructeurs dédiés permettant de spécifier des librairies de migration personnalisables. Refactory [Ref15] adapte les modèles de façon automatique. La stratégie de migration n'est pas modifiable en soi, mais il permet de définir des post-processeurs qui peuvent être utilisés pour mettre en œuvre de nouvelles transformations du modèle après l'exécution des opérateurs génériques.

Gruschko [BP07] et Wachsmuth [Wac07] ont par hypothèse que les opérateurs sont atomiques, c'est-à-dire qu'ils sont appliqués un par un de façon individuelle. Dans l'approche proposée par [CREP08], par contre, les modifications sont composables, ils peuvent détecter les modifications en utilisant son approche basée sur les différences.

Compte tenu d'un opérateur existant, on peut toujours envisager des variantes des opérateurs, ayant le même effet sur le niveau de méta-modèle, mais avec une migration un peu différente. Mais plus le nombre d'opérateurs de co-évolution augmente, plus il est difficile de trouver un opérateur approprié. Et parmi plusieurs opérateurs similaires, il est difficile de décider lequel appliquer. Pour cette raison les *macro-opérateurs* qui intègrent une séquence d'opérateurs réduisent le nombre d'opérateurs, mais ne limitent pas la complexité des transformations à exécuter.

Modif [BK11] [KB11] est un langage dédié à la description des évolutions d'un méta-modèle et à la co-évolution des modèles. Dans Modif, les transformations sont faites à partir d'une *spécification de refactoring*. Modif génère automatiquement les spécifications de migration et offre à l'utilisateur la possibilité de faire des modifications du code de migration comme dans l'approche par spécification manuelle (section 2.5.1).

TABLEAU 2.1 – Opérateurs simples de Modif

Operator	Parameter	Implementation
rename	*package* *class* *attribute* *reference*	If *package* Then If *name* is not yet defined by the package's parent Then *Rename(package, name)* Else If *class* If *name* is not yet defined by the class's parent Then *Rename(class, name)* Else If *attribute* If *name* is not yet defined by the attribute's parent Then *Rename(attribute, name)* Else If *reference* If *name* is not yet defined by the reference's parent Then *Rename(reference, name)*
remove	*package* *class* *attribute* *reference*	If *package* Then ForAll *class* in *package* *Remove(class)* ForAll *sub-package* in *package* *Remove(sub-package)* *Remove(package)* Else If *class* ForAll *attribute* in *class* *Remove(attribute)* ForAll *reference* in *class* *Remove(reference)* *Remove(class)* Else If *attribute* *Remove(class)* Else If *reference* *Remove(reference)*
change multiplicity	*attribute*	If lower bound is 0 and upper bound is 1 Then *Generalize Attribute(attribute, 0, *)* Else If lower bound is 1 and upper bound is 1 *Generalize Attribute(attribute, 1, *)*
change abstract	*class*	If *class* is abstract Then *Drop Class Abstract(class)* Else *Make Class Abstract(class)*
change containment	*reference*	If *reference* is containment Then *Drop Reference Containment(class)* Else *Make Reference Containment(reference)*

TABLEAU 2.2 – Opérateurs complexes de Modif

Operator	Parameter	Implementation
flatten	*class*	ForAll *subClass* of *class* *Unfold Super Class(class, subClass)* ForAll *otherClass* different from *class* ForAll reference from *otherClass* to *class* ForAll *subClass* of *class* *Create Reference(otherClass, name, subClass, lowerBound, upperBound, false)*
hide	*package* *class*	If *package* Then If *package* is not the root package Then Let *parentPackage* be the parent package of *package* in If *package* has sub-packages Then ForAll *subPackage* of *package* If *subPackage* has classes Then ForAll *class* in *subPackage* *Change Package(class, parentPackage)* Else *Change Package(subPackage, parentPackage)* Else ForAll *class* in *package* *Change Parent(class, packageParent)* *Remove(package)* Else If *class* ForAll *superClass* of class ForAll *subClass* of class *AddSuperClass(superClass, subClass)* ForAll *incommingReference* of class ForAll *reference* of class *AddDerivedReference*(*incommingReference*.source, *reference*.target, *incommingReference*.lowerBound* *reference*.lowerBound, *incommingReference*.upperBound* *reference*.upperBound) ForAll *reference* of class *Remove(reference)* ForAll *attribute* of class *Remove(attribute)* *Remove(class)*

Le tableau 2.1 présente les opérateurs simples de Modif. Ils permettent d'effectuer une évolution du méta-modèle dans une étape atomique. Le tableau 2.2 présente les opérateurs complexes de Modif. Ils peuvent être décomposés dans une séquence d'opérateurs simples.

La figure 2.18 illustre le résultat après application de l'opérateur hide sur la classe Vertex du méta-modèle de la figure 2.1. la classe Vertex est masquée, FSM et State sont reliées via la référence *vertices_State*.

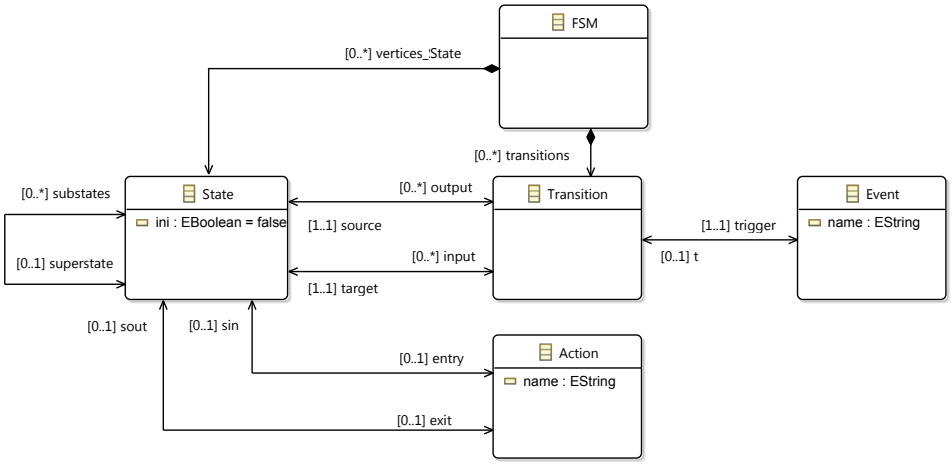


FIGURE 2.18 – Application de l'opérateur hide de Modif

La figure 2.19 illustre le résultat d'appliquer l'opérateur flatten sur la classe Vertex du méta-modèle de la figure 2.1. L'attribut *name* est déplacé vers la sous-classe de Vertex.

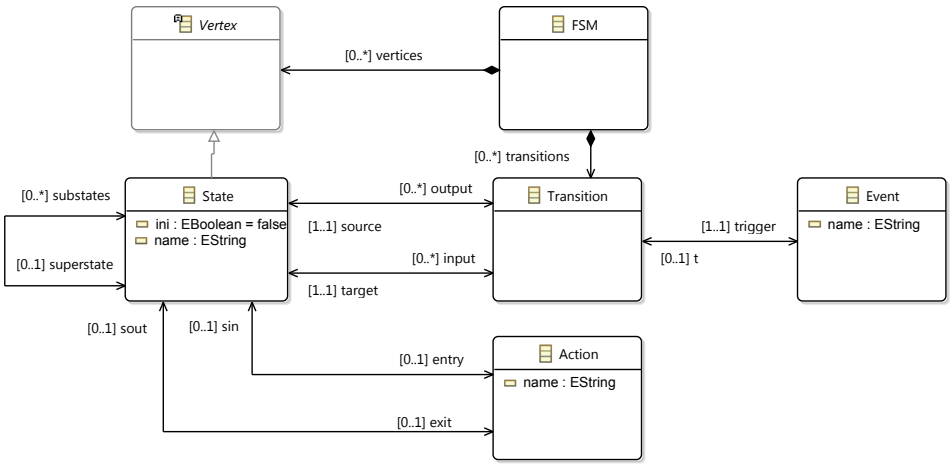


FIGURE 2.19 – Application de l'opérateur flatten de Modif

Le listing 2.2 illustre la syntaxe textuelle des opérateurs Modif utilisés pour migrer le modèle de notre exemple (figure 2.2) à un modèle conforme au méta-modèle de la figure 2.3 (le modèle migré coïncide avec le modèle de la figure 2.16). Ce langage est celui utilisé pour la suite.

Listing 2.2 – Synstaxe textuelle Modif

```
class {
        FSM { ref vertices to states bounds (0,-1)
              ref transitions bounds (0,-1) } ;
        flatten hide Vertex { att name bounds (0,1) } ;
        State { att ini bounds (0,1)
                ref input bounds (0,-1)
                ref output bounds (0,-1)
                ref entry bounds (0,1)
                ref exit bounds (0,1)
                ref superstate bounds (0,1)
                ref substates bounds (0,-1) } ;
        Transition { ref trigger bounds (1,1)
                     ref target bounds (1,1)
                     ref source bounds (1,1) } ;
        Event { ref t bounds (0,1)
                att name bounds (0,1) } ;
        remove Action { att name bounds (0,1)
                        ref sin bounds (0,1)
                        ref sout bounds (0,1) }
}
```

Le tableau 2.3 illustre une partie des approches précédemment citées. Chaque ligne du tableau présente les opérateurs appliqués au méta-modèle d'application de l'aplatisseur de notre exemple, afin : de cacher la classe Vertex ; de déplacer l'attribut *name* de Vertex vers State ; de renommer la référence *vertices* ; et de supprimer la classe Action. Nous pouvons apprécier que Modif ne nécessite que l'application d'un opérateur pour chacune des actions requises. Par exemple, l'opérateur remove appliqué à une classe, supprime la classe, ses attributs, ses références et toutes les références qui pointent vers la classe supprimée. Les autres approches, par contre doivent supprimer un par un tous les éléments de la classe et tout ce qui se réfère à la classe avant de la supprimer. Les opérateurs flatten et hide cachent la classe Vertex et préservent l'attribut *name*. Par rapport aux travaux précédemment cités, Modif introduit des opérateurs originaux (*i.e.* hide) qui augmentent l'efficacité de la spécification de la réduction.

Guerra [GdLKP10] présente un langage déclaratif, formel, visuel, et de haut niveau permettant de spécifier des transformations exogènes de modèles. Son but n'est pas d'implémenter les transformations mais d'exprimer ce que la transformation doit faire et les propriétés auxquelles les modèles transformés doivent répondre. Ce langage fournit du support pour l'analyse et la conception des transformations.

Kermeta [MFJ05] explore l'idée d'utiliser la modélisation orientée aspects pour permettre de construire un méta-langage exécutable en composant des *action metamodels* avec des méta-langages existants.

Il existe d'autres langages qui permettent de transformer des modèles tels que : VIATRA [CHM+02], qui génère automatiquement des migrations adaptables à partir de la mise à jour directe du code Java généré [VB07], Kent Model Transformation Language [AHMM05], Tefkat [Tef15], GReAT [GRe15], UMLX [UML15], AToM [LV02], BOTL [BMV+03], MOLA [MOL15], Weaving MTL [SFS05], YATL [Pat04]. Il existe également de nombreux outils libres qui implémentent ce type d'approches tels que : Fujaba [Fuj15] et MTF [MTF15]. Czarnecki [CH06] et Rose [RPKP09] présentent des comparaisons entre un ensemble d'outils et langages de transformation de modèles.

Les langages et frameworks précédemment présentés fournissent des solutions pour répondre à la question 3. Ensuite nous allons explorer les solutions pour répondre à la question 4 (migrer les modèles transformés par l'outil).

TABLEAU 2.3 – Comparaison des opérateurs

Approach	Operators
Cicchetti [CDRP09]	1. Rename metaelement (vertices, FSM, states) 2. Eliminate metaproperty (entry, State) 3. Eliminate metaproperty (exit, State) 4. Eliminate metaproperty (sin, Action) 5. Eliminate metaproperty (sout, Action) 6. Eliminate metaproperty (name, Action) 7. Eliminate metaclass (Action) 8. Change metaproperty type (Vertex, states, State) 9. Flatten hierarchy (Vertex)
Edapt/COPE [HVW10]	1. Delete Opposite Reference (entry, State) 2. Delete Feature (entry, State) 3. Delete Opposite Reference (exit, State) 4. Delete Feature (exit, State) 5. Delete Feature (sin, Action) 6. Delete Feature (sout, Action) 7. Delete Feature (name, Action) 8. Delete Class (Action) 9. Create Reference (states, FSM, State) 10. Make Ref. Comp. (states, FSM) 11. Switch Reference Composite (states, vertices) 12. Delete Feature (vertices, FSM) 13. Inline Super Class (Vertex)
Wachsmuth [Wac07]	1. Eliminate property (entry, State) 2. Eliminate property (exit, State) 3. Eliminate class (Action) 4. introduce property (FSM, states, State) 5. Eliminate property (vertices, FSM) 6. flatten hierarchy (Vertex) 7. introduce property
Modif [BK11]	1. Rename (vertices, FSM, states) 2. Flatten (Vertex) 3. Hide (Vertex) 4. Remove (Action)

2.6 Inversion de migration

L'objectif d'inverser une migration est de migrer les modèles qui avaient été déjà migrés afin de les rendre à nouveau conformes au méta-modèle d'origine (méta-modèle source dans le cadre de la co-évolution). Les transformations dites *bidirectionnelles* [CFH+09] ont vocation à maintenir la cohérence entre deux sources d'information connexes.

Stevens [Ste08] cite deux approches pour réaliser les transformations bidirectionnelles :

- La migration (source vers cible) et la migration inverse (cible vers source) sont implémentées indépendamment en utilisant un langage de transformations unidirectionnelles (par exemple ATL).
- La migration et la migration inverse sont définies simultanément avec un langage de transformations bidirectionnelles (par exemple QVT [ACE+03]).

Dans les deux cas, il faut garantir que la deuxième transformation est l'inversion de la première et que leur composition correspond à l'identité.

L'approche par *lentilles* (*lenses*) [Wid11] aborde les transformations bidirectionnelles. Elle spécifie la façon d'adapter un modèle pour qu'il coïncide avec un autre qui a été modifié. La partie délicate de la spécification des lentilles est la définition de la migration inverse. Notamment, si la migration supprime des éléments du modèle, il faut les restaurer de façon appropriée. La difficulté provient de choisir une restauration appropriée parmi les différentes solutions possibles. En l'absence d'une option préférable, le choix est dans les mains du concepteur de la transformation [GMPS03].

Dans notre exemple, le modèle modifié par l'outil est présenté par la figure 2.5. Pour le rendre à nouveau conforme au méta-modèle de la figure 2.1 il faut d'abord rétablir la référence *vertices*, ce qui est facilement automatisable. Par contre, il est difficile a priori de remettre les actions a1 et a2 qui avaient été supprimées lors de la migration (figure 2.20). En effet, les états conteneurs (A et B) de ces actions ont été supprimés. Dans ce cas, la restauration nécessite une intervention humaine soit pour ajouter les actions à ABC, soit pour les ajouter à D, soit pour les oublier. Cet exemple met en évidence le besoin de garder une trace des éléments des modèles.

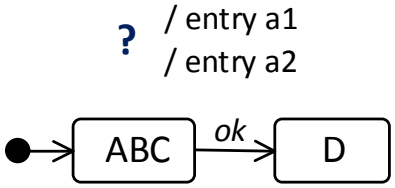


FIGURE 2.20 – Lentilles [GMPS03]

Il montre aussi que la migration inverse peut être délicate lorsque le modèle migré intègre des nouveaux éléments ou lorsque des éléments existants ont été fusionnés. Dans les deux cas, il faut se poser la question de l'intégration des nouveaux éléments au contexte d'origine.

En général, les approches de transformations bidirectionnelles s'appuient sur un round-trip de migration qui implique un modèle initial, une migration et une migration inverse. Les approches de transformation bidirectionnelles ne tiennent pas

compte des transformations faites par l'outil. Elles prennent le modèle modifié par l'outil comme étant un modèle sans aucun rapport avec le modèle migré, ni avec le modèle initial. Ces approches ne permettent pas alors d'automatiser la restauration des informations supprimées lors de la migration, surtout si l'outil crée des informations nouvelles qui peuvent être recontextualisables de façons différentes. Pour atteindre notre objectif, le round-trip de migration, devrait intégrer lors de la migration inverse, les opérateurs inverses des opérateurs appliqués lors de la migration. Ceci requiert des mécanismes d'identification des objets tout au long du round-trip.

Identifiants Langer [LWG+12] propose une approche d'identification par *signifiers* pour étendre les systèmes de contrôle de version de modèles et faire face à des problèmes liés à la fusion (*merge*) des éléments du modèle dans le contexte de la construction de modèles par équipes. Modif [KB11] implémente un mécanisme de *clés* pour identifier les instances des modèles et garder une trace des transformations aux instances.

En général, les systèmes de contrôle de versions s'appuient sur des identifiants uniques *UUID* (Universally Unique IDentifier) pour déterminer si un élément d'un modèle correspond à un autre dans deux versions successives d'un modèle : si deux éléments partagent le même UUID, ils sont considérés comme deux versions du même élément du modèle. Autrement ils sont considérés comme étant deux éléments indépendants.

2.7 Synthèse

Il existe de nombreuses approches qui répondent aux quatre questions posées dans la section 2.2. Les techniques de réduction permettent de cibler uniquement les données nécessaires pour résoudre une problématique ciblée. La problématique de la migration inverse suite à des modifications appliquées au modèle migré est plus originale. Les approches de réduction permettent de cibler un ensemble de données. Cependant elles ne sont pas suffisantes dans le contexte de la réutilisation d'un outil de réécriture.

Afin d'évaluer les approches de différentiation et de réduction existantes, nous introduisons les critères suivants :

① Déduction automatique des différences : les différences entre les deux méta-modèles sont calculées de façon automatique.
② Facilités de réduction : l'approche facilite la réduction du méta-modèle.
③ Facilités de refactoring : l'approche permet d'appliquer du refactoring à un méta-modèle.

Le tableau 2.4 présente une évaluation de quelques solutions existantes (qui proposent des réponses aux questions 1 et 2 vis à vis de ces critères.

TABLEAU 2.4 – Comparaison des approches de différentiation et de réduction

Approach / Tool	①	②	③
EMF Compare [BP08]	✓		
UMLDiff [XS05]	✓		
SiDiff [SG08]	✓		
Cicchetti [CRP07]	✓		✓
Pruning [SMBJ09]		✓	
Slicing [BCBB11]	✓	✓	
Blouin [BCBB11]	✓	✓	
MCL [NLBK09]		✓	✓
Sprinkle [SK04]		✓	✓
Epsilon Flock [RKPP10]			✓
Modif [BK11]		✓	✓

Il existe de nombreuses solutions au problème des migrations et migration inverse que nous traitons dans cette thèse. Afin de les synthétiser, nous introduisons les critères suivants qui permettent de faire une évaluation au niveau modèle :

① Génération de la migration : le modèle de migration de modèles doit être généré automatiquement, suite aux transformations appliquées au niveau méta-modèle.

② Génération conjointe de la migration inverse : typiquement le problème de la migration inverse se pose en termes de migrations simples, comme si migration et inversion de la migration étaient deux problèmes symétriques. Elle doit être générée automatiquement et conjointement à la migration.

③ Cohérence : la composition de la migration et la migration inverse produit l'identité.

④ Adaptabilité : la migration obtenue par défaut doit pouvoir être adaptée et éditée afin de prendre en compte des cas très spécifiques de migration.

⑤ Prise en compte de l'action de l'outil : la migration inverse doit pendre en compte l'action de l'outil dans le domaine cible afin de garantir qu'elle même ne défait pas l'action de l'outil.

Le tableau 2.5 présente une évaluation des principales solutions existantes (qui proposent des répondes aux questions 3 et 4 vis à vis de ces critères. Aucune des approches ne répond à tous les critères.

Les tableaux 2.4 et 2.5 résument les différentes approches étudiées dans ce chapitre. Nous avons dans un premier temps abordé la problématique de la différentiation de méta-modèles. Ensuite nous avons abordé les problématiques liées à la réduction de méta-modèles et modèles. Puis nous avons abordé la problématique de la co-évolution et de la migration. Enfin, nous avons abordé la migration inverse. Les quatre problématiques présentées ne sont pas traitées de manière simultanée par une solution unique. Les approches présentées proposent des solutions pour traiter une problématique, indépendamment des autres.

TABLEAU 2.5 – Comparaison des approches de migration

Approach / Tool	①	②	③	④	⑤
MCL [NLBK09]	✓	✓			
Sprinkle [SK04]	✓				
Epsilon Flock [RKPP10]	✓				
Cicchetti [CREP08]	✓				
Garcés [GJCB09]	✓	✓			
ATL	✓				
Cicchetti [CDRP09]	✓				
Ruegg [RMvH11]	✓	✓			
Edapt/COPE [HBJ09]	✓	✓			
EMFMigrate [WIRP12]	✓	✓			
Refactory	✓				
VIATRA [CHM+02]	✓	✓			
QVT [ACE+03]	✓	✓	✓		

Les différences entre les méta-modèles sont souvent difficiles à calculer parce qu'elles impliquent la déduction des transformations complexes appliquées aux méta-modèles. La même différence peut avoir des interprétations différentes. Par exemple, le déplacement d'un attribut d'une classe vers une autre classe, est de fois interprété comme la suppression de tel attribut et l'ajout d'un nouvel attribut. Il faudrait que le mécanisme de détection des différences soit capable d'identifier et de gérer les différences produites par des transformations complexes, ou qu'il permette à l'utilisateur de définir les différences entre deux méta-modèles.

La réduction de méta-modèles et modèles est faite au niveau méta-modèle et au niveau modèle, l'une indépendamment de l'autre. Celle-ci est une cause de non conformité entre méta-modèles et modèles. La réduction d'un méta-modèle via l'extraction d'un méta-modèle effectif, peut produire un méta-modèle avec des concepts qui ne sont plus nécessaires au niveau modèle. La réduction peut produire un méta-modèle réduit mais structuré différemment, par exemple, à cause de renommages non gérés. Il faudrait une solution qui intègre la réduction au deux niveaux (méta-modèle et modèle) et qui permette la gestion des transformations supplémentaires.

Parmi les approches qui garantissent l'évolution de méta-modèles et la co-évolution de modèles, nous choisissons d'utiliser l'approche par opérateurs de co-évolution, et en particulier Modif parce qu'il permet de définir des transformations complexes en utilisant peu d'opérateurs. Ceci facilite leur utilisation et l'obtention efficace de modèles adaptés. Les migrations inverses sont traitées comme des inversions simples des migrations. Nous constatons que ça ne suffit pas pour conserver les données qui font partie d'un domaine donné. La migration inverse devrait dépendre de la migration mais aussi du modèle transformé par l'outil réutilisé et du modèle initial.

Comme nous pouvons le voir dans les tableaux récapitulatifs, les approches étudiées ne permettent pas d'intégrer des solutions aux problématiques liées à la réutilisation. Nous proposons de tirer parti du savoir-faire existant pour proposer une solution pour faciliter la réutilisation d'outils existants dans des contextes différents. Le chapitre suivant présente notre proposition.

Deuxième partie

Une approche pour faciliter la réutilisation d'outils

Chapitre 3

Approche

Sommaire

3.1 Introduction du chapitre

Ce chapitre présente une approche pour faciliter la réutilisation d'un outil via les migrations de modèles asymétriques. À la demande d'un utilisateur, l'approche permet de réutiliser un outil existant dans un contexte autre que celui pour lequel il est défini. Le résultat de l'outil est lui-même adapté au contexte d'utilisation et peut être mis à jour par l'utilisateur.

L'approche consiste à adapter les modèles d'entrée et de sortie de l'outil au lieu de modifier l'outil lui-même. Les données conformes au méta-modèle d'application sont conjointement transformées en données conformes au méta-modèle de l'outil. Une fois traitées par l'outil, le modèle est remis dans son contexte d'origine grâce à une migration inverse spécifique qui permet de récupérer les éléments qui ont été supprimés préalablement.

Dans ce contexte, la réutilisation s'appuie sur des préoccupations de co-évolution (la transformations de méta-modèles et la migration de modèles) et la migration inverse de modèles. Assurer que les modèles sont bien dans le contexte de l'outil, que les actions menées par l'outil ne sont pas défaites et remettre la sortie de l'outil dans son contexte initial sont trois défis importants.

Nous commençons par présenter l'approche d'une façon générale, ses objectifs et son fonctionnement (section 3.2). Nous détaillons ensuite les différentes étapes de cette approche de réutilisation : la Migration (section 3.3), la Réutilisation de l'outil (section 3.4) et la Migration inverse (section 3.5). Enfin nous présentons une formalisation de l'approche pour la validation de certaines propriétés de conception (section 3.6).

3.2 Aperçu de l'approche

Cette approche a pour but de faciliter la réutilisation d'un outil via les migrations de modèles. Cette finalité est assurée par différents moteurs. Ils gèrent et adaptent les modèles, tout en offrant une solution syntaxiquement correcte et des utilitaires pour que l'utilisateur valide la sémantique. L'approche suit l'IDM et repose sur un ensemble de méta-modèles qui permettent de matérialiser les connaissances.

Avant de donner les détails concernant l'approche, nous considérons certaines hypothèses :

- Les données requises par l'outil doivent exister dans le contexte où cet outil va être réutilisé : les données requises par l'outil sont disponibles dans le domaine d'application (elles peuvent cependant être structurées différemment).
- Cette approche exclue toute considération sémantique liée au domaine de l'outil et au contexte de réutilisation. Notre objectif est de pousser au maximum l'automatisation de la réutilisation d'un outil en ne s'appuyant que sur des aspects syntaxiques et structurels. Et si nous pouvons prouver que les actions de l'outil sont conservées (*c.f.* section 3.6), nous ne pouvons pas garantir la validité sémantique du modèle recontextualisé. En contrepartie, notre approche fournit à l'expert du domaine des facilités de diagnostic en désignant précisément les éléments mis à jour par l'outil.
- N'importe quelle approche de co-évolution peut être choisie pour effectuer la migration, à condition qu'une instance migrée soit produite à partir d'une et

seulement une instance du modèle initial. De ce fait, l'approche de co-évolution doit respecter les exigences suivantes :

1. *Pas de création :* les classes, les attributs et les références peuvent être renommés. Les attributs et les références peuvent être déplacés. Mais la migration ne crée pas de données.
2. *Pas de fusion de classes :* le déplacement des attributs et des références doit être suffisant pour rassembler au sein d'une seule classe, des extraits d'autres classes.
3. *Pas de division de classes :* étant donné que la division des classes implique la création d'instances, ceci n'est pas pris en charge par l'approche.

Les migrations asymétriques constituent un *round-trip* de migration. Le round-trip permet de placer les données dans le contexte de l'outil, et une fois modifiées par l'outil, de les remettre dans le contexte du domaine d'application. Il est composé de trois étapes : *Migration*, *Réutilisation de l'outil* et *Migration inverse*. Ces étapes sont illustrées par la figure 3.1 et sont décrites comme suit :

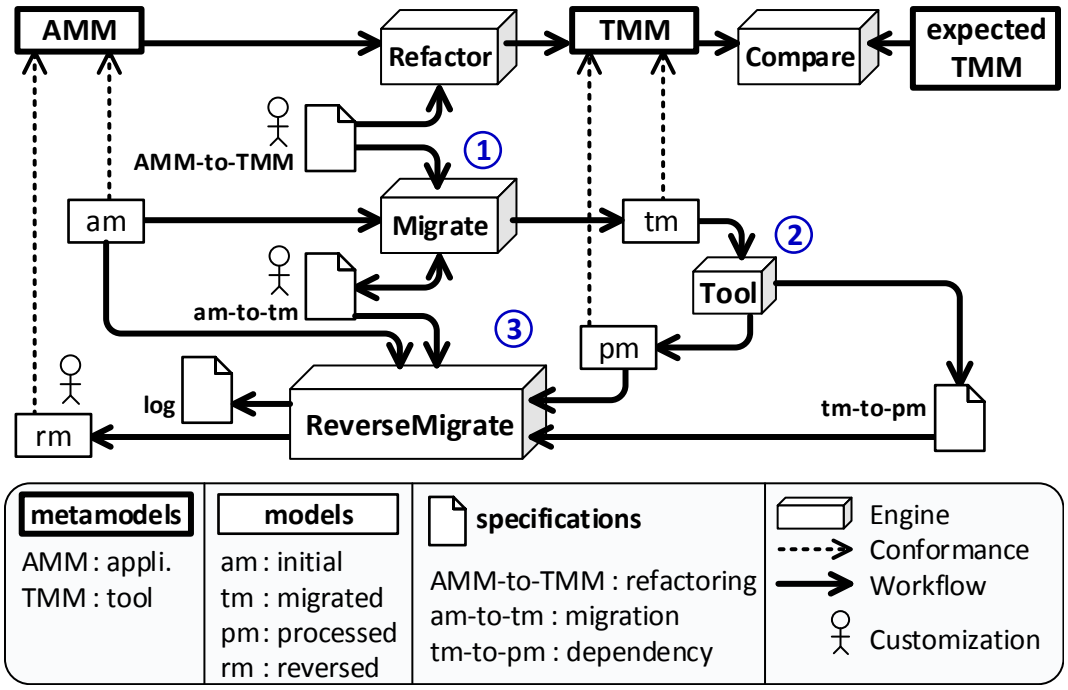


FIGURE 3.1 – Round-trip de migration pour réutiliser un outil

Migration. Place les données d'am (figure 3.1) dans le contexte de l'outil via le moteur de *migration* Migrate. Les données d'am sont conformes au méta-modèle de l'application AMM. Le résultat de la migration est conforme au méta-modèle de l'outil TMM. Dans un contexte de co-évolution, la migration s'appuie sur l'application d'opérateurs de refactoring (essentiellement des opérateurs de *réduction*). Le refactoring est géré par le moteur de *refactoring* Refactor.

Réutilisation de l'outil. Elle implique l'utilisation d'un *outil* Tool et la récupération d'un *graphe de dépendances* (tm-to-pm) pour expliciter le lien entre les instances de sortie et les instances d'entrée de l'outil.

Migration inverse. Adapte la sortie de l'outil pour la rendre conforme au méta-modèle d'application AMM (figure 3.1), et ré-injecte les données supprimées lors de la migration. Cette adaptation s'appuie sur la sortie de l'outil, le graphe de dépendances, le modèle initial et la spécification de migration.

L'étape de *Migration* repose sur les principes de l'évolution de méta-modèles et la co-évolution de modèles. L'étape de *Réutilisation de l'outil* consiste essentiellement à utiliser un outil existant. Cet outil doit être une boîte grise pour lequel les dépendances entre les entrées et les sorties peuvent être déterminées à partir d'une spécification de l'outil sans connaitre les détails des calculs effectués. La *Migration inverse* est l'étape la plus complexe et la plus originale. Cette étape est scindée en quatre parties : 1) Migration retour, 2) Recontextualisation par clés, 3) Recontextualisation par graphe de dépendances et 4) Gestion des contraintes.

Le principal objectif de cette approche est de rendre le processus de réutilisation plus facile en l'automatisant le plus possible et en guidant l'utilisateur. Ceci réduit le temps de développement dans un environnement orienté modèle. Des expériences qui mettent en évidence cette situation seront présentées dans le chapitre 5. Par la suite, les étapes du round-trip sont présentées avec plus de détails.

3.3 Migration

Avant de rentrer dans les détails de la migration, nous listons nos hypothèses. Notre première hypothèse est que nous considérons que le méta-modèle de l'outil est une variante du méta-modèle initial. Nous pouvons supposer qu'ils partagent des concepts sémantiquement proches. De ce fait, il est possible d'établir des liens entre deux méta-modèles. Il est donc possible de traduire les instances du méta-modèle initial en des instances du méta-modèle de l'outil.

Un outil fait des modifications sur un modèle pour produire en sortie, un modèle modifié. Un outil peut être du code qui exécute une fonctionnalité. Nous utilisons les termes *outil* et *code* de manière indifférenciée pour désigner un outil à réutiliser. Nous nous intéressons uniquement aux outils développés conformément aux bonnes pratiques de programmation, et dont les fonctions de calcul sont connues (spécification en fonction de l'entrée et la sortie). Notre deuxième hypothèse est que ces outils sont testés et validés. Les entrées et les sorties de l'outil sont conformes au même méta-modèle et ce méta-modèle est connu.

Le migrateur Migrate est un générateur de *modèles migrés* à partir de *spécifications de migration*. Il applique la suppression d'instances, d'attributs et de références dans le modèle source pour produire le modèle cible. Il peut aussi renommer les attributs et les références et établir des références dérivées à partir de références existantes. Cette étape implique une migration au niveau modèle, mais elle est la conséquence d'une préoccupation du niveau méta-modèle, notamment la nécessité de faire correspondre

le méta-modèle du domaine d'application (AMM) avec le méta-modèle de définition de l'outil (TMM).

Par la suite, les éléments importants qui mènent à la migration sont détaillés.

3.3.1 Réduction, Refactoring et Comparaison

Afin de réutiliser un outil existant, les données d'un domaine d'application doivent être placées dans le contexte de l'outil. Nous considérons que l'outil à réutiliser est un outil dédié qui ne se concentre que sur les données strictement nécessaires pour exécuter ses actions. Nous faisons l'hypothèse que toutes les données requises par l'outil sont disponibles dans le domaine d'application (section 3.2). En conséquence, l'extraction des informations pertinentes est nécessaire. La partie extraite peut être structurée de manière différente par rapport à ce qui est attendu par l'outil. Dans ce cas, il devient nécessaire d'appliquer des transformations de refactoring.

Les approches de co-évolution proposent des solutions pertinentes pour réduire et transformer les modèles au travers d'opérateurs définis au niveau méta-modèle. Parmi les approches de co-évolution que nous avons étudié (voir chapitre 2), nous choisissons Modif parce qu'il fournit des opérateurs de réduction et de refactoring qui permettent l'extraction et l'adaptation de manière efficace. Avec Modif, peu d'opérateurs (rename, flatten, hide et remove) suffisent pour faire une extraction. Modif ne fournit pas d'opérateurs pour l'addition d'éléments. Mais cela ne représente pas un problème pour nous du fait de notre hypothèse « les données nécessaires à l'outil sont présentes dans le modèle initial ». Une fois transformé, le méta-modèle obtenu doit être comparé au méta-modèle de l'outil afin de vérifier qu'ils coïncident. Cette vérification est faite pour garantir que les opérateurs appliqués sont les opérateurs appropriés pour placer les données du domaine d'application dans le domaine de l'outil.

Concrètement, dans notre approche, la réduction et le refactoring sont gérés par le moteur de *refactoring* Refactor. La comparaison entre le méta-modèle transformé et le méta-modèle de l'outil est gérée par le moteur de *comparaison* Compare.

Les figures 3.2 et 3.3 illustrent, respectivement, un méta-modèle d'un domaine d'application et un méta-modèle d'un outil. Le méta-modèle du domaine est composé de trois classes concrètes (A, B et C) et d'une classe abstraite (D) à partir de laquelle les classes concrètes héritent les attributs *nb* et *label*. La classe B a une référence simple (r_2) vers elle-même. Il y a deux références composites (r_1 et r_3).

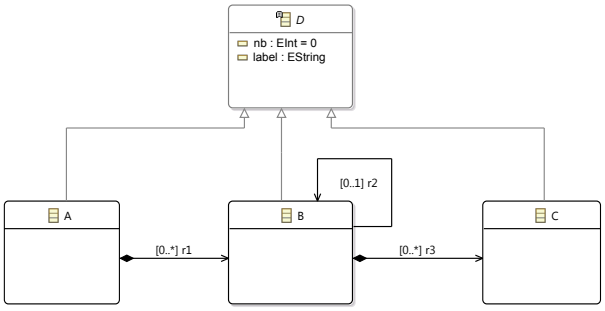


FIGURE 3.2 – Méta-modèle du domaine d'application

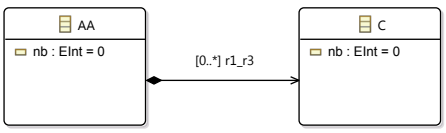


FIGURE 3.3 – Méta-modèle de l'outil

Le méta-modèle de l'outil est composé de deux classes concrètes (AA et C). Chacune des classes a un attribut *nb*. La référence composite $r_1\_r_3$ relie AA et C.

Le tableau 3.1 indique la façon dont les données sont représentées dans les deux méta-modèles. Dans le méta-modèle de l'outil, elles peuvent être modifiées ou déplacées, mais on trouve toujours son origine dans le méta-modèle du domaine. Par exemple, dans le méta-modèle du domaine *nb* est un attribut de la classe abstraite D, tandis que dans le méta-modèle de l'outil, *nb* fait partie des sous-classes de D.

TABLEAU 3.1 – Les données du méta-modèle de l'outil sont dans le méta-modèle du domaine

Domain metamodel	**Tool metamodel**
A	AA
nb (from D)	*nb* (AA)
	nb (C)
r1	*r1_r3*
r3	
B, D, *r2*, *label*	—

Le listing 3.1 illustre l'ensemble d'opérateurs de Modif nécessaires pour extraire les concepts du méta-modèle d'application et converger vers le méta-modèle de l'outil. Le listing 3.2 illustre l'ensemble simplifié des opérateurs du listing 3.1. Si l'opérateur remove est appliqué à un attribut ou une référence d'une super classe, il n'est pas nécessaire de l'appliquer sur les attributs ou les références des sous classes. La composition de la référence $r1\_r3$ est déduite de l'application de l'opérateur hide.

Listing 3.1 – Opérateurs de transformation Modif

```
A to AA { att nb
          remove att label
          ref r1 } ;
hide B { att nb
         remove att label
         remove ref r2
         ref r3 } ;
C { att nb
    att label } ;
flatten hide D { att nb
        remove att label }
```

Listing 3.2 – Opérateurs de transformation Modif (simplifié)

```
A to AA { att nb
          att label
          ref r1 } ;
hide B { att nb
         att label
         remove ref r2
         ref r3 } ;
C { att nb
    att label } ;
flatten hide D { att nb
            remove att label }
```

Une fois que la comparaison est validée, tous les modèles am conformes à AMM sont migrés vers des modèles conformes à TMM (figure 3.1). Il est à noter que si une instance de am peut être migrée vers une instance de tm, elle peut également être supprimée. Par contre, une instance de tm est toujours produite à partir d'une et seulement une instance de am. La section suivante présente une représentation des modèles sous forme de graphe qui facilite leur manipulation.

3.3.2 Le modèle : un graphe d'objets

Dans le contexte de l'orienté-objet, un modèle d'objets manipulés est considéré comme un simple graphe d'objets [**?**]. Cette vision des modèles est une caractéristique commune des frameworks qui gèrent la méta-modelisation orientée-objet.

Par exemple, le framework de méta-modélisation EMF, inclut implicitement cette notion de graphe d'objets. Un graphe est une structure de données largement utilisée parce que sa représentation est simple ; elle illustre la structure du modèle ; il est facile d'identifier les liens qui existent entre les éléments ; et la manipulation est facilitée. Cette structure est bien adaptée pour faire des parcours et des transformations de modèles basées sur un parcours de graphe.

Dans cette thèse un modèle est vu comme un graphe d'objets. Plus précisément, c'est un graphe orienté et étiqueté composé de *sommets* et d'*arcs*. Il comporte deux ensembles de sommets : des *instances* et des *valeurs scalaires*. Il comporte également deux ensembles d'arcs : des *références* et des *attributs*. Une référence relie une instance avec elle-même ou avec une autre instance. Un attribut relie une instance avec une valeur scalaire. Les deux types d'arcs sont étiquetés avec des *noms*.

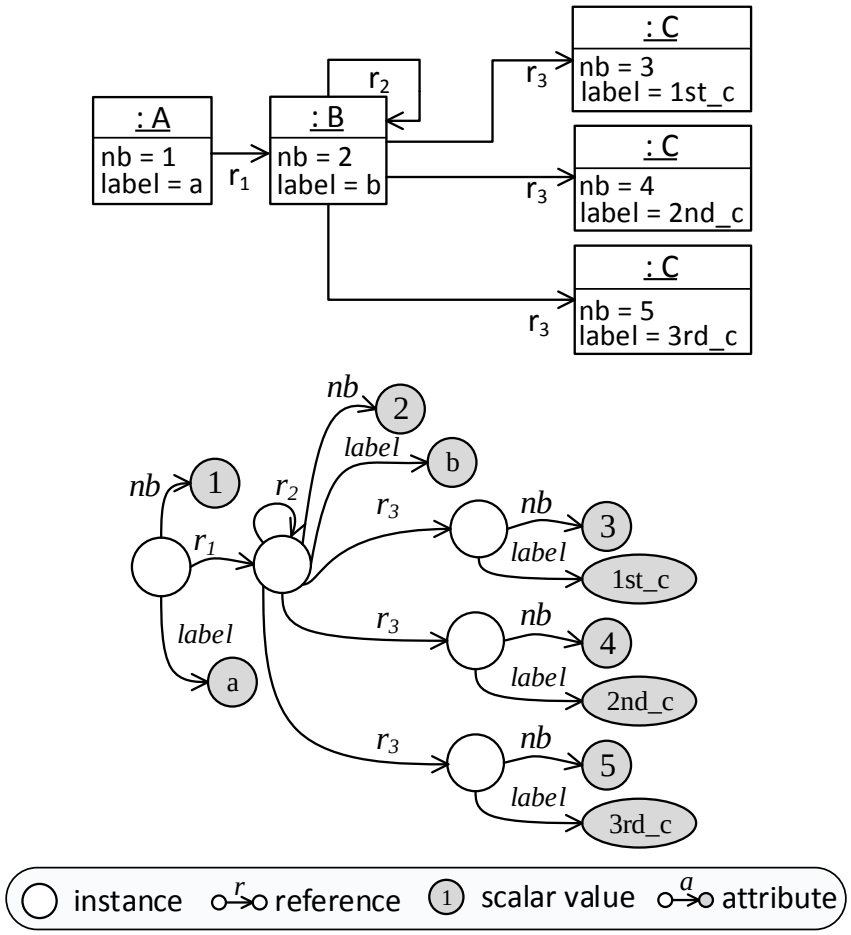


FIGURE 3.4 – Modèle et graphe d'objets

La figure 3.4 illustre un graphe d'objets. Les instances sont représentées par des cercles blancs, les valeurs scalaires sont représentées par des ellipses grises. Les arcs qui relient des instances correspondent aux références. Les arcs qui relient une instance et une valeur scalaire correspondent aux attributs. Concrètement ce graphe d'objets correspond à un modèle conforme au méta-modèle de la figure 3.2. Dans ce graphe d'objets il y a 5 instances. Chacune possède un attribut *nb* : 1, 2, 3, 4 et 5. Chacune des instances possède un attribut *label* : a, b, $1st\_c$, $2nd\_c$ et $3rd\_c$. r_1 relie deux instances ; une instance est reliée elle-même via r_2, et avec trois autres instances par

l'intermédiaire des trois arcs r_3. La notation présentée est utilisée dans la suite de ce document.

Dans le graphe d'objet que nous proposons, nous faisons abstraction de certaines caractéristiques propres des modèles Ecore (méta-méta-modèle référence de la thèse). Cette abstraction rend le modèle de graphes entièrement indépendant du méta-modèle. Ceci facilite sa manipulation et nous permettra de définir des migrations génériques et indépendantes de tout méta-modèle. Les caractéristiques d'Ecore qui restent hors de la portée du graphe d'objets sont :

- *Multiplicité* : les contraintes de multiplicité associées aux attributs et aux références ne sont pas prises en compte. Nous considérons qu'il n'y a pas de restrictions dans la quantité d'instances et de valeurs scalaires auxquels une instance est liée.
- *Unicité* : les arcs du modèle d'instances sont uniques. Il n'y a pas deux arcs identiques (deux arcs avec la même étiquette qui relient la même paire d'instances).
- *Références opposées* : nous ne prenons pas en charge la relation entre une référence et sa référence opposée. Une référence et sa référence opposée sont représentées dans le modèle de graphe comme deux références indépendantes l'une de l'autre.
- *Références composites* : les références composites (relation de contenance) sont traitées comme des références simples.

Nous ne prenons pas ses caractéristiques en charge dans notre approche car elles ne sont pas indispensables pour représenter les objets et les liaisons entre eux. De plus, leur abstraction facilite la migration des graphes d'objets. Nous allons ensuite présenter le mécanisme qui permet de différencier chaque instance afin de l'identifier lors de diverses transformations.

3.3.3 Mécanisme d'identification des instances

Les instances doivent être *identifiables* afin de les différentier dans le graphe d'objets et afin de tracer leur évolution (suppression, mise à jour ou création) lors des diverses migrations (migration, réutilisation de l'outil, migration inverse). Concrètement, dans cette thèse chacune des instances du graphe d'objets est identifiée par un *identifiant unique* (*Universally Unique IDentifier UUID*). Les attributs et les références n'ont pas d'identifiants uniques parce que nous considérons que leurs évolutions peuvent être tracées à partir de leur instance source. Néanmoins ils sont étiquetés avec le nom de l'attribut ou la référence qu'ils représentent. Les valeurs scalaires n'ont pas d'identifiants parce qu'elles sont liées aux attributs qui sont étiquetés et qu'elles sont uniques. Si il y a plusieurs attributs avec la même valeur, seulement un sommet avec cette valeur apparaît dans le graphe.

Afin de présenter le principe d'identification des instances, dans ce chapitre les identifiants sont constitués d'un littéral et d'un chiffre. Pour garantir l'unicité, un registre central contient les identifiants utilisés. À titre d'illustration, la figure 3.5 présente un modèle d'objets avec identifiants (a_1, b_1, c_1, c_2 et c_3).

Si un identifiant est unique dans un graphe d'objets, on peut trouver le même identifiant dans plusieurs graphes d'objets. Les identifiants permettent alors de déterminer si deux instances de deux graphes d'objets correspondent entre elles : si deux

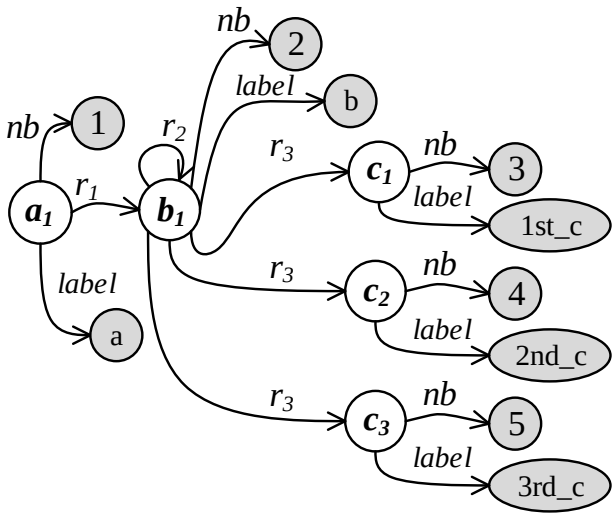


FIGURE 3.5 – Graphe d'objets avec identifiants

instances partagent le même identifiant, elles sont considérées comme deux versions de la même instance ; sinon elles sont considérées comme deux instances différentes.

Lors d'une migration, les identifiants des instances sont conservés, sauf si la migration implique une suppression. Dans le cas d'une suppression d'instance, l'instance, ses attributs et ses références sont supprimés. La suppression d'un attribut, d'une référence ou d'une valeur scalaire ne modifie pas les identifiants des instances du graphe transformé. Si une transformation implique la création d'une instance, un nouvel identifiant est généré et associé à la nouvelle instance. La création d'un attribut, d'une référence ou d'une valeur scalaire n'a pas d'impact sur les identifiants des instances du graphe.

En conclusion, les identifiants sont uniques par graphe d'objets. Ils permettent de repérer les instances dans un graphe d'objets et ils sont préservés lors d'une migration du graphe d'objets. Nous présentons maintenant la façon dont les graphes d'objets sont transformés et l'utilité des identifiants dans ce contexte.

3.3.4 Spécification de migration

En général, les approches de co-évolution réalisent une transformation conjointe d'un méta-modèle et des modèles conformes. Pour effectuer les migrations de modèles elles proposent de générer des codes de migration qui sont automatiquement exécutés sur les modèles.

Les codes de migration gèrent la logique de la migration en même temps que les fonctions d'accès et de contrôle pour manipuler les modèles. Cela rend les codes de migration complexes. Ceci est la cause de grandes quantités de code qui se répètent et qui ont peu de différences entre eux. De plus, les codes de migration sont générés à partir d'un langage de transformation de méta-modèles. Ils ne sont pas réutilisables pour migrer des modèles dont les méta-modèles auxquels ils sont conformes sont transformés avec d'autres langages. Ceci implique que les codes de migration dépendent du langage de transformation. Ces observations nous conduisent à identifier l'absence d'un mécanisme qui sépare les préoccupations liées au langage de transformation de

méta-modèles des préoccupations de migration de modèles. De plus, on souhaite lors de la migration séparer les préoccupations d'accès et de contrôle de données de la logique de migration.

Pour répondre à ces questions, nous proposons un moteur de migrations générique : Migrate (figure 3.1). Ce moteur contient le code d'accès et de contrôle nécessaires pour manipuler des graphes d'objets. L'effort pour développer ce code a été fait une fois et il n'est pas nécessaire de le générer à chaque fois qu'une transformation est appliquée.

Contrairement aux approches traditionnelles, au lieu de générer le code de migration, nous générons donc une *spécification de migration* (représentée par am-to-tm dans la figure 3.1) à partir du modèle source à migrer. Elle indique les modifications qui doivent être appliquées sur les éléments du modèle. La spécification de migration et le modèle source sont utilisés conjointement par Migrate pour produire le *modèle migré*. Dans cette thèse, une spécification de migration est produite à partir d'une spécification de refactoring provenant d'un langage de transformation donné. Mais elle pourrait être générée à partir d'autres langages de transformation.

Le méta-modèle représenté par la figure 3.6 organise les concepts définis dans la spécification de migration que nous détaillons maintenant. La base de ce méta-modèle est une migration (*Migration*) représentant un modèle composé d'instances (*Instance*). *Migration* référence les modèles source (*inputModel*) et migré (*outputModel*). Une instance est identifiée par son identifiant unique *UUID*. Lors de la migration une instance peut être supprimée ou migrée.

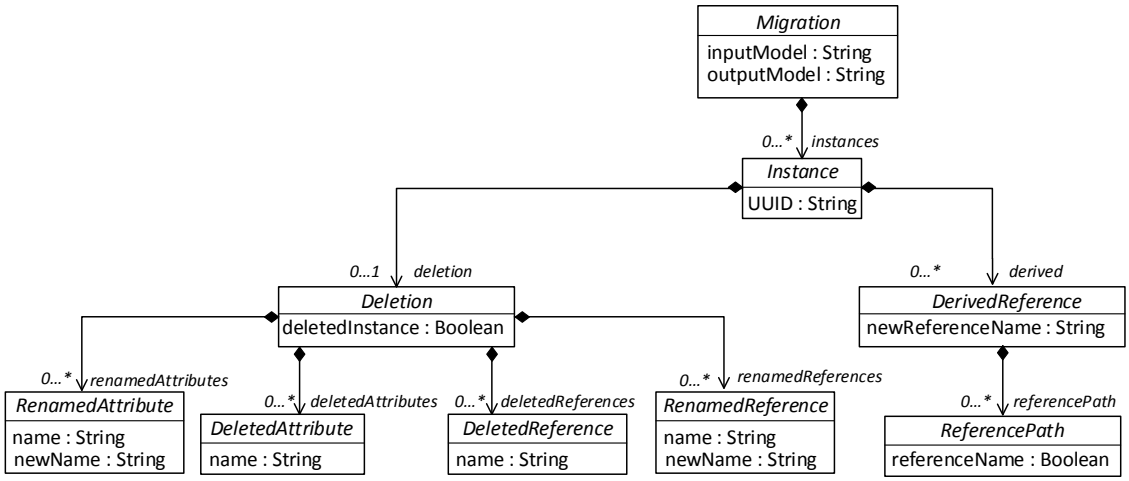


FIGURE 3.6 – Méta-modèle pour la spécification de migration

Pour indiquer la suppression d'une instance, l'attribut *deletedInstance* est mis à *true* pour l'instance concernée, dans ce cas l'instance est supprimée ainsi que leur attributs et leur références. Pour indiquer la suppression d'un attribut ou d'une référence sans supprimer l'instance qui le contient, *DeletionAttribute* ou *DeletionReference* sont utilisés en indiquant le nom de l'attribut ou de la référence à supprimer.

Le renommage d'un attribut ou d'une référence peut être indiqué grâce à *RenamedAttribute* et à *RenamedReference*. *name* indique le nom de l'attribut ou de la référence. *newName* indique le nouveau nom.

DerivedReference ajoute une référence dérivée à partir d'un chemin de références (*ReferencePath*). Il existe un *ReferencePath* entre deux instances i_1 et *i2*, si i_1 et i_2 sont connectées par un ensemble de références (à partir de i_1, il est possible de référencer directement ou indirectement i_2). Un *ReferencePath* est constitué d'un ensemble de références et d'instances intermédiaires connectées.

Une spécification de migration est une séquence de déclarations de mise à jour du graphe d'objets. Dans un graphe d'objets la suppression d'une référence implique la suppression de l'arc qui représente la référence. La suppression d'un attribut implique la suppression de l'arc qui représente l'attribut, mais aussi de la valeur de l'attribut. La suppression d'une instance implique la suppression de leur références, leur attributs, les références qui vont vers l'instance et enfin, suppression de l'instance elle-même.

Le listing 3.3 présente un exemple d'une spécification de migration. Le *inputModel* est le graphe d'objets de la figure 3.5. La première ligne indique la suppression de l'attribut $label$ et de la référence r_1 liés à l'instance a_1. La deuxième ligne indique l'ajout d'une référence dérivée $r_1\_r_3$ entre l'instance a_1 et chaque instance référencée par r_3. La troisième ligne indique la suppression de l'instance b_1 (leur attributs et leur références sont également supprimés). Les dernières lignes indiquent la suppression de l'attribut $label$ associé aux instances c_1, c_2 et c_3 respectivement.

Listing 3.3 – Exemple de spécification de migration

```
a1 { deletedAttribute label ,
     deletedReference r1,
     derivedReference { r1_r3,
                        referencePath r1,
                        referencePath r3 } } ;
deletedInstance b1 { };
c1 { deletedAttribute label } ;
c2 { deletedAttribute label } ;
c3 { deletedAttribute label } ;
```

La figure 3.7 illustre le résultat après application de la spécification de migration. Un ensemble des éléments du modèle initial est mis de côté. Ces éléments sont représentés par DeletedInstance, DeletedAttribute et DeletedReference dans la spécification de migration (Un registre central contient les éléments supprimés). Les autres éléments sont migrés (Migrated model). La spécification de migration est la traduction d'une spécification de refactoring en opérations à appliquer dans un graphe d'objets.

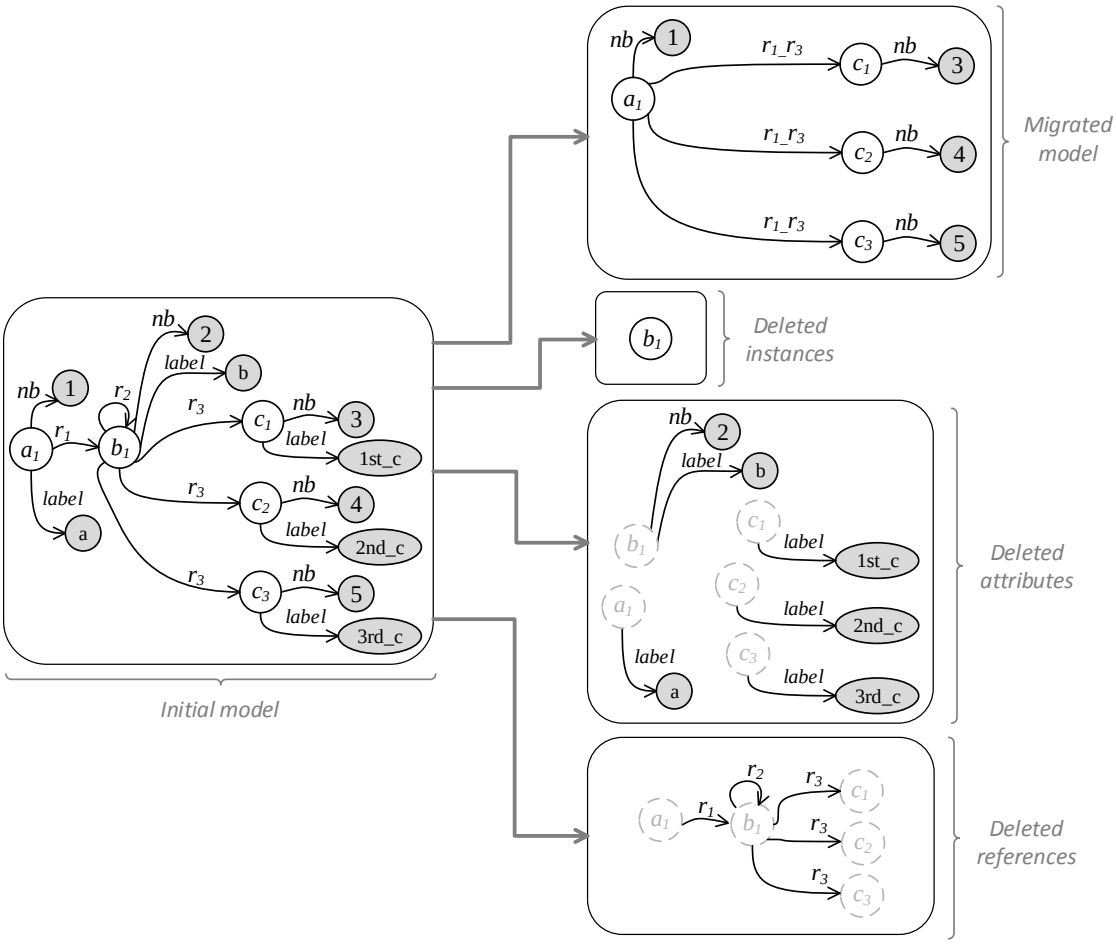


FIGURE 3.7 – Migration d'un modèle

3.3.5 Génération de la spécification de migration

Maintenant, nous présentons de manière plus systématique la liaison entre les opérateurs de Modif et la génération de la spécification de migration. Les tableaux 3.2, 3.3, 3.4 et 3.5 présentent la traduction des opérateurs rename, remove, hide et flatten, respectivement.

Dans les tableaux 3.6, 3.7 et 3.8, les opérateurs Modif (*Modif operators*) sont appliqués au méta-modèle de la figure 3.2 et produisent un méta-modèle cible (*target metamodel*). Ces opérateurs sont traduits en une spécification de migration (*migration specification*) qui est appliquée au modèle source de la figure 3.4 pour produire un modèle cible (*target model*).

Le tableau 3.6 présente la traduction de l'opérateur rename. Dans le premier cas, cet opérateur est appliqué à une classe, A devient AA. Le renommage d'une classe n'affecte pas le modèle d'objets, cela se traduit par une spécification de migration *vide* (aucune opération est appliquée au graphe d'objets). Son application produit un modèle cible identique au modèle source.

Dans le deuxième cas, le renommage est appliqué à un attribut, *nb* devient *val*. Dans la spécification de migration, le nouveau nom est indiqué (renamedAttribute) pour chaque instance qui contient cet attribut. Le modèle migré est un modèle dans lequel nb est remplacé par val.

TABLEAU 3.2 – Génération de la spécification de migration pour l'opérateur rename

Operator	Parameter	Migration specification
rename	*class* *attribute* *reference*	If *class* Then If *class* is concret Then ForAll *instance* of class *instance*.UUID Else If *class* is abstract Then ForAll *subclass* of class ForAll *instance* of subclass *instance*.UUID Else If *attribute* Then If *containing* class is concret Then ForAll *instance* of containing class attribute.name renamedAttribute attribuite.newName Else If *containing* class is abstract Then ForAll *subclass* of containing class ForAll *instance* of subclass attribute.name renamedAttribute attributte.newName Else If *reference* Then If *containing* class is concret Then ForAll *instance* of containing ForAll *instance* of class reference.name renamedReference reference.newName Else If *containing* class is abstract Then ForAll *subclass* of containing ForAll *instance* of subclass reference.name renamedReference reference.newName

TABLEAU 3.3 – Génération de la spécification de migration pour l'opérateur remove

Operator	Parameter	Migration specification
remove	*class* *attribute* *reference*	If *class* Then If *class* is concret Then ForAll *instance* of class deletedInstance *instance*.UUID Else If *class* is abstract Then ForAll *subclass* of class ForAll *instance* of subclass deletedInstance *instance*.UUID Else If *attribute* Then If *containing* class is concret Then ForAll *instance* of containing class deletedAttribute *attribute*.name Else If *containing* class is abstract Then ForAll *subclass* of containing class ForAll *instance* of subclass deletedAttribute *attribute*.name Else If *reference* Then If *containing* class is concret Then ForAll *instance* of containing class deletedReference *reference*.name Else If *containing* class is abstract Then ForAll *subclass* of containing class ForAll *instance* of subclass deletedReference *reference*.name

TABLEAU 3.4 – Génération de la spécification de migration pour l'opérateur hide

Operator	Parameter	Migration specification
hide	*class*	ForAll *otherClass* ForAll *otherReference* of otherClass If *otherReference*.to class Then ForAll *reference* of class ForAll *instance* of otherClass *instance*.UUID derivedReference otherReference.name _reference.name referencePath otherReference.name referencePath reference.name If *class* is concret Then ForAll *instance* of class deteledInstance *instance*.name Else If *class* is abstract Then ForAll *subclass* of class ForAll *instance* of subclass deteledInstance *instance*.name

TABLEAU 3.5 – Génération de la spécification de migration pour l'opérateur flatten

Operator	Parameter	Migration specification
flatten	*class*	If *class* is concret Then ForAll *instance* of class *instance*.UUID Else If *class* is abstract Then ForAll *subclass* of class ForAll *instance* of subclass *instance*.UUID

En dernier, nous appliquons le renommage à une référence, r_1 devient r_4. Dans la spécification de migration, le nouveau nom est indiqué (avec renamedReference) pour toutes les références de la classe A dont le nom est r_1. Le modèle résultat est un graphe dans lequel la référence r_1 est renommée.

Le tableau 3.7 présente la traduction de l'opérateur remove. L'application de l'opérateur remove sur une classe (remove B) est traduit par deleteInstance sur toutes les instances du modèle conformes à la classe à supprimer. La suppression d'une instance dans le graphe d'objets implique la suppression de l'instance, leurs attributs et les références qui pointent vers l'instance. La migration supprime l'instance b_1, supprime toutes les références dont la source est l'instance b_1 (r_2 et toutes les références r_3), supprime tous les attributs de l'instance b_1 (nb et $label$) et supprime toutes les références dont la cible est l'instance b_1 (r_1). Le modèle migré ne contient pas l'instance b_1 ni les arcs qui étaient liés à elle.

Ensuite nous présentons l'application de l'opérateur remove sur un attribut, remove nb se traduit dans la spécification de migration par deleteAttribute sur l'attribut à supprimer. Le modèle migré ne contient pas d'attribut nb ni des valeurs scalaires associées à cet attribut.

La suppression d'une référence (remove r_2) se traduit dans la spécification de migration par deletedReference sur la référence concernée. La migration supprime la référence r_2.

Le tableau 3.8 présente la traduction des opérateurs hide et flatten. L'opérateur hide appliqué sur une classe se traduit par la suppression des instances conformes à la classe et par l'ajout d'une référence dérivée ($r1\_r3$) entre chaque instance ayant une référence vers $b1$ et chaque référence de $b1$. Dans le cas général, pour chaque instance de la classe, n références vers l'instance et m références à partir de l'instance, produisent $n \times m$ références dérivées.

L'opérateur flatten appliqué à une classe génère une spécification de migration *vide*. Cet opérateur modifie la classe du méta-modèle, mais le graphe objets n'est pas impacté par les modifications. Le modèle migré est donc identique au modèle source.

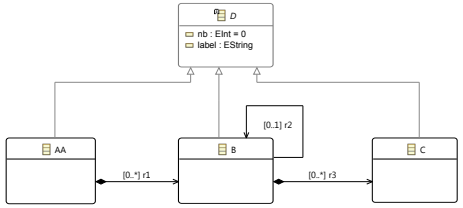
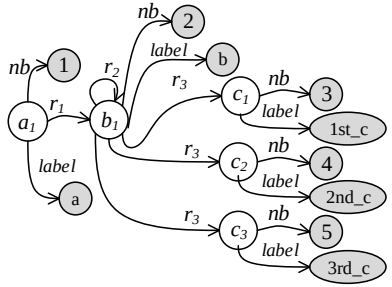
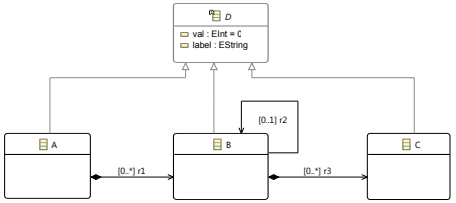
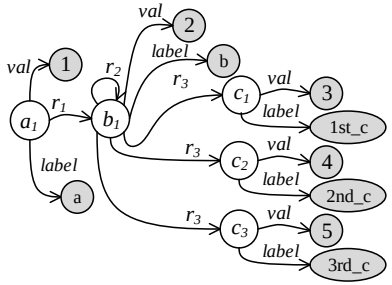
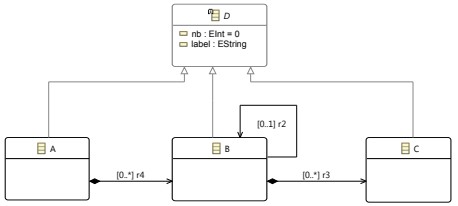
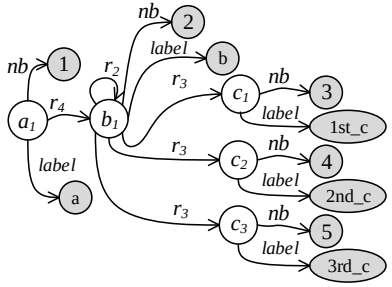
Modif operators	target metamodel	migration specification	target model
A to AA {} ;		$a1$ { } ; $b1$ { } ; $c1$ { } ; $c2$ { } ; $c3$ { } ;	
D { nb to val } ;		$a1$ { nb renamedAttribute val } ; $b1$ { nb renamedAttribute val } ; $c1$ { nb renamedAttribute val } ; $c2$ { nb renamedAttribute val } ; $c3$ { nb renamedAttribute val } ;	
A { $r1$ to $r4$ } ;		$a1$ { $r1$ renamedReference $r4$ } ; $b1$ { } ; $c1$ { } ; $c2$ { } ; $c3$ { } ;	

TABLEAU 3.6 – Traduction de l'opérateur rename

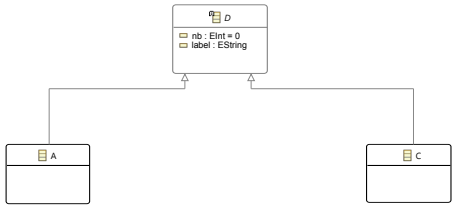
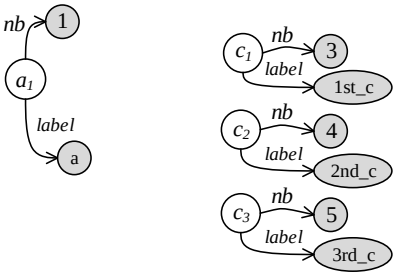
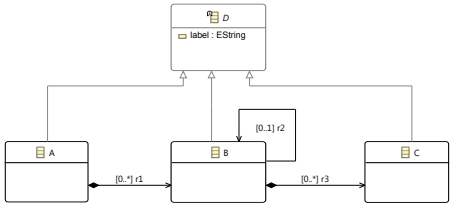
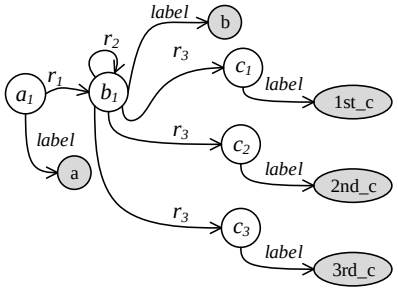
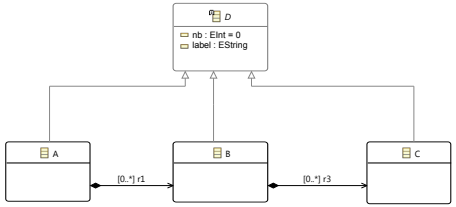
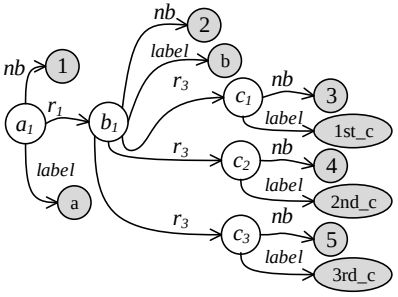
Modif operators	target metamodel	migration specification	target model
remove B { } ;		$a1$ { } ; deleteInstance $b1$ { } ; $c1$ { } ; $c2$ { } ; $c3$ { } ;	
D { remove nb } ;		$a1$ { deletedAttribute nb } ; $b1$ { deletedAttribute nb } ; $c1$ { deletedAttribute nb } ; $c2$ { deletedAttribute nb } ; $c3$ { deletedAttribute nb } ;	
B { remove $r2$ } ;		$a1$ { } ; $b1$ { deletedReference $r2$ } ; $c1$ { } ; $c2$ { } ; $c3$ { } ;	

TABLEAU 3.7 – Traduction de l'opérateur remove

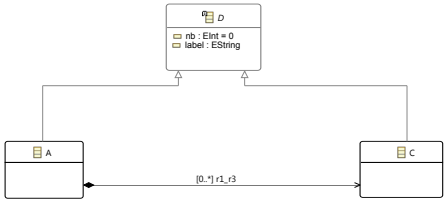
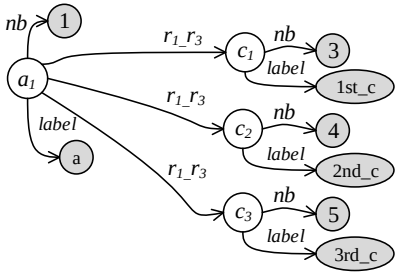
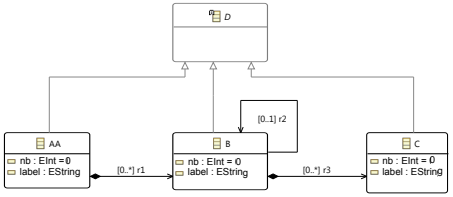
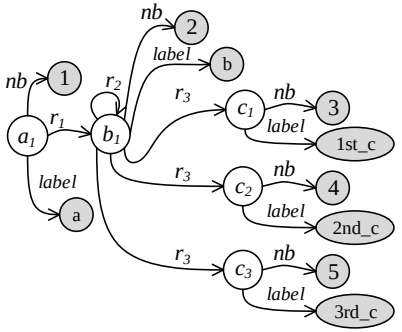
Modif operators	target metamodel	migration specification	target model
hide B { } ;		$a1$ { derivedReference { $r1\_r3$, referencePath $r1$, referencePath $r3$ } } ; deleteInstance $b1$ { } ; $c1$ { } ; $c2$ { } ; $c3$ { } ;	
flatten D { } ;		$a1$ { } ; $b1$ { } ; $c1$ { } ; $c2$ { } ; $c3$ { } ;	

TABLEAU 3.8 – Traduction des opérateurs hide et flatten

À la fin de la première étape du round-trip de migration, le modèle initial am est transformé selon la spécification de migration vers le modèle migré tm, lequel est *par construction* conforme à TMM. Cette transformation est faite par le moteur générique Migrate.

Pour conclure, nous listons les avantages d'utiliser un moteur de migrations générique et une spécification de migration :

- Si le langage de transformation change, la logique de la migration n'est pas affectée.
- Le code de contrôle qui permet de interpréter la spécification de migration et de produire le modèle migré est toujours le même.
- La migration est applicable à n'importe quel modèle, car elle est indépendante du méta-modèle.
- La spécification de migration est éditable, permettant de l'adapter si nécessaire.

3.3.6 Personnalisation de la spécification de migration

Dans le contexte de la réutilisation, le même outil peut être utilisé en fonction de besoins différents et spécifiques. Ceci implique que la migration doit permettre la génération de différents modèles pour répondre à des besoins spécifiques. Cependant, dans le contexte de l'évolution de méta-modèles et la co-évolution de modèles, les transformations du méta-modèle sont reflétées automatiquement sur les modèles. Par exemple, si une classe est supprimée au niveau méta-modèle, toutes les instances de la classe sont automatiquement supprimées. À l'inverse, si une classe est conservée, alors toutes ses instances sont conservées.

La nature automatique de la co-évolution empêche que les spécificités du niveau modèle soient spécifiées. Pour répondre à ce besoin, il existe des outils tels qu'Edapt [1] qui permettent de spécifier manuellement les migrations de modèle quand elles sont spécifiques au contexte du modèle. Edapt génère du code de migration pour les instances affectées par les transformations du méta-modèle. Ce code peut être modifié seulement si la classe est impactée, sinon, le code de migration n'est pas généré et il n'est pas possible de spécifier la migration des instances.

Une autre façon de faire des migrations spécifiques est d'utiliser des langages de transformation classiques, comme ATL, ces langages permettent de définir du code spécifique pour effectuer les migrations. Cette solution implique la définition d'un code différent pour chaque migration spécifique. Cette solution n'est pas générique et est orientée au langage de transformation.

Dans notre approche, nous proposons d'éditer la spécification de migration initialement générée automatiquement. L'édition permet de *personnaliser* la migration en définissant des objectifs précis qui dépendent de préoccupations spécifiques. La même spécification de migration peut être éditée différemment pour produire des modèles migrés différents.

Concernant les personnalisations qui peuvent être faites, nous considérons les suivantes :

- Une classe est conservée, mais certaines de ses instances doivent être supprimées.
- Une classe est conservée et toutes ses instances doivent être supprimées.

1. http://www.eclipse.org/edapt/

Dans les deux cas, il y a deux préoccupations, la première liée aux classes et la seconde liée aux instances du modèle. Avec la spécification de migration éditable, nous assurons la séparation de ces préoccupations. La spécification de migration facilite la production de différents modèles migrés : (1) sans développer ou adapter le code de migration ; (2) sans généraliser la transformation au niveau méta-modèle et (3) en définissant des modifications sur des instances spécifiques.

Afin d'illustrer l'utilité des spécifications de migration éditables, nous reprenons le modèle de la figure 3.5, la spécification de migration du listing 3.3 et le modèle migré de la figure 3.7. Maintenant, le besoin est de migrer explicitement les instances a_1, c_1 et c_2 et de supprimer les autres. La spécification de migration générée automatiquement est éditée, afin d'indiquer explicitement que l'instance c_3 ne doit pas être migrée (deletedInstance c_3 dans la spécification du listing 3.4). La même action pourrait être appliquée à c_1 ou à c_2. Le modèle migré est illustré par la figure 3.8.

Listing 3.4 – Exemple de spécification de migration personnalisée

```
a1 { deletedAttribute label , deletedReference r1,
     derivedReference { r1_r3, referencePath r1, referencePath r3 } } ;
deletedInstance b1 { };
c1 { deletedAttribute label } ;
c2 { deletedAttribute label } ;
deletedInstance c3 { } ;
```

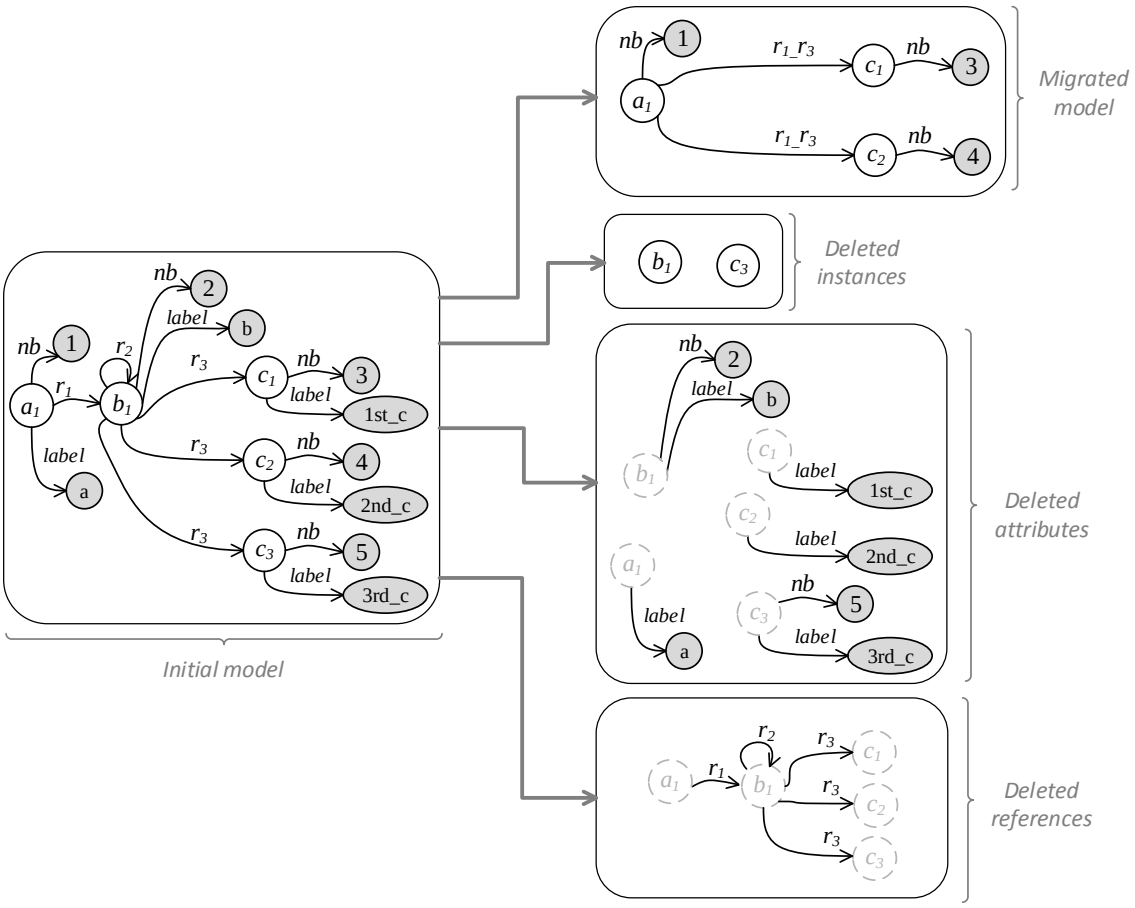


FIGURE 3.8 – Migration personnalisée

À chaque fois que la spécification de migration est éditée, une validation est faite avant de réaliser la migration. Cette validation permet de vérifier que le graphe d'objets respecte les contraintes de conformité imposées par le méta-modèle de l'outil.

La spécification de migration permet de gérer des spécificités qui ne sont pas faciles à aborder si la migration est faite automatiquement. La spécification de refactoring n'est pas modifiée. Nous constatons que les préoccupations spécifiques du niveau modèle sont séparées des préoccupations du niveau méta-modèle. Cette approche combine la génération automatique de la spécification de migration avec la personnalisation.

3.4 Réutilisation de l'outil

Nous nous intéressons à la réutilisation d'outils dédiés, et plus précisément d'outils de réécriture. Dans cette thèse, nous utilisons le terme *outil* pour désigner une *transformation* de modèles. Un outil à partir d'un modèle d'entrée tm produit un modèle de sortie modifié pm.

Nous identifions trois cas représentatifs de l'action de l'outil sur le modèle d'entrée :

1. L'action de l'outil est neutre sur le modèle. Il n'applique aucune modification et fournit en résultat le modèle d'entrée inchangé. Les outils d'analyse sont typiquement de cette catégorie.
2. L'action consiste à créer entièrement le modèle résultant en ne conservant rien du modèle d'entrée. Les traducteurs sont typiquement de cette catégorie.
3. L'action se situe entre les extrêmes représentés par les items 1 et 2. L'outil fournit en résultat une partie non modifiée du modèle d'entrée et une partie spécifiquement créée ou modifiée.

Nous détaillons ensuite toutes les modifications possibles sur le modèle d'entrée : la suppression de données, la mise à jour de données et l'ajout de *nouvelles données*.

La suppression d'un élément implique qu'il n'est plus présent dans la modèle de sortie de l'outil. Instances, attributs et références peuvent être supprimés.

Lorsque il s'agit de la mise à jour ou de l'ajout, nous faisons l'hypothèse qu'un élément est crée ou modifié à partir du même type d'éléments. Les instances sont créées ou modifiées à partir des informations d'autres instances ; les valeurs scalaires sont modifiées ou créées à partir d'autres valeurs scalaires de la même instance ; les références sont modifiées ou crées à partir d'autres références ; et les attributs sont crées à partir des informations d'autres attributs. La mise à jour d'un élément du modèle d'entrée implique que cet élément est présent dans le modèle de sortie, mais qu'il a subi une modification. Il est à noter que la modification d'une instance est faite via les modifications de leurs attributs et leurs références. La modification d'un attribut est faite via la modification de sa valeur scalaire, la source de l'attribut peut être également modifiée.

L'ajout d'un élément implique que le modèle de sortie a des nouvelles données (instances, attributs, références) par rapport au modèle d'entrée de l'outil. L'ajout d'une instance implique la création d'un identifiant pour cette instance.

Dans tous les cas, il est important de tracer l'évolution des éléments du modèle d'entrée. Conserver la trace implique que nous pouvons connaitre comment ont évolué les données d'entrée afin de produire le modèle de sortie. Nous organisons l'évolution des instances autour de l'évolution des attributs et des références.

Afin de tracer l'évolution des instances, nous considérons que l'outil à réutiliser est un outil Java boîte grise, pour lequel les détails d'*implémentation* ne sont pas disponibles. Mais les liens entre chaque instance de la sortie de l'outil et les instances de l'entrée de l'outil qui ont permis de le calculer peuvent être calculés. Ces liens sont utilisés pour spécifier un *graphe de dépendances* représenté par tm-to-pm dans la figure 3.1. Ce graphe est vu comme une spécification du comportement de l'outil. Concrètement, il détermine l'ensemble d'instances de l'entrée de l'outil qui ont été utilisées pour mettre à jour ou créer une instance de la sortie de l'outil.

Les concepts du graphe de dépendances que nous proposons sont illustrés par le méta-modèle de la figure 3.9. La base de ce méta-modèle est un graphe *Dependency*. Il est composé d'instances *Instance* (d'entrée et de sortie de l'outil). Une instance est identifiée avec l'identifiant unique *UUID*. Si une instance est créée par l'outil, un identifiant est également crée. La référence *dependsOn* représente la dépendance entre une instance de sortie et d'autres instances du modèle d'entrée de l'outil. Dès qu'une instance est modifiée, on ajoute un lien *dependsOn* entre chaque instance qui a participé à la modification et l'instance modifiée.

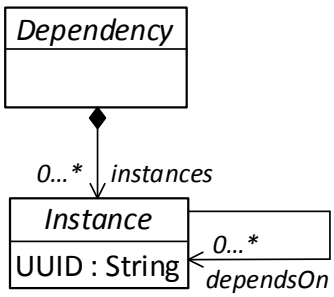


FIGURE 3.9 – Méta-modèle pour le graphe de dépendances

La figure 3.10 illustre un exemple typique des informations contenues dans un graphe de dépendances.

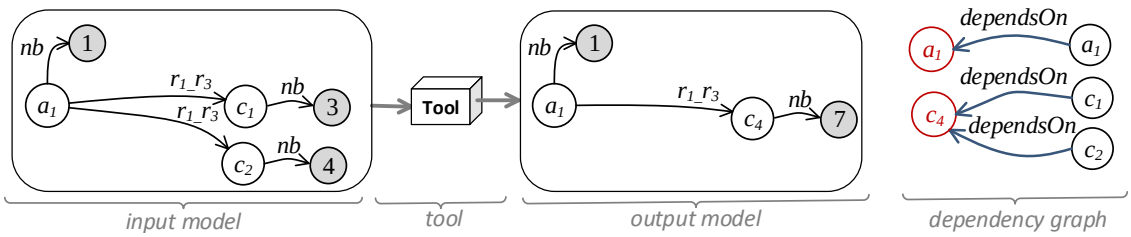


FIGURE 3.10 – Graphe de dépendances

En premier, un modèle d'entrée constitué des instances a_1, c_1 et c_2 est présenté. Ensuite, un outil Tool traite le modèle d'entrée et produit le modèle de sortie. Ce dernier, est constitué de deux instances a_1 et c_4, les instances c_1 et c_2 ont été supprimées par l'outil et la nouvelle instance c_4 a été créée (les détails de calcul ne sont pas connus).

Enfin, le graphe de dépendances rend explicite que l'instance de sortie a_1 a été produite à partir de l'instance d'entrée a_1 et que l'instance c_4 de la sortie a été produite à partir des informations des instances c_1 et c_2 de l'entrée de l'outil. Dans ce graphe, la source d'un lien *dependsOn* est toujours une instance d'entrée de l'outil et la cible du lien est une instance de la sortie de l'outil.

Une fois que l'outil est appliqué, le graphe de dépendances (dans lequel les instances sont identifiées) permet d'établir une liaison entre les instances du modèle migré et celles du modèle modifié par l'outil. Lorsque une nouvelle instance est créée, *a priori* celle-ci n'a pas de lien avec les instances qui existaient dans le modèle d'entrée de l'outil. Le graphe de dépendances établit une liaison entre cette nouvelle instance et les éléments du modèle qui ont participé à sa création.

Le graphe de dépendances est utile lors de la migration inverse, qui est présentée dans la section suivante.

3.5 Migration inverse

Jusqu'ici, l'utilisation de la co-évolution reste classique. Dans l'approche proposée, ce qui est plus original et qui représente un défi plus important est la migration inverse. Cette étape est particulièrement délicate lorsque l'outil crée des nouveaux éléments ou lorsqu'il *fusionne* des éléments existants. Ces deux cas posent la question de l'intégration des nouveaux éléments dans le contexte initial. Ce problème est résolu en utilisant le graphe de dépendances et les identifiants uniques pour tracer l'évolution des instances, à travers les différentes étapes de la réutilisation.

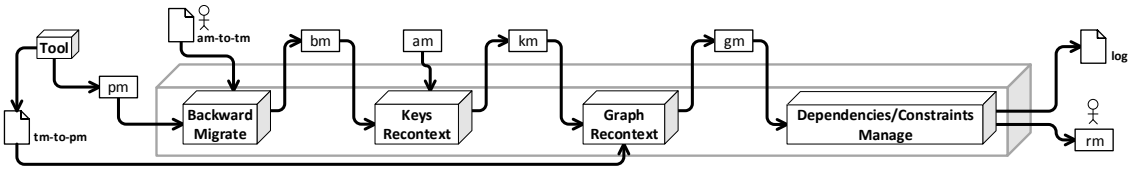


FIGURE 3.11 – Sous-étapes de la migration inverse

L'intégration du résultat de l'outil dans le contexte initial est l'objectif de cette dernière étape du round-trip de migration. Elle est représentée par la figure 3.11 et peut être décrite comme suit :

- *Migration retour* (*BackwardMigrate*) : cette étape inverse la migration initiale sur les instances de la sortie de l'outil ; elle transforme pm en un modèle inversé bm sans modification de la structure du graphe (seul le nom des attributs et des références peut être modifié).
- *Recontextualisation par clés* (*KeysRecontext*) : dans cette étape, bm est étendu avec les éléments du modèle initial am qui ont été supprimés lors de la migration. Les instances récupérées peuvent être déconnectées des autres, notamment lorsque l'outil a supprimé des instances.
- *Recontextualisation par graphe* (*GraphRecontext*) : cette étape permet de construire des *liens* entre le modèle recontextualisé par clés km et les informations fournies par le graphe de dépendances tm-to-pm.

– *Gestion des contraintes liées au méta-modèle (Dependencies/ConstraintsManage)* : cette étape vise à faire que gm soit conforme à AMM en respectant les contraintes de type.

Ces étapes sont maintenant présentées.

3.5.1 Migration retour

La migration retour est considérée comme une migration inverse de la migration initiale. Elle annule les opérations de migration qui ont été appliquées à un modèle et elle produit un nouveau modèle qui est dans le contexte du méta-modèle initial. La migration retour se base sur la spécification de migration qui a permis de migrer le modèle initial et de le placer dans le contexte de l'outil. Il est à noter que le modèle migré a été traité par un outil et que la migration retour s'applique sur le modèle produit par l'outil. Concernant la migration retour, nous traitons les cas suivants d'inversion de migration :

– Aucune modification.
– Migration retour simple.
– Inversion d'une suppression.
– Inversion d'une migration complexe.

Ces cas sont expliqués par la suite.

Aucune modification. La migration ne modifie pas le graphe de modèle. Si le modèle migré n'a pas été modifié, alors la migration inverse est simple et le modèle migré inversé est le modèle modifié par l'outil. Ce cas s'applique lorsque les opérateurs appliqués pour faire la migration sont le renommage de classes rename et l'aplatissement de la hiérarchie des classes flatten. Dans le cas du renommage, la classe prend son nom initial, mais ça ne modifie pas le graphe d'objets.

Migration retour simple. Le renommage d'un attribut ou d'une référence (*i.e* l'attribut *nb* devient l'attribut *val*) est annulé par le renommage inverse (*val* devient *nb*). Cette inversion consiste à renommer les attributs et les références (affectés par la migration) du modèle traité par l'outil.

Inversion d'une suppression. Si l'opérateur remove est appliqué lors de la migration, le seul opérateur inverse envisageable est un opérateur d'ajout. Il faudrait alors disposer de constructeurs par défaut, par exemple, basés sur le calcul de propriétés dérivées. Dans ce cas, ces opérateurs ne s'appuient pas sur l'information initiale issue de la migration, information qui est donc perdue. La création de l'information est alors arbitraire. Pour éviter ce problème, pour le moment et pour cette étape, la migration retour n'apporte pas de modifications sur le modèle traité par l'outil.

Inversion d'une migration complexe Si la migration implique un opérateur complexe comme hide, la migration retour n'est pas calculable parce qu'elle implique l'ajout de références qui ont construit les références dérivées. Pour cette étape, la migration retour n'apporte pas de modifications par rapport au modèle traité par l'outil.

Au final, l'inversion de la migration initiale pour obtenir le modèle bm est simple par choix et construction. Ainsi, tous les éléments de pm sont copiés dans bm. Certains attributs et certaines références sont renommés si leur nom a été initialement modifié. À ce stade, bm ne contient pas de données supplémentaires par rapport à pm.

À titre d'illustration nous prenons le modèle de la figure 3.10 modifié par l'outil et qui doit être inversé. La figure 3.12 illustre le modèle de la figure 3.10, mais cette fois ci, inversé. Ici l'inversion n'a pas apporté de modifications sur le modèle modifié par l'outil. Le modèle est maintenant dans le contexte du domaine d'application mais les éléments du modèle initial (*c.f* figure 3.5) qui ont été supprimés lors de la migration ($label$, b_1, c_1 avec leur attributs et leur références), ne sont pas récupérés (ils ne sont pas mis dans le modèle).

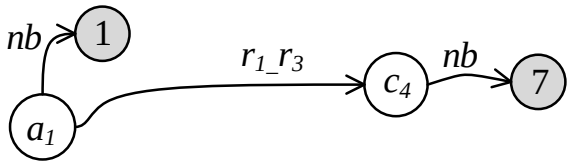


FIGURE 3.12 – Modèle migré inversé

Afin d'améliorer la migration retour pour récupérer les éléments supprimés, nous introduisons l'étape de recontextualisation par clés.

3.5.2 Recontextualisation par clés

Dans l'étape précédente, le modèle traité par l'outil est placé dans le domaine d'application. La migration inverse des opérateurs rename et flatten est appliquée. Mais la migration inverse des opérateurs remove et hide n'a pas été appliquée. Les éléments du modèle initial qui ont été supprimés lors de la migration n'ont pas été récupérés. Dans cette étape, l'objectif est de récupérer (mettre dans le modèle recontextualisé) ces éléments du modèle initial am. La recontextualisation par clés est gérée par le moteur KeysRecontext. La migration par clés est possible grâce à l'application du concept de *jointure naturelle* des bases de données relationnelles.

Tout d'abord, les éléments du modèle initial à récupérer sont identifiés, ces éléments correspondent aux instances, aux attributs et aux références supprimés lors de la migration. La recontextualisation par clé permet de placer les éléments du modèle initial dans sa version migrée inversée (après la traitement par l'outil).

L'*identifiant* de chaque instance est utilisé comme une *clé* et permet de reconnecter un modèle migré inversé à son contexte initial. La recontextualisation par clés prend le modèle migré inversé et réalise *dans l'ordre* les étapes décrites ensuite :

1. Toutes les instances supprimées de am sont ajoutées à km.
2. Tous les attributs supprimés de am sont récupérés *si et seulement si* leurs instances initiales sont dans km.
3. Les attributs dont la source ne fait pas partie du modèle recontextualisé sont récupérés, mais non connectés aux instances.

4. Toutes les références supprimées de am sont récupérées *si et seulement si* leur instance source initiale et leur instance cible initiale sont dans km.
5. Les autres références sont récupérées.

Par définition de la recontextualisation par clés, toutes les instances supprimées lors de la migration sont récupérées, de même toutes les valeurs scalaires sont récupérées. Mais il est possible que tous les éléments récupérés ne soient pas reconnectés au modèle. Dans cette étape, les instances du modèle migré inversé qui ont été crées par l'outil ne peuvent pas être connectées à des éléments du modèle initial.

À titre d'illustration, la figure 3.13 présente les éléments du modèle initial à récupérer et le modèle migré inversé dans lequel ils doivent être placés. Par la suite, nous illustrons les actions de la recontextualisation par clés appliquées au modèle inversé de la figure 3.13.

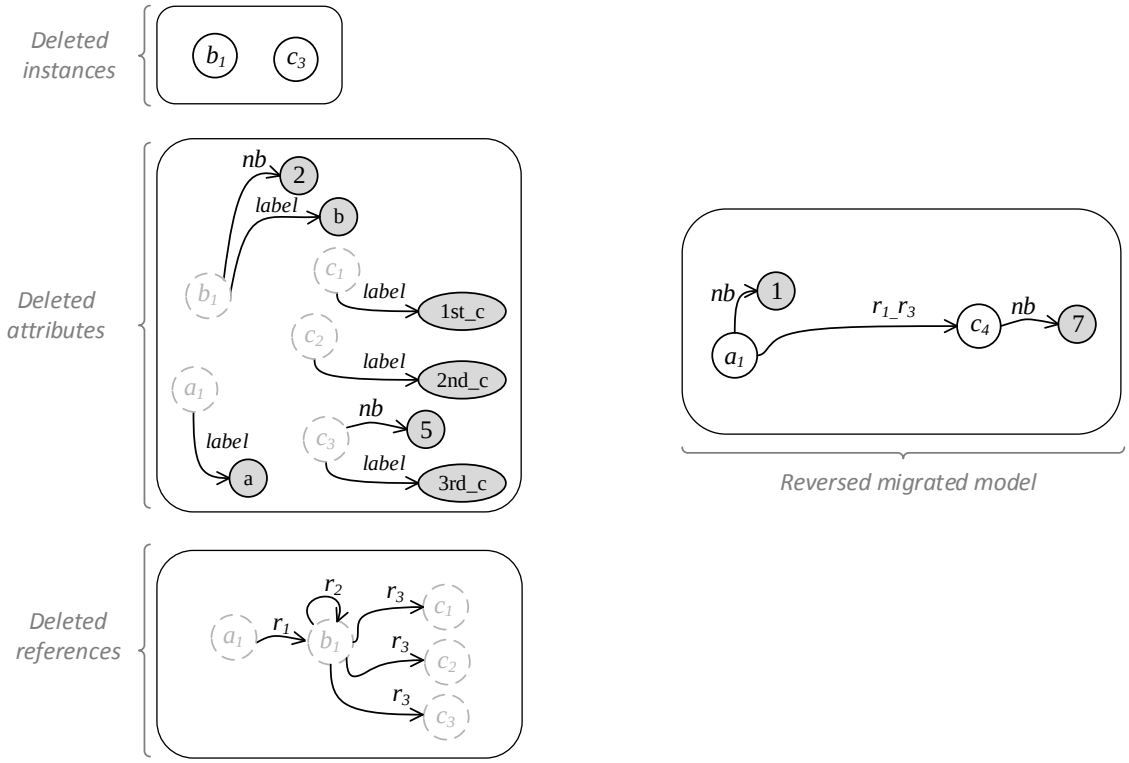


FIGURE 3.13 – Éléments supprimés lors de la migration

La figure 3.14 illustre la récupération des instances. b_1 et c_3 sont maintenant placées dans le modèle recontextualisé par clés.

La figure 3.15 illustre la recontextualisation des attributs dont leur cible est dans le modèle recontextualisé par clés. *label* a, *nb* 2, *label* b, *nb* 5 et *label* 3rd_c sont maintenant dans le modèle.

La figure 3.16 illustre la recontextualisation des attributs dont la source ne fait pas partie du modèle recontextualisé. *label* 1st_c et *label* 2nd_c sont maintenant dans le modèle.

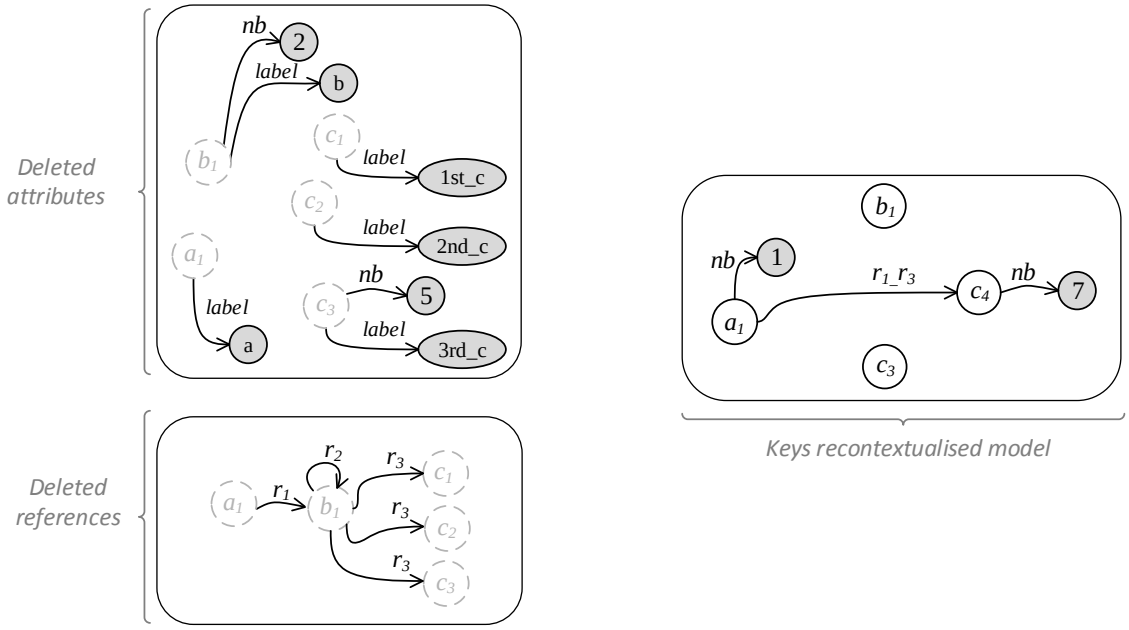


FIGURE 3.14 – Récupération des instances (première étape)

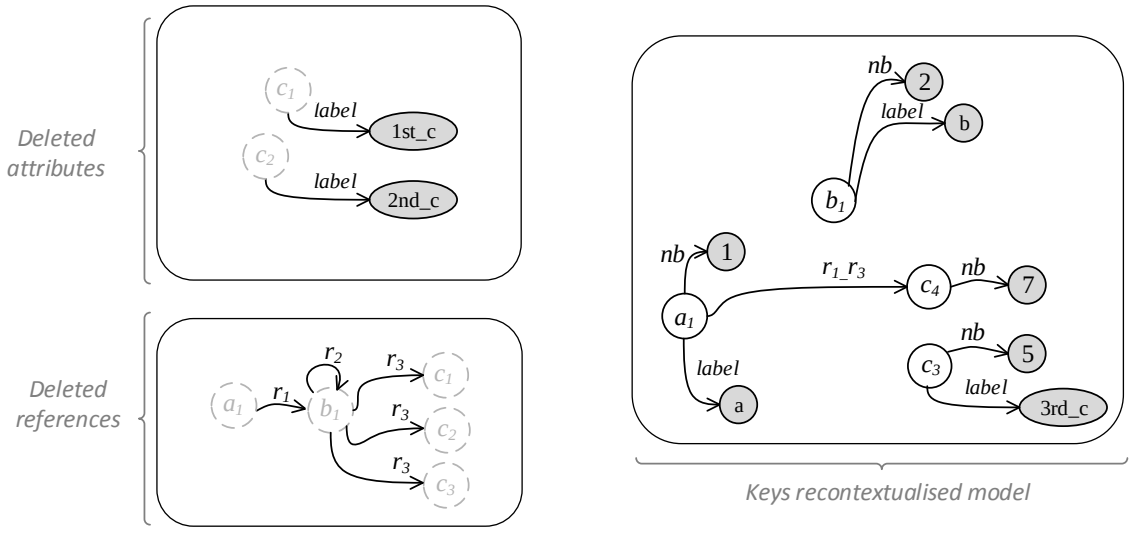


FIGURE 3.15 – Récupération des attributs (deuxième étape)

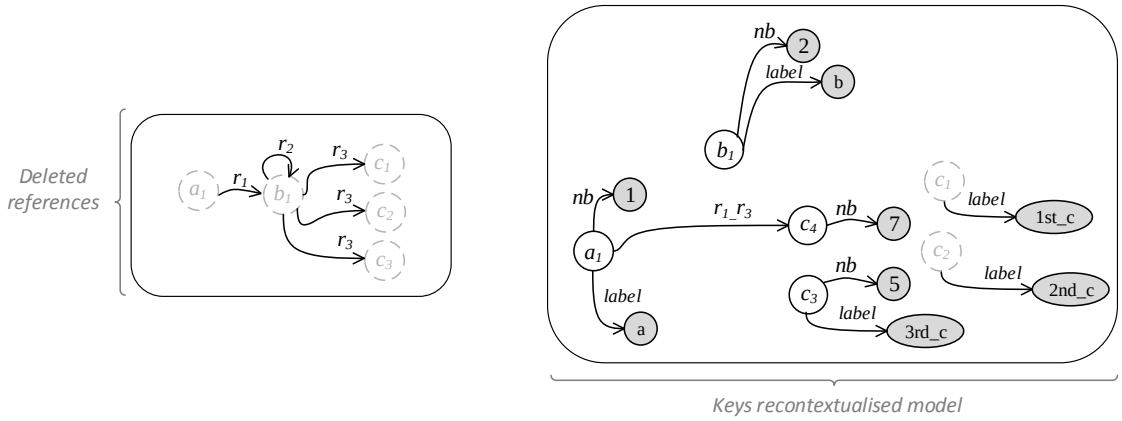


FIGURE 3.16 – Récupération des attributs (troisième étape)

La figure 3.17 illustre la recontextualisation des références dont la source et la cible sont dans le modèle recontextualisé par clés. r_1, r_2 et r_3 (entre b_1 et c_3) sont maintenant dans le modèle. Les instances b_1 et c_3 sont connectées au reste du modèle.

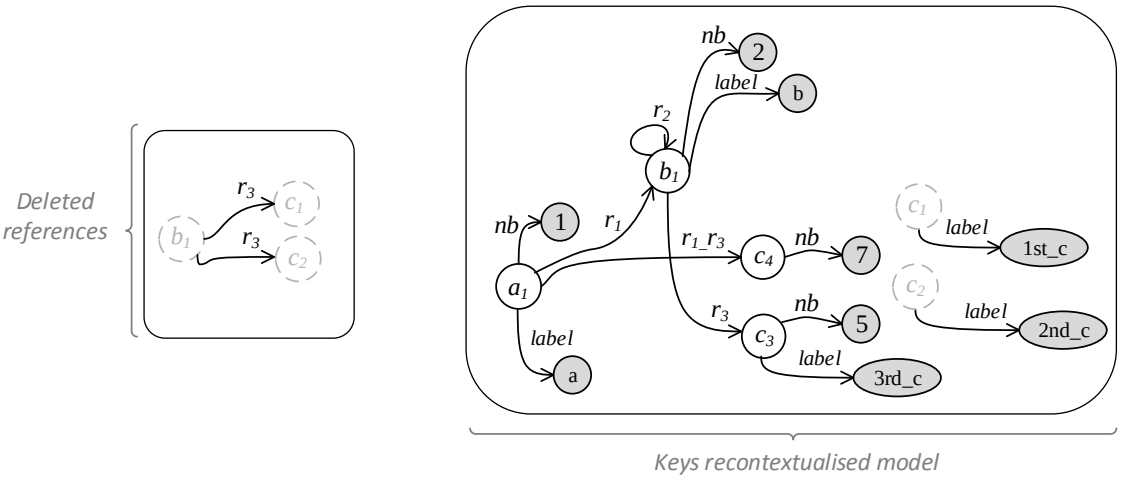


FIGURE 3.17 – Récupération des références (quatrième étape)

La figure 3.18 illustre le modèle recontextualisé par clés. À ce point, tous les éléments à récupérer sont effectivement récupérés, le résultat est un graphe non connexe dans lequel certains attributs et certaines références restent déconnectés du reste du modèle (r_3 dont la cible est c_1, r_3 dont la cible est c_2, $label$ 1st_c et $label$ 2nd_c). Et l'instance c_4 (crée par l'outil) n'a pas d'attribut $label$.

Les instances de sortie de l'outil sont reconnectées aux instances du modèle initial qui ont été récupérées. Cependant les instances nouvelles ne peuvent pas être connectées avec des instances existantes parce qu'elles n'ont pas de liens avec les éléments du modèle initial. Les relations entre les instances sont construites à partir des informations données par les clés.

Cette étape de recontextualisation par clés présente des limites lorsque l'outil crée des instances et lorsque l'outil supprime des instances parce que certaines références et certains attributs récupérés ne retrouvent plus leurs instances source. Nous présen-

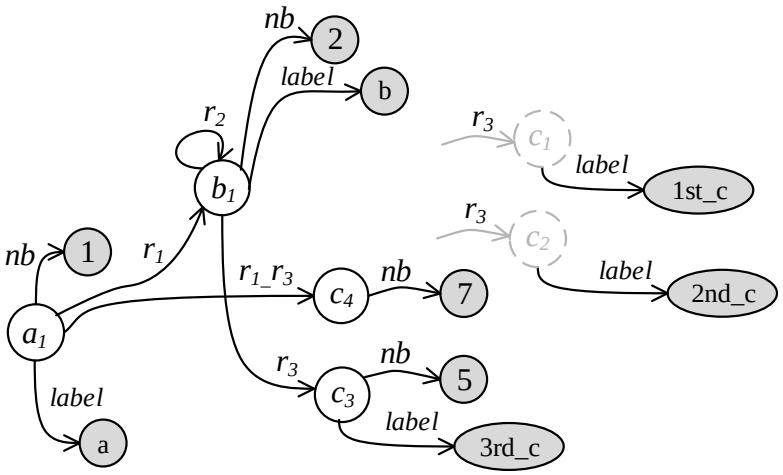


FIGURE 3.18 – Modèle recontextualisé par clés (cinquième étape)

tons maintenant l'utilisation d'un graphe de dépendances pour recontextualiser les instances supprimées, mais aussi les nouvelles.

3.5.3 Recontextualisation par graphe de dépendances

La recontextualisation par graphe de dépendances améliore la recontextualisation par clés. En effet, les liens entre le modèle migré inversé et son contexte initial peuvent être recréés non seulement entre les instances initialement présentes et conservées, mais aussi entre les instances créées par l'outil et les instances qui ont été utilisées pour les calculer.

Le premier choix de recontextualisation est fait grâce à une interprétation par défaut du graphe de dépendances. Cependant cette interprétation par défaut peut être personnalisée. Notamment, tous les attributs et toutes les références de *chaque* instance pourraient être mis sur toutes les instances qui ont été utilisées pour le calculer ou pour le mettre à jour. Les étapes réalisées par le moteur de recontextualisation par graphe de dépendances sont :

1. Un attribut a dont la source dans le modèle initial est i et qui n'a pas de source dans le modèle recontextualisé par clés (km), est connecté à chacune des instances (*i.e.* $i2$, $i3$) du km qui ont été calculées ou mises à jour par l'outil à partir de a. Le graphe de dépendances indique que $i2$ et $i3$ ont été produites à partir de i. Ceci est fait pour tous les attributs non connectés dans km.
2. Une référence r qui est déconnectée dans le modèle km mais dont l'instance source i est présente dans km, est connecté entre l'instance source et chacune des instances crées ou calculées par l'outil à partir de l'instance cible i_2 de la référence dans am. Le graphe de dépendances indique que i_3 et i_4 ont été calculées à partir de i_2. Ceci est fait pour chacune des instances non connectées dans km.
3. Une référence r qui est déconnectée dans le modèle km est connectée entre chaque instance de km calculée à partir de la source de r dans am et chaque instance de km calculée à partir de la cible de r dans am. Ceci est fait pour toutes les instances non connectées de km.

Cette utilisation du graphe est une utilisation possible. Elle reste adaptable selon la stratégie de recontextualisation applicable. Un autre comportement proposé est de copier *toutes* les caractéristiques (attribut ou référence) d'une instance supprimée sur toutes les instances (récupérables ou non) qui ont été calculées à partir de cela.

La figure 3.19 illustre le modèle initial, le modèle recontextualisé par clés et le graphe de dépendances. Le modèle recontextualisé sera amélioré en utilisant les informations fournies par le graphe de dépendances et le modèle initial.

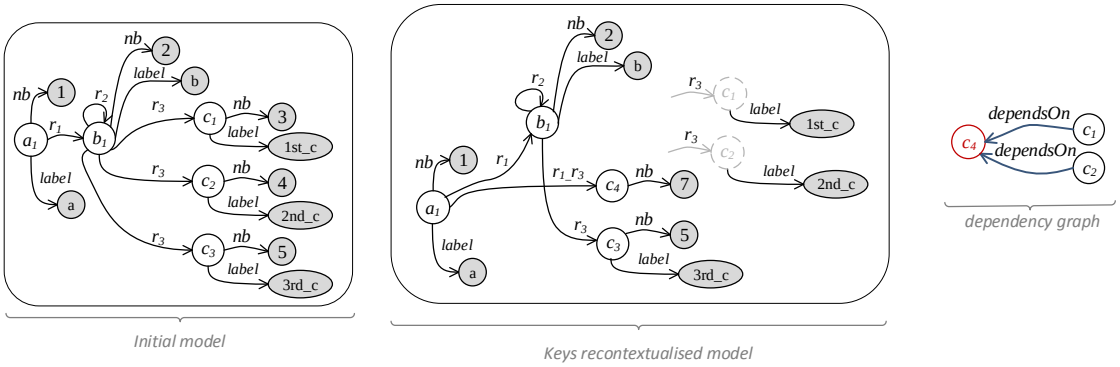


FIGURE 3.19 – Exemple de recontextualisation par graphe de dépendances

Les figures suivantes illustrent les étapes pour réaliser la recontextualisation par graphe de dépendances. La figure 3.20 illustre la recontextualisation des attributs $label$ 1st_c et $label$ 2nd_c. $label$ 1st_c est un attribut dont la cible c_1 n'est pas dans le modèle recontextualisé, le graphe de dépendances indique que c_1 à participé dans le calcul de l'instance c_4. $label$ 1st_c est donc connecté à c_4. $label$ 2nd_c est connecté à c_4 parce que l'instance source c_2 dans am à participé à la création de c_4.

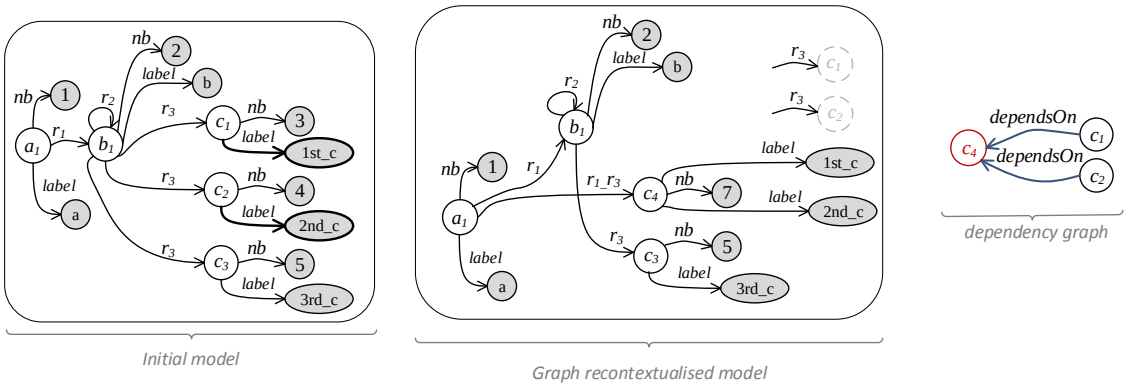


FIGURE 3.20 – Recontextualisation des attributs (première étape)

La figure 3.21 illustre la recontextualisation de la référence r_3. Dans le modèle initial r_3 reliait les instances b_1 et c_1. Le graphe de dépendances indique que c_4 à été créée à partir de c_1, r_3 est alors placée entre b_1 et c_4. L'autre référence r_3 reliait les instances b_1 et c_2, c_2 à participé à la création de c_4, r_3 est placée entre les instances b_1 et c_4. Étant donnée qu'une référence r_3 relie déjà ces deux instances, cette dernière est ignorée.

La troisième étape de la recontextualisation par graphe de dépendance, n'apporte pas de modifications dans cet exemple parce que toutes les références sont connectées dès la deuxième étape.

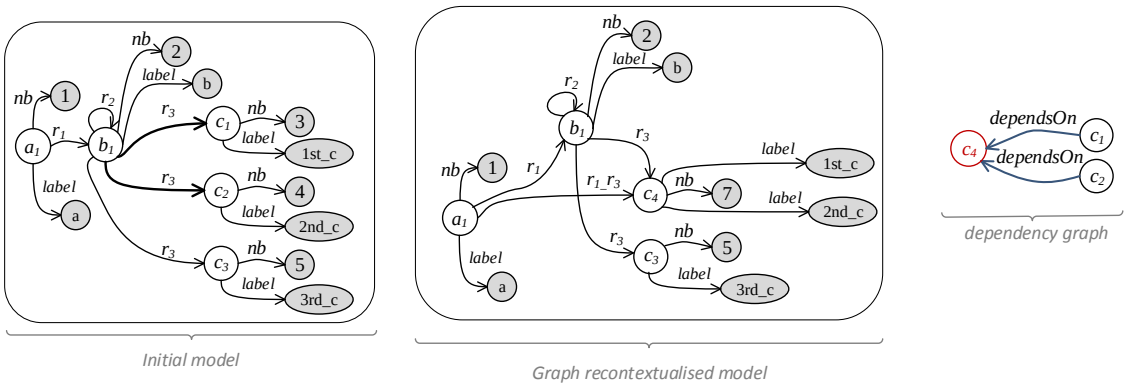


FIGURE 3.21 – Recontextualisation des références (deuxième étape)

Dans cet exemple, tous les éléments ont été recontextualisés. Cependant, le graphe d'objets résultat ne correspond pas à un modèle valide du point de vue du méta-modèle initial (*c.f.* figure 3.2). En effet, le modèle recontextualisé par graphe de dépendances contient une référence $r_1\_r_3$ entre les instances b_1 et c_4 alors que le méta-modèle n'a pas cette référence. En plus, l'instance c_4 est munie de deux attributs $label$ alors qu'au plus un attribut $label$ peut être associé à une instance.

À ce stade, tous les éléments récupérés lors de la recontextualisation par clés sont recontextualisés. Même les instances crées par le modèle ont désormais un contexte provenant du modèle initial. Nous proposons, maintenant une gestion des contraintes liées au méta-modèle.

3.5.4 Gestion des contraintes liées au méta-modèle

Les étapes précédentes fournissent une version recontextualisée du résultat de l'outil. Elles combinent le résultat de l'outil et les éléments du modèle initial qui avaient été supprimés. Ce modèle est destiné à être conforme au méta-modèle du domaine d'application (AMM dans la figure 3.1). La représentation de modèles sous la forme de graphe d'objets permet plus de souplesse au moment de manipuler les modèles. Mais elle peut produire des modèles non conformes au méta-modèle.

Cette dernière étape de la migration inverse est gérée par le moteur Dependencies/ConstraintsManage et consiste à assurer la conformité de modèle recontextualisé en vérifiant les contraintes de conformité suivantes :

- La cardinalité.
- La cohérence des références opposées.
- La cohérence de conteneurs.

La conformité de ces contraintes est fixée automatiquement par le moteur. Les références du modèle qui ne font pas partie du méta-modèle sont supprimées automatiquement. La contrainte de cardinalité est fixée en faisant des choix par défaut afin de garder la quantité d'éléments requise. En général, l'élément choisi est celui qui apparait en premier dans la liste d'éléments connectés à l'instance. Le code de gestion de contraintes est *ouvert* et peut être modifié par l'utilisateur. Les éléments qui ne sont pas conservés sont enregistrés dans un fichier log. Ce fichier contient des informations sur ces éléments et les possibles instances auxquelles ils peuvent être connectés.

À titre d'illustration, la figure 3.22 présente deux possibilités de gestion de la contrainte de cardinalité de l'attribut *nb* pour le modèle de la figure 3.21. Dans les deux cas, l'attribut qui n'est pas conservé est enregistré dans un fichier log. L'utilisateur peut consulter le fichier et ajouter au modèle les éléments qui sont enregistrés. Ceci permet à l'utilisateur de valider les choix par défaut ou de modifier le résultat.

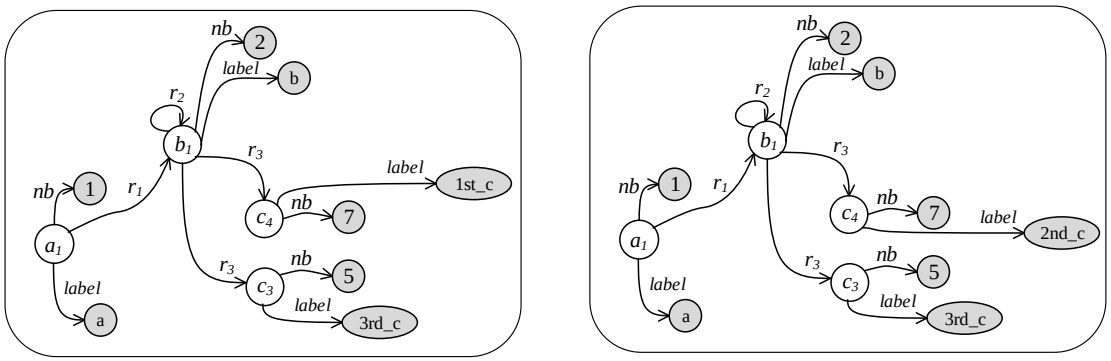


FIGURE 3.22 – Exemple de gestion des contraintes

La gestion de contraintes garanti que le modèle résultant est syntaxiquement correct, la sémantique doit être validée par l'utilisateur qui joue le rôle d'*expert du domaine*.

3.6 Formalisation

Dans cette section nous présentons la formalisation des principes de notre approche. Ces principes ont été introduits dans les sections précédentes. Cette formalisation n'est décrite que pour l'opérateur remove (elle est également valide pour l'opérateur flatten). Nous considérons que le modèle source $\mathfrak{m}$ est un modèle valide. Nous faisons l'hypothèse que l'outil à réutiliser est un outil de réécriture.

Les différentes étapes du round-trip de migration grâce aux fonctions Migrate, Tool, BackwardMigrate, KeysRecontext et GraphRecontext sont synthétisées ainsi :

$$\mathfrak{m} \xrightarrow{\text{Migrate}} \mathfrak{tm} \xrightarrow{\text{Tool}} \mathfrak{pm} \xrightarrow{\text{BacwardMigrate}} \mathfrak{bm} \xrightarrow{\text{KeysRecontext}} \mathfrak{km} \xrightarrow{\text{GraphRecontext}} \mathfrak{rm}$$

Au final, on obtient par composition la réutilisation de Tool dans le domaine étendu de $\mathfrak{m}$, ce qui permet d'obtenir $\mathfrak{rm}$.

3.6.1 Graphe d'objets

Les *espaces de noms* correspondant aux instances, aux valeurs scalaires, aux attributs et aux références sont introduites par la suite. Ces espaces de noms sont des *alphabets, i.e.* des ensembles finis non-vides de symboles :

$\mathcal{I}$: instances $\quad$ $\S$: valeurs scalaires $\quad$ $\mathcal{A}$: attributs $\quad$ $\mathcal{R}$: références

Nous appelons *modèle* et notons $\mathfrak{m}$ le triplet composé d'un ensemble d'instances correspondant aux sommets noté V, et deux ensembles d'arcs notés E_a et E_r (*c.f.* définition 3.1). Le premier ensemble d'arcs dénote les attributs. Ils relient les instances

à des valeurs scalaires (par l'intermédiaire de noms). Le second dénote les références. Elles relient les instances entre elles (par l'intermédiaire de noms). On note $\mathfrak{m}.\mathrm{V}$, $\mathfrak{m}.\mathrm{E}_a$ et $\mathfrak{m}.\mathrm{E}_r$ les composantes V, E_a et E_r d'un modèle $\mathfrak{m}$ donné.

Definition 3.1 *Graphe d'objets*

$$\mathfrak{m} \triangleq \left(\mathrm{V}\,,\,\mathrm{E}_a\,,\,\mathrm{E}_r\right) \quad avec \begin{cases} \mathrm{V} & \subseteq & \mathcal{I} \\ \mathrm{E}_a & \subseteq & \mathrm{V}\times\mathcal{A}\times\S \\ \mathrm{E}_r & \subseteq & \mathrm{V}\times\mathcal{R}\times\mathrm{V} \end{cases}$$

À titre d'illustration, on note $\mathfrak{m}$ le modèle de la figure 3.5. Il est formellement et explicitement défini par ses composants :

$$\begin{array}{rcl} \mathfrak{m}.\mathrm{V} & = & \{a_1, b_1, c_1, c_2, c_3\} \\ \mathfrak{m}.\mathrm{E}_a & = & \{(a_1, nb, 1), (a_1, label, a), (b_1, nb, 2), (b_1, label, b), (c_1, nb, 3), \\ & & (c_1, label, 1st\_c), (c_2, nb, 4), (c_2, label, 2nd\_c), (c_3, nb, 5), (c_3, label, 3rd\_c)\} \\ \mathfrak{m}.\mathrm{E}_r & = & \{(a_1, r1, b_1), (b_1, r_2, b_1), (b_1, r3, c_1), (b_1, r3, c_2), (b_1, r3, c_3)\} \end{array}$$

3.6.2 Spécification de migration

Pour un modèle $\mathfrak{m}$ donné, nous appelons *spécification de migration* et notons $\overrightarrow{\mathfrak{m}}$ le quadruplet composé de $\mathfrak{m}$ et de trois ensembles noté D_i, D_a et D_r. Ces ensembles spécifient les instances, les attributs et les références de $\mathfrak{m}$ qui doivent être *supprimés* (*c.f.* définition 3.2). On note $\overrightarrow{\mathfrak{m}}.\mathrm{D}_i$, $\overrightarrow{\mathfrak{m}}.\mathrm{D}_a$ et $\overrightarrow{\mathfrak{m}}.\mathrm{D}_r$ les composants D_i, D_a et D_r d'une spécification de migration $\overrightarrow{\mathfrak{m}}$ donnée.

Definition 3.2 *Spécification de migration*

$$\overrightarrow{\mathfrak{m}} \triangleq \left(\mathfrak{m}\,,\,\mathrm{D}_i\,,\,\mathrm{D}_a\,,\,\mathrm{D}_r\right) \quad avec \begin{cases} \mathrm{D}_i & \subseteq & \mathfrak{m}.\mathrm{V} \\ \mathrm{D}_a & \subseteq & \mathfrak{m}.\mathrm{V}\times\mathcal{A} \\ \mathrm{D}_r & \subseteq & \mathfrak{m}.\mathrm{V}\times\mathcal{R} \end{cases}$$

D_i, D_a et D_r correspondent respectivement aux classes, attributs et références à *supprimer* au niveau modèle.

3.6.3 Migration

Nous appelons *Migration* et notons Migrate l'outil de génération de *modèles migrés* à partir de *spécifications de migration*. Il applique la suppression des instances, attributs et références dans le modèle source et produit ainsi le modèle cible (*c.f.* définition 3.3).

Definition 3.3 *Calcul du modèle migré*

$$\begin{array}{rcl} \mathfrak{tm} & = & \mathit{Migrate}(\overrightarrow{\mathfrak{m}}) \\ \mathfrak{tm}.\mathrm{V} & = & \mathfrak{m}.\mathrm{V} \setminus \overrightarrow{\mathfrak{m}}.\mathrm{D}_i \\ \mathfrak{tm}.\mathrm{E}_a & = & \mathfrak{m}.\mathrm{E}_a \setminus \{(i,a,s)\in\mathcal{I}\times\mathcal{A}\times\S \mid i\in\overrightarrow{\mathfrak{m}}.\mathrm{D}_i \vee (i,a)\in\overrightarrow{\mathfrak{m}}.\mathrm{D}_a\} \\ \mathfrak{tm}.\mathrm{E}_r & = & \mathfrak{m}.\mathrm{E}_r \setminus \{(i,r,i')\in\mathcal{I}\times\mathcal{R}\times\mathcal{I} \mid i\in\overrightarrow{\mathfrak{m}}.\mathrm{D}_i \vee i'\in\overrightarrow{\mathfrak{m}}.\mathrm{D}_i \vee (i,r)\in\overrightarrow{\mathfrak{m}}\mathrm{D}_r\} \end{array}$$

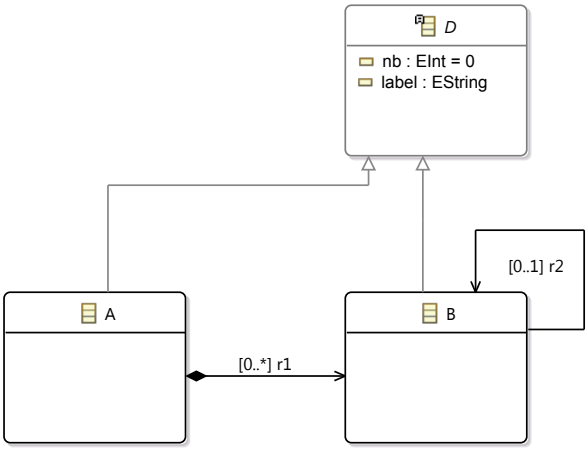


FIGURE 3.23 – Méta-modèle sans la classe C

Afin d'illustrer une spécification de migration, on note $\overrightarrow{\mathfrak{m}}$ la spécification de migration de $\mathfrak{m}$ vers un modèle conforme au méta-modèle de la figure 3.23 (*i.e.* un modèle dans lequel les instances c_1, c_2 et c_3 ont été *supprimées*).

Cette spécification est formellement et explicitement définie par ses composants :

$$\overrightarrow{\mathfrak{m}}.\mathrm{D}_i = \{c_1, c_2, c_3\} \qquad \overrightarrow{\mathfrak{m}}.\mathrm{D}_a = \emptyset \qquad \overrightarrow{\mathfrak{m}}.\mathrm{D}_r = \emptyset$$

Une fois que le migrateur Migrate a été appliqué à la spécification de migration $\overrightarrow{\mathfrak{m}}$, on obtient le *modèle migré* $\mathfrak{tm}$ formellement défini par ses composantes :

$$\begin{aligned}
\mathfrak{tm} &= \mathsf{Migrate}(\overrightarrow{\mathfrak{m}}) \\
\mathfrak{tm}.\mathrm{V} &= \{a_1, b_1\} \\
\mathfrak{tm}.\mathrm{E}_a &= \{(a_1, nb, 1), (a_1, label, a), (b_1, nb, 2), (b_1, label, b)\} \\
\mathfrak{tm}.\mathrm{E}_r &= \{(a_1, r1, b_1), (b_1, r2, b_1)\}
\end{aligned}$$

3.6.4 Outils de réécriture

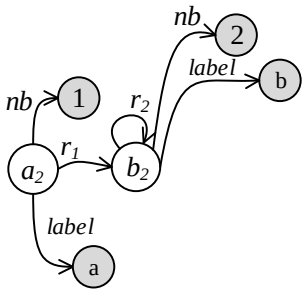
Dans un premier temps, nous caractérisons ce type d'outils dans le domaine sémantique des graphes. Nous appelons *outil de réécriture* et notons Tool l'outil produisant un *modèle de sortie* à partir d'un *modèle d'entrée*, et conforme au même méta-modèle. Tool peut être vue comme une *transformation endogène*. On note $\mathsf{Tool}(\mathfrak{pm})$ le modèle de sortie produit par Tool à partir d'un modèle d'entrée $\mathfrak{tm}$. Pour le moment, nous considérons Tool comme une *boîte grise*. Par conséquent, l'action de Tool ne peut être spécifiée que par les différences entre un modèle d'entrée donné et le modèle de sortie correspondant. Dans notre domaine sémantique, l'action de Tool est ainsi spécifiée par un ensemble d'éléments de graphe *supprimés* et un ensemble d'éléments *ajoutés* (*c.f.* définition 3.4).

Definition 3.4 *Spécification d'un outil de réécriture*

$$\textit{éléments supprimés} : \begin{cases} \mathsf{T}^{i}_{\mathrm{D}}(\mathfrak{tm}) = \mathfrak{tm}.\mathrm{V} \setminus \mathsf{T}(\mathfrak{tm}).\mathrm{V} & \textit{(instances supprimées)} \\ \mathsf{T}^{a}_{\mathrm{D}}(\mathfrak{tm}) = \mathfrak{tm}.\mathrm{E}_a \setminus \mathsf{T}(\mathfrak{tm}).\mathrm{E}_a & \textit{(attributs supprimés)} \\ \mathsf{T}^{r}_{\mathrm{D}}(\mathfrak{tm}) = \mathfrak{tm}.\mathrm{E}_r \setminus \mathsf{T}(\mathfrak{tm}).\mathrm{E}_r & \textit{(références supprimées)} \end{cases}$$

$$\textit{éléments ajoutés} : \begin{cases} \mathsf{T}^{i}_{\mathrm{A}}(\mathfrak{tm}) = \mathsf{T}(\mathfrak{tm}).\mathrm{V} \setminus \mathfrak{tm}.\mathrm{V} & \textit{(instances ajoutées)} \\ \mathsf{T}^{a}_{\mathrm{A}}(\mathfrak{tm}) = \mathsf{T}(\mathfrak{tm}).\mathrm{E}_a \setminus \mathfrak{tm}.\mathrm{E}_a & \textit{(attributs ajoutés)} \\ \mathsf{T}^{r}_{\mathrm{A}}(\mathfrak{tm}) = \mathsf{T}(\mathfrak{tm}).\mathrm{E}_r \setminus \mathfrak{tm}.\mathrm{E}_r & \textit{(références ajoutées)} \end{cases}$$

Afin d'illustrer ces outils de réécriture et leur caractérisation, on présente un exemple d'outil qui s'applique au modèle *migré* $\mathfrak{tm}$ de la figure 3.7 (conforme au métamodèle de la figure 3.23).



$$\begin{aligned} \mathfrak{pm} &= \mathsf{Cp}(\mathfrak{tm}) \\ \mathfrak{pm}.\mathrm{V} &= \{a_2, b_2\} \\ \mathfrak{pm}.\mathrm{E}_a &= \{(a_2, nb, 1), (a_2, label, a), (b_2, nb, 2), (b_2, label, b)\} \\ \mathfrak{pm}.\mathrm{E}_r &= \{(a_2, r_1, b_2), (b_2, r_2, b_2)\} \end{aligned}$$

FIGURE 3.24 – Copie du modèle migré sans actions

On appelle *copieur* et on note Cp l'outil qui crée une copie complète du modèle fourni en entrée. Il produit un nouveau modèle où toutes les instances sont nouvelles. Appliqué au modèle $\mathfrak{tm}$ de la figure 3.7, il produit le modèle $\mathfrak{pm}$ illustré et formellement défini dans la figure 3.24.

Par définition, l'action de Cp sur $\mathfrak{tm}$ est formellement définie ainsi :

$$\text{suppressions} : \{\mathsf{Cp}^{i}_{\mathrm{D}}(\mathfrak{tm}) = \mathfrak{tm}.\mathrm{V} \; ; \; \mathsf{Cp}^{a}_{\mathrm{D}}(\mathfrak{tm}) = \mathfrak{tm}.\mathrm{E}_a \; ; \; \mathsf{Cp}^{r}_{\mathrm{D}}(\mathfrak{tm}) = \mathfrak{tm}.\mathrm{E}_r$$

$$\text{ajouts} : \begin{cases} \mathsf{Cp}^{i}_{\mathrm{A}}(\mathfrak{tm}) = \{a_2, b_2\} \\ \mathsf{Cp}^{a}_{\mathrm{A}}(\mathfrak{tm}) = \{(a_2, nb, 1), (a_2, label, a), (b_2, nb, 2), (b_2, label, b)\} \\ \mathsf{Cp}^{r}_{\mathrm{A}}(\mathfrak{tm}) = \{(a_2, r_1, b_2), (b_2, r_2, b_2)\} \end{cases}$$

3.6.5 Migration retour

On appelle *migrateur retour* et on note BackwardMigrate l'outil qui annule les opérations de migration qui ont été appliquées à un modèle initial donné $\mathfrak{m}$. Ainsi, BackwardMigrate s'applique à un modèle qui est *supposé* avoir été migré. Si la spécification de migration est connue et si le modèle migré n'a pas été modifié, alors la migration inverse est triviale et déjà disponible sans calcul :

$$\mathsf{BackwardMigrate}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})) = \mathfrak{m}.$$

Nous nous concentrons maintenant sur le problème plus subtil d'un modèle initial $\mathfrak{m}$ qui a été migré via une spécification de migration $\overrightarrow{\mathfrak{m}}$, *et qui a ensuite été modifié* par

un outil de réécriture Tool. On note $\mathfrak{bm}$ le modèle produit par le migrateur inverse dans cette situation :

$$\mathsf{BackwardMigrate}(\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))) = \mathfrak{bm} \quad \text{(modèle migré inverse)}$$

Dans ce contexte, nous essayons de récupérer autant que possible le modèle initial *sans défaire l'action de* Tool.

Dans un premier temps, dans la migration retour, rien n'est supposé être ajouté ou supprimé :

$$\mathsf{BackwardMigrate}(\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))) = \mathsf{Migrate}(\overrightarrow{\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))})$$

$$\text{avec : } \overrightarrow{\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))} = (\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})), \emptyset, \emptyset, \emptyset)$$

La migration retour est obtenue à partir de cette spécification de migration conformément aux principes de la définition 3.3. Par définition, elle correspond donc à l'*identité* :

$$\mathsf{ReverseMigrate}(\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))) = \mathsf{Migrate}(\overrightarrow{\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))}) = \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))$$

Afin d'illustrer ce mécanisme, nous reprenons l'exemple d'outil introduit dans la section précédente : le copieur (Cp). Il s'applique à des modèles conformes au méta-modèle de la figure 3.23. Dans ce cas, nous étudions le résultat de la migration retour du modèle migré sans actions (section 3.6.3) après son traitement par le copieur. Le résultat est un modèle supposé conforme au méta-modèle de la figure 3.2.

Conformément à la définition du *copieur*. Le résultat du copieur appliqué au modèle de la figure 3.4 est le modèle noté $\mathfrak{pm}$ de la figure 3.24 (*c.f.* section 3.6.4), et ainsi :

$$\mathsf{BackwardMigrate}(\mathsf{Cp}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))) = \mathsf{Cp}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})) = \mathfrak{pm} \quad \text{(modèle de la figure 3.24)}$$

Le résultat de la migration retour est un modèle équivalent au modèle migré sans actions. Mais dans lequel les instances ont été *récrées*. Les instances c_1, c_2 et c_3 originales ne sont pas prises en compte par la migration retour et ne peuvent pas être retrouvées et sont perdues.

3.6.6 Recontextualisation par clés

Tout d'abord, on introduit le *contexte initial* d'un modèle $\mathfrak{m}$ comme l'ensemble des éléments de $\mathfrak{m}$ qui doivent être supprimés conformément à une spécification de migration $\overrightarrow{\mathfrak{m}}$. Par définition des spécifications de migration (*c.f.* définition 3.2), ce contexte initial correspond à $\overrightarrow{\mathfrak{m}}.\mathrm{D}_i \cup \overrightarrow{\mathfrak{m}}.\mathrm{D}_a \cup \overrightarrow{\mathfrak{m}}.\mathrm{D}_r$.

On appelle *recontextualisation* l'outil qui permet de replacer le contexte initial d'un modèle $\mathfrak{m}$ donné dans sa version *migrée inversée* après le traitement d'un *outil de réécriture*. On note KeysRecontext cet outil :

$$\mathsf{KeysRecontext}(\mathsf{BackwardMigrate}(\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})))) = \mathfrak{tm}$$

(modèle migré inversé recontextualisé)

Le comportement de KeysRecontext (décrit informellement dans 3.5.2) est spécifiée par :

Definition 3.5 *Calcul d'un modèle recontextualisé par clés*

$$
\begin{aligned}
\mathfrak{bm} &= \mathsf{ReverseMigrate}(\mathit{Tool}(\mathit{Migrate}(\overrightarrow{\mathfrak{m}})) && \textit{(modèle migré inversé)}\\
\mathfrak{km} &= \mathit{KeysRecontext}(\mathfrak{bm}) && \textit{(modèle migré inversé recontextualisé)}\\
\mathfrak{km}.\mathrm{V} &= \mathfrak{bm}.\mathrm{V} \cup \overrightarrow{\mathfrak{m}}.\mathrm{D}_i\\
\mathfrak{km}.\mathrm{E}_a &= \mathfrak{bm}.\mathrm{E}_a \cup \{(i,a,s) \in \mathfrak{m}.\mathrm{E}_a \mid i \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_i \vee ((i,a) \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_a \wedge i \in \mathfrak{bm}.\mathrm{V})\}\\
\mathfrak{km}.\mathrm{E}_r &= \mathfrak{bm}.\mathrm{E}_r \cup \{(i,r,i') \in \mathfrak{m}.\mathrm{E}_r \mid (i,i') \in \mathfrak{km}.\mathrm{V}^2 \wedge (i \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_i \vee i' \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_i\\
&\qquad\qquad \vee (i,r) \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_r)\}
\end{aligned}
$$

Pour illustrer ce mécanisme, nous reprenons l'outil *copieur* (Cp). Comme mentionné précédemment, le copieur de Migrate($\overrightarrow{\mathfrak{m}}$) produit le modèle de la figure 3.24, lequel est équivalent au modèle migré (sans c_1, c_2, c_3) dans lequel les instances ont été recrées :

$$
\begin{aligned}
\mathfrak{bm} &= \mathsf{BackwardMigrate}(\mathsf{Cp}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))) && \text{(modèle de migré inversé)}\\
\mathfrak{km} &= \mathsf{KeysRecontext}(\mathfrak{bm}) && \text{(modèle migré inversé recontextualisé)}
\end{aligned}
$$

Par conséquent et par définition de la recontextualisation par clés (*c.f.* définition 3.5), cette mise à jour n'est pas défaite. Le modèle résultant contient bien les instances qui avaient été supprimées, mais pas les liens de référence entre le contexte initial et les instances du modèle, toutes nouvellement recréées :

$$
\begin{aligned}
\mathfrak{km}.\mathrm{V} &= \mathfrak{bm}.\mathrm{V} \cup \{c_1, c_2, c_3\}\\
\mathfrak{km}.\mathrm{E}_a &= \mathfrak{bm}.\mathrm{E}_a \cup \{(c_1, nb, 3), (c_1, label, 1st\_c), (c_2, nb, 4), (c_2, label, 2nd\_c),\\
&\qquad (c_3, nb, 5), (c_3, label, 3rd\_c)\}\\
\mathfrak{km}.\mathrm{E}_r &= \mathfrak{bm}.\mathrm{E}_r
\end{aligned}
$$

Les instances c_1, c_2 et c_3 supprimées sont effectivement récupérées mais elles ne sont pas connectées à l'instance b_2. Dans ce cas le graphe n'est pas *connexe*. Chaque composante connexe correspond à la sémantique d'un sous-modèle conforme au méta-modèle initial (de la figure 3.2 dans cet exemple), et qui peut être valide ou non selon les contraintes de multiplicité ou de composition.

3.6.7 Recontextualisation par graphe de dépendances

Pour un modèle d'entrée $\mathfrak{m}$ donné, on suppose que l'outil Tool fournit le modèle Tool($\mathfrak{m}$) résultant accompagné d'un *graphe de dépendances* noté Graph(Tool, $\mathfrak{m}$). Pour une instance i donnée de Tool($\mathfrak{m}$), ce graphe rend explicite les instances de $\mathfrak{m}$ qui ont été utilisées pour *créer* ou *mettre à jour* i.

Le graphe est formellement défini par fonction totale des instances de sortie (Tool($\mathfrak{m}$).V) vers l'ensemble des *parties* des instances d'entrée. Ainsi, Graph(Tool, $\mathfrak{m}$)(i) fournit l'ensemble (éventuellement vide) des instances d'entrée qui ont été utilisées pour calculer i :

Definition 3.6 *Graphe de dépendances*

$$
\mathsf{Graph}(\mathsf{Tool}, \mathfrak{m}) \triangleq \mathsf{Tool}(\mathfrak{m}).\mathrm{V} \rightarrow \P(\mathfrak{m}.\mathrm{V}) \qquad \textit{(instances en sortie vers instances en entrée)}
$$

On note GraphRecontext l'outil qui utilise les instances en tant que sommets du graphe de dépendances pour reconnecter un modèle migré inversé à un contexte initial. L'action de GraphRecontext complète l'action de KeysRecontext (*i.e.* recontextualisation par clés). Il s'applique à un couple *modèle* et *graphe de dépendances* :

$$\mathsf{GraphRecontext}\left(\mathsf{KeysRecontext}(\mathsf{BackwardMigrate}(\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))),\right.$$

$$\left.\mathsf{Graph}(\mathsf{Tool},\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))\right) = \mathfrak{gm}$$

Le comportement de GraphRecontext (décrit informellement dans la section 3.5.3) est spécifiée par :

Definition 3.7 *Calcul d'un modèle recontextualisé par graphe de dépendances*

$$\begin{aligned}
\mathfrak{km} &= \mathsf{KeysRecontext}(\mathsf{BackwardMigration}(\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})))\\
&\qquad\qquad \textit{(modèle migré inversé recontextualisé)}\}\\
g &= \mathsf{Graph}(\mathsf{Tool},\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))\\
&\qquad\qquad \textit{(graphe de dépendances)}\}\\
\mathfrak{gm} &= \mathsf{GraphRecontext}(\mathfrak{km}, g)\\
&\qquad\qquad \textit{(modèle recontextualisé par graphe)}\}\\
\mathfrak{gm}.\mathrm{V} &= \mathfrak{km}.\mathrm{V}\\
\mathfrak{gm}.\mathrm{E}_a &= \mathfrak{km}.\mathrm{E}_a \cup \{(i,a,s) \in \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V} \times \mathcal{A} \times \S \mid\\
&\qquad\qquad \exists i' \in g(i) \setminus \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V},\ (i',a,s) \in \mathfrak{m}.\mathrm{E}_a \wedge (i',a) \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_a\}\\
\mathfrak{gm}.\mathrm{E}_r &=\\
&\mathfrak{km}.\mathrm{E}_r \cup \{(i_1,r,i_2) \in \mathfrak{km}.\mathrm{V} \times \mathcal{R} \times \mathfrak{km}.\mathrm{V} \mid\\
&\quad \exists i'_1 \in g(i_1) \setminus \mathfrak{km}.\mathrm{V},\ (i'_1,r,i_2) \in \mathfrak{m}.\mathrm{E}_r \wedge ((i'_1,r) \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_r \vee i_2 \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_i)\}\\
&\vee \exists i'_2 \in g(i_2) \setminus \mathfrak{km}.\mathrm{V},\ (i_1,r,i'_2) \in \mathfrak{m}.\mathrm{E}_r \wedge ((i_1,r) \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_r \vee i_1 \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_i)\}\\
&\vee \exists (i'_1,i'_2) \in (g(i_1) \setminus \mathfrak{km}.\mathrm{V}) \times (g(i_2) \setminus \mathfrak{km}.\mathrm{V}),\ (i'_1,r,i'_2) \in \mathfrak{m}.\mathrm{E}_r \wedge (i'_1,r) \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_r\}
\end{aligned}$$

Pour illustrer ce mécanisme, nous reprenons le *copieur* (Cp). On considère à nouveau le modèle initial $\mathfrak{m}$ de la figure 3.5, et le modèle migré $\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})$. L'application du *copieur* produit un modèle équivalent dans lequel toutes les instances ont été recréées. Dans ce cas, le graphe de dépendances est défini comme suit. Il met simplement en relation les nouvelles instances et les instances originales dont elles sont les copies :

$$\mathsf{Graph}(\mathsf{Cp},\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})) \quad : \quad \begin{array}{l} \mathsf{Cp}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V} \rightarrow \P(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})\mathrm{V})\\ a_2 \mapsto \{a_1\}; b_2 \mapsto \{b_1\} \end{array}$$

Par définition de la recontextualisation par graphes (définition 3.7), on obtient :

$$\begin{aligned}
\mathfrak{gm} &= \mathsf{GraphRecontext}\left(\mathsf{KeysRecontext}(\mathsf{BackwardMigrate}(\mathsf{Cp}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))),\right.\\
&\qquad \left.\mathsf{Graph}(\mathsf{Cp},\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))\right)\\
\mathfrak{gm}.\mathrm{V} &= \mathsf{KeysRecontext}(\mathsf{BackwardMigrate}(\mathsf{Cp}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))).\mathrm{V}\\
\mathfrak{gm}.\mathrm{E}_a &= \mathsf{KeysRecontext}(\mathsf{BackwardMigrate}(\mathsf{Cp}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))).\mathrm{E}_a \cup \emptyset\\
\mathfrak{gm}.\mathrm{E}_r &= \mathsf{KeysRecontext}(\mathsf{BackwardMigrate}(\mathsf{Cp}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))).\mathrm{E}_r \cup\\
&\qquad \{(b_2,r_3,c_1),(b_2,r3,c_2),(b_2,r_3,c_3)\}
\end{aligned}$$

Grâce au graphe, les instances c_1, c_2 et c_3 liées à l'instance b_1 qui avaient disparu peuvent maintenant être connectées à la copie de l'instance b_1 dans le résultat. Le

modèle recontextualisé par graphe de dépendances est explicitement défini par ses composants :

$$\begin{aligned}
\mathfrak{rm}.\mathrm{V} &= \{a_2, b_2, c_1, c_2, c_3\} \\
\mathfrak{rm}.\mathrm{E}_a &= \{(a_2, nb, 1), (a_2, label, a), (b_2, nb, 2), (b_2, label, b), (c_1, nb, 3), \\
&\quad (c_1, label, 1st_c), (c_2, nb, 4), (c_2, label, 2nd_c), (c_3, nb, 5), (c_3, label, 3rd_c)\} \\
\mathfrak{rm}.\mathrm{E}_r &= \{(a_2, r_1, b_2), (b_2, r_2, b_2), (b_2, r_3, c_1), (b_2, r_3, c_2), \\
&\quad (b_2, r_3, c_3)\}
\end{aligned}$$

3.6.8 Propriétés

La recontextualisation par graphe de dépendances complète la recontextualisation par clés. La recontextualisation par graphe de dépendances (et donc aussi par clés) ne doit pas annuler les transformations opérées par l'outil de réécriture que l'on tente de réutiliser. Ces propriétés peuvent être vérifiées dans le cadre formel de notre domaine sémantique.

La recontextualisation par graphe de dépendances étend la recontextualisation par clés

Soit $\mathfrak{m}$, un modèle donné, $\overrightarrow{\mathfrak{m}}$ une spécification de migration, et Tool un outil de réécriture. Soit Graph(Tool, Migrate($\overrightarrow{\mathfrak{m}}$)), un graphe de dépendances défini sur Tool(Migrate($\overrightarrow{\mathfrak{m}}$)).

Un arc de Graph(Tool, Migrate($\overrightarrow{\mathfrak{m}}$)) reliant une instance i à elle-même n'apporte aucune information. En effet, cet arc indique que i est une instance dans le résultat qui était déjà dans le modèle d'entrée et dont la valeur dépend en partie d'elle-même. On dit que Graph(Tool, Migrate($\overrightarrow{\mathfrak{m}}$)) est *sémantiquement vide* si aucun de ses arc n'a de valeur ajoutée. Plus formellement, par définition :

$$\mathsf{Graph}(\mathsf{Tool}, \mathsf{Migrate}(\overrightarrow{\mathfrak{m}})) \text{ est } \textit{sémantiquement vide} \triangleq$$

$$\forall i \in \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V},\ \mathsf{Graph}(\mathsf{Tool}, \mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))(i) \subseteq \{i\}$$

Propriété 1 *Si* Graph(Tool, Migrate($\overrightarrow{\mathfrak{m}}$)) *est* sémantiquement vide*, alors la recontextualisation par graphe de dépendances correspond à la recontextualisation par clés :*

$$\mathsf{GraphRecontext}\left(\mathsf{KeysRecontext}(\mathsf{BackwardMigrate}(\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})))\ ,\right.$$
$$\left.\mathsf{Graph}(\mathsf{Tool}, \mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))\right) = \mathsf{KeysRecontext}(\mathsf{BackwardMigrate}(\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))))$$

Preuve *Soit* $\mathfrak{km}$ = KeysRecontext(BackwardMigrate(Tool(Migrate($\overrightarrow{\mathfrak{m}}$))))*, le modèle migré inversé et recontextualisé par clés. Soit* g = Graph(Tool, Migrate($\overrightarrow{\mathfrak{m}}$))*, un graphe de dépendances* sémantiquement vide*. Soit* gm = GraphRecontext(km, g)*, la version recontextualisée par le graphe* g*. Par définition de* GraphRecontext *(c.f. définition 3.7) :* $\mathfrak{gm}.\mathrm{V} = \mathfrak{km}.\mathrm{V}$*. Par définition de* GraphRecontext*, on a également :*

$$\begin{aligned}
\mathfrak{gm}.\mathrm{E}_a &= \mathfrak{km}.\mathrm{E}_a \cup \{(i, a, s) \in \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V} \times \mathcal{A} \times \S \mid \\
&\qquad \exists i' \in g(i) \setminus \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V},\ (i', a, s) \in \mathfrak{m}.\mathrm{E}_a \wedge (i', a) \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_a\}
\end{aligned}$$

Or, g est sémantiquement vide : $\forall i \in \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V},\ g(i) \subseteq \{i\}$, *et donc :*

$$\forall i \in \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V},\ g(i) \setminus \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V} = \emptyset$$

Et donc on a :

$$\begin{aligned}\mathfrak{gm}.\mathrm{E}_a &= \mathfrak{km}.\mathrm{E}_a \cup \{(i,a,s) \in \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V} \times \mathcal{A} \times \S \mid \exists i' \in g(i) \setminus \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V}, \ldots\} \\ &= \mathfrak{km}.\mathrm{E}_a \cup \{(i,a,s) \in \mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})).\mathrm{V} \times \mathcal{A} \times \S \mid \exists i' \in \emptyset, \ldots\} \\ &= \mathfrak{km}.\mathrm{E}_a \cup \emptyset = \mathfrak{km}.\mathrm{E}_a\end{aligned}$$

Par définition de GraphRecontext, *on a finalement :*

$$\begin{aligned}&\mathfrak{gm}.\mathrm{E}_r \quad = \\ &\mathfrak{km}.\mathrm{E}_r \cup \{(i_1,r,i_2) \in \mathfrak{km}.\mathrm{V} \times \mathcal{R} \times \mathfrak{km}.\mathrm{V} \mid \\ &\quad \exists i'_1 \in g(i_1) \setminus \mathfrak{km}.\mathrm{V},\ (i'_1,r,i_2) \in \mathfrak{m}.\mathrm{E}_r \wedge ((i'_1,r) \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_r \vee i_2 \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_i)\} \\ &\vee \exists i'_2 \in g(i_2) \setminus \mathfrak{km}.\mathrm{V},\ (i_1,r,i'_2) \in \mathfrak{m}.\mathrm{E}_r \wedge ((i_1,r) \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_r \vee i_1 \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_i)\} \\ &\vee \exists (i'_1,i'_2) \in (g(i_1) \setminus \mathfrak{m}_\mathsf{k}.\mathrm{V}) \times (g(i_2) \setminus \mathfrak{km}.\mathrm{V}),\ (i'_1,r,i'_2) \in \mathfrak{m}.\mathrm{E}_r \wedge (i'_1,r) \in \overrightarrow{\mathfrak{m}}.\mathrm{D}_r\}\end{aligned}$$

Pour la même raison et par définition des graphes sémantiquement vides, si $i_1 \in \mathfrak{km}.\mathrm{V}$, *alors* $g(i_1) \subseteq \{i_1\} \subseteq \mathfrak{km}.\mathrm{V}$ *et si* $i_2 \in \mathfrak{km}.\mathrm{V}$, *alors* $g(i_2) \subseteq \{i_2\}\mathfrak{km}.\mathrm{V}$. *Et ainsi,* $g(i_1) \setminus \mathfrak{km}.\mathrm{V} = \emptyset$ *et* $g(i_2) \setminus \mathfrak{km}.\mathrm{V} = \emptyset$. *D'où :*

$$\begin{aligned}\mathfrak{gm}.\mathrm{E}_r &= \mathfrak{m}_\mathsf{k}.\mathrm{E}_r \cup \{(i_1,r,i_2) \in \mathfrak{m}_\mathsf{k}.\mathrm{V} \times \mathcal{R} \times \mathfrak{m}_\mathsf{k}.\mathrm{V} \mid \\ &\qquad\qquad \exists i'_1 \in \emptyset, \cdots \vee \exists i'_2 \in \emptyset, \cdots \vee \exists (i'_1,i'_2) \in \emptyset^2, \ldots\} \\ &= \mathfrak{m}_\mathsf{k}.\mathrm{E}_r \cup \emptyset = \mathfrak{m}_\mathsf{k}.\mathrm{E}_r\end{aligned}$$

$$\begin{aligned}\mathfrak{gm}.\mathrm{E}_r &= \mathfrak{km}.\mathrm{E}_r \cup \{(i_1,r,i_2) \in \mathfrak{km}.\mathrm{V} \times \mathcal{R} \times \mathfrak{km}.\mathrm{V} \mid \\ &\qquad\qquad \exists i'_1 \in \emptyset, \cdots \vee \exists i'_2 \in \emptyset, \cdots \vee \exists (i'_1,i'_2) \in \emptyset^2, \ldots\} \\ &= \mathfrak{km}.\mathrm{E}_r \cup \emptyset = \mathfrak{km}.\mathrm{E}_r\end{aligned}$$

Les trois composants de $\mathfrak{gm}$ *et de* $\mathfrak{km}$ *coïncident et donc on a bien :* $\mathfrak{gm} = \mathfrak{km}$. □

La recontextualisation n'annule pas la réécriture

Comme mentionné dans le chapitre 3.6.4, l'action d'un outil de réécriture Tool est décrite par les différences entre un modèle d'entrée donné et le modèle de sortie correspondant. Cette spécification est formalisée dans la définition 3.4 par les ensembles d'*éléments supprimés* :
$\mathsf{Tool}^i_\mathrm{D}(\mathfrak{m}), \mathsf{Tool}^a_\mathrm{D}(\mathfrak{m}), \mathsf{Tool}^r_\mathrm{D}(\mathfrak{m})$ et d'*éléments ajoutés* $\mathsf{Tool}^i_\mathrm{A}(\mathfrak{m}), \mathsf{Tool}^a_\mathrm{A}(\mathfrak{m}), \mathsf{Tool}^r_\mathrm{A}(\mathfrak{m})$.

La recontextualisation a pour but de replacer le résultat de Tool dans un contexte initial sans défaire l'action de Tool. Plus précisément, les éléments respectivement *ajoutés* et *supprimés* par l'outil ne doivent pas être respectivement *supprimés* et *récupérés* par la recontextualisation.

Propriété 2 *Soit* $\mathfrak{m}$, *un modèle donné et* Tool, *un outil de réécriture. Soit* $\mathfrak{gm}$, *le modèle migré inversé et recontextualisé par graphe :*

$$\begin{aligned}\mathfrak{gm} = \mathsf{GraphRecontext}\ \big(\ &\mathsf{KeysRecontext}(\mathsf{BackwardMigrate}(\mathsf{Tool}(\mathsf{Migrate}(\overrightarrow{\mathfrak{m}})))\ , \\ &\mathsf{GraphRecontext}(\mathsf{Tool}, \mathsf{Migrate}(\overrightarrow{\mathfrak{m}}))\ \big)\end{aligned}$$

On a :

(1) les éléments supprimés ne sont pas récupérés :
$\text{Tool}^{i}_{\text{D}}(\text{Migrate}(\overrightarrow{\mathfrak{m}})) \cap \mathfrak{gm}.\text{V} = \text{Tool}^{a}_{\text{D}}(\text{Migrate}(\overrightarrow{\mathfrak{m}})) \cap \mathfrak{gm}.\text{E}_a = \text{Tool}^{r}_{\text{D}}(\text{Migrate}(\overrightarrow{\mathfrak{m}})) \cap \mathfrak{gm}.\text{E}_r = \emptyset$
(2) les éléments ajoutés ne sont pas supprimés :
$\text{Tool}^{i}_{\text{A}}(\text{Migrate}(\overrightarrow{\mathfrak{m}})) \subseteq \mathfrak{gm}.\text{V} \wedge \text{Tool}^{a}_{\text{A}}(\text{Migrate}(\overrightarrow{\mathfrak{m}})) \subseteq \mathfrak{gm}.\text{E}_a \wedge \text{Tool}^{r}_{\text{A}}(\text{Migrate}(\overrightarrow{\mathfrak{m}})) \subseteq \mathfrak{gm}.\text{E}_r$

Principes de preuve *Le second point est trivial puisque ni* KeysRecontext *ni* GraphRecontext *ne supprime pas des éléments du modèle* $\text{Tool}(\text{Migrate}(\overrightarrow{\mathfrak{m}}))$ *résultant de* Tool. *Concernant le premier point, on considère les éléments ajoutés par* KeysRecontext, *puis par* GraphRecontext. *Par définition de* KeysRecontext *et de* $\text{Tool}^{i}_{\text{D}}$, *sachant que* BackwardMigrate *est l'*identité *et que* $\text{Migrate}(\overrightarrow{\mathfrak{m}}).\text{V} = \mathfrak{m}.\text{V} \setminus \overrightarrow{\mathfrak{m}}.\text{D}_i$ *on a :*

$$\begin{aligned}\mathfrak{gm}.\text{V} &= \text{KeysRecontext}(\text{BackwardMigrate}(\text{Tool}(\text{Migrate}(\overrightarrow{\mathfrak{m}})))).\text{V} \\ &= \text{BackwardMigrate}(\text{Tool}(\text{Migrate}(\overrightarrow{\mathfrak{m}})).\text{V} \cup \overrightarrow{\mathfrak{m}}.\text{D}_i \\ &= \text{Tool}(\text{Migrate}(\overrightarrow{\mathfrak{m}})).\text{V} \cup \overrightarrow{\mathfrak{m}}.\text{D}_i\end{aligned}$$

$$\begin{aligned}\text{Tool}^{i}_{\text{D}}(\text{Migrate}(\overrightarrow{\mathfrak{m}})) &= \text{Migrate}(\overrightarrow{\mathfrak{m}}).\text{V} \setminus \text{Tool}(\text{Migrate}(\overrightarrow{\mathfrak{m}})).\text{V} \\ &= (\mathfrak{m}.\text{V} \setminus \overrightarrow{\mathfrak{m}}.\text{D}_i) \setminus \text{Tool}(\text{Migrate}(\overrightarrow{\mathfrak{m}})).\text{V}\end{aligned}$$

Et donc on a :

$$\begin{aligned}&\text{Tool}^{i}_{\text{D}}(\text{Migrate}(\overrightarrow{\mathfrak{m}})) \cap \mathfrak{gm}. \\ &= \left((\mathfrak{m}.\text{V} \setminus \overrightarrow{\mathfrak{m}}.\text{D}_i) \setminus \text{Tool}(\text{Migrate}(\overrightarrow{\mathfrak{m}})).\text{V} \right) \cap \left(\text{Tool}(\text{Migrate}(\overrightarrow{\mathfrak{m}})).\text{V} \cup \overrightarrow{\mathfrak{m}}.\text{D}_i \right) \\ &= \left(\mathfrak{m}.\text{V} \setminus (\overrightarrow{\mathfrak{m}}.\text{D}_i \cup \text{Tool}(\text{Migrate}(\overrightarrow{\mathfrak{m}}))\text{V}) \right) \cap \left(\text{Tool}(\text{Migrate}(\overrightarrow{\mathfrak{m}})).\text{V} \cup \overrightarrow{\mathfrak{m}}.\text{D}_i \right) = \emptyset\end{aligned}$$

La preuve de $\text{Tool}^{a}_{\text{D}}(\text{Migrate}(\overrightarrow{\mathfrak{m}})) \cap \mathfrak{gm}.\text{E}_a = \emptyset$ *et de* $\text{Tool}^{r}_{\text{D}}(\text{Migrate}(\overrightarrow{\mathfrak{m}})) \cap \mathfrak{gm}.\text{E}_r = \emptyset$ *suivent les mêmes principes. Par définition de* $\text{Migrate}(\overrightarrow{\mathfrak{m}}).\text{E}_a$ *et de* $\text{Migrate}(\overrightarrow{\mathfrak{m}}).\text{E}_a$, Tool *ne prend en entrée aucun arc (attribut ou référence) supprimé (*$\overrightarrow{\mathfrak{m}}.\text{D}_a$ *ou* $\overrightarrow{\mathfrak{m}}.\text{D}_r$*), ou dont la source et/ou la cible est supprimée (*$\overrightarrow{\mathfrak{m}}.\text{D}_i$*).* Tool *ne peut pas supprimer des éléments déjà supprimés. Or, par définition de* KeysRecontext *et de* GraphRecontext *(c.f. définition 3.5 et définition 3.7 les seuls arcs ajoutés sont pris parmi* $\overrightarrow{\mathfrak{m}}.\text{D}_a$ *et* $\overrightarrow{\mathfrak{m}}.\text{D}_r$*, ou ils sont liés à des instances supprimées extraites de* $\overrightarrow{\mathfrak{m}}.\text{D}_i$. □

3.7 Conclusion

L'approche proposée dans cette thèse pour réutiliser du code repose sur la génération automatique de spécification de migration de modèles à partir de transformations qui s'appliquent au niveau méta-modèle. La migration retour est elle-même générée automatiquement et conjointement à cette migration initiale. La spécification de migration obtenue par défaut est éditable et permet de prendre en compte des cas très spécifiques de migration. Et ce qui est plus important, la migration retour prend en compte l'action de l'outil dans le domaine cible afin de garantir qu'elle même ne défait pas l'action de l'outil. Notre approche répond aux critères présentés par le tableau 2.5 : génération de la migration, génération conjointe de la migration inverse, cohérence, adaptabilité et prise en compte de l'action de l'outil.

Le round-trip de migration de l'approche s'appuie sur des moteurs génériques et réutilisables dont les entrées sont des modèles et des spécifications. Ces spécifications décrivent le refactoring de méta-modèles, la migration des modèles et la description

des outils au moyen des graphes de dépendance qui relient les entrées et les sorties d'un outil. Les avantages de notre approche de réutilisation sont :

- *Développement rapide* de l'outillage pour un DSML. Grâce à que la plupart des modèles et des spécifications impliqués dans le round-trip sont générés automatiquement à l'automatisation des migrations réversibles de un DSML vers le domaine de définition de l'outil.
- *Réutilisation sans risque* grâce au *modèle de réutilisation* qui relie le DSML au domaine de définition de l'outil.
- *Simplification* du DSML dans la mesure où les préoccupation de calcul sont en dehors du côté outil. Toutes les données qui ne sont pas nécessaires à l'outil sont mises de côté pour alléger le méta-modèle. On ne se concentre que sur les données vraiment utiles et nécessaires.
- *Personnalisation facile* du modèle de réutilisation. Nous proposons une migration et une recontextualisation par défaut. La migration peut être spécialisée sur mesure pour produire des résultats différents selon les besoins spécifiques.

Troisième partie

Implémentation et validation

Chapitre 4

Mise en œuvre

Sommaire

4.1 Introduction

Dans cette thèse nous étudions la réutilisation de *code source*, plus particulièrement la réutilisation d'algorithmes développés dans un contexte orienté-objet (des méthodes de classes). Le travail réalisé nous permet de fournir à l'utilisateur un environnement complet de réutilisation. La majorité des étapes est automatisée. L'intervention de l'utilisateur n'est requise que lorsque des aspects sémantiques liés au domaine d'application de l'outil sont nécessaires pour achever une étape de la réutilisation, pour la raffiner ou pour la valider. Dans ce contexte, un prototype de *framework* de réutilisation a été developpé [1]. Il fournit notamment un ensemble d'assistants pour l'automatisation et la réutilisation de code Java.

Dans ce chapitre nous décrivons les fonctionnalités de ce prototype (section 4.2). La conception générale de l'architecture logicielle de ce prototype est présentée ensuite (section 4.3).

4.2 Scénario type d'utilisation

Dans cette section, nous décrivons l'usage typique du prototype du framework de réutilisation. Pour les besoins de cette description, nous supposons qu'un utilisateur dispose des éléments suivants : un méta-modèle initial décrivant le domaine ; un modèle source conforme sur lequel appliquer un outil ; et un outil écrit en Java, qui manipule de modèles conformes à un méta-modèle adapté à l'outil. Le besoin de notre utilisateur se résume alors à ceci : *appliquer l'outil au modèle source.*

La figure 4.1 présente la boîte de dialogue par l'intermédiaire de laquelle notre utilisateur va pouvoir indiquer les données nécessaires pour avoir une réponse à son besoin. Les carrés en ligne double représentent les données dont l'utilisateur dispose et qui doivent être indiquées avant de commencer le processus de réutilisation.

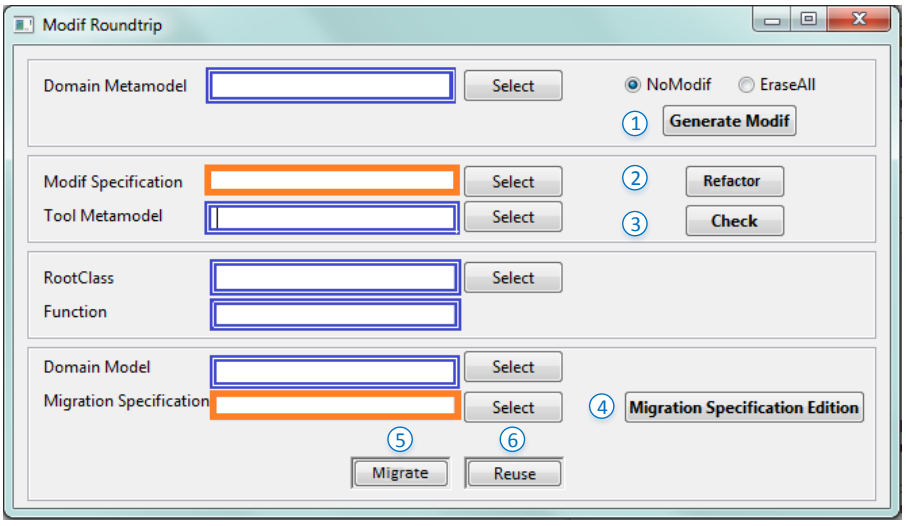


FIGURE 4.1 – Interface homme machine du framework de réutilisation

1. http://lab-sticc.univ-brest.fr/~babau/modif/modif.htm

Pour réutiliser la méthode, l'utilisateur doit entreprendre dans l'ordre les tâches listées ci-dessous :

1. Créer et éditer une spécification de refactoring (bouton Generate Modif dans la figure 4.1).
2. Effectuer la transformation du méta-modèle du domaine d'application (bouton Refactor).
3. Vérifier que le méta-modèle transformé coïncide avec le méta-modèle de l'outil (bouton Check).
4. Créer et éditer une spécification de migration (bouton Migration Specification Edition).
5. Lancer la migration du modèle source (bouton Migrate).
6. Réutiliser l'outil (bouton Reuse).

Les tâches et les utilitaires du framework qui mettent en œuvre l'enchainement de ces 6 étapes sont détaillés par la suite.

4.2.1 Création et édition d'une spécification de refactoring

Dans un premier temps, l'utilisateur doit pouvoir exprimer les transformations à appliquer à son méta-modèle initial pour le faire converger vers le méta-modèle de l'outil. Pour rendre cette tâche plus facile, nous proposons dans notre framework un *langage textuel* dûment documenté à base d'opérateurs de co-évolution. Ce langage fournit notamment un ensemble d'opérateurs pour *spécifier le refactoring* qui s'applique au méta-modèle source. Une spécification de refactoring est divisée en deux parties, l'en-tête et les transformations. L'en-tête indique des modifications sur le package principal du méta-modèle initial. L'autre partie indique les opérateurs à appliquer. Les opérateurs disponibles sont rename, remove, hide, flatten, flattenAll, changeAbstract, changeContainment, changeBounds.

À ce niveau, on a constaté deux besoins fréquents. Soit l'utilisateur conserve presque touts les concepts, soit il n'a besoin que de peu de concepts. Nous fournissons donc à l'utilisateur deux *templates* qui créent des spécifications de refactoring par défaut. Si le besoin est de garder la majorité de concepts, le template *NoModif* est proposé. L'application de ce template produit une spécification pour une copie du méta-modèle. L'utilisateur peut ensuite éditer cette spécification pour transformer et supprimer des concepts selon le besoin. Dans l'autre situation, *EraseAll*, permet de générer une spécification de refactoring qui supprime tous les concepts d'un méta-modèle. L'application de ce template produit une spécification pour obtenir un méta-modèle vide. L'utilisateur l'édite ensuite pour conserver et transformer les concepts dont il a besoin.

À titre d'exemple, la figure 4.2 présente une spécification de refactoring *NoModif* et une spécification de refactoring *EraseAll* produites pour le méta-modèle de la figure 3.2 (chapitre 3). Une fois la spécification de refactoring éditée, l'utilisateur possède le méta-modèle initial et la spécification de refactoring associée à ce méta-modèle.

NoModifABCD.... abcd2.ecore modifServic...

```
root abcd to abcd2
Prefix abcd to abcd2
URI "http://abcd" to "http://abcd"

class {
    A to A {
        ref r1 to r1 bounds (0,-1) to (0,-1)
    } ;
    B to B {
        ref r3 to r3 bounds (0,-1) to (0,-1)
        ref r2 to r2 bounds (0,1) to (0,1)
    } ;
    C to C ;
    D to D {
        att nb to nb bounds (0,1) to (0,1)
        att label to label bounds (0,1) to (0,1)
    }
}
```

EraseAllABCD.modif

```
root abcd to abcd2
Prefix abcd to abcd2
URI "http://abcd" to "http://abcd"

class {
    remove A to A {
        remove ref r1 to r1 bounds (0,-1) to (0,-1)
    } ;
    remove B to B {
        remove ref r3 to r3 bounds (0,-1) to (0,-1)
        remove ref r2 to r2 bounds (0,1) to (0,1)
    } ;
    remove C to C ;
    remove D to D {
        remove att nb to nb bounds (0,1) to (0,1)
        remove att label to label bounds (0,1) to (0,1)
    }
}
```

FIGURE 4.2 – Spécifications de refactoring *NoModif* et *EraseAll*

4.2.2 Création et vérification de méta-modèle cible

Une fois transformé, le méta-modèle obtenu doit être comparé au méta-modèle cible afin de vérifier qu'ils coïncident effectivement. Pour faire la comparaison, l'utilisateur dispose du moteur de comparaison du framework de réutilisation. Ce moteur détecte et indique l'existence des différences entre les deux méta-modèles. Si les deux méta-modèles ne coïncident pas, alors il n'est pas permis de passer à la tâche suivante et la spécification de refactoring doit être modifiée et appliquée à nouveau.

Une réduction implique l'application d'une spécification de refactoring (illustrée par la figure 4.3). Cependant, l'utilisateur peut avoir à effectuer des transformations en cascade pour aboutir au méta-modèle cible de l'outil. Ceci implique l'application de plusieurs spécifications de refactoring, l'une après l'autre. La réduction en cascade est un processus de réduction incrémentale (figure 4.4). L'utilisateur peut par exemple, appliquer une spécification *EraseAll* pour avoir un méta-modèle réduit et qui ne contient que des concepts nécessaires dans le domaine de l'outil, même s'ils sont structurés différemment. L'utilisateur peut alors ensuite appliquer une spécification *NoModif* pour transformer les éléments du méta-modèle réduit. L'utilisateur peut appliquer des spécifications de refactoring autant de fois que nécessaire.

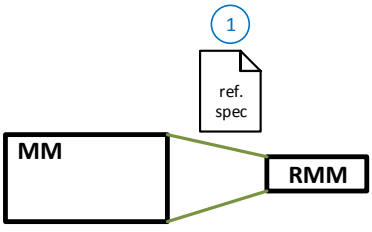


FIGURE 4.3 – Réduction classique

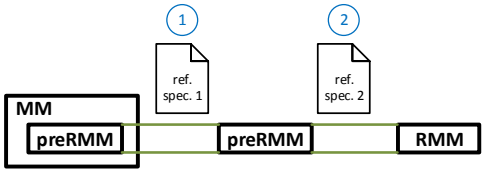


FIGURE 4.4 – Réduction en cascade

4.2.3 Création et édition de spécification de migration

À ce stade, l'utilisateur dispose du méta-modèle initial, du méta-modèle de l'outil, d'un modèle et d'une spécification de refactoring. Maintenant, il peut faire migrer son modèle, afin de le rendre conforme au méta-modèle de l'outil.

Cette migration s'effectue en deux temps ; d'abord, la spécification de refactoring est utilisée pour générer automatiquement une *spécification de migration* qui s'applique au modèle à migrer. Cette spécification est concrètement un modèle conforme au méta-modèle *Migration.ecore*. Dans un deuxième temps, la spécification de migration est utilisée pour produire le modèle migré. Cette migration en deux temps permet à l'utilisateur d'adapter, si nécessaire la spécification de migration selon le modèle considéré.

Lorsque l'utilisateur souhaite personnaliser la spécification de migration, un éditeur est proposé. L'éditeur de spécification de migration (figure 4.5) présente les instances du modèle séparées en deux ensembles. Le premier (Delete) contient les instances qui doivent être supprimées. Le deuxième (Keep) contient les instances qui doivent être conservées. Les boutons To Keep et To Delete permettent le passage d'une instance d'un ensemble à l'autre. Parmi les instances à supprimer, il y a des instances qui sont obligatoirement supprimées parce que la classe correspondante a été supprimée (application de l'opérateur remove sur la classe concernée). Ces instances ne peuvent donc pas être placées dans l'ensemble Keep.

À chaque fois qu'une modification est faite sur la spécification de migration, une vérification de conformité avec le méta-modèle de l'outil est effectuée. Cette vérification empêche la production de modèles non conformes au méta-modèle de l'outil, typiquement cette vérification permet de contrôler les contraintes de cardinalité.

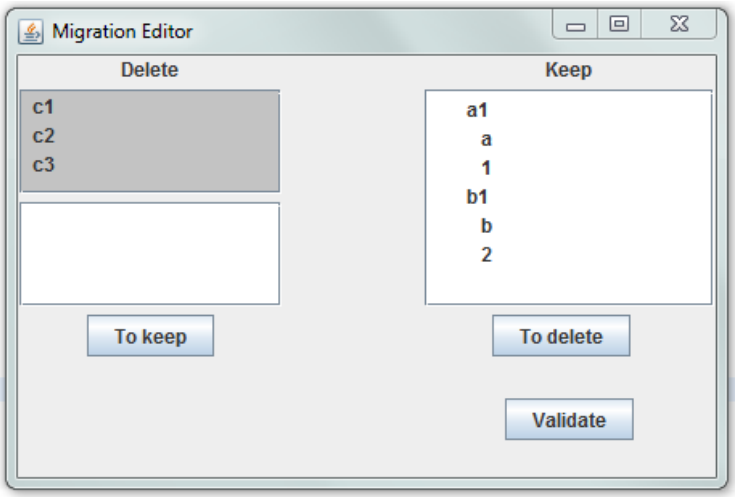


FIGURE 4.5 – Éditeur de la spécification de migration

Arrivé à ce stade, l'utilisateur dispose des méta-modèles (initial et de l'outil), d'une spécification de migration et du modèle à migrer. Il peut donc lancer l'exécution de la migration. La migration est gérée par un moteur de migration générique et réutilisable. L'utilisateur peut migrer plusieurs modèles (un par un) ou migrer le même modèle plusieurs fois en fonction de spécifications de migrations différentes.

4.2.4 Migration, réutilisation de code et recontextualisation

Toutes les actions précédentes ont permis à l'utilisateur d'atteindre son principal objectif : réutiliser du code avec des données qui n'étaient pas initialement utilisables pour ce code. Concrètement, le code à réutiliser dans notre prototype correspond à une méthode d'une certaine classe Java.

La migration inverse est un processus géré par le moteur de migration inverse et qui ne nécessite pas de l'intervention de l'utilisateur. Il garantit un résultat syntaxiquement correct, mais dont la sémantique peut être différente de ce que l'utilisateur attend. Les éléments qui ne peuvent pas être rajoutés au modèle (parce que leur ajout concerne des spécificités de la sémantique du domaine et nécessite de l'expertise métier de l'utilisateur) sont placés dans un fichier. L'utilisateur dispose alors des moyens pour spécialiser la migration inverse en fonction de ses besoins. Il peut agir directement sur le graphe de dépendances pour indiquer des dépendances additionnelles par rapport à celles calculées automatiquement à partir des actions de l'outil. L'édition du graphe est faite par la modification directe du modèle de graphe de dépendances en utilisant les facilités d'édition d'EMF. Le graphe de dépendances est concrètement un modèle conforme au méta-modèle *Dependency.ecore* (figure 4.8). Dans cette mise en œuvre, nous considérons que l'outil est une boîte grise qui fournit des méta-informations concernant les fonctionnalités qu'il offre. Nous utilisons ces informations pour construire le graphe de dépendances.

La section suivante présente les choix qui concernent l'architecture logicielle du framework de réutilisation.

4.3 Choix de conception

Le framework de réutilisation est conçu afin de traiter des données qui appartiennent à deux niveau de modélisation, niveau méta-modèle et niveau modèle. Par la suite, les choix de conception qui permettent de manipuler les données à ces deux niveaux sont expliqués.

4.3.1 Niveau méta-modèle

Afin d'expérimenter les idées présentées dans cette thèse, nous proposons une mise en œuvre dans l'environnement EMF basé sur Ecore. Le framework de réutilisation est fait en Java et utilise les facilités d'EMF pour charger, enregistrer, éditer et parcourir les méta-modèles et les modèles. Nous expliquons par la suite les traitements et les technologies utilisées pour gérer les données de niveau méta-modèle.

Spécification de refactoring

Une spécification de refactoring est une instance du méta-modèle *Modif.ecore*. Ce méta-modèle est illustré par la figure 4.6. Ce méta-modèle n'a pas été conçu lors de la thèse, mais nous considérons qu'il est important de l'illustrer ici pour faciliter la compréhension du framework de réutilisation [BK11]. En effet, la génération de la spécification de migration et l'interprétation des opérateurs Modif pour l'inversion sont évidement fortement dépendants des opérateurs proposés par Modif.

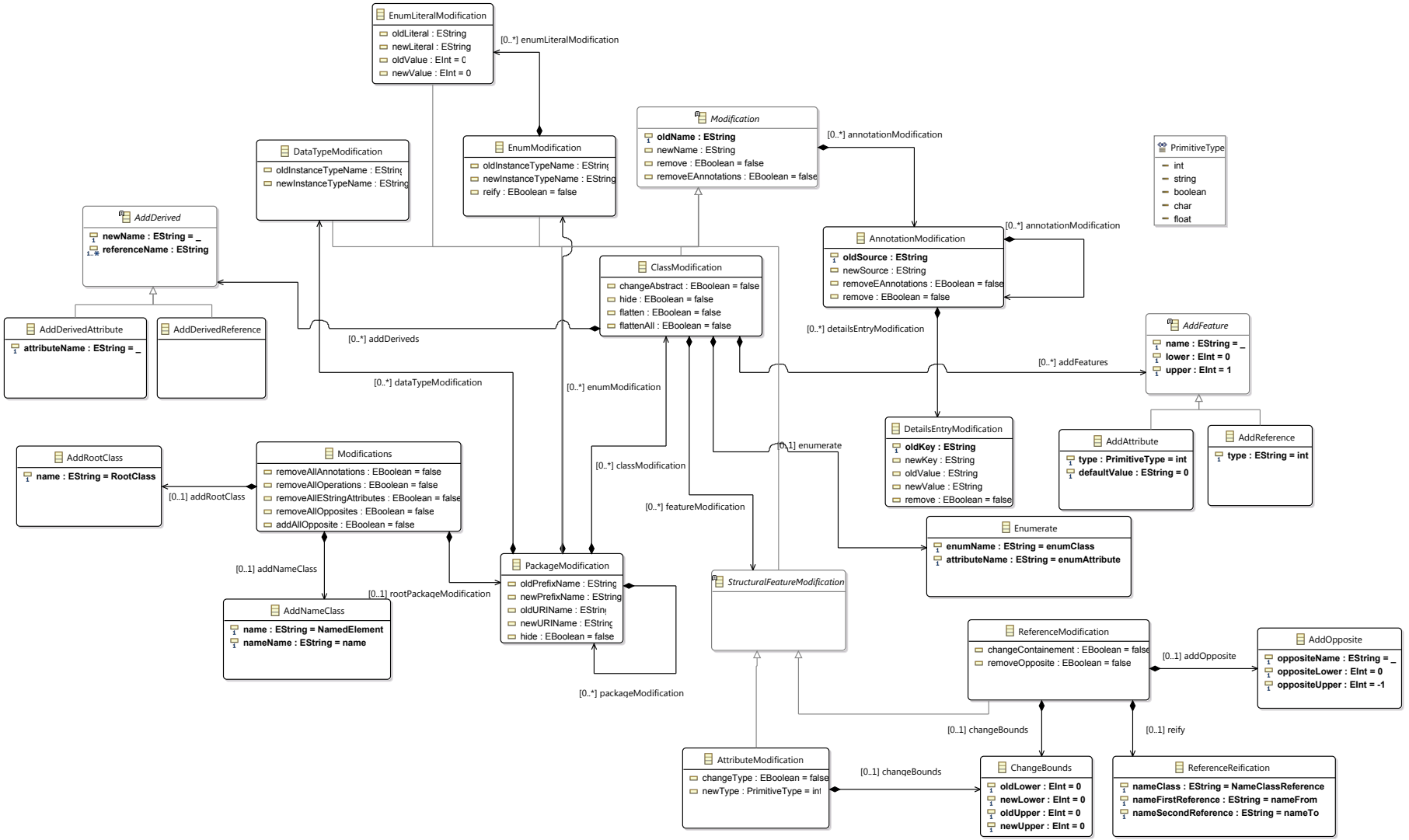


FIGURE 4.6 – Méta-modèle Modif.ecore

Pour la génération d'une spécification de refactoring, l'idée principale de Modif est d'associer à chaque concept d'Ecore du méta-modèle d'application certains opérateurs d'édition classiques (remove, to (rename), etc.) pour obtenir un méta-modèle transformé. Par exemple, le concept Modif de *ClassModification* est un calque du concept Ecore d'*EClass*. Des macro-opérateurs sont également fournis (hide, flatten, flattenAll). Ils réduisent la quantité d'opérateurs à utiliser. Ces macro-opérateurs sont implémentés au travers des séquences d'appels aux opérateurs simples. Par exemple, l'opérateur hide fait appel à remove des attributs de la classe, remove des références de la classe, addDerived et remove classe.

Les opérateurs de Modif sont appliqués dans l'ordre qui suit : to (rename), ChangeAbstract, flatten, flattenAll, hide, remove. Cette séquence nous donne la sémantique opérationnelle de Modif. En raison de cette sémantique opérationnelle, il peut être nécessaire d'appliquer les opérateurs nombreuses fois pour obtenir le méta-modèle ciblé. La syntaxe concrète de la spécification de refactoring a été construite à l'aide de Xtext [2].

Refactoring

En plus du méta-modèle *Modif.ecore*, nous utilisons le méta-modèle *EcoreModif.ecore*. Celui-ci est une extension de *Modif.ecore* avec certains raccourcis et des références dérivées pour permettre de faire un parcours de méta-modèles plus efficace.

La transformation d'un méta-modèle avec Modif est gérée par un moteur de transformation codé en Java qui suit les principes du patron de conception *visiteur*. L'utilisation de ce patron facilite l'ajout de nouveaux opérateurs sans avoir à modifier les opérateurs existants.

Identifiants

Afin de garantir le traçage des instances d'un modèle lors du round-trip de migration, un attribut *UUID* est utilisé. Concrètement, si le méta-modèle initial n'as pas un attribut *UUID* de type *EString*, le moteur de refactoring crée une copie exacte du méta-modèle initial et dans cette copie, toutes les classes possèdent directement ou indirectement (via une super classe) un attribut *UUID* de plus. Pour raisons de simplicité dans les exemples présentés dans ce document, les identifiants sont constitués d'une lettre et un numéro, mais dans l'implémentation, ces identifiants sont générés par la méthode *generateUUID* de la classe *EcoreUtil*. " *_Fj9QPC0XEd-zxLXA4jfyXA* " est un exemple d'un identifiant. À aucun moment, ces identifiants n'altèrent le fonctionnement de la méthode à réutiliser. À la fin du round-trip, les identifiants sont enlevés et le modèle résultant est rendu conforme au méta-modèle initial.

Comparaison

La comparaison du méta-modèle transformé avec le méta-modèle de l'outil est faite grâce à l'utilisation des bibliothèques de EMF (*EMFCompare*) pour comparer des

2. http://www.eclipse.org/Xtext/

méta-modèles. Notamment, elle utilise la méthode *getDifferences*. Le moteur de comparaison fait un appel à cette bibliothèque, et donne le résultat de la comparaison entre les deux méta-modèles.

4.3.2 Niveau modèle

Dans son état actuel, le prototype prend en charge, au niveau modèle, les opérateurs rename, remove, flatten et hide de Modif.

Les moteurs de migration, réutilisation, et migration inverse son instrumentés en Java. Ils utilisent l'ensemble de classes Java produites par EMF et manipulent les modèles grâce au framework de persistance d'EMF (XMI). Dans ces moteurs génériques, le code de contrôle est applicable à n'importe quel modèle. Ils implémentent les algorithmes présentés dans la section 3.6 du chapitre 3. Ils s'appuient sur l'API de réflexivité d'Ecore. Ainsi ils ne sont pas dépendants d'un quelconque méta-modèle source concret.

Spécification de migration

Une spécification de migration est un instance du méta-modèle *Migration.ecore*. Les concepts principaux du méta-modèle ont été illustrés par la figure 3.6. Le méta-modèle complet est lui-même représenté dans un modèle Ecore grâce à EMF. La figure 4.7 présente le méta-modèle de spécification de migration tel que vu depuis l'éditeur graphique intégré d'Ecore. Ce méta-modèle permet d'exprimer les modifications à appliquer sur les instances d'un modèle. Ces modifications correspondent à des opérations basiques sur le graphe.

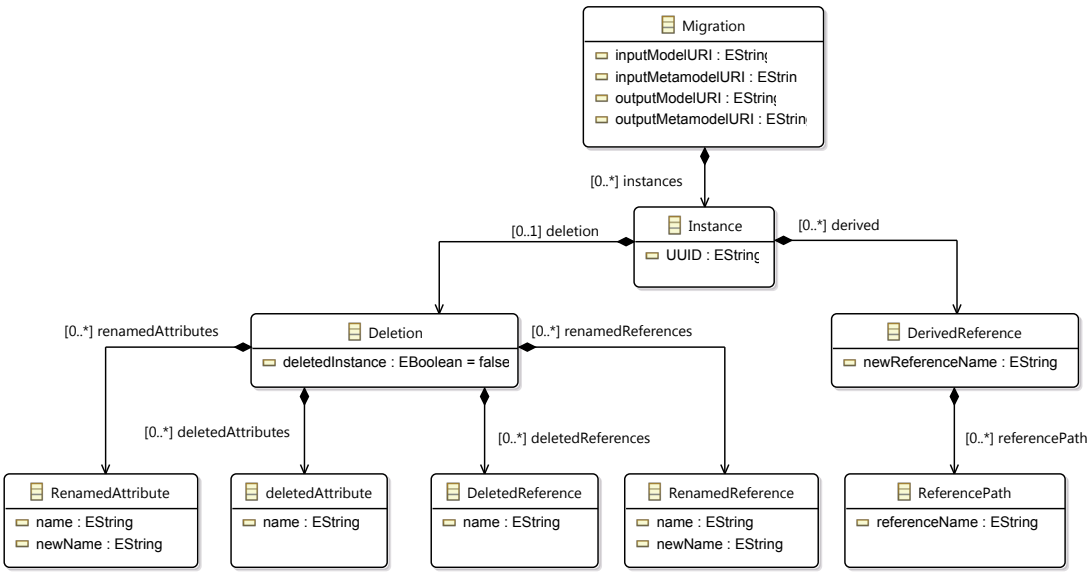


FIGURE 4.7 – Méta-modèle Ecore pour la représentation de spécifications de migration

Pour chaque instance du modèle, il est possible d'exprimer la suppression d'instances, des attributs ou de références. Il est possible également de définir l'ajout des

références dérivées à partir des références existantes. Cet ajout est fait lorsque l'opérateur hide est appliqué au niveau méta-modèle.

Pour l'instant, le prototype ne propose pas de syntaxe concrète pour les spécifications de migration et l'édition de la spécification de migration se fait par la modification directe du modèle de migration ou en utilisant l'interface d'édition (figure 4.5).

Pour assurer le lien avec le méta-modèle, la génération de la spécification de migration est faite en Java à partir de la spécification de refactoring (Modif).

Lors des modifications, une validation du modèle est faite avec *EMF Validation*.

Spécification de graphe de dépendances

Une spécification de graphe de dépendances est un modèle conforme au méta-modèle *Dependency.ecore*. Ce méta-modèle est illustré par la figure 4.8. Il permet de définir les dépendances entre les instances d'entrée et de sortie d'un outil. Un Vertex correspond à un sommet du graphe de modèle (présenté dans la section 3.3.2).

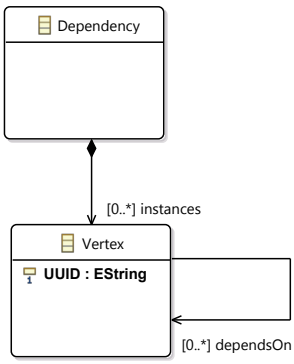


FIGURE 4.8 – Graphe de dépendances

Dans cette mise en œuvre, la production du graphe de dépendances est obtenu par la modification directe des outils. Nous enregistrons l'identifiant de chaque concept de la sortie de l'outil et l'ensemble des identifiants des concepts de l'entrée qui participent à sa création ou modification.

Round-trip de migration

À ce niveau, une copie du modèle qui doit être migré est faite. La copie est conforme au méta-modèle du domaine étendu avec l'attribut *UUID*. Chaque instance de cette copie a une valeur unique pour cet attribut.

À aucun moment, les identifiants n'altèrent le fonctionnement de la méthode à réutiliser. À la fin du round-trip, les identifiants sont enlevés et le modèle résultant est rendu conforme au méta-modèle du domaine.

Toutes les étapes du round-trip de migration sont codées dans la classe Java *MigrationRoundtrip*. Cette classe permet de faire le round-trip de migration étape par étape à partir d'un modèle d'entrée. Les méthodes principales de cette classe sont : *onwardMigration*, *reverseMigration*, *recontextualizationKey* et *recontextualizationGraph*. Elles correspondent aux étapes du round-trip de migration. Cette même classe dispose de

méthodes utilisatires dédiées pour charger et enregistrer les modèles : *loadModel*, *serializeMigratedModel* et *saveModel*.

onwardMigration effectue la première étape (migration aller) du round-trip, elle permet la migration du modèle initial vers un nouveau modèle conforme au méta-modèle cible. Après avoir effectué cette étape, le modèle migré est sérialisé (*serializeMigratedModel*). *reverseMigration* réalise la deuxième étape du round-trip de migration (migration retour), elle permet la migration d'un modèle transformé par un outil vers un modèle conforme au méta-modèle initial. Cette action ne peut être effectuée avant la migration aller. *recontextualizationKey* effectue la troisième étape du round-trip (dite recontextualisation par clés), elle permet de remettre le contexte original sur le modèle migré inverse. Cette action ne peut être effectuée avant la migration retour. *recontextualizationGraph* réalise la quatrième et dernière étape du round-trip (recontextualisation par graphe), elle permet une recontextualisation plus précise grâce au graphe de dépendances. Cette action ne peut être effectuée avant la recontextualisation par graphe. La personnalisation du résultat peut être atteinte par l'édition du graphe de dépendances ou par l'édition du code Java du moteur de migration inverse.

4.4 Conclusion

Dans ce chapitre, nous avons présenté le besoin d'un utilisateur. Et nous avons présenté les utilitaires du framework de réutilisation qui répondent à ces besoins.

L'utilisateur est guidé et son intervention dans le processus de réutilisation est minimale. Il n'a besoin d'agir que lorsque son expertise métier est requise.

Les spécifications générées par le framework sont éditables et facilitent la production de résultats différents à partir du même méta-modèle ou modèle.

Modif est le langage de co-évolution que nous avons choisis pour illustrer les principes de notre approche. Le méta-modèle de migration est fait pour faire les migrations correspondantes aux opérateurs proposés par Modif.

Chapitre 5

Évaluation

Sommaire

5.1 Introduction de chapitre

L'approche proposée dans cette thèse qui vise à favoriser la réutilisation de code a été expérimentée sur deux cas d'étude. Les deux cas permettent de mettre en évidence l'intérêt de cette approche grâce à de critères comme le nombre de lignes de code réutilisées.

Le premier cas porte sur la réutilisation d'un outil de visualisation de cartes, dans le contexte des données géographiques d'une ville. Ce cas met en évidence l'intérêt de pouvoir éditer une spécification de migration générée à partir d'une spécification de refactoring de méta-modèles.

Le deuxième cas concerne la réutilisation de deux outils génériques (*HSFlattener* et *GScheduler*) pour un domaine donné. Ce cas met en évidence l'intérêt de la recontextualisation dans le but de réutiliser un outil de réécriture.

5.2 CityIs to MapViewer

5.2.1 Présentation du cas d'étude

Nous utilisons le framework pour réutiliser un outil *d'analyse* qui affiche des lieux précis sur la carte d'une ville. Cet outil existe et s'applique dans le cadre d'un méta-modèle dédié à l'outil. On souhaite le réutiliser dans le contexte d'un méta-modèle différent. Le méta-modèle d'origine contient les informations sur les citoyens d'une ville et leurs liens avec les bâtiments de la ville. Ce méta-modèle est d'abord présenté en détail. Nous présentons ensuite le méta-modèle du contexte dans lequel l'outil est réutilisé. Puis, nous présentons la génération et la personnalisation de la spécification de migration. Enfin, nous présentons la réutilisation de MapView.

5.2.2 MapView

MapView est l'outil que nous allons réutiliser. Il affiche une carte dans laquelle l'emplacement de certains bâtiments est signalé. L'outil MapView a été développé grâce à l'API de Google Maps [1]. Il permet d'obtenir une vue graphique des données géographiques (emplacement des bâtiments) comme illustré par la figure 5.1.

La couleur et l'étiquette du marqueur dépendent du type de bâtiment. Un marqueur bleu avec étiquette U indique que le bâtiment est une université. Un marqueur rouge avec étiquette H représente la présence d'un centre de santé. Un marquer marron avec l'étiquette D indique que le bâtiment est une résidence universitaire. Une école est représentée par un marqueur violet avec l'étiquette S. Une banque est représentée par un marqueur jaune avec l'étiquette B. Un marqueur vert avec l'étiquette M indique la présence d'un supermarché. Un marqueur orange avec l'étiquette L indique la présence d'une maison.

Un extrait du méta-modèle qui définit le format de données d'entrée de cet outil est illustré par la figure 5.2. Le méta-modèle introduit le concepts de City, Place, Address et Reference. Il y a différent types de bâtiments (*e.g.* University, Dormitory et HealthFacility). L'emplacement de chaque bâtiment est définit par une adresse spécifique.

1. https://developers.google.com/maps/

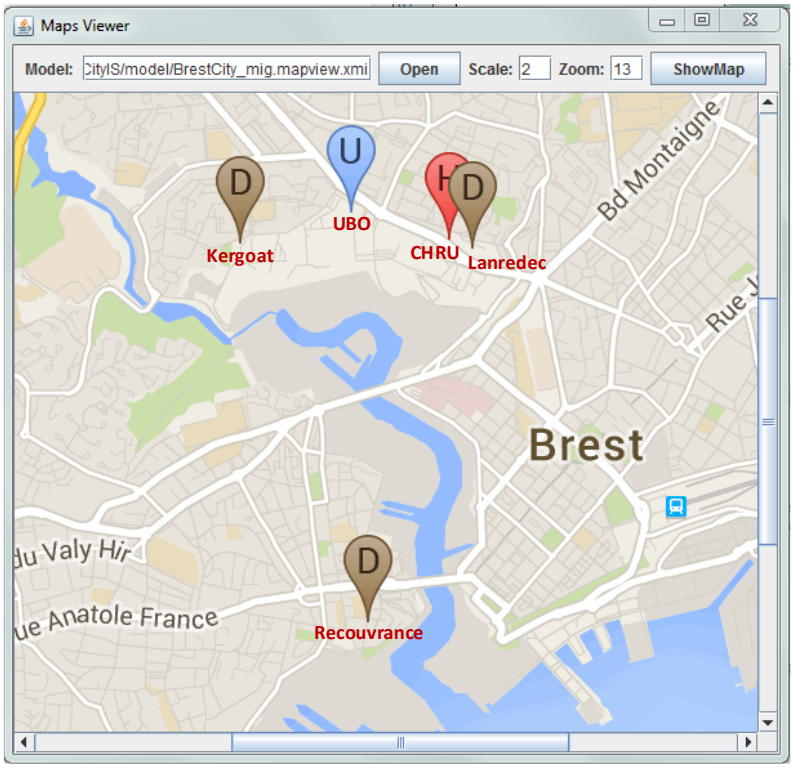


FIGURE 5.1 – Exemple d'utilisation de MapView

5.2.3 CityIS

Nous présentons maintenant le méta-modèle du contexte pour lequel cet outil est réutilisé. Il s'agit d'une variante du méta-modèle défini par l'outil MapView. La figure 5.3 présente cette variante. Il s'agit d'un extrait d'un Système d'information, lequel représente la population d'une ville (*City*). Il contient des informations concernant les citoyens (*Citizen*), leur emploi (référence *worksAt*), leur études (référence *studiesAt*) et leur logement (référence *livesIn*). Il contient les bâtiments (*Place*) de chaque quartier (*Neighborhood*) de la ville. Ces informations sont typiquement recueillies lors d'un recensement.

Par rapport à *MapView.ecore*, *CityIS.ecore* contient des éléments supplémentaires, notamment : la classe NamedElement, la classe Neighborhoods et la référence *places*, la classe Citizen, la référence *citizens*, la référence *neighhorhoods*, l'attribut *zipCode* et l'attribut *country*, les références *worksAt*, *studiesAt* et *livesIn*. Il y a beaucoup de ressemblances entre les deux méta-modèles, ils ont été construits de cette façon pour mettre en évidence l'utilité des spécifications de migration.

La figure 5.4 présente un extrait d'un modèle conforme à ce méta-modèle. Le modèle représente la ville de *Brest*. Deux de ces habitants sont : Anne et Robert. Deux de ces quartiers sont : Bellevue et Recouvrance. Dans Bellevue il y a une université (UBO), deux résidences universitaires (Kergoat et Lanredec) et un centre de santé (CHRU). Dans Recouvrance il y a une résidence universitaire (Recouvrance). Anne étude à l'UBO et habite à la résidence Kergoat. Robert travaille à l'UBO et habite à la résidence Recouvrance.

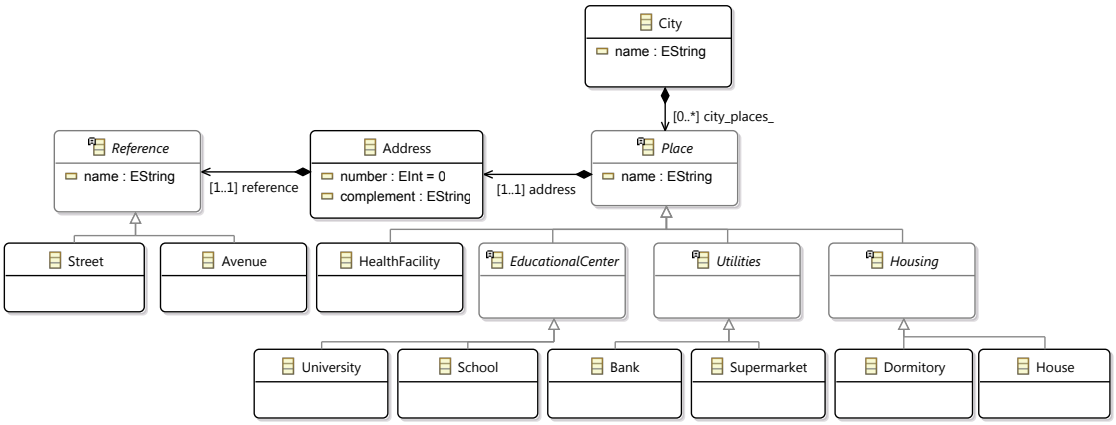


FIGURE 5.2 – Vue graphique du méta-modèle Mapview.ecore

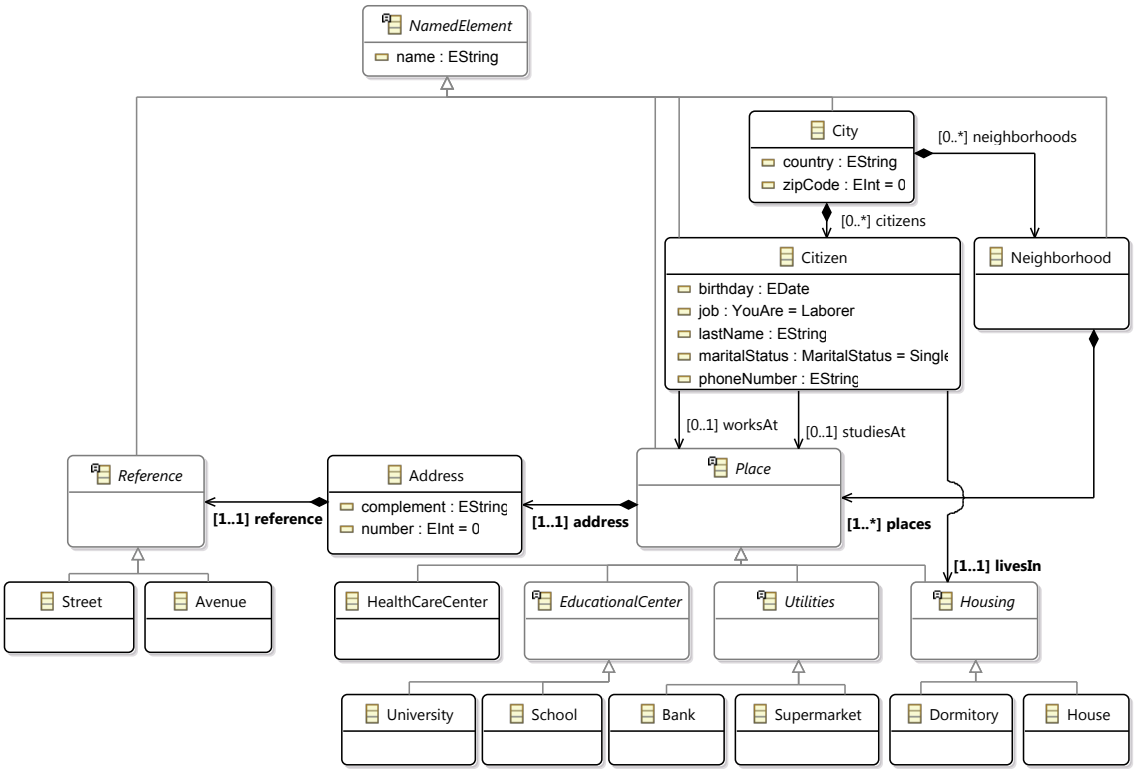


FIGURE 5.3 – Vue graphique du méta-modèle CityIS.ecore

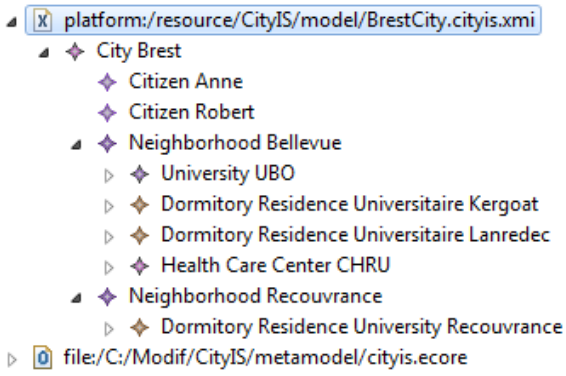


FIGURE 5.4 – Extrait du modèle de la ville de Brest

Ce modèle doit être migré pour être utilisé comme entrée de MapView.

5.2.4 Génération de la spécification de migration

Afin de placer les données du contexte CityIS dans le contexte de l'outil, La première étape de la réutilisation consiste à appliquer les opérateurs de co-évolution pour supprimer les éléments supplémentaires de *CityIS.ecore*. Notamment, les opérateurs utilisés sont les suivants : remove, rename, hide et flatten. La figure 5.5 illustre la spécification de refactoring Modif appliquée pour transformer *CityIS.ecore* en un méta-modèle qui coïncide avec *MapView.ecore*.

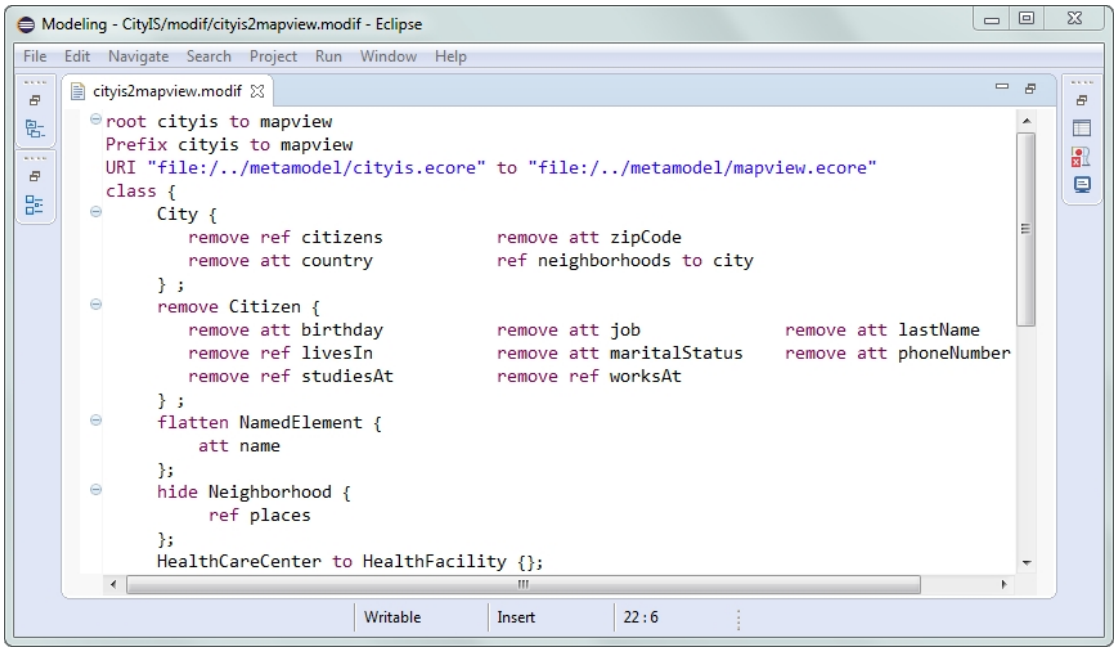


FIGURE 5.5 – Spécification de refactoring cityis2mapviewer.modif

Dans d'autres approches de co-évolution, les transformations du méta-modèle sont reflétées automatiquement dans les migrations (niveau modèle). Dans cette approche,

nous générons une spécification de migration à partir de la spécification de refactoring, l'application de telle spécification sur le modèle source, produit automatiquement un modèle migré. Mais lorsque les instances sont migrées automatiquement, la définition de migrations spécifiques n'est pas possible. La réutilisation est limitée dans son application.

Un exemple typique de réutilisation du même outil dédié avec des besoins spécifiques est la visualisation d'endroits spécifiques sur une carte paramétrée. Par exemple, on doit pouvoir afficher *tous* les bâtiments d'une ville, on doit pouvoir aussi afficher les bâtiments qui sont placés dans une *région spécifique*, on doit pouvoir faire un zoom avant pour n'afficher qu'un nombre limité de bâtiments, on doit pouvoir afficher les bâtiments qui partagent une caractéristique commune. Tous ces paramètres pourraient être ajoutés au méta-modèle de l'outil de visualisation, mais l'ajout polluerait le méta-modèle avec des informations liées à l'utilisation de l'outil.

On voit qu'il y a deux types de préoccupations, l'une liée au domaine de définition de l'outil et l'autre liée à l'utilisation de l'outil. Ces deux préoccupations doivent être séparées de manière que les spécificités d'utilisation soient définies indépendamment du méta-modèle de l'outil. Dans ce but, on doit pouvoir agir directement sur les modèles pour déterminer les éléments qui doivent être pris en compte par l'outil au moment de produire un résultat (*i.e.* affichage dans le cas de la visualisation de cartes).

Dans le contexte de la réutilisation et de la co-évolution, les migrations automatiques ne suffisent pas pour faire des migrations spécifiques. Afin de surmonter ce manque de personnalisation au niveau modèle, nous utilisons des spécifications de migrations éditables (section 3.3.6 du chapitre 3).

Pour notre étude de cas, la spécification de migration générée à partir des opérateurs de Modif appliqués lors du refactoring (figure 5.5) et du modèle initial (figure 5.4) est présentée par la figure 5.6. La colonne gauche contient les instances et les attributs qui doivent être supprimés obligatoirement afin de respecter les contraintes établies par le méta-modèle du MapView. La colonne droite contient les instances et les attributs qui sont conservés.

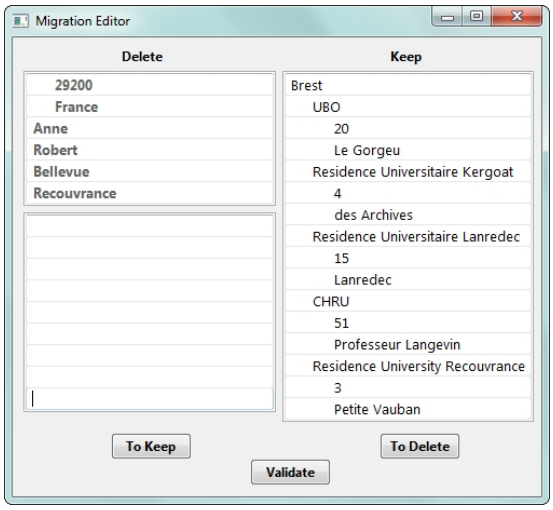


FIGURE 5.6 – Spécification de migration par défaut

La spécification de migration par défaut indique la suppression des instances Anne, Robert, Bellevue et Recouvrance ainsi que les attributs *29200* (*zipCode*) et France (*country*) de la ville de Brest. Toutes les autres instances sont conservées lors de la migration.

Dans la section suivante, nous présentons des cas spécifiques de visualisation des bâtiments en *MapView*. Ces cas simples montrent l'intérêt d'avoir des spécifications de migrations éditables.

5.2.5 Personnalisation de la spécification de migration

Dans notre cas d'étude, nous représentons l'édition des spécification de migration, par l'intermédiaire de trois exemples d'utilisation spécifique de *MapView* :

1. Visualiser seulement les résidences universitaires.
2. Visualiser l'université et les résidences universitaires qui sont a proximité.
3. Visualiser le centre de santé et la résidence universitaire la plus proche.

Dans le premier cas, la migration implique la suppression de toutes les instances de la classe *University* et de la classe *HealthCareCenter*. Dans le deuxième cas, les instances de la classe *HealthCareCenter* sont supprimées, quelques instances de *Dormitory* doivent être supprimées et les autres doivent être conservées. La façon de choisir les instances à conserver dépend du concept subjectif de *proximité* qui n'est pas automatisable et nécessite l'intervention de l'utilisateur au niveau modèle. Dans le troisième cas, seulement une instance de *HealthCareCenter* et une instance de *Dormitory* sont conservées. Dans ces trois cas, la classe correspondante aux instances est conservée au niveau méta-modèle, et la suppression ne concerne que certaines instances.

Solutions existantes. Comme nous l'avons déjà énoncé, la suppression des instances peut être atteinte en utilisant les opérateurs de co-évolution. Dans ce contexte, si une classe est supprimée au niveau méta-modèle, toutes ses instances sont supprimées au niveau modèle. Et si une classe est conservée, toutes les instances sont également conservées. Pour cet exemple, les opérateurs de co-évolution n'apportent pas une solution adéquate pour supprimer quelques instances lorsque que la classe est conservée dans le méta-modèle.

Certains outils comme Edapt [2], permettent de modifier le code de migration des instances manuellement, lorsque la migration est spécifique au modèle. Edapt ne génère le code de migration que pour les instances qui sont affectées par les transformations des classes. Dans aucun des trois exemples d'utilisation spécifique de *MapView*, les transformations du méta-modèle affectent les classes auxquels les bâtiments sont conformes. Le code de migration n'est pas généré, il est impossible de spécifier les migrations spécifiques. L'utilisation de langages de co-évolution qui génèrent du code de migration éditable, n'est pas une solution convenable parce que c'est impossible de définir le code de migration pour une instance dont la classe n'a pas été transformée.

Une autre façon de faire des migrations spécifiques est d'utiliser des langages classiques de transformation comme ATL [3]. Le listing 5.1 présente un extrait du code ATL qui conserve uniquement les résidences universitaires du modèle d'entrée. La règle

2. http://www.eclipse.org/edapt/
3. http://www.eclipse.org/atl/

Dormitory2Dormitory représentent la conservation de toutes les instances de *Dormitory*. Les autres règles présentent la suppression des instances de *University* et de *HealthCareCenter*.

Listing 5.1 – Code ATL pour conserver les instances de *Dormitory*

```
-- @path City=/.../metamodel/cityis.ecore
-- @path Mapview=/.../metamodel/mapview.ecore
module city2mapview;
create OUT : Mapview from IN : City;
rule Dormitory2Dormitory {
    from
        c : City!Dormitory
    to
        m : Mapview!Dormitory (
            name <- m.name ...)
}
rule deleteUniversity {
    from
        c : City!University
    to
        drop
rule deleteHealthCareCenter {
    from
        c : City!HealthCareCente
    to
        drop
```

Le listing 5.2 présente le code de migration ATL pour conserver l'université et des résidences qui sont à proximité. La règle *Dormitory2Dormitory* présente la condition (c.address.reference.name = 'Lanredec' || c.address.reference.name = 'Archives') à appliquer pour conserver certaines instances de *Dormitory*.

Listing 5.2 – Code ATL pour conserver quelques instances de *Dormitory*

```
-- @path City=/.../metamodel/cityis.ecore
-- @path Mapview=/.../metamodel/mapviewer.ecore
module city2mapview;
create OUT : Mapview from IN : City;
rule Dormitory2Dormitory {
    from
        c : City!Dormitory(
             c.address.reference.name = 'Lanredec' || c.address.refrence.
                name = 'Archives')
    to
        m : Mapview!Dormitory (
        name <- m.name ...)
}
rule University2University {
    from
        c : City!University(
        m : Mapview!University (
        name <- m.name ...)
}
rule deleteHealthCareCenter {
    from
        c : City!HealthCareCenter(
    to
        drop
}
```

Le listing 5.3 présente le code ATL pour conserver uniquement la résidence qui est à proximité du centre de santé. La règle *Dormitory2Dormitory* contient la condition (c.address.reference.name = 'Lanredec') pour conserver l'instance la plus proche du centre de santé.

Listing 5.3 – Code ATL pour conserver le centre de santé et la résidence la plus proche

```
-- @path City=/.../metamodel/cityis.ecore
-- @path Mapview=/.../metamodel/mapviewer.ecore
module city2mapview;
create OUT : Mapview from IN : City;
rule Dormitory2Dormitory {
    from
        c : City!Dormitory(
            c.address.reference.name = 'Lanredec' )
    to
        m : Mapview!Dormitory (
        name <- m.name ... )
}
rule deleteUniversity {
    from
        c : City!HUniversity(
    to
        drop
}
rule HealthCareCenter2HealthCareCenter {
    from
        c : City!HealthCareCenter(
        m : Mapview!HealthCareCenter (
        name <- m.name ...)
}
```

La différence entre les deux derniers listing est la comparaison de l'adresse. Il devient alors nécessaire de spécifier un code de migration pour chaque cas spécifique. La solution est alors dépendante du modèle, orientée au langage de transformations et non générique.

Édition de la spécification de migration générée Nous présentons maintenant, l'édition des spécifications de migration.

À partir de la spécification de migration que notre approche génère par défaut, l'utilisateur peut produire des spécifications de migration personnalisées.

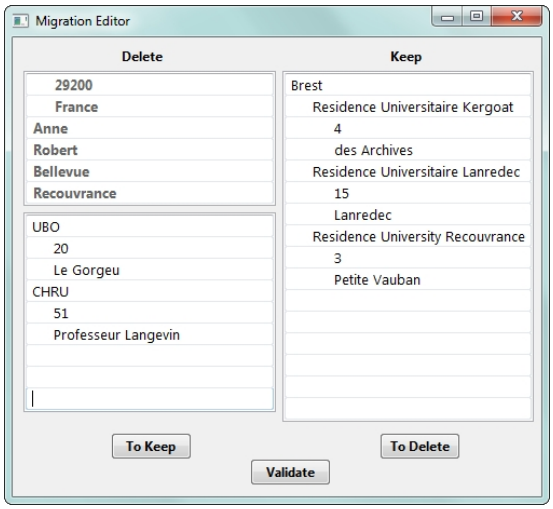


FIGURE 5.7 – Spécification de migration personnalisée 1

Pour le premier cas de spécification de migration personnalisée, il déplace l'UBO et le CHRU vers la colonne *Delete*. La spécification de migration personnalisée est pré-

sentée par la figure 5.7. Son application produit un modèle qui ne contient que les résidences universitaires (Lanredec, Kergoat et Recouvrance).

La spécification de migration personnalisée qui permet de conserver uniquement l'université et les résidences qui sont a proximité peut être produite de deux façons : modification de la spécification de migration par défaut ou modification de la spécification de migration personnalisé de la figure 5.7. Si cette spécification de migration est produite à partir de la spécification de migration par défaut, l'utilisateur déplace le CHRU et la résidence universitaire Recouvrance vers la colonne *Delete*, il conserve l'université UBO et les deux résidences qui sont a proximité (Lanredec et Kergoat). Si l'utilisateur décide de produire la spécification de migration à partir de la première spécification de migration personnalisée, il déplace l'UBO vers la colonne *Keep* et la résidence Recouvrance vers la colonne *Delete*. La spécification de migration est illustrée par la figure 5.8.

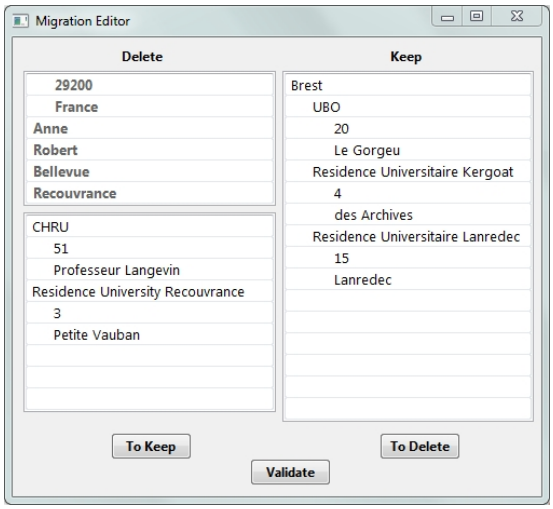


FIGURE 5.8 – Spécification de migration personnalisée 2

La dernière spécification de migration qui permet de conserver le centre de santé CHRU et la résidence universitaire la plus proche peut être produite à partir de la spécification de migration par défaut ou à partir des spécifications de migration personnalisées existantes. Pour cet exemple, elle est produite à partir de la spécification de migration par défaut, cela implique que l'utilisateur déplace vers la colonne *Delete* toutes les instances (UBO, Lanredec, Kergoat et Recouvrance) sauf la résidence universitaire Lanredec et le centre de santé CHRU. La spécification de migration personnalisée est présentée par la figure 5.9.

Une fois les spécifications de migration éditées et validées, nous pouvons réutiliser l'outil.

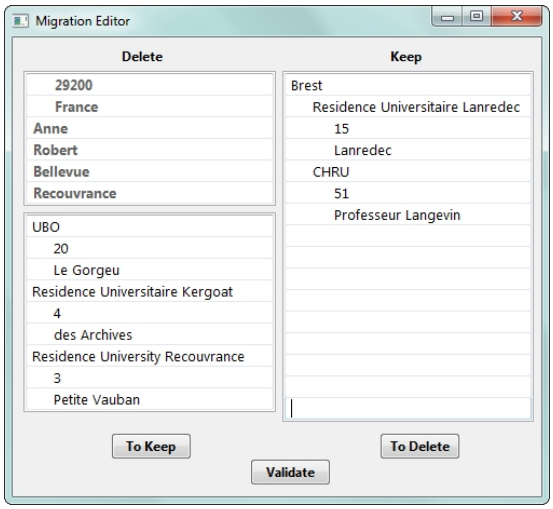


FIGURE 5.9 – Spécification de migration personnalisée 3

5.2.6 Migration et réutilisation de MapView

Dans cette étape de la réutilisation, Migrate est exécuté pour obtenir un modèle migré conforme au méta-modèle *MapView.ecore* (figure 5.2). Nous présentons les modèles migrés produits à partir des différentes spécifications de migration. Les carte produites par MapView à partir des différents modèles migrés sont également présentées.

Le modèle migré ne contient que les endroits de la ville car les citoyens ont été supprimés, tous les bâtiments ont été ajoutés directement à la ville, enfin les quartiers Bellevue et Recouvrance ont été supprimés. La carte affiche le centre de santé CHRU (marqueur rouge avec étiquette H), l'université UBO (marqueur bleu avec étiquette U) et les trois résidences universitaires Lanredec, Kergoat et Recouvrance (marqueur marron avec étiquette D).

La figure 5.11 illustre le modèle migré qui ne contient que les résidences universitaires. La carte affiche donc les trois résidences représentées par le marqueur D.

La figure 5.12 illustre le résultat de l'outil quand la spécification de migration est celle de la figure 5.8. La carte affiche l'UBO et les résidences universitaires Lanredec et Kergoat.

Enfin, la figure 5.13 illustre la carte de la ville de Brest avec le centre de santé CHRU et la résidence universitaire la plus proche (Lanredec).

5.2.7 Conclusion

Cet exemple simple nous permet de comparer la définition de migrations spécifiques de notre approche et deux approches existantes. Dans cette étude de cas, l'application de la migration inverse n'est pas nécessaire parce que l'outil réutilisé est un outil d'analyse qui ne fait pas de modifications sur le modèle migré.

Dans le contexte de la co-évolution, les migrations sont faites de façon automatique et ne permettent pas de personnaliser les migrations. Si les approches de migration fournissent du code généré éditable pour définir des migrations spécifiques, le code risque d'être long et l'adaptation risque d'être prendre beaucoup de temps, et la mi-

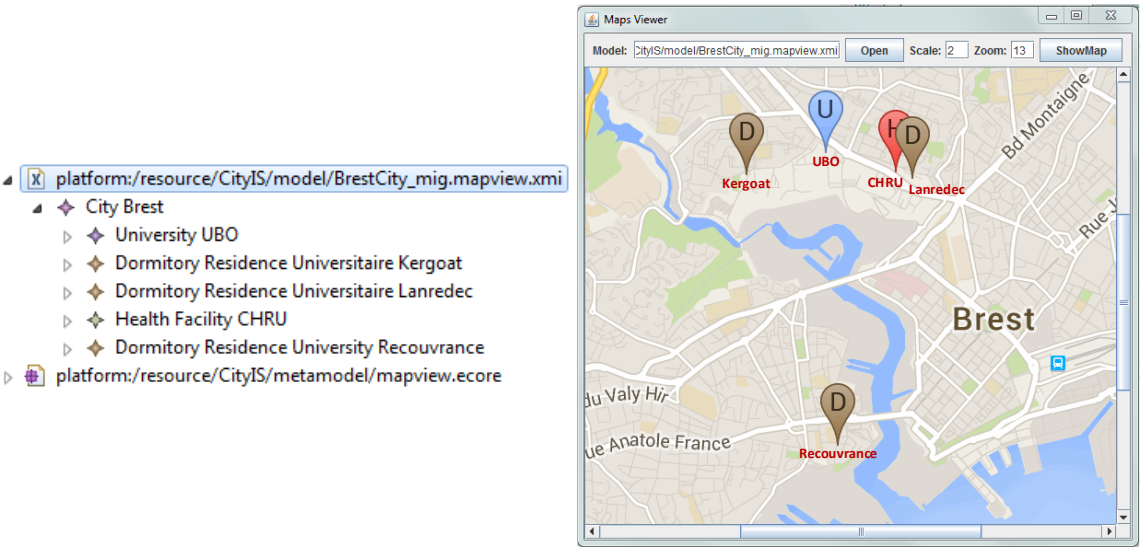


FIGURE 5.10 – Modèle de la ville migré et tous les bâtiments sur la carte

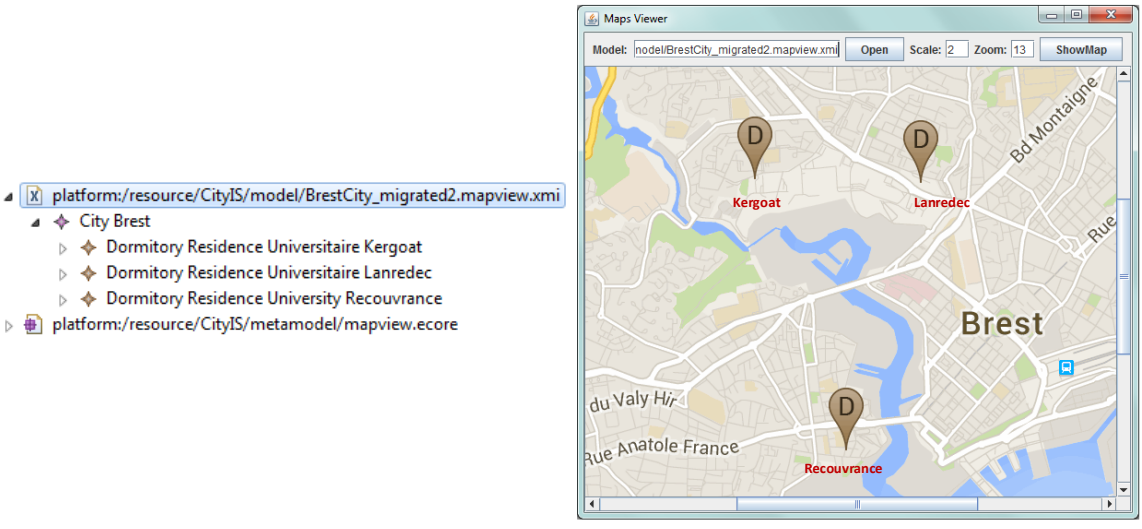


FIGURE 5.11 – Modèle de la ville migré et les résidences universitaires sur la carte

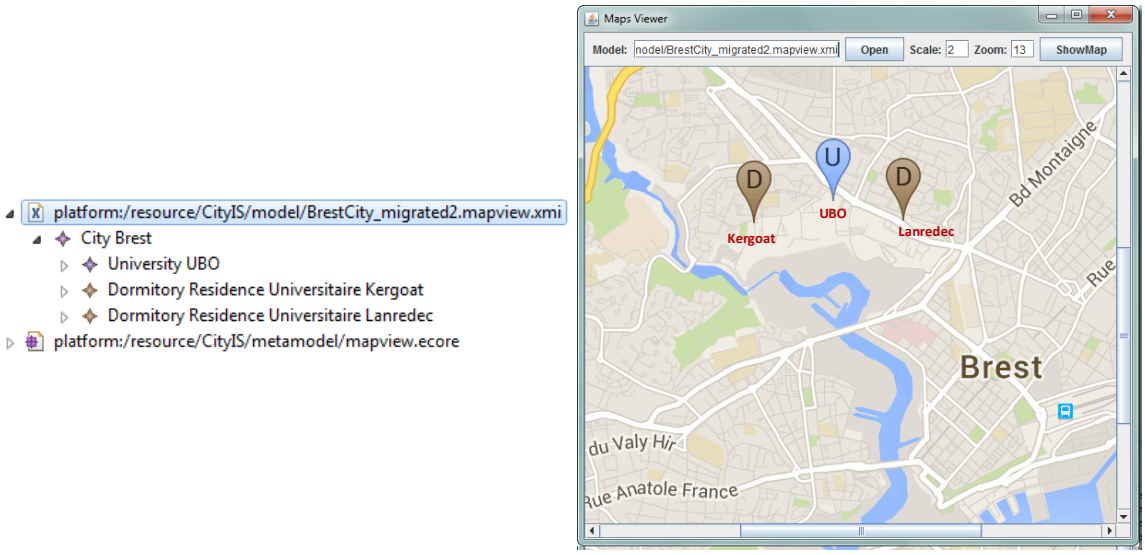


FIGURE 5.12 – Modèle de la ville migré et université et résidences sur la carte

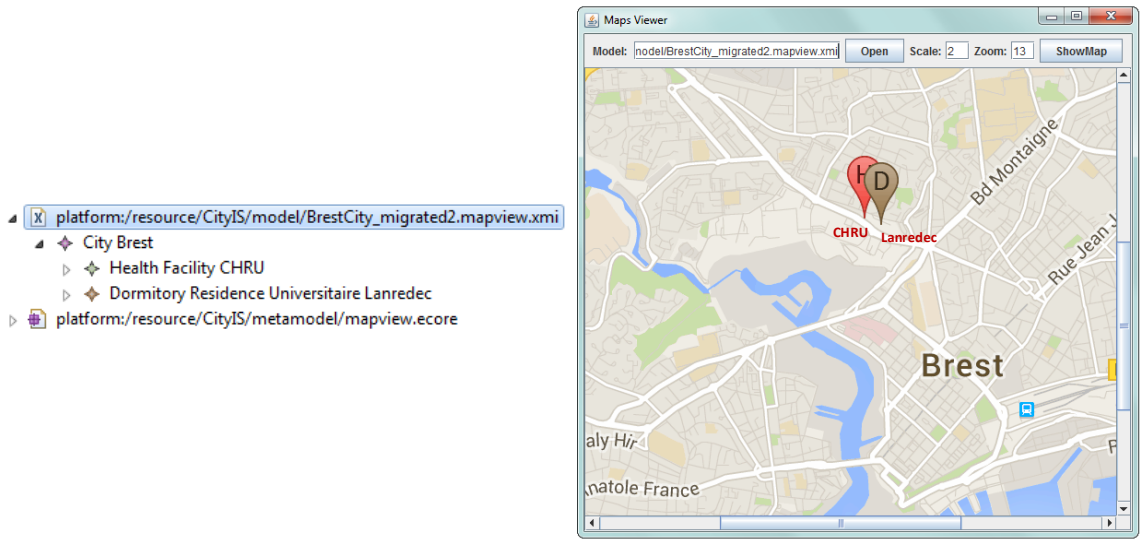


FIGURE 5.13 – Modèle de la ville migré et les résidences universitaires à proximité du CHRU sur la carte

gration peut déclencher des erreurs à cause de la complexité du code. Notre approche permet de mettre à jour des données spécifiques et non pas de code de migration. La mise à jour est faite de manière contrôlée car chaque modification dans la spécification de migration est validée par l'interface d'édition.

Les spécifications de migrations permettent d'agir sur chaque instance individuelle du modèle car elle sépare les préoccupations liées au méta-modèle des préoccupation liés aux modèles.

Les spécifications de migration éditables permettent de définir différentes possibilités de migration sans modifier le code de la migration et sans avoir besoin de maitriser un langage de transformations de modèles spécifique. Une seule spécification de migration suffit pour générer des modèles migrés divers. Aucune ligne du code de migration n'a été modifiée.

5.3 Réutilisation dans le contexte d'Orcc

5.3.1 Présentation du cas d'étude

Afin de valider notre approche pour faciliter la réutilisation d'outils, nous proposons de réutiliser deux outils de réécriture dans le contexte d'*Orcc*. Orcc est un framework pour la génération de code logiciel et matériel pour le développement d'applications multimédia. La première étude de cas a permis de mettre en évidence l'intérêt de générer une spécification de migration intermédiaire dans le processus de la coévolution. Au delà de ce premier aspect, notre approche a pour principal objectif de permettre la recontextualisation du résultat d'un outil de réécriture : l'outil est réutilisé dans un contexte autre que son domaine de définition.

Cette deuxième étude de cas, permet plus précisément d'évaluer les bénéfices de la recontextualisation. Nous nous intéressons à Orcc parce que c'est un framework classique de modélisation qui doit fournir beaucoup de fonctionnalités (*i.e.* propagation de constantes, suppression du code mort, aplatissement des réseaux, ordonnancement de composants). Ces fonctionnalités modifient le modèle Orcc afin de produire un modèle permettant la génération de code optimisé.

Dans le cadre d'Orcc, nous nous intéressons à la réutilisation des outils *HSFlattener* et *GScheduler*. Ces outils implémentent des algorithmes génériques qui répondent à deux fonctionnalités qui doivent être fournies par Orcc (aplatissement des réseaux, ordonnancement de composants). Dans un premier temps nous présentons un aperçu d'Orcc. Ensuite nous présentons l'outil *HSFlattener* et l'outil*GScheduler* en détail. Et les bénéfices d'utiliser le framework de réutilisation sont présentés.

5.3.2 Contexte d'Orcc

L'approche de flot de données et RVC-CAL

Cette section présente RVC-CAL (Reconfigurable Video Coding [ZB14]), un DSML dédié à l'implémentation d'applications vidéo. RVC-CAL nécessite beaucoup d'outils spécifiques qui sont disponibles dans d'autres contextes. La réutilisation de ces outils est une façon de répondre efficacement aux exigences de RVC-CAL.

L'approche de *flot de données* permet de représenter explicitement à la fois le parallélisme spatial et le parallélisme temporel d'une application. L'approche de flot de données rassemble une large communauté qui propose plusieurs langages et outils [Sou12]. Dans ce paradigme de programmation, les calculs sont modélisés au moyen d'un graphe orienté. Les données sont transportées vers les unités de calcul, elles sont transformées par les unités de calcul.

Un programme de flot de données est généralement spécifié via une interface visuelle dans laquelle les unités de calcul sont représentées par ce qu'on appelle des *acteurs*. Les données (liste de *tokens*) transitent d'un acteur à l'autre par l'intermédiaire de *canaux* de type *FIFO* (First In First Out). Cette représentation décrit le parallélisme concernant les tâches de l'application. Le comportement d'un programme est définit par un modèle de calcul (*Model of Computation* MoC) qui spécifie un ensemble de règles concernant l'exécution du programme.

Dans RVC-CAL il y a deux niveaux : les *acteurs* et les *réseaux*. Un acteur peut être un réseau d'acteurs ou un acteur *final*. Un acteur final est spécifié par un langage impératif qui décrit son comportement.

À titre d'illustration, la figure 5.14 présente un réseau, composé de trois acteurs, qui représente la lecture d'un signal audio et de parole entrant (Source), un *linear predictive coding (LPC) analyzer* (GSM_LPC) et un acteur qui affiche la sortie (TestOutput). Les flèches représentent le flot de données entre ces acteurs.

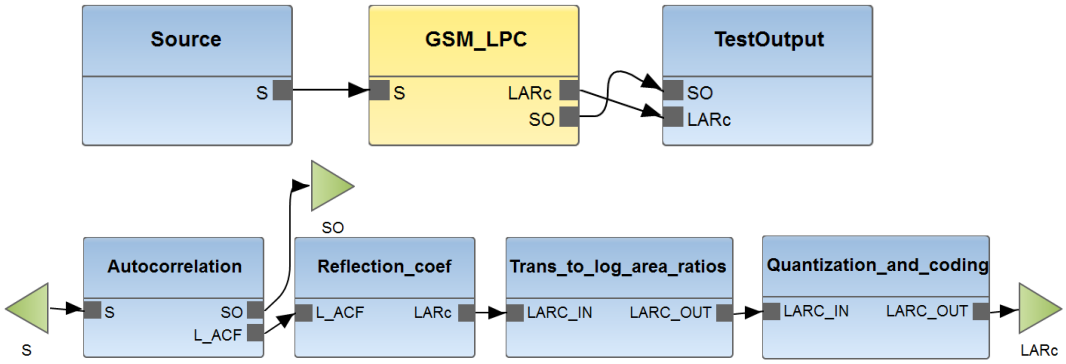


FIGURE 5.14 – Vue graphique d'un réseau d'acteurs

GSM_LPC englobe un réseau d'acteurs, dont le réseau interne est composé d'un port d'entrée (S), deux ports de sortie (SO et LARc) et quatre acteurs (Autocorrelation, Reflection_coef, Trans_to_area_ratios et Quantization_and_coding).

Open RVC-CAL Compiler (Orcc)

Orcc est l'éditeur et compilateur pour des applications RVC-CAL. L'éditeur repose sur EMF et Xtext. Comme présenté dans la figure 5.15, Orcc est composé d'un *front-end*, d'un *middle-end* et de plusieurs *back-ends* qui génèrent du code (*e.g.* C, VHDL, LLVM, Java). Le front-end est principalement un analyseur de RVC-CAL. Il gère la syntaxe concrète du langage. Le middle-end effectue plusieurs passes de compilation. Ces passes sont divisées en trois catégories : l'optimisation, l'analyse et la génération. Une passe consiste à appeler au moins une fonction dédiée. Dans cette étude de cas, l'accent est mis sur ces fonctions et leur potentiel de réutilisation. Nous nous intéressons à

la réutilisation d'une passe d'optimisation parce que le résultat doit être remis dans le contexte RVC-CAL. Ceci nous permet d'illustrer l'utilisation de la migration inverse de notre approche. Les autres types de passes (analyse et génération) exigent seulement une migration et ne nécessitent pas de migration inverse.

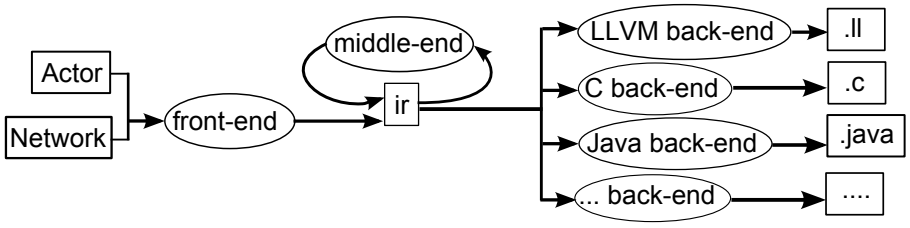


FIGURE 5.15 – Infrastructure d'Orcc

Dans le middle-end, 46 % des passes sont spécifiques. Elles sont liées au fait qu'Orcc peut gérer des modèles hétérogènes de calcul grâce à une classification dédié des acteurs (selon leur MoC). Ces passes doivent être implémentées directement dans le framework d'Orcc et ne peuvent pas être réutilisées parce qu'elles sont dépendantes d'Orcc.

D'autre part, 54 % des passes sont génériques. Elles manipulent des concepts et des algorithmes qui sont implémentés ailleurs (en particulier dans des DSMLs dédiés). Notamment, le comportement des acteurs repose sur les graphes de contrôle de flot de données *Control Data Flow Graphs* (CDFG, qui ne sont pas spécifiques à RVC-CAL). Et les réseaux sont liés aux flots de données hiérarchiques *Hierarchical Data Flow* (HDF) qui ne sont pas non plus spécifiques à RVC-CAL.

Méta-modèle Orcc

Le méta-modèle *Orcc.ecore* est présenté par la figure 5.16. Il comporte 69 EClasses, 118 EReferences et 48 EAttributes. L'implémentation d'une passe dans ce contexte implique la compréhension et la manipulation de tous les concepts du méta-modèle. Ceci rend difficile la tâche d'implémentation des passes. Pour les passes spécifiques, cette difficulté ne peut pas être éludée. Pour les passes génériques, on peut la contourner grâce à la réutilisation.

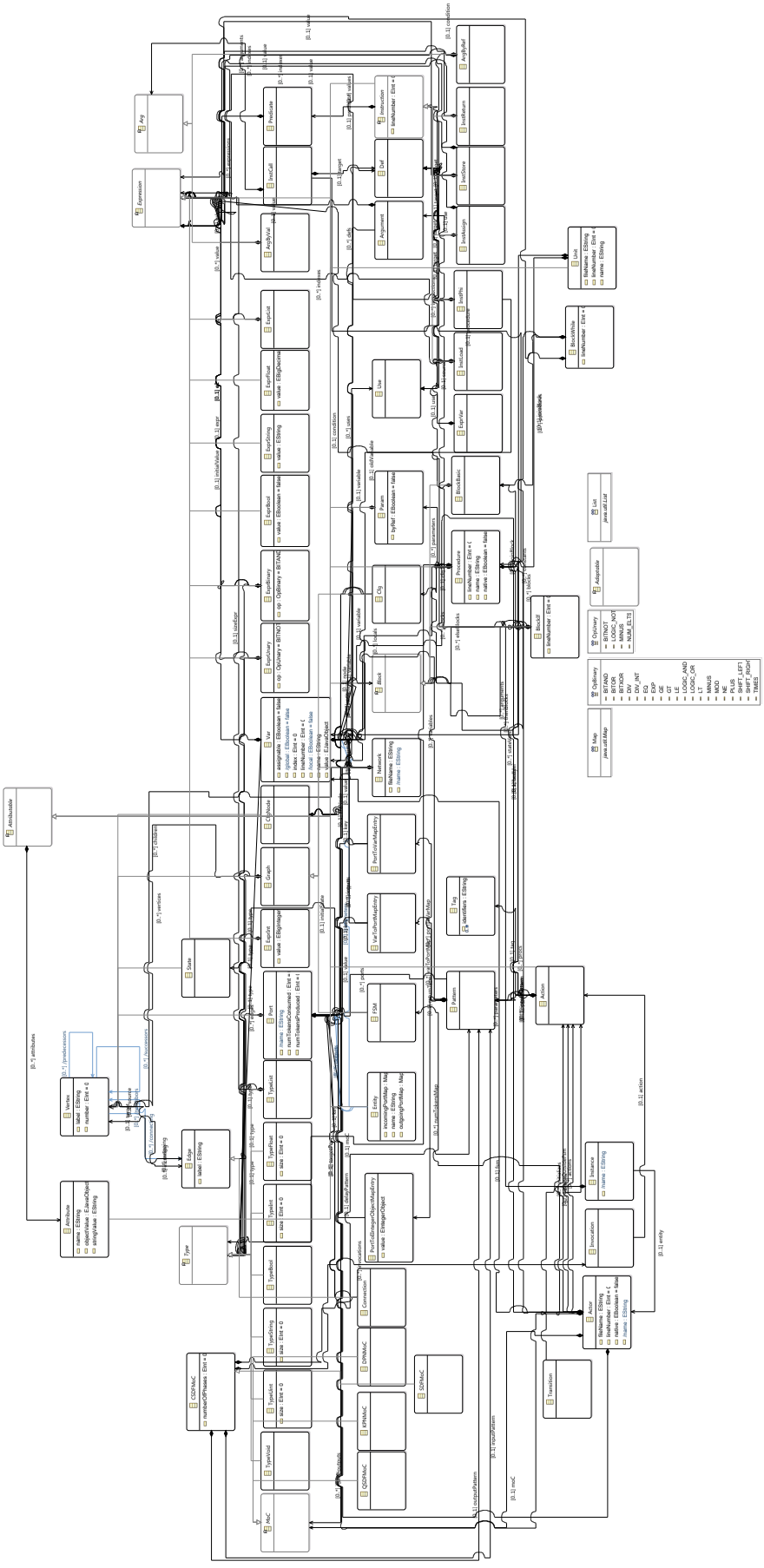


FIGURE 5.16 – Vue graphique du méta-modèle Orcc.ecore

Réutilisation pour fournir les passes génériques

Parmi les passes génériques d'Orcc, nous choisissons l'aplatissement de réseaux (*NetworkFlattener*). L'aplatisseur de réseaux doit produire un réseau d'acteurs dans lequel tous les acteurs sont au même niveau. L'algorithme 1 précise l'ajout d'aplatissement mis en œuvre comme indiqué dans [Wip10]. Il s'agit d'un algorithme bien connu. Cela fait qu'aucune hypothèse spécifique à propos de RVC-CAL n'est nécessaire pour son implémentation. Sa mise en œuvre dans le contexte d'Orcc pourrait être évitée et une des implémentations disponibles pourraient être réutilisées à sa place.

```
input  : A hierarchical network of actors A and FIFO buffers F
output: A flattened network of actors
foreach a ∈ A do
    if a is refined by a network then
        let A_s be the subnetwork of actors;
        flatten(A_s);
        A ← A ∪ {a ∈ A_s};
        F ← F ∪ {f ∈ F_s | s(f) ∈ A_s ∧ t(f) ∈ A_s};
        connect(A, A_s, a);
        A ← A \ {a}
    end
end
```

Algorithm 1: Description de l'algorithme d'aplatisement

La réutilisation d'un aplatisseur a été choisi pour les raisons suivantes :
- L'algorithme d'aplatissement est très connu et nombreuses implémentation de celui-ci sont déjà disponibles.
- La sortie de l'outil réutilisé doit être intégrée dans le contexte du Orcc.
- Les concepteurs d'Orcc ont mis en place leur propre aplatisseur, la comparaison de l'effort de réutiliser un aplatisseur existant et l'effort de mise en œuvre de leur aplatisseur est pertinente.

Nous présentons maintenant les bénéfices apportés par notre approche de réutilisation pour fournir une passe d'optimisation au moyen d'une réutilisation. Nous allons réutiliser la fonctionnalité *flatten* proposée par l'outil *HSFlattener*.

5.3.3 Réutilisation de *HSFlattener*

Dans un premier temps nous présentons l'outil *HSFlattener*. Il s'agit d'un outil générique indépendant d'Orcc et qui fournit la fonctionnalité d'aplatissement dont nous avons besoin. Ensuite nous présentons en quoi notre approche facilite la réutilisation de cet outil dans le contexte d'Orcc. Nous analysons les avantages issues de l'utilisation de l'approche.

HSFlattener

Le méta-modèle de l'outil *HSFlattener* est présenté dans la figure 5.17. Il représente essentiellement des graphes hiérarchiques dont les sommets sont des *composants génériques* (Composite), des ports (Port) et des *sous-graphes* (Component). Un port est

contenu dans un composant. Il relie les composants et les sous-graphes spécifiques par le biais des connections (Connection). Les sous-graphes spécifiques qui sont reliés aux ports sont nommés *composites*. Ce méta-modèle n'implique que les concepts qui sont nécessaires pour effectuer l'aplatissement d'une structure hiérarchique.

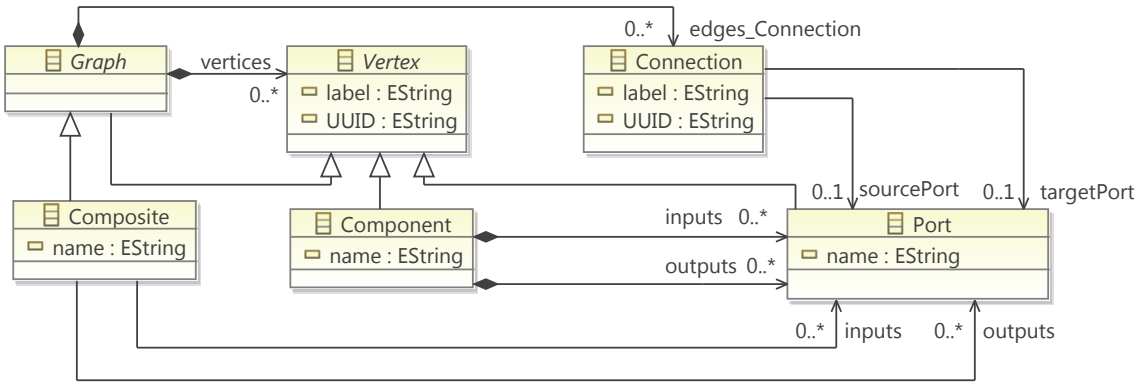


FIGURE 5.17 – Vue graphique du méta-modèle HierarchicalStructure.ecore

Le modèle de la figure 5.14 est conforme au méta-modèle d'Orcc. La réutilisation de l'aplatisseur est appliquée sur ce modèle.

Les différents étapes du processus de la réutilisation sont brièvement présentées.

Simplification d'Orcc

Dans un premier temps, nous extrayons le sous-ensemble d'Orcc qui concerne les structures hiérarchiques. Nous appelons cette partie *OrccSimplified.ecore*. L'extraction de la partie pertinente d'un méta-modèle du point de vue d'un outil à réutiliser facilite sa compréhension et la manipulation des modèles. Toutes les préoccupations de mise en œuvre et les préoccupations qui concernent d'autres fonctionnalités sont écartées.

L'extraction est possible par l'application de la spécification de refactoring *Orcc2OrccSimplified.modif*. Elle est le résultat du processus suivant :

- Génération d'une spécification de refactoring *EraseAllOrcc.modif*, dans laquelle tout est supprimé par défaut.
- Suppression de tous les concepts qui ne sont pas présents dans la définition de la structure hiérarchique. Ceci implique que l'application de l'opérateur remove est annulée pour les classes qui sont dans la structure hiérarchique : Edge, Graph, Vertex, Connection, Port et Network, Actor.
- Conservation des concepts qui encapsulent quelques informations utiles pour une analyse ultérieure. Ceci implique que l'application de l'opérateur remove est annulée pour les classes : Attributable et Instance.

Le méta-modèle simplifié obtenu est illustré par la figure 5.18. Ce méta-modèle ne contient que les concepts qui concernent la partie hiérarchique des réseaux d'acteurs. Il a été obtenue rapidement en utilisant peu d'opérateurs.

Du SimpleOrcc vers HierarchicalStructure

Le méta-modèle *SimpleOrcc.ecore* est proche du méta-modèle de structures hiérarchiques du *HSFlattener*. *SimpleOrcc.ecore* est alors transformé en un méta-modèle

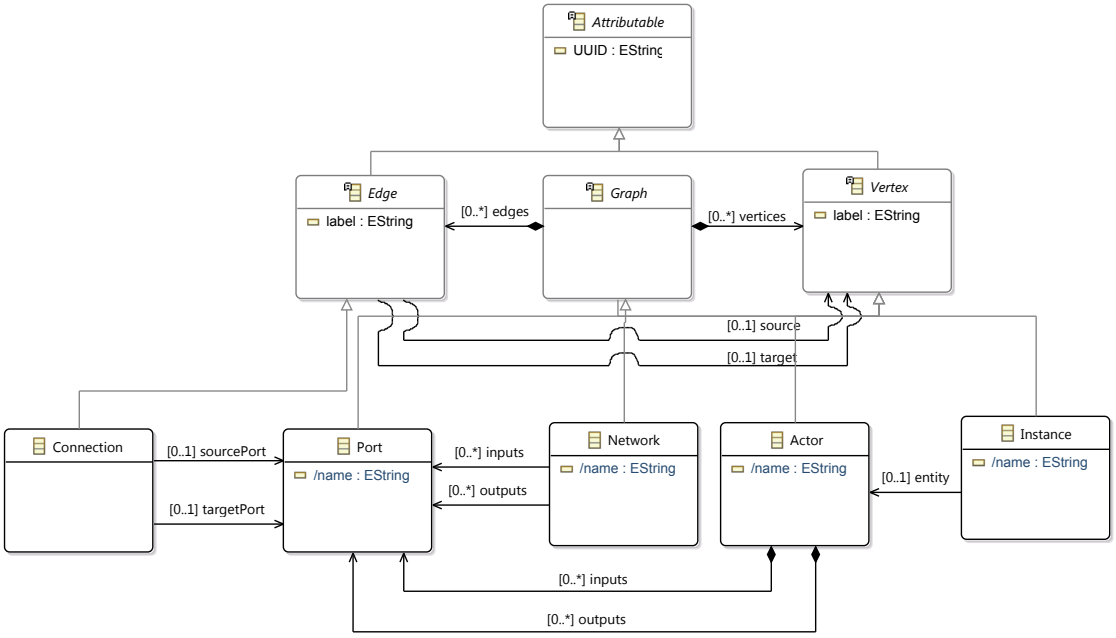


FIGURE 5.18 – Vue graphique du méta-modèle SimpleOrcc.ecore

identique au méta-modèle *HierarchicalStructure.ecore*. Afin de faire cette transformations nous utilisons la spécification de *SimpleOrcc2HierarchicalStructure.modif* (présentée dans le listing 5.4), elle est le résultat des actions suivantes :

- Génération d'une spécification de refactoring qui par défaut n'apporte pas de modifications (NoModif).
- Édition de la spécification de refactoring de la façon suivante :
  - Application des opérateurs *flatten* et *hide* sur les classes Attributable et Edge. Ces opérateurs suppriment les classes, mais gardent certains attributs et références qui sont dans le méta-modèle de structures hiérarchiques.
  - Suppression de la classe *Instance* parce qu'elle n'est pas présente dans le méta-modèle de structures hiérarchiques.
  - Renommage de Actor en Component.
  - Renommage de Network en Composite.

Listing 5.4 – Édition d'Orcc-to-hierarchicalstructure.modif

```
root SimpleOrcc to HierarchicalStructure
Prefix SimpleOrcc to HierarchicalStructure
URI "http://SimpleOrcc.sf.net/model/2011/Orcc"
     to "http://model/2015/HS"

class {
   Port {att name} ;
   Actor to Component {ref inputs att name ref outputs} ;
   Connection {ref sourcePort ref targetPort} ;
   flatten hide Attributable {att UUID} ;
   Graph {ref edges ref verties} ;
   Vertex {att label} ;
   flatten hide Edge {att label} ;
   remove Instance {remove ref entity remove att name} ;
```

Après la compréhension du méta-modèle d'Orcc, une demi-journée à suffit pour définir les deux spécifications de refactoring. Nous avons obtenu de manière simple un méta-modèle qui ne comporte que 6 classes au lieu de manipuler les 69 classes du méta-modèle d'Orcc. Il est à noter que la création de plusieurs spécifications de refactoring n'est pas en fait obligatoire. Toutes les modifications auront pu être indiquées dans la même spécification Modif.

Réutilisation de *HSFlattener*

La spécification de migration générée par défaut est appliquée pour migrer le modèle de la figure 5.14. La figure 5.19 illustre un extrait du graphe d'objets du modèle. Les éléments en pointillé sont supprimés automatiquement lors de la migration. Tous les autres éléments font partie du graphe migré. Une fois le modèle migré, *HSFlattener* peut être réutilisé sur le modèle migré.

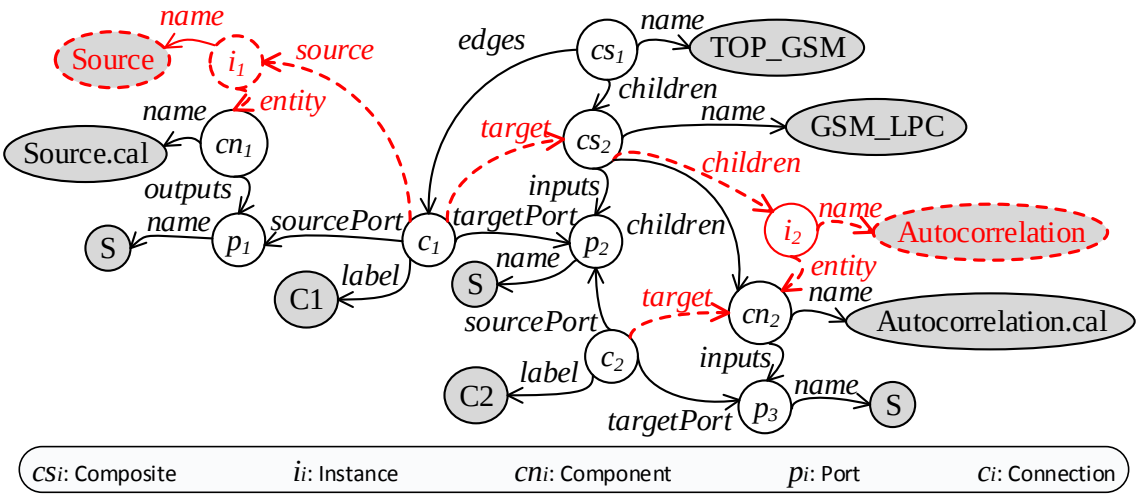


FIGURE 5.19 – Extrait du graphe d'objets correspondant au modèle de la figure 5.14

HSFlattener prend le modèle migré en entrée et l'aplatit. Tous les composants sont mis dans le même niveau de hiérarchie, et le graphe *GSM_LPC* et leur ports sont supprimés. Les noms des composants sont modifiés en ajoutant un préfixe qui correspond à l'ancien nom du graphe (*GSM_LPC* dans ce cas). La connection c_1 entre le port p_1 et le port p_2, et la connection c_2 entre le port p_2 et le port p_3 sont remplacées par une nouvelle connection c_3 entre le port p_1 et le port p_3. La figure 5.20 illustre plus précisément le résultat obtenu par rapport à la figure 5.19. Les éléments pointillés sont supprimés par l'outil et les éléments représentés par une ligne épaisse sont crées par l'outil.

Pour résumer, les actions réalisées par *HSFlattener* sur le modèle migré de la figure 5.19 consistent à *supprimer*, à *créer* ou à *mettre à jour* les éléments du modèle. Si la création ou la mise à jour d'un élément est calculée à partir d'au moins un élément existant, alors ces liaisons sont exprimées dans le *graphe de dépendances* correspondant. Pour cette étude nous avons implémentée la création du graphe de dépendances directement dans l'outil. Notamment dans ce graphe, les connections c_1 et c_2 sont liées à c_3 parce que c_1 et c_2 sont utilisées pour produire c_3.

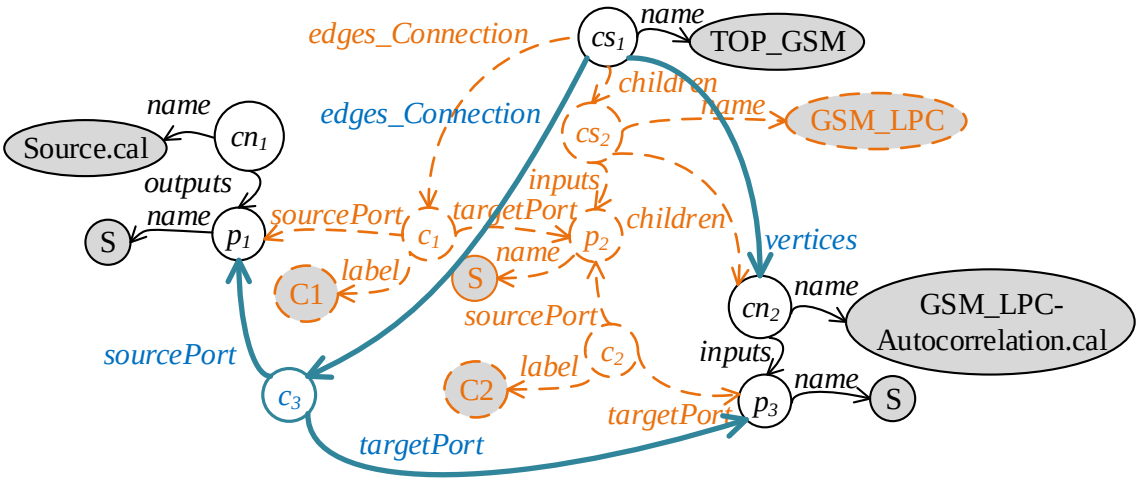


FIGURE 5.20 – Extrait du graphe d'objets aplati

Jusqu'ici seulement une intervention de la part de l'utilisateur a été faite (définition des opérateurs dans la spécification de refactoring), elle est minimale parce que la majorité des concepts sont supprimés par défaut. Aucune modification dans le code de l'outil n'a été faite et l'outil a été effectivement réutilisé. Le résultat est comme prévu, un modèle aplati. Ce modèle aplati est toujours conforme au méta-modèle de la figure 5.17. Il est remis dans son contexte d'origine (conforme au méta-modèle d'Orcc) grâce à la recontextualisation.

Du HierarchicalStructure vers Orcc

La recontextualisation est faite automatiquement. La figure 5.21 illustre le *graphe recontextualisé*. Les éléments qui ont été supprimés lors de la migration (en pointillé dans la figure 5.19) sont récupérés et reconnectés au modèle. Par rapport au graphe de dépendances, les éléments qui étaient associés à c_1 et c_2 sont maintenant associés à c_3. Enfin, le modèle obtenu est illustré par la figure 5.22. Ce modèle est une version aplatie du modèle de la figure 5.14 et il peut être édité par les éditeurs dédiés d'Orcc.

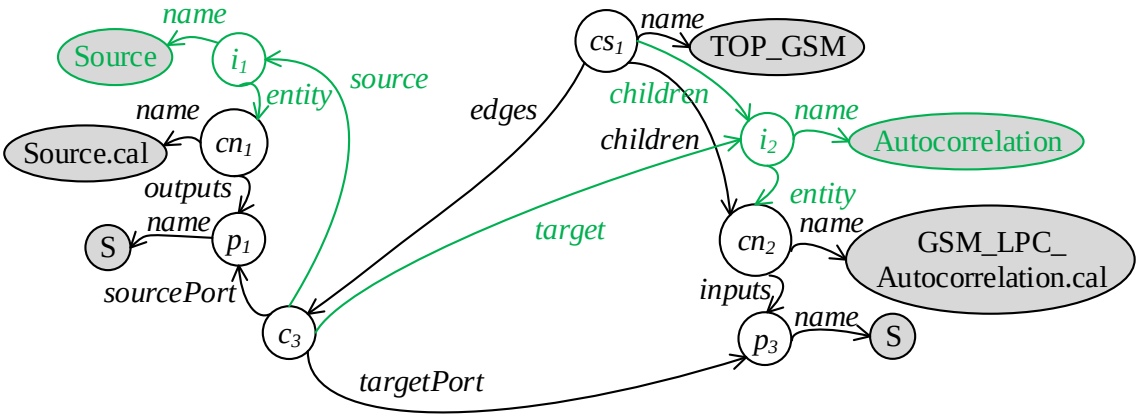


FIGURE 5.21 – Extrait du graphe d'objets correspondant au modèle de la figure 5.22

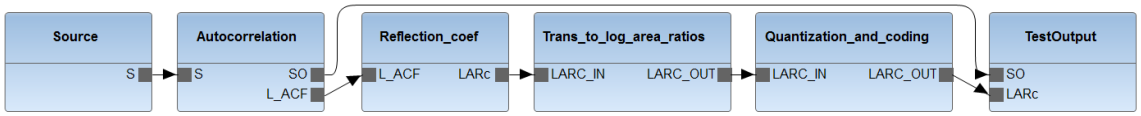


FIGURE 5.22 – Version aplatie du modèle de la figure figure 5.14

5.3.4 Évaluation

Résultat attendu

Un des critères pour évaluer notre approche est l'obtention d'un résultat correct du point de vue de ce qui doit être fourni par le framework Orcc.

NetworkFlattener est une version de l'aplatisseur qui a été implémentée directement dans le framework d'Orcc. Pour notre exemple, le résultat obtenu avec l'aplatisseur codé directement dans Orcc est illustré par la figure 5.23.

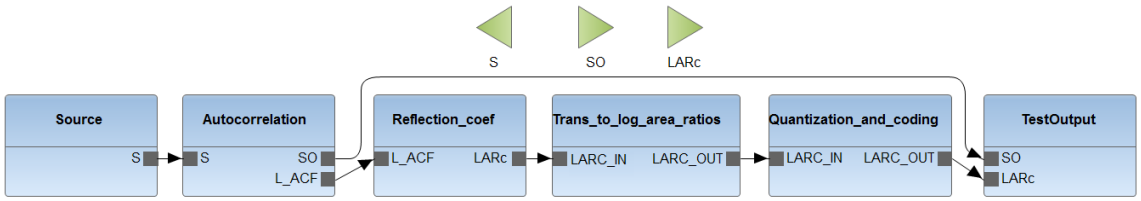


FIGURE 5.23 – Résultat du *NetworkFlattener* spécifique à Orcc

Le modèle aplati contient des ports (S, SO, LARc) qui auraient du être supprimés. Il ne sont plus utiles dans le modèle car ils sont déconnectés des autres éléments du modèle. Il faudrait appliquer une fonctionnalité de *post-processing* afin de supprimer les éléments inutiles.

On peut noter que à l'inverse, le résultat obtenu au moyen de la réutilisation est correct. La raison est certainement qu'il est plus facile de ne manipuler que les concepts nécessaires à une fonctionnalité pour la coder correctement.

Lignes de code

Le deuxième critère que permet de valider notre approche est l'effort réalisé en terme de lignes de code. L'effort requis dans notre approche pour réutiliser le *HSFlattener* est comparé avec l'effort nécessaire pour implémenter le *NetworkFlattener* dans Orcc. Les résultats sont présentés dans le tableau 5.1.

56 lignes de code on été utilisées pour implémenter le *NetworkFlattener* dans Orcc. Le *Flattener* réutilisé était déjà développé (pour information, il a 68 lignes de code), il n'y aucune ligne de code à implémenter. Pour les spécifications de refactoring, nous n'avons modifié que 12 lignes car les suppressions sont gérées par la spécification *EraseAll*. Les moteurs de migration Migrate et de migration inverse ReverseMigrate sont génériques et peuvent être utilisés pour n'importe quel méta-modèle, aucune modification du code est nécessaire. Enfin, le modèle aplati au moyen de la réutilisation de *HSFlattener* a été modifiée parce que l'outil réutilisé renomme les états d'une manière différente de celle qui est préconisée. Cette mise en forme, nécessite la modification

TABLEAU 5.1 – Comparaison de l'effort en terme de lignes de code

	flatten
Existing NetworkFlattener	56
HSFlattener	– (already exists)
Modif	12
Migrate + Reverse Migrate	– (generic)
Final adaptation	4

des noms de 4 acteurs (Autocorrelation, Reflection_coef, Trans_to_log_area_ratios et Quantization_and_codig). La mise à jour des noms peut être faite directement sur le modèle aplati.

Pour conclure, seulement 16 (12 Modif + 4 adaptations finales) lignes nécessitent l'intervention de l'utilisateur dans cet exemple. Cette quantité est réduite par rapport aux 56 lignes de code nécessaires pour implémenter le *NetworkFlattener* à partir de zéro dans le contexte d'Orcc.

Complexité

Le troisième critère utilisé pour évaluer l'utilisation de notre approche est la quantité d'éléments du méta-modèle qui doivent être manipulés pour implémenter la fonctionnalité de aplatissement. Le tableau 5.2 présente la synthèse des éléments qui ont été manipulés dans chaque cas (implémentation de *NetworkFlattener* et réutilisation de *HSFlattener*).

TABLEAU 5.2 – Éléments du méta-modèle

	Class	DataType	Enum	Reference	Attribute
Existing Orcc Network-Flattener	69	2	2	118	48
HSFlattener	6	0	0	8	7

Ce tableau met en évidence les avantages de développer un outil en considérant uniquement les concepts qui sont utiles pour implémenter leur fonctionnalités. Les préoccupations d'édition et d'implémentation peuvent être séparées des préoccupations de traitement des modèles. Le méta-modèle Orcc contient des attributs dérivés qui complexifient et polluent le méta-modèle et rendent difficile l'implémentation des outils dans ce contexte.

HSFlattener a été testé sur de nombreuses configurations.

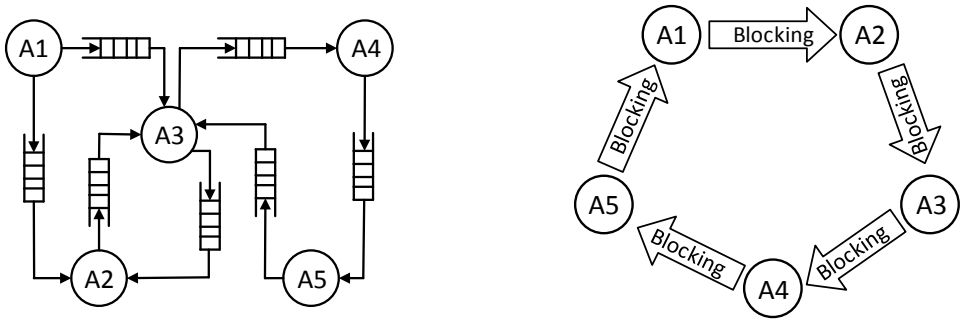
5.4 Réutilisation de *GScheduler*

Nous présentons maintenant les bénéfices apportés par notre approche de réutilisation pour fournir la fonctionnalité d'*ordonnancement* des acteurs. D'abord, nous présentons une politique d'ordonnancement existante dans Orcc et les inconvénients liés à son utilisation. Ensuite nous pressentons l'outil *GScheduler* à réutiliser. Enfin

nous analysons les avantages de réutiliser l'outil au moyen de l'approche de réutilisation.

Politique d'ordonnancement Round-robin

Dans l'approche de flot de données, il est nécessaire de déterminer l'ordre d'exécution des acteurs. Une des politiques d'ordonnancement des plus utilisées est le *round-robin*. Il s'agit d'une politique d'ordonnancement simple qui gère l'exécution d'un seul acteur à un moment donnée. Round-robin évalue pour chaque acteur, les conditions nécessaires pour l'exécuter et il l'exécute jusqu'à ce que les conditions soient respectées. Cette politique garantie que chaque acteur a la même chance d'être exécuté, et évite le blocage et la famine. La figure 5.24 illustre un réseau d'acteurs et son exécution dans un ordre circulaire. Par exemple, A1, A2, A3, A4 et A5 sont exécutés successivement, puis A1 est exécuté à nouveau et ainsi de suite [YCWR11].



(a) Réseau d'acteurs

(b) Ordonnancement

FIGURE 5.24 – Exemple d'ordonnancement avec la politique round-robin [YCWR11]

Dans Orcc, cette politique est la politique d'ordonnancement effectuée par défaut. Elle est proposée sous la forme d'une bibliothèque en langage C, qui n'est utilisable que si le code C est généré (*C back-end*).

Vu que l'ordonnancement ne s'applique qu'aux réseaux aplatis, pour faire des tests, nous choisissons un modèle dans lequel tous les acteurs sont au même niveau hiérarchique. Il s'agit du modèle de la figure 5.22, un modèle conforme au méta-modèle de la figure 5.16. Nous supposons que le code C a été généré par le back-end et que la politique round-robin est appliquée. La figure 5.25 illustre l'application de cette politique au réseau de la figure 5.26. Les actions suivantes sont effectuées :

① La chance de s'exécuter est donnée à l'acteur Source, et il est exécuté.
② La chance de s'exécuter est donnée à l'acteur TestOutput, mais les conditions qui permettent de l'exécuter ne sont pas encore satisfaites.
③ L'acteur GSM_LPC_Autocorrelation est exécuté.
④ L'acteur GSM_LPC_ReflectionCoefficients est exécuté.
⑤ L'acteur GSM_LPC_TransformationToLogAreaRation est exécuté.
⑥ L'acteur GSM_LPC_QuantizationAndCoding est exécuté.
⑦ La chance de s'exécuter est à nouveau donnée à l'acteur Source, mais cette fois-ci il ne peut pas être exécuté.
⑧ L'acteur TestOutput est enfin exécuté.

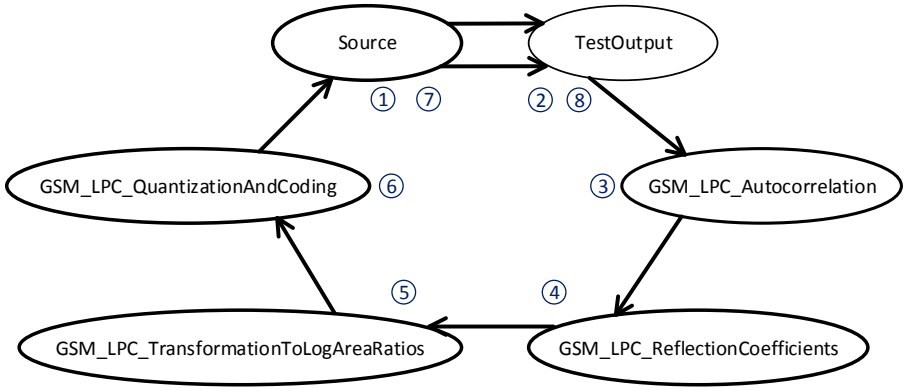


FIGURE 5.25 – Round-robin appliqué au réseau d'acteurs de la figure 5.26

Nous observons que l'ordre dans lequel le round-robin donne la chance aux acteurs pour s'exécuter, ne correspond pas à l'ordre des acteurs du modèle de la figure 5.22. Cela arrive parce que selon la sémantique opérationnelle d'Orcc, la séquence d'exécution correspond à l'ordre interne de stockage des acteurs. L'ordre interne des acteurs dans le modèle de la figure 5.22 est représenté dans la figure 5.26. Elle montre que l'acteur TestOutput est stocké avant l'acteur GSM_LPC_QuantizationAndCoding alors qu'il doit être stocké après.

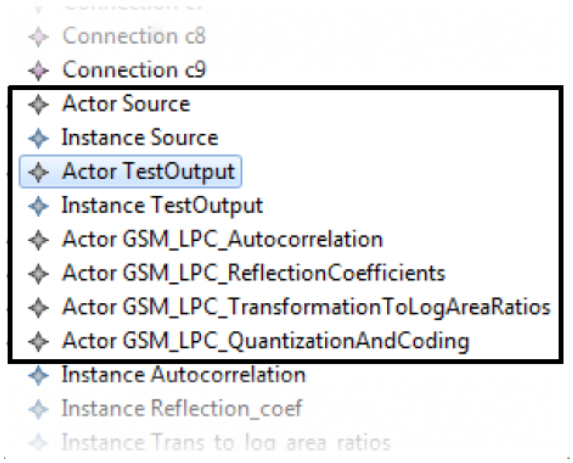


FIGURE 5.26 – Stockage par défaut des acteurs de la figure 5.22

Cette situation met en évidence le besoin d'avoir un ordre de stockage qui correspond bien à l'ordre des acteurs dans le modèle pour rendre plus efficace l'application du round-robin. Un outil qui garanti cette correspondance et qui puisse être utilisé facilement est envisagé.

GScheduler

Avant de présenter cet outil, nous exposons les raisons qui nous amènent à choisir cet outil d'ordonnancement :

– Il s'agit d'un outil de réécriture, sa sortie doit être intégrée dans le contexte d'Orcc.

– Il n'est pas implémenté dans Orcc au niveau modèle, c'est l'occasion d'ajouter à Orcc une nouvelle fonctionnalité.
– L'algorithme d'ordonnancement au niveau modèle est simple et indépendant d'Orcc.

Nous présentons maintenant, les concepts gérés par *GScheduler*. Il ne gère que des graphes simples (Graph) composés de nœuds (Node) et d'arcs (Edge). Ces graphes sont conformes au méta-modèle de la figure 5.27. Ils doivent être acycliques. Si cette condition est satisfaite, alors le *GScheduler* calcule un ordre de nœuds (basé sur l'ordre partiel induit par le graphe) qui reflète un ordre d'exécution.

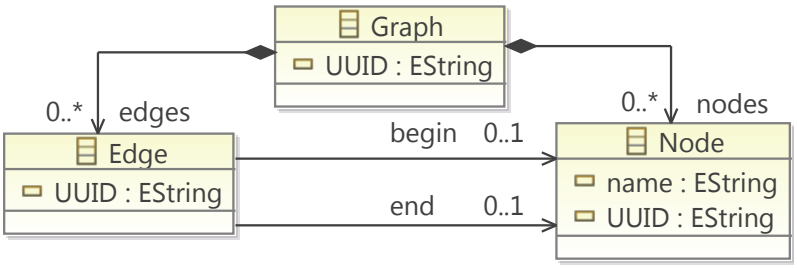


FIGURE 5.27 – Vue graphique du méta-modèle du *GScheduler*

Nous reprenons le modèle de la figure 5.22 et son arbre de stockage 5.26. *GScheduler* met à jour cet arbre afin de faire en sorte que la séquence des acteurs (modèle) corresponde à l'ordre d'exécution (stockage).

Comme dans les études de cas précédentes, le passage d'un modèle conforme au méta-modèle d'Orcc vers un modèle valide pour le *GScheduler* est fait grâce à une spécification de refactoring. Pour cet exemple, la spécification de refactoring ne conserve que les classes Graph, Actor (renommée en Edge) et Connection (renommée en Node). Elle est produite rapidement et l'intervention de la part de l'utilisateur est minimale. Le modèle migré est présenté dans la figure 5.28.

GScheduler met les acteurs en ordre et produit le modèle ordonnancé (sorted model) de la figure 5.28. Mettre le modèle ordonnancé dans le contexte d'Orcc, implique la récupération des éléments supprimés lors de la migration (instances : Source, TestOutput, Autocorrelation, Reflection_coef, Trans_to_log_area_ratios). L'ordre dans lequel ces éléments sont récupérés et reconnectés au modèle n'a pas d'importance parce qu'elle ne remet pas en cause l'ordre des acteurs (*c.f.* figure 5.28).

Évaluation

Cette évaluation est faite par rapport aux mêmes critères d'évaluation que pour la réutilisation de l'aplatisseur.

Résultat attendu

En réutilisant *GScheduler*, nous avons obtenu un modèle dans lequel l'ordre d'exécution des acteurs correspond bien à leur ordre de stockage dans l'arborescence. Contrairement au cas de l'aplatisseur, ici nous ne faisons pas la comparaison avec un ordonnanceur au niveau modèle existant dans le contexte d'Orcc parce qu'il n'existait pas un tel ordonnanceur. Mais nous appliquons le round-robin à un modèle qui a été

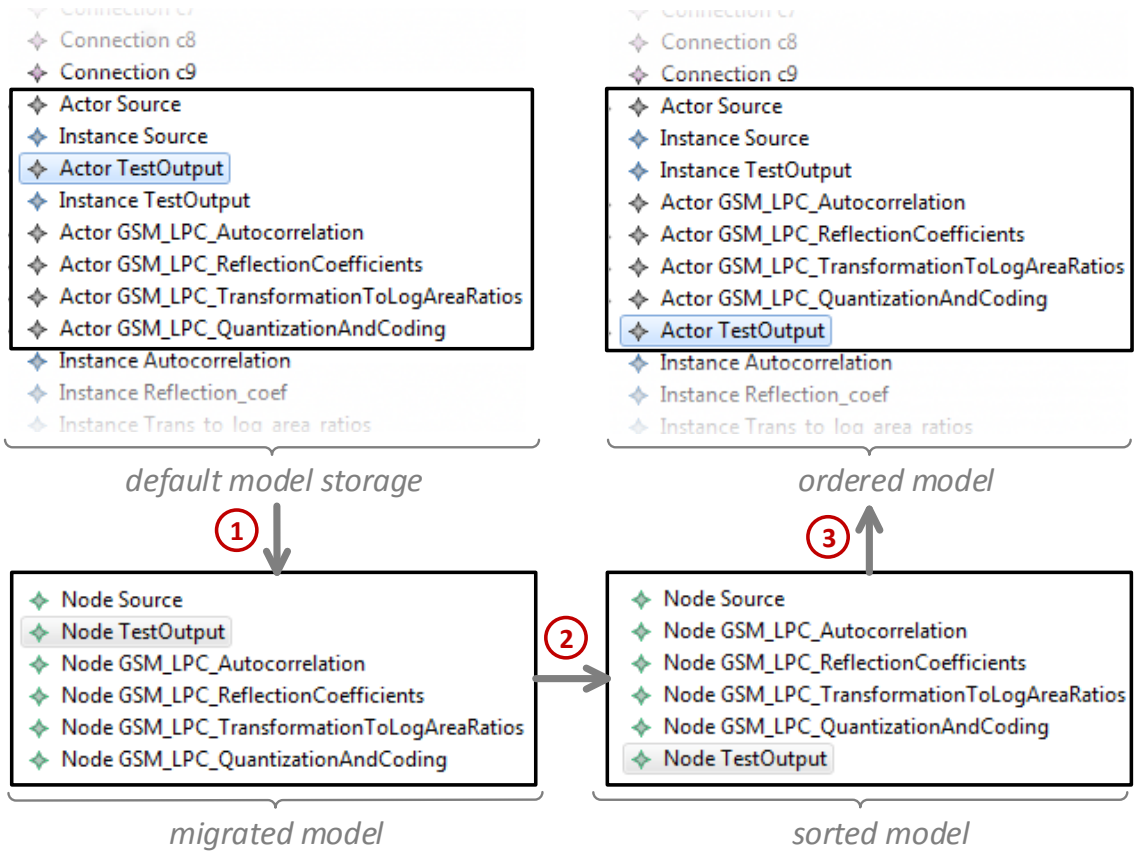


FIGURE 5.28 – Round-trip de migration pour la réutilisation de *GScheduler*

ordonné au niveau modèle (ordered model de la figure 5.28). Le résultat est illustré par la figure 5.29, dans ce cas, tous les acteurs sont exécutés dans l'ordre dans un seul cycle du round-robin. L'approche permet de fournir facilement une fonctionnalité nouvelle dans le contexte d'Orcc.

Lignes de code

Concernant l'effort demandé pour réutiliser l'ordonnanceur en fonction des lignes de code, nous mettons en évidence qu'aucune ligne de code a été implémenté lors de la réutilisation. Et la production de la spécification de refactoring n'a nécessité que la modification de 3 lignes (pour indiquer les 3 classes à conserver). Les moteurs de migration et de migration inverse restent inchangés parce qu'ils sont génériques. Et aucune adaptation du modèle n'a été nécessaire puisque *GScheduler* ne modifie pas les attributs ni les références des instances.

Complexité

Le tableau 5.3 met en évidence le nombre réduit de concepts qui sont utiles pour faire l'ordonnancement des acteurs. Si l'ordonnanceur avait été fait directement dans le contexte d'Orcc un ensemble de concepts inutiles du point de vu de l'ordonnancement auraient été manipulés. La manipulation de ces concepts aurait provoqué des résultats non attendus et aurait nécessité plus de temps et d'effort pour les traiter.

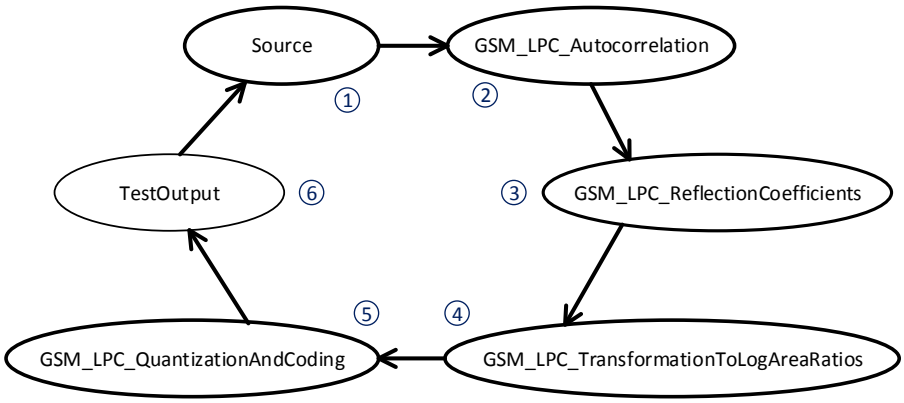


FIGURE 5.29 – Round-robin appliqué au réseau d'acteurs ordonnancé de la figure 5.28

TABLEAU 5.3 – Éléments des méta-modèles Orcc et GScheduler

	Class	DataType	Enum	Reference	Attribute
Orcc	69	2	2	118	48
GScheduler	3	0	0	4	4

5.5 Conclusion

Cette section présente trois études de cas dans lesquelles notre approche a été appliquée. La première étude nous a permis de présenter les avantages des spécifications de migration et surtout des spécifications de migrations éditables.

Dans la deuxième étude, nous présentons les facilités de l'approche pour réutiliser des outils existants. Elle prouve que la réutilisation d'un outil dédié qui se concentre sur une préoccupation spécifique donne de meilleures résultats que si l'on implémente une fonctionnalité générique dans un contexte complexe. *HSFlattener* a été testé sur de nombreuses configurations et le résultat obtenu est correct. De plus, nous constatons que le *NetworkFlattener* spécifique à Orcc n'a pas passé tous les tests. Le modèle aplati contient des *ports* qui auraient du être supprimés. Les interventions de l'utilisateur dans le processus de réutilisation sont minimales. Les données manipulées sont les données nécessaires pour appliquer la fonctionnalité d'un outil. Cet exemple simple montre la difficulté de mettre en œuvre un algorithme simple et bien connu dans un contexte complexe impliquant des concepts qui concernent des préoccupations différentes de celles de la fonctionnalité.

La troisième étude de cas représente la réutilisation d'un outil pour fournir une fonctionnalité qui n'est pas encore implémentée dans le domaine d'application. Le résultat obtenu est le résultat attendu. L'intervention de la part de l'utilisateur est minimale. Et aucune ligne de codé n'a été développée.

L'approche peut être vue comme un moyen efficace pour implémenter des DSML via la réutilisation d'outils existants. Elle garantie que les actions de l'outil ne sont pas défaites grâce aux propriétés définies dans son cadre formel.

Conclusion

Dans le contexte d'un DSML, la production de l'outillage associé nécessite d'être en grande partie développé de manière *ad-hoc*. Or, on trouve souvent plusieurs DSMLs qui appartiennent au même domaine. Et il est fréquent de trouver souvent les fonctionnalités déjà implantées pour un DSML du même domaine. La réutilisation de fonctionnalités existantes pour des DSMLs proches devrait réduire l'effort nécessaire à produire l'outillage pour un DSML.

Dans cette thèse, nous considérons la réutilisation via la migration des données du DSML vers ce que les outils permettent de prendre en compte. Dans ce type de migration, les données d'un DSML deviennent des données valides par rapport à l'outil à réutiliser. Une fois l'outil réutilisé, les données sont à nouveaux migrées (recontextualisées) pour devenir des données conformes au DSML.

Pour faciliter la mise en place de ce type de réutilisation, nous avons proposé un framework de réutilisation. Ce framework permet de réutiliser un outil dédié dans un contexte différent du contexte où il a été conçu. Un outil dédié ne se concentre que sur les données qui lui sont utiles et nécessaires pour l'exécution de l'algorithme. Dans ce cas, comme dans la réutilisation d'outils du même domaine, le framework garantit que les données d'un DSML sont placées dans le contexte de l'outil dédié (il facilite l'extraction des données utiles) ; que l'outil est appliqué et que le résultat produit par l'outil est placé à nouveau dans le contexte du DSML. La migration des données du DSML vers l'outil est basée sur une approche de co-évolution à base d'opérateurs. Cette solution permet que toutes les données soient automatiquement migrées. Nous proposons de ne pas générer un code de migration modifiable, mais de générer à sa place une *spécification de migration*. Une spécification de migration indique les modifications à appliquer sur chaque élément du modèle afin de le faire migrer. Cette spécification peut être éditée afin de faciliter la définition de migrations spécifiques liées à des éléments spécifiques du modèle. L'édition est contrôlée afin de produire des migrations valides (*i.e.* elle n'autorise pas la suppression d'un élément obligatoire). Une spécification de migration éditée et appliquée plusieurs fois, produit des versions migrées différentes à partir du même modèle. Elle facilite la réutilisation de l'outil en spécialisant son utilisation par modèle.

Dans le contexte de l'orienté-objet, un modèle d'objets est considéré comme un graphe d'objets simple. Cette vision des modèles est une caractéristique commune des frameworks qui gèrent la méta-modélisation orientée objet. Cette structure est bien adaptée pour faire des parcours et des transformations de modèles basées sur un parcours de graphe. C'est pourquoi nous avons défini une sémantique de graphes pour représenter les modèles. La migration d'un modèle devient un ensemble de transformations du graphe d'objets. Cette façon de représenter et de traiter les modèles facilite leur manipulation sans prendre en considération la relation de conformité entre le mo-

dèle et le méta-modèle. Dans notre framework de migration, la migration est gérée par un moteur générique, qui n'a pas à être généré pour chaque migration et qui migre les modèles sans prendre en compte leur méta-modèle.

Si les données sont migrées et l'outil est réutilisé, il reste à mettre le résultat produit par l'outil dans le contexte du DSML pour lequel cet outil est réutilisé. En général, cette migration est vue comme une migration *inverse* simple en réponse à la migration appliquée préalablement. Par exemple, si la migration appliquée implique la *suppression* d'un élément, alors la migration inverse *rajoute* cet élément au résultat produit par l'outil. L'effet de bord de cette façon de voir la migration inverse est que le résultat de l'outil n'a pas de lien avec le *contexte initial* et des éléments qui étaient importants pour le DSML sont perdus. Pour palier à ce problème, nous proposons un mécanisme de *recontextualisation* qui récupère des méta-informations de l'outil. Ces méta-informations permettent de connaitre le lien entre les éléments de l'entrée et de la sortie de l'outil. En fait, la recontextualisation établit des liens entre la sortie de l'outil et les éléments du modèle initial. Notre approche garantie que quand on utilise des opérateurs simples, la recontextualisation ne défait pas les actions réalisées par l'outil et que tous les éléments supprimés lors de la migration sont récupérés. Par contre nous ne pouvons pas garantir que le résultat est sémantiquement correct.

Dans le framework de réutilisation, la majorité des actions effectuées par la migration, la migration inverse et la recontextualisation sont automatisées. Si l'utilisateur veut produire des migrations spécifiques ou s'il veut modifier le résultat de la recontextualisation, il est guidé pour que la réalisation de ces tâches soit plus facile. Mais nous proposons à l'utilisateur des *helpers* qui lui permettent de valider ou de modifier le résultat. L'utilisateur peut alors se concentrer sur les aspects sémantiques, mais le processus de réutilisation est transparent pour lui.

Enfin, deux cas d'étude ont permis d'évaluer l'approche. Le premier permet de réutiliser un outil d'analyse et le deuxième permet de réutiliser des outils de réécriture. Dans les deux cas, nous avons obtenu le résultat attendu. Les lignes de code à développer sont minimales. L'approche réduit la complexité des données à manipuler parce qu'il extrait les données utiles pour l'application de l'outil.

Perspectives

Les travaux présentés dans cette thèse ouvrent la voie à des nombreuses perspectives que nous présentons maintenant.

Jusqu'à aujourd'hui, le méta-modèle cible est généré à partir d'un méta-modèle source *plus* une spécification de refactoring Modif. La spécification de refactoring Modif est elle-même générée automatiquement à partir du méta-modèle source. Nous proposons deux *templates*, l'un utile quand la majorité des éléments du méta-modèle doit être supprimée, et l'autre pour conserver la majorité des éléments. L'utilisateur doit ensuite faire des modifications à la main pour indiquer les opérateurs à appliquer afin de produire le méta-modèle cible. Ces modifications impliquent un effort à l'utilisateur. Une perspective est de faciliter la définition des opérateurs de migration. Une piste est de faire le calcul des opérateurs à appliquer de façon automatique. La spécification de refactoring pourrait par exemple se baser sur une notion de *distance* (Levenshtein) entre les deux méta-modèles. Cette distance est une métrique de

la différence entre deux séquences à l'égard des opérateurs d'édition basiques (*remove*, *rename*) comme dans l'approche présentée par Brun [BP08].

Actuellement, l'approche ne considère pas les contraintes OCL [OCL15]. Si le méta-modèle source ou cible inclut des contraintes, elles sont mises de côté avant de générer la spécification de refactoring. De ce fait, la spécification de migration ne tient pas compte des contraintes. Et les modèles recontextualisés peuvent être invalides du point de vue des contraintes. Une perspective est de prendre en considération ces contraintes afin de produire des modèles qui soient toujours valides. Celle-ci pourrait être atteinte en modifiant la façon de générer la spécification de migration. Elle devrait être générée à partir des opérateurs Modif, et une traduction des contraintes devrait être faite afin de les considérer lors de la migration.

Les travaux en cours sur le framework de réutilisation concernent la production automatique des spécifications de migration pour un nombre restreint d'opérateurs de transformation du langage Modif. Suite aux différentes expérimentations réalisées, nous avons constaté que certains opérateurs qui ne sont pas disponibles dans Modif seraient utiles pour faciliter la transformation d'un méta-modèle. Nous pourrions proposer des opérateurs basiques tels que l'ajout de classe simple (avec constructeur par défaut), l'ajout de références opposées et d'autres macro-opérateurs. L'ajout d'opérateurs implique évidemment de proposer également la migration et la recontextualisation associées. Le round-trip est évolutif, il pourrait être enrichi afin de prendre en charge d'autres langages de co-évolution.

La façon de faire la recontextualisation peut être modifiée en spécialisant le code de recontextualisation ou en éditant directement le modèle produit par la réutilisation. Cette façon de spécialiser la recontextualisation n'est pas adaptée à un utilisateur qui n'a pas l'intérêt ou qui n'a pas l'expertise requise pour modifier le code. Nous travaillons dans implantation d'une interface pour faciliter la personnalisation de la recontextualisation. Elle devrait se rapprocher de l'interface qui permet de faire la spécification de la migration.

Nous envisageons enfin de faciliter l'intégration du framework de réutilisation dans Eclipse. Le prototype est actuellement orienté sur la réutilisation de méthodes de classes Java. Ceci est restrictif, nous envisageons de l'étendre pour réutiliser des outils codés dans d'autres langages.

Publications

Les travaux présentés dans cette thèse ont donnés lieu aux publications suivantes :

Conférence Internationale :

- Paola Vallejo, Mickaël Kerboeuf, Kevin J. M. Martin and Jean-Philippe Babau. Improving Reuse by means of Asymmetrical Model Migrations : An Application to the Orcc Case Study. In *MODELS 2015 – Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems*, Ottawa, Canada, 2015.
- Paola Vallejo, Mickaël Kerboeuf and Jean-Philippe Babau. Specification of Adaptable Model Migrations. In *MODELSWARD 2015 – Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, Angers, France, 2015.

Revue Nationale :

- Paola Vallejo, Mickaël Kerboeuf and Jean-Philippe Babau. Formalisation de la recontextualisation de modèles par graphe de dépendance. In *Numéro spécial de TSI (Technique et Science Informatiques) sur le thème Ingénierie du logiciel*. Numéro n°5/2015, 2015.

Workshop :

- Ahmed Ahmed, Paola Vallejo, Mickaël Kerboeuf and Jean-Philippe Babau CdmCL, a Specific Textual Constraint Language for Common Data Model. In *International Workshop on OCL and Textual Modelling, co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, Valencia, Spain, 2014.
- Paola Vallejo, Mickaël Kerboeuf and Jean-Philippe Babau. Specification of a Legacy Tool by Means of a Dependency Graph to Improve its Reusability. In *7th MoDELS Workshop on Models and Evolution, co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, Miami, United States, 2013.

Congrès national :

- Paola Vallejo, Mickaël Kerboeuf and Jean-Philippe Babau. Graphe de Dépendance pour la Recontextualisation de Modèles. In *Journées Nationales GDR-GPL-CIEL 2014*, Paris, France, 2014.

Poster :

– Paola Vallejo, Mickaël Kerboeuf and Jean-Philippe Babau. Improving Reuse of Tools by means of Model Migrations. In *Journées Nationales GDR-GPL (Génie de la programmation et du logiciel) 2015*, Bordeaux, France, 2015.

Bibliographie

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin : Advanced concepts and tools for in-place emf model transformations. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems : Part I*, MODELS'10, pages 121–135, Oslo, Norway, 2010. Springer-Verlag. 24

[ACE+03] Biju K. Appukuttan, Tony Clark, Andy Evans, Girish Maskeri, Sreedhar Reddy, Paul Sammut, Laurence Tratt, R. Venkatesh, and James S. Willans. QVT-Partners revised submission to QVT RFP, 2003. 17, 51, 54

[AD09] Ali Abou Dib. *Une approche formelle de l'interopérabilité pour une famille de langages dédiés*. PhD thesis, Université Paul Sabatier - Toulouse III, 2009. 1

[AHMM05] David H. Akehurst, Gareth Howells, and Klaus D. McDonald-Maier. Kent model transformation language. In *Proc. Model Transformations in Practice Workshop, part of MoDELS 2005*, Montego Bay, Jamaica, 2005. 49

[AKP03] David Akehurst, Stuart Kent, and Octavian Patrascoiu. A Relational Approach to Defining and Implementing Transformations in Metamodels. *Software and Systems Modeling*, 2(4) :182–196, 2003. 16

[AKS03] Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph transformations on domain-specific models. 2003. 45

[ATL14] Atl. https://eclipse.org/atl/, 2014. 17

[Bé05] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2) :171–188, 2005. 15

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10) :104–116, 1996. 11, 12

[BCBB11] Arnaud Blouin, Benoit Combemale, Benoit Baudry, and Olivier Beaudoux. Modeling Model Slicers. In *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*, volume 6981, pages 62–76, Wellington, New Zealand, 2011. 41, 42, 53

[Ber03] Philip A. Bernstein. Applying model management to classical meta data problems, 2003. 22

[Béz04] Jean Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, 5(2) :21–24, 2004. 14

[BF05] R. Ian Bull and Jean-Marie Favre. Visualization in the context of model driven engineering. In *MDDAUI '05, Model Driven Development of Advanced User Interfaces 2005, Proceedings of the MoDELS'05 Workshop on Model*

Driven Development of Advanced User Interfaces, Montego Bay, Jamaica, 2005. 16

[BJRV04] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture, European MDA Workshops : Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, 2003 and Linköping, Sweden, 2004, Revised Selected Papers*, pages 33–46, 2004. 13

[BK11] Jean-Philippe Babau and Mickael Kerboeuf. Domain Specific Language Modeling Facilities. In *5th MoDELS Workshop on Models and Evolution*, pages 1–6, Wellington, New Zealand, 2011. 45, 50, 53, 109

[BMV+03] Peter Braun, Frank Marschall, Alle Rechte Vorbehalten, Peter Braun, Frank Marschall, and Technische Universität München. Botl – the bidirectional object oriented transformation language. Technical report, 2003. 49

[BP07] Boris Gruschko and Dimitrios S. Kolovos and Richard F. Paige. Towards synchronizing models with evolving metamodels. In *In Proc. Int. Workshop on Model-Driven Software Evolution held with the ECSMR*, 2007. 21, 22, 45

[BP08] Cédric Brun and Alfonso Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 2008. 38, 53, 147

[CDRP09] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. Managing dependent changes in coupled evolution. In RichardF. Paige, editor, *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 35–51. Springer Berlin Heidelberg, 2009. 22, 23, 45, 50, 54

[CFH+09] Krzysztof Czarnecki, J.Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and JamesF. Terwilliger. Bidirectional transformations : A cross-discipline perspective. In *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer Berlin Heidelberg, 2009. 51

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003. 17

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3) :621–645, 2006. 16, 17, 31, 49

[CHM+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and 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*, ASE '02, Washington, DC, USA, 2002. IEEE Computer Society. 49, 54

[CREP08] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object*

Computing Conference, EDOC '08, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society. 22, 24, 44, 45, 54

[CRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9) :165–185, 2007. 40, 53

[Cyb96] Jacob L. Cybulski. Introduction to software reuse. *Department of Information Systems, The University of Melbourne, Parkville, Australia*, 1996. 11, 12

[DFV07] Marcos Didonet Del Fabro and Patrick Valduriez. Semi-automatic model integration using matching transformations and weaving models. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 963–970, New York, NY, USA, 2007. ACM. 23

[DV09] Didonet and Patrick Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and Systems Modeling*, 8(3) :305–324, 2009. 23

[Eda15] Edapt. http://www.eclipse.org/edapt, 2015. 45

[EKK+13] Juergen Etzlstorfer, Angelika Kusel, Elisabeth Kapsammer, Philip Langer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger, and Manuel Wimmer. A survey on incremental model transformation approaches. In *Proceedings of the Workshop on Models and Evolution colocated with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA*, pages 4–13, 2013. 31

[EMF15a] The eclipse modeling framework (emf). http://www.eclipse.org/modeling/emf/, 2015. 1, 17

[EMF15b] Emf migrate. http://www.emfmigrate.org/, 2015. 45

[eMo14] emoflon. http://www.emoflon.org/emoflon/, 2014. 17

[Eps14] Epsilon. http://www.eclipse.org/epsilon/, 2014. 17

[Fav04] Jean-Marie Favre. Cacophony : Metamodel-driven software architecture reconstruction, 2004. 16

[FF95] William B. Frakes and Christopher J. Fox. Sixteen questions about software reuse. *Commun. ACM*, 38(6), 1995. 2

[FHLN08] Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel matching for automatic model transformation generation. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, 2008. Proceedings*, pages 326–340, 2008. 21, 44

[FNM+14] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. Kevoree modeling framework (KMF) : efficient modeling techniques for runtime use. *CoRR*, abs/1405.6817, 2014. 1

[Fre83] Peter Freeman. Reusable software engineering : Concepts and research directions. Los Angeles, California, USA, 1983. 11

[Fuj15] http://www.fujaba.de/, 2015. 49

[GCD+12] Clément Guy, Benoit Combemale, Steven Derrien, Jim Steel, and Jean-Marc Jézéquel. On Model Subtyping. In *ECMFA - 8th European Conference on Modelling Foundations and Applications*, Kgs. Lyngby, Denmark, 2012. 30

[GdLKP10] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, and Richard F. Paige. A visual specification language for model-to-model transformations. In *VL/HCC*, pages 119–126, 2010. 49

[GGKH03] T. Gardner, C. Griffin, J. Koehler, and R. Hauser. Review of omg mof 2.0 query/views/transformations submissions and recommendations towards final standard, 2003. 17

[GJCB09] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing Model Adaptation by Precise Detection of Metamodel Changes. In *In Proc. of ECMDA 2009*, pages 34–49, Enschede, Netherlands, 2009. Springer. 24, 44, 54

[GMPS03] M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. A language for bi-directional tree transformations, 2003. 51

[Gra01] I. Graham. *Object-Oriented Methods. Principles and Practice*. Harlow, England : Addison-Wesley, an imprint of Pearson Education, third edition edition, 2001. 12

[GRe15] Graph rewriting and transformation language. http://www.isis.vanderbilt.edu/tools/great, 2015. 49

[Guy13] Clément Guy. *Typing facilities for language engineering*. Theses, Université Rennes 1, 2013. 30

[Has11] Kahina Hassam. *Adaptation des contraintes OCL lors de l'évolution des métamodèles et modèles*. PhD thesis, 2011. Thèse de doctorat dirigée par Sadou, Salah STIC Lorient 2011. 21

[HBJ09] Markus Herrmannsdöerfer, Sebastian Benz, and Elmar Juergens. Cope - automating coupled evolution of metamodels and models. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 52–76, Italy, 2009. Springer-Verlag. 24, 54

[Hem92] T. Hemmann. Reuse in software and knowledge engineering. National Research Center for Computer Science, 1992. 12

[Hen14] Henshin. https://www.eclipse.org/henshin/, 2014. 17

[Heu04] W. Heuvel. Matching and adaptation. core techniques for mda-(adm)-driven integration of new business. In *Proceeding of MELS Workshop (EDOC). IEEE Press*, 2004. 40

[HK10] Markus Herrmannsdöerfer and Maximilian Koegel. Towards semantics-preserving model migration. In *International Workshop on Models and Evolution*, 2010. 24, 45

[HR10] Markus Herrmannsdöerfer and Daniel Ratiu. Limitations of automating model migration in response to metamodel adaptation. In *Proceedings of the 2009 International Conference on Models in Software Engineering*, MODELS'09, pages 205–219, Denver, Colorado USA, 2010. Springer-Verlag. 24

[Hud97] Paul Hudak. Domain-specific languages. *Handbook of Programming Languages*, 3 :39–60, 1997. 21

[HVW10] Markus Herrmannsdörfer, Sander Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, 2010, Revised Selected Papers*, volume 6563 of *Lecture Notes in Computer Science*, pages 163–182. Springer, 2010. 17, 24, 50

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, Waikiki, Honolulu, HI, USA, 2011. ACM. 20

[IK04] Igor Ivkovic and Kostas Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society. 16

[Jam15] Jamda. http://jamda.sourceforge.net/docs/api/, 2015. 17

[JCV12] Jean-Marc Jézéquel, Benoit Combemale, and Didier Vojtisek. *Ingénierie Dirigée par les Modèles : des concepts à la pratique...* Références sciences. Ellipses, 2012. 13, 15

[JK06a] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1188–1195, Dijon, France, 2006. ACM. 44

[JK06b] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, pages 128–138, Montego Bay, Jamaica, 2006. Springer-Verlag. 44

[Kad05] H. Kadima. *MDA : Conception orientée objet guidée par les modèles*. Dunod, 2005. 1, 13

[KB11] Mickaël Kerboeuf and Jean-Philippe Babau. A DSML for reversible transformations. In *11th OOPSLA Workshop on Domain-Specific Modeling*, pages 1–6, Portland, United States, 2011. 45, 52

[Ker14] Kermeta. http://www.kermeta.org/examples/, 2014. 17

[Kle08] Anneke G. Kleppe. *Software language engineering : creating domain-specific languages using metamodels*. University of Grenoble France, 2008. 20

[Kru89] C. Krueger. *Models of reuse in software engineering*. School of computer science. Carnegie Mellon University, 1989. 12

[Kuh06] Thomas Kuhne. Matters of (meta-) modeling. *Software and Systems Modeling*, 5(4) :369–385, 2006. 14

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained : The Model Driven Architecture : Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 15, 16

[Lä04] Ralf Lämmel. Coupled software transformations (extended abstract), 2004. 43

[LBNK10] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan, and Gabor Karsai. A novel approach to semi-automated evolution of dsml model transformation. In *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 23–41. Springer Berlin Heidelberg, 2010. 21, 43

[Lim94] Wayne C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Softw.*, 11(5) :23–30, 1994. 12

[LV02] Juan de Lara and Hans Vangheluwe. Atom3 : A tool for multi-formalism and meta-modelling. In *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2002. 49

[LWG+12] Philip Langer, Manuel Wimmer, Jeff Gray, Gerti Kappel, and Antonio Vallecillo. Language-specific model versioning based on signifiers. *Journal of Object Technology*, 11(3) :4 : 1–34, 2012. 52

[Man09] Giulio Manzonetto. Models and theories of lambda calculus. *CoRR*, abs/0904.4756, 2009. 15

[MEMC10] David Mendez, Anne Etien, Alexis Muller, and Rubby Casallas. Towards Transformation Migration After Metamodel Evolution. In *Model and Evolution Workshop*, Model and Evolution Workshop, Olso, Norway, 2010. 21, 29

[Met14] Meta object facility (MOF) 2.4.2 core specification, 2014. Version 2.4.2. 14

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 264–278, Montego Bay, Jamaica, 2005. Springer-Verlag. 49

[MOL15] Mola. http://mola.mii.lu.lv/, 2015. 49

[MTF15] http://www.ibm.com/developerworks/rational/library/05/503_sebas/, 2015. 17, 49

[MVG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152 :125–142, 2006. 16

[NLBK09] Anantha Narayanan, Tihamer Levendovszky, Daniel Balasubramanian, and Gabor Karsai. Automatic domain model migration to manage metamodel evolution. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 706–711. Springer Berlin Heidelberg, 2009. 43, 53, 54

[OADFP08] Ileana Ober, Ali Abou Dib, Louis Féraud, and Christian Percebois. Towards interoperability in component based development with a family of DSLs. In R. Morrison, D. Balasubramaniam, and K. Falkner, editors, *European Conference on Software Architecture (ECSA), Chypre, 29/09/2008-01/10/2008*, volume LNCS, pages 148–163. Springer-Verlag, 2008. 1

[OCL15] Object managment group. object constraint language. http://www.omg.org/spec/OCL/, 2015. 147

[OMG15] Object managment group. http://www.omg.org/, 2015. 14

[Pae15] Patrik Paetau. Object-oriented reuse – on the benefits and problems object-oriented software reuse. 2015. 11, 12

[Pat04] Octavian Patrascoiu. Yatl : Yet another transformation language. In *University of Twente, the Nederlands*, pages 83–90, 2004. 49

[PD89] R. Prieto-Diaz. Software reusability : Vol. 1, concepts and models. chapter Classification of Reusable Modules, pages 99–123. ACM, New York, NY, USA, 1989. 11

[Pha12] Quyet Thang Pham. *Model Transformation Reuse : A Graph-based Model Typing Approach*. PhD thesis, Institut Mines-Télécom-Télécom, 2012. 32, 42

[Pla15] Platypus. http://dossen.univ-brest.fr/apl/index.php/Platypus, 2015. 1

[QVT15] QVT. http://www.omg.org/spec/QVT/1.1/, 2015. 17

[Rad99] A. Radding. Fast track to app success. In *InformationWeek*, Manhasset, New York, USA, 1999. 11

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4) :334–350, 2001. 23

[Ref15] Refactory. http://www.modelrefactoring.org/index.php/Refactoring, 2015. 45

[RKP+14] LouisM. Rose, DimitriosS. Kolovos, RichardF. Paige, Fiona A. C. Polack, and Simon Poulding. Epsilon flock : a model migration language. *Software and Systems Modeling*, 2014. 21

[RKPP10] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model migration with epsilon flock. In *Proceedings of the 3rd ICMT conference*, pages 184–198. Springer, 2010. 23, 40, 53, 54

[RMvH11] Ulf Rüegg, Christian Motika, and Reinhard von Hanxleden. Interactive transformations for visual models. In *3rd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2011) at conference INFORMATIK 2011*, GI-Edition – Lecture Notes in Informatics (LNI), Berlin, Germany, 2011. Bonner Köllen Verlag. 45, 54

[RPKP09] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. An analysis of approaches to model migration. In *Proc. Models and Evolution (MoDSE-MCCM) Workshop, 12th ACM/IEEE International Conference on Model Driven Engineering, Languages and Systems*, 2009. 23, 49

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. 15

[SE05] Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. In Stefano Spaccapietra, editor, *Journal on Data Semantics IV*,

volume 3730 of *Lecture Notes in Computer Science*, pages 146–171. Springer Berlin Heidelberg, 2005. 23

[Sei03] Ed Seidewitz. What models mean. *IEEE Softw.*, 20(5) :26–32, 2003. 14

[SF05] Arnor Solberg and Robert France. Navigating the metamuddle. In *Proceedings of the 4th Workshop in Software Model Engoneering*, 2005. 16, 41

[SFS05] Raul Silaghi, Frédéric Fondement, and Alfred Strohmeier. "weaving" mtl model transformations. In *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*, pages 123–138. Springer Berlin Heidelberg, 2005. 49

[SG08] Maik Schmidt and Tilman Gloetzner. Constructing difference tools for models using the sidiff framework. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 947–948, Leipzig, Germany, 2008. ACM. 39, 53

[SJ07] Jim Steel and Jean-Marc Jézéquel. On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4) :401–414, 2007. 30

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation : The heart and soul of model-driven software development. *IEEE Softw.*, 20(5) :42–45, 2003. 15, 16

[SK04] Jonathan Sprinkle and Gabor Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3) :291–307, 2004. 22, 44, 53, 54

[SMBJ09] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel. Meta-model pruning. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2009. 41, 53

[Sou12] T. B. Sousa. Dataflow programming concept, languages and applications. In *Doctoral Symposium on Informatics Engineering*, 2012. 130

[SPLTJ01] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In *Proceedings of UML 2001*, volume 2185 of *LNCS*, pages 134–148. Springer Verlag, 2001. 16

[Ste08] Perdita Stevens. A landscape of bidirectional model transformations. In *Generative and transformational techniques in software engineering II*, pages 408–424. Springer Berlin Heidelberg, 2008. 51

[Tef15] http://tefkat.sourceforge.net/, 2015. 17, 49

[Tra88] W. Tracz. Software reuse : Motivators and inhibitors. *Digest of Papers COMPCON. Computer Society Press of the IEEE*, (764) :358 – 363, 1988. 12

[UML14] Object managment group. uml infrastructure specification. http://www.omg.org/spec/UML/2.4.1, 2014. 1

[UML15] Umlx. https://www.eclipse.org/gmt/umlx/, 2015. 49

[VB07] Dániel Varró and András Balogh. The model transformation language of the {VIATRA2} framework. *Science of Computer Programming*, 68(3) :214 – 234, 2007. Special Issue on Model Transformation. 49

[Wac07] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *Proceedings of ECOOP*, pages 600–624. Springer-Verlag, 2007. 21, 22, 24, 44, 45, 50

[Wid11] A. Wider. Towards lenses for view synchronization in metamodel-based domain-specific workbenches. In *In : 3rd Workshop 'Methodische Entwicklung von Modellierungswerkzeugen' at INFORMATIK 2011*, Berlin, Germany, 2011. Proceedings, GI-Edition of Lecture Notes in Informatics (LNI) (2010). 51

[Wip10] Matthieu Wipliez. *Compilation infrastructure for dataflow programs*. phd thesis, INSA de Rennes, 2010. 133

[WIRP12] Dennis Wagelaar, Ludovico Iovino, Davide Di Ruscio, and Alfonso Pierantonio. Translational semantics of a co-evolution specific language with the EMF transformation virtual machine. In *Theory and Practice of Model Transformations - 5th International Conference, ICMT 2012, Prague, Czech Republic, May 28-29, 2012. Proceedings*, pages 192–207, 2012. 45, 54

[XG03] Y. Xia and M. Glinz. Rigorous EBNF-based definition for a graphic modeling language. In *Software Engineering Conference, 2003. Tenth Asia-Pacific*, pages 186–196, 2003. 15

[XML15] Omg formal versions of xml. http://www.omg.org/spec/XML/, 2015. 15

[XS05] Zhenchang Xing and Eleni Stroulia. Umldiff : An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, Long Beach, CA, USA, 2005. ACM. 39, 53

[Xte15] Xtext. https://eclipse.org/Xtext/, 2015. 17

[YCWR11] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet. Efficient multicore scheduling of dataflow process networks. In *Signal Processing Systems (SIPS), 2011 IEEE Workshop*, Lebanon, 2011. 140

[ZB14] Vadim Zaytsev and AnyaHelene Bagge. Parsing in a broad sense. In *Model-Driven Engineering Languages and Systems*, volume 8767 of *LNCS*, pages 50–67. Springer International Publishing, 2014. 129

[ZLG05] Jing Zhang, Yuehua Lin, and Jeff Gray. Generic and domain-specific model refactoring using a model transformation engine. In *Model-Driven Software Development*, pages 199–217. Springer Berlin Heidelberg, 2005. 16