



HAL
open science

Model Transformation Reuse: A Graph-based Model Typing Approach

Quyêt Thang Pham

► **To cite this version:**

Quyêt Thang Pham. Model Transformation Reuse: A Graph-based Model Typing Approach. Software Engineering [cs.SE]. Télécom Bretagne, Université de Rennes 1, 2012. English. NNT : . tel-00816982

HAL Id: tel-00816982

<https://theses.hal.science/tel-00816982>

Submitted on 23 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Sous le sceau de l'Université européenne de Bretagne

Télécom Bretagne

En habilitation conjointe avec l'Université de Rennes 1

Ecole Doctorale – MATISSE

Model Transformation Reuse: A Graph-based Model Typing Approach

Thèse de Doctorat

Mention : Informatique

Présentée par **Quyêt Thang – PHAM**

Département : Informatique

Equipe : PASS

Directeur de thèse : Antoine Beugnard

Soutenue le 19/12/2012

Jury :

M. Jean-Philippe Babau, Professeur, UFR Sciences et Techniques - Brest	(Président/Rapporteur)
M. Marc Pantel, Maître de conférences, ENSEEIHT - Toulouse	(Rapporteur)
M. Benoit Baudry, Chargé de Recherche, INRIA - Rennes	(Examineur)
M. Jordi Cabot, Maître Assistant, Ecole de Mines - Nantes	(Examineur)
M. Antoine Beugnard, Professeur, Télécom Bretagne	(Directeur de thèse)

Acknowledgements

I would like to thank all people that supported me during of my PhD study and writing of this thesis :

- First of all, I would like to sincerely thank my supervisor Prof. Dr. Antoine Beugnard for giving me the opportunity to learn about research in the field of Model Driven Engineering. This thesis would not have been finished without his guidance, support and motivation at every step throughout my study.
- I want to thank Prof. Dr. Jean-Philippe Babau and Dr. Marc Pantel for spending time to review my thesis. Their comments and suggestions helped me to significantly improve my thesis. I would also like to thank my other thesis committee members, Dr. Benoit Baudry and Dr. Jordi Cabot for examining my thesis.
- Special thanks go to my colleagues from the Processes for Adaptive Software Systems group in Département Informatique (in alphabetical ordering) Ali Hassan, Aladin El Hamdouni, Jean-Baptiste Lézoray, An Phung-Khac for interesting discussions, and particularly to Fahad Rafique Golra for checking English of the manuscript. Thanks to everyone in the department, particularly to Armelle Lannuzel who helped me a lot on paperwork.
- I gratefully thank all my friends for sharing their leisure with me. Their support, both direct or indirect, and encouragement had significant influence on this thesis.
- Finally, I want to thank my parents, and my two sisters. They are always supported and encouraged me to do my best in every moment of my life.

Abstract

Identical domain concepts reified in different (meta)modelling projects may be named, represented and connected differently. It turns out that a transformation defined for a particular metamodel cannot be directly used for another metamodel; that is, the reuse of transformations is restricted. To tackle this problem, in this dissertation, we propose a solution for automatically migrating legacy transformations. Such a transformation is adapted to the new metamodel that has a slightly different representation in comparison with the original one, while preserving the original semantics of the transformation. To this end, we first introduce MetaModMap, a Domain Specific Language that allows the description of the correspondences of intended semantics between the elements of two metamodels that model the same domain. Then we provide a rewriting mechanism using these user-defined correspondences to migrate the transformation automatically. The proposed solution uses a graph-based model typing relation that enables safe adaptations. Our approach has been prototyped with MOMENT2 and can be used with any framework based on the same graph transformation paradigm.

Keywords : Transformation Reuse, Model Typing, Model-Driven Engineering

Résumé

Pour réaliser un modèle dans un domaine métier, les mêmes concepts peuvent être traduits, nommés, représentés ou reliés différemment dans des méta-modèles différents. Ainsi, une transformation définie pour un méta-modèle particulier ne peut pas être utilisée pour un autre méta-modèle. La réutilisation des transformations est donc limitée. Face à ce problème, nous proposons dans cette thèse une solution pour migrer automatiquement les transformations existantes pour pouvoir les appliquer à un autre méta-modèle. Une telle transformation est adaptée pour le nouveau méta-modèle qui a une représentation légèrement différente par rapport à celle d'origine, tout en préservant la sémantique de la transformation. À cette fin, nous introduisons d'abord MetaModMap, un langage spécifique qui permet de décrire des correspondances de la sémantique intentionnelle entre les éléments de deux méta-modèles qui modélisent le même domaine. Ensuite, nous proposons un mécanisme de réécriture pour migrer automatiquement la transformation en utilisant ces correspondances définies par l'utilisateur. La solution proposée utilise une relation de typage de modèle basée sur des graphes qui permet de faire des adaptations en toute sécurité. Notre approche a été prototypée avec MOMENT2 et peut être utilisée pour d'autres frameworks basés sur le même paradigme de transformation de modèles basé sur les graphes.

Mots-clés : Réutilisation de Transformation, Typage de Modèle, Ingénierie Dirigée par les Modèles

Contents

Résumé en français	1
1 Introduction	1
2 Etude de Cas	2
2.1 Besoin de Réutilisation de Transformation	2
2.2 Choix de Langage de Transformation	3
2.3 Discussion	3
3 Approche de Typage de Modèle basée sur les Graphes	4
3.1 Graphe de Type avec Multiplicité	4
3.2 Inférence des TGMs Effectives	6
3.3 Un DSL pour la Description des Correspondances	7
3.4 Adaptation de Transformation	9
4 Adaptation de Transformation en Action	11
4.1 Prototype de MetaModMap	12
4.2 Exemple de Validation	12
5 Travaux Connexes	13
6 Conclusion	15
1 Introduction	1
1.1 General Context	1
1.2 Problems and Challenge	2
1.3 Objective, Approach, and Contributions	3
1.4 Structure of Thesis	5
2 State of the Art	7
2.1 Chapter Overview	7

2.2	Concepts in Model-Driven Engineering	8
2.2.1	Model-Driven Engineering	8
2.2.2	Model	10
2.2.2.1	Token models	11
2.2.2.2	Type models	11
2.2.3	Metamodel	12
2.2.4	Meta-metamodel	13
2.2.5	The MOF Metamodelling Architecture	14
2.2.6	Domain Specific Modelling Language	16
2.2.7	Model Transformation	22
2.3	Model Transformation Reuse	34
2.3.1	A Case Study	35
2.3.2	Choice of Transformation Language	41
2.3.3	Discussion	51
2.4	Related Work	56
2.4.1	Transformation adaptation	58
2.4.2	Model adaptation	61
2.4.3	Metamodel adaptation	63
2.4.4	Categorization Overview	65
2.5	State-of-the-Art Summary	65
3	A Graph-Based Model Typing Approach	67
3.1	Chapter Overview	67
3.2	Graph Topology	68
3.2.1	Graph Topology in Metamodels	69
3.2.2	Graph Topology in Graph-based Transformations	71
3.3	Typing Models and Transformations with Graphs	74
3.3.1	Type Graph with Multiplicity as Model Type	74
3.3.2	Model Typing	76
3.3.3	Model Sub-typing	83
3.3.4	Model Transformation Typing	87
3.4	Non-isomorphic Transformation Reuse	96
3.4.1	DSL for Correspondence Description	98

3.4.2	Transformation Adaptation	109
3.5	Summary and Discussion	120
4	MetaModMap: A Transformation Migration Toolkit	123
4.1	Chapter Overview	123
4.2	Architecture of MetaModMap Toolkit	124
4.2.1	Eclipse Plug-in Platform	126
4.2.2	Language Development with XTEXT	134
4.3	Realization of MetaModMap Toolkit	140
4.3.1	XTEXT-based Editor for MetaModMap	141
4.3.2	Higher-Order Transformation	147
4.4	Transformation Adaptation in Action	148
4.4.1	MetaModMap Prototype	149
4.4.2	Customer Management Case Study	149
4.5	Comparison to Transformation Reuse Approaches	151
4.6	Prototype Summary and Discussion	155
5	Conclusion	157
5.1	Contribution Summary	157
5.2	Limitations	160
5.3	Perspectives	161
	Bibliography	162
	Figures	172
	Tables	176
	Listings	178
	Appendices	181
A	Transformation Definitions	182
1.1	Transformation for CDV1 and RDB metamodels	183
1.2	Adjusted Transformation from CD2RDB	192

B MetaModMap Implementation Detail	201
2.1 A Java implementation for the <i>Trace</i> interface	201

Résumé en français

Sommaire

1	Introduction	1
2	Etude de Cas	2
3	Approche de Typage de Modèle basée sur les Graphes . .	4
4	Adaptation de Transformation en Action	11
5	Travaux Connexes	13
6	Conclusion	15

1 Introduction

Dans l'Ingénierie Dirigée par les Modèles, une question essentielle est la possibilité de réutiliser les transformations à travers des outils de modélisation. La réutilisation de transformation est particulièrement nécessaire lorsque le même domaine peut être modélisé différemment, ou lorsqu'il peut changer légèrement au fil du temps. Dans ce travail, la réutilisation de transformation est obtenue quand une transformation existante écrite pour un méta-modèle (MM) peut être utilisée systématiquement pour un MM similaire. Les MMs sont similaires si on peut présenter une correspondance entre leurs différentes représentations du même concept de modélisation.

Il existe deux stratégies principales permettant la réutilisation des transformations par rapport à l'évolution des MMs : *l'adaptation de modèle* et *l'adaptation de transformation*. Dans la première stratégie [Kerboeuf 11], une transformation supplémentaire est développée afin d'adapter les modèles du nouveau MM à conformer au MM original et alors elle est composée avec la transformation existante en une chaîne. Dans la deuxième stratégie, la transformation existante à son tour est adaptée pour être valide sur le nouveau MM. Cette stratégie s'appuie sur la co-évolution métamodèle/transformation comme dans [Levendovszky 10]. Après une évolution, une transformation pourrait devenir incompatible à la nouvelle version du MM en raison de la modification du nom des éléments, de la valeur de multiplicité ou du remplacement d'une relation par un nouveau type, etc.

Dans ce travail, nous proposons un processus d'adaptation à des transformations définies basées sur la réécriture de graphes pour des MMs semblables. Un DSL, à savoir MetaModMap, est introduit pour décrire les correspondances intentionnelles

entre des différentes représentations du même concept de modélisation. Ces correspondances sont utilisées pour vérifier la substituabilité de MMs en utilisant un *a priori type-check* basé sur la nature graphique coïncidente entre des MMs et des transformations. À la fin, elles sont utilisées pour effectuer l'adaptation de transformation.

Le reste de ce résumé est organisé de la façon suivante. Section 2 présente une étude de cas. Section 3 décrit notre approche globale qui a été implémentée dans un prototype. Section 4 illustre l'approche en utilisant ce prototype. Section 5 discute des travaux connexes. Section 6 conclut et anticipe les travaux futurs.

2 Etude de Cas

2.1 Besoin de Réutilisation de Transformation

Une transformation dans laquelle les modèles de classe sont transformés en des modèles relationnels (RDB) est considérée comme l'étude de cas. Le MM de source, à savoir CDV1, d'un outil CASE concevant des modèles de classe et le MM de cible pour les modèles de RDB sont respectivement présentés dans la figure 1(a) et la figure 1(c).

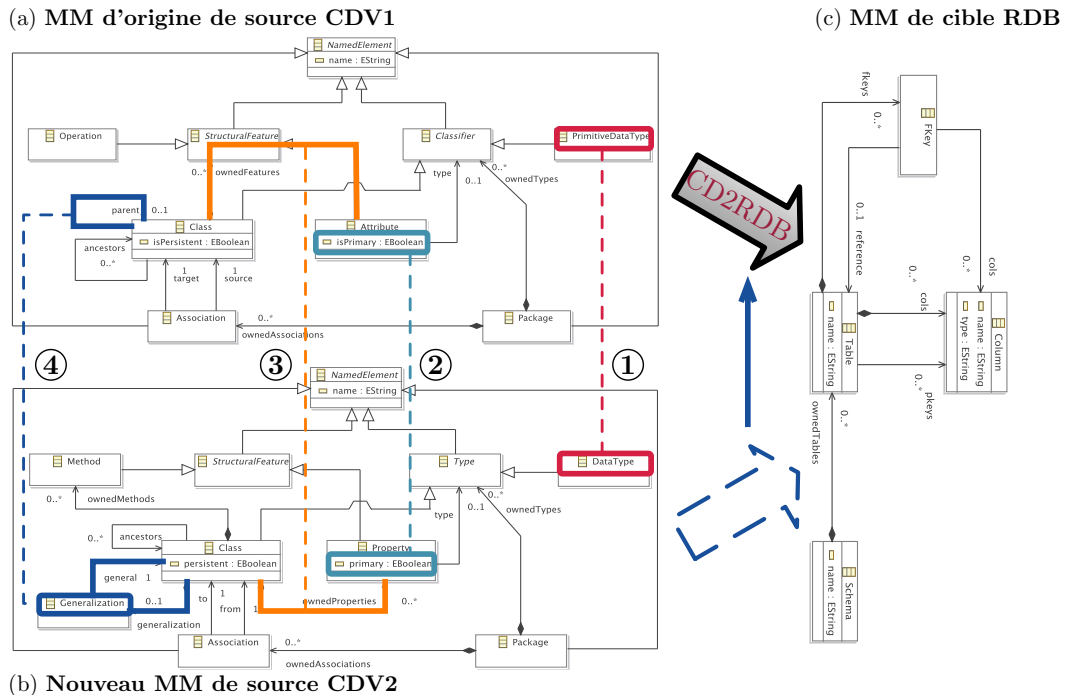


FIGURE 1 – Métamodèles de la transformation CD2RDB.

Si on veut transformer les modèles de classe conçus par un autre outil CASE basé sur un nouveau MM, à savoir CDV2, cf. FIGURE 1(b), la transformation devient incompatible avec le nouveau MM de source, il devrait ainsi être adaptée.

L’automatisation de cette tâche est significative pour promouvoir la réutilisation de transformation et pour éviter une adaptation manuelle fastidieuse et “error-prone”.

2.2 Choix de Langage de Transformation

La transformation CD2RDB est écrite en MOMENT2¹, un langage de transformation à base de graphes [Boronat 09]. Une transformation en MOMENT2 est donnée comme un ensemble d’équations de modèle qui contiennent des patterns de graphe. Nous avons choisi MOMENT2 pour l’implantation² car il fournit une manière de définir des transformations à un haut niveau d’abstraction. Puisque nous considérons que la similitude de MMs comme une similitude de graphes, des frameworks à base de graphes sont plus pertinents que d’autres paradigmes pour étudier la substituabilité de MMs. Un extrait de la transformation est indiqué dans la FIGURE 2.

```

1  transformation CD2RDB (cd : CDV1; rdb : RDB) {
2    rl InitAncestor {
3      nac cd noAncestor {
4        ...
5      };
6      lhs {
7        cd {
8          ...
9        }
10     };
11     rhs {
12     cd {
13       c1 : Class {
14         parent = c2 : Class {},
15         ancestors = c2 : Class {}
16       }
17     }
18   };
19 }
20 ...

21 rl DataTypeAttributeOfTopClass2Column {
22   nac rdb noColumn {
23     ...
24   };
25   lhs {
26     cd {
27       c : Class {
28         name = cname,
29         ownedFeatures = a : Attribute {
30           name = aname,
31           type = p : PrimitiveDataType {
32             name = pname
33           }
34         }
35       }
36     }
37     ...
38   };
39   rhs {
40     ...
41   };
42 }

```

FIGURE 2 – Extrait du fichier CD2RDB.mgt.

2.3 Discussion

Lorsque le MM de source est changé, certains types, attributs, ou références référés par la transformation ne peut être trouvés dans le nouveau MM en raison de la différence de nom, cf. (1) (3) (4) dans la FIGURE 1(a) (b). En outre, la relation *parent* est “réifiée” par la présence du type *Généralisation*, cf. (2), ce qui rend les patterns existants utilisant cette relation deviennent incompatibles. Ces incompatibilités limitent l’utilisation de la transformation pour les modèles de classe du nouveau MM, même si l’intention du *metamodeler* sur les concepts de modélisation est en fait la même chose. Cela suggère un besoin de décrire manuellement les correspondances de l’intention du *metamodeler* entre les éléments de MMs. Ces correspondances peuvent

1. www.cs.le.ac.uk/people/aboronat/tools/moment2-gt/
2. disponible à perso.telecom-bretagne.eu/quyetpham/software/

alors être utilisées comme directives pour adapter la transformation en réponse à des différences de MMs tout en préservant l'intention originale de la transformation.

3 Approche de Typage de Modèle basée sur les Graphes

Cette section présente globalement l'approche de réutiliser les transformations alors que seuls les MMs de source sont changés. En particulier, pour réutiliser une telle transformation, les quatre étapes suivantes doivent être effectuées :

1. Construire deux points de vue graphique, Graphe de Type avec Multiplicité (TGM), pour le MM d'origine et de nouveau.
2. Dédire des parties *effectives* du TGM d'origine qui sont effectivement utilisées par la transformation.
3. Décrire les correspondances entre les éléments de deux MMs à l'aide de notre DSL. Cette tâche est assistée par un *type-checker* à l'aide des résultats des étapes 1 et 2.
4. Exécuter un interpréteur qui adapte la transformation pour que cela fonctionne avec le nouveau MM de source.

Étape 3 nécessite une saisie manuelle, les autres sont automatisées par l'outil. Les sections suivantes donnent des détails sur ces étapes.

3.1 Graphe de Type avec Multiplicité

Les MMs sont conçus de manière concise en utilisant le *héritage* fournit en Ecore, comme le montre FIGURE 1(a) (b). Cependant, cette représentation ne désigne pas explicitement la topologie du graphe de MMs. Comme des frameworks à base de graphe, par exemple MOMENT2, utilisent la nature de graphe de MMs pour définir les transformations, des MMs basés sur Ecore ont besoin d'exhiber leur topologie.

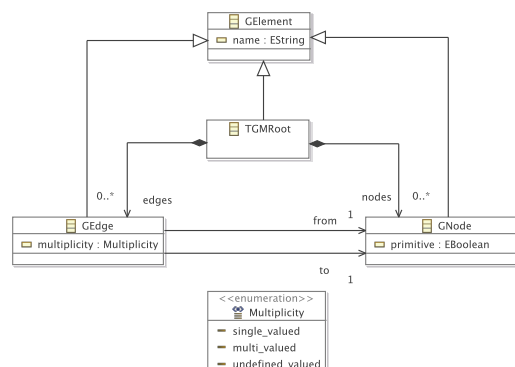


FIGURE 3 – Métamodèle de TGMs.

Pour ce faire, nous introduisons la vue Graphe de Type avec Multiplicité (TGM). Cette abstraction, cf. FIGURE 3, est utilisée pour représenter la topologie de graphe de MMs. Comme mentionné précédemment, les TGMs de MMs sont générés automatiquement selon les règles suivantes :

1. Les types de données de base sont transformés en *GNodes* avec les mêmes noms et la valeur de l'attribut *primitive* est mise à *vrai*.
2. Chaque type défini par l'utilisateur est transformé en un *GNode* avec le même nom et la valeur de l'attribut *primitive* est mise à *faux*.
3. Les attributs et les références sont transformées en *GEdges* en tenant compte transitivement des *héritages*.
4. Pour les références, si le méta-attribut *upperBound* est > 1 ou $= -1$, l'attribut de *multiplicity* correspondant à un *GEdge* est alors pris égal à la valeur *multi_valued* (dénote *), sinon il prend la valeur *single_valued* (dénote 1).

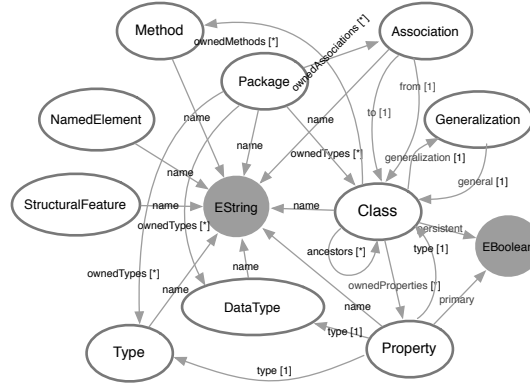


FIGURE 4 – TGM du nouveau MM CDV2.

FIGURE 4 montre le TGM de CDV2. On peut utiliser n'importe quel sous-graphe d'un TGM pour définir des patterns. Généralement, une transformation utilise seulement un sous-graphe plutôt que tous les éléments du graphe, i.e. certains éléments de MM sont ignorés. Cela nous amène à déduire des parties *effectives* d'un TGM qui sont réellement utilisées par la transformation à l'étape 2.

Grâce à l'inclusion entre des graphes, nous pouvons utiliser des TGMs comme des types de modèles d'où une approche de sous-typage est dérivé, comme illustré dans FIGURE 5. Laissez TGM1 et TGM2 être les TGMs de MM1 et MM2, respectivement, cf. (1), et TGM_{eff} est un TGM *effectif* déduit de TGM1 et de la transformation, cf. (2). Nous avons :

$$TGM_{eff} \subseteq_m TGM2 \Leftrightarrow MM2 \preceq MM1 \quad (1)$$

par une cartographie m entre les éléments de MM1, MM2. La relation de sous-typage (\preceq) entre MMs est *partielle* puisque le contexte d'utilisation (i.e. la transformation) d'un type de modèle doit être pris en considération. En outre, il est *non-isomorphe* car elle doit être dérivée par une correspondance entre les MMs.

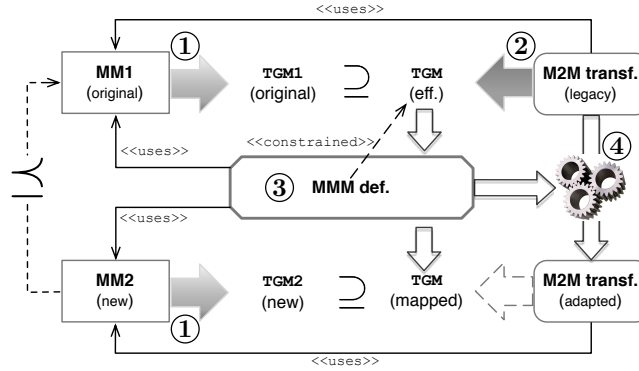


FIGURE 5 – Approche de sous-typage.

3.2 Inférence des TGMs Effectives

Les paragraphes suivants décrivent l’algorithme pour déduire un TGM *effectif* en analysant les patterns d’une transformation.

Le principe est de ne conserver que $GNode(s)$ et $GEdge(s)$ correspondant à des éléments du MM qui sont référés par des patterns, voir FIGURE 8 (en bas à gauche). Initialement, tous les *patterns* sont collectés dans une liste. L’algorithme de visite pour chaque pattern est réalisée de manière récursive, à partir du pattern d’objet racine. A chaque pattern d’objet visité, le $GNode$ correspondant à un type référé sera ajouté à un *ensemble* de noeuds nécessaire. Tous les $GEdge(s)$ correspondant à des attributs ou des références référés de patterns de propriété doit être ajoutés dans un *ensemble* d’arcs nécessaire. Lors qu’une visite un pattern de propriété qui réfère à une référence, nous visitons récursivement le pattern d’objet inférieur. En fin, nous obtenons les *ensembles* de noeuds et d’arcs nécessaires, qui est un sous-graphe du TGM d’origine. Pour compléter ce processus, tous les noeuds et les arcs qui ne sont pas référés par les patterns sont enlevés pour obtenir le TGM *effectif*.

Les contraintes sur *Multiplicity* nécessitent un traitement spécial. Nous observons que un pattern défini en tant que un pattern singulier (par exemple entre deux types A et B, e.g. $o1 : A \{refB = o2 : B \{..\}\}$) peut être appliqué aux relations singulier (= 1) ou multiple (= *). Au contraire, lorsque des patterns multiples (par exemple, $o1 : A \{refB = o2 : B \{..}, refB = o3 : B \{..}\}$) sont définis, que relations multiple peuvent s’appliquer. Cela conduit à l’ordre où *undefined_valued* (dénote?) est le cas le plus général :

$$undefined_valued \preceq single_valued \preceq multi_valued \quad (2)$$

Cet ordre résulte dans deux cas. Dans le premier cas où une relation singulier est définie dans le MM d’origine, nous ne changeons pas la valeur de *multiplicity* dans le TGM *effectif* puisque les relations multiples peuvent également être appliquées. Dans le deuxième cas où une relation multiple est présente, on peut définir à la fois un pattern singulier et multiple. Lorsqu’on utilise un pattern singulier, nous changeons la

valeur de *multiplicity* de *multi_valued* à *undefined_valued* car tous les cas ne peuvent pas être appliqués, sinon, la valeur de *multiplicity* est laissée à *multi_valued* puisque seulement les relations multiples peuvent être appliquées.

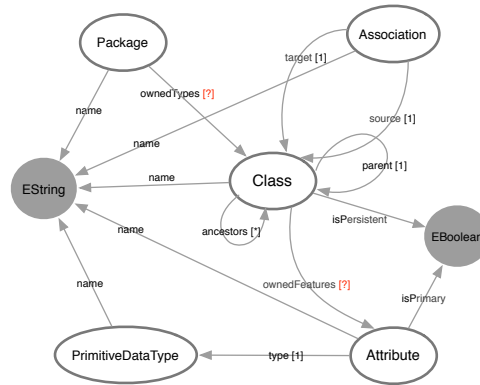


FIGURE 6 – TGM effectif de CDV1.

Le TGM *effectif* déduit de CDV1 est donné dans FIGURE 6. Le TGM *effectif* est utilisé comme une contrainte lors de la description des correspondances entre les éléments de MMs dans notre DSL. Plus de détails sont donnés dans la section suivante.

3.3 Un DSL pour la Description des Correspondances

Pour résoudre le problème mentionné dans la section 2.3, nous proposons un langage dédié, appelé MetaModMap, pour le but d'exposer les correspondances entre les éléments de MMs. Ensuite, nous fournissons un interpréteur qui est en charge de la génération automatique d'une transformation existante à une nouvelle transformation adaptée.

Pour l'étude de cas, les sortes suivantes de correspondance sont nécessaires : trois liés au changement de noms, l'un sur la *réification*. La cartographie établit les correspondances de nommage de type 1 : 1 pour chaque type, attribut et référence requise dans le MM d'origine à, respectivement, un type, un attribut et une référence dans le nouveau MM, cf. (1) (4) (3) de Listing 1. Pour supporter la réification, un pattern de correspondance à partir d'une référence d'origine vers un nouveau chemin est également fourni. Par exemple, l'élément de référence *parent* est réifié par l'élément de type *Generalization*, cf. (2). Dans le sens de la théorie des graphes, ces correspondances supportent d'identifier un morphisme entre deux graphes étiquetés quand il y a des différences au nom de noeuds et d'arcs. Dans notre approche, ce morphisme est entre un TGM *effectif* et celui d'un nouveau MM de source.

Listing 1 – Une définition en MetaModMap : CDV1 vs. CDV2

```

1 import "platform:/resource/CD2RDB/MTs/CD2RDB.mgt" ;
2 import "platform:/resource/CD2RDB/MMs/CDV2.ecore" ;
3 mapping for CD2RDB ( CDV1ToCDV2 : CDV1 correspondsTo CDV2)
4 {
5   CDV1ToCDV2 {
6     concept PrimitiveDataType correspondsTo DataType; /* (1)*/
7     concept Class correspondsTo Class {
8       att2att isPersistent correspondsTo persistent,
9       ref2ref ownedFeatures correspondsTo ownedProperties, /* (3)*/
10      ref_reification parent correspondsTo generalization . Generalization . general /* (2)*/
11    };
12    concept Attribute correspondsTo Property {
13      att2att isPrimary correspondsTo primary /* (4)*/
14    };
15    concept Association correspondsTo Association {
16      ref2ref source correspondsTo from,
17      ref2ref target correspondsTo to
18    }
19  }
20 }

```

Les correspondances sont décrites au niveau de Ecore. Toutefois, en ce qui concerne l'inférence des TGMs *effectives* dans la section 3.2, ces descriptions sont imposées par des contraintes supplémentaires pour assurer la validation des éléments de MM *cartographiés*. En particulier,

- seulement des types référés, c'est à dire ayant un correspondant *GNode* dans le TGM *effectif*, peut être cartographiés sur des types dans le nouveau MM. Par exemple, le type *Operation* ne peut pas être cartographié.
- des attributs doivent être mis en correspondance avec d'autres attributs ayant le même type primitif. De même que pour les types, les attributs cartographiés doit être présents dans le TGM *effectif*.
- seulement des références qui ont un correspondant *GEdge* dans le TGM *effectif* peut être cartographiées.

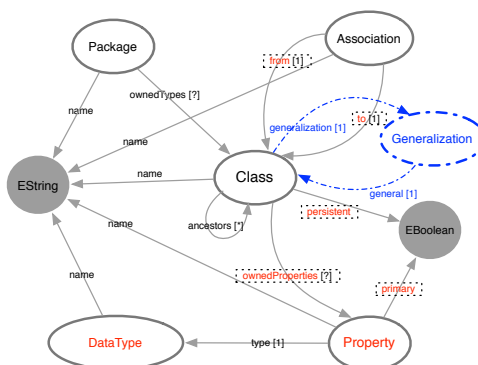
De plus, une relation de sous-graphe basée sur le nom de TGMs est définie en prenant en compte *Multiplicity* comme suit.

TGM sous-graphe : $G'(N', E') \subseteq G(N, E)$ alors que

- **Node** : $\forall n' \in N', \exists n \in N (n'.isMatch(n))$
- **Edge** : $\forall e' \in E', \exists e \in E ($
 $e'.[from/to].isMatch(e.[from/to]) \wedge$
 $e'.name = e.name \wedge$
 $e'.multiplicity.isMatch(e.multiplicity))$

où *undefined_valued* est "matched" à *single_valued* et à *multi_valued*; et *single_valued* est "matched" à *multi_valued* (cf. formule 2).

Ainsi, tous les noeuds et les arcs d'un TGM *effectif*, selon les correspondances,

FIGURE 7 – TGM effectif *cartographié*.

peuvent trouver leur miroir dans le TGM du nouveau MM. La relation de sous-graphe est vérifiée entre le TGM effectif *cartographié*, cf. FIGURE 7, et le TGM de CDV2, cf. FIGURE 4. Cette relation est utilisée pour valider une définition en MetaModMap pour la substituabilité de MMs (cf. la formule 1).

3.4 Adaptation de Transformation

Compte tenu d’une définition en MetaModMap et d’une transformation en MOMENT2, un moteur d’adaptation est en charge de relier des patterns se référant à des éléments du MM d’origine à celles du nouveau MM, et puis de textualiser tout le modèle de transformation de sortie dans un nouveau fichier. Le pseudo-code de l’algorithme de re-liaison pour un pattern racine d’objet contre une cartographie de concept est illustré dans l’algorithme en Page 10.

Une fois avoir obtenu le sous-typage entre MMs (cf. le formule 1) par la cartographie, l’adaptation est effectuée pour tous les patterns. Nous faisons d’abord la re-liaison basée sur les concordances, via la procédure *rebindChangedElem()* dans l’algorithme en Page 10, et puis pour les éléments restants inchangés. FIGURE 8 (en bas à gauche) représente des patterns en mémoire avant de faire la re-liaison. Basée sur la cartographie, cf. le listing 1, nous relierons la référence à *ownedFeatures* du MM d’origine vers *ownedProperties* du nouveau MM, cf. la ligne 17 dans l’algorithme en Page 10. De même, la référence à *Attribute* est transmise à *Property*. Cependant, une réification de relation (un cas particulier de *Ref2FeaturePath*, cf. la ligne 18), e.g. l’introduction du type *Generalization*, cause l’adaptation dans la fonction *createGraphPattern()*, cf. la ligne 20, plus complexe. Dans ce cas, nous créons une nouvelle paire de patterns d’objet et de propriété, comme le montre FIGURE 8 (en bas à droite), et puis les faire référer aux éléments *Generalization* et *general*, respectivement.

D’autre part, les éléments restants inchangés, comme *Class* et *name*, peuvent être automatiquement relié aux éléments correspondants par leur nom, sans aucun soutien de la cartographie. FIGURE 8 illustre un extrait des patterns en la représentation textuelle et en mémoire avant et après l’adaptation.

```
rebindChangedElem(conceptMap, ote)
```

Require: (1) la cartographie de concept *conceptMap* qui contient les correspondances entre les deux concepts de type. (2) un pattern d'objet *ote* .

Ensure: Retourner le pattern d'objet relié en fonction de la cartographie

{*Perform lookup at the being-visited object template expression.*}

```

1: if conceptMap.conceptSource.equal(ote.referredClass) then
2:   ote.referredClass ← conceptMap.conceptTarget
3:   for all pte in ote.PROPTEMPEXPS do
4:     for all propertyMap in conceptMap.PROPERTYMAPS do
5:       if propertyMap.featureSource.equal(pte.referredProperty) then
6:         if pte instanceof AttributeExp then
7:           if propertyMap instanceof Att2Att then {Single attribute map}
8:             pte.referredProperty ← propertyMap.featureTarget
9:           else if propertyMap instanceof Att2FeaturePath then
10:            topFeaturePathExp ← propertyMap.featureTarget
11:            newPTE ← createGraphPattern(topFeaturePathExp, pte)
12:            ote.PROPTEMPEXPS ← ote.PROPTEMPEXPS \ pte
13:            ote.PROPTEMPEXPS ← ote.PROPTEMPEXPS ∪ newPTE
14:          end if
15:        else if pte instanceof ReferenceExp then
16:          if propertyMap instanceof Ref2Ref then {Single reference map}
17:            pte.referredProperty ← propertyMap.featureTarget
18:          else if propertyMap instanceof Ref2FeaturePath then
19:            topFeaturePathExp ← propertyMap.featureTarget
20:            newPTE ← createGraphPattern(topFeaturePathExp, pte)
21:            ote.PROPTEMPEXPS ← ote.PROPTEMPEXPS \ pte
22:            ote.PROPTEMPEXPS ← ote.PROPTEMPEXPS ∪ newPTE
23:          end if
24:        end if
25:      end if
26:    end for
27:  end for
28: end if
   {call recursively for lower object template expressions}
29: for all pte in ote.PROPTEMPEXPS do
30:   if pte instanceof ReferenceExp then
31:     rebindChangedElem(conceptMap, pte.value)
32:   end if
33: end for

```

```

1  rhs {
2    cd {
3      /* RHS graph patterns */
4      c1 : Class { /* GP1 */
5        parent = c2 : Class {},
6        ancestors = c2 : Class {}
7      }
8    }
9  };
10 ...
11
12 /* another rule */
13 lhs {
14   cd {
15     /* LHS graph patterns */
16     c : Class { /* GP2 */
17       name = cname,
18       ownedFeatures = a : Attribute
19         {
20         name = aname,
21         ...
22       }
23     }
24   }
25 };

```

```

1  rhs {
2    cd {
3      /* RHS graph patterns */
4      c1 : Class { /* GP1 */
5        generalization = c2 :
6          Generalization {
7            general = c2 : Class {}
8          },
9        ancestors = c2 : Class {}
10     }
11   };
12 /* another rule */
13 lhs {
14   cd {
15     /* LHS graph patterns */
16     c : Class { /* GP2 */
17       name = cname,
18       ownedProperties = a :
19         Property {
20         name = aname,
21         ...
22       }
23     }
24   }
25 };

```

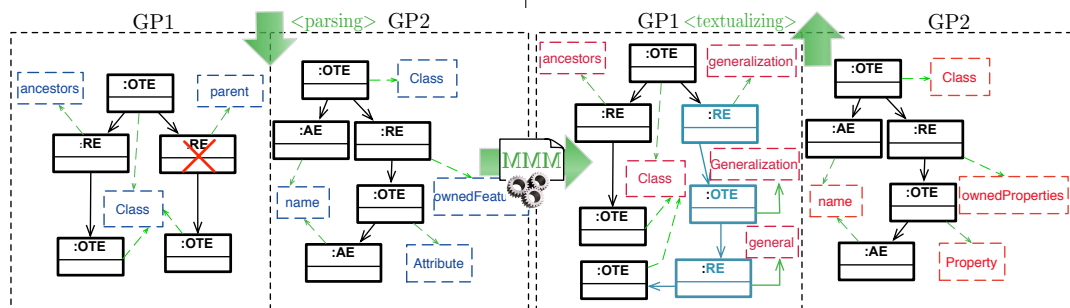


FIGURE 8 – Extrait de la représentation textuelle et en mémoire des patterns de graphe (modèle) dans le processus d’adaptation.

4 Adaptation de Transformation en Action

Comme prototype³, nous avons développé un plugin de Eclipse⁴ pour fonctionner avec MOMENT2.

3. disponible à perso.telecom-bretagne.eu/quyetpham/software/
4. en utilisant XText à xtext.itemis.com

4.1 Prototype de MetaModMap

Le plugin fournit un éditeur de texte, cf. FIGURE 9 pour notre DSL qui met en oeuvre l'approche de typage présentée dans les sections 3.1, 3.2 et 3.3, et un interpréteur pour adapter les transformations comme dans cette section.

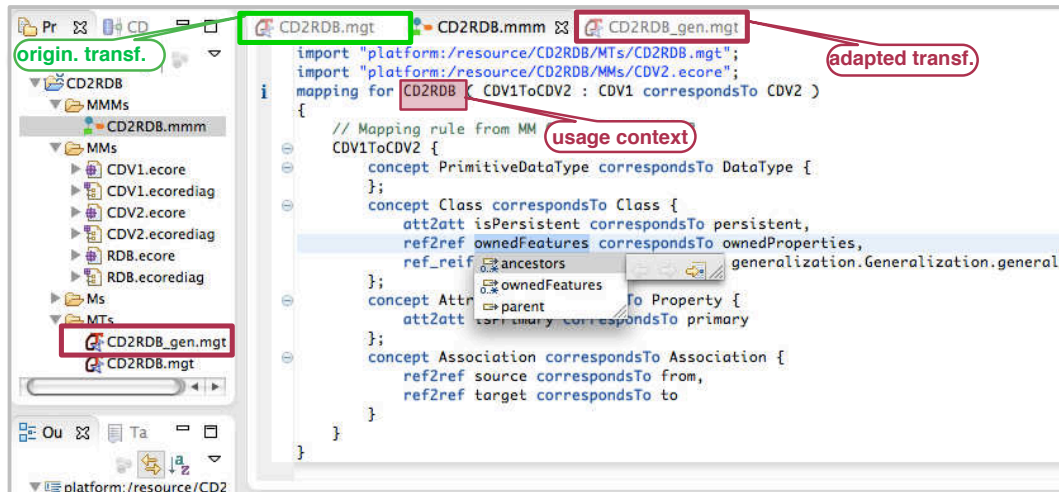


FIGURE 9 – Editeur de MetaModMap.

L'éditeur est équipé du mécanisme de typage et ses *aspects de validation et de scoping* fournissent des suggestions et des alertes pour les utilisateurs.

4.2 Exemple de Validation

Nous appliquons notre approche pour l'application simplifiée *Gestion de Clients* inspirée de [Bézivin 05a] comme une étude de cas. La vue de schéma de classes de cette application peut être modélisée dans les outils CASE différents qui sont basés sur les métamodèles différents, résultant en deux représentations, comme indiquées dans FIGURE 10 (au milieu à gauche et à droite). *Le même modèle intentionnel d'entrée*, cf. FIGURE 10 (en haut), est reifié en deux représentations différentes avec le changement de nom (Attribute/Property, isPersistent/persistent, isPrimary/primary, ownedFeatures/ownedProperties) et une réification de relation (parent/generalization.Generalization.general).

Lorsqu'on change d'outil de conception, la transformation doit être adaptée. Compte tenu d'une définition en MetaModMap, comme le montre le Listing 1, une nouvelle transformation peuvent être générée par l'interprète de MetaModMap. Nous obtenons, comme prévu, *le même modèle de sortie* comme dans FIGURE 10 (en bas) pour tous les deux représentations du modèle intentionnelle d'entrée, en appliquant respectivement la transformation d'origine et celle adaptée.

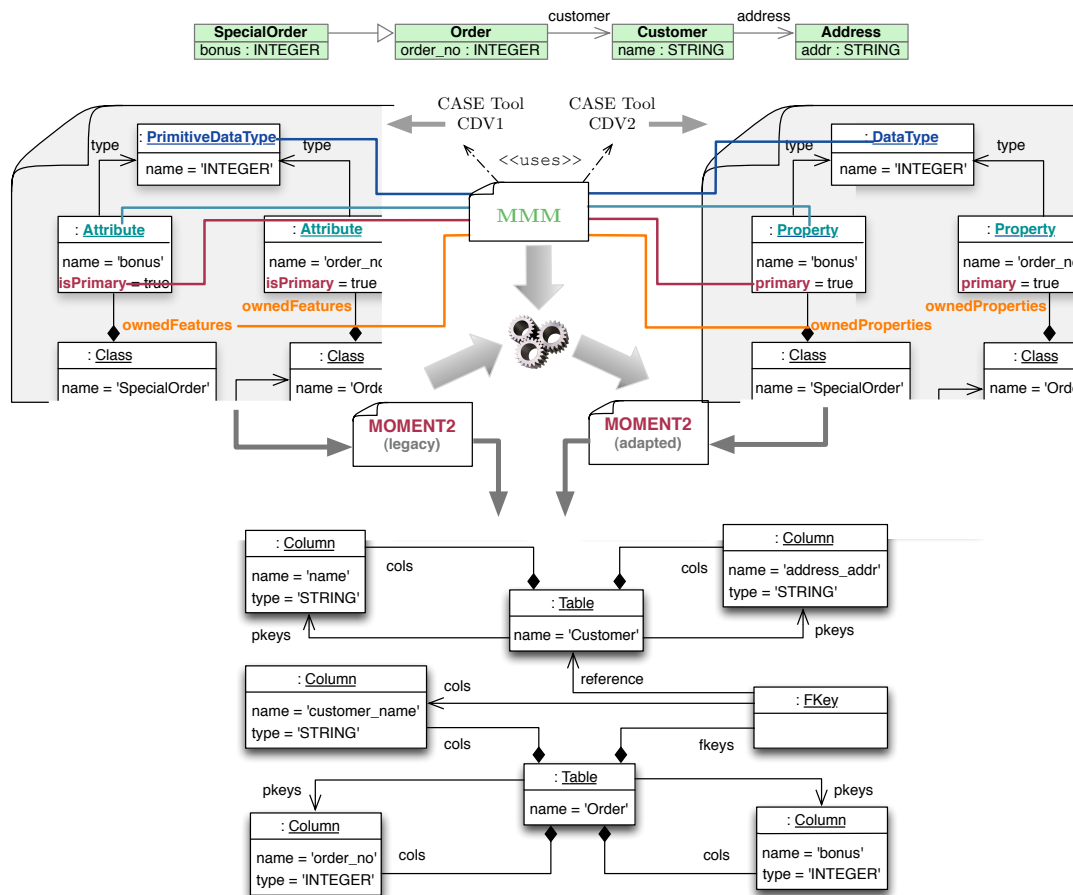


FIGURE 10 – Modèles d’entrée et sortie de la transformation CD2RDB.

5 Travaux Connexes

Réutilisation de transformation grâce à l’adaptation en réponse aux changements de métamodèle a été étudiée avec deux stratégies principales. Nous avons adopté la première qui considère des transformations comme des boîtes blanches et les transforme. La deuxième stratégie regarde une transformation comme une boîte noire et génère des adaptateurs pour la transformation sans modifier l’original.

Dans la première stratégie, le concept de *transformation générique* présenté dans [de Lara 10] qui est définie sur un MM emphconceptuel et ensuite spécialisé pour un MM spécifique par un DSL de liaison simple, Cuadrado *et al.* [Cuadrado 11] et Wimmer *et al.* [Wimmer 11] ont étendu le DSL de liaison pour hétérogénéités structurelles de MMs. Les adaptations apportées dans [Wimmer 11] sont générées en utilisant “*helpers*” d’ATL ou directement adaptée aux transformations. En raison du manque de *helpers* en MOMENT2, les règles de transformations doivent être directement adaptées dans notre approche. Néanmoins, avec les patterns de graphe de MOMENT2, le processus n’a pas besoin de tenir compte des *méta-attributs* tels que : *ordered*, *lowerBound*, sauf *upperBound* qui nécessite un traitement spécial. Par

rapport à ces travaux, la substituabilité de MMs est définie clairement (et formellement) dans notre approche par TGM avec un mécanisme d'inférence pour le typage de transformation.

En revanche, les approches présentées par Levendovszky *et al.* dans [Levendovszky 10] et Di Ruscio *et al.* dans [Ruscio 11] s'appuie sur l'évolution de point de vue. L'approche adoptée dans [Levendovszky 10] est également dédiée pour le paradigme de transformation de graphe, mise en oeuvre pour fonctionner avec GReAT⁵. Toutefois, le principe de la réutilisation n'est pas basé sur l'isomorphisme de graphes comme le nôtre, mais sur les évolutions mineures de MMs. Par conséquent, quand une référence et un type représentant le même concept de modélisation comme la paire *parent/generalization.Generalization.general*, le pattern original ne peut pas s'évoluer automatiquement par leur outil, en opposition avec notre traitement dans lequel le pattern adapté encore conserve, à un certain niveau, la topologie du graphe même de celle de l'origine.

Les auteurs de [Ruscio 11] proposent EMFMigrate⁶ pour évoluer des transformations écrites en ATL. Une paire de DSL est introduite pour la spécification de différenciation de MMs (EDelta) et les règles de migration (personnalisable) (ATL-Migrate). Les modèles de EDelta sont en fait la trace rétablie pour spécifier l'évolution de MM de l'étape par étape. Les règles de migration pourraient être appliquées si les patterns conditionnels sont "matched" sur le modèle de différence. Ensuite, les informations extraites seront utilisées dans des règles de réécriture pour évoluer la transformation. En comparaison avec leur solution, notre approche n'est pas basée sur les traces de l'évolution de MMs, mais sur l'observation des différences de la représentation de concepts de modélisation.

La deuxième stratégie considère une transformation en tant qu'une boîte noire. Kerboeuf *et al.* [Kerboeuf 11] présentent un DSL dédié pour l'adaptation qui gère la copie ou la suppression des éléments d'un modèle conformement à un nouveau MM pour le faire devenir une instance du MM d'entrée d'un outil de transformation endogène. Le mécanisme d'adaptation fournit également une trace pour transformer la sortie de l'outil vers une instance du nouveau MM.

Outre les stratégies ci-dessus, Sen *et al.* proposent dans [Sen 10] une autre approche pour rendre une transformation réutilisable en étendant un MM. Elle se base sur la fonction de tissage fournie dans Kermeta⁷. Contrairement à notre approche qui permet aux utilisateurs de définir des correspondances entre MMs à un niveau élevé et en effectuant automatiquement des adaptations, les adaptations de cette approche sont ajoutées par des utilisateurs au niveau du langage de transformation. Cette approche utilise un mécanisme d'élagage de MM [Sen 09] pour réduire les besoins sur l'adaptation via le tissage des aspects pour satisfaire au principe de typage dans [Steel 07]. L'approche est similaire à notre solution, cependant, notre approche de typage permet plus de flexibilité sur la multiplicité puisque nous appliquons l'ordre dans la formule 2 alors que [Steel 07] utilise une comparaison d'égalité stricte.

5. www.isis.vanderbilt.edu/tools/GReAT

6. www.emfmigrate.org

7. www.kermeta.org

6 Conclusion

Dans ce travail, nous avons présenté une approche de typage basée sur les graphes pour les modèles et les transformations (sections 3.1 et 3.2) qui prend en compte le problème de la multiplicité des associations. Cette approche a été utilisée avec succès pour adapter automatiquement des transformations pour permettre leur réutilisation lorsque les MMs sont différents. Ceci est possible grâce à un mapping, décrit avec notre DSL MetaModMap, est exposé (sections 3.3 et 3.4). En outre, l'éditeur du DSL est équipé d'un système d'alerte de suggestion qui est la conséquence de l'approche de typage que nous proposons.

Il y a encore quelques problèmes dans l'approche présentée. Tout d'abord, nous avons appliqué notre processus d'adaptation aux transformations exogènes lorsque leurs métamodèles de *source* sont modifiés. D'autres cas dans lesquels les métamodèles de *cible* change doivent être étudiés plus. Néanmoins, nous pensons que le même principe peut être utilisé en prenant *composition* en compte, par exemple, un objet qui est à un type ne peut pas être créé en dehors de certains objets du type de conteneur.

D'autre part, concernant l'expressivité, nous prévoyons d'étendre notre DSL pour soutenir d'autres types de représentation différente de MM, par exemple un type est divisé en plusieurs parts, en basant sur notre approche de typage à base de graphes. Autrement dit, les correspondances définies dans notre DSL sont considérées comme "zooms (in/out)" sur des noeuds, des arcs ou mineures sous-graphes connexes du *TGM* effectif à condition que l'isomorphisme entre les TGMs de MMs est justifiée.

Chapter 1

Introduction

Contents

1.1	General Context	1
1.2	Problems and Challenge	2
1.3	Objective, Approach, and Contributions	3
1.4	Structure of Thesis	5

1.1 General Context

With the explosion of advanced technologies in hardware and network, software in pervasive systems that collaborate to provide services to society has become more and more complex. Software in these systems operate in distributed environments consisting of diverse devices and communicates using various interaction paradigms [France 07a]. These software systems are often required on one hand to keep up with constantly evolving business processes and on the other hand to adapt to the changes in operating environments. Despite significant advanced programming languages supported within *integrated development environments* (IDEs), software development using traditional code-centric technologies faces many challenges. Thus, software development requires new methodologies, architectures, and infrastructures to support changes to business processes that are defined at the business level and propagated to the level of technological systems. In response to these requirements, a modern trend in software engineering practice is to use models instead [Forward 08].

Model-centric or model-driven development (also known as Model-Driven Engineering – MDE) provides techniques to realize and automate seamlessly the propagation of changes from the business level to the technical level. Currently, there is a variety of approaches, and standards with supported tools which realize this methodology like OMG’s Model-Driven Architecture (MDA) [MDA 03], Microsoft’s Software Factories [Greenfield 03], Agile Model-Driven Development [Ambler 12], Domain-Oriented Programming [Thomas 03], or Model-Integrated Computing (MIC) [Sztipanovits 97], and so forth. This software development methodology treats mod-

els as primary artifacts to develop, uses these models to raise the level of abstraction at which software developers create and evolve software, and reduces the complexity of these artifacts by separation of concerns (aspects) of a system being developed.

1.2 Problems and Challenge

In order to foster software development in practice, *Model Driven Engineering* (MDE) [Kent 02] proposes the development of software systems using *models* [Kuhne 06] as *first-class citizens*. In this software development perspective, “*everything is a model*” [Bezevin 05] and *model transformation* [Sendall 03, Mens 06] is a pivot activity that consists of transforming models of source to target *modelling language* [Seidewitz Ed 03]. To specify a model transformation, one of the most appropriate approaches is to base on (*linguistic*) *metamodels* [Kuhne 06, Karagiannis 06] which define *abstract syntax* of modelling languages. This choice is natural because the *metamodel-based transformation* approach [Levendovszky 02] permits specifications of mappings focused on the syntactical (structure) translation in using predefined elements in metamodels. Hence, a model transformation specification can be applied to every model conforming structurally to the base metamodel. Since model transformations are often not straightforward to specify and require a lot of investment and effort, it is necessary to be able to customize legacy model transformations for our business and underlying operating environments in a timely manner to solve the problems of interoperability that come from the modelling languages’ usage and the recurring evolution of modelling languages. Also, software companies need to *reuse existing model transformations* with as less effort as possible.

Problems. The differences of syntax (form) and semantics (meaning) in representation formats between modelling languages’ metamodels are main obstacles that hinder the efficient customization and evolution of legacy model transformations.

- Modelling languages and their associated metamodels often differ over different software companies, although they were developed to describe the same domain of application. In this case, semantically similar concepts are probably reified in different ways in different metamodels. As a result, a model transformation tightly coupled with a metamodel cannot be used with another metamodel to perform transformation on similar concepts.
- Another case is the recurring evolution of metamodels as their new versions are released, e.g. the released versions of the metamodels for the UML modelling languages from 1.5 to 2.0. Since model transformations are tightly coupled with metamodel specifications, software companies not only have to deal with the evolution of metamodels, but also with the evolution of model transformations.

Challenge. It is necessary to efficiently customize or evolve legacy model transformations for supporting the use of various modelling languages, despite differences in their syntax and semantics and the evolution of metamodels through different

released versions.

1.3 Objective, Approach, and Contributions

This section identifies the objective to deal with the problems and challenges in the scope of model transformation customization and evolution as described in Section 1.2. Figure 1.1 gives a glance at the objective in different scenarios of this dissertation. In following parts, we first describe the approach that we use to achieve this objective, then we clarify the list of contributions of this thesis.

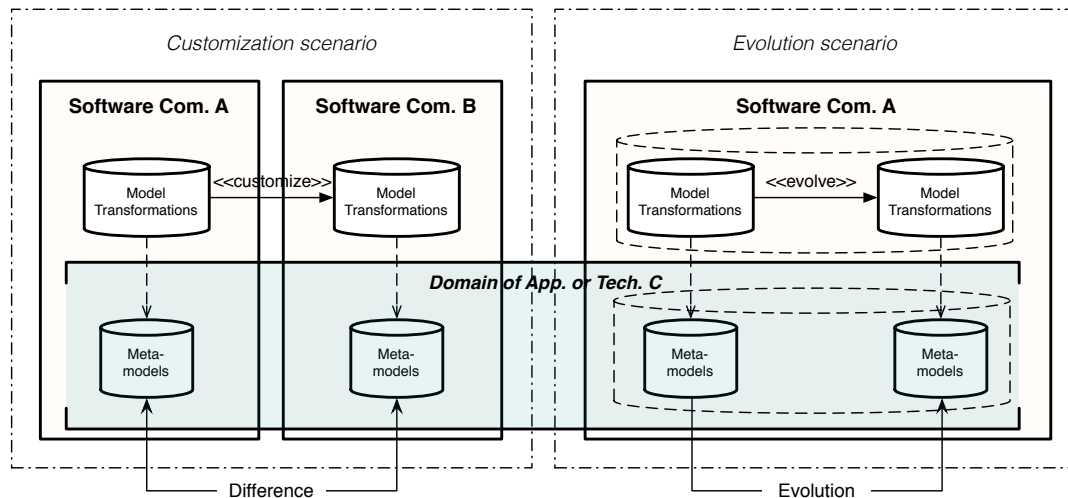


Figure 1.1: Objective overview in different scenarios.

Model Transformation Customization and Evolution. Using different meta-models in different software companies in a group results in the need to synchronize model transformations owned by these companies (see the left part of Figure 1.1). To assure the consistency between their model transformations, a company can customize one from its partner company to align with the owned metamodels. Another situation that necessitates the change of model transformations is the evolution of metamodels by different released versions over the life-cycle of modelling languages (see the right part of Figure 1.1). When a metamodel evolves, every legacy model transformation specified upon the old version of the metamodel has to be adjusted. In order to efficiently adapt model transformations, it is necessary to overcome the heterogeneity in syntax and semantics of modelling languages.

Objective. *Facilitate the reuse and evolution of model transformations to overcome the syntactic and semantic differences in modelling languages.*

Approach. To address this objective, we apply the *design-science paradigm*. According to Hevner [Hevner 04], most of the research methodologies in information sys-

tems can be categorized into two paradigms: behavioral-science and design-science. The behavioral-science paradigm seeks to develop and verify theories that explain or predict human or organizational behavior, while the design-science paradigm seeks to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts. Hence, following the design-science paradigm, we develop an approach which provides mechanisms to automatically customize and evolve legacy model transformations to overcome the differences in syntax and semantics of modelling languages that benefit software companies with more efficient reuse and synchronized model transformations. This approach is based on a small language which exhibits the semantics bridge between different representations of the same or similar concepts in modelling languages' metamodels and on a particular interpreter of this language to migrate legacy model transformations.

Literature Review at a Glance. Investigation in literature to increase the re-usability of model transformations in response to changes in metamodels can be categorized into two main strategies: *model adaptation* and *transformation adaptation*. An additional transformation (specified in a new language coupled with its own particular interpreter) is developed to adapt models of the new metamodel to conform to the original metamodel in the former strategy (cf., e.g. [Kerboeuf 11]). The additional transformation and its coupled-inverse transformation are then composed with the legacy transformation i.e. $T_{inverse} \circ T_{legacy} \circ T_{adapt}$; for reuse. In this strategy, the new model transformation is composed of the legacy model transformation and the additional interoperability transformations, which are specified in different transformation languages. As a consequence, the obtained model transformations become more complex and difficult to manage for further adaptation because of heterogeneities in using different languages to specify model transformations.

In the latter strategy, the main idea is to adapt the legacy transformation itself to be valid on the new metamodel. This strategy is widely used to support metamodel/transformation co-evolution [Mendez 10, Levendovszky 10]. After an evolution on metamodel, any transformation written for the previous version of a metamodel might become inconsistent when being used for the new version. For instance, the *name* meta-attribute of a metamodel element may be changed or a relation element among type elements may be replaced by a new type element in the new version of metamodel. Some of these evolutions may be identified and used as directives for a transformation adaptation process, i.e. $T_{legacy} \rightsquigarrow T_{adapted}$. One advantage of this approach is that the adaptation is executed only once to obtain a new reliable transformation in the same monolithic transformation language for permanent reuse.

Contributions. In order to extend the reuse capability, we develop the following artifacts to support the customization and evolution of model transformations.

- We propose a novel notion of *model type* based on the graph nature of metamodels, namely *Type Graph with Multiplicity* (TGM). We also introduce an inference mechanism on the transformation, whose definition is based on the respective metamodels. This mechanism allows to derive the *effective* model

type in a specific context in which the transformation itself is used as a filter to select the used metamodel elements. From this point, a sub-typing relationship is defined between the *effective* model type and another model type of the respective metamodel for checking substitution possibility in order to reuse the transformation with the new metamodel. The proposed model typing approach is used as the theoretical basis to develop the language in our second contribution.

- We develop the *Metamodel-based Mapping (MetaModMap) approach*, which provides utilities to automatically deal with customization and evolution scenarios on model transformations. Our approach is based on the typing approach (in our first contribution) and the semantic correspondences between (combinations of) elements of metamodels. We implement a MetaModMap prototype and experiment it using simple case studies.
- We develop *concepts and techniques* to reify the MetaModMap approach. First, we provide a Higher-Order Transformation (HOT) [Mens 06] language for model transformations specified in graph rewriting-based model transformation languages. The MOMENT2-GT language [Boronat 09] is chosen to describe model transformations. Second, we implement a compatibility checking mechanism between metamodels under a HOT definition and a migration algorithm in order to rewrite model transformations. Thus, our approach promotes indirect reuse of legacy model transformations.

Overall Contribution. We propose in this work a transformation migration approach to transformation reuse. Nevertheless, instead of analyzing metamodel evolution as in related work, we consider the graph nature of metamodels and we align metamodels that may have very different history (in the customization scenario). The abstract graph view plays a centric role for defining metamodel substitution relation. A higher-order model transformation language is proposed to specify intended semantic correspondences between elements of the two metamodels under consideration. Metamodel correspondences are defined by the users and used to check substitution possibility. In the end, they are used to proceed the transformation migration process.

1.4 Structure of Thesis

The remainder of the thesis is organized as follows:

- **Chapter 2** begins with explaining the core concepts employed in model-driven engineering like models, metamodels and model transformations. Then, a case study and existing approaches to supporting transformation reuse are discussed regarding the identified challenge in Section 1.2.
- **Chapter 3** formulates the approach of Metamodel-based Mapping (Meta-

ModMap). First, we give theoretical basis for our approach. These notions are: model type based on the graph nature of metamodels, the formal description of a mechanism to infer *effective* model type for transformations and the initiative of model sub-typing definition via mappings. Second, we present the MetaModMap language which provides a solution that is based on exhibiting the semantic correspondences between different representation formats of metamodels. The substitution ability between metamodels in the transformation is based on the model sub-typing defined in our approach. MetaModMap fosters automated evolution, and customization of model transformations. The approach will be presented with an illustration on the case study. Main contributions in our approach are also discussed in detail in this chapter.

- **Chapter 4** describes concepts and techniques used to implement a prototype of MetaModMap which realizes the principles of our approach and explains it working in application scenarios.
- **Chapter 5** summarizes the contributions presented in the dissertation and anticipates some perspectives of future work.
- **Appendices A and B** give some implementation details.

Chapter 2

State of the Art

Contents

2.1	Chapter Overview	7
2.2	Concepts in Model-Driven Engineering	8
2.3	Model Transformation Reuse	34
2.4	Related Work	56
2.5	State-of-the-Art Summary	65

2.1 Chapter Overview

This chapter describes the concepts and principles in Model-Driven Engineering which are used in the context of this dissertation, then presents a motivating example and finally discusses some approaches related to transformation reuse in the current literature.

Section 2.2 first presents key notions in Model-Driven Engineering. Then, this section describes the notion of Domain Specific Modelling Language (DSML), a basic concept used in Model-Driven Engineering for separating of concerns of a system, a particular methodology for Model-Driven Engineering. This section also focuses on model transformations between model spaces specified by DSMLs and presents a category of transformation approaches.

Section 2.3 introduces a concrete and simple case study on the need of model transformation reuse problem. Then, by discussing on the characteristics of various model transformation languages, we chose the most relevant transformation language to realize our approach identified in Section 2.3.2. In the end, the transformation reuse problem will be discussed in detail as the specific context of this thesis.

Section 2.4 presents some related approaches to address the problem of adapting model transformations in response to changes in metamodels. These approaches are categorized in: *model adaptation*, *transformation adaptation* and *metamodel adaptation* strategies. The limitations and advantages of these approaches will be discussed

in detail for each concrete related work to identify existing challenges which are improved in this thesis.

Finally, Section 2.5 summarizes the content presented in the chapter.

2.2 Concepts in Model-Driven Engineering

This section provides the background knowledge relevant to the research domain considered in this dissertation. First, *Model-Driven Engineering* (MDE), an emerging software development methodology in the recent decades, will be introduced. This section then continues presenting the key notions used in MDE such as: *model*; *metamodel* and *meta-metamodel*, which are the first-class entities defined in the *Meta-Object Facility* architecture. Next, *Domain Specific Modelling Language* (DSML), a notion used to (meta-)model concerns of interest in various *Model-Driven Engineering* approaches, will be given, which includes the background information on the main activities on models such as: *refactoring*, *transforming* and *refinement*. In the end, this section focuses on presenting a particular kind of model transformation, i.e. from a model space defined by a DSML to another one, which includes the categories of model transformation approaches dedicated particularly to model-to-model transformation tasks.

2.2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) [Bézivin 03b] is a methodology which focuses on raising the level of abstraction to alleviate the complexity in software development process. The main characteristic of MDE to accomplish the abstraction requirement is the use of model for representing the artefacts of a system under consideration rather than employing the computing (or algorithmic) concepts. Thus, MDE covers a range of software development approaches in which abstract models are created, exploited and then transformed into more concrete implementations, ranging from system skeletons to complete or deployable products.

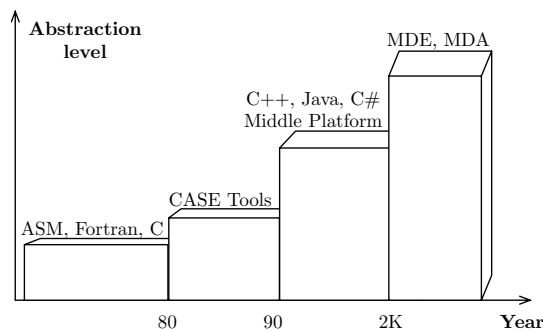


Figure 2.1: Software development methodology roadmap.

The history of software development methodology, cf. Figure 2.1, can be divided

in four periods. From the invention of early programming languages and platform technologies until in the 1980s, developers programmed on operating system platforms, such as OS/360 and Unix equipped with Assembly and Fortran or C languages. Although these languages and platforms raised the level of abstraction, i.e. shielded developers from complexities of programming to machine code or hardware, they still employed a *computing-oriented* focus. Using these computing technologies made a wide gap between the problem space of application domains, such as banking, telecom and biology, etc., and the solution space provided by the technologies. As a result, the developers at that period needed to spend much effort for realizing a large-scale business problem into a technical solution.

Computer-Aided Software Engineering (CASE) [Chen 89] then was promoted as an approach to enable developers to express their design choice by general-purpose graphical programming representations rather than implementing directly in the first-generation languages. In using the thorough analysers and synthesizers equipped with CASE tools, developers can avoid errors incurred in programming by traditional languages – for example, memory corruption or leaks when using languages like C – and manually coding, debugging and porting program, respectively. However, CASE did not have much success in practice. One of the reasons is that they did not support many application domains effectively. Another reason is the lack of common nowadays middle-ware platforms to integrate implementation codes of the complex, distributed and large-scale systems on different platforms. This challenge limits the capability to support multiple platforms in using CASE tools.

To fill the gap between business space and technical space. Since the 1990s, many expressive Object-Oriented (OO) languages, such as C++, Java, C# have been used to develop reusable class libraries and application framework platforms. Thus, these application platforms have leveraged the abstraction of technical space and hence facilitated developers in developing, maintaining and reusing middle-ware systems [Booch 94]. Nevertheless, with the rapid growth of the complexity of platform technologies, such as .NET, J2EE and CORBA, thousands of classes were needed to reach a desired level of expressiveness. As a result, porting implementation codes from a platform technology to another one (or another version) is a really difficult issue. In addition, using only OO languages is still unable to express the domain concepts effectively.

There were a lot of efforts in the last decade to develop technologies in order to address platform complexity and the need of expressing concepts of particular domains for solving the problem-implementation gap [France 07b] that exists during the software development process. As summarized by Schmidt [Schmidt 06], these technologies use models as a centric concept and build development supporting facilities around the model concept including: 1) *Domain Specific Modelling Languages* (DSML) that are designed especially to express concepts within particular domains or even the domain of middle-ware platform. These DSMLs allow developers to work at the high-level abstraction rather than OO languages; 2) *transformations and generators* equipped within DSMLs to handle the mapping of high-level abstraction models to lower implementation details. These facilities have been fully shielding the

developers from complexity of the technical solution space.

In recent years, organizations both in academy and industry, i.e. Object Management Group (OMG) [OMG 00] have been trying to give a clarity and consensus for the basic notions to support building MDE-based frameworks. In consequence, standard specifications such as: Meta-Object Facility (MOF) [MOF 06] which provides the way to develop DSMLs by using *metamodels*; Model-Driven Architecture (MDA) [MDA 03] which guides how to mapping high-level *models* to lower-level ones and Query/View/Transformation (QVT) [QVT 08] to give a standard on building a high-level Model Transformation Languages was provided. Based on these guidelines, many MDE meta-tools (e.g. GMF [GMF 11], MetaCase [MetaCase 11], amongst others) have been developed which remain successful in both commercial and research software development tools based on the model concept. In the following sections, a review on the basic concepts in MDE such as: model, metamodel, meta-metamodel, model transformation will be represented for more details.

2.2.2 Model

There are many discussions about the meaning of the notion “*model*” in the MDE context. But the following definition is adopted in the scope of this dissertation:

“*A model is an abstract representation of (some aspects of) a system under study (SUS).*” [Czarnecki 00]

According to the definition, models are used to reduce the complexity in order to better understand the systems (not yet or already existing) which we want to develop or maintain, respectively. Models are constructed to serve for the given purposes such as representing human understandable specification of some particular aspects or machine mechanically analysable description of aspects of interest [France 07b]. Although models cannot represent all aspects of the being-developed (or -maintained) systems, they allow us to deal with the systems in a simplified manner, to avoid the complexity of the solution spaces [Rothenberg 89]. Using models of a familiar domain rather than implementation code helps a broader range of stakeholders, from system engineers to experienced software architectures, to ensure that the software systems meet user requirements [Schmidt 06].

The term “*model*” in MDE refers to all language-based formulated artifacts [Kuhne 06], unlike, e.g. physical dimensional models, or opposed to mathematical models which are understood as an interpretation of a theory. However, when using models, they can play as two different kinds of role. The two kinds of models are *token* and *type* models [Kuhne 06] depending on their relationship to the system being developed (maintained).

2.2.2.1 Token models

A *token model* is a model whose elements capture *singular* (as opposed to *universal*) aspects of a system’s elements, i.e. they model individual properties of the elements in the system under consideration.

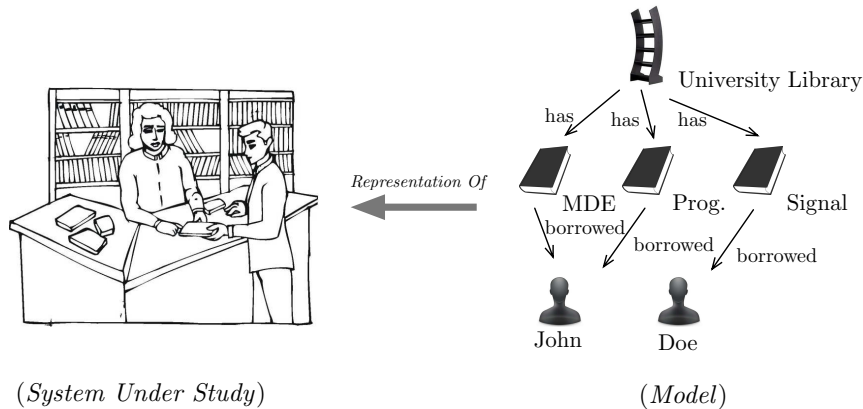


Figure 2.2: An example model of a library.

Figure 2.2 illustrates the relation “*Representation Of*” between a token model and an intended system of which the model will represent. In that figure (on the right hand-side), we use depictions of the real world system for model elements for illustrative purpose only, so the model can be regarded as real subject rather than models themselves. This model is created by using a UML object diagram, cf. Figure 2.3 (at the bottom of the right hand-side), to capture the system’s elements (and relationships between elements) of interest in a one-to-one mapping and represent them with their individual attributes values, e.g. in this case the value of the *title* attribute of each book is captured. In conclusion, the meaning of *abstraction* for *token models* is a projection on the system being modelled.

Token models are typically referred to as “*snapshot models*” since they capture a single configuration of a dynamic system, i.e. a system whose state changes over the time. They also have other possible names such as: “*representation models*” (because of the representation character) or “*instance models*” (as they are instances as opposed to types).

2.2.2.2 Type models

Although using token models has many useful applications, e.g. capturing an initial configuration of a modelled system for a simulation process or representing information, which is a one-to-one mapping with a real system. However, they do not reduce complex systems in concise descriptions [Kuhne 06]. Using “*type models*” is thus more adequate for that requirement. For this reason, objects which have the same properties can be *classified* into an *object class* – also referred to as a concept. Hence, instead of memorizing all particular observations, the human mind

just collects the concepts and their *universal* properties into a *type model* to model the system under consideration. As a consequence, most of the models in MDE are *type models* whose elements capture the *universal* aspects of a system’s elements by means of a *classification*.

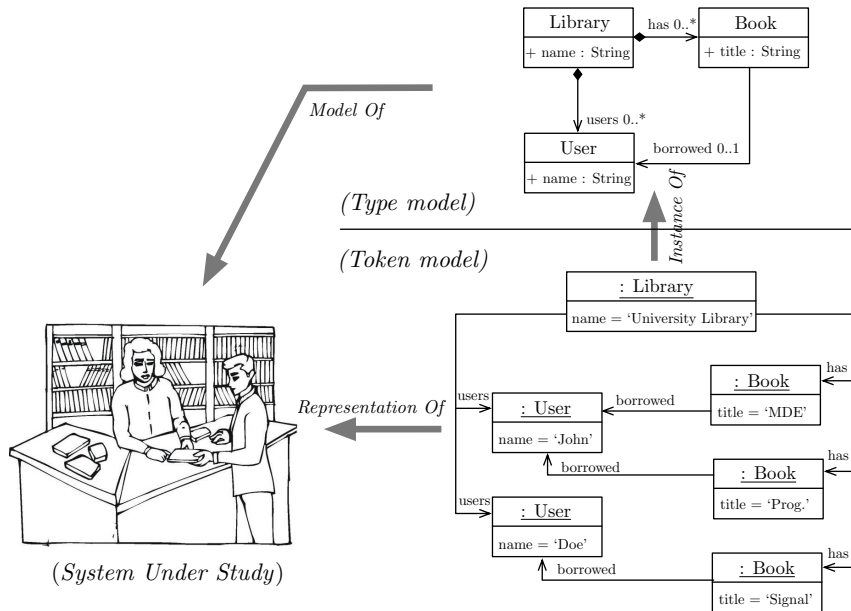


Figure 2.3: Different kinds of model roles.

Figure 2.3 shows (at the top of the right hand-side) a *type model* for the modelled library, created by a UML class diagram. Figure 2.4 and most other figures in this dissertation use standard UML notation [Rumbaugh 99] with the meaning of objects, classes, associations, dashed dependency lines with “*instance-of*” stereotypes to denote instantiation, etc. As opposed to *token models* which represent all particular elements of a system and their links, *type models* capture the types of interest and their associations only. Thus, *type models* can also be called as “*schema models*” or “*classification models*” [Kuhne 06]. The relation between *token models* and *type models* is called *Instance Of*, as shown in the Figure 2.3.

2.2.3 Metamodel

A metamodel is defined in [Seidewitz Ed 03] as

“*a specification model for which the systems under study being specified are models in a certain modelling language*”.

According to the definition, a modelling language is defined by a metamodel which

plays a role as a *type model* for classifying a universe of models into different modelling languages. A metamodel is not directly used to specify the systems under-study, however, it defines a set of concepts of the modelling language (and relationships between concepts) to express the possible model corresponding to (some particular aspects of) the system. Thus, a model corresponding to a system plays a role as a *token model* and is verified by the relation “*Instance Of*” with the metamodel of the modelling language.

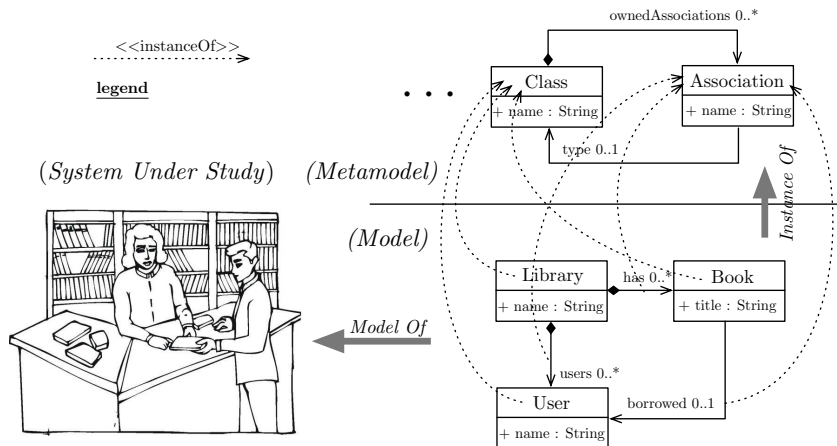


Figure 2.4: A metamodel to model the Class aspect of the Library system.

In using the MDE methodology, a being-developed (or -maintained) system can be modeled by considering different aspects (also called views). Figure 2.4 shows a model (at bottom-right) of library, which is respected to the class view, suppose that the system is reflected as an object-oriented software system. The definition of this view is based on a metamodel (at top-right) with concepts of the modeling language such as: *Class*, *Association*, etc. In addition to the class view, there are also other views – the user interface view, the database view, etc. They can be interesting when modelling a software system. Each view corresponds to a modelling language, which is based on a particular metamodel and is used to model a particular aspect of a system.

2.2.4 Meta-metamodel

A metamodel is itself a *token model* if we consider its described language as the system under-study. Thus, it can be expressed in some modelling language. One particular interesting case is when a metamodel of a certain modelling language uses the same modelling language. That is, all elements in the metamodel can be expressed in the same language that the metamodel is describing. Such a metamodel is called a *reflexive metamodel*.

A *meta-metamodel* in that case is constructed by a *minimal reflexive metamodel* [Seidewitz Ed 03] that uses the minimum set of elements of the modelling language. That is, every modelling language’s metamodel elements can be expressed using this

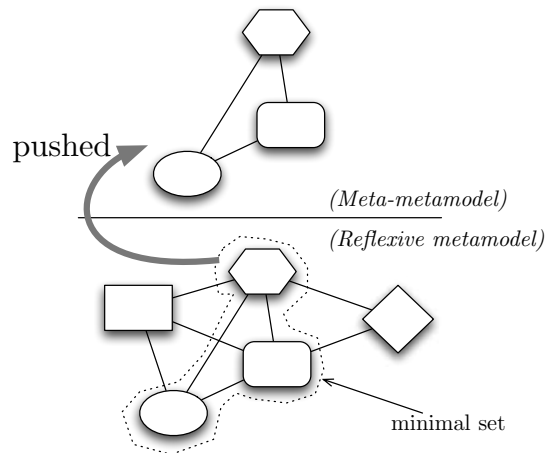


Figure 2.5: Illustration of a reflexive metamodel.

minimal set of modelling elements. However, if any element in this set is removed, some essential metamodel elements could not be expressed.

The UML metamodel is an example of *reflexive metamodel* since it respects to the previously mentioned characteristics. In fact, the entire UML’s abstract syntax [UML 05] can be expressed by using a minimal subset of UML’s static-structure modelling constructs – classes, packages, associations and generalizations, etc. – as a meta-metamodel. Using this subset of UML, in addition to UML metamodel itself, various modelling language’s metamodels, to consider different aspects of the being-developed (-maintained) system, can also be expressed as: database, user interface, etc. Natural languages such as English, and the Object Constraint Language (OCL) [OCL 06] are also used as complementary metamodelling languages to express metamodels.

2.2.5 The MOF Metamodelling Architecture

In a similar vein as the aforementioned notions, OMG [OMG 00] proposed a specification of a metamodelling framework, called OMG Meta-Object Facility (OMG-MOF) architecture, as a standard to develop model-driven systems. The OMG-MOF specification adopts a four-layer metamodelling architecture, as shown in Figure 2.6, to define specific modelling languages.

The top layer M3 is a specification of the metamodelling language – defined by a *reflexive metamodel*, named Meta-Object Facility (MOF) [MOF 06] – that is used to express metamodels in layer M2. Only a set of essential concepts (and a part of relationships) of the MOF meta-metamodel is needed to create metamodels. Figure 2.7 presents an excerpt of the *minimal set* of MOF, named Essential MOF (EMOF) meta-metamodel. This metamodel is *reflexively* and *minimally* specified, so no higher layers are needed.

Metamodels defining modelling languages (for example, an abstract syntax model

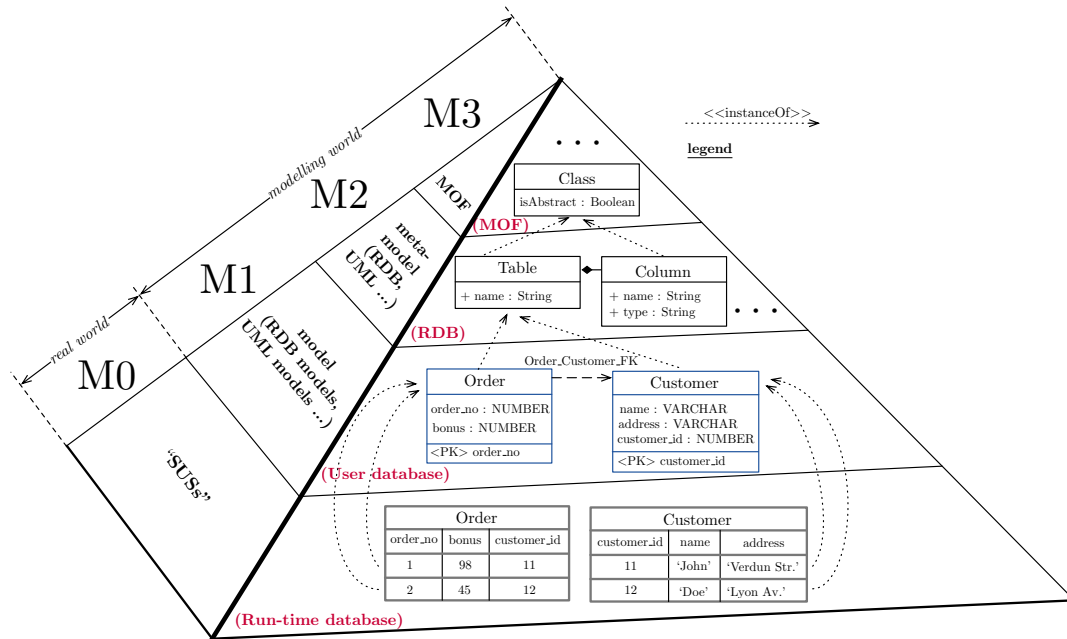


Figure 2.6: MOF four-layer metamodelling architecture.

in the UML specification [UML 05]), which reside in layer M2 of Figure 2.6, could be expressed by instantiating basic meta-concepts of the (E)MOF meta-metamodel. Then, we give particular names to define new concepts and relationships of the being-constructed modelling language. For instance, to create a metamodel which defines the abstract syntax of a modelling language for designing the database (i.e. a model for a particular aspect) of a software system, basic meta-concepts such as *Class*, *Property* of EMOF at layer M3, see Figure 2.7, will be instantiated to define modelling concepts such as *Table* and *Column*, by means of a reflection mechanism from the object-level to the concept-level. These concepts are connected by relationships, e.g. the *ownedColumns* association (i.e. an instance of the *Property* concept of EMOF) from the *Table* concept to the *Column* which is coupled with an inverse one. Relationships may also express composition semantics, by setting the *isComposite* attribute to *true*, for example in the case of the *ownedColumns* association, whereby *Column* instance-objects are composed by a *Table* instance-object to control object life-cycle. In addition, additional constraints can be added to the modelling language for some specific requirements, usually specifying in OCL. In this example, *Table* instances must have unique names and *Column* instances belonging to the same *Table* instance ought to have exclusive names, etc.

In a similar way, models which appear at layer M1 can be expressed by instantiating the concepts in a modelling language whose abstract syntax is defined by a certain metamodel at the layer M2. For instance, in the case of the Database Schema modelling language, an user-defined model in this language will represent the database designed for the being-developed software system. This model, in fact, contains objects which are instances of the concepts of the metamodel. This nature

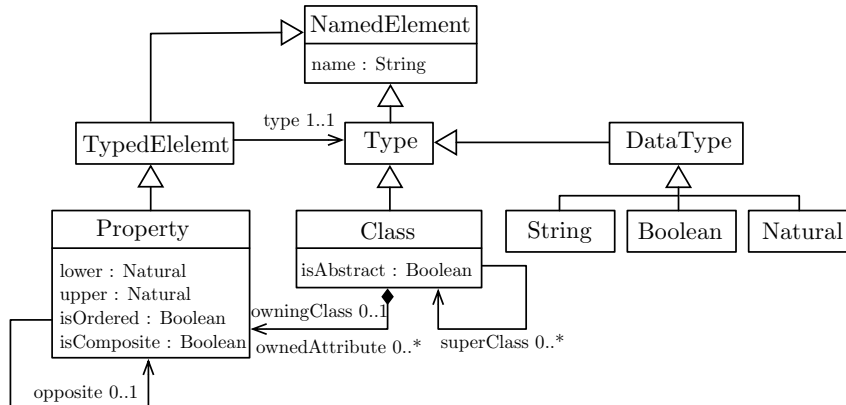


Figure 2.7: Excerpt of the EMOF meta-model.

of instantiation is given by the metamodeling language, this case is MOF, which also provides basic facilities to manipulate models at the metadata-level such as introspection, intercession, etc. Of course, a particular modelling language need to be added additional constraints into the structural definition of its metamodel in order to validate a model. These constraints can be described in the OCL language of OMG-MOF.

At the bottom level, layer M0 usually represents the run-time data, which could be considered a *token model* of a system under-study. When systems are operated, they change their states themselves. However, the “*instance of*” relation of this level against its higher level i.e. layer M1, is not common to all other levels in the modelling world. *Token models* at this level is often checked by complex (structural and non-structural) constraints in a specific technical platform, e.g. a database management platform such as Oracle [Oracle 11] or MySQL [MySQL 11], while models at other higher levels can be verified by means of a simpler manner by syntactic checking combined with OCL descriptions and an OCL query engine.

As the previous presentation, the OMG-MOF modelling framework has formalized a mechanism to construct a hierarchical modelling world, presenting three layers used in most cases when modelling a complex software system. With respect to the main objective of MDE, i.e. filling the problem-implementation gap by means of modelling, the most important contribution of the OMG-MOF hierarchy is an initiative document to optimize the understanding of bringing the modelling idea to the development of model-driven systems, i.e. systems used to develop systems. Other more philosophical discussions in detail about modelling and model-driven engineering, including model relationship kinds and number of modelling-layers can be found in [Seidewitz Ed 03], [Bézivin 04] and [Kuhne 06].

2.2.6 Domain Specific Modelling Language

As aforementioned in Section 2.2.1, having learned lessons from the failure of CASE tools, i.e. using “one-size-fits-all” graphical representations is not effective

to support many application domains because of too generic and non customizable characteristics. MDE approaches today combine Domain Specific Modelling Languages (DSML) with transformation engines and generators [Schmidt 06]. A DSML is a modelling language that is developed to express the requirements of a particular business domain or the solutions of a certain technological domain. The basic use of DSMLs in MDE approaches is that they foster the separation of concerns. When dealing with a given system, one may observe and work with different models of the same system, each one characterized by a given metamodel of a certain modelling language [Bezevin 05]. Currently, this usage is realized in a variety of MDE approaches like Model Driven Architecture (MDA) [MDA 03], Software Factories [Greenfield 03]. In the following, we present MDA approach proposed by OMG and Software Factories realized by Microsoft which are the most prominent representatives of the use of DSMLs.

Model Driven Architecture

Model Driven Architecture is a trademark of OMG that specifies an MDE approach to build systems using models. Software development solutions under the guideline of the approach MDA start with a description of the problem in terms of the problem domain and then apply chains of transformations, which lead to a model specifying the solution in terms of the solution domain. MDA is not a single specification, but is a set of OMG standards like MOF, UML, SPEM, QVT, XMI, etc.

According to [MDA 03, Kleppe 03], MDA initiatives described MDA as an MDE approach building on a set of different kinds of artifacts: Computation Independent Models (CIMs), Platform Independent Models (PIMs), Platform Specific Models (PSMs), Platform Models (PMs) and executable code, in which the following three kinds of models [Swithinbank 05] are used in most current MDA-based solutions:

- **Computation Independent Model (CIM):** A CIM is a view on a system from the computation independent viewpoint. A CIM model describes the business requirements of a being-developed (or -maintained) system and the business context in which the system will be used. The model specifies what a system will be used for, but does not show how it is implemented, neither does it show the structure of the system. The model is often expressed in business domain specific modelling languages with the vocabulary of the domain that is familiar to the practitioners. By this way, CIMs plays an important role where it bridges the gap between the experts of the application domain and its requirements, and the experts of the design domain and construction of the artifacts of the system.
- **Platform Independent Model (PIM):** A PIM is a view on a system from the platform independent viewpoint. A PIM model describe how the system will be constructed, but without reference to the technologies used to implement the system. The model does not specify the mechanisms used to build the solution targeted to a specific platform. However, it exhibits a specified degree of independence with certain platform and it may be more suited to be

implemented by one platform rather than another, or it may be suitable for implementation on many platforms.

- **Platform Specific Model (PSM):** A PSM is a view on the system from the platform specific viewpoint. A PSM is a model of a solution from a particular platform perspective. It includes both the details from the PIM that describe how the CIM can be implemented, and details specifying how the implementation is realized into a specific platform.

The key concept of MDA is the transformation from PIMs to PSMs for particular platforms. The process of transforming model of a modelling language to another one of another language is called model transformation. When specifying a model transformation, it is possible to integrate additional information that is described in Platform Models (PM). Figure 2.8 illustrates a high-level view on a transformation of a PIM, possibly with a PM, into a PSM, in which the empty bounding-box may be additional information playing a role as PM.

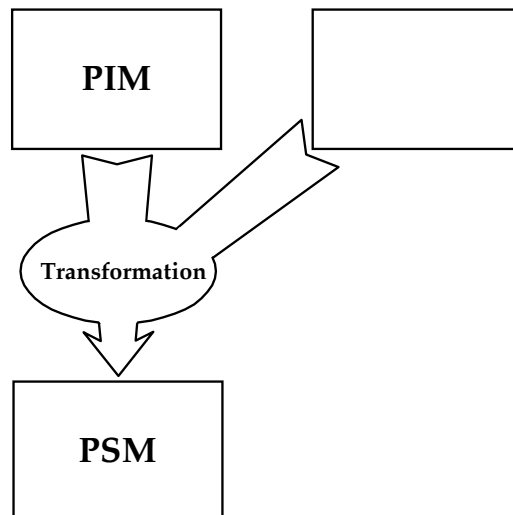


Figure 2.8: Transformation of a PIM to PSM (from [MDA 03]).

The MDA specification [MDA 03] does not treat these kinds of model as fixed layers. That means PIMs and PSMs can be layered with each model being considered as a PSM with respect to a more abstract-level model and a PIM with respect to a more concrete-level model. For instance, we could have start building a software with a high-level design model, then transform it into a detailed design model, and finally to an implementation model. At each moment using the MDA transformation pattern, we introduce further assumptions regarding the lower-implementation platform. In this example, the detailed design model plays as a PSM with respect to the high-level design model and as a PIM with respect to the implementation model.

In MDA, (domain specific) modelling languages for specifying above kinds of models should be developed by the means of UML profiles or Meta Object Facility (MOF). The current guideline of MDA promotes to develop modelling languages in using of MOF instead of UML profiles [MDA 03]. The use of MOF as the common

meta-modelling language to describe modelling languages has a lot of advantages. As MOF is an OMG's standard language to define metadata [MOF 06], it ensures interoperability to exchange models and metamodels between MDA-based tools and the ability to realize automated model transformations. MDA-based approaches are not performed automatically by transformation engines. They use a mixture of manual and automatic transformations in a chain [MDA 03].

In summary, MDA provides a guideline to develop software development approaches on building models that represent domain concepts directly rather than computer technology concepts. These models are expressed in domain specific modelling languages. These modelling languages may either be built on top of general languages such as UML via the UML's profile mechanism (see more detail on using UML profile at [Moreira 04]) or through hierarchical metamodelling frameworks such as MOF. In using frameworks that explicitly model assumptions about the business domain and the technological environment, automated tools can analyse models for flaws and transform them into implementations in concrete technological platforms. This avoids the need to write a large amount of low-level code and eliminates the error-prone caused by manual programming [Booch 04].

Software Factories

The term *Software Factories* [Greenfield 03] has been coined by Microsoft and is an initiative towards MDE. This MDE approach promotes the use of DSLs (DSMLs), which are software development languages that have been developed for particular problem domains. In other words, a Software Factory is an IDE specifically configured for the efficient development of a specific kind of application, i.e. applications in a specific domain, e.g. applications in the Online eCommerce domain. According to [Greenfield 03], the main infrastructure elements of a Software Factory are a *software factory schema* and a *software factory template* which is based on the software factory schema. The software factory template configures extensible tools, processes, and content to form a product facility for the software product line.

Software factory schemas define viewpoints that are necessary for building a system of the respective domain. A common approach to realize this is to use a grid, as shown in Figure 2.9. In the figure, the columns define concerns, while the rows define levels of abstraction. Each grid cell defines a viewpoint from which we can build some aspect of the software. For each viewpoint, the schema identifies core artifacts that must be developed to produce an application. These are DSLs with editing tools to build models, refactorings that can improve models, or transformations that support mapping models. Once the schema has been constructed, we can populate it with development artifacts for a specific software product.

Such a schema is, therefore, a conceptual framework for separating the concerns in the respective domain, based on abstraction level or position of each concern in the architectural or development process. The schema also identifies the commonalities as well as the differences among the applications in the domain defined by the schema.

Domain Specific Languages	Business	Information	Application	Technology
Conceptual	<ul style="list-style-type: none"> ▪ Use cases and scenarios ▪ Business Goals and Objectives 	<ul style="list-style-type: none"> ▪ Business Entities and Relationships 	<ul style="list-style-type: none"> ▪ Business Processes ▪ Service factoring 	<ul style="list-style-type: none"> ▪ Service distribution ▪ “Abilities” strategy
Logical	<ul style="list-style-type: none"> ▪ Workflow models ▪ Role Definitions 	<ul style="list-style-type: none"> ▪ Message Schemas and document specifications 	<ul style="list-style-type: none"> ▪ Service Interactions ▪ Service definitions ▪ Object models 	<ul style="list-style-type: none"> ▪ Logical Server types ▪ Service Mappings
Implementation	<ul style="list-style-type: none"> ▪ Process specification 	<ul style="list-style-type: none"> ▪ DB schemas ▪ Data access strategy 	<ul style="list-style-type: none"> ▪ Detailed design ▪ Technology dependent design 	<ul style="list-style-type: none"> ▪ Physical Servers ▪ Software Installed ▪ Network layout

Figure 2.9: A grid for categorizing development artifacts (from [Greenfield 03]).

Software factory templates implement software factory schemas in the previous step. Schemas are basically on paper, i.e. describing the assets used to build a family of products, thus we do not actually have these assets yet. Hence, we must implement the software factory schema by defining the DSLs, patterns, frameworks, and tools that are described in the documented schema. Then, these assets need to be packaged and made available to application developers. All these assets form a software factory template. They include code and metadata that will be loaded into an Interactive Development Environment (IDE, usually Visual Studio) to configure the IDE for developing the respective product family. Since assets of a software factory template are also software artifacts, we need to have the necessary tools to build them. For DSLs development, tools for defining metamodels, concrete syntax, and transformation are required to develop a software factory.

Summing up, similar to MDA approach, the Software Factories approach promotes the direct representation of software artifacts by working with (business or technical) domain concepts and advocates the automation by transforming models between DSLs. The main difference between them is the Software Factories approach is not based on open standards, like UML and MOF as the MDA approach.

DSMLs in MDA and Software Factories

As mentioned in [Bezevin 05], DSMLs facilitate the separation of concerns in software system development. The use of DSMLs helps the software developers in observing and working with different models of the same system, each one characterized by a given metamodel of the respective modelling language.

However, there is a little difference between MDA and Software Factories in the use and development of DSMLs. The MDA approach treats UML, which is a general purpose modelling language for modelling software system, as the modelling language of choice for most of application modelling, certainly with customization and extension via the UML Profiles mechanism [UML 05]. Nevertheless, the MDA approach also acknowledges the value of custom languages in certain specialized circumstances. This is the purpose of OMG Meta-Object Facility (MOF) [MOF 06] standard that plays an important role to specify metamodels for DSMLs. UML itself is defined using MOF and there are also other MOF-defined metamodels for other languages. MDA acknowledges the value of non-UML based DSMLs as a necessary technique to be applied judiciously [Swithinbank 05]. Two possible approaches for defining domain specific modelling languages are illustrated in Figure 2.10.

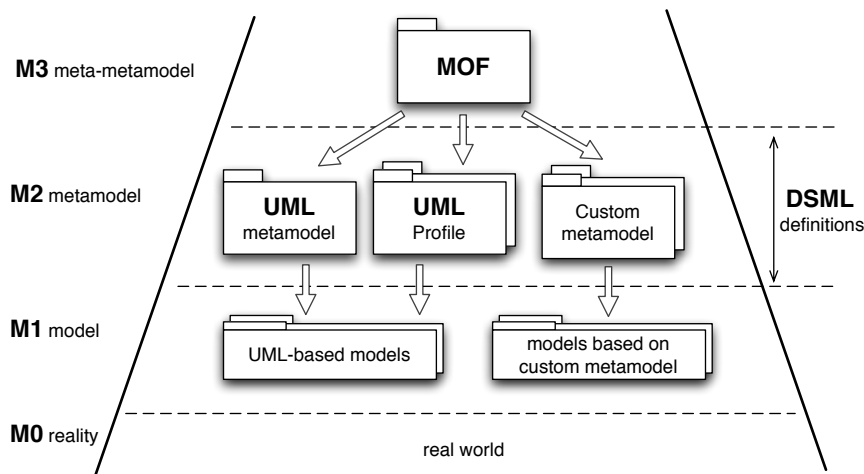


Figure 2.10: DSML definition approaches proposed by OMG-MDA.

In contrast to MDA, the Software Factories approach suggests using UML for developing sketches, white boarding and documentation, i.e. UML conceptual drawings that do not directly relate to code. Instead, it recommends that non-UML based DSMLs should be used for developing models at precise abstractions from which code is generated or mappings between DSMLs [Microsoft 05]. Each DSML is positioned in a cell of the table in Figure 2.9.

Each approach has its advantages and disadvantages. Defining a new non-UML based language will produce notations that will perfectly fit the concepts and nature of the specific domain. Nevertheless, this language does not respect semantics in UML. As a consequence, available commercial UML tools cannot be used for drawing diagrams, generating code, reverse engineering, etc. In contrast, using UML Profiles to define metamodels may not provide such an elegant and perfectly matching notations as may be required for systems of the domain [Moreira 04]. Thus, decisions to create a new language from scratch or to define a set of extensions to the standard UML metamodel by using the UML Profiles mechanism are not always easy. According to [Swithinbank 05], there are arguments that illustrate further advantages of defining new modelling language from scratch. Advantages of this alternative are:

- Non-UML based DSMLs are designed to specific user groups, application domains, context, and user groups. For that reason, it is easier for users to define models since the language provides exactly the concepts of the domain that they need for modelling. While UML Profiles only permit a limited amount of customization. It is not possible to introduce new modelling concepts that cannot be expressed by extending existing elements of the UML metamodel.
- The semantics of non-UML based DSMLs is better understandable to the experts of the application domain. Even to different stakeholders, it is easier to interpret models right or the same way.
- The scope of non-UML based DSMLs is optimized to its application domain and use. These languages guide the users towards directly certain types of solutions. While the use of UML Profiles does require familiarity with UML modelling concepts. In many cases, experts of the domain may have knowledge that could be used for purposes of code generation, but they may not have the expertise to express these concepts using UML's concepts.

In this thesis, we restrict the reuse problem of transformations only for DSMLs which are not based on the UML Profiles mechanism. As in the Figure 2.10, DSMLs considered in this thesis are MOF-defined languages. More precisely, the metamodels of these languages are designed in Eclipse Modelling Framework (EMF) [EMF 04], an open source component of Eclipse, which provides a concrete implementation of MOF, i.e. Ecore. It generates a Java implementation for working with the respective MOF-defined language and basic Eclipse tooling to create instance models of the language. Moreover, it is possible to design full graphical editors for the language. In the following section, we will present the overview of model transformation and existing approaches developed to define transformations. These approaches are synthesized in different categories based on some previous review works in the literature.

2.2.7 Model Transformation

As aforementioned (see Section 2.2.6), MDE approaches like Model Driven Architecture [MDA 03], Software Factories [Greenfield 03] use models as first-class entities in driving software development processes. In these approaches, model transformations play a key role since they make the seamless transition in the model space. However, model transformations are a contentious topic, partly because they are not very well understood, and partly because their merit in practical model-driven development scenarios is not very clear [Stahl 06].

Model transformations become an important mechanism since it bridges some abstraction gaps that occur in MDE, e.g. vertical model transformations refine abstract models to more concrete models while horizontal model transformations describe mappings between models of the same or equivalent level of abstraction. Thus, it is not surprising that there are many attempts to develop transformation languages in both academia, open-source and commercial tools. According to [Czarnecki 06], these languages are currently used in various application scenarios, including:

- Generating lower-level models, (and eventually code,) from higher-level models [Sendall 03, Kleppe 03]
- Mapping and synchronizing among models at the same level or different levels of abstraction [Ivkovic 04]
- Creating query-based views of a system [Bull 05, Solberg 05]
- Model evolution tasks such as model refactoring [Sunye 01, Zhang 05]
- Reverse engineering of higher-level models from lower-level models or code [Favre 04]

In what follows, we introduce model transformations, requirements, characteristics, related approaches and tools based on the works in [Gardner 03] and [Czarnecki 03, Czarnecki 06, Mens 06], respectively.

Model-to-model transformation – referred to as M2M transformation – is *information* on mapping from one model to another, described by a *transformation engineer*, in order to automate a translation process by means of a *transformation engine*. An M2M transformation reads a model to create another model. However, the output model is typically based on a different metamodel than the input model. The information encoded in such transformations is the descriptions on how the modelling concepts defined in the source metamodel are mapped to the modelling concepts defined in the target metamodel.

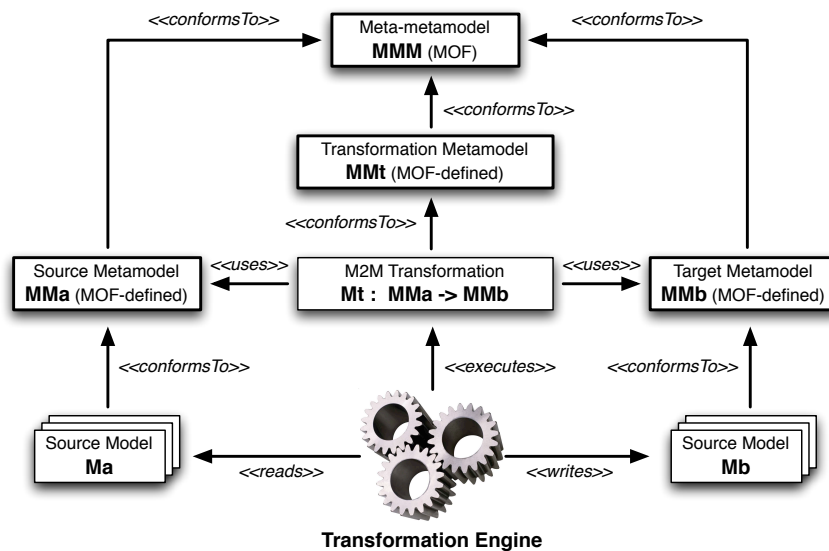


Figure 2.11: Basic concepts of M2M transformation frameworks.

Since model transformations are defined to transform concepts of modelling languages, they must be defined with respect to the metamodels, as shown in Figure 2.11. In this figure, when the transformation engine executes a transformation Mt ,

input models like Ma conforming to the source metamodel MMa are transformed to output models like Mb conforming to the target metamodel MMb . In general, model transformations can be defined based on multiple source and target metamodels. Since model transformations themselves are models [Bezevin 05], they conform to a metamodel MMt which represents a model transformation language. This kind of metamodel can be defined on the basis of MOF, like other ordinary metamodels.

Requirements on Transformation Languages

Before we delve into the details of model transformation approaches, it is useful to discuss some important and perhaps obvious requirements for model transformation languages and their implementations. These requirements have been specified in the literature, such as in [Sendall 03, Gardner 03, Czarnecki 06, Mens 06]. Authors in [Gardner 03] provided a review on the submissions and recommendations of QVT-RFP [QVT-RFP 05] and proposed concepts and requirements that they thought necessary for a final adopted standard of the OMG-QVT language. From this mentioned work, we introduce some of the most important items which provide an overview of necessary requirements for application scenarios that transformation languages should fulfil.

- Transformations should be written in a *simple manner*. A simple transformation is one that transforms single elements of the input model into single elements of the output models.
- Transformation engines may have to support the *trace of transformation executions*. Transactional transformations should be definable (commit, rollback), which prevent an invalid model resulting from a transformation that has failed during execution.
- Transformation should support the *propagation of changes* occurring in one model to the other model. By this way, the changes made manually by model designers can be maintained.
- Transformations should be able to *implement incremental updates*, i.e. the target model is the same as the source model.
- Transformation languages should support the mechanism of *reuse and extension* of generic transformations. It should provide mechanism to inherit and override transformations and the ability to instantiate templates or patterns.
- The use of additional transformation data not contained in the source model, but *parametrized transformation process* should be possible.
- Transformation languages should allow *composing, decomposing transformations* to support modularity.
- *Bidirectional mapping* should be supported.

- A transformation language should allow to specify *many-to-many transformations* with multiple source and target models.

Beneath the request of OMG and many practical needs, a variety of languages to model transformations have been proposed (and developed) over the last decade. However, the mature status of model transformation frameworks are still not available, and this area of research continues to be a contentious subject. One is suggested referring to [Czarnecki 06] for the list of model transformation frameworks which have been published in literature and implemented in open-source and commercial tools.

Characteristics of Model Transformations

Since model transformations have been applied to various scenarios in model-driven software development, they have many different characteristics. Authors in [Mens 06] proposed a taxonomy of model transformations based on a multi-dimensional classification which allows to characterize model transformations, to group and compare between them. These dimensions are:

- **Endogenous versus Exogenous:** *Endogenous* transformations are transformations between models expressed in the same modelling language. *Exogenous* transformations are transformations between models expressed using different languages. In [Visser 01], endogenous transformations are also called *rephrasing*, whereas *translations* are referred to as exogenous transformations. Typical examples of translation are *synthesis* of a higher-level specification into a lower-level one, *reverse engineering*, i.e. the inverse of synthesis, and *migration* from a program written in one language to another. Typical of rephrasing are *optimization* (i.e. improving certain operational qualities, while preserving the intended semantics), *Refactoring* (i.e. changing internal structure of software without changing its observable behaviour), *simplification* and *normalization* of models.
- **Horizontal versus Vertical:** *Horizontal* transformations are transformations where the source and target models reside at the same level of abstraction. Typical examples are *refactoring* (an endogenous transformation), and *migration* (an exogenous transformation) when changing the version of modelling languages. In contrast, *vertical* transformations are transformations where the source and target models reside at different abstraction levels. A typical example is *refinement*, where a specification is gradually refined into a full-fledged implementation, by means of successive refinement steps that add more concrete details. Some concrete examples of application scenarios classified into above dimensions are given in the Table 2.1.
- **Syntactical versus Semantical:** *Syntactical* transformations are transformations that merely transform the syntax while *semantical* transformations take the semantics of the model into account and transform it through a more sophisticated way. A typical example of *syntactical* transformations are *imports* or *exports* of models in a specific format. Another example is a parser

	horizontal	vertical
endogenous	<i>Refactoring</i>	<i>Formal refinement</i>
exogenous	<i>Language migration</i>	<i>Code generation</i>

Table 2.1: Two dimensions of transformations with examples [Mens 06].

which transforms the concrete syntax of a program (resp. model) in some programming language (resp. modelling language) into an abstract syntax. Then, a *semantical* transformation use the abstract syntax (i.e. the internal representation of the program) to apply *refactoring* or *optimization*.

In addition to the above dimensions, model transformations also have other quantitative characteristics such as:

- **Automation:** This characteristic is observed to distinct between model transformations that can be automated and model transformations that need to be performed manually (or at least need a certain amount of human intervention). The typical example of the latter is a transformation from requirement documents to a formal analysis model. As the requirements are usually expressed in natural language, it is necessary to interpret the ambiguity, incompleteness and inconsistency of the document by a manual intervention.
- **Complexity:** Model transformations in different scenarios may differ in their complexity. Some transformations, such as simple model mappings or model refactoring, could be considered as small, while others, like parsers, compilers and code generators, are much more complex. As a consequence, different scenarios of transformation application can require an entirely different set of techniques and tools.
- **Preservation:** Although there is a wide range of different kinds of transformations. However, each transformation should preserve certain aspects of the source model in the transformed target model, depending on the kind of transformation. For example, the (external) behaviour of a program (or model) is needed to be preserved in refactoring, while the structure is modified. The technical space also heavily influences what needs to be preserved. For instance, in case of a program transformation, we need to preserve the syntactic well-formedness and the type correctness of the program.

The above taxonomy has introduced the important characteristics of model transformations to give an overview of the requirements in the research domains of model transformation. In what follows, the detailed features of existing transformation languages will be presented via a hierarchical classification based on features diagrams. The following parts are synthesized from the work of Czarnecki and Helsen in [Czarnecki 06].

Features of Model Transformation Approaches

The earlier result of Czarnecki *et al.* on the variability of existing model transformation languages was published in [Czarnecki 03]. The publication described and clarified the most relevant terms and concepts to model transformations that have been represented by using feature diagrams [Kang 90]. This taxonomy of transformation languages was revised in [Czarnecki 06]. An overview of key features of model transformation languages is shown in the feature diagram (from [Czarnecki 06]), at the second layer, in Figure 2.12. Note that this feature diagram treats *model-to-model* and *model-to-text* approaches uniformly. We will distinguish them later in the classification of the “**Categories of Model Transformation Approaches**” section. These key features in Figure 2.12 are summarized as follows:

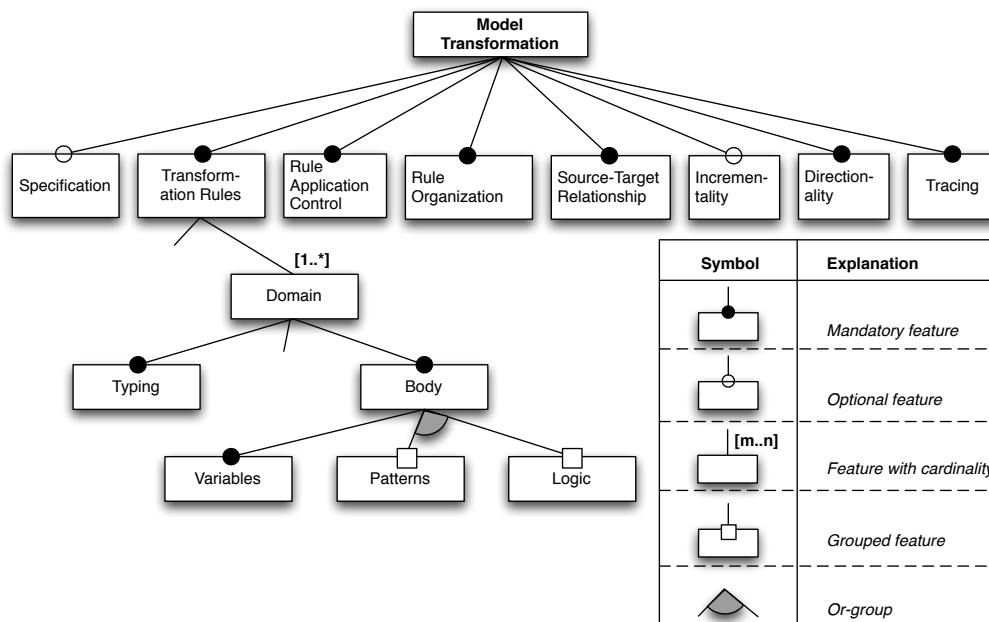


Figure 2.12: An excerpt of the feature diagram of model transformation languages (for more details, see [Czarnecki 06]).

- **Specification:** Some transformation approaches provide a dedicated specification mechanism, like preconditions and postconditions expressed in Object Constraint Language (OCL) [OCL 06]. A particular transformation specification may represent a function between source and target models and be executable or describe relations and are not executable.
- **Transformation Rules:** Transformation rules are understood as a broad term describing the smallest units of transformations. Rewriting rules are an example where a left-hand side (LHS) accesses the source model, while a right-hand side (RHS) expands the target model. In addition, transformation rules could be functions, procedures or even templates.

Transformation rules are defined over *domains*, see the third layer of the diagram. A *domain* is a part of a rule responsible for accessing one of the models on which the rule operates. Rules usually have a source and a target domain, but they may also involve more than two domains. In this case, a set of domains can be seen as one large composite domain; however it is useful to distinguish among individual domains to meet the modularity requirement when writing transformations.

Each domain has a *body* which can be divided into three subcategories: *variables*, *patterns*, and *logic*. Variables may hold elements from the source and/or target models (or sometime intermediate elements). *Patterns* are model fragments with zero or more *variables*. Sometimes, as in the case of using templates to define rules, *patterns* can have not only *variables* embedded in the *body*, but also expressions and statements of the transformation language. *Patterns* can be strings, terms, or graph patterns, depending on the internal representation of the models being transformed. *Logic* expresses computations and constraints on model elements.

Variables and *patterns* defined in a transformation can be typed. Typing rules may be defined *syntactically* or *semantically*. In the case of syntactic typing, a *variable* is associated with a metamodel element whose instances it can hold, while semantic typing allows stronger properties to be asserted, such as well-formedness rules (static semantics) and behavioral properties (dynamic semantics). A type system for a transformation language could statically ensure for a transformation that the models produced will satisfy a certain set of syntactic and semantic properties, provided the input models satisfy some syntactic and semantics properties.

- **Rule Application Control:** Control mechanisms usually provide strategies for two aspects: *location determination* and *scheduling*. Location determination is the strategy for determining the model locations to which transformation rules are applied. Scheduling mechanisms determine the order in which individual transformation rules are executed.
- **Rule Organization:** This comprises the structuring and composing issues of transformation rules, such as modularization mechanisms (e.g. packaging rules into modules and allowing to import them into other modules) and reuse mechanisms (e.g. rule inheritance, module inheritance).
- **Source-Target Relationship:** This feature is concerned with issues such as whether source and target are one and the same model or two different models. For example, ATL [Bézivin 03a, Jouault 06] mandates the creation of a new target model that has to be separated from source model. Nevertheless, one can simulate an in-place transformation in ATL through an automatic copy mechanism. Some other approaches, such as VIATRA [Varró 02, Varró 04] and AGG [Taentzer 04], support only in-place update; that is, source and target are always the same model. Yet, other approaches, such as QVT Relations [Bast 05]

and MTF [IBM 05], allow creating a new model or updating an existing one, in which QVT Relations also supports in-place update.

- **Incrementality:** This feature refers to the ability to update existing target models based on changes in the source models. It involves three different features: *target incrementality*, *source incrementality*, and *preservation of user edits in the target*; in which, the first one is the basic feature of all incremental transformations. In these transformations, target models are created if they are missing on the first execution. A subsequent execution with the same source models as in the previous execution has to detect that the needed target elements already exist. This can be achieved by using traceability links. When any source models are modified and the transformation is executed again, the necessary changes to the target are determined and applied, while other target elements are preserved.
- **Directionality:** Transformations may be *unidirectional* or *multidirectional*. Unidirectional transformations could be executed in one direction only, in which case a target model is computed (or updated) based on a source model. Multidirectional transformations can be executed in multiple directions, which is particularly useful in the context of model synchronization. These transformations can be achieved by using multidirectional rules or by defining several separate complementary unidirectional rules, one for each direction.
- **Tracing:** This is concerned with the mechanisms for recording different aspects of transformation execution, such as creating and maintaining trace links between source and target model elements. *Tracing* can be understood as the run-time footprint of transformation execution where it uses *traceability links* to connect source and target elements. *These links* can be established by recording the transformation rule and the source elements that were involved in creating a given target element. Trace information is useful in performing impact analysis (i.e. analyzing how changing one model would affect other related models), determining the target of a transformation as in model synchronization, model-based debugging (i.e. mapping the stepwise execution of an implementation back to its high-level model), and debugging model transformation themselves.

We have presented essential features provided by existing transformation languages that are based on the survey from 2006 [Czarnecki 06]. Although there have been many other transformation languages (or transformation frameworks) developed until now, they still comply with these features. The following section classifies model transformations approaches into “major categories”, depending on which paradigm they use to describe transformations rules.

Categories of Model Transformation Approaches

The previous section introduced necessary terms and concepts describing major features that were provided in existing transformation approaches. We now

present a classification of model transformation approaches, which is also based on [Czarnecki 06]. At the top level, transformation approaches are distinguished between *model-to-model* (M2M) and *model-to-text* (M2T) approaches. A model-to-model transformation creates its target as an instance of the target metamodel, while the target of a model-to-text transformation is just string. Model-to-text transformations are also referred to as *model-to-platform* or *model-to-code* transformations [Stahl 06]. An overview of categories is shown hierarchically in Figure 2.13. In following parts, each category will be explained with examples of concrete realizations.

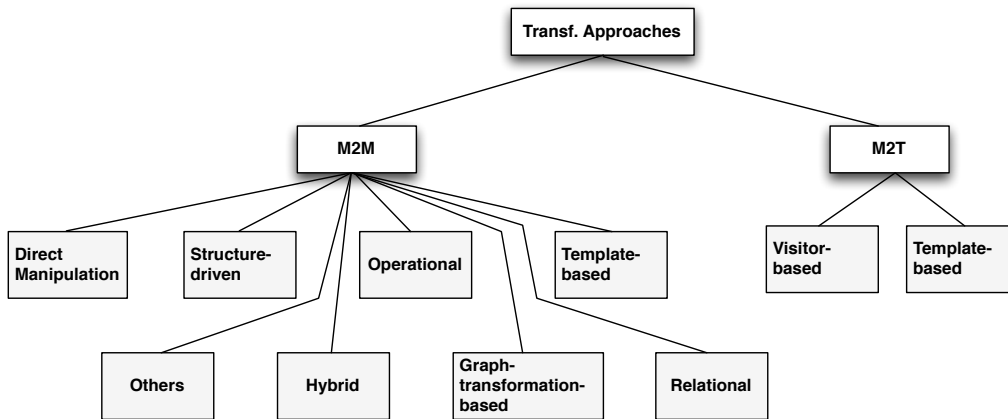


Figure 2.13: Categorization of model transformation approaches.

Model-to-model approaches

The model-to-model category can be distinguished among direct-manipulation, structure-driven, operational, template-based, relational, graph-transformation-based, and hybrid approaches.

- **Direct manipulation approach:** Approaches in this category offer an internal model representation and some APIs to manipulate it. A typical example of such a category is Java Metadata Interface (JMI) [JMI 02]. These approaches are usually implemented as an object-oriented framework, which may also provide some minimal infrastructure, i.e. a set of abstract classes and interfaces to be used in writing transformations. Using frameworks of this approach, users themselves usually have to implement transformation rules, scheduling, tracing, and others facilities, mostly from the beginning, in a programming language, such as Java.
- **Structure-driven approach:** This category of approach has two distinct phases: The first phase is concerned with creating the hierarchical structure of the target model; whereas, the second phase sets the attributes and references in the target. Rule scheduling and application strategy is determined by the overall framework; users are only concerned with providing transformation rules. An example of such a category is the M2M transformation framework provided by OptimalJ [Compuware 05]. This framework provides incremental

copiers that users have to subclass to define customized transformation rules. Rules are not allowed to have side-effects, and scheduling is completely determined by the framework.

- **Operational approach:** Approaches in this category are similar to *direct manipulation* approaches, but offer more dedicated support for model transformation. A typical solution in this category is to extend the used metamodelling formalism, i.e. concepts in the meta-metamodel, with facilities for expressing computations. Object Constraint Language (OCL) [OCL 06] is an example providing a query language with imperative constructs extended from the constructs of the MOF meta-metamodel. Other examples in this category are QVT Operational mappings [Bast 05], MTL [Vojtisek 04], C-SAW [Gray 06], and Kermet [Muller 05]. Specialized facilities such as tracing may be offered through dedicated libraries.
- **(Model-)Template-based approach:** In this category, transformation rules are defined by model templates which are models with embedded metacodes that compute the variable parts of the resulting template instances. Model templates are usually expressed in the concrete syntax of the target language, which helps the transformation designers to predict the result of template instantiation. The metacodes can have the form of annotations on model elements. They are conditions, iterations, and expressions, all being part of the metalanguage. Thus, OCL constructs can be used in the metalanguage to express these annotations. A concrete model-template-based approach is given by Czarnecki and Antkiewicz in [Czarnecki 05].
- **Relational approach:** This category groups declarative approaches in which the main concept is mathematical relations. In general, relational approaches can be seen as a form of constraint solving. Examples of this category are QVT Relations [Bast 05], MTF [IBM 05], Kent Model Transformation Language [Akehurst 02, Akehurst 10], Tefkat [Gerber 02, Lawley 06], and AMW [Bézivin 05b]. The common idea in these approaches is to specify the relations among source and target elements' types using constraints. In its pure form, such a specification is non-executable. However, declarative constraints can be given executable semantics, as in logic programming. All of the relational approaches are side-effect-free and create target model elements implicitly. Most of them require strict separation between source and target models; that is, they do not allow in-place update.
- **Graph-transformation-based approach:** This category groups model transformation approaches based on the theoretical work on graph transformations [Taentzer 05]. In particular, this category operates on typed, attributed, labelled graphs [Andries 99], which can be thought of as formal representation of simplified metamodels in the MOF-defined format. Concrete examples include AGG [Taentzer 04], AToM3 [Lara 02], VIATRA [Varró 02, Varró 04], GReAT [Agrawal 03], UMLX [Willink 03], BOTL [Braun 03, Marschall 03], MOLA [Kalnins 05], and Fujaba [Fujaba 97].

Graph transformation rules have a left-hand side (LHS) and a right-hand side (RHS) graph pattern. The LHS pattern is matched in the model being transformed and replaced by the RHS pattern in place. The LHS often contains conditions in addition to the LHS pattern, for example, negative conditions. Some additional logic, such as computational logic in String and Numeric domains, is needed to compute target attribute values, e.g. the *name* attribute of model elements.

Graph patterns can be rendered in the concrete syntax of their respective source or target modelling language (e.g. in VIATRA) or in the MOF-defined abstract syntax (e.g. in AGG). The advantage of the concrete syntax is that it is more familiar to experts working with a given modelling language than the abstract syntax. However, it is easier to provide a default rendering for the abstract syntax that will work for any metamodel, which is useful when no specialized concrete syntax is available.

- **Hybrid approach:** Approaches in the hybrid category combine different techniques from the previous categories. These techniques can be combined as separate components, such as in QVT [Bast 05] with three components: Relations, Operational mappings, and Core; or in a fined-grained fashion at the level of individual rules as in ATL [Bézivin 03a, Jouault 06] and YATL [Patrascoiu 04]. An individual rule described in ATL may be fully declarative, hybrid, or fully imperative. In case of fully declarative, the LHS part consists of a set of syntactically typed variables with an optional OCL constraint as filter or navigation logic. The RHS contains a set of variables and some declarative logic to bind the values of the attributes in the target elements. In a hybrid rule, the source and target patterns are complemented with a block of imperative logic, which runs after the application of the target pattern. A fully imperative rule (so-called procedure) has a name, a set of formal parameters, and an imperative body, but no pattern.
- **Other approaches:** In addition to the described categories, there are two more approaches: transformation implemented using Extensible Stylesheet Language Transformation (XSLT) [W3C 99] and the application of metaprogramming to model transformation. Since models can be serialized as Extensible Markup Language (XML) using the XML Metadata Interchange (XMI) [OMG 05], model transformations can be implemented using XSLT, which is a standard technology for transforming XML documents. Nevertheless, the use of XMI and XSLT has scalability limitations. Manual implementation of model transformations in XSLT quickly leads to non-maintainable implementations because of verbosity and poor readability of XMI and XSLT. A more promising direction in applying traditional metaprogramming techniques to model transformations is a domain-specific language for model transformations proposed in [Tratt 06], which is embedded in a metaprogramming language.

Model-to-text approaches

Model-to-text approaches are not only useful for code generation but also for non-code artifacts such as structural documents. Code generation can be seen as a special case of model-to-model transformations if one provides a metamodel for the target programming language. Nevertheless, for practical of reusing existing compiler technology and for simplicity, code is often fed into a compiler. In the model-to-text category, we distinguish between *visitor-based* and *template-based* approaches.

- **Visitor-based approach:** A very basic code generation approach consists in providing some visitor mechanism to traverse the internal representation of a model and write text to text stream. A typical example for this category is Jamda [JAMDA 03] – an object-oriented framework providing a set of classes to represent UML models, an API for manipulating models, and a visitor mechanism (CodeWriters) to generate code. Jamda does not support the MOF-defined metamodels; however, new model element types can be introduced by subclassing the existing java classes that represent the predefined model element types. The most important benefit of using Jamda is the saving of a huge amount of time, i.e. it takes much less time than writing the code by hand from scratch.
- **Template-based approach:** Most of currently available MDA *model compiler* tools based on a templating approach, such as Xtend [Xtend 08], JET [Popma 05], AndroMDA [AndroMDA 04], MetaEdit+ [MetaCase 00], and OptimalJ [Compuware 05]. A template usually consists of the target text containing splices of metacode to access information from the source and to perform code selection and iterative expansion. There is no clear syntactic separation between the LHS and RHS of a rule. The LHS can be understood as executable logic to access source, and the RHS is the combination of untyped string patterns with executable logic for code selection and iterative expansion. Template-based approaches usually offer user-defined scheduling in the internal form of calling a template from within another template. Compared with a visitor-based transformation, the structure of a template resembles more closely the code to be generated. Templates lend themselves to iterative development as they can be easily derived from examples.

Summing up, there are a lot of model transformation frameworks based on various above categories. However, choosing an adequate approach in a specific context of transformation is not an easy task. According to the recommendation of Gardner *et al.* in [Gardner 03], for simple transformations and for identifying relations between source and target model elements, a relational approach should be used. In case of defining complex many-to-many transformations that involve detailed model analysis, an operational approach seems to be preferable. In practice, a variety of transformation languages, such as QVT or ATL, offer the possibility to combine elements of both approaches in transformation specifications.

2.3 Model Transformation Reuse

MDE approaches for software development are getting more sophisticated when they foster the description of software systems using appropriate modelling formalisms and at different levels of abstraction. For example, the MDA approach proposes developing software development approaches on building modelling languages that represent domain concepts directly rather than computer technology concepts, while Software Factories promotes the use of a set of specific modelling formalisms for each group of products in a particular problem domains. As a result, there is an explosion in developing specific modelling languages, serving the specific purposes of the particular application domains and technology platforms. The application and usage of Domain Specific Modelling Languages (DSMLs) is extremely efficient in the context of automated system and code generation. Hence, model transformation becomes a key enablers in MDE approaches to transform models seamlessly between different DSMLs and levels of abstraction.

Using DSMLs with specialized metamodels, i.e. different metamodels for the same modelling language of a domain, in software development scenarios has a limitation concerned with transformation developing. Since transformations are tightly coupled to the metamodels that they are defined upon, thus with the use of different metamodels for the language of the same domain, transformations already developed by an organization cannot be directly used in another organization. In such a situation, there is a need of customizing the legacy model transformation to avoid developing a new transformation from scratch. Moreover, with the recurrent evolution of metamodels in the organization, i.e. releasing a new version of a metamodel, it is necessary to adjust existing model transformations built over the old metamodel version.

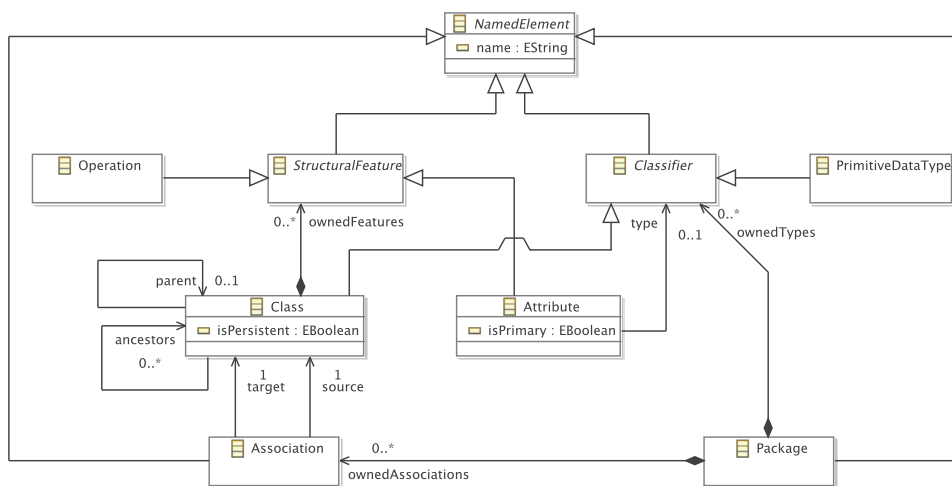
DSMLs are realized through metamodels that are an exact representation of domain concepts and a document describing the semantics of each element in the representation. Since model transformations are defined over the concrete representation of the metamodel, syntactic and semantic differences in representation formats hinder the efficient reuse of existing model transformations. These differences occur recurrently by the evolution and the use of different metamodels in organizations. Since transformations are also software artifacts, to enable their efficient reuse in different contexts (i.e. different metamodels), it is essentially to develop migrating solutions for model transformation reuse and evolution, independent of modelling languages.

In summary, a solution for automatically adjusting and reusing existing model transformations is necessary, namely in the context of the evolution of metamodels or the use of different metamodels. Such a solution helps avoid errors and tedious tasks of developing new transformations. In the following parts, first, we present a concrete case study to demonstrate the need of transformation reuse when using different metamodels of the same modelled domain. Second, we give a brief view of the transformation approach (language), i.e. MOMENT2-GT, which is further chosen to describe transformations in this thesis. This section will be enclosed by a discussion on the obstacles which hinder the reuse of model-to-model transformations.

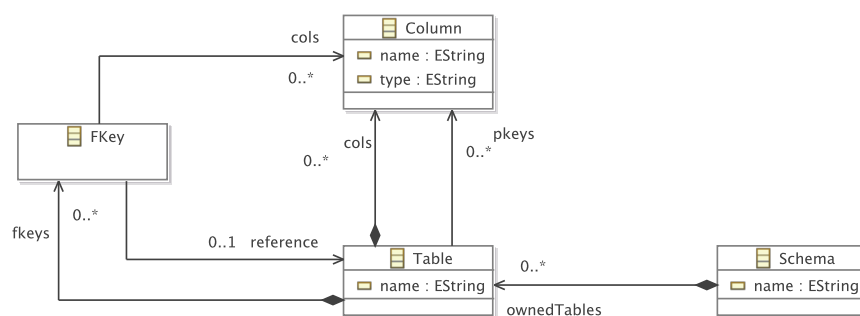
2.3.1 A Case Study

In order to illustrate the problem to be solved in this thesis, an exogenous model-to-model transformation in which class models are transformed into relational database models for a relational database management system (RDBMS) is under consideration as the motivating example. In the context of the Software Factories software development approach [Greenfield 03], one can consider this transformation as a facility for automatically generating models from the object models viewpoint of the Application concern at the logical layer into the database models viewpoint of the Information concern at the implementation layer, see the table in Figure 2.9 of Section 2.2.6.

Metamodels



(a) Source metamodel CDV1 in Ecore-defined representation



(b) Target metamodel RDB in Ecore-defined representation

Figure 2.14: Class models to relational database models transformation's metamodels.

Figures 2.14a and 2.14b show the source and target metamodels expressed graph-

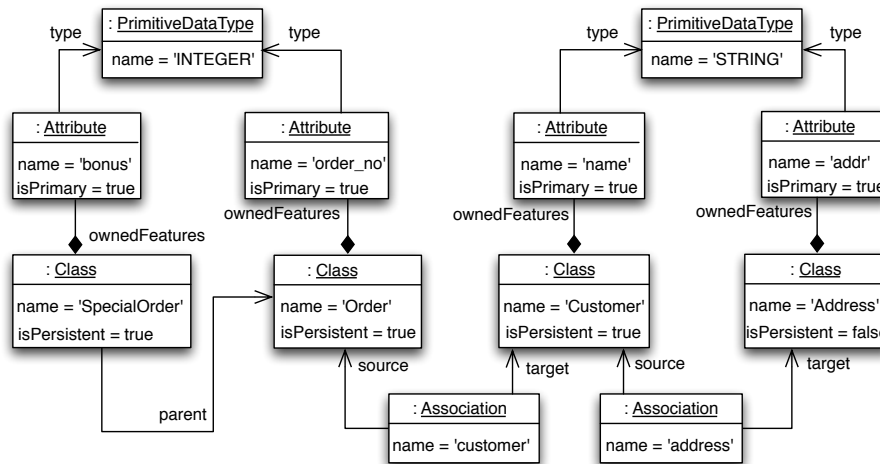
ically in UML class diagram notation, respectively. Note that all metamodels in this thesis are MOF-defined, More precisely Ecore [EMF 04] is used as a concrete implementation of MOF in Eclipse EMF modelling framework. Figure 2.14a gives a simplified metamodel for class models that includes the abstract concept of classifiers (*Classifier*), which comprises classes (*Class*) and primitive data type (*PrimitiveDataType*). It also presents the abstract concept of structural features (*StructuralFeature*), which comprises operations (*Operation*) and attributes (*Attribute*). *Classifier* and *StructuralFeature* concepts are inherited concept *name* owned by the abstract concept of named elements (*NamedElement*). There are inheritance links between concept Association, Package and the NamedElement concept. Packages contain classes, primitive data types, and associations. Classes contain attributes and operations. Associations define relationships between classes through concepts *source* and *target* owned by the association. Classes can be marked as persistent (by setting the value of attribute concept *isPersistent*) and so on attributes can be marked as primary or not by the value of attribute concept *isPrimary*. We call this metamodel CDV1 (also referred to as the *original metamodel* in this thesis) to distinguish with a *new metamodel* which defines the same or similar concepts of the domain of class models.

Figure 2.14b shows a simple metamodel for defining relational database schemas for a relational database management system. A schema (*Schema*) contains tables (*Table*), and tables contain columns (*Column*). Attribute concept *type* of concept Column has values of string. Every table has primary-key columns, to which are pointed by the reference concept *pkeys* which is owned by concept Table. Additionally, the concept of foreign-keys is modelled by FKey which relates foreign-key columns to tables.

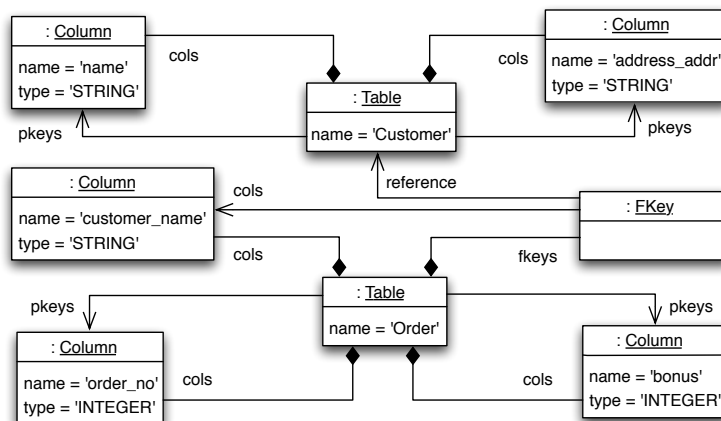
Sample Models

Sample instance models of the metamodels using the UML object diagram notation are shown in Figures 2.15a and 2.15b. The instance model in Figure 2.15a represents a class model with one package, for simplicity we do not show the package object in the diagram, containing four classes with values of attribute concept *name*: ‘Customer’, ‘Address’, ‘Order’, and ‘SpecialOrder’. While classes Customer, Order, SpecialOrder are persistent by setting values of attribute concept *isPersistent* to *true*, class Address is not. This model also defines two primitive data types, ‘INTEGER’ and ‘STRING’, to set values of reference concept *type* for attributes: ‘addr’, ‘name’, ‘order_no’, ‘bonus’, owned by classes Address, Customer, Order, SpecialOrder, respectively. The links between class Customer and classes Address, Order are represented by association named ‘address’ and ‘customer’, respectively. The value of reference concept *parent* of class SpecialOrder is set to class Order to establish the inheritance relation between them.

Figure 2.15b shows the instance model of the relational database model. The instance model represents a schema, same as the above package, the schema object is not shown in the diagram, that can be used to make Customer, Order objects persistent. This model defines two tables, i.e. Customer and Order. Customer table contains two columns, i.e. ‘name’ and ‘address.addr’ with the type values are



(a) Sample input model in CDV1-defined representation



(b) Sample out model in RDB-defined representation

Figure 2.15: Sample input/output models.

‘STRING’, that are also primary keys of the table. Order table has two primary keys, i.e. ‘order_no’ and ‘bonus’. This table also has a foreign key object which refers to the ‘customer_name’ column. The foreign key refers to the Customer table. To achieve this model from the class model, it is necessary to describe a transformation between two metamodels. The informal description of this transformation is presented as follows.

Class Models to Relational Database Models

As a case study, we consider transforming class models into relational database models described in the previous paragraphs. Such a transformation needs to realize at least the following main mappings:

1. Package-to-Schema: The root package in the class model should be mapped to

a schema with the same name as the package.

2. Class-to-Table: A persistent class should be mapped to a table with the same name as the class and all its attributes or associations to columns in the table. If the type of an attribute (represented by reference concept *type*) or of an association (represented by reference concept *target* of the type concept Association) is another persistent class, a foreign key to the corresponding table is established. Furthermore, the table should have primary-key columns corresponding to transformed attributes marked as primary.
3. Attribute-to-Column: The class attributes have to be appropriately mapped to columns, and some columns may need to be related to other tables by foreign key definitions.

In addition, if class hierarchies are transformed, only the top classes are mapped to tables. Additional attributes and associations of subclasses result in additional columns of the tables corresponding to these top classes. To this end, it is necessary to compute the transitive closure of class inheritance. The result closure is stored in the value of the helper reference concept *ancestors* of subclasses. In the end, attributes and associations of subclasses are transformed to columns of the top class' table.

In the transformation, non-persistent classes are not mapped to tables. However, it is necessary to preserve all information in the class model after transforming into the relational database model. That means all attributes and associations of non-persistent classes are distributed over tables stemming from persistent classes which access non-persistent classes. To this end, it has to transform all attributes and associations of connected non-persistent classes to columns of referring persistent classes' tables.

The above transformation would map the class model in Figure 2.15a to the relational database model in Figure 2.15b. The result model has a schema, which is not shown in the UML object diagram for simplicity, with the same name as the package. The schema consists of two tables, one from the *Customer* class and the other from class *Order*. The primary attribute *name* of class *Customer* is mapped directly to the column *name* with the type is 'STRING' and is set as a primary key of table *Customer*. Furthermore, the attribute *addr* of non-persistent class *Address* is mapped to column *address_addr* of table *Customer* because there is an association *address* from class *Customer* to class *Address*. These results are handled by mappings Class-to-Table and Attribute-to-Column and the above rule considering non-persistent classes. Similarly, table *Order* has column *order_no* corresponding to the result of the Attribute-to-Column mapping. Moreover, the *bonus* column of the table is achieved from attribute *bonus* of class *SpecialOrder* by taking into account the inheritance relation with class *Order*. In the end, a foreign key is created that points to column *customer_name* derived from the *customer* association between two persistent classes, *Order* and *Customer*.

Changing metamodel

As a case study, we consider the case in which the original source metamodel of the transformation will be replaced by another one, but always representing the same or similar modelling concepts of the modelled domain. More precisely, one wants to transform the same intentional class model (as shown in Figure 2.17), however, designed by another CASE tool based on a different metamodel, without developing a new transformation from scratch. This situation usually occurs when different companies often use different modelling languages and tools to model problems in their owned software development factory or in case of evolution of metamodels in the same company, see Figure 1.1 of Chapter 1.

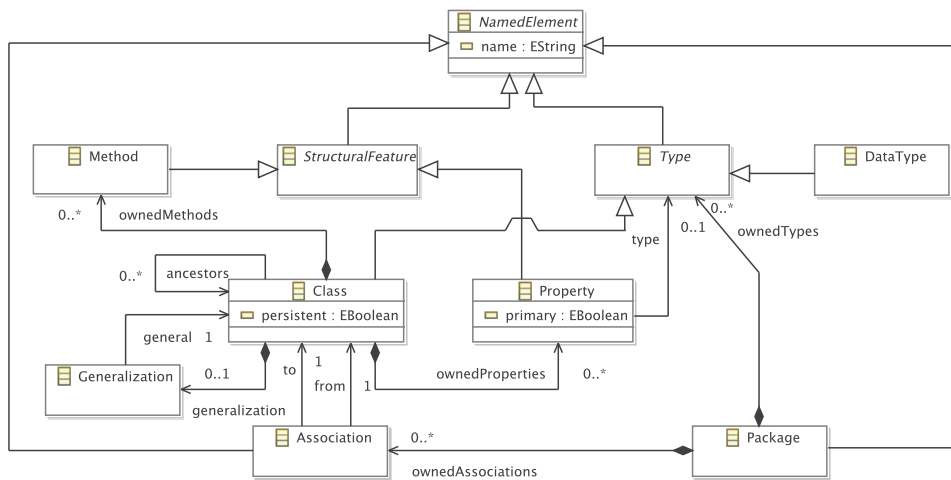


Figure 2.16: New source metamodel CDV2 in Ecore-defined representation.

Figure 2.16 show a new source expressed graphically in UML class diagram notation as the previous metamodels, namely CDV2. As shown in this figure, this metamodel defines a similar structure (or *abstract syntax*) for class models in comparison with the original metamodel, but some modelling concepts are named differently. For example, the modelling concept *attribute* which is reified in the previous metamodel as a type concept (a terminology from the Ecore technological space, i.e. a named instance of Ecore::EClass metaclass) with the name ‘Attribute’ now is defined as another one with the name ‘Property’. Moreover, modelling concepts may be connected differently, as shown in the figure the reference concept *parent* defining the inheritance relationship between classes in the old metamodel is changed by presenting a new type concept, namely ‘Generalization’. The legacy transformation described in a certain transformation language will become inconsistent with the new source metamodel because of these changes, thus should be adjusted or maintained. Obviously, automating this task will significantly promote transformation reuse and avoid a tedious and error-prone manual adaptation.

To describe the impact of metamodel change on the representation format of models, Figure 2.18 shows the concrete representation in the format defined by the new metamodel of the same intentional class model (Figure 2.17). The figure is depicted in the UML object diagram annotation. Changes between two metamodels

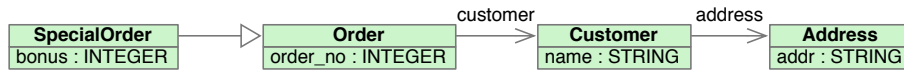


Figure 2.17: Sample intentional input model designed by a CASE tool.

make the representation of the model changing slots' names of instance objects in comparison with the old representation, depicted in blue. Since metamodel-based transformation languages define transformations using names of metamodel elements and the object-oriented structure of metamodel, the transformation engine cannot access to the model represented in another format to achieve and then compute values in slots of instance objects. As a result, the difference in abstract syntax (i.e. different metamodels) is the main obstacle which hinders the reuse of transformations even though transformations are described for the objective of transforming intentional models between modelling languages, and not for any exact representation format of models.

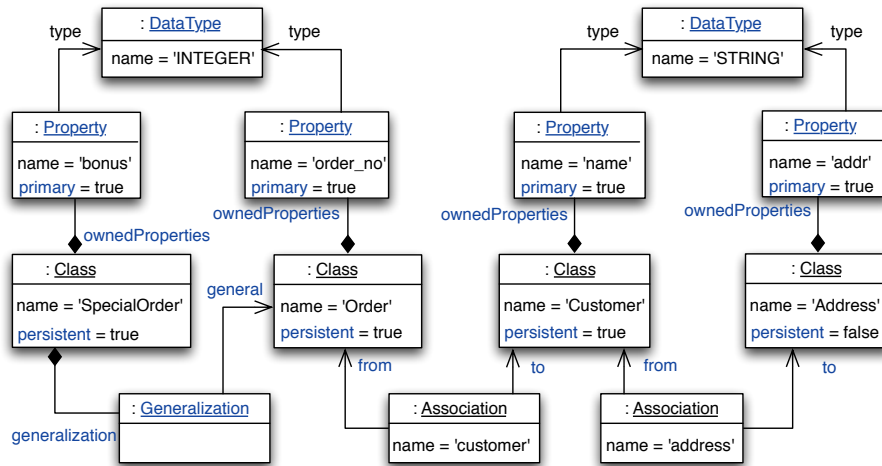


Figure 2.18: Sample input model in CDV2-defined representation.

To summarize, there are a lot of cases where different companies use different metamodels for the same modelled domain. Furthermore, over a period of time, these companies will apply new versions of modelling languages, i.e. evolving their metamodels, and modelling styles. Thus, existing transformations will need to be maintained, and adjusted in order to avoid duplicating and to reduce the cost in transformation development. The following section will give a short discussion on the choice of transformation approach (language) to formally describe the transformation of the case study. This section will result in choosing the MOMENT2-GT language as the transformation approach focused in this thesis. As MOMENT2-GT is further important in our approach on transformation reuse, we also provide more detail information. The description of the language is based on the publication of MOMENT2-GT in [Boronat 09].

2.3.2 Choice of Transformation Language

As aforementioned in Section 2.2.7, a transformation is constituted by a set of transformation rules. However, transformation rules are understood as a broad term describing the smallest unit of transformations. These rules can be functions, procedures, (model) templates, or patterns depending on the paradigm employed in each transformation approach. The way to define transformation rules in each paradigm should be declarative, operational, or hybrid. While the declarative way is suitable to describe simple transformations and provides concise constructs for identifying relations between source and target model elements. The operational approach is preferable for the definition of complex transformations that have more sophisticated computation. From this point of view, such a transformation as in the case study can be described in transformation approaches which are in the graph-transformation-based category [Taentzer 05], see the overview of categories of transformation approaches in Figure 2.13. This category covers all approaches in which graph patterns are used to define transformation rules. These approaches provide a declarative, intuitive way for defining transformations. A quick introduction on graph-transformation-based approaches can be found on page 31.

The transformation CD2RDB is written in MOMENT2-GT¹, a language based on the main concepts of graph transformation systems and supporting directly EMF model transformation [Boronat 09]. A MOMENT2-GT transformation is given as a set of rules — also referred to as model equations/rewrites. Each model equation/rewrite contains Left-Hand Side model patterns, Right-Hand Side model patterns, and Negative Application Condition model patterns for host models (under manipulation). MOMENT2-GT model patterns correspond to object template patterns in the QVT [QVT 08] terminology, or graph patterns in the graph transformation terminology.

Based on the graph transformation characteristic, MOMENT2-GT can specify a model transformation at a high-level as the graph nature of metamodels to reduce the gap between metamodel design and transformation implementation. Thus, since we consider the metamodel similarity as graph similarity, graph-based model transformation frameworks, like MOMENT2-GT, are more relevant than other lower-level transformation paradigms to initially study the substitution relation of metamodels in transformation. Moreover, MOMENT2-GT has been formalized into the rewriting logic framework Maude² which provides built-in tools for formal verification of transformation. For these reasons, we chose MOMENT2-GT on which our implementation is based. However, the proposed approach can be used with any other graph-based model transformation frameworks.

MOMENT2-GT Framework

MOMENT2-GT [Boronat 09] is a model transformation tool based on the *Single Pushout* graph rewriting approach [Ehrig 06]. In MOMENT2-GT, EMF-based

1. <http://www.cs.le.ac.uk/people/aboronat/tools/moment2-gt/>

2. <http://maude.cs.uiuc.edu/>

models are considered as graphs of which nodes are attributed and typed, taking inheritance into account. A transformation definition is constituted by a set of production rules, which are defined in a QVT-based textual format, where simple OCL expressions are used either as guards in (possibly negative) application conditions or as manipulation expressions for attribute values. Such a transformation definition in MOMENT2-GT is compiled into a rewrite theory in the Maude language [Martí-Oliet 07]. In MOMENT2-GT, transformation rules can be defined as either deterministic or non-deterministic production rules and then projected into equations or rewriting rules in Maude, respectively. Since EMF models in this tool can be treated as terms in the Maude algebraic framework, various Maude-based formal analysis techniques [Martí-Oliet 07] can be applied to perform model checking to model-based systems, such as model checking of invariants or LTL model checking (see model checking examples in MOMENT2-GT in [Boronat 09]), in a straightforward way.

As the input EMF model is represented as a term of a specific sort that is defined in a rewriting theory, the execution of a compiled MOMENT2-GT transformation definition can be handled by the Maude's algorithm for term rewriting modulo associativity and commutativity matching. This process finishes when the host term achieves a normal form (i.e. there is no more equations or rules match the term to apply). The resulting term is then parsed by MOMENT2-GT to backward as an output EMF model again.

Figure 2.19 presents a high-level view of the transformation process from a class model to a relational model. Steps to perform a transformation process in MOMENT2-GT are the following:

- (1) and (2): Both class and relational metamodels are specified at layer M2 by means of the Ecore graphical editors provided in the EMF modelling framework.
- (3): The QVT-like MOMENT2-GT transformation is defined at layer M1, but it relates the constructs of the source and the target metamodels. As a model, the transformation defined by the user has to conform to the MOMENT2-GT metamodel.
- (4): A class model is defined by means of the Reflective Model editor or a graphical editor based on EMF. The model must be serialized in the XMI representation.
- (5) and (6): Both metamodels are projected as algebraic specifications, i.e. as Maude theories, by means of the interoperability bridges that have been implemented in the MOMENT2-GT framework. This bridge is implemented based on Ecore libraries provided in EMF.
- (7): The transformation is projected into the Maude code as a rewriting theory, which contains the specification of the CD2RDB transformation operator and its rewriting rules compiled from user-defined transformation rules.
- (8): The class model, which is defined in step (4) at layer M1, is projected as

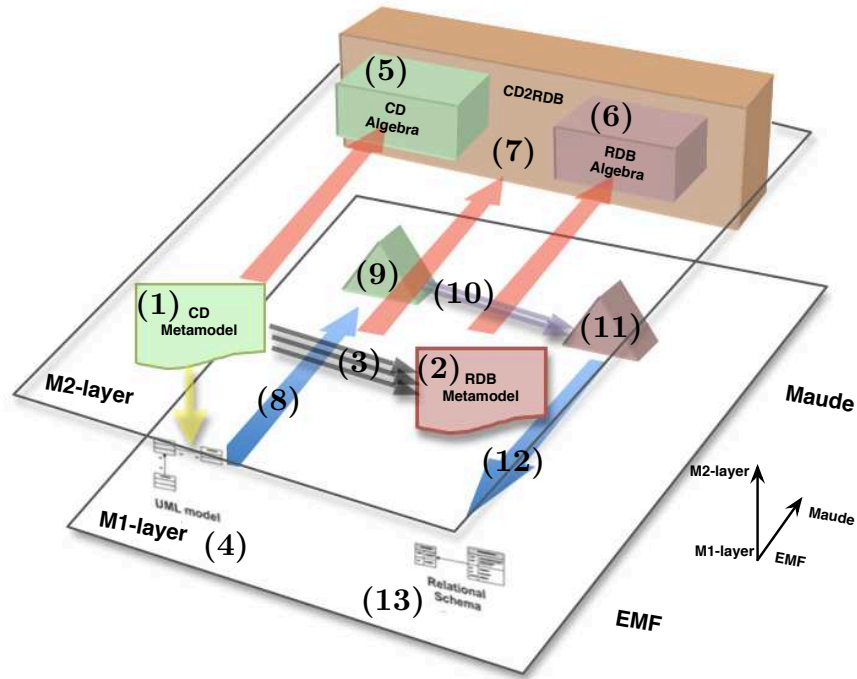


Figure 2.19: Steps performed in MOMENT2-GT (adapted from [Boronat 07]).

a term of the theory corresponding to the class metamodel (9).

- (10): Maude engine applies the CD2RDB operator through its term rewriting mechanism, obtaining a term of the RDB theory (11). Thus, Maude is as an underlying virtual machine for the MOMENT2-GT framework.
- (12): The last step parses the term (11), achieving an EMF-based model (13) in layer M1, which conforms to the target metamodel defined at layer M2.

In the model transformation process, the user only defines the source and target metamodels (steps (1) and (2)), the MOMENT2-GT transformation between the metamodels (step (3)) and the input model (step (4)). The other steps are automatically carried out by the MOMENT2-GT framework.

Summing up, MOMENT2-GT provides a model transformation framework for EMF-based models. Transformations in MOMENT2-GT are written in a textual QVT-based language to describe rules by following the *Single Pushout* graph rewriting approach. The execution semantics of transformation rules are given through a compilation of rules into rewriting rules within the Maude rewriting framework. Rules in MOMENT2-GT language are purely declarative, i.e., the framework does not use any control structure on the application of rules. In order to process each match only once, MOMENT2-GT uses negative application conditions (NAC) and OCL constraints, instead of maintaining instances of the applied rules to remember the matched nodes. In the end, formal analysis of transformation definitions is available in MOMENT2-GT as provided by the underlying Maude engine.

MOMENT2-GT Language

As the QVT-based language in the MOMENT2-GT tool is further important in this thesis, we provide more detail information about the abstract syntax (i.e. the language’s metamodel), the textual concrete syntax, and an informal description of the execution semantics for the language’s constructs. The rewriting logic semantics of the language in Maude are not presented. One can read [Boronat 09] for more details on the compilation of a transformation definition into rewriting logic. In the following paragraphs, the most important syntactic constructs of the language will be introduced by using the UML class diagram annotation. Since MOMENT2-GT has been implemented by using XTEXT technology [Xtext 08], a partial concrete syntax will be presented in XTEXT Grammar, similar to EBNF Grammar, with the motivating transformation implementation of the case study, i.e. the CD2RDB transformation.

Transformation and ModelTypes In the MOMENT2-GT language a transformation (Transformation) between models is specified as a set of rule (Rule) that specifies transition steps in a transformation, cf. Figure 2.20. A transformation can be applied to models that conform to a model type (TypedModel), which is a specification what kind of model elements any conforming model can have. The types of elements these models can have are restricted to those within the referenced package of metamodels (the *modelType* reference). The models for which a transformation is specified are parameters (the *parameters* containment reference) of the transformation.

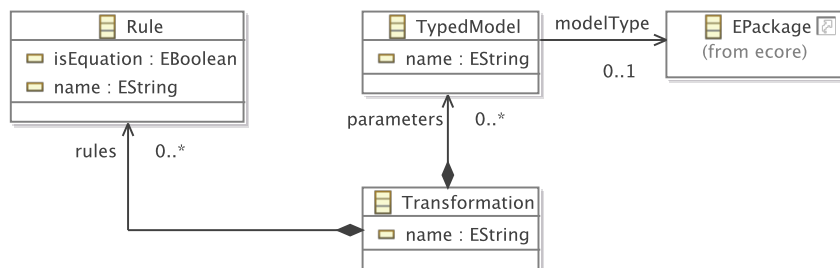


Figure 2.20: MOMENT2-GT metamodel: transformation and model types.

Listing 2.1: MOMENT2-GT XTEXT Grammar: transformation and model types

```

import "http://www.eclipse.org/emf/2002/Ecore" as.ecore
2 Transformation :
  (imports+=Import)+
4  "transformation" name=ID "(" parameters+=TypedModel ";" parameters+=
  TypedModel)* ")"
6  "{"
  (rules+=Rule)+
8  "}"
;
10 Import:
  "import" importURI=STRING ";"
12 ;
TypedModel:
14 name=ID ":" modelType=[ecore::EPackage]
;

```

Listing 2.1 shows the corresponding concrete syntax for constructs in Figure 2.20. A transformation is defined by providing a *name*, a list of *parameters* and a set of *rules*. Each parameter is specified as “name=ID ":" modelType=[ecore::EPackage]”, where *name* is used in the transformation to refer to the model argument and an EPackage is referenced by the meta-property *name* of the root package of the meta-model. The cross reference mechanism is supported through the *import* mechanism provided by XTEXT.

Listing 2.2 specifies a transformation named CD2RDB between the model arguments *cd* and *rdb*. The model argument *cd* declares the CDV1 package as its metamodel, and the *rdb* model argument declares the RDB package as its meta-model.

Listing 2.2: Transformation and model type declarations

```

1  import "platform:/resource/CD2RDB/metamodels/CDV1.ecore";
2  import "platform:/resource/CD2RDB/metamodels/RDB.ecore";
3  transformation CD2RDB ( cd : CDV1 ; rdb : RDB ) {
4    /* transformation Rule declarations */
5
6  }

```

Rules and DomainPatterns Rules (Rule) in a transformation can be either model equations or rewrites, distinguished by a boolean value of attribute *isEquation*, cf. Figure 2.21. A rule is provided by a *name*, a list of LHS domain patterns, a list of RHS domain patterns and a list of NAC domain patterns (DomainPat-

tern and NacDomainPattern). Each domain pattern has a list of object templates (ObjectTemplateExp) and it refers to a model argument (the *domain* reference of DomainPatternExp) of which the model type constraints graph patterns which can be matched in a model of a given model type (TypedModel).

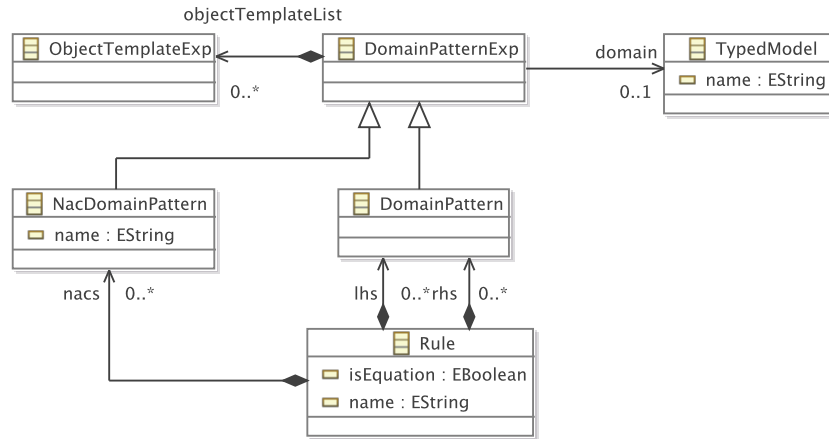


Figure 2.21: MOMENT2-GT metamodel: rules and domain patterns.

Listing 2.3: MOMENT2GT XTEXT Grammar: rules and domain patterns

```

16 Rule:
   (isEquation?="eq"|"rl") name=ID "{"
18   (
   ("nac" nacs+=NacDomainPattern ";"*)
20   "lhs" "{" (lhs+=DomainPattern)* "}" ";"
   "rhs" "{" (rhs+=DomainPattern)* "}" ";"
22   )
   "}"
24 ;
   DomainPatternExp:
26   DomainPattern | NacDomainPattern
   ;
28 DomainPattern:
   domain=[TypedModel] "{"
30   (objectTemplateList+=ObjectTemplateExp)*
   "}"
32 ;
   NacDomainPattern:
34   domain=[TypedModel] name=ID "{"
   (objectTemplateList+=ObjectTemplateExp)*
36   "}"
   ;
  
```

Listing 2.3 shows the corresponding concrete syntax for constructs in Figure 2.21. A rule is defined as either an equation (by keyword `eq`) or a rewriting rule (by keyword `rl`). The LHS, RHS, and NAC parts are distinguished by keywords `lhs`, `rhs`, `nac`, respectively. The *domain* of a domain pattern is given by a name which refers to one of model arguments.

Listing 2.4: Rule and domain pattern declarations

```

1  rl Package2Schema {
2    nac rdb noSchema {
3      /* Object Template specs. */
4      s1 : Schema {
5        }
6    };
7  lhs {
8    cd {
9      /* Object Template specs. */
10     p : Package {
11       name = pname
12     }
13   }
14   rdb {
15     }
16 };
17 rhs {
18   cd {
19     /* Object Template specs. */
20     p : Package {
21       name = pname
22     }
23   }
24   rdb {
25     /* Object Template specs. */
26     s : Schema {
27       name = pname
28     }
29   }
30 };
31 }
32 /* begin another rule */

```

In Listing 2.4 the `Package2Schema` rule is constituted by a negative application condition with a name `noSchema` for the `rdb` model argument, an LHS and a RHS which define domain patterns over both model arguments `cd` and `rdb`. The model types of these arguments, i.e. declared packages of metamodels, give constraints on what types, attributes, reference links can be used to define graph patterns in a domain pattern.

Graph Patterns LHS and RHS domain patterns can specify an arbitrarily complex graph pattern by means of template expressions, i.e. Object Template Expressions (OTEs) (`ObjectTemplateExp`) and Property Template Expressions (PTEs) (`PropertyTemplateExp`) as shown in Figure 2.22.

Object templates (`ObjectTemplateExp`) of a domain pattern can be viewed as a graph of object nodes originating from an instance of the model type. In other words, a domain pattern can be viewed as a set of variables that model elements bound to those variables must satisfy to qualify as a valid binding of the pattern and these variables may be used in other domain patterns. An object template can have other object templates nested inside it to an arbitrary depth as long as the pattern is satisfied with the topology constrained by the metamodel of the corresponding model type. OTEs have a type specified by means of a class name which refers to an existing class defined in the imported metamodels (*referredClass*). An OTE is specified by

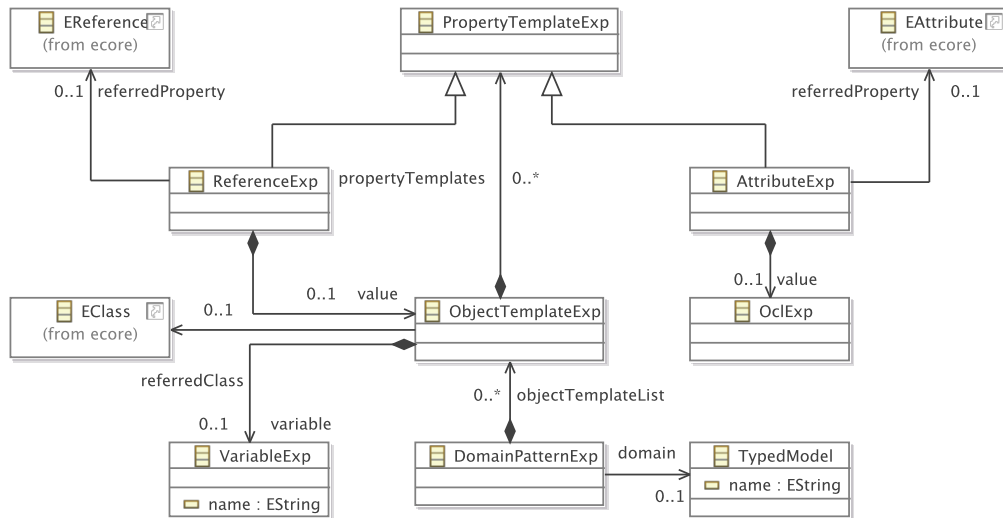


Figure 2.22: MOMENT2-GT metamodel: graph patterns.

a collection of PTEs (*propertyTemplates*), each corresponding to different properties (attributes or (containment) references) of the referred class. PTEs are used to specify constraints on the *values* of the *slots* of the model element matching the container OTE. The constraint is given by either a simple OCL expression (OclExp) for attributes or a nesting OTE for references. Simple OCL expressions can be a variable (VariableExp), an instance value or simple operators on primitive types.

Listing 2.5: MOMENT2-GT XTEXT Grammar: graph patterns

```

38 ObjectTemplateExp:
    variable=VariableExp ":" referredClass=[ecore::EClass] "{"
40     (propertyTemplates+=PropertyTemplateExp ("," propertyTemplates+=
        PropertyTemplateExp)*)?
42     "}"
    ;
44 PropertyTemplateExp:
    ReferenceExp | AttributeExp
46 ;
    AttributeExp:
48     referredProperty=[ecore::EAttribute] "=" (value=OclExp)
    ;
50 ReferenceExp:
    referredProperty=[ecore::EReference] "=" (value=ObjectTemplateExp)
52 ;
    VariableExp :
54     name=ID
    ;

```

Listing 2.5 shows the corresponding concrete syntax for the constructs in Figure 2.22. In this grammar, *referredClass*, *referredProperty* in `ObjectTemplateExp` should be provided by names referencing to existing elements in the model type in which the object template is defined.

The *Class2Table* rule in Listing 2.6 defines several OTEs specifying graph patterns which will be used to match instance models. For example, in the LHS of the rule one OTE is associated with the *cd* model argument, which is typed by metamodel CDV1. Since there is not any reference definition from type concept `Class` to type concept `Package` in the metamodel, we start defining the pattern from a root variable *p* of type concept `Package`. Pattern matching will bind all the variables in the pattern (i.e. *p*, *pname*, *c*, and *cname*). As the variable *pname* is used in another domain pattern (i.e. the OTE associated with model argument *rdb* in the LHS), the matching proceeds by filtering all of the objects of type `Package` in the instance model of *cd*, selecting objects which have the same literal value of attribute *name* as one of any object of type `Schema` in the under-manipulating model of *rdb*.

Listing 2.6: Graph patterns declarations

<pre> 1 rl Class2Table { 2 nac rdb noTable { 3 t1 : Table { 4 name = cname 5 } 6 }; 7 nac cd noParent { 8 c : Class { 9 parent = pClass : Class { 10 } 11 } 12 }; 13 lhs { 14 cd { 15 p : Package { 16 name = pname, 17 ownedTypes = c : Class { 18 isPersistent = true, 19 name = cname 20 } 21 } 22 } 23 rdb { 24 s : Schema { </pre>	<pre> 25 name = pname 26 } 27 } 28 }; 29 rhs { 30 cd { 31 p : Package { 32 name = pname, 33 ownedTypes = c : Class { 34 isPersistent = true, 35 name = cname 36 } 37 } 38 } 39 rdb { 40 s : Schema { 41 name = pname, 42 ownedTables = t : Table { 43 name = cname 44 } 45 } 46 } 47 }; 48 }</pre>
---	---

For property whose values are compared to nested object template expressions. In the case of the property pattern `ownedTypes = c : Class { isPersistent = true, ... }` for example, the matching proceeds by filtering all of objects of type `Class` of which

object ids are in slot *ownedTypes* of the bound object of type Package, eliminating any Class object with its *isPersistent* property not set to *true*.

For properties that are compared to variables (i.e. for attribute concepts), such as `name = cname`, it arises in two cases. If the variable *cname* already has been bound with a value, any filtered Class object in the previous step that does not have the same value for its *name* attribute is eliminated. If the variable *cname* is always free, as in the example, then it will get a binding to the value in the slot *name* for all filtered Class objects. The value of *cname* will be used in another domain pattern, i.e. for filtering out objects in other comparisons in LHS or creating new objects with initial values and modifying attributes of existing identified objects in RHS.

Graph Rewriting Semantics The MOMENT2-GT language allows to specify model transformations in a more declarative way based on a powerful graph rewriting mechanism, i.e. the *Single Pushout* approach [Ehrig 06]. A transformation based on graph-transformation in MOMENT2-GT is defined as a set of graph transformation rules. Each graph transformation rule $r : LHS \rightarrow RHS$ consists of a pair of graph patterns *LHS*, *RHS* which respect to the graph topology of metamodels. From a high-level point of view, the *LHS* graph pattern presents the pre-conditions of the rule, while the *RHS* graph pattern describes the post-conditions. $LHS \cap RHS$ defines a graph part which has to exist to apply the rule, but is not changed. $LHS \setminus (LHS \cap RHS)$ defines the part to be deleted, while $RHS \setminus (LHS \cap RHS)$ defines the part which should be created. Furthermore, a rule can specify modifications on attributes of model objects.

The execution semantics of a rule first conducts a matching step. This step finds a match of the *LHS* graph pattern in the current object graph (i.e. instance model) to make bindings for free variables. Performing a rule is carried out in two steps. First, based on bindings from the matching step, objects and links which are matched by $LHS \setminus (LHS \cap RHS)$ are removed from the model. Second, the intermediate model is glued with $RHS \setminus (LHS \cap RHS)$ to obtain the derived model. To this end, bindings which are in $LHS \cap RHS$ is used to glue the intermediate model with newly created objects and links. To restrict the application of rule with a match, additional simple conditions can be defined over the variables of simple types in the *LHS*, in the *when* part of a rule that we do not present in the grammar of MOMENT2-GT language for simplicity. In addition, a special kind of application conditions are *negative application conditions* which are often used to avoid performing the rule with a match twice. This graph rewriting semantics has been implemented by a compiler which compiles a MOMENT2-GT transformation definition into a rewriting theory in the Maude framework.

In summary, the MOMENT2-GT framework proposes a transformation language, which is based on the graph transformation paradigm. Transformation writers use provided constructs of the language to describe a transformation in thinking about the graph topology of metamodels, which is expressed in an object-oriented concrete syntax, e.g. the UML class diagram notation. The use of graph topology in transformation writing makes the transformation description more declarative and

representative. These features are useful to study the notion of metamodel similarity when we consider the metamodel similarity is as graph similarity, thus derive a substitution possibility between them. Furthermore, these constructs are expressive enough to describe formally the transformation of the case study. For these reasons, we chose MOMENT2-GT for our implementation. The following section backs to the case study in which an original metamodel of a transformation is replaced by a new metamodel (i.e. a different metamodel version of the same modelled domain), and then discusses in detail where are exactly obstacles which hinder the reuse of a legacy transformation.

2.3.3 Discussion

Developing and maintaining model transformations are a tedious and error-prone tasks and require a deep knowledge of modelling languages (i.e. metamodels and their semantics description). Thus, it is significant to support a solution that automates model transformation adjustment task from a legacy transformation when changing the metamodel of the modelling language to avoid developing a new transformation from scratch. However, the main obstacle to maintenance of model transformations is multiple representation formats of concepts in the same modelled domain, serving the particular modelling tool in different companies or evolving over time even in the same company. More precisely, this obstacles can be seen as the following points of view.

- **Different representation:** The explosion in the use of Domain Specific Languages (DSLs) results in a variety of different languages and metamodels. There is a common situation in which different companies develop their in-house metamodels (abstract syntax) for specific domains. Since a concept of some domain reflects a set of real world things which share the same properties, thus companies often model the same domain, but in different metamodels even though they use the same meta-metamodelling language, e.g. MOF. Also, new versions of metamodels, e.g. metamodel version for UML 1.x and UML 2.x, are released over time. Whenever changing to a new version, the existing model transformation defined based the old version might become incompatible, and thus should be adapted or adjusted.
- **Similar semantics:** Since modelling a domain is something very intuitive, different people and companies may observe the same real word and define their own concepts with different associated semantics. However, these concepts can be considered that they have similar semantics because they classify the same real world entities. As a transformation is defined to transform modelling concepts, there is no reason that it cannot be used for transforming similar concepts although transformations are described tightly coupled with metamodels (abstract syntax). From this point of view, it is necessary to identify semantics relation between concepts in order to provide directives for adjusting an existing model transformation.

Two above points of view reflect a phenomenon called *heterogeneity* between metamodels in the metamodeling task. This phenomenon cannot be avoided since metamodeling a domain is carried out by different people via human observation on the real world. More precisely, heterogeneities occur if same (or similar) real world concepts are reified in different ways resulting differently abstract syntaxes (structures) expressed by metamodels. Back to the case study in Section 2.3.1, when metamodel CDV2 replaces metamodel CDV1 in the transformation CD2RDB defined in Section **MOMENT-GT Language** (at page 44 and detail is in Appendix 1.1), intuitively, a human observation on the UML class diagram notation of these metamodels can easily identify similar concepts between two metamodels. Figure 2.23 depicts certain concepts in two metamodels as examples need to be made inter-relation by a human intervention. These relations (cf. (1)–(4)) show positions in metamodel where the heterogeneity phenomenon occurs.

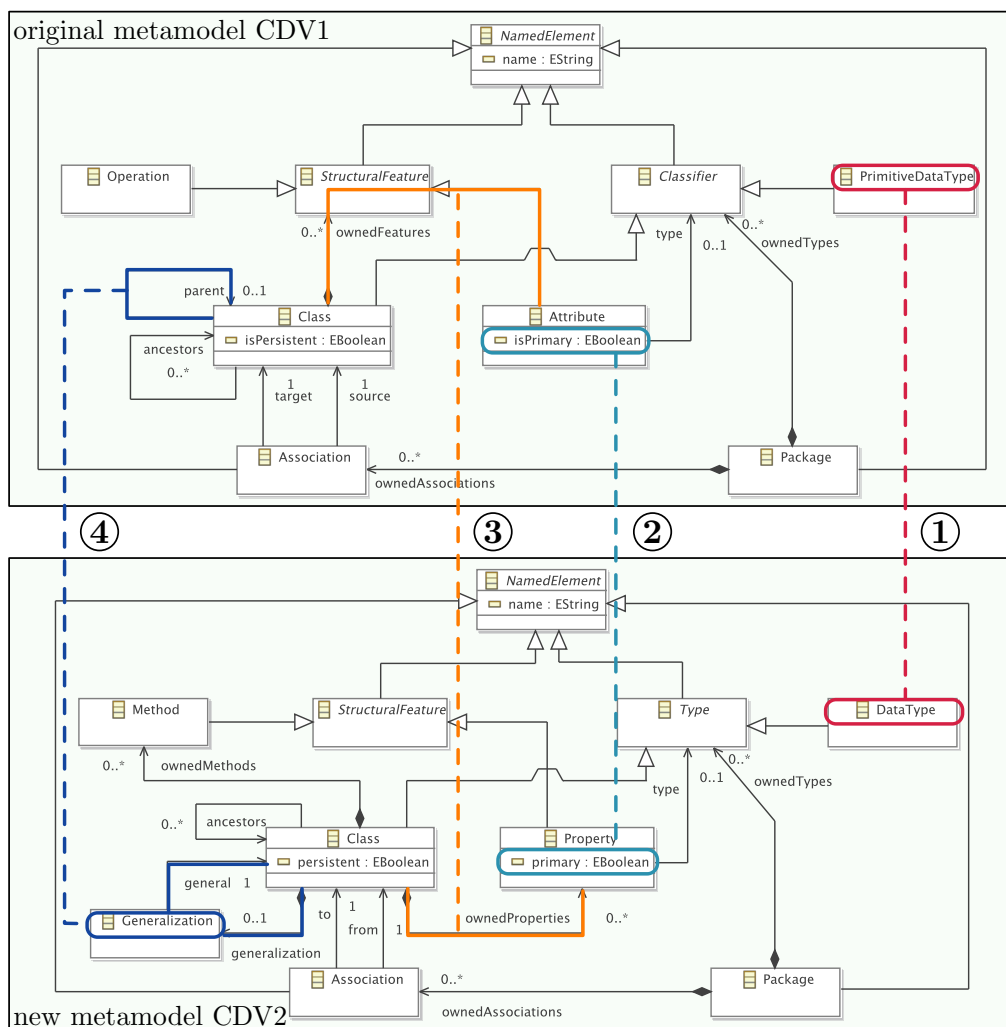


Figure 2.23: Similar concepts between metamodels.

As illustrated in Figure 2.23, there is no way that an existing transformation can find new elements when the source metamodel is replaced by a different metamodel. More precisely, “names” defined in the transformation that refer to type concepts, attribute concepts, (composition) reference concepts (from the MOF (Ecore) terminology) of the original metamodel cannot be bound automatically with those of the new metamodel due to the lack of semantics bridges between similar concepts. Heterogeneities shown in Figure 2.23 are detailed as the following.

- (1) **PrimitiveDataType** \equiv **DataType**. The desired domain concept “*Primitive Type*” which groups basic data types in the real world can be reified differently in using Ecore technological space. More precisely, this “concept” is realized as an instance of Ecore::EClass with the *name* meta-feature set to ‘PrimitiveDataType’ in the metamodel CDV1, whereas the same “concept” in metamodel CDV2 has the name as ‘DataType’. From the semantics point of view, these two type concepts describe the same “concept” in the real world. However, without establishing an explicit semantics bridge between them, there is no way that a transformation which uses the first type concept can be switched to the second. Figure 2.24 shows the difference between two type concepts in Ecore-defined data representation.

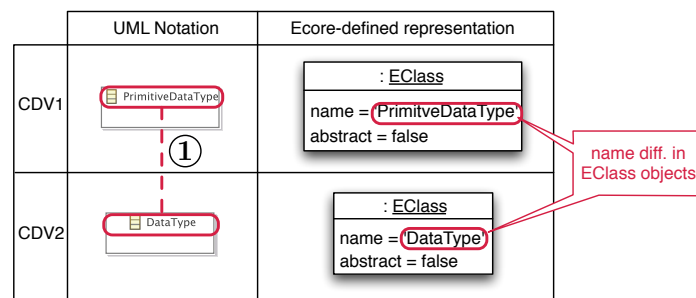


Figure 2.24: Similar type concepts (1) with different reifications in Ecore.

- (2) **isPrimary** \equiv **primary**. In the context of two similar type concepts Attribute of metamodel CDV1 and Property of metamodel CDV2, the desired “concept” which indicates that an attribute/property is primary or is not reified by Ecore::EAttribute with different values for the name meta-feature. Although

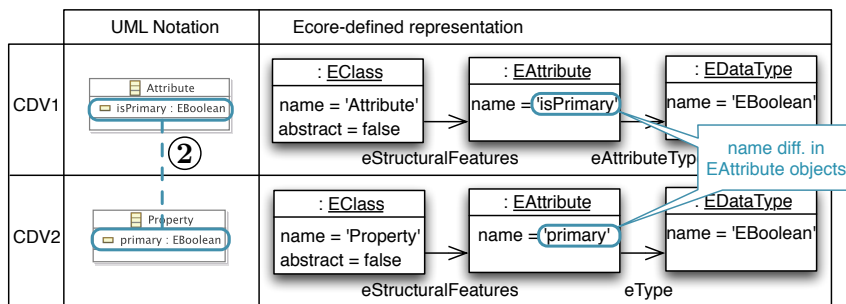


Figure 2.25: Similar attribute concepts (2) with different reifications in Ecore.

these attribute concepts have the same primitive type, i.e. EBoolean, a transformation which refers to attribute concept *isPrimary* of the first metamodel cannot understand that there is a similar concepts in the second metamodel without a directive from human. Figure 2.25 shows the name difference between two attribute concepts, with an assumption that they are in the same (or similar) type concepts.

- (3) **ownedFeatures** \equiv **ownedProperties**. The compositional relation between two real world concepts ‘Class’ and ‘Attribute’ is reified indirectly by an instance of the Ecore::EReference meta-class of which the type property is set to the abstract concept ‘StructuralFeature’. This abstract concept is special-

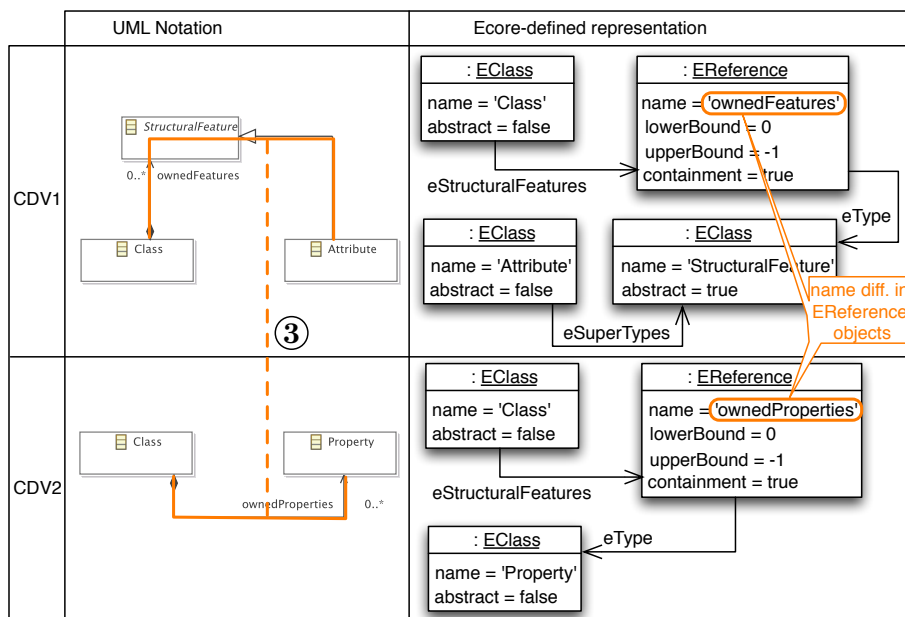


Figure 2.26: Similar reference concepts (3) with different reifications in Ecore.

ized by type concept ‘Attribute’ via the inheritance mechanism provided in the Ecore technological space, cf. Figure 2.26 for the Ecore-defined representation of metamodel CDV1. In metamodel CDV2, the same compositional relation is reified differently, i.e. directly rather than a hierarchy. In this context, reference concepts between metamodels are not only reified with different names, but also different target types. With an assumption that the semantics correspondence between ‘Attribute’ of CDV1 and ‘Property’ of CDV2 has already identified, there is no reason that a transformation uses reference type ‘ownedFeatures’ of CDV1 cannot be applied for ‘ownedProperties’ in CDV2. The heterogeneities which occur in the third situation are shown in Figure 2.26.

- (4) **parent** \equiv **generalization.Generalization.general**. Finally, there is a heterogeneity when defining the generalization relation between classes. Whereas in metamodel CDV1 this relation is reified by an instance of the Ecore::EReference meta-class which is named as ‘parent’, it is reified by three instances of different

Ecore meta-classes, i.e. reference concept ‘generalization’; type concept ‘Generalization’ and another reference concept ‘general’, in metamodel CDV2 cf. Figure 2.27. Although these instances represent different structures (abstract

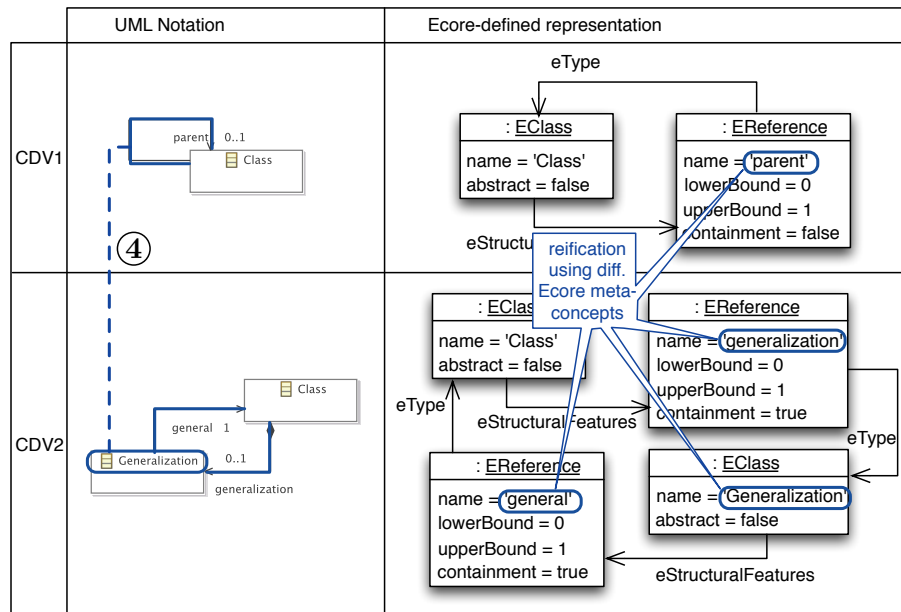


Figure 2.27: Similar concept groups (4) with different reifications in Ecore.

syntax) for models. However, a combination of them in metamodel CDV2 describes the same concept, i.e. the generalization relation between classes, as done by reference type ‘parent’ of metamodel CDV1. Since transformations are defined for transforming instances of domain concepts, it is necessary to solve these heterogeneities. This enables to reuse knowledge encoded in the existing transformation across metamodels which define different structures.

As illustrated in Figure 2.23, when the source metamodel of the motivating transformation is changed, some “name” in the transformation which refers to the type concepts, attribute concepts, or reference concepts of the original metamodel cannot be found in the new metamodel due to the difference in name, cf. (1)(2)(3) in Figure 2.23. Note that most of binding mechanisms provided in transformation frameworks for transformation rules, e.g. MOMENT2 - see Listing 2.5 at page 48, uses values of the name meta-feature of concepts. We call heterogeneities that arise by only the difference in the name meta-feature *isomorphic* heterogeneities, because they do not change the global graph topology of metamodels from the UML Class Diagram notation view. In contrast, when the concept of “generalization between classes” is concretized by reference concept *parent* in the original metamodel and by a combination of concepts (i.e. reference concept *generalization*, type concept *Generalization* and reference concept *general*) in the new metamodel, we call that heterogeneity *non-isomorphic* heterogeneity. Non-isomorphic heterogeneities affect not only on the binding aspect, but also on the structural incompatibility problem of transformation code, e.g. in MOMENT2 they make the legacy model patterns

become incompatible to the new structure of the new metamodel. As a result, two kinds of heterogeneities limit the use of the transformation for the new popular of class models in the second format, i.e. CDV2, even when the metamodeler's intent on modelling concepts and relationships between them are actually the same. Thus, to reuse knowledge encoded in existing transformations, there is a need to investigate a solution to describe manually the correspondences of metamodeler's intent between metamodel elements with different representation. These correspondences then can be used as directives to adapt the legacy transformation in response to heterogeneities in metamodels while preserving the original intention of transformation.

Summing up, we have introduced basic concepts in the domain of Model Driven Engineering in Section 2.2 and stated the problem of transformation reuse through a case study in Section 2.3 that are further important in this thesis. The concepts and definitions presented for modelling, metamodelling, and the use of Domain Specific Modelling Languages (cf. Sections 2.2.2 2.2.3, 2.2.6, respectively) build a foundation on the basis of which we develop our solution for model transformation reuse problem. It is also important to understand transformation approaches in the literature, cf. Section 2.2.7, since each approach has a particular solution at a certain abstraction level in describing transformations. Transformation reuse means the reuse of knowledge encoded in transformations defined in a particular approach, thus the knowledge themselves have a level of abstraction. In the next section we give an introduction on recent works in the literature which proposed different solutions for the problem of transformation reuse, which is the main topic of this thesis.

2.4 Related Work

With the evolution of metamodels it is significant to deal with the evolution of various software artifacts in MDE, such as models, model transformations, constraints, editors, etc. Since the main topic of this thesis focuses on the transformation reuse aspect, i.e. the reuse of knowledge encoded in existing model transformations, we will introduce a category on approaches that deal with to this problem, i.e. reuse through adaptation. Transformation reuse through adaptation in response to metamodel changes had been studied with two main different strategies. The first considers transformations as white-boxes (i.e. transformations as models that can be manipulated) and transforms them. The second strategy regards transformations as black-boxes (transformations as tools) and generates adaptors for the transformation without modifying the original one. Most approaches in both strategies are based on establishing mappings that specify the delta (i.e. changes) between the old and the new metamodel. From mappings, depending on the chosen strategy, existing transformations might be adapted (or adjusted) to obtain new transformations that can be executed directly on the model population of the new metamodel or should be composed with generated adaptor transformations (i.e. intermediate transformations) that convert models of the old metamodel to the format defined by the new metamodel.

Categorization

We will classify several approaches have been proposed and already developed in the recent years to provide direct supports for transformation reuse in the context of metamodel evolution in MDE. Most of works in [de Lara 10, Cuadrado 11, Wimmer 11, Levendovszky 10, Ruscio 11] and [Babau 11, Kerboeuf 11] fall into two above strategies. We also introduce a complementary strategy for the work in [Sen 10, Steel 07] which performs adaptation directly on metamodels since metamodels in that approach have operational semantics like transformations. Note that most of approaches presented here are proposed to solve a new requirement over the recent years, i.e. transformation reuse problem in the context of MDE, and are developed concurrently with this thesis. Thus, excepting the approach in [Steel 07], others approaches are considered as complementary sources to compare with our work. In addition to above strategies, other categories (some of them is derived from the work of [Rahm 01] on the schema matching problem in database application domains), are explained as follows.

Mappings. We consider only mappings that are established to express correspondences at metamodel level. From the metamodel mapping support point of view, the following sub-categories are used to distinguish approaches.

- **Cardinality:** Mappings can be defined from one or more elements of one metamodel to one or more elements of another, thus four cases are considered including: $1:1$, $1:m$, $n:1$, and $m:n$.
- **Total vs Partial Mapping:** This sub-category indicates whether mappings must be defined over the whole metamodels or only metamodel parts which are really used in a specific context, i.e. in the transformation code. This sub-category relates to the *validation* checking aspect when describing the mappings.
- **Element vs Structure Mapping:** Mappings can be defined for individual metamodel elements, such as attribute concepts, type concepts, reference concepts, or for combinations of metamodel elements, such as complex metamodel structures.
- **Constraint :** Various kinds of constraints can be defined on metamodels and their elements, such as value range on a certain data type, optionality, uniqueness, types of relationships and cardinalities. In mapping approaches that take constraints into account, constraints are usually understood as a part of structures, where the structure topology and different element kinds are used to define mappings.

Transformation Language. This category considers which model transformation languages (or paradigms) can be supported by the approaches. Two level abstractions are used. The *Paradigm* dimension indicates which model transformation paradigms

that the approach can support in principle. The *Language* dimension enumerates the concrete transformation languages for which the approach has already been implemented.

Application Scenarios. This category concerns scenarios directly supported by the approaches. *M2M transformations* often means exogenous transformations in which an output model is produced from an input model. In contrast, *in-place updates* often are refactoring transformations which perform an optimization on models without changing external behaviour their semantics.

Declaration automation. This category varies from the use of *explicit* mechanisms without any automation supports in describing mappings to the use of *implicit* mechanisms which are equipped with some reasoning mechanisms to produce automatically mappings. To this end, these approaches usually not only rely on the metamodels, but also on auxiliary information, such as dictionaries; global ontologies, which are integrated into the metamodel definitions.

2.4.1 Transformation adaptation

As aforementioned, in this strategy, from an existing model transformation, which is described on the basis of old metamodels, a new model transformation is produced for the new metamodels. To this end, transformations must be written in a certain transformation language so that we can consider them as models to manipulate. We call this strategy *white-box* transformation reuse mechanism.

Reusing Model Transformations across Heterogeneous Metamodels. The work described in [Wimmer 11] presents an approach to transformation reuse via a flexible binding mechanism being able to automatically resolve recurring structural heterogeneities between metamodels. To this end, the binding models are analyzed and required adaptors are automatically added to the transformation in order to obtain a new adapted one.

This work is based on several previous works published in [de Lara 10, Cuadrado 11]. From the *concept* mechanism presented in [de Lara 10], i.e. a specific transformation for a particular metamodel can be generated from a generic transformation defined over a conceptual metamodel and a strict structural binding between the particular and the conceptual metamodel, Cuadrado *et al.* [Cuadrado 11] has extended the binding DSL for structural heterogeneities between metamodels, and then Wimmer *et al.* [Wimmer 11] has continued improving the level of automation of the DSL. First, the binding DSL proposed in [Cuadrado 11] has been improved by removing complex OCL-related constructs. This allows users specifying binding definitions in a simpler manner, thus the automation level of adaptation was also improved. Second, OCL-based adaptors which are scattered across the specific model transformation as in the version of [Cuadrado 11] are replaced by composable adaptors which are

separately added into the generic model transformation. This improvement makes adapted transformations become easier to maintain.

The approach for reusing model transformations across heterogeneous metamodels allows specifying *1:1* and *1:n* bindings for metamodel elements. The binding in this approach is defined totally to assure the new metamodel can be replaced in every context (transformations) in which the old metamodel is declared. The approach allows defining bindings for single elements of the metamodels. Furthermore, it is possible to describe some kinds of complex structure correspondence, e.g. an attribute concept of a type concept in the old metamodel has a semantic equivalence (or equality) in comparison with an attribute concept, which is resided in another type concept of the new metamodel. The approach provides an adaptation mechanism, which is able to overcome constraint on multiplicity. For example, a transformation rule defined upon a reference, which is defined as single in the old metamodel can be adapted by adding an adaptor which works with the reference defined as multiple in the new metamodel. Constraints on value range of attributes can be defined through a filtering mechanism. The approach supports the hybrid transformation paradigm, i.e. having declarative parts and also imperative constructs for navigating in complex computation, and has been implemented for the ATL language. The approach solves the problem which occurs in model-to-model transformations (also called exogenous transformations). It currently supports only for the case in which the source metamodel changes. In-place update transformations (also called endogenous transformations) are not supported because this kind of transformation relates not only to accessing desired information stored in objects, but also removing and creating objects that have to respect complex constraints which are defined in metamodels, such as composition, mandatory, etc. Bindings must be defined explicitly by the users. There is neither any helping (or reasoning) mechanism for editing bindings nor a notion of valid binding for checking user-defined inputs.

Semi-automated Evolution of DSML Model Transformation. The proposed approach in [Levendovszky 10] introduces an evolution method for model transformations. This method is based on an assumption that a change description is available in a modelling language specific to the evolution. Based on such a change description, their method is able to automate certain parts of the evolution. Moreover, the method can automatically alert the user about the missing semantic information that is not described in the change description for a later manual correction.

Concretely, a change description can be described in a migration DSL, i.e. Model Change Language or MCL. The MCL language which has first been presented in [Narayanan 09] is initially developed to manage the metamodel/model co-evolution problem. The authors reuse this language to define minor changes that occur in an evolved metamodel, in addition, provide an evolver tool for transformations rather than models of the domain. The evolver tool interprets a change description written in MCL language and performs migration steps on transformations defined upon the old metamodel to obtain semi-migrated transformations. For this reason, we categorize this approach into transformation migration strategy.

The proposed approach in [Levendovszky 10] is also dedicated for graph rewriting-based transformation paradigm, implemented in GReAT³. In using the MCL language, one can describe $1:1$ and $1:n$ mappings between two different metamodel versions of a modelling language. A change description is defined totally without regarding any particular transformation. The language allows to define only mappings between single elements of metamodels, without supports for more complex structural mappings. Multiplicity contraction on references is not considered as the work in [Wimmer 11]. The published paper of this work [Levendovszky 10] shown only an example of an exogenous transformation in which a source metamodel can be replaced by an evolved metamodel while the target metamodel has not any evolution. Thus, we consider that this work currently supports only for varying source metamodels. A metamodel of an in-place update transformation, in our point of view, shares several characteristics as a target metamodel, thus the approach cannot support evolutions on this kind of transformation. Finally, concerning to the automation in declaring a change description, users must define explicitly mapping by means the MCL language without any suggestion mechanism.

The main difference of this approach in comparison with the one in [Wimmer 11] is the additional support of partially automated transformation migration. For example, in cases of attribute deletion that cause a lack of information, the evolver tool will mark the deleted attribute in related code fragments of the transformation. Then, it is necessary to have some corrections from a transformation developer. Accordingly, from a metamodel change description, an automated phase is performed completely automatically, then a second manual phase is required, where someone performs manual tasks to correct the semi-migrated transformation annotated with suggestion marks.

Managing Co-evolution in MDE. Authors in [Ruscio 11] propose EMFMigrate⁴ as a unified approach to the metamodel co-evolution problem with dependent artifacts such as models, transformations and tools. For ATL transformations, they introduced a coupled DSLs which encompasses the specification of metamodel differencing (EDelta) and (customizable) migration rules (ATLMigrate). EDelta models are actually the recovered trace input by users to specify step-by-step metamodel evolution. Migration rules might be applied if conditional patterns are matched on the difference model. Then, the information extracted from the holds will be used in rewriting rules to migrate transformations.

Similar to the approach in [Levendovszky 10], metamodel-changes which affect the existing transformations are classified into three groups: 1) *fully automated* are changes that we can migrate automatically existing transformations without user intervention; 2) *partially automated* are changes that existing transformations can be adapted automatically although some manual fine-correction is required; 3) *fully semantics* are changes affecting transformations which cannot be automatically migrated. All these changed can be described at a generic level by means of the meta-

3. <http://www.isis.vanderbilt.edu/tools/GReAT>

4. <http://www.emfmigrate.org/>

model differencing language, i.e. EDelta. For a specific metamodel evolution, generic libraries written in the EDelta language are invoked through migration rules written in the ALTMigrate language.

The approach is based on the idea of evolution for single elements, thus at some level of abstraction, the approach supports to specify $1:1$ and $n:1$ (merge references) mappings between two versions of a metamodel. Because of without regrading the usage context of metamodels, users have to specify every change occurred in the evolution even though some metamodel elements may not be used in a considered transformation. In using this coupled DSLs, a transformation migrator can only make bridges for single elements between the original and the evolved metamodels. In this approach, we do not see any support for the multiplicities contraction neither for adding constraints into an adapted transformation. The work has been implemented for transformations written in the ATL language, a language in the hybrid transformation paradigm. The paper [Ruscio 11] shows only an exogenous transformation example in which the source metamodel evolves. Other cases such as varying versions of a target metamodel or an in-place transformation are not considered in this work. In similarly to two above approaches, migration rules (can be understood as bindings at some level of abstraction) must be declared explicitly by a domain experts. There is no mention of validation checking for migration rules.

2.4.2 Model adaptation

In the second strategy, a new model transformation is produced to transform models conforming to the old metamodels into models conforming to the new metamodel. The produced model transformation is then composed (i.e. chaining model transformations) with the existing model transformation without modifying it. Transformations produced according to this strategy plays a role as adapters for executing legacy model transformations to process models conforming to the new metamodel. Since the strategy do not take the structure of existing model transformation into consideration, i.e. independent of transformation languages, we call this *black-box* model transformation reuse mechanism.

A DSML for Reversible Transformations. One of approaches in the second strategy that considers a transformation as a black-box is proposed by Kerboeuf *et al.* [Kerboeuf 11, Babau 11], which presents an adaptation DSL, named Modif, with which handles copy or deletion of elements from a model conforming to a new metamodel to make it become an instance model of the input metamodel of an endogenous transformation tool. The adaptation transformation also provides a trace to recover the output result of the transformation to an instance of the new metamodel.

Related to criteria in our classification, the adaptation language Modif allows users to specify $1:1$ operations in order to build a new metamodel from an existing metamodel. These operations are separated into three families: 1) *update operators* are dedicated to specify metamodel concept renaming or other changes in meta-attributes; 2) *deleting operators* are used to indicate that a metamodel concept is no

longer present in the new version of metamodel; 3) *refactoring operators* that enable to state simple refactorings such as hidden a given type concept or flattening an inheritance relationship. Operations proposed in Modif are fine-grained operations, hence it should be necessary to apply these operations in many times to obtain an expected new version of a metamodel. In addition to the operational semantics at the metamodel level, the Modif tool also supports generating an adaptation transformation in order to translate the models complying with the first version of metamodel to models complying the target version. Since transformations are considered as black-box tools, thus a Modif specification must be defined totally even though most of rewriting tools do not use all concepts defined in the input metamodel. From the mapping point of view, Modif supports both bindings for single elements which are present in two versions of the same metamodel by means of *update operators*, and bindings for more complex structural differences by making the use of *deleting operators* and *refactoring operators*. The update operator when changing multiplicity from multi to single keeps only the first element of the set, thus it results a loss of information. Due to the respect to the second strategy, the approach can work with any transformation paradigm. The transformation language Kermeta is used as a target language for generated adaptation transformations from Modif specifications. The extension of Modif in [Kerboeuf 11] makes the approach usable for in-place update transformations. However, as well as other approaches, Modif specifications have to be defined explicitly and they are not checked based on a validation principle.

Model Adaptation by Precise Detection of Metamodel Changes. The approach presented in [Garcés 09] by Garcés *et al.* captures different kinds of both simple and complex matchings between metamodel concepts using matching models that conform to the AtlanMod Matching Language (AML). An adaptation transformation can be derived from a matching model by means of a HOT. The approach aims to support the model evolution scenario and the exchange of models. Unlike the approach in [Kerboeuf 11, Babau 11], model transformation reuse is not supported directly in this work. However, this could be realized by chaining a generated adaptation transformation with an exogenous transformation. This approach provides an automatic heuristic matching process that enables user intervention to produce matching models.

The matching models allows to use *equivalence* and *difference* mapping concepts. Basic difference mapping concepts are Added and Deleted which mark a metamodel element as deleted/added from/into an original metamodel. Equivalence mapping concepts are those that enable to describe simple (elementary) mappings or more complex mappings. Complex mappings are derived from Added and Deleted simple mappings via a heuristic step in the whole matching process, e.g. a mapping can be inferred for the case of reference reification by the adding of a new class. However, final result mappings are merely *1:1* correspondences and exhaustive. According to explanations in [Garcés 09], multiplicities defined on two references must be the same (lower and upper bounds) to be able to assign a similarity value between them. In regrading the context of transformation reuse, a generated adaptation transformation derived from a matching model can be composed with exogenous transformation

written in any languages. This work has chosen the ATL language as the target language of generated adaptation transformations. It is possible to achieve matching models by using heuristic inference libraries provided within the language, thus user does not need to specify them explicitly.

As a conclusion, the common point among above approaches in both strategies is trying to capture the changes between metamodels. These changes are described by users in DSL languages and then used as directives for migration process, on either models or transformations.

2.4.3 Metamodel adaptation

Reusable Model Transformations. Beside to above strategies, another approach is also proposed in [Sen 10] by Sen *et al.* to make a transformation reusable by extending an original metamodel based on aspect weaving and derived property adding provided in Kermeta⁵. This approach uses a metamodel pruning mechanism [Sen 09] to automatically remove unaffected metamodel elements in a legacy transformation definition to obtain an effective metamodel. This mechanism helps reduce adaptation requirements via aspect weaving for satisfying the model typing principle in [Steel 07]. After the metamodel adaptation process, a legacy transformation definition become usable for the new population of models without any modification on transformations or models.

Based on the metamodel pruning mechanism used in their work, added adaptations can be declared in a partial manner. These adaptations are actually *1:1* bindings from the mapping point of view. In using complex navigation facilities supported directly in Kermeta, users can add or override *accessor* and *mutator* methods to make bridges not only for single elements, but also between a single reference and a reference path. The constraint related to the multiplicity has not been relaxed, i.e. a reference defined as multiple cannot be matched against a single reference. The approach is tightly coupled with the imperative language Kermeta which provides the AOP (Aspect-Oriented Programming) *static introduction* mechanism. Due to the use of complex imperative constructs for the adding of derived properties in classes, the approach can be applied to various kinds of transformations, from exogenous to endogenous. Nevertheless, adaptation parts which are written in Kermeta AOP-supports have to be added manually by users at the lower-level of an imperative language. In order to support manual adaptation, Kermeta employs a validation mechanism based on the model-type matching principle that has been originally presented in [Steel 07].

5. www.kermeta.org

	Reusing Model Transformations across Heterogeneous Metamodels	Semi-automated Evolution of DSML Model Transformation	Managing Co-evolution in MDE	A DSML for Reversible Transformations	Model Adaptation by Precise Detection of Metamodel Changes	Reusable Model Transformations
Adaptation Strategy						
<i>transformations</i>	✓	✓	✓	NO	NO	NO
<i>models</i>	NO	NO	NO	✓	✓	NO
<i>metamodels</i>	NO	NO	NO	NO	NO	✓
Mapping						
<i>Cardinality</i>	1:1 1:n	1:1 1:n	1:1 n:1	1:1	1:1	1:1
<i>Total vs. Partial</i>	Total	Total	Total	Total	Total	Partial
<i>Element vs. Structure</i>	Element and several structural supports	Element supports	Element supports	Element and several structural supports	Element and structural supports	Element and several structural supports
<i>Constraint</i>	Multiplicities contraction, properties filtering	Without multiplicities contraction support	Without multiplicities contraction support	Multiplicities contraction (lost information)	Without multiplicities contraction support	Without multiplicities contraction support
Transformation Language						
<i>Paradigm</i>	Hybrid (decl.& imp.)	Graph rewriting-based	Hybrid (decl.& imp.)	Any	Any	Imperative with AOP supports
<i>Language</i>	ATL	GReAT	ATL	–	–	Kermeta
Application Scenarios						
<i>M2M Transf.</i>	✓ (Source MM change)	✓ (Source MM change)	✓ (Source MM change)	✓ (Source MM change)	✓ (Source MM change)	✓ (Source, Target MM change)
<i>In-place Up.</i>	NO	NO	NO	✓	NO	✓
Declaration Automation						
<i>explicit</i>	✓ (without checking)	✓ (without checking due to semi-automation)	✓ (without checking due to semi-automation)	✓ (without checking)	✓ (user intervention but without checking)	✓ (a priori checking)
<i>implicit</i>	NO	NO	NO	NO	✓ (heuristic infer.)	NO

Table 2.2: Related approaches to transformation reuse.

2.4.4 Categorization Overview

Table 2.2 gives an overview of the approaches supporting transformation reuse when metamodels change in the context of MDE. The categories that we presented above are used in the table to classify the discussed approaches in previous sections.

2.5 State-of-the-Art Summary

In this chapter, we have presented basic concepts and principles in the domain of Model-Driven Engineering that are used in this thesis. They are the key notions such as: *model*, *metamodel*, *meta-metamodel* and the four-layered architecture *Meta-Object Facility* of models. We have also present the notion of *Domain Specific Language* (DSL) which is usually used in various MDE approaches based on the principle of separation of concerns. The overview and a classification of model transformation approaches (languages) are also provided.

Then, we introduce a concrete simple case study on the need of model transformation reuse problem. We chose the MOMENT2 language among various languages as the target transformation language on which our solution will be implemented. The transformation reuse problem is discussed in detail on the motivating case study.

Finally, we have discussed the related approaches addressing to the problem of model transformations reuse in response to changes in metamodels. These approaches are based on different strategies: *model adaptation*, *transformation adaptation* and *metamodel adaptation*. The limitations and also advantages are discussed in detail for each concrete related work to justify existing issues in the model transformations reuse problem which will be solved in the next chapter.

Chapter 3

A Graph-Based Model Typing Approach

Contents

3.1	Chapter Overview	67
3.2	Graph Topology	68
3.3	Typing Models and Transformations with Graphs	74
3.4	Non-isomorphic Transformation Reuse	96
3.5	Summary and Discussion	120

3.1 Chapter Overview

Since metamodels are considered as models, they are created to define the abstract syntax (i.e. the structure aspect) of modelling languages. As transformations are usually described at the abstract syntax level, thus tightly coupled with metamodels, the word ‘metamodel’ is currently often used to denote some kind of “model type”. The metamodel describes the rules and constraints that a model must fulfill to be correct. The model is said to *conform* the metamodel. This point of view is currently applied in all tools that handle models. This definition makes the reuse of transformation difficult because of the strength of this definition of “type”.

In order to tackle this issue, we divide the model transformation reuse problem into two sub-problems. The first sub-problem is the reuse of transformations when alternative metamodels are slightly different and satisfy a “subtype” relationship. To this end, we propose a type abstraction of models. This type abstraction, first, is used to leverage the abstraction level of the notion “model type” and to relax some constraints if possible, instead of using “metamodel” as usual and, second, to define a “subtype” relation by taking the metamodel usage context (i.e. the transformation) into account that derives the substitution between metamodels. The use of this type abstraction is to allow more flexibility in the use of transformation when metamodels

are seen as *isomorphic* in our notion of model type. The second sub-problem is the reuse of transformations for metamodels which are more different that we call *non-isomorphic*. The solution is to use the one proposed in the first sub-problem as basic premises, then extends the transformation reuse possibility through a mapping between metamodels in order to rewrite model transformations. The combination of these solutions allows to improve transformation reuse across metamodel versions or different metamodels used in different organizations.

This chapter gives a description on above solutions in detail in response to the transformation reuse problem when changing metamodels. The first sub-problem and its solution is presented in Sections 3.2 and 3.3 while Section 3.4 details the proposed solution of the second sub-problem. Section 3.5 summarizes and discussion on the whole proposed model typing approach.

More precisely, first, Section 3.2 gives an overview about the graph topology, which is the common characteristic of metamodels in the object-oriented (meta-)modelling methodology. The graph topology is analysed in the context of transformations where it is used as the underlying basis to describe transformations in graph rewriting-based transformation frameworks.

Next, Section 3.3 introduces a notion of “model type”, i.e. Type Graph with Multiplicity, that captures the graph topology of models plus some useful information. As the use of this notion of “model type”, we present a typing function to construct model types over metamodels. In addition, a “model subtype” relation is also formally defined. This relation is used to check the substitution possibility between metamodels that we call isomorphism in the space of our notion of “model type”. We also provide an additional typing function for transformations, which produces what we call *effective* model types since this function analyses the real usage context of a declared “model type” in transformations. As a consequence of the “subtype” relation, the substitutability between metamodels should be checked between the *effective* model types of the declared metamodels and those of the replacing metamodels.

Finally, Section 3.4 explains a reuse mechanism for more complex difference situations between metamodels, i.e. metamodels are different in element names or partial graph topology that we call non-isomorphic. The proposed mechanism includes a mapping DSL language dedicated to describe semantic correspondences between metamodel elements or combinations of metamodel elements. Thanks to these correspondences between metamodels and the typing mechanism in Section 3.3, an adaptation engine can recognize important changes and proceed migrating existing transformations when more complex differences occur.

3.2 Graph Topology

Models, in the context of object-oriented metamodeling paradigm, are a collection of objects linked together using (bi-)directional relationships. From the data structure point of view, the structure (objects and their relationships) of a models family are

typically defined by a MOF (or MOF-like) metamodel. The main advantage of the use of the object-oriented metamodeling paradigm is that object-oriented metamodels are able to capture the structure of modeled systems, thus it is easy to understand them. The natural physical structure of a system family is usually defined as the graph topology constraints by means of object-oriented metamodels in the object-oriented metamodeling paradigm or type graphs in the graph-based metamodeling paradigm.

From the semantics point of view, object-oriented metamodels are designed to reflect concepts of the real world and relationships between concepts. To this end, the MOF (or a MOF-like) meta-metamodeling language provides basic constructs and their object-oriented semantics to define modelling concepts. In the EMF meta-modeling framework, such constructs are realized in the Ecore meta-metamodel as *EClass*, *EAttribute*, *EReference*, etc. with certain properties on them to provide additional constraints such as: the *abstract* meta-attribute on the *EClass* construct, the *containment* meta-attribute and the *eOpposite* meta-reference on the *EReference* construct. As a consequence, when an Ecore-defined metamodel is reflected in the object-oriented modeling space, e.g. using the UML Class Diagram notation, the result graphical representation provides a blueprint of the being-modeled systems. We call the blueprint representation of a metamodel the *graph topology constraints* of models. The graph topology constraints are widely used as the underlying footprint to describe model transformations in graph rewriting-based transformation systems.

Since the model transformation reuse problem investigated in this thesis focuses on particular transformations which are defined in graph rewriting-based transformation systems, e.g. the MOMENT2-GT framework. The following sections will give a detail on the graph topology aspect, which is encoded in Ecore-defined metamodels and in graph-based transformation definitions. From these analysis, the remaining sections will present our model typing approach, which is based on the notion of *graph topology*.

3.2.1 Graph Topology in Metamodels

This section will detail an analyse on the graph topology, which is encoded in object-oriented metamodels. As aforementioned, metamodels are designed using the object-oriented metamodeling paradigm provide a blueprint representation for the natural physical structure of modeled systems. In using the UML Class Diagram notation, a metamodel of a domain specific can show a view on graph topology constraints based on which transformation writers define particular transformations.

In order to illustrate the graph topology and their similarity in metamodels, we take the two alternative source Ecore-defined metamodels of the motivating case study in Section 2.3.1. Figure 3.1 shows these two metamodels in the UML Class Diagram notation that we can consider as the blueprint of the physical structure of modeled systems. These two metamodels make the use of basic constructs of the Ecore metamodeling language, i.e. *EClass*, *EAttribute* and *EReference*, to define domain modeling concepts and their relationships between them. Note that constructs

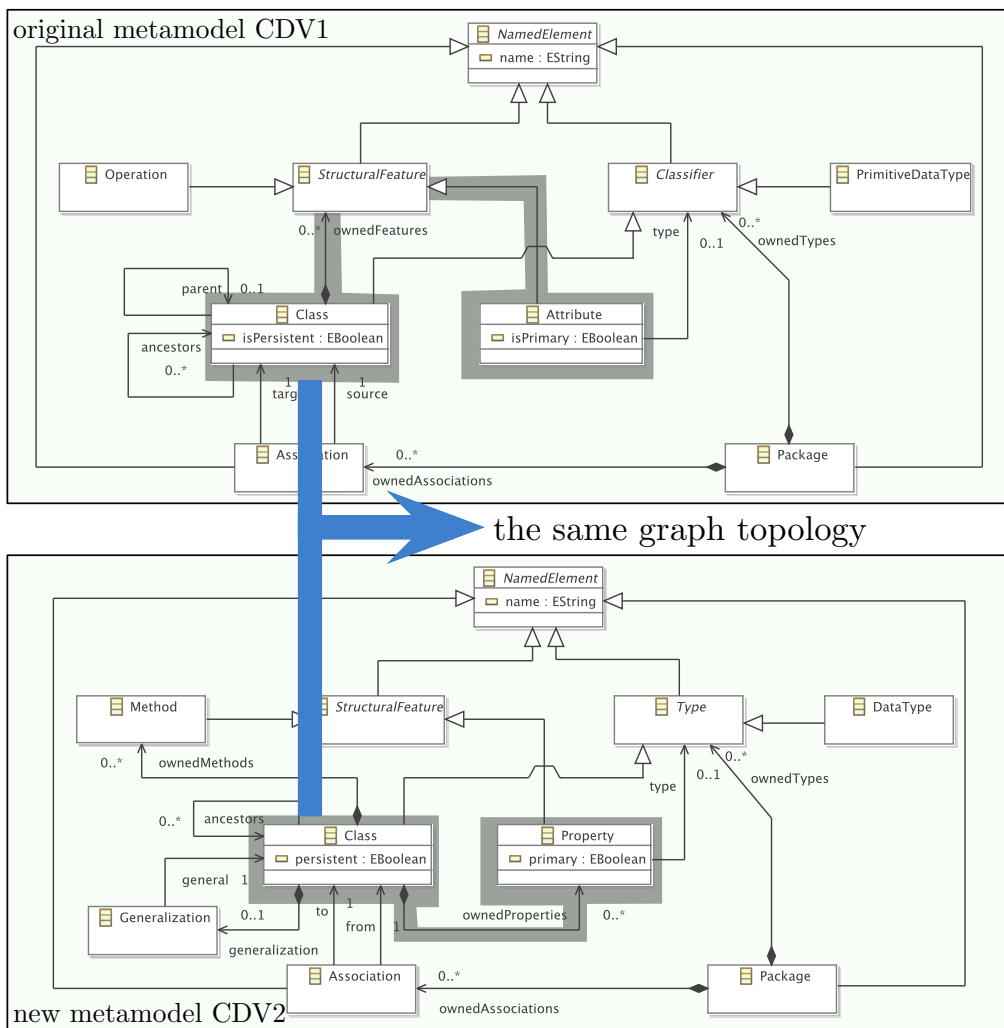


Figure 3.1: The same graph topology between metamodels.

which are used to declare user-defined metamodel *operations* are ignored because they do not relate to the structure aspect of models. As an example, the original source metamodel CDV1 defines a modeling concept named *Class* by instantiating the *EClass* metamodeling concept of Ecore and gives the value ‘Class’ for the *name* meta-attribute of the instantiated object. Similarly, the *Attribute* modeling concept is also defined by the same way. A relationship is defined between these two modeling concepts, but in an indirect manner, i.e. via a generalization mechanism provided by Ecore. More precisely, an “abstract” modeling concept, i.e. *StructuralFeature*, which generalizes the *Attribute* modeling concept is defined. Next, an instance of the *EReference* metamodeling concept is created as a compositional part of the *Class* modeling concept and is set the abstract concept *StructuralFeature* as the *eType* value. Because of the generalization relation already defined between *Attribute* and *StructuralFeature*, the relationship between *Class* and *Attribute* is inferred indirectly. That means, an instance of the *Class* concept can have multiple links with several

instances of the *Attribute* concept.

We can say these three above concepts and their relationships give a part of graph topology constraints for models of the being-modeled systems. If one considers a modeling concept in the UML Class Diagram plan as a node of a directed graph, thus relationships (direct or indirect via generalization relations) form edges between the graph nodes. The part of graph topology constraints, which is encoded by the above elements of the metamodel CDV1 is illustrated in the gray bounding-box, cf. the diagram at top of Figure 3.1.

The new metamodel CDV2 in fact defines the same part of graph topology constraints, cf. the gray bounding-box of the diagram at bottom of Figure 3.1, as the original metamodel CDV1. One can see the *Class* and *Property* modeling concepts form two graph nodes. However, an edge between them is now encoded directly via an instance of *EReference*, thus not necessary to be inferred by a generalization relation. In comparison with the original metamodel, although the new one has a different configuration, i.e. *Property* is a similar concept of *Attribute*, the persistent attribute concepts resided in the same concept *Class* have different names or type concepts are connected differently, it has the same (or similar) graph topology as one of the origin. Thus, at some level of abstraction, these two metamodels are called similar in term of the graph topology constraints.

The graph topology is in fact an important attribute of metamodels which describes how graph patterns are defined and constrained when using a graph rewriting-based transformation framework to design transformations. As a consequence, we can consider the transformation reuse problem as reusing graph topology constraints which are used by transformations for an alternative metamodel having an equivalence on graph topology.

The next section will detail the usage of metamodels' graph topology in defining transformation in graph rewriting-based transformation frameworks. In particular, the transformation example will be explained by using constructs of the MOMENT2-GT language. These details present the relation between graph pattern definitions in a transformation and graph topology of metamodels over which the transformation is well defined.

3.2.2 Graph Topology in Graph-based Transformations

Metamodel-based transformations are defined tightly coupled with the structure of metamodels, in other words that is the abstract syntax of the modeling language. In particular, transformations written in graph rewriting-based approaches focus on the graph topology attribute of metamodels and use it as the underlying footprint to define graph patterns in a transformation rule.

In MOMENT2-GT, a graph pattern is defined by using the following language constructs: *ObjectTemplateExp*, *AttributeExp* and *ReferenceExp*, cf. Figure 2.22 and Listing 2.5 in Section **MOMENT2-GT Language** at page 44 for more details on the abstract syntax and the textual concrete syntax of the MOMENT2-GT transfor-

mation language. These language constructs are connected by a recursive containment mechanism that can be instantiated to forms a graph pattern definition. Thus, a well-formed graph pattern definition is obviously constrained by graph topology constraints of the declared metamodels in a transformation.

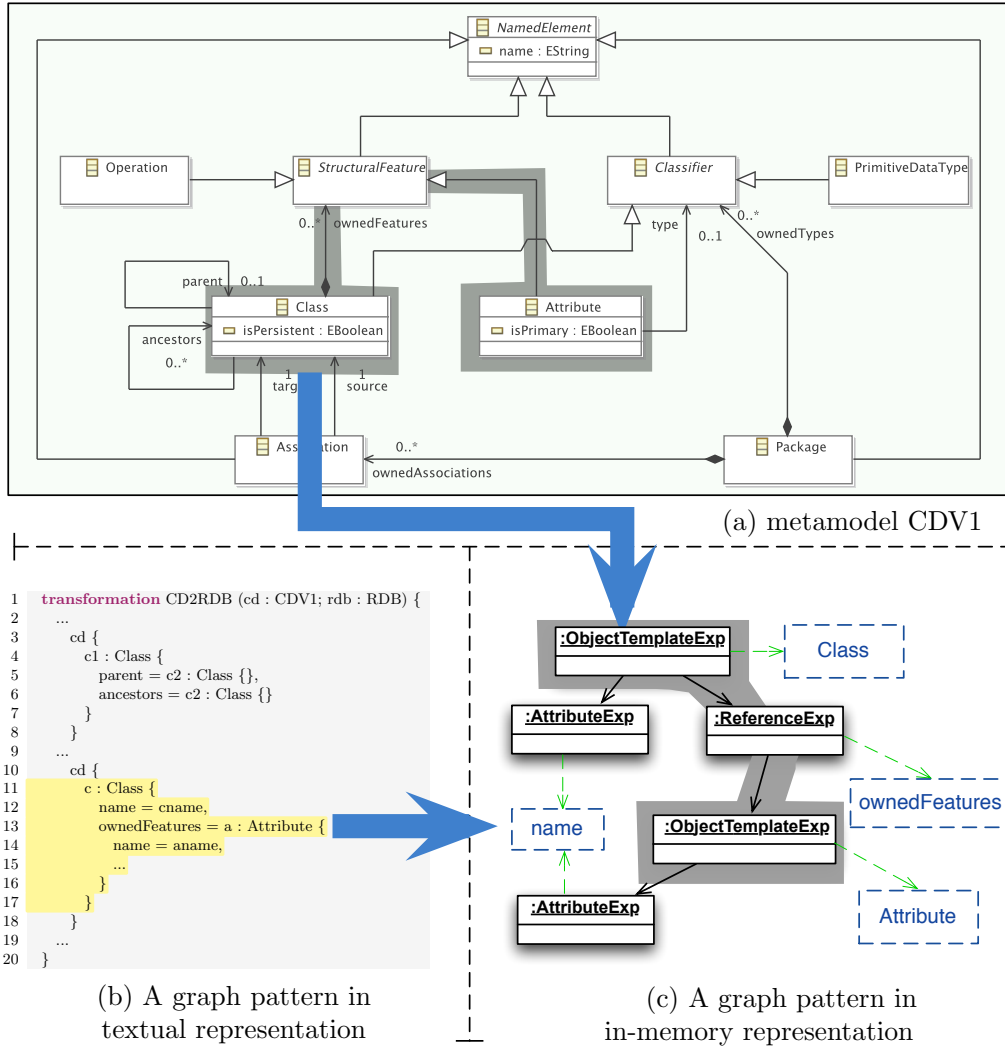


Figure 3.2: The graph topology in transformations.

Figure 3.2(a), Figure 3.2(b), Figure 3.2(c) shows the original source metamodel of the motivating case study, an excerpt of some graph patterns in textual representation format (input by transformation writers) and the in-memory representation (the abstract syntax representation after loading graph patterns in textual format) of those graph patterns, respectively. As shown in the highlight part of Figure 3.2(b), a transformation writer can define a graph pattern with an arbitrary depth in regarding the graph topology of the declared metamodel, cf. the gray bounding box in Figure 3.2(a), and in making the use of template expressions, Object Template Expression (OTEs) (*ObjectTemplateExp*), Property Template Expression including

Attribute Expression (AEs) (*AttributeExp*) and Reference Expression (REs) (*ReferenceExp*). In this example, the indirect relationship between the *Class* type concept and the *Attribute* type concept is used to define the graph pattern by making the use of the *ownedFeatures* reference concept. In addition, attribute concepts of type concepts might be used to constrain on the value of the slots of model elements matching the object template expressions. As a consequence, the used part of graph topology of the metamodel is encoded in memory by interconnected instance objects of OTE, RE, AE constructs, cf. the gray bounding box in Figure 3.2(c). Each template instance object refers to a metamodel element, either type concept or attribute concept or reference concept, in respect to graph topology constraints of the declared metamodels.

Although the graph topology of metamodels is an attribute fairly independent to the *name* property of metamodel elements, however, without names a metamodel element contains no meaning. As a result, the graph pattern, which is defined upon these metamodel element has no meaning too, i.e. the transformation engine cannot understand which objects of what concept kind we want to do a match. For that reason, the textual graph patterns are usually defined by taking the name (which plays a role as an identifier) of metamodel elements. Then, their in-memory reflection in a particular graph rewriting-based transformation language, e.g. MOMENT2-GT, maintains these names via references from template expression instances to real metamodel elements through a *static* name-based binding mechanism, obviously in respect to the metamodel definition.

From the graph topology perspective, one can recognize that graph patterns of transformations encode used graph topologies which must respect to those of metamodels. The encoded knowledge is reified through interconnected instance objects of basic constructs of a graph rewriting-based transformation language; for example in MOMENT2-GT these are OTE, RE, AE constructs. Although these objects are hidden to users, they play a role as a wrapper on certain part of graph topology constraints that the transformation writer want to use. From this point of view, when we alter the original metamodel by another metamodel which has a graph topology equivalent to the original, these hidden objects can be reused because they encode graph topology knowledge of the transformation. That is the notion of transformation reuse, which is focused in our approach, i.e. transformation reuse from the point of view of the graph topology similarity between metamodels. The remaining issue is that how to make wrapper objects be aware of new metamodel elements to rebind these objects correctly. In other words, we should provide a *rebinding* mechanism for instance objects of the transformation language's constructs to make them usable with new contexts, i.e. metamodels having similar graph topology to the original one, if possible.

In the remainder of the chapter we give the overall theoretic part of our work on the transformation reuse problem based on the coincident point of view of the graph topology attribute between metamodels and graph rewriting-based transformation approaches, presented in Sections 3.3–3.4. More precisely, in Section 3.3 we first define a notion of model type that is based on graphs. Then, two typing functions

for metamodels and transformations that result *declared* graph-based model types and *effective* graph-based model types, respectively, are presented. These foundations are introduced to solve the first sub-problem aforementioned in Section 3.1 for alternative metamodels that we call *isomorphic*. That means metamodels have slight differences in some meta-attributes of the constraint aspect, e.g. on multiplicity, or in connections between type concepts, but define the same set of domain concepts. Section 3.4 provides a solution to improve the re-usability of transformations when alternative metamodels are more different (the second sub-problem mentioned in Section 3.1), i.e. they do not define unify concepts, but rather similar or equivalent concepts. The proposed solution includes a mapping language, which is dedicated to establish correspondences between non-identical concepts defined in two alternative metamodels, an underlying compatibility check mechanism based on our notion of model type and a migration mechanism for transformations to be adapted to the new metamodel context.

3.3 Typing Models and Transformations with Graphs

In this section we provide a simple structure as the type of models that is based on the graph characteristic of metamodels. These model types are derived through a model typing function upon metamodels. Furthermore, a subtype relation is also established between metamodels based on a graph inclusion relation between the derived model types. This derived relation declares that one metamodel may be substituted for another under some conditions.

In order to extend the maximum possible flexibility and reuse of transformations, we provide two additional mechanisms. The first one will be presented in Section 3.3.4 that is performed via a transformation typing function which infers a so-called *effective* model type for a derived declared model type. This typing function is used to reduce constraints in a declared model type by an analyse on the actually used part on this type. The second mechanism will be introduced in Section 3.4 that includes a *mapping* DSL language, which is dedicated to describe the semantics correspondence between alternative metamodels' elements or combinations of these elements; and an interpreter coupled with this language for migrating an existing transformation to a new one which functions correctly with the new context of metamodels.

3.3.1 Type Graph with Multiplicity as Model Type

Metamodels are designed concisely by using *inheritance* provided in Ecore, as shown in Figure 3.1 (cf. Section 3.2.1). However, this representation does not denote explicitly the graph topology of metamodels. Since graph rewriting-based frameworks, e.g. MOMENT2, use the graph nature of metamodels to define transformations, Ecore-defined metamodels need to exhibit their graph topology.

For this purpose, we introduce a type graph in considering multiplicity constraint, namely Type Graph with Multiplicity (TGM) view. This abstraction, cf. Figure 3.3,

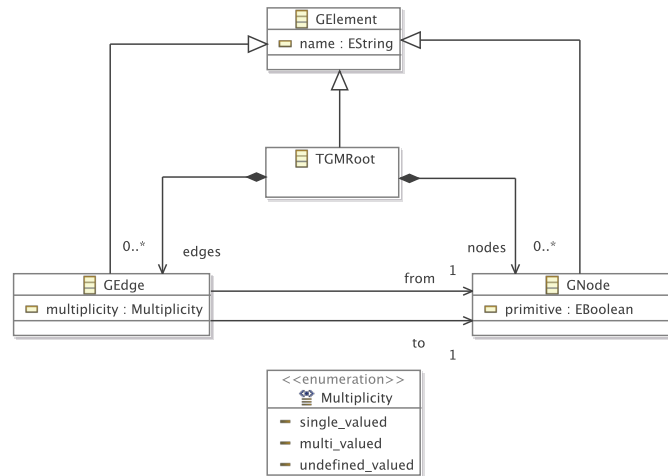


Figure 3.3: Metamodel of TGMs.

is used as an explicit representation of relationships between type definitions in meta-models when we consider each modelling type concept as a graph node and a direct (or indirect) relationship between two type concepts as a graph edge between their corresponding graph nodes. From this reflection, the use of TGM is to capture the graph topology of a model family plus some useful meta-information like entity names, property names, primitive types and multiplicity features.

Using this new graph structure for the notion of “model type” has two advantages. First, this graph structure plays a role as a middle layer for the gap between the metamodelling layer and the transformation layer. More precisely, from the graph topology point of view, the relationships between modelling type concepts defined in an Ecore-defined metamodel need to be (potentially) inferred from the inheritance relation of Ecore, while defining a graph pattern in a transformation rule, it seems only certain derived relationships are used. We call this situation *redundancy gap* in which the inheritance relation makes an explosion of modelling concept relationships, but only a subset of relationships is necessary from the transformation writer point of view. This middle layer provides a buffer data layer that we can freely infer and remove redundancies from two directions, i.e. Ecore-defined metamodels and transformations, without modifying elements in the spaces of metamodelling languages and transformation languages.

Second, by using a new graph structure, we can establish an abstraction level of “model type” notion. In our work, this level is in fact based on the abstraction level of the transformation language. For example, in the MOMENT2-GT language the *lowerBound* meta-attribute on a property of a type concept is omitted in type checking rules of the language because the semantics of the language is overcome this constraint. Another example is related to the *upperBound* meta-attribute (also called *multiplicity* attribute) that the language defines a semantics variation point at compile time. That means the MOMENT2-GT compiler, depending on a concrete value of *upperBound*, can vary the compilation behavior for an user-defined graph

pattern. For this reason, depending on the transformation language, we can select pertinent meta-attributes of the metamodelling language which have a semantics variation point in the transformation language and add them into the graph structure. In our work we consider the *upperBound* meta-attribute as an interesting semantics variation point to add into the graph structure. This meta-attribute is leveraged to the *Multiplicity* enumeration data type with three values: *single_valued*, *multi_valued* and *undefined_valued*. The semantics and the order relation of these values will be detailed in the next following sections. Having established “model type” based on a graph structure, we can define a relation between elements in the space of that graph structure to answer the question about the substitutability between them.

The following section will introduce a typing function dedicated to infer the TGMs from Ecore-defined metamodels that we call *model typing* function. The main operational semantics of this function is to produce a graph for a metamodel that represents explicitly the graph topology of the metamodel by flattening the inheritance relations used in the metamodel. Furthermore, this function eliminates constraint meta-attributes which do not affect the semantics with respect to the transformation language, i.e. MOMENT2-GT. For constraint meta-attributes related to a semantics variation point of the language, e.g. the *upperBound* meta-attribute, we leverage their semantics by introducing a new smaller domain, e.g. the previously introduced *Multiplicity* enumeration data type, and establishing an order relation between elements of the domain to verify the substitutability between TGMs.

3.3.2 Model Typing

Model types, i.e. TGM graphs in our notion, of metamodels are automatically generated by means of a *model typing* function as previously mentioned. The generation is implemented as a particular complex transformation from the Ecore view, i.e. the UML class diagram reflection, into the TGM view. Such a transformation alleviates the use of inheritance and leverages abstraction level of some essential constraint meta-attributes of Ecore. The typing function τ_{TGM} is defined by construction with the following algorithm over metamodels. We rely on an Ecore description of metamodels. Followings are the basic rules of the algorithm:

1. The root package, i.e. an instance of *EPackage*, that contains all other elements of a metamodel is transformed into a *TGMRoot* with the same name.
2. Used primitive data types (*EString*, *EInt*, *EBoolean*, etc.) are transformed into *Nodes* with the same names and the *primitive* attribute set to *true*.
3. Each user-defined type is transformed into a *Node* with the same name and the *primitive* attribute set to *false*.
4. Attributes and references are transformed into *Edges* by taking into account the Ecore semantics of transitive inheritances to explicit in-directed relationships between type concepts. Note that an attribute or a reference can have multiple derived *Edges* when a (transitive) inheritance is used on the container of the attribute or reference; or on the referenced type of the reference, c.f. Figure 3.4 for the illustration of this rule.

5. When transforming references, if the value of the *upperBound* meta-attribute is >1 or $=-1$, the *Multiplicity* attribute of the corresponding *Edge*(s) is then set to *multi_valued* (denoting ‘*’), otherwise it is set to *single_valued* (denoting ‘1’).

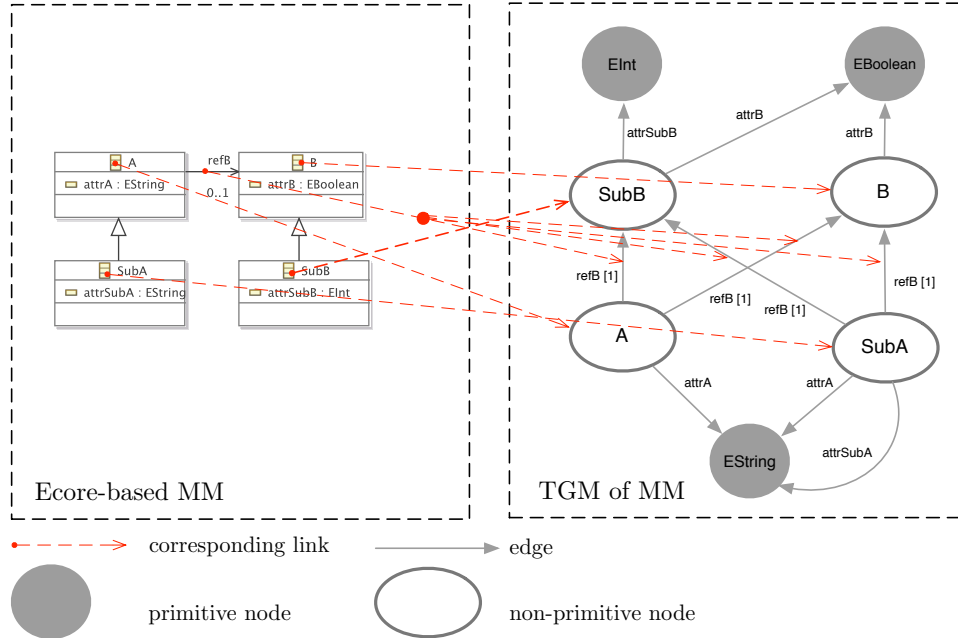


Figure 3.4: An example of transforming a metamodel to its derived TGM.

Note that setting the same name for the corresponding TGMRoot of a root package in the first rule is optional because this does not influence on the topology aspect that we are interested in metamodels. In contrast, every step in basic rules (2), (3), (4) must be respected when these rules construct the graph topology of a metamodel that is based on names as identifiers of graph elements. The last rule leverages the abstraction level of certain interesting meta-attributes. More precisely, in rule (5) the semantics of the *upperBound* meta-attribute on a reference from the domain of integers is leveraged to a smaller domain having only three literal values, i.e. Multiplicity enumeration data type.

Tracing

In order to stock the correspondence links between Ecore-defined metamodels' elements and TGM views' elements, we employ a generic interface named Trace, cf. Listing 3.1, to instantiate a global trace for our typing system. A concrete implementation of this interface is realized simply that allows to stock “one-to-many” bidirectional relationship between two sets of objects. In using this interface, we can find a corresponding metamodel element of a TGM view's element by invoking the *getSourceElem* method from the global trace object. On the contrary, the list of corresponding TGM view's elements of a metamodel element can be achieved through the *getTargetElems* method from the global trace object. The details of an implementation for the Trace interface is shown in Appendix 2.1.

Listing 3.1: Model typing function: the Trace interface in the Java language

```

1 package typesystem.trace;
2
3 import java.util.List;
4
5 /**
6  * This interface represents a simple one to many
7  * bi-directional mapping
8  */
9 public interface Trace<SRC, TGT> {
10
11     /* get a source element */
12     public SRC getSourceElem(TGT tgt);
13
14     /* get a list of target elements */
15     public List<TGT> getTargetElems(SRC src);
16
17     /* store a trace */
18     public void storeTrace(SRC src, TGT tgt);
19
20     /* remove a trace */
21     public void removeTrace(SRC src, TGT tgt);
22
23 }

```

The *model typing* function is charged in creating a TGM model type for each metamodel and uses a global trace for stocking corresponding links between them.

The Algorithm

The *model typing* algorithm (shown in Algorithm 1) has two inputs: (a) A metamodel mm , i.e. an instance of `EPackage`, for typing into the TGM space; (b) The initial global trace *TRACES* for keeping correspondences between metamodel elements and derived elements of the TGM view of the input metamodel.

The output of the algorithm is the TGM view tgm corresponding to the input metamodel mm and all corresponding links stored in the global trace. We briefly describe the working of the algorithm. Initially, cf. lines 1-2, the output TGM view tgm is created for the input metamodel root package mm , and then, a corresponding link between them is established and stored in the global trace. This step performs Rule 1 of the *model typing* function as aforementioned. The rest of the algorithm is divided into two main phases: (1) Creating non-primitive node for all user-defined types in the input metamodel; (2) Creating edges for (derived) properties of user-defined types, including attributes of a primitive type and references that refer to an user-defined type. Each time a new element of the output TGM is created, we always keep a link between it and the original element of the metamodel in the global trace.

The first phase of the algorithm implements Rule 3 that involves the creation of non-primitive nodes corresponding to user-defined types, i.e. types that are assigned an intentional semantics to describe a concept of the real world. These nodes are

Algorithm 1 *modelTypingFunction*(*mm*, TRACES)**Require:** the root package *mm* which contains user-defined types**Ensure:** the TGM of the input metamodel and corresponding links between them which are stored in TRACES

```

  {Rule 1. Creating a TGMRoot corresponding to the mm package}
1: tgm ← new TGMRoot(mm.name)
2: TRACES ← TRACES ∪ (mm, tgm)

  {Rule 3. Creating non-primitive GNodes for user-defined types}
3: for all eCl in mm.ECLASSIFIERS | eCl instanceof EClass do
4:   node ← new GNode(eCl.name, false)
5:   tgm.NODES ← tgm.NODES ∪ node
6:   TRACES ← TRACES ∪ (eCl, node)
7: end for

  {Rule 4. Creating GEdges for attributes and references of user-defined types}
8: for all eCl in mm.ECLASSIFIERS | fromNode ← TRACES[eCl][0] do

9:   for all eAtt in eCl.EALLATTRIBUTES do
10:    edge ← new GEdge(eAtt.name)
11:    edge.from ← fromNode

    {Rule 2. Creating primitive GNodes for used primitive types}
12:    if !tgm.NODES.exist(n | (n.name==eAtt.type.name) ⇒ edge.to ← n) then
13:      node ← new GNode(eAtt.type.name, true)
14:      tgm.NODES ← tgm.NODES ∪ node
15:      edge.to ← node
16:    end if
17:    tgm.EDGES ← tgm.EDGES ∪ edge
18:    TRACES ← TRACES ∪ (eAtt, edge)
19:  end for

20: for all eRef in eCl.EALLREFERENCES do
21:   for all subCl in getAllSubClasses(eRef.type) do
22:    edge ← new GEdge(eRef.name)

    {Rule 5. Abstracting the upper bound value of references}
23:    if eRef.upperBound == 1 then
24:      edge.multiplicity ← Multiplicity.SINGLE_VALUED
25:    else
26:      edge.multiplicity ← Multiplicity.MULTI_VALUED
27:    end if

28:    edge.from ← fromNode
29:    toNode ← TRACES[subCl][0]
30:    edge.to ← toNode
31:    tgm.EDGES ← tgm.EDGES ∪ edge
32:    TRACES ← TRACES ∪ (eRef, edge)
33:  end for
34: end for
35: end for

```

distinguished from those of primitive types, e.g. boolean or integer that has only the semantics of calculation, by the *primitive* attribute. In the algorithm, we first process user-defined types and leave primitive types for processing in the second phase because we do not want to take into account primitive types which are not used to construct user-defined types in the metamodel. From line 3 to line 7, we browse all user-defined types defined in the metamodel, i.e. elements in the set *mm.ECLASSIFIERS* that are instances of the EClass meta-class. For each type we create an instance of type GNode with the same name of that type and add the new instance into the set of nodes of the TGM root element *tgm*, i.e. *tgm.NODES*. Simultaneously, a pair of that type and its node is added into the global trace.

The second phase of the algorithm implements Rule 4, 2 and 5 that consists of the creation of edges corresponding to attributes, references between user-defined types and primitive types and between user-defined types themselves, respectively. From line 8 to line 35, we revisit all user-defined types of the metamodel and get simultaneously corresponding nodes for each type through the trace system *TRACES*. We separate the processing of properties of a being-visited user-defined type into two sub-phases: (1) creating edges for attributes; (2) creating edges for references. Note that the process also takes into account properties which are derived from the transitive inheritance between types thanks to *EALLATTRIBUTES*, *EALLREFERENCES* and *getAllSubClasses()*.

In the first sub-phase from line 9 to line 19, for each attribute of the type, including all derived attributes from its transitive super types via the inheritance relation, an object of the type GEdge is instantiated with the same name of the being-browsed attribute, then we set the corresponding node of the being-visited type to the *from* property of the new edge object. The next step relates to Rule 2 that processes for only used primitive types by the metamodel. More precisely, we solely create a new primitive node if the primitive type used to define the type property of the being-browsed attribute has not already existed in the output TGM view. In case that primitive node has already created, we set the node to the *to* property of the new edge object. Otherwise, we create a new primitive node then add it into the set of nodes of the TGM root elements. The final step includes adding the result edge into the set of edges of the TGM root element and pushing the pair of the being-browsed attribute and its derived edge into the global trace.

In the second sub-phase from line 20 to line 34, for each (derived) reference of the type, similarly to the first sub-phase, but for each other type that has an (transitive) inheritance relation with the type referred by the being-browsed reference, an object of the type GEdge is instantiated. The next step relates to Rule 5 that abstracts the interesting upper bound attribute of the being-browsed reference. More precisely, if the *upperBound* value is equal to 1, then we assign *SINGLE_VALUED* for the *multiplicity* property of the new created edge. Otherwise, that property is set to *MULTI_VALUED*. Then, we configure the *from* property of the new edge such that it refers to the corresponding node of the being-visited type, the *to* property to the node corresponding to the being-visited subtype of the reference's type. Finally, the edge is added into the set of edges of the TGM root element and then the pair of

being-browsed reference and its derived edge is pushed into the global trace.

Typing Judgement

The *model typing* function τ_{TGM} that is realized by Algorithm 1 is total by construction and we have:

$$\forall MM \in \text{metamodels}, \exists G \in TGMs \mid \tau_{TGM}(MM) = G$$

Note that we do not use directly the metamodel definition as the model type representation like most other approaches. Instead, by using the above model typing function, we shift the space of metamodels to another space in which constraints on the graph topology of a model family are exhibited explicitly. In addition, the TGM space gives to the notion of model type an interesting level of abstraction by selecting only certain interesting non-topology constraints and making relaxations on them, the constraint defined via *upperBound* attribute of references is an example. The selection of the *upperBound* attribute on the multiplicity constraint of references in our work is only a choice to show that we can choose a certain constraint attribute of the metamodeling language as interesting point of view for the notion of model type. From a selection, we can develop a computing language on models that can accept a larger set of models as legal inputs. In respect of this point of view, our model typing function introduces the judgement in an environment Γ for all model M that conforms to MM , M is of type G :

$$\frac{\Gamma \vdash M \hat{=} MM \quad \Gamma \vdash G = \tau_{TGM}(MM)}{\Gamma \vdash M : G} \quad (3.1)$$

In the above typing rule, the notation “ $\hat{=}$ ” denotes the typical conformance relation between a model and a metamodel, usually understood as the instantiation relation while the notation “ $:$ ” denotes the typing relation, which is defined in our work, i.e. the TGM space. Figure 3.5 shows the TGMs of metamodel CDV1 and CDV2.

Discussion on Model Type

Leverage Ecore-defined metamodels to a more abstraction type level such as the TGM view has a major advantage. That is, the introduced model typing function $\tau_{TGM}(MM)$ is not a one-to-one (also referred as *injective*) function, therefore an element in the TGM space can be derived from several different elements in the Ecore space. As a result, if we have two metamodels $MM1$ and $MM2$ in the domain defined by Ecore which have the same image G in the TGM space, then we can say all models that conform either metamodel $MM1$ or $MM2$ has the same type G , cf. Figure 3.6. Thus, a model transformation language that is developed at this level of abstraction can accept a larger set of models as inputs of a transformation definition. Among different transformation languages, MOMENT2-GT provides language constructs which have a semantics at the same abstraction level what we desire. Related

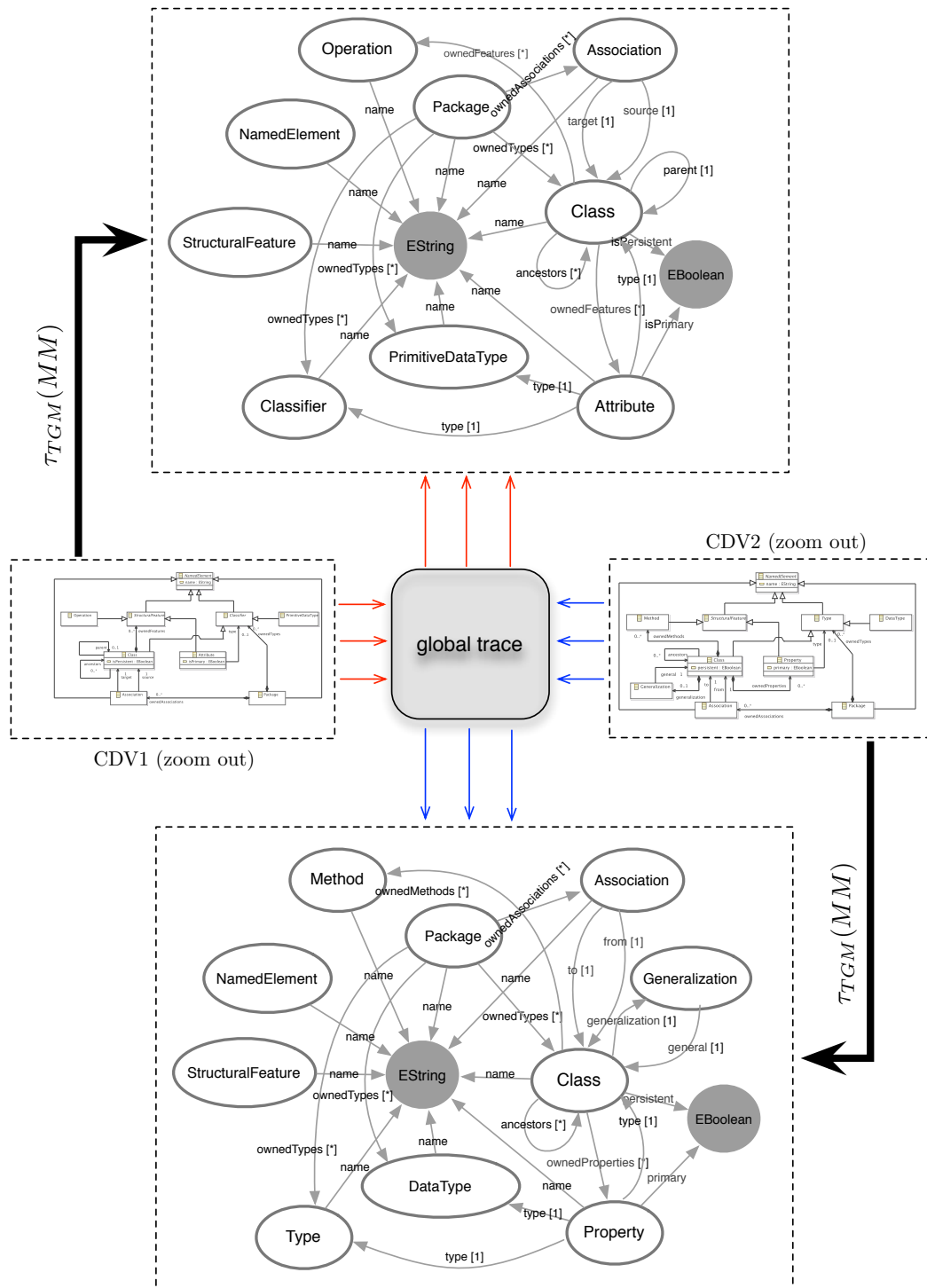


Figure 3.5: TGMs of metamodels CDV1 and CDV2.

to computations for primitive types, this language provides basic operators with simple typing rules, e.g. a variable declared as type String cannot be used as an Integer variable. Moreover, the language focuses on the graph characteristic of metamod-els and is able to overcome unimportant meta-attributes or in some cases provides a flexible compilation depending on the meta-attributes' values. In conclusion, the TGM abstraction view is aimed, on one hand, to capture topology constraints of a metamodel and on the other hand, to describe the semantic level in provided graph pattern-related constructs of the MOMENT2-GT language.

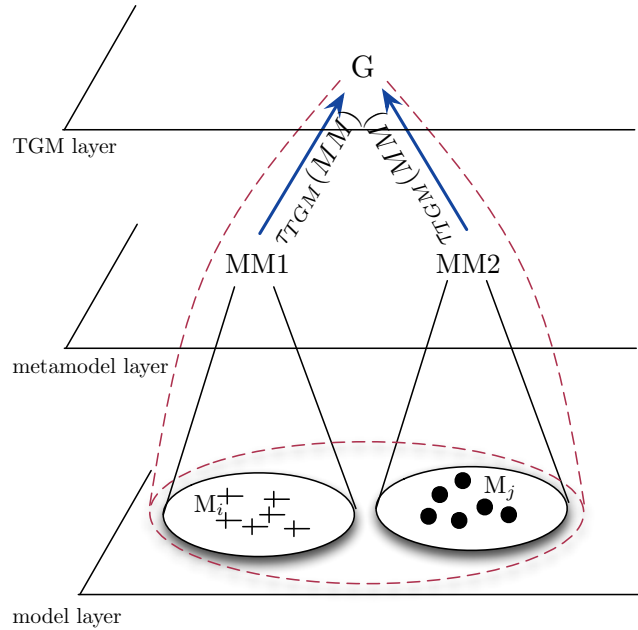


Figure 3.6: Models of different metamodels with the same model type.

3.3.3 Model Sub-typing

Having established a graph structure as TGM view with which we type models, it is necessary to answer a remaining question: under what conditions may one TGM model type be considered substitutable for another? In other words, we need to define an order relation between elements in the TGM space. This relation is not necessary to be total, but with such a relation, the acceptable set of models for a transformation can be extensible by an earlier checking at each time we want to use a transformation.

Sub-graph Overview

Thanks to the sub-graph relationship defined over graph, we make use of the model typing function $\tau_{TGM}(MM)$ to derive a sub-typing for models. We consider

that a GNode (resp. Vertex) is equal to another GNode (resp. Vertex) when their names are equals. The multiplicity of GEdges information needs a special treatment. We consider the following order over multiplicities:

$$\textit{undefined_valued} \preceq \textit{single_valued} \preceq \textit{multi_valued} \quad (3.2)$$

This means that *multi_valued* cardinality (‘*’) is more general than *single_valued* (‘1’) which is itself more general than the undefined situation, i.e. *undefined_valued*. The latter case which relates to how infer an *undefined_valued* multiplicity for an edge and the use of this value in typing will be discussed more detailed in Sections 3.3.4 and 3.4.1.

Hence, a GEdge with the same name as another one can be considered as equal in the sub-graph algorithm when the multiplicity of the sub-graph candidate is smaller than the super-graph multiplicity of the compared GEdge. If not, even with the same name, edges are different. From the graph point of view, such a sub-graph algorithm performs a look-up of mirrors for all nodes and edges of a sub-graph in the super-graph. The look-up operation depends basically on the *name* property of graph elements. Moreover, for edges the algorithm is also constrained with the multiplicity property. The algorithm for checking sub-graph relation between two TGM graphs is described in the Algorithm 2.

The Sub-graph Checking Algorithm

The *sub-graph checking* algorithm (shown in Algorithm 2) has two inputs: (a) A TGM root element *subTGM* which is a sub-graph candidate; (b) Another TGM root element *supTGM* which is a super-graph candidate. These root elements correspond to metamodel root packages which have been already inferred by applying the *model typing* function.

The output of the algorithm is a boolean value which indicates that the sub-graph checking between two graph is successful or not. If, after this look-up process is complete, every node and edge of the sub-graph candidate can find a mirror in the super-graph, then the sub-graph checking is successful. Note that the matching signature that we used is based on the *name* property of nodes and edges. Thus, since the algorithm is only applied for checking graphs that are derived from metamodels constrained by the uniqueness of name on concepts and properties, so each element in the sub-graph candidate cannot find more than one in its mirror in the super-graph. The algorithm is divided into main phases: (1) looking up correspondences in the super-graph for nodes in the sub-graph; (2) looking up correspondences in the super-graph for edges in the sub-graph. Each time an element of the sub-graph cannot find a correspondence, the algorithm immediately exits and the checking process is unsuccessful.

The first phase of the algorithm, cf. lines 1–11, involves to look up a correspondence (or a mirror) in the super-graph for nodes of the sub-graph. The signature we use to find a mirror node is based on the *name* and *primitive* properties of the type

Algorithm 2 $isTGMSubgraph(subTGM, supTGM)$

Require: the subgraph candidate $subTGM$ and the supergraph candidate $supTGM$. They are instances of type $TGMRoot$ and can be achieved from the global trace for each metamodel root package.

Ensure: return **false** if it can not find any mirror for at least a graph element of the subgraph candidate $subTGM$ in the supergraph candidate $supTGM$, otherwise return **true**

```

  {Looking up mirrors for all nodes of the subgraph in the supergraph}
1: for all  $subNode$  in  $subTGM.NODES$  do
2:    $isFound \leftarrow$  false {Initially, each  $subNode$  has not any mirror in  $supTGM$ }

3:   for all  $supNode$  in  $supTGM.NODES$  do
4:     if  $subNode.isMatch(supNode)$  then
5:        $isFound \leftarrow$  true
6:       break
7:     end if
8:   end for

   {exit if there is not any mirror in  $supTGM$  for the being-browsed  $subNode$ }
9:   if  $isFound ==$  false then
10:    return false
11:   end if
12: end for

  {Looking up mirrors for all edges of the subgraph in the supergraph}
13: for all  $subEdge$  in  $subTGM.EDGES$  do
14:    $isFound \leftarrow$  false {Initially, each  $subEdge$  has not any mirror in  $supTGM$ }

15:   for all  $supEdge$  in  $supTGM.EDGES$  do
16:     if  $subEdge.isMatch(supEdge)$  then
17:        $isFound \leftarrow$  true
18:       break
19:     end if
20:   end for

   {exit if there is not any mirror in  $supTGM$  for the being-browsed  $subEdge$ }
21:   if  $isFound ==$  false then
22:    return false
23:   end if
24: end for

  {If reach here, return true}
25: return true

```

GNode. More precisely, the auxiliary method *isMatch* defined for type GNode has the following semantics described in the OCL language:

```

1 context GNode def :
2   isMatch(n : GNode) : Boolean =
3     self.name = n.name
4     and self.primitive = n.primitive

```

Similarly, the second phase of the algorithm, cf. lines 12–22, involves to look up a correspondence (or a mirror) in the super-graph for edges of the sub-graph. The checking for matching edges require that they have the same name and matching multiplicity. In addition, their referred nodes (from/to) have to be matched, respectively. The OCL semantics of the *isMatch* method defined for type GEdge is following:

```

1 context GEdge def :
2   isMatch(e : GEdge) : Boolean =
3     self.name = e.name
4     and self.from.isMatch(e.from)
5     and self.to.isMatch(e.to)
6     and self.multiplicity.isMatch(e.multiplicity)

```

The *isMatch* method defined for enumeration type Multiplicity implements the order relation over multiplicity values in Formula 3.2.

Since the algorithm is based on cross-products of nodes and edges between two graph structures, thus the matching algorithm is $O(n^2)$ complex and reasonable costly. Even in the case of large Ecore-defined metamodels, the complexity is manageable for derived TGM model types.

The Sub-typing Approach

Figure 3.7 illustrates our sub-typing approach. Lets $M1$ and $M2$ be two models that conform (are defined with reference) to $MM1$ and $MM2$, respectively. $M1$ (resp. $M2$) does not conform to $MM2$ (resp. $MM1$) since $MM1$ and $MM2$ are not comparable in the Ecore space. Based on the model typing function proposed in the previous section, our sub-typing approach introduces $\tau(MM1)$ and $\tau(MM2)$, i.e. two graph views which derived from Ecore-defined metamodels. If we have $\tau(MM2) \subseteq \tau(MM1)$, then we can derive $M1 : \tau(MM1)$, $M2 : \tau(MM2)$ but also $M2 : \tau(MM1)$.

More formally, we have:

$$\tau(MM1) \subseteq \tau(MM2) \Leftrightarrow MM2 \preceq MM1 \quad (3.3)$$

Intuitively, a metamodel $MM2$ is a subtype of a metamodel $MM1$ if and only if the TGM model type derived from $MM1$ (i.e. $\tau_{TGM}(MM1)$) is a sub-graph of its derived TGM model type (i.e. $\tau_{TGM}(MM2)$).

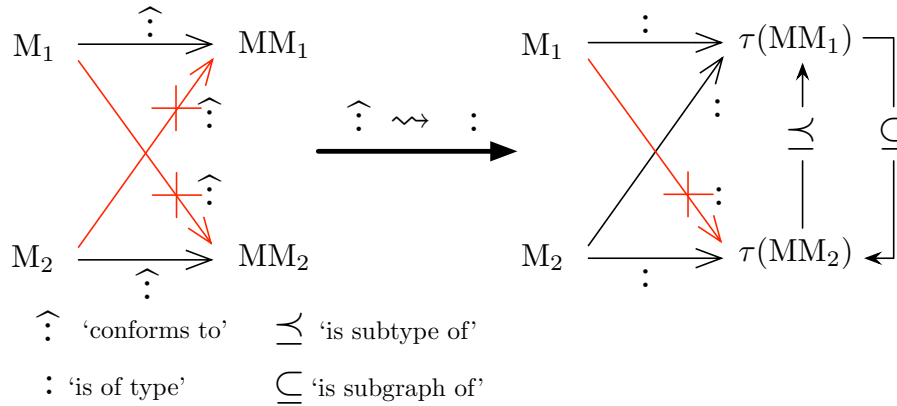


Figure 3.7: Sub-typing approach.

Discussion on Model Sub-typing

Now back to the main problem of transformation reuse that is focused in this thesis. Although we have already introduced a *model typing* function for metamodels and a sub-typing mechanism based on that function, it is even not sufficient to solve the transformation reuse problem. One reason is the redundancy when we declared a metamodel as the constraint context in transformation designing. That is, a transformation uses a small part of the declared metamodel. This argument results to the need of typing also the transformation that produces the real model type for an existing transformation. Intuitively, a real type of a transformation is usually “smaller” than the declared type. In the next section, we will introduce such a typing mechanism. Our transformation typing approach is based on metamodel elements which are used to define a transformation, but results in an artifact at a higher abstraction level than metamodel level, i.e. an element in the TGM space.

3.3.4 Model Transformation Typing

Model transformation languages are defined with references to metamodels as constraints over model they accept as input or output parameters. This way of defining transformations makes them hardly reusable when the source of the model relies on a slightly different metamodel. In order to tackle this issue and accept more models as input (or output), we propose in this thesis two typing functions for model transformations.

First, considering a transformation as a “function” form one metamodel to another metamodel (the second could be the same in case of so-called endogenous transformations), we define the type of a transformation model as an arrow type from the type derived from the input metamodel to the type derived from the output metamodel. More formally, if T is a model transformation from MM_i to MM_o :

$$\Gamma \vdash T : \tau_{TGM}(MM_i) \rightarrow \tau'_{TGM'}(MM_o) \quad (3.4)$$

The above arrow type could be generalized to multi-input, multi-output model parameters. Note that we distinguish metamodels that have served as constraints for input parameters and those that have served as constraints for output parameters of a transformation. That is we are not allowed to create new objects in an input model that conforms to an input metamodel, in contrast to the output model. This distinction comes from the *containment* meta-attribute defined on a reference that relates to the creation of object, i.e. an object cannot be created outside its container. The *model typing* function τ_{TGM} that we introduced in Section 3.3.2 based on the notion of model type TGM proposed in Section 3.3.1 focuses currently on reusing an existing transformation when changing input metamodels. Thus, the constraint on object creation is not in consideration. That is why we do not take into account the *containment* meta-attribute as a selection to add to the set of properties of type GEdge of the TGM graph structure. Thus, we do not apply the *model typing* function τ_{TGM} for metamodels declared in output parameters. Model transformation reuse when changing metamodels in output parameters requires more investigation. This can be done by developing another *model typing* function $\tau'_{TGM'}$ based on another graph structure TGM' by taking into account the *containment* meta-attribute.

The second typing function of model transformation relies on the code of the transformation itself which encodes the transformation designer's intention. We observe that most transformation do not use all concepts (entities and their attributes, or associations) of the referenced metamodels. We call this the effective model type as [Sen 09] for models. In the type-theoretic sense, it represents a *super-type* of the declared type. The *effective* TGM model type is useful since it reduces constraints on types of a legacy transformation and makes the set of substitutable metamodels larger.

$$\Gamma \vdash T : \tau_{eff}(\tau_{TGM}(MMi)) \rightarrow \tau_{eff}(\tau'_{TGM'}(MMo)) \quad (3.5)$$

The *effective* TGM model types of a transformation needs to be inferred from its implementation. To this end, the function τ_{eff} in formula 3.5 have to implement an algorithm that is dependent on the transformation language. We briefly describe it in the context of MOMENT2-GT in the following paragraphs. This function takes a metamodel and an existing transformation definition as main inputs to infer the *effective* TGM model type instead of using directly the derived TGM model type as an input as in formula 3.5. We remind that MOMENT2-GT describes a transformation by using the *graph pattern* structure, cf. Section **MOMENT2-GT Language** at page 44 for more detail on the *graph pattern* structure.

Algorithm Overview

In Figure 3.8, we present an overview of the *effective* TGM model type inference mechanism. The inputs to the algorithm are: (1) A metamodel MM which has a derived TGM model type through the application of the *model typing* function τ_{TGM} introduced in Section 3.3.2. The derived model type is large, for instance the TGM of simple metamodel such as CDV1 in our motivating example has 11 nodes and

24 edges as shown in Figure 3.5 (at the top); (2) The transformation MT which encodes the real usages of graph topology corresponding to the metamodel MM by the transformation designer's intention.

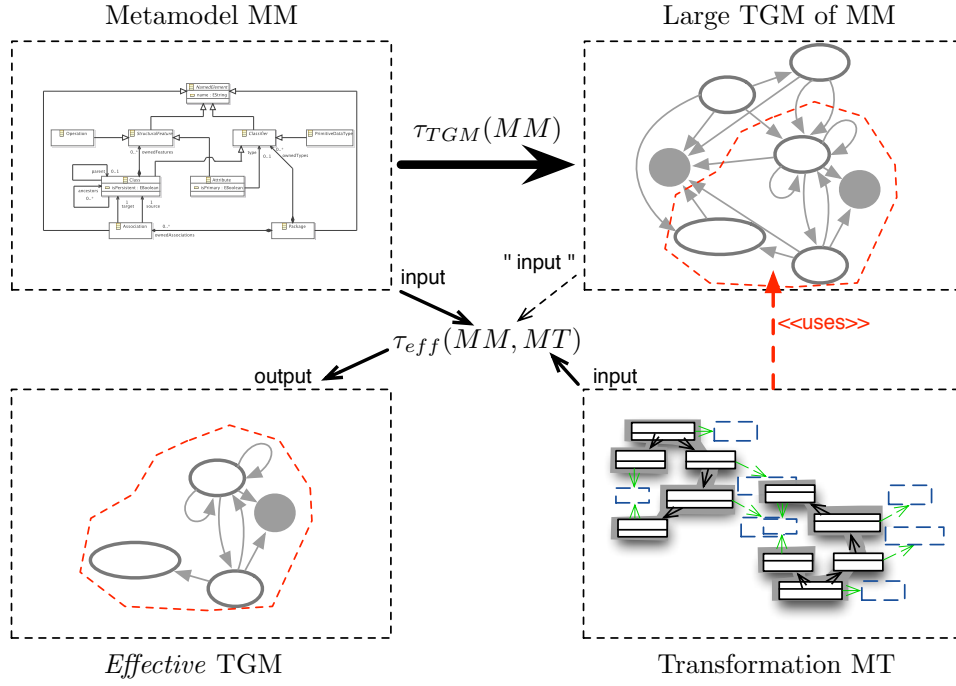


Figure 3.8: Transformation typing overview.

The output of the algorithm is an inferred *effective* TGM model type of the input metamodel MM. This *effective* TGM model type contains only graph topology parts used to define the transformation, cf. nodes and edges within the red bounding line of the large TGM graph of MM. Some of attributes on reserved graph elements may be changed during the inference process. For example, the *multiplicity* attribute on an edge can be changed from *multi_valued* to *undefined_valued* when we find somewhere in the transformation an usage that can be more abstracted. All other unused graph elements are removed in the original large TGM graph and simultaneously in the global trace. The pruned *effective* TGM graph is certainly a subset and presents a supertype of the original large TGM graph. Note that in our approach we do not infer (or modify) on the input metamodel, but on the derived model type from it.

The Algorithm

The transformation typing function τ_{eff} corresponds to the *transfTypingFunction* function shown in Algorithm 3. This function takes three inputs: (a) A metamodel *mm* that we want to infer the *effective* parts of its derived TGM model type; (b) The transformation *mt* that is considered as an usage context on that metamodel; (c) The global trace *TRACES* that already keeps correspondences between elements of the metamodel and derived elements of the TGM model type of the metamodel.

Algorithm 3 *transfTypingFunction*(mm, mt, TRACES)

Require: (1) the metamodel mm to infer the *effective* TGM model type. (2) the transformation mt as a usage context. (3) the global trace TRACES stocks correspondences between the metamodel and its original TGM model type.

Ensure: Return the *effective* TGM model type corresponding to the input metamodel in taking into account usages encoded in the transformation.

```

1:  $\text{OTES}_{top} \leftarrow \text{collectTopOTES}(mt, mm)$  {Collect top OTES constrained by mm}
2:  $\text{NODES}_{req}, \text{EDGES}_{req} \leftarrow \emptyset$  {Initialize require sets of nodes and edges}
3:  $\text{EDGES}_{multi} \leftarrow \emptyset$  {Initialize sets of edges multi-used multiply in a graph pattern}
4:  $tgm \leftarrow \text{TRACES}[mm][0]$  {Get the TGM model type of mm}

   {Phase 1: Collect all required nodes and edges used in the transformation mt}
5: for all  $ote$  in  $\text{OTES}_{top}$  do
6:    $\text{objTETyping}(tgm, ote, \text{NODES}_{req}, \text{EDGES}_{req}, \text{EDGES}_{multi}, \text{TRACES})$ 
   {Visit recursively beginning at the top object template expression ote}
7: end for

   {Phase 2: Remove all non-required graph elements}
   {2.1: Remove non-primitive nodes and its edges not in required sets}
8: for all  $eCl$  in  $mm.ECLASSIFIERS$  |  $node \leftarrow \text{TRACES}[eCl][0]$  do
9:   if  $node \notin \text{NODES}_{req}$  then
10:     $\text{TRACES} \leftarrow \text{TRACES} \setminus (eCl, node)$  {for non-required nodes}
11:     $tgm.NODES \leftarrow tgm.NODES \setminus node$ 
12:   end if
13:   for all  $p$  in  $eCl.ESTRUCTURALFEATURES$  do
14:     for all  $edge$  in  $\text{TRACES}[p]$  |  $edge \notin \text{EDGES}_{req}$  do
15:        $\text{TRACES} \leftarrow \text{TRACES} \setminus (p, edge)$  {for non-required edges}
16:        $tgm.EDGES \leftarrow tgm.EDGES \setminus edge$ 
17:     end for
18:   end for
19: end for

   {2.2: Remove primitive nodes not in the required set of nodes}
20: for all  $node$  in  $tgm.NODES$  |  $node \notin \text{NODES}_{req}$  do
21:    $tgm.NODES \leftarrow tgm.NODES \setminus node$ 
22: end for

   {Phase 3: Infer multiplicity value on edges}
23: for all  $edge$  in  $tgm.EDGES$  |  $edge \notin \text{EDGES}_{multi}$  do
24:   if  $edge.multiplicity == \text{Multiplicity.MULTI\_VALUED}$  then
25:      $edge.multiplicity \leftarrow \text{Multiplicity.UNDEFINED\_VALUED}$ 
26:   end if
27: end for

```

The outputs of the function are: (a) The same TGM model type tgm of the input metamodel, but without all elements, i.e. nodes and edges that are not used in the usage context, i.e. the input transformation mt ; (b) The global trace $TRACES$ that has been updated.

We briefly describe the inference process of the algorithm. The main idea of the algorithm is first to detect all graph nodes and edges that are used in all user-defined graph patterns of the transformation, then remove all unused nodes and edges in the derived model type. To this end, initially we collect all top object template expressions (OTE) that are constrained by the input metamodel mm . The algorithm then is divided into three main phases: (1) Computing set of all required nodes and edges by visiting all top object template expressions. This visiting process is performed recursively due to the recursion on defining an object template expression. The required sets of nodes and edges are accumulated over the visiting process; (2) Removing all nodes and edges that are not in the required sets; (3) Inferring the *multiplicity* value on edges that are multi-used, if any, based on a feedback from the so far visiting process. We will go into details of Algorithm 3 in the further paragraphs.

The initial phase, from lines 1–4, first collects all top object template expressions into $OTES_{top}$ by using an auxiliary function, i.e. $collectTopOTES$, with the transformation mt and the metamodel mm as inputs. Then, the sets of required nodes and edges, i.e. $NODES_{req}$ and $EDGES_{req}$, respectively, are initialized as empty sets. So on for set $EDGES_{multi}$ that will contain edges multi-used in graph patterns. In the end of this phase, we obtain the derive TGM model type corresponding to the input metamodel based on the global trace.

The first phase of the algorithm involves the computation of the entire set of required nodes and edges that will be added into sets $NODES_{req}$ and $EDGES_{req}$, respectively. From lines 5–7, the initial sets $NODES_{req}$ and $EDGES_{req}$ are passed into the $objTETyping$ function, and accumulated over all top object template expressions in the set $OTES_{top}$. The $objTETyping$ function acts as a visitor that travel recursively the entire graph pattern defined by the top object template expression. Algorithm 4 that details the mechanism of that function on selecting required nodes and edges will be explained later. In addition to above arguments, this function also receives others such as $EDGES_{multi}$ and the global trace.

In the second phase of the algorithm we remove nodes and edges from the derived TGM model type of the metamodel mm . From lines 8–12, we remove all nodes that are not in $NODES_{req}$ and simultaneously update $TRACES$ by removing the correspondence link between the node and its corresponding class. Similarly, from lines 13–19 we remove all edges that are not in $EDGES_{req}$ and simultaneously update $TRACES$ by removing the correspondence link between the edge and its corresponding property. In the end, from lines 20–22 we remove all primitive nodes that are not used without a need on updating $TRACES$.

The third phase of the algorithm consists of inferring *multiplicity* value on required edges that are multi-used in a graph pattern, if any. So in theory, we face two cases. In the first one where single relation is employed in the metamodel, we do

not change the *multiplicity* value in the *effective* TGM since multiple relations also can be applied. In the second case where a multiple relation is present, we can use both single and multiple pattern constraints to define a model pattern. When using single pattern constraints, we change the *multiplicity* value in the *effective* TGM from *multi_valued* to *undefined_valued* (denoting ‘?’) since all cases can be applied, otherwise, the value of *multiplicity* is left to *multi_valued* since only multiple relations can be applied. This rationale is in fact based on a semantics variation point when defining a graph pattern in using the Object Template Expression construct of the MOMENT2 language. In order to improve transformation reuse capability, we exploit this semantics variation point and try to abstract this based on an analyse on the real usage of the transformation designer. In practice, in the algorithm we use $EDGES_{multi}$ to stock edges that are multi-used in a graph pattern that are detected in the visiting process described in Algorithm 4. From lines 23–27, we scan all remaining edges, if the being scanned edge is in $EDGES_{multi}$ and has the current *multiplicity* value as MULTI_VALUED, then changing its *multiplicity* value to UNDEFINED_VALUED.

Now we go in-depth on Algorithm 4 that describes the *objTETyping* function used in the first phase of Algorithm 3. This function is the kernel of our transformation typing approach that implements a semantics to detect all required nodes and edges in the derived model type from a graph pattern definition in MOMENT2-GT.

The inputs of function *objTETyping* are: (a) A derived TGM model type *tgm*; (b) An object template expression *ote* at some level in the being-processed graph pattern; (c) A set of required nodes $NODES_{req}$; (d) A set of required $EDGES_{req}$; (e) A set of edges that are multi-used $EDGES_{multi}$; (f) the global trace *TRACES*. Among these inputs, $NODES_{req}$, $EDGES_{req}$, $EDGES_{multi}$ are also outputs that accumulate required elements. The function is designed into three phases: (1) Adding to $NODES_{req}$ the node corresponding to the class referred by the being-visited object template expression *ote*; (2) Adding to $EDGES_{req}$ all edges used to define property template expression of the *ote*, performing a depth-first visit when processing a reference template expression by invoking this function itself; (3) Computing edges that are multi-used in the being-visited object template expression *ote*.

The first phase of the algorithm, from lines 1–3, first get the class referred by the being-visited object template expression, then looks up in the global trace the corresponding node that is called *fromNode* in this current context. Second, we add *fromNode* to $NODES_{req}$. Note that by using a structure of type Set, the \cup operator will not add an element if it already exists in the list.

The second phase of the algorithm is divided into two alternative processes. To begin, we initialize the set $EDGES_{visited}$ as an empty list, cf. line 4. This list is used to compute edges with multi-used in the third phase. The main processing is performed by scanning all property template expressions that are defined in the being-visited object template expression. In the first case (from lines 6–15), i.e. where that property template expression is an attribute expression, we look up in the global trace the corresponding edge of the referred attribute, then add it to $EDGES_{req}$ and the primitive node referred by it to $NODES_{req}$. In the second case (from lines 16–

Algorithm 4 *objTETyping*(*tgm*,*ote*,*NODES_{req}*,*EDGES_{req}*,*EDGES_{multi}*,*TRACES*)

Require: (1) TGM model type *tgm*, (2) an object template expression *ote*, (3) the required set of nodes, (4) the required set of edges, (5) the set of edges multi-used, (6) the global trace.

Ensure: the required sets of nodes and edges that are used in the object template expression, and a set of edges with multi-use in a graph pattern.

```

    {Phase 1: add the node corresponding to the class referred by ote to NODESreq}
1: referredClass ← ote.referredClass
2: fromNode ← TRACES[referredClass][0]
3: NODESreq ← NODESreq ∪ fromNode

    {Phase 2: add edges used to define property template exp. of ote to EDGESreq}
4: EDGESvisited ← ∅ {Initialize a list of visited edges, for counting edges multi-used}

5: for all pte in ote.PROPTEMPEXPS do
6:   if pte instanceof AttributeExp then
7:     referredAttribute ← pte.referredProperty
8:     if tgm.NODES.exist(n | (n.name == referredAttribute.type.name) ⇒ toNode ← n) then
9:       for all edge in TRACES[referredAttribute] do
10:        if edge.from.equals(fromNode) and edge.to.equals(toNode) then
11:          EDGESreq ← EDGESreq ∪ edge
12:          NODESreq ← NODESreq ∪ toNode
13:        end if
14:      end for
15:    end if
16:  else if pte instanceof ReferenceExp then
17:    referredReference ← pte.referredProperty
18:    lowerOTE ← pte.value
19:    referredLowerClass ← lowerOTE.referredClass
20:    toNode ← TRACES[referredLowerClass][0]
21:    for all edge in TRACES[referredReference] do
22:      if edge.from.equals(fromNode) and edge.to.equals(toNode) then
23:        EDGESreq ← EDGESreq ∪ edge
24:        EDGESvisited.add(edge) {add to a list to treat multiplicity in phase 3}
25:      end if
26:    end for

    {call recursively for the lower object template expression}
27:    objTETyping(tgm,lowerOTE,NODESreq,EDGESreq,EDGESmulti,TRACES)
28:  end if
29: end for

    {Phase 3: add edges which are used more one time to the set of edges multi-used}
30: for all edge in EDGESvisited do
31:   if EDGESvisited.count(edge) > 1 then
32:     EDGESmulti ← EDGESmulti ∪ edge
33:   end if
34: end for

```

28), i.e. where that property template expression is a reference expression, we get the lower object template expression that referred by the property template expression. Next, the *toNode* corresponding to the class referred by the lower object template expression is achieved from the global trace. We look up an edge in the list of correspondence edges of the referred reference that its two endings are identical with *fromNode* and *toNode*. Then, we add the found edge to $EDGES_{req}$ and update it to the list $EDGES_{visited}$. The final step of the second phase is invoking the *objTETyping* function itself by passing the lower object template expression, cf. line 27.

In the third phase of the algorithm, we count the number of visited time for edges that are used to define the being-visited object template expression. For those with visited time number more than one, we add them to the set $EDGES_{multi}$. This information is used to infer the *multiplicity* value of required edges in the third phase of Algorithm 3.

Application on the Motivating Case Study

We apply the transformation typing function τ_{eff} for the source metamodel CDV1 in taking into account its usage context in the transformation CD2RDB. The inferred *effective* TGM model type of metamodel CDV1 is given in Figure 3.9 (at the bottom). This *effective* TGM model type is reduced a lot in comparison with the original one (at the top of Figure 3.9). It contains only 7 nodes (including 2 primitive nodes) and 13 edges. In addition, some edges are modified on the *multiplicity* attribute to relax the constraint on edges, e.g. the *ownedTypes* and *ownedFeatures* edges. The *effective* TGM model type of CDV1 represents the real type for the input model parameter after finishing the design of the transformation. In applying our sub-typing approach in Section 3.3.3, whatever metamodel that has a derived type, which is satisfied the sub-graph relation with that *effective* model type can safely replace directly the metamodel CDV1 in the transformation CD2RDB without a need of adaptation.

Discussion on Model Transformation Typing

Providing a model transformation typing function as the τ_{eff} function facilitates to complete a better solution for the first sub-problem mentioned in Section 3.1. It is recalled that in the first sub-problem of the transformation reuse issue, with a given solution we can only reuse transformation across *isomorphic* metamodels. That is, metamodels must share a set of same modelling concepts (i.e. these concepts are identical in names) and the same graph topology. These assumptions in fact are rare in real situations where metamodels could arrive from different organizations. In reality, even in a particular specific domain, organizations use different, but similar modelling concepts, i.e. the meaning of two modelling concepts are the same or similar, but the names that they assign to them are different. Moreover, even in the same modelling concept, the name of attributes may be defined differently and so on for relationships between modelling concepts. In addition, the designs of a metamodel for a specific domain in organizations are very different because metamodel design is a human activity so it depends on different human decisions. For example, a relation-

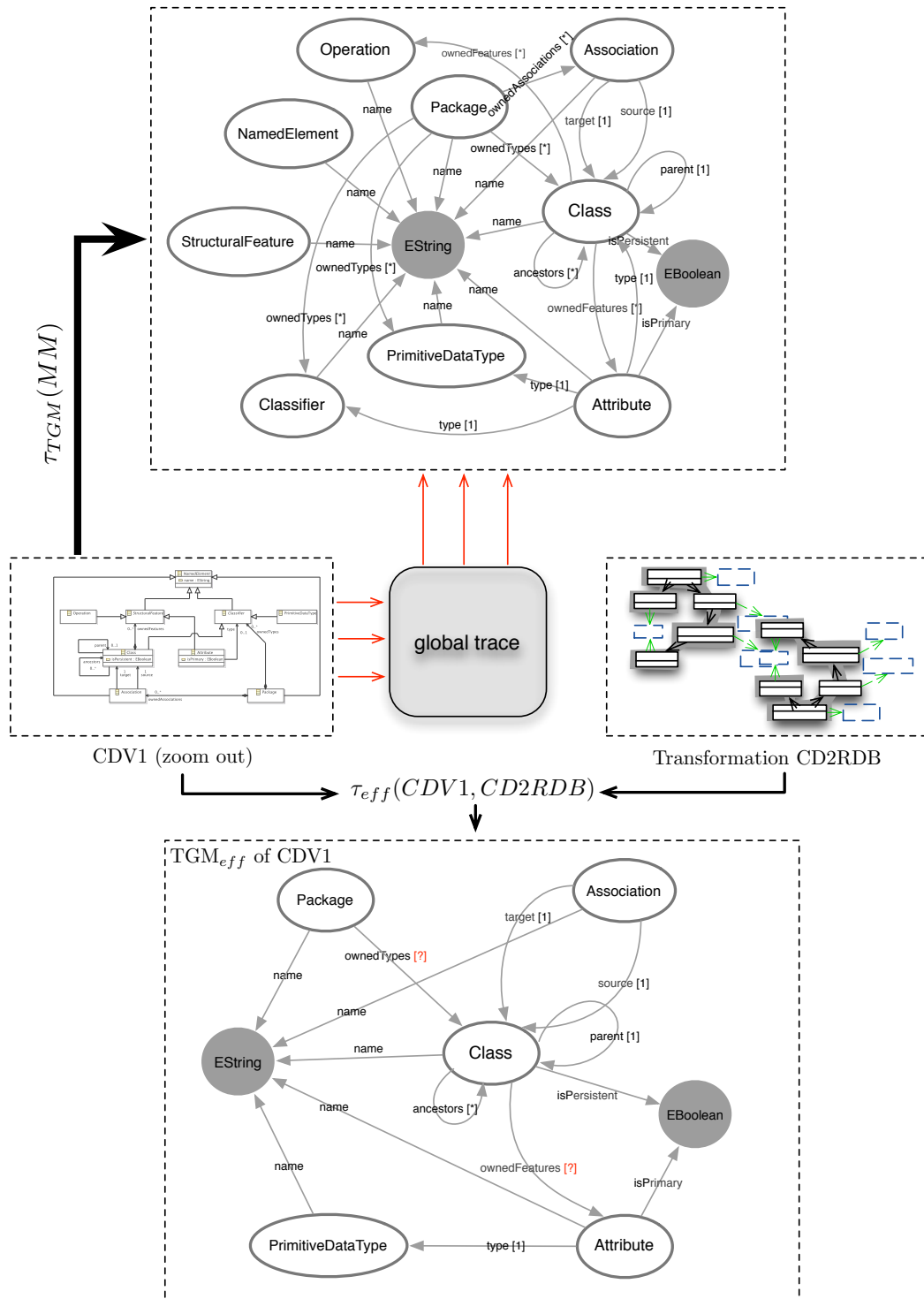


Figure 3.9: Inferred effective TGM for CDV1 (at the bottom).

ship between two modelling concepts designed by an organization can be reified in another organization through an additional concept. It is recalled that we classified these situations in the second sub-problem in Section 3.1 in which metamodels are called *non-isomorphic*.

For *non-isomorphic* metamodels, there is no way to reuse directly existing transformations without modifying (or adapting) them. To this end, the adapting process requires some additional directives from an external source. A simple and straightforward choice is to develop a *mapping* DSL language dedicated to describe semantic correspondences between metamodel elements or combination of metamodel elements, even they have different names or partial different topology. A *mapping* definition in principle can be derived from a dictionary of synonyms or a global ontology shared within organizations. However, using these resources requires another complex inference process and needs a transformation from these spaces into the *mapping* DSL space. Since semantics of a concept is a notion that is assigned by human, in this thesis, we chose a solution in which a *mapping* definition is defined directly by a human input. After having an explicit *mapping* definition, we use the proposed typing solution for the first sub-problem as primarily premises to validate the correctness of the mapping definition. Obviously, a *mapping* definition should be correct or not depending also on the adaptation semantics of the language that we can support. The solution for the second sub-problem on developing a *mapping* DSL language will be presented in the next section.

3.4 Non-isomorphic Transformation Reuse

The previous sections introduce model and model transformation typing functions coupled with a sub-typing mechanism for models. Thanks to this abstraction approach, the reuse capability of transformations is improved. In fact, it allows models that conforms to different metamodels to be used by a transformation when their model type is a subtype of the *effective* model types used by the transformation.

The proposed typing calculus is very simple and it is dedicated to solve the first sub-problem known in Section 3.1. This situation is also mentioned in a recent publication [Guy 12] and is named isomorphic. More complex situations happen when we want to reuse a transformation on models that conform to a metamodel with different names or partial different topology (called non-isomorphic in [Guy 12] or known as the second sub-problem in Section 3.1). This section presents a solution to reuse transformations for the second sub-problem and when only their source metamodels are changed (such as the motivating example). Particularly, to reuse such a transformation, the following four steps need to be performed:

1. Build two graph-based views, i.e. TGM, for the original and new metamodels, respectively.
2. Infer the effective parts of the original TGM that are actually used by the transformation.

3. Describe the correspondences between elements (or combinations of elements) of two metamodels in a proposed *mapping* DSL language. This user task is assisted and type-checked using the results of steps 1 and 2.
4. Execute an interpreter that adapts the transformation such that it works correctly with the new source metamodel.

While Step 3 requires manual input, the other steps could be automated by tools. More precisely, Step 1 is performed by means of the *model typing* function presented in Section 3.3.2; Step 2 applies the *model transformation* typing function that is introduced in Section 3.3.4 and Step 4 will be carried out automatically by the HOT of the language if the mapping definition is validated at the end of Step 3.

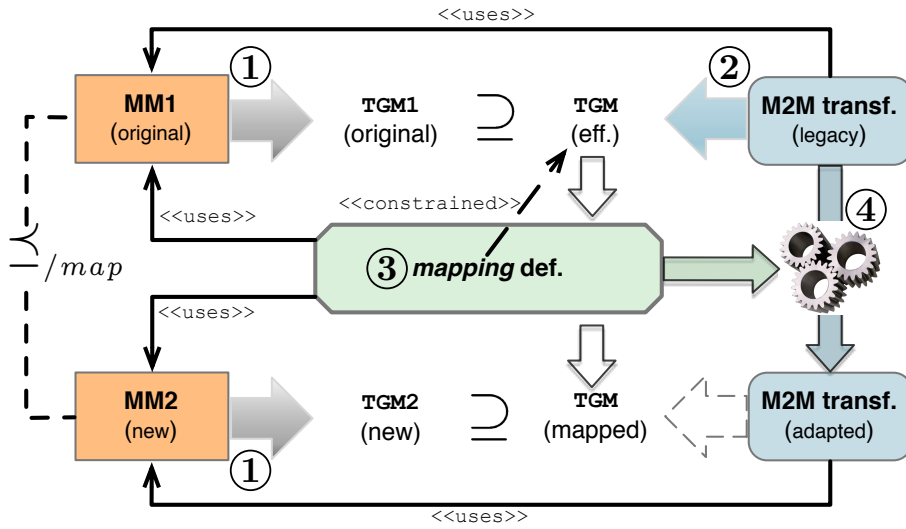


Figure 3.10: Overview of the non-isomorphic transformation reuse approach.

Figure 3.10 illustrates the overview of the non-isomorphic transformation reuse approach. Let TGM1 and TGM2 be the TGMs of respectively non-isomorphic metamodels MM1 and MM2, cf. (1) in Figure 3.10. We expect an implication:

$$TGM1 \subseteq TGM2 \Leftrightarrow MM2 \preceq_{/map} MM1 \quad (3.6)$$

through a mapping between elements of two metamodels MM1, MM2. However, the graph inclusion relation \subseteq (detailed in Section 3.4.1) between TGM1 and TGM2 is not directly checked. Instead, we analyse the legacy transformation to infer a smaller sub-graph of TGM1 – referred to as the type of transformation models (cf. (2) in Figure 3.10), namely $TGM_{\text{eff.}}$. The graph inclusion relation is then verified between $TGM_{\text{eff.}}$ and TGM2 after a mapping step, cf. (3) in Figure 3.10. If the relation is satisfied, we adapt the transformation to achieve a new transformation based on the mapping definition, cf. (4) in Figure 3.10. In respect to this process, the sub-typing relation $\preceq_{/map}$ between metamodels in fact is *partial* since the usage context (i.e. the transformation) of a declared metamodel must be considered. Furthermore, it is

non-isomorphic since it has to be derived through a mapping between metamodels. For this reason, we call our transformation reuse approach *non-isomorphic* transformation reuse. The following sections respectively give details of steps 3 and 4.

3.4.1 DSL for Correspondence Description

As aforementioned in Section 2.3.3, the main obstacle that prevents the reuse of knowledge encoded into an existing model transformation is a phenomenon called *heterogeneity* between alternative metamodels. That is, there exists multiple representation format of concepts in the same modelled domain or a domain might be modelled by similar (or non-identical) concepts. Heterogeneity results to the different abstract syntaxes (structures) expressed by metamodels in domain specific languages of the same specific domain. Since model transformations are described tightly coupled with metamodels (abstract syntax), it is difficult to rebind an existing model transformation to elements of the new metamodel due to the lack of semantics bridge between similar concepts, namely the mechanism, which is often used in defining a transformation is basically the “name-based” binding.

To address these issues we propose a DSL language that is dedicated to explicitly declare the semantic correspondences between alternative (combinations of) metamodel elements. Then, we also provide an interpreter for the DSL which encompasses automatic generation from a legacy transformation to a new adapted one. The main idea of such a DSL is to foster the maintenance of model transformations, despite structural and semantics differences of alternative metamodels. Differences in representation format and in semantics are overcome by means of correspondence patterns that are supported in the language. However, the interpretation semantics of the language must respect to the typing principle that is introduced in Section 3.3. The following paragraphs present the syntax and the semantics of our semantics correspondence description language in detail.

Syntax

To better understand the abstract syntax of the *mapping* DSL language, we recall the abstract syntax of the transformation language considered in this thesis, i.e. MOMENT2-GT language. The structure and the semantics of transformation language MOMENT2-GT has been presented in detail in Section 2.3.2. However, we regroup main constructs of the language and the relation of these constructs to basic constructs of meta-metamodel Ecore in Figure 3.11 in order to give a global view of the language. Note that like most of other transformation languages, MOMENT2-GT does not support directly the notion of model type. Instead, a declared model variable, resp. an instance object of construct TypedModel, has a reference to an EPackage instance, i.e. through the *modelType* reference. Since a package contains all design concepts (including classes, attributes and references), a graph pattern (represented by a recursive structure by means of constructs ObjectTemplateExp, AttributeExp and ReferenceExp) which is defined in a corresponding model variable has to respect the graph topology encoded in design concepts in the package. The

graph topology (represented by the Type Graph with Multiplicity graph structure) of a declared package and the effective used graph topology taking into account the real usage context, i.e. the transformation definition, can be inferred by means of two *typing functions* of our typing system provided in Sections 3.3.2 and 3.3.4.

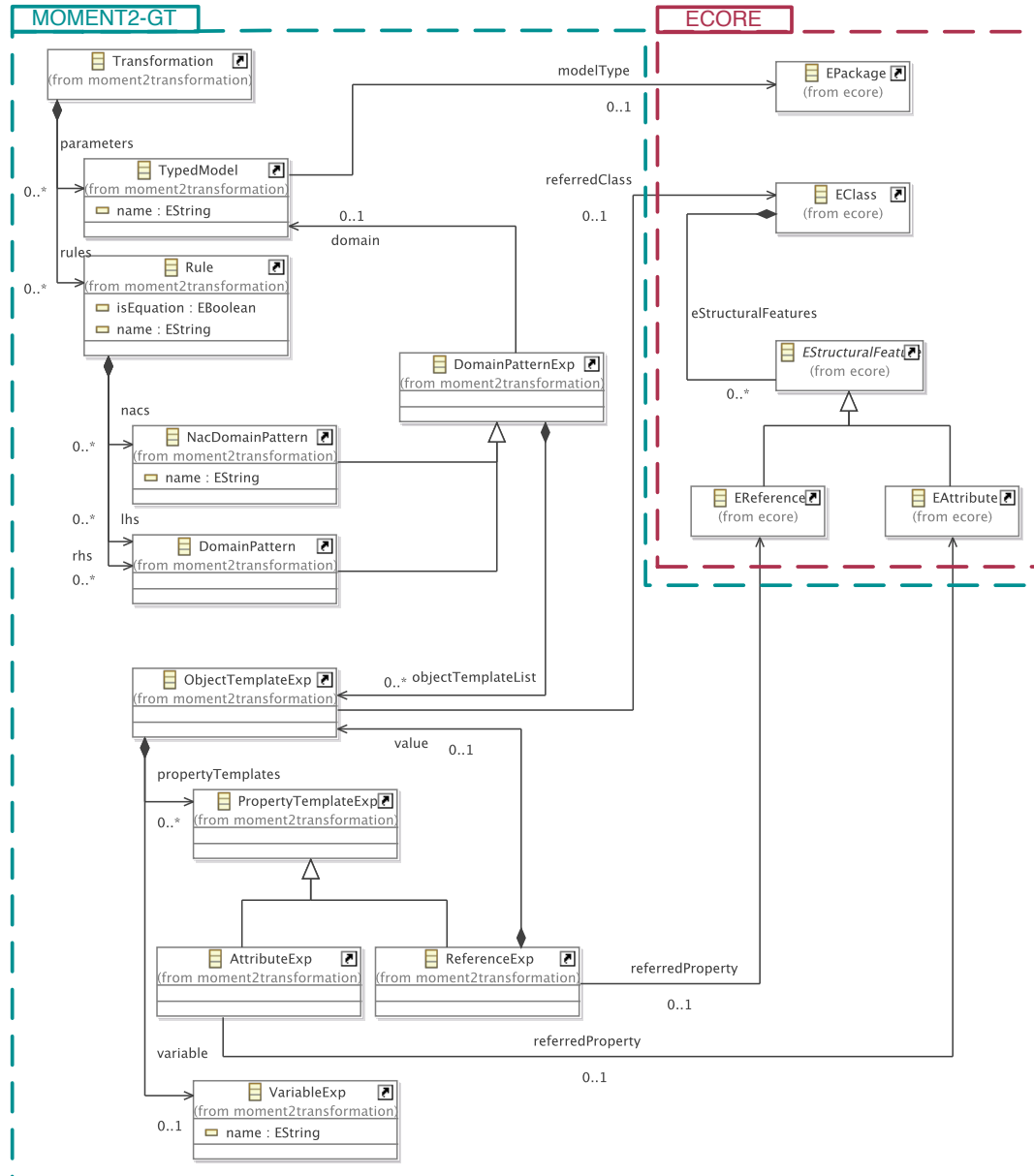


Figure 3.11: Abstract syntax of transformation language MOMENT2-GT.

The abstract syntax of our *mapping* DSL language in fact is an extension of the transformation language MOMENT2-GT. Figure 3.12 shows the metamodel, which represents the abstract syntax of our *mapping* DSL language, namely MetaModMap. As seen in the figure, the MetaModMapRoot construct is defined with the *transformation* reference which refers to the Transformation construct of the abstract syntax

of MOMENT2-GT. This reference definition in fact is used to declare the context usage of a metamodel that is in consideration. A MetaModMapRoot can contains a list of mapping declarations for different metamodels that we want to replace. In addition, a MetaModMapRoot also contains mapping rules corresponding to different mapping declarations.

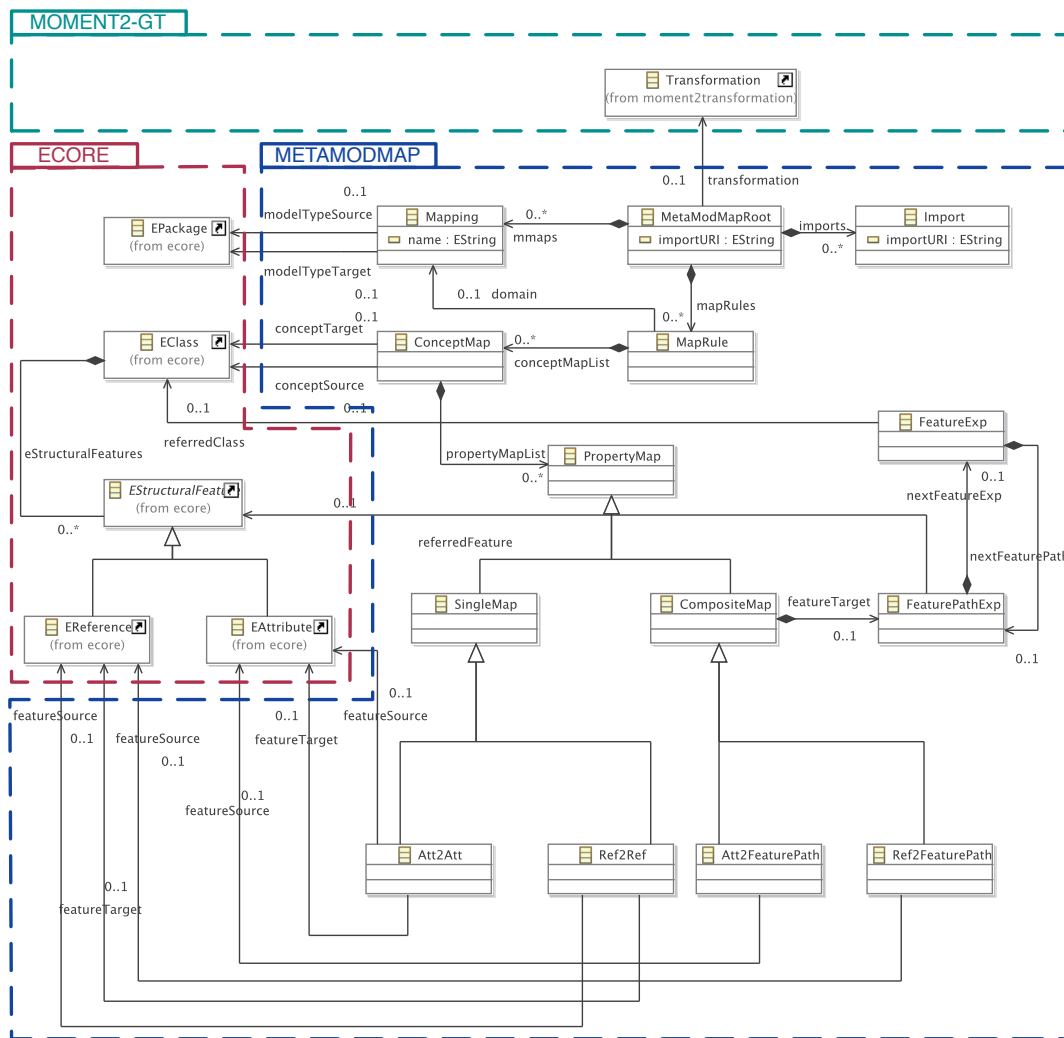


Figure 3.12: Abstract syntax of correspondence language MetaModMap.

A Mapping represents a correspondence declaration between two metamodels in consideration. A mapping declaration references (*modelTypeSource*, *modelTypeTarget*) the metamodels (represented by EPackage from Ecore) between which a mapping rule will be specified in detail. A Mapping is referenced by a specific mapping rule (a MapRule) which consists of ConceptMaps (*conceptMapList*). ConceptMaps define maps between classes (*conceptSource*, *conceptTarget*). A concept mapping can be detailed by specifying PropertyMaps that are categorized in SingleMaps and Compos-

iteMaps. A SingleMap can be either an attribute-to-attribute mapping (Att2Att) or a reference-to-reference mapping (Ref2Ref), which has a mandatory source (*featureSource*) and a mandatory target (*featureTarget*) pointing to an Ecore attribute or an Ecore reference, respectively. A CompositeMap can be either an Att2FeaturePath or a Ref2FeaturePath. Both of Att2FeaturePaths or Ref2FeaturePaths have a mandatory source which refers to an Ecore attribute or an Ecore reference, respectively, however, the target of these mappings is a feature path (represented by FeaturePathExp). FeaturePathExp is a recursive structure that represents a combination of connected elements in a metamodel. More precisely, this data structure plays a role as a wrapper for connected classes, attributes and references. In using such a structure, we can define a property mapping under the form: a graph edge is mapped onto a graph path. As a result, the expressiveness of the *mapping* language is improved and moreover the graph-based typing approach is still respected.

The concrete syntax of the *mapping* DSL language is defined in the XTEXT grammar language, an Extended Backus-Naur Form (EBNF)-like language, in Listings 3.2 and 3.3:

Listing 3.2: MetaModMap XTEXT Grammar : mappings and mapping rules

```

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
2 import "http://www.ac.uk/le/cs/moment2/mt/Moment2transformation" as moment2
MetaModMapRoot:
4   "import" importURI=STRING ";"
   (imports+=Import)+
6   "mapping_for" transformation=[moment2::Transformation]
   "(" mmaps+=Mapping ";" mmaps+=Mapping)* ")"
8   "{"
   (mapRules+=MapRule)+
10  "}"
;
12 Import:
   'import' importURI=STRING ';'
14 ;
Mapping :
16 name=ID ":" modelTypeSource=[ecore::EPackage] "correspondsTo"
   modelTypeTarget=[ecore::EPackage]
18 ;
MapRule :
20 domain=[Mapping]
   "{"
22   (conceptMapList+=ConceptMap ";" conceptMapList+=ConceptMap)*?
   "}"
24 ;

```

Listing 3.3: MetaModMap XTEXT Grammar : concept and property mappings

```

ConceptMap :
2  "concept" conceptSource=[ecore::EClass] "correspondsTo" conceptTarget=[ecore::EClass]
   ("{"
4   propertyMapList+=PropertyMap ("," propertyMapList+=PropertyMap)*
   "}")?
6 ;
PropertyMap :
8  SingleMap | CompositeMap
;
10 SingleMap:
   Att2Att | Ref2Ref
12 ;
   CompositeMap:
14  Att2FeaturePath | Ref2FeaturePath
;
16 Att2Att:
   "att2att" featureSource = [ecore::EAttribute] "correspondsTo"
18   featureTarget = [ecore::EAttribute]
;
20 Ref2Ref:
   "ref2ref" featureSource = [ecore::EReference] "correspondsTo"
22   featureTarget = [ecore::EReference]
;
24 Att2FeaturePath:
   "att_reification" featureSource = [ecore::EAttribute] "correspondsTo"
26   featureTarget = FeaturePathExp
;
28 Ref2FeaturePath:
   "ref_reification" featureSource = [ecore::EReference] "correspondsTo"
30   featureTarget = FeaturePathExp
;
32 FeaturePathExp:
   referredFeature = [ecore::EStructuralFeature] ("." nextFeatureExp = FeatureExp)?
34 ;
   FeatureExp :
36  referredClass= [ecore::EClass] "." nextFeaturePath = FeaturePathExp
;

```

The default nature of XTEXT is to begin with a grammar definition, then uses this to generate an Ecore-defined metamodel, a corresponding ANTLR-based parser and an Eclipse-based text editor. In a grammar definition, XTEXT allows to reference existing metamodels (in our grammar that are the Ecore metamodel and MOMENT2-GT metamodel) by using an import mechanism. Based on this feature, the generated text editor supports the importation of existing artifacts of which elements can be

referred (or bound). In using the text editor generated by XTEXT, a *mapping* definition can be input by a user. The user defines semantics correspondence between metamodel elements by referencing directly to them in the imported artifacts. In this case, these artifacts are the transformation and metamodels in consideration.

Example

Applying the concrete syntax of the *mapping* DSL language to the motivating case study in this thesis, see details in Section 2.3.3 at page 51 for the case study, we can now specify the semantic correspondences between the metamodels CDV1 and CDV2 (cf. Listing 3.4).

Listing 3.4: MetaModMap definition: CDV1 vs. CDV2

```

1 import "platform:/resource/CD2RDB/MTs/CD2RDB.mgt" ;
2 import "platform:/resource/CD2RDB/MMs/CDV2.ecore" ;
3
4 mapping for CD2RDB ( CDV1ToCDV2 : CDV1 correspondsTo CDV2)
5 {
6   /* Mapping rule from metamodel CDV1 onto metamodel CDV2 */
7   CDV1ToCDV2 {
8     concept PrimitiveDataType correspondsTo DataType; /* (1)*/
9     concept Class correspondsTo Class {
10      att2att isPersistent correspondsTo persistent,
11      ref2ref ownedFeatures correspondsTo ownedProperties, /* (3)*/
12      ref_reification parent correspondsTo generalization . Generalization . general /* (2)*/
13    };
14    concept Attribute correspondsTo Property {
15      att2att isPrimary correspondsTo primary /* (4)*/
16    };
17    concept Association correspondsTo Association {
18      ref2ref source correspondsTo from,
19      ref2ref target correspondsTo to
20    }
21  }
22  /* Define other mapping rules for other pairs of metamodels, if any */
23  ...
24 }

```

As seen in Listing 3.4, a *mapping* definition between two metamodels declares only correspondences between different elements in two metamodels in consideration. These declarations are declarative and stated locally. That is, from a type concept in the original metamodel that has a different (but similar) semantics with a certain type concept in the replacement metamodel, we make a correspondence between them. Then, we go inside details of that type concept, i.e. property concepts nested in that type concept, to make correspondences for their details, if necessary. For the motivating example, only the following kinds of correspondence pattern are needed:

three related to intuitive “renaming”, cf. (1) (3) (4), and one to relation (reference) reification, cf. (2).

More precisely, the type concept `PrimitiveDataType` of metamodel CDV1 that is actually used in transformation CD2RDB is mapped onto type concept `DataType` of metamodel CDV2. For the type concept `Class`, due to the difference in properties of that type concept in two metamodels, so two entities that are named `Class` may not have the same semantics. Actually, the semantics of an entity is not only dependent on its own semantics, but also on semantics of their properties. In this case, we chose to make a correspondence between two entities even they have the same name in two metamodels, then make correspondences between their different properties, if any. Simple cases for mapping different properties between two entities `Class` are: the *isPersistent* attribute of the first entity is mapped onto the *persistent* attribute of the second entity; and the *ownedFeatures* reference is mapped onto the *ownedProperties* reference. Similarly, we do correspondences for the pairs `Attribute/Property` and `Association/Association`.

In order to support more complex cases, such as attribute reification or reference reification, we provide two composite correspondence patterns that are from an attribute to a feature path and from a reference to a feature path, represented by the `Att2FeaturePath` and `Ref2FeaturePath` constructs, respectively.

To demonstrate the use of the `Att2FeaturePath` correspondence pattern, see Figure 3.13 in which the *author* attribute of class `Book` in the first metamodel in fact represents the same semantics as one of the combination of the *writer* reference, the `Author` class and the *name* attribute of such class in the second metamodel. For this case, the semantic equivalence between them can be expressed by the provided `Att2FeaturePath` correspondence pattern (cf. Listing 3.5).

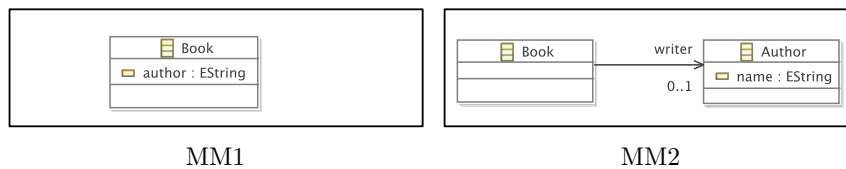


Figure 3.13: Attribute reification example.

Listing 3.5: Mapping illustration for attribute reification.

```

1 ...
2 concept Book correspondsTo Book {
3   att_reification author correspondsTo writer . Author . name
4 }
5 ...

```

There is not any similar case as the example given above in the motivating case study, instead, we have a situation that needs to use the `Ref2FeaturePath` correspondence pattern. See Figure 3.14 which extracts the interesting metamodel fragments of two metamodels CDV1 and CDV2 that present similar semantics. More precisely, the *parent* reference of class `Class` in metamodel CDV1 represents the same semantics

as one of the combination of the *generalization* composite reference, the Generalization class and the *general* reference of such class in metamodel CDV2. In fact, this semantic correspondence can be represented explicitly in making the use of the Ref2FeaturePath correspondence pattern as in Listing 3.6.

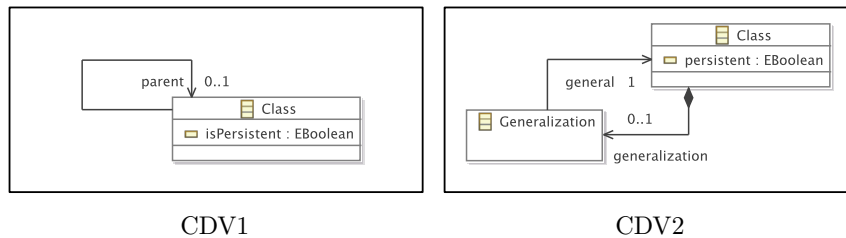


Figure 3.14: Reference reification example.

Listing 3.6: Mapping illustration for reference reification.

```

1 ...
2 concept Class correspondsTo Class {
3   ...
4   ref_reification parent correspondsTo generalization . Generalization . general
5 }
6 ...

```

In the graph-theoretic sense, semantic correspondence descriptions as those in Listing 3.4 support the identification of a morphism between two labeled graphs, note that we consider a metamodel as a graph by means of the TGM graph structure. More precisely, this morphism is between an effective TGM graph of an input source metamodel and the graph representation of the new source metamodel.

The adjusted model transformation that is generated by interpreting the *mapping* definition (cf. Listing 3.4), taking into account the complete model transformation of the motivating case study (cf. Appendix 1.1) as the subject to be manipulated, can be found in Appendix 1.2.

Semantics

In order to work with the mapping DSL language, it is necessary to distinguish the semantics of the language from two different levels. The first level considers alternative metamodels as existing inputs and declares semantic correspondences between their elements. At this level, these declarations are defined in a declarative manner and the whole mapping definition is checked based on the graph typing approach, which is proposed in previous sections. We call this *model type compatibility check* semantics. The second level considers the existing transformation as the subject to be manipulated. That is, based on semantic correspondence declarations at the metamodel level, a modification engine will interpret the whole mapping definition to rebind elements of the transformation model to elements of the new metamodel. We call this *transformation adaptation* semantics of the language.

Another imagination to work with the mapping language is to consider the *model*

type compatibility check semantics as those implemented in an *a priori* type checking utility of advanced editors for programming languages, and the *transformation adaptation* semantics as those implemented in their compiler. The main difference in comparison with programming languages is users do not need to consider the operational semantics of the mapping language, but only establishing semantic bridges between two alternative metamodels' elements. The advanced editor for the language will give an early feedback validation to user. This feedback indicates that the mapping is possible or not, certainly, depending on the operational semantics provided by the language. This utility is performed by a type system that implements the *model type compatibility check* semantics.

The following part of this section will focus on the *model type compatibility check* semantics and how it respects our graph-based model typing approach. The *transformation adaptation* semantics of the mapping language will be detailed in Section 3.4.2

Model type compatibility check semantics. We distinguish two kinds of check semantics: (1) *scoping* check semantics and (2) *validation* check semantics. The *scoping* check semantics relates to the use of an existing element in a certain context. For example, in the most simple case, when defining a correspondence between type concepts of two alternative metamodels, the left-hand side of the correspondence definition can only refer to a certain type concept that is defined in the first metamodel, and so on the right-hand side must point to a certain type concept that is in the second metamodel. The *validation* check semantics relates to the semantics of the compiler that are usually more complex than the *scoping* check semantics. Our type system is an example, before interpreting the whole *mapping* definition, it is essential to infer the effective model types of the original metamodels, and perform other additional tasks.

Scoping check semantics As described above, correspondences are described at the metamodel level, i.e. making the use of elements in already existing Ecore-defined metamodels. However, with respect to the *effective* TGM model type inference when taking into account a real usage context, i.e. the existing transformation that we want to reuse, these descriptions are imposed by some additional scoping constraints to ensure the compatibility of mapped elements in their left-hand side parts. Concretely,

- Only type concepts that are used in the transformation, i.e. having a corresponding GNode in the *effective* TGM model type, can be mapped to a certain type concept defined in the new metamodel. In the motivating case study, type concept Operation cannot be used in any correspondence definition.
- For both single and composite PropertyMaps, attribute concepts and reference concepts have to be owned (taking into account inheritance) by the type concept referred by the nesting type concept correspondence. In addition, they must have at least a corresponding GEdge in the *effective* TGM model type.

The right-hand side parts of correspondence definitions have less scoping constraints than their left-hand side parts. Precisely,

- The condition to use a type concept in the new metamodel to define a type concept correspondence, i.e. a `ConceptMap`, is less constrained than the usage condition of the original metamodel’s type concepts. That is any type concepts defined in the new metamodel can be used in the right-hand side of a `ConceptMap` as long as it has not been used in other correspondence definitions.
- For single `PropertyMaps`, similar to the left-hand side parts, attribute concepts and reference concepts have to be owned (including transitive inheritance relation) by the type concept referred by the right-hand side of the nesting type concept correspondence.
- Composite `PropertyMaps` require more complex scoping constraints. More precisely, in the root `FeaturePathExp` object, which is owned by a `CompositeMap` (*featureTarget*), the referred feature (*referredFeature*) must always be a reference concept of a mapped type concept. The type concept that is referred by the next feature expression (`FeatureExp`) has to be the type of the referred feature or a sub type concept of that type. In addition, the last referred feature in a `Att2FeaturePath` must be an attribute concept, and in contrast the last referred feature in a `Ref2FeaturePath` has to be a reference concept.

These above *scoping* checking semantics are defined precisely. Thus, they can be implemented to support the scoping aspect of an advanced editor for the mapping DSL language. The scoping aspect is a very important feature in most of modern editors since we can provide more intelligent suggestions about what users can do in the next steps.

Validation check semantics This kind of checking semantics is more tightly coupled with our model typing approach that is based on the notion of graph similarity. We recall that according to Section 3.3, we have proposed the TGM graph structure as the notion of model type. In addition, based on the name of graph nodes and graph edges, we defined a sub-graph relation between TGMs in taking *multiplicity* values on graph edges into account. This relation can be formalized concisely as follows:

TGM sub-graph \subseteq : $G'(N', E') \subseteq G(N, E)$ when

- **Node:** $\forall n' \in N', \exists n \in N (n'.isMatch(n))$
- **Edge:** $\forall e' \in E', \exists e \in E ($
 $e'.[from/to].isMatch(e.[from/to]) \wedge$
 $e'.name = e.name \wedge$
 $e'.multiplicity.isMatch(e.multiplicity))$

Multiplicity order: $? \preceq 1 \preceq *$

where, *undefined_valued* is matched to both *single_valued* and *multi_valued*; and *single_valued* is matched to *multi_valued* (cf. formula 3.2).

Based on above definitions, the *model type compatibility* validation check seman-

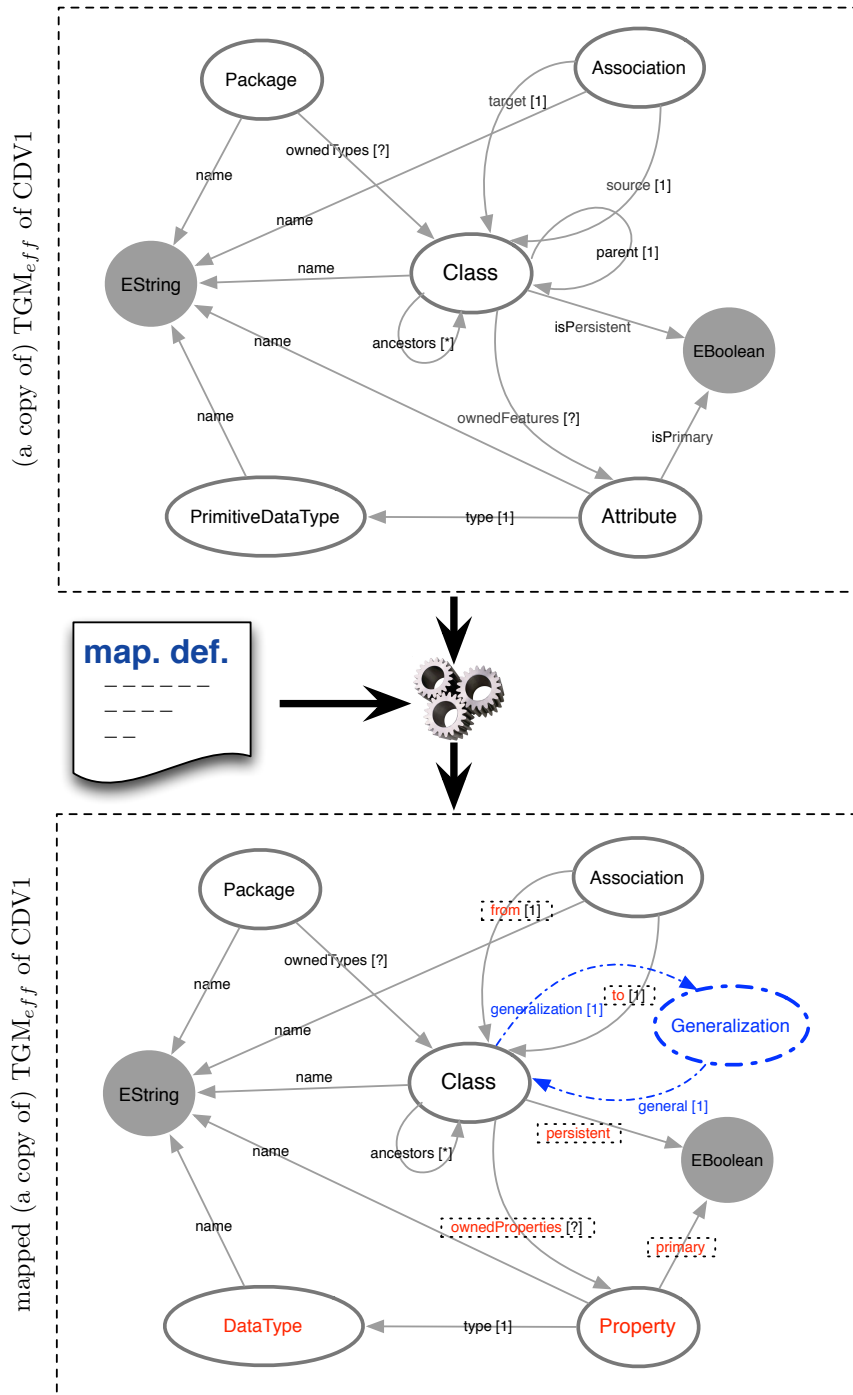


Figure 3.15: From *effective* TGM to *mapped* effective TGM.

tics is performed at a coarse-grained level, i.e. between two alternative metamodels. This task can be carried out by taking the (copy of the) *effective* TGM model type corresponding to the original metamodel, then trying to rename graph nodes and graph edges according to the *mapping* definition. In case of composite correspondences, we need to perform renaming and create additional edges and nodes based on the elements of the new metamodel, simultaneously. The final step is verifying the TGM sub-graph relation between the *mapped* effective TGM model type, i.e. the output from the previous step, with the TGM model type corresponding to the new metamodel. Once this relation is validated, we can say the new metamodel is a sub-type of the original metamodel in taking the transformation into account as following formula 3.3 in Section 3.3.3. Otherwise, the *mapping* definition is considered as a non-validated definition. That means though a user-defined semantics mapping, two metamodels are still non-isomorphic. Thus, the mapping cannot be passed to the interpreter that implements *transformation adaptation* semantics for migrating the legacy transformation.

Figure 3.15 illustrates the inputs and the output of the *mapping* action semantics in the TGM model type space for the motivating case study CD2RDB. The action semantics has two input: (a) The *effective* TGM model type corresponding to metamodel CDV1 when taking the legacy transformation definition into account, cf. at the top of the figure; (b) The *mapping* definition which defines correspondence between elements of the original metamodel CDV1 and the new metamodel CDV2, cf. in the middle of the figure. The output is the *mapped* effective TGM model type according to the *mapping* definition. Note that in the output TGM, labels (i.e. the *name* attribute of TGM graph elements) of modified graph nodes and graph edges are represented as **red** labels and new-created graph nodes and edges are represented in **blue** dotted-lines.

After the mapping according to the correspondence descriptions, all nodes and edges of an *effective* TGM can find their mirror in the TGM model type of the new metamodel. The sub-graph relation is verified between the *mapped* effective TGM, cf. Figure 3.15 at the bottom, and the TGM of CDV2, cf. Figure 3.5 at page 82 in Section 3.3.2. This relation is used as an underlying signature to validate a MetaModMap correspondence definition for the substitution possibility between two metamodels (cf. formula 3.3 at page 86). All these constraints will be implemented in the *scoping and validating aspects* of the MetaModMap editor, which provides an interactive warning-suggestion system to users.

As a conclusion, we can say the distance between two metamodels is close with respect to a given mapping language when it is possible to exhibit a mapping between these two metamodels. Listing 3.4 is an example of such a mapping showing that CDV1 and CDV2 are two close metamodels w.r.t. MetaModMap.

3.4.2 Transformation Adaptation

Given a valid MetaModMap correspondence definition and a transformation written in MOMENT2-GT, an adaptation interpreter is in charge of rebinding model

patterns referring to elements of the original metamodel with those of the new one, then textualizing the whole rebound transformation model into a new transformation file. This process is possible and safe because through a valid user-defined semantics mapping, two metamodels become isomorphic. Thus, the mapping can be passed to the interpreter that implements *transformation adaptation* semantics for migrating the legacy transformation.

Algorithm Overview

Figure 3.16 illustrates the overview of transformation adaptation mechanism. The inputs of the algorithm are: (a) A *mapping* definition between two metamodels that uses provided correspondence patterns in the MetaModMap language; (b) An original transformation (already declared in the *mapping* definition) that refers to the original metamodel. The output of the algorithm is an adapted transformation that can be safely used to transform models which conform to the new metamodel.

The whole *transformation adaptation* semantics is performed in five main phases: (1) Parsing the textual representation of the *mapping* definition in order to obtain its in-memory representation, also referred as the *mapping* model; (2) Similarly, parsing the textual representation of the legacy transformation definition to achieve the transformation model; (3) Performing a rebinding process on graph pattern models of the transformation model that relate to semantics-changed elements according to the *mapping* model; (4) rebinding graph pattern models for remaining unchanged elements based on names to reference elements in the new metamodel model; (5) serializing the changed transformation model into the textual representation of the transformation language. Three phases (1) and (2), (5) are realized in making use of the text parsers of the mapping language and the transformation language, respectively. The adaptation semantics of the algorithm mainly focuses on phases (3) and (4) which manipulate the transformation at model level. The followings will present phases (3) and (4) of the adaptation algorithm which do a rebinding action on graph pattern models of the transformation model.

Algorithm

The transformation model *rebinding* algorithm (shown in Algorithm 5) has two inputs: (a) the map rule *mapRule* which contains details of correspondence definitions between elements of two metamodels; (b) the transformation model *mt* which will be rebound to the new metamodel. The output of the algorithm is the rebound transformation model, i.e. all references to the original metamodel after the *rebinding* process are forwarded to the elements of the new metamodels.

We briefly go through the steps of the algorithm. In the initial step, cf. lines 1–4, we get the original metamodel and the new metamodel from the mapping declaration of the mapping rule. Then, all top object template expressions (OTE) of the transformation model that are constrained by the original metamodel are collected into a list, i.e. $OTES_{top}$. The remaining of the algorithm is divided into two main steps: (1) The first step corresponds to phase 3 of the *transformation adaptation* semantics

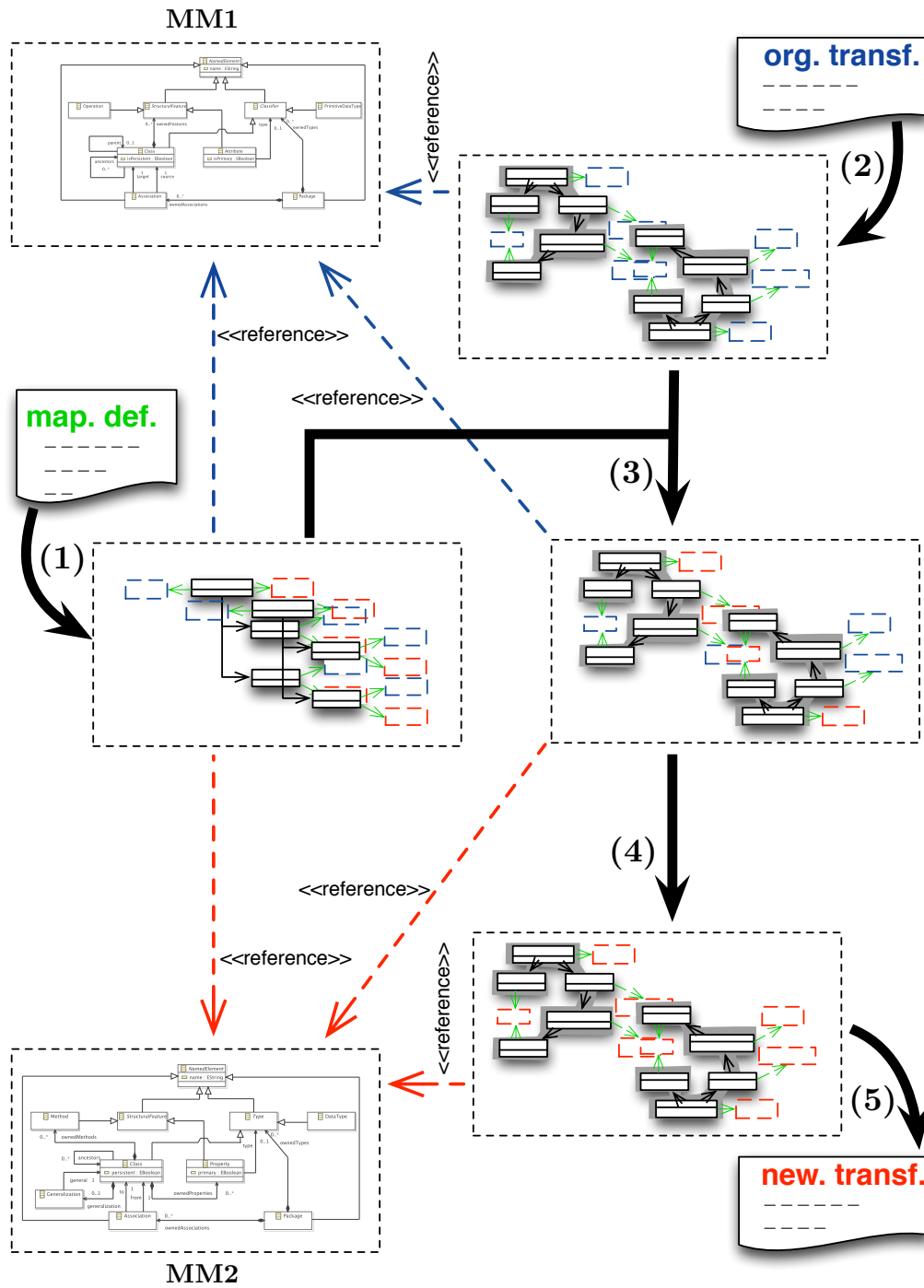


Figure 3.16: Overview of transformation adaptation algorithm.

Algorithm 5 *doRebind*(*mapRule*, *mt*)

Require: (1) the map rule *mapRule* which contains correspondences between elements of two metamodels. (2) the transformation model *mt* which will be rebound to the new metamodel.

Ensure: Return the rebound transformation model.

```

1: mapDcl ← mapRule.domain { Get the mapping declaration of mapRule }
2: mmSrc ← mapDcl.modelTypeSource { Get root package of original metamodel }
3: mmTgt ← mapDcl.modelTypeTarget { Get root package of new metamodel }
4: OTEStop ← collectTopOTES(mt,mmSrc) { Collect top OTEs constrained by mmSrc }
   { Phase 3: Rebind for semantic-changed elements }
5: for all conceptMap in mapRule.CONCEPTMAPS do
6:   for all ote in OTEStop do
7:     rebindChangedElem(conceptMap,ote) { Visit recursively beginning at the top object
      template expression ote, cf. Algorithm 6. }
8:   end for
9: end for
   { Phase 4: Rebind for remaining unchanged elements }
10: for all ote in OTEStop do
11:   rebindUnchangedElem(ote,mmSrc,mmTgt) { Visit recursively beginning at the top ob-
      ject template expression ote, cf. Algorithm 7. }
12: end for

```

shown in Figure 3.16. This step is charged of rebinding object template expressions according to the mapping rule; (2) The second step corresponds to phase 4 of the *transformation adaptation* semantics that does rebinding action mainly based on the names of metamodel's elements for the remaining unprocessed parts of OTEs.

The first step of the algorithm, cf. lines 5–9, involves the rebinding of references in object template expressions that relate to the changed elements between two metamodels. To begin with, for each mapping between two type concepts, we browse all top object template expressions to look up elements that are in consideration by the mapping. Each pair (*conceptMap*,*ote*) is passed into the *rebindChangeElem* procedure. The *rebindChangeElem* procedure acts as a visitor that traverses recursively the entire graph pattern defined by the top object template expression. Algorithm 6 that details the mechanism of that procedure on rebinding changed elements will be explained later.

In the second step of the algorithm, cf. lines 10–12, we perform the rebinding for the remaining unchanged elements. The inputs for this step are graph patterns that have been rebound partially after the first step. Beginning with the top object template expressions of these graph patterns, we re-visit them through the *rebindUnchangeElem* procedure. Similarly to the *rebindChangeElem* procedure, this procedure traverses the entire partial-rebound graph pattern to rebind remaining elements. The semantics of this process will be detailed in Algorithm 7 in the next paragraphs.

Now we go in-depth on Algorithm 6 that describes the *rebindChangeElem* procedure used in the first step of Algorithm 5. This procedure is the kernel of our

Algorithm 6 *rebindChangedElem(conceptMap, ote)*

Require: (1) the concept mapping *conceptMap* which contains the correspondences between two type concepts. (2) the object template expression *ote*.

Ensure: Return the rebound object template expression according to the mapping.

```

1: {Perform lookup at the being-visited object template expression.}
2: if conceptMap.conceptSource.equal(ote.referredClass) then
3:   ote.referredClass  $\leftarrow$  conceptMap.conceptTarget
4:   for all pte in ote.PROPTEMPEXPS do
5:     for all propertyMap in conceptMap.PROPERTYMAPS do
6:       if propertyMap.featureSource.equal(pte.referredProperty) then
7:         if pte instanceof AttributeExp then
8:           if propertyMap instanceof Att2Att then {Single attribute map}
9:             pte.referredProperty  $\leftarrow$  propertyMap.featureTarget
10:          else if propertyMap instanceof Att2FeaturePath then {Attribute reification map}
11:            topFeaturePathExp  $\leftarrow$  propertyMap.featureTarget
12:            newPTE  $\leftarrow$  createGraphPattern(topFeaturePathExp, pte)
13:            ote.PROPTEMPEXPS  $\leftarrow$  ote.PROPTEMPEXPS  $\setminus$  pte
14:            ote.PROPTEMPEXPS  $\leftarrow$  ote.PROPTEMPEXPS  $\cup$  newPTE
15:          end if
16:        else if pte instanceof ReferenceExp then
17:          if propertyMap instanceof Ref2Ref then {Single reference map}
18:            pte.referredProperty  $\leftarrow$  propertyMap.featureTarget
19:          else if propertyMap instanceof Ref2FeaturePath then {Reference reification map}
20:            topFeaturePathExp  $\leftarrow$  propertyMap.featureTarget
21:            newPTE  $\leftarrow$  createGraphPattern(topFeaturePathExp, pte)
22:            ote.PROPTEMPEXPS  $\leftarrow$  ote.PROPTEMPEXPS  $\setminus$  pte
23:            ote.PROPTEMPEXPS  $\leftarrow$  ote.PROPTEMPEXPS  $\cup$  newPTE
24:          end if
25:        end if
26:      end for
27:    end for
28:  end if
29:  {call recursively for lower object template expressions}
30:  for all pte in ote.PROPTEMPEXPS do
31:    if pte instanceof ReferenceExp then
32:      rebindChangedElem(conceptMap, pte.value)
33:    end if
34:  end for

```

rebinding approach that implements a semantic to process changed elements between two metamodels.

The inputs of procedure *rebindChangeElem* are: (a) A concept mapping *conceptMap* which defines the correspondences between two type concepts; (b) An object template expression *ote* at some level in the being-processed graph pattern. The algorithm is designed as a depth-first search process. That is if the being-visited object template expression refers to the same type concept as the source type concept used in the concept mapping definition, we will perform rebinding at that level of the graph pattern, cf. lines 1–28. Then, we invoke this procedure itself for object template expressions at the lower level through reference template expressions (ReferenceExps) of the already-visited object template expression *ote*, cf. lines 29–33.

The output at each level of the visiting process is an object template expression, which is (partially) rebound according to the concept mapping, if any. The process is divided in two phases. First, once having a match between the referred class (*referredClass*) of the being-visited object template expression *ote* and the source type concept (*conceptSource*) of the concept mapping (ConceptMap or CM) *conceptMap*, the reference *referredClass* of that object template expression will be forwarded to the target type concept (*conceptTarget*) of the concept mapping, cf. line 2 and Figure 3.17 for an illustration example.

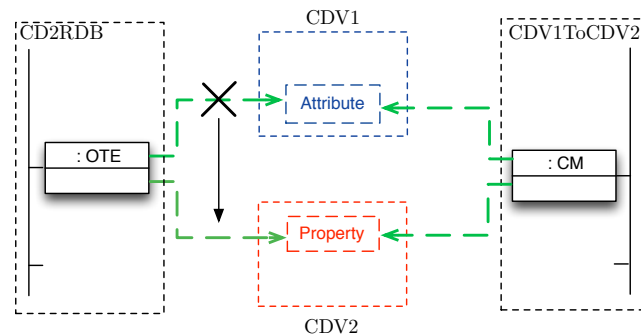


Figure 3.17: Rebinding example for a type concept mapping.

Further, the rebinding is performed for property template expressions (PTEs) of *ote*, cf. lines 3–27. Similarly, we look up a match between the referred property (*referredProperty*) of a PTE declared in the being-visited object template expression and the source feature (*featureSource*) of a property mapping (PropertyMap) declared in the concept mapping, cf. lines 3–5. If there is a match, the rebinding process for PTEs will be treated as the following cases:

The property mapping is a Att2Att mapping. This is the case that defines a single mapping for two attribute concepts nested in two type concepts of two metamodels. This case is processed after rebinding for its type concept mapping container. Having a look at the mapping definition in Listing 3.4 (see page 103) from lines 14–16, the type concept mapping between type concept Attribute of metamodel CDV1 and type concept Property of metamodel CDV2 contains an attribute concept mapping between attribute *isPrimary* and attribute *primary*. After rebinding-

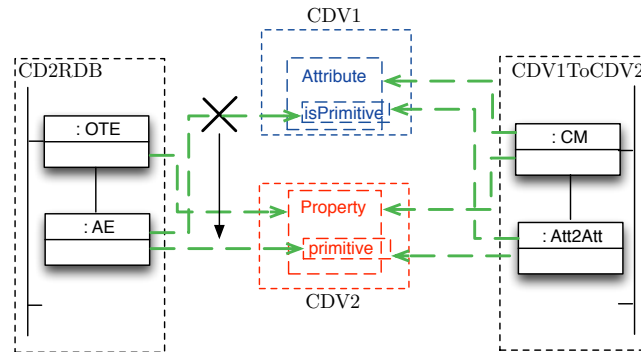


Figure 3.18: Rebinding example for a single attribute mapping.

ing for the pair (Attribute,Property) on the considering object template expression, reference *referredProperty* of the interesting attribute expression (AttributeExp or AE) is switched from *isPrimary* to *primary* according to the mapping definition (cf. Figure 3.18 for the illustration).

The property mapping is a Ref2Ref mapping. Similar to single mapping for attribute concepts, this case presents a semantics equivalence between two reference concepts.

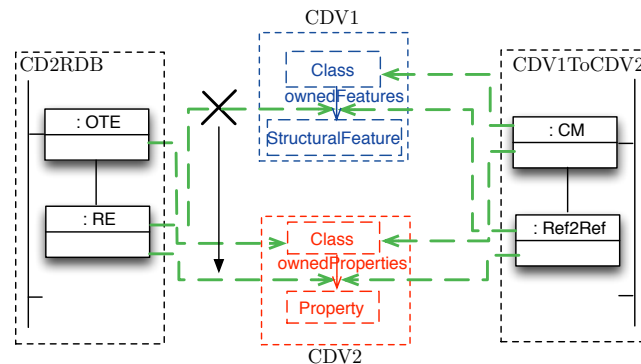


Figure 3.19: Rebinding example for a single reference mapping.

Having a look at the mapping definition in Listing 3.4 (cf. page 103) at line 12, the *ownedFeatures* reference concept of type concept Class in metamodel CDV1 is declared as equivalent to the *ownedProperties* of the same type concept Class in metamodel CDV2. For this case, we preserve the current reference expression (RE) and simply remove the link (*referredProperty*) referring to reference *ownedFeatures* then re-establish this link to reference *ownedProperties*, see Figure 3.19 for the illustration of this case.

The property mapping is a Att2FeaturePath mapping. This is the case that defines a composite mapping for an attribute in the original metamodel that is reified in several elements in the new metamodel.

Similarly, it is processed after rebinding for its type concept mapping container.

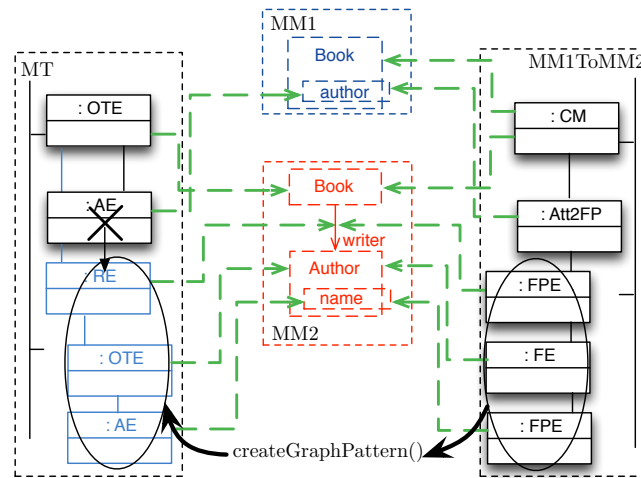


Figure 3.20: Rebinding example for a composite attribute mapping.

Having a look at the example in Figure 3.13 and a fragment of mapping in Listing 3.5, the *author* reference in metamodel MM1 is considered as an equivalence of the combination of elements, i.e. *writer.Author.name*, in metamodel MM2. The declaration of a combination of elements is supported by a recursive structure by means of constructs FeaturePathExp (FPE) and FeatureExp (FE) of our DSL language. As aforementioned, this data structure plays a role as a wrapper for connected classes, attribute and references, however, in the form of graph paths. As a result, the above mapping is considered as a mapping from a graph edge onto a graph path in our notion of model type. To begin with, based on the definition declared in the right-hand side of the property mapping, we create a graph pattern (more precisely, a graph path) under the representation form of ReferenceExp (ER), ObjectTemplateExp (OTE) and AttributeExp (AE) by means of auxiliary function *createGraphPattern()*. Then, the original attribute expression is removed from its container to be replaced by the new graph pattern. Note that when creating the new graph pattern, we preserve the *value* property of the original attribute expression into the last attribute expression of the new graph pattern. Figure 3.20 illustrates an example of rebinding for attribute reification.

The property mapping is a Ref2FeaturePath mapping. This case often occurs when one needs to detail a relationship between two type concepts. This is performed by introducing one or more type concepts replacing a reference, however, the connectivity between two type concepts in consideration is always preserved. That is an indirect and transitive connection.

For the motivating case study, we have defined such a mapping to describe a semantics equivalence between the *parent* reference in metamodel CDV1 and a combination of elements, i.e. *generalization.Generalization.general*, in metamodel CDV2. Similar to the case of attribute reification, the recursive structure based on constructs FeaturePathExp and FeatureExp is used in the right-hand side of the correspondence pattern Ref2FeaturePath to declare this equivalence. From the provided information,

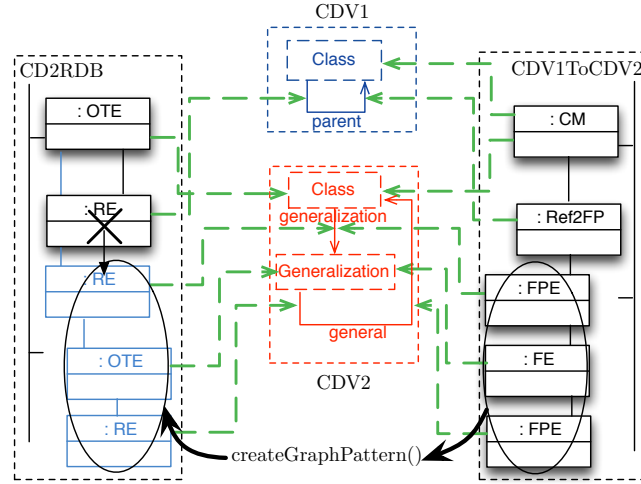


Figure 3.21: Rebinding example for a composite reference mapping.

we create a graph path under the representation form of language MOMENT2-GT by means of the *createGraphPattern*. In the end, the original reference expression is removed from its container and replaced by the new graph path. Note that when creating the new graph path, we preserve the *value* property, i.e. the lower object template expression, of the original reference expression into the last reference expression of the new graph path. Figure 3.21 illustrates an example of rebinding for reference reification.

Once having performed the rebinding for all changed elements by means of Algorithm 6, the rebinding process will proceed for unchanged elements by Algorithm 7. This algorithm acts as a visitor on a graph pattern which has been partially rebound to the new metamodel and does the rebinding for the remaining parts of the graph pattern. Details of the algorithm are described in the following.

The input of the *rebindUnchangedElem* procedure is composed of: (a) a partial-rebound graph pattern, i.e. an object template expression *ote* at some level that has been processed by procedure *rebindChangeElem*; (b) the root package of the original metamodel *mmSrc*; (c) the root package of the new metamodel *mmTgt*. The output at each level of the visiting process is an object template expression, which is (totally) rebound to elements of the new metamodel *mmTgt*.

A depth-first search algorithm is employed to implement the procedure. This algorithm is separated in two main phases: (1) performing rebinding for the being-visited object template expression; (2) rebinding for property template expressions contained in the object template expression.

In the first phase of the algorithm, cf. lines 1–4, we first verify whether the referred class (via *referredClass*) of the begin-visited object template expression is an entity that is nested in the root package of the original metamodel or not by means of pseudo binary operator **contain**. This operator returns whether the second object (operand) is directly or indirectly contained by the first object, i.e., whether

Algorithm 7 *rebindUnchangedElem(ote, mmSrc, mmTgt)*

Require: (1) the object template expression *ote*. (2) the root package of original metamodel *mmSrc*. (3) the root package of new metamodel *mmTgt*.

Ensure: Return the rebound object template expression.

```

  { Visit and rebind at the being-visited object template expression. }
1: referredClass ← ote.referredClass
2: if mmSrc contain referredClass then {the referred class is still not rebound}
3:   ote.referredClass ← mmTgt.getType(referredClass.name) {Get class by name}
4: end if

5: for all pte in ote.PROPTEMPEXPS do
6:   if pte instanceof AttributeExp then
7:     referredAtt ← pte.referredProperty
8:     if mmSrc contain referredAtt then {referred att. is still not rebound}
9:       attTgt ← mmTgt.getFeature(referredClass.name, referredAtt.name)
10:      pte.referredProperty ← attTgt
11:    end if

12:   else if pte instanceof ReferenceExp then
13:     referredRef ← pte.referredProperty
14:     if mmSrc contain referredRef then {referred ref. is still not rebound}
15:       refTgt ← mmTgt.getFeature(referredClass.name, referredRef.name)
16:       pte.referredProperty ← refTgt
17:     end if

    { call recursively for lower object template expressions. }
18:     lowerOTE ← pte.value
19:     rebindUnchangedElem(lowerOTE, mmSrc, mmTgt)
20:   end if

21: end for

```

the second object is in the content tree of the first. If the referred class is contained in the root package of the original metamodel, i.e. the *referredClass* reference is still not rebound, thus we look up a class contained in the new metamodel that has the same *name* as the original class and assign the found class to the value of reference *referredClass*. The look-up functionality is realized by means of auxiliary function *getType()* of packages.

In the second phase of the algorithm, cf. lines 5–20, the rebinding is proceeded for property template expressions of *ote*. Similar to the processing in the first step, for each property template expression (both an *AttributeExp* and a *ReferenceExp*), the referred property (via *referredProperty*) is verified whether it is an entity in the root package of the original metamodel or not. In case of un-rebound reference, we do a look-up for a property having the same name, but in the new metamodel and contained in the class with the same name of the original class (by means of auxiliary function *getFeature()* of packages). Next, the found property is assigned for the value of *referredProperty* of the being-browsed property template expression. The visiting process proceeds by invoking the procedure itself with a lower object tem-

plate expression when the being-browsed property template expression is a reference expression (ReferenceExp).

Application Example

Figure 3.22 shows the application of transformation adaptation semantics to (a part of) graph patterns which are defined in the transformation of the case study CD2RDB.

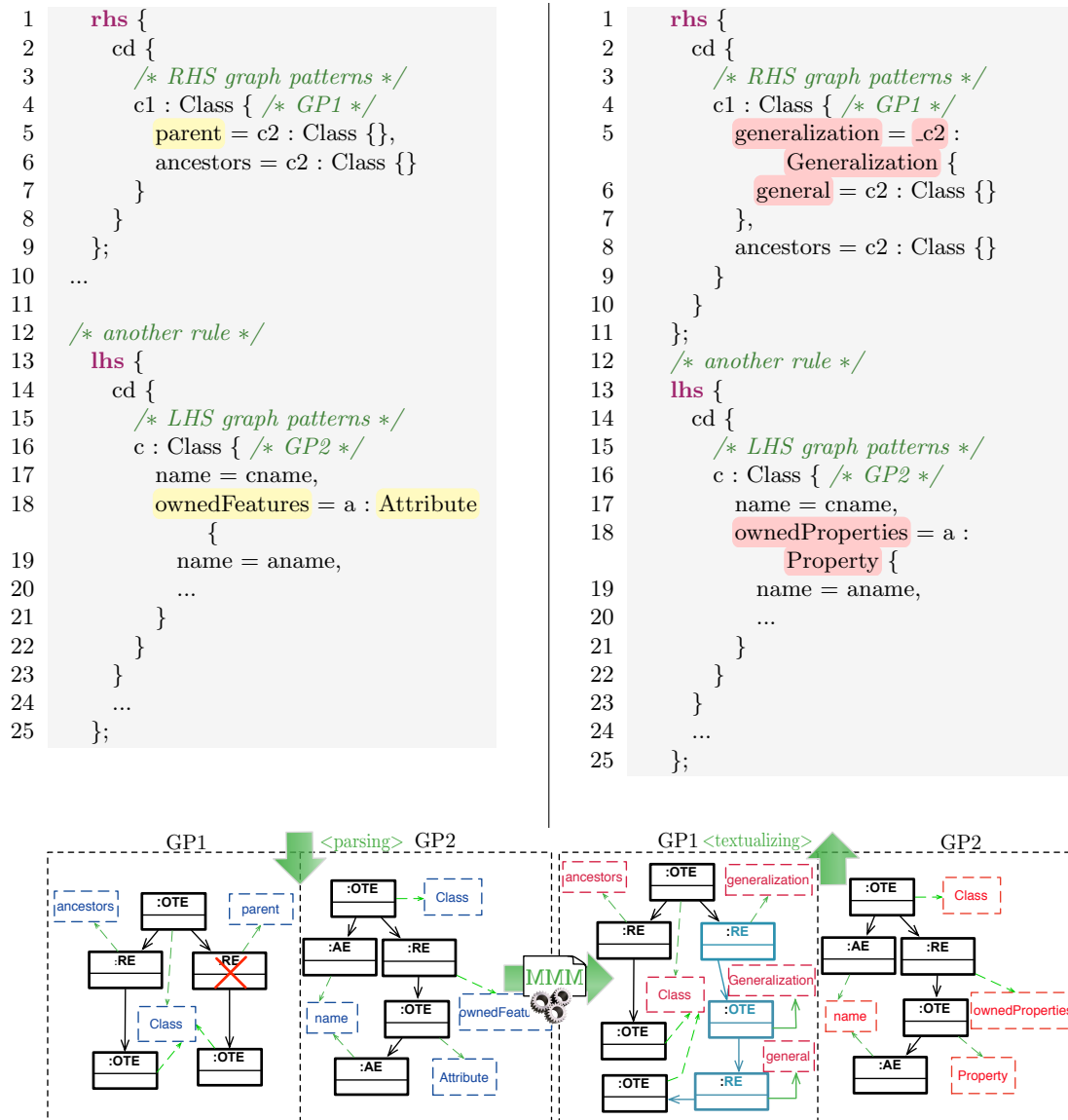


Figure 3.22: Excerpt of textual and in-memory representation of (model) graph patterns in adaptation process.

The left part of the figure shows the textual definition and the in-memory repre-

sentation of (a part of) two graph patterns of transformation CD2RDB, respectively. The right part of the figure illustrates the textual definition and the in-memory representation of these graph patterns after performing the rebinding action.

Once having established the sub-typing between metamodels (cf. formula 3.3) via the mapping CDV1ToCDV2 in Listing 3.4, the adaptation on the original transformation CD2RDB is performed for all graph patterns by means of the *doRebind()* procedure in Algorithm 5. In this procedure, we do the rebinding for correspondences first, through the *rebindChangedElem()* procedure in Algorithm 6, and then for the remaining unchanged elements by means of procedure *rebindUnchangedElem()* in Algorithm 7. For graph pattern GP2, from the in-memory representation (i.e. the model level) of the transformation and based on the mapping definition, we rebind the reference to *ownedFeatures* of the original metamodel CDV1 toward *ownedProperties* of the new metamodel CDV2. This case is a single mapping between two references, i.e. a Ref2Ref. Similarly, the reference to Attribute is forwarded to Property and so on for the pair (Class, Class). For graph pattern GP1, a reference reification (a Ref2FeaturePath), i.e. introducing the Generalization type, makes the adaptation process more complex. In this case, after rebinding the root OTE, we create a graph path to replace the original reference expression, as shown in Figure. 3.22. In the end, for the remaining unchanged elements such as *ancestors* and *name*, both graph patterns GP1 and GP2 can be automatically rebound based on these elements' names without any support from the mapping definition. For the full definition of the original transformation CD2RDB and its adjusted version, one can see Appendixes 1.1 and 1.2.

3.5 Summary and Discussion

This chapter has presented an approach to more efficiently reuse model transformations when changing their metamodel contexts. The approach is based on the graph topology of metamodels and uses it to define a notion of model type. More precisely, we have introduced a graph structure representation, i.e. Type Graph with Multiplicity (TGM), to exhibit the graph topology of Ecore-defined metamodels by flattening the Ecore *inheritance* relation between type concepts. Furthermore, this notion of model type uses the *upperBound* meta-attribute that is defined in metamodel Ecore as an example on selecting interesting constraints to define a notion of model type. A function that we call *model typing* function is also provided as a particular transformation to leverage every metamodel in the Ecore space into the more abstract space TGM.

From the proposed notion of model type, we show that the knowledge encoded in transformations can be captured by means of a so-called *transformation typing* function which infers user-defined graph patterns of a transformation written in transformation language MOMENT2-GT. The inference process produces real used parts of declared metamodels and also the abstraction level of the encoded knowledge. The outputs of the process are presented in the TGM space as *effective* TGM model types.

A notion of metamodel substitutability has also presented base on TGM model types, thanks to a name-based graph inclusion relation. However, inferred *effective* TGM model types are used to perform typing an existing transformation in place of the declared metamodels or their corresponding TGM model types. As a result, any metamodel whose the TGM image is a super-graph of the *effective* TGM of a declared metamodel could be used in the transformation directly without a need of adaptation. We call these cases *isomorphic* transformation reuse.

The simple above typing system in fact is insufficient to solve the problem of transformation reuse when considering metamodels that are more heterogeneous, e.g. name difference between elements or partial difference in topology. In order to extend the reuse capability, we developed a mapping DSL language, namely MetaModMap, dedicated to identify semantic correspondences between metamodel elements (also supporting for simple combinations of elements) and to adapt transformations. The language uses the proposed typing approach as primary premises to validate a mapping definition before the execution of a transformation adaptation. In making the use of a mapping definition, users can more recuperate the knowledge encoded in an existing transformation for a larger set of acceptable metamodels, nevertheless it is necessary to perform an adaptation step. We call these cases *non-isomorphic* transformation reuse.

As a conclusion, in this chapter we introduce a general solution for the problem of transformation reuse, both without and with transformation adaptation. The principles and also the mapping DSL language (MetaModMap), that have presented here are implemented together in the next chapter to work with model transformation language MOMENT2-GT.

Chapter 4

MetaModMap: A Transformation Migration Toolkit

Contents

4.1	Chapter Overview	123
4.2	Architecture of MetaModMap Toolkit	124
4.3	Realization of MetaModMap Toolkit	140
4.4	Transformation Adaptation in Action	148
4.5	Comparison to Transformation Reuse Approaches	151
4.6	Prototype Summary and Discussion	155

4.1 Chapter Overview

This chapter describes the prototype of a transformation migration toolkit which provides support for transformation reuse via adaptation with end-user directives, called the *Metamodel Mapping toolkit* or MetaModMap for short.

First, Section 4.2 introduces components involved in the toolkit including: an Eclipse-based *textual DSL editor* that is used to define semantic correspondences between metamodels in consideration; and an additional *interpreter* used for automatically migrating an existing transformation definition according to a valid mapping definition defined in the textual DSL editor. The textual DSL editor is developed as a *projectional editor*, that is a modern editor that allows users to edit the abstract syntax tree representation (model) of the mapping definition directly, while integrating the default behaviour of a textual DSL editor to some extent on completion, semantics and type checking. The user sees and edits text on the screen, however, actually the edited text is only an illusion (a projection) of a model. The interpreter acts as a Higher-Order Transformation (or HOT) [Mens 06], that means a program takes an user-defined transformation model in form of the abstract syntax of some

transformation language then transforms or re-factorizes it into another transformation model. In our context, the interpreter uses a mapping model as directives to migrate an existing transformation model.

Next, Section 4.3 details the implementation of the MetaModMap toolkit. In particular, we will describe, how the graph-based model typing approach introduced in Chapter 3 can be implemented into some necessary run-time concepts of the text editor. These run-time concepts are infrastructures for scoping and validation that support user-oriented functionalities like auto-completion and on-the-fly error detection when defining a mapping. Furthermore, we present how the interpreter is integrated as an automatic code generator of the text editor for on-the-fly transformation migration. By such integration, we aim to provide an easy to use transformation migration toolkit within Eclipse to end-users. The adaptation process is made automatic as much as possible, only requires some directives input from the user.

Then, Section 4.4 illustrates the use of the provided toolkit by describing an adaptation process for the motivating case study, i.e. adapting a transformation written to transform class models into relational database models in which input models could be designed in using two CASE Tools based on two different metamodels.

Finally, Section 4.6 summarizes the toolkit prototype¹.

4.2 Architecture of MetaModMap Toolkit

The goal of *Metamodel Mapping toolkit* (or MetaModMap) development is to build an additional set of Eclipse plug-ins around the MOMENT2-GT framework to make them available to migrate (or reuse via adaptation) existing model transformations written in the MOMENT2-GT language. Since framework MOMENT2-GT has been developed based on XTEXT, a DSL framework tool for developing textual languages, thus XTEXT DSL framework tool seems to be a suitable technology to build the MetaModMap toolkit. Using the same language development technology facilitates the integration of our toolkit into transformation framework MOMENT2-GT. Figure 4.1 illustrates an overview on the architecture of the toolkit, which is built to work with MOMENT2-GT and within the Eclipse platform.

The Eclipse Platform is structured around the concept of *plug-ins*, a mechanism that allows to contribute additional functionalities to the system. Based on this mechanism, new tools can be added to the platform by building plug-ins that extend the system. As show in Figure 4.1, Eclipse Modelling Framework (EMF) is a new tool added to the Eclipse system to provide a basic infrastructure for metamodeling such as metamodel editing, code generating for metamodels and run-time essential libraries. As a set of plug-ins, EMF provides on the one hand end-user facilities to create, edit and store metamodels, on the other hand, it exports infrastructure libraries that can be used to developing other tools.

1. Source code of the prototype, meta-models, transformations of the motivating example and a demonstration video are available to download at [\[MetaModMap 11\]](#).

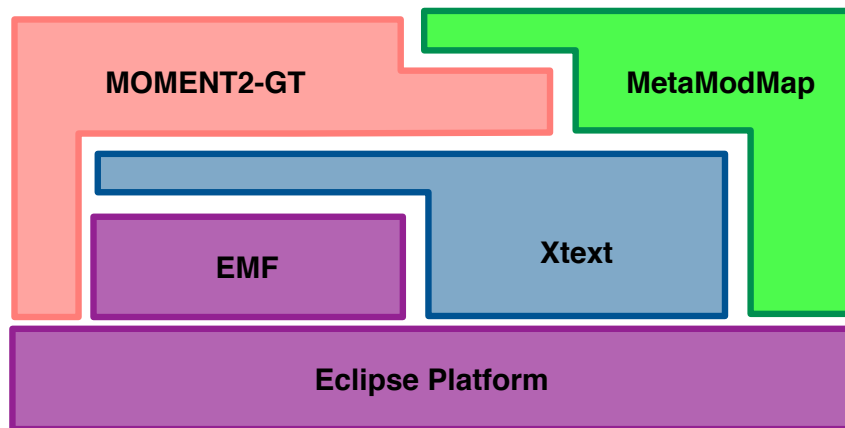


Figure 4.1: MetaModMap architecture at a glance.

XTEXT is a language development framework that is built upon the infrastructure libraries of EMF and Eclipse Platform. The XTEXT suite is composed of end-user facilities for defining textual concrete syntax of languages, a language component generator and a core infrastructure. XTEXT end-user facilities are constructed as extensions that are plugged into GUI extension points of the Eclipse Platform, where the language component generator automatically generates necessary components for a particular language.

The MOMENT2-GT transformation framework is developed by using the XTEXT suite. The framework provides a textual editor plug-in that is an extension plugged into the Editor extension point of the Eclipse Platform to write a transformation definition. Most of components of the MOMENT2-GT editor are generated by the XTEXT generator and customized according to particular semantics of the MOMENT2-GT language. For example, critical checks for a transformation definition could be added into the *scoping* and *validation* components of the editor. In contrast to the plug-in mechanism employed in Eclipse, XTEXT uses a dependency injection mechanism, with Google Guice, to manage bindings of concrete implementations to interfaces, thus a functionality can be easily extended its behavioral semantics over the default implementation in the XTEXT core infrastructure. In addition to the textual transformation editor, the MOMENT2-GT framework also develops other plug-ins providing GUI interfaces for configuring and launching a transformation by means of a specific transformation engine.

Since MetaModMap toolkit is developed for migrating MOMENT2-GT transformations when changing metamodels, it can be considered as an auxiliary tool for the MOMENT2-GT framework. Similar to MOMENT2-GT, the toolkit also provides a textual editor as an Eclipse extension to define a mapping between metamodels. The essential components generated by the XTEXT suite for the mapping language provide advance functionalities as those usually offered in modern IDEs. In this dissertation, we focus on the *scoping* and *validation* aspects that would be present in a modern textual editor to assist users in editing contents and report errors or warning in an on-the-fly manner. More precisely, check requirements in our graph-based

typing approach is integrated into components related to these aspects and these components are injected to the XTEXT-based generated editor. In addition to these important components, the *generator* component (also referred as a *build* component) of the textual editor is extended by inserting our own *interpreter*. The interpreter is responsible for the transformation to migrate once having a valid mapping on the editor.

We give in the followings a brief introduction of Eclipse Plug-in Platform, followed by a simple text editor plug-in example. Next, we present XTEXT and its architectural supports for automating the development of rich editors of textual languages in the Eclipse platform. Furthermore, in Section 4.3.1 an overview of XTEXT-based development for the MetaModMap text editor will be explained. In the last Section 4.3.2 the integration of the interpreter (a HOT) into the *generator* component of the XTEXT-based editor is presented.

4.2.1 Eclipse Plug-in Platform

The Eclipse Platform is built around the concept of **plug-ins** for integrating additional functionality and extending the platform with bundles of code (or modules). Additional functionality can be added in the form of code libraries (with public API of Java classes), platform **extensions** through custom plug-ins. A plug-in itself can add new content types by defining **extension points**, places where other plug-ins can add functionality. Plug-ins can also provide new functions for existing content types of the platform and plug-in specific UI contributions to the Eclipse workspace. When an extended Eclipse Platform is launched, an Integrated Development Environment (IDE) composed of available plug-ins is exposed to users.

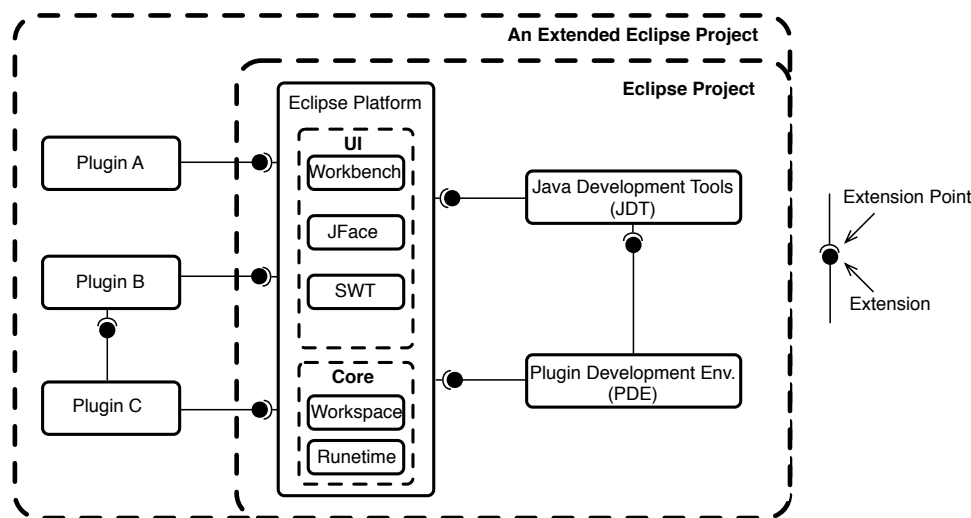


Figure 4.2: Eclipse plug-in based platform architecture.

Figure 4.2 shows an architecture overview of Eclipse and its extension mechanism through **extension points** and **extensions**. The Eclipse Platform is composed of

the Core modules and the UI modules. The Runtime Core module implements the run-time engine to start the platform base and dynamically discover and run plug-ins. The workbench UI module implements the workbench UI and defines a number of Eclipse Platform extension points that allow other plug-ins to contribute menu and toolbar actions, custom views and editors. The Standard Widget Toolkit (SWT) is a low-level, cross platform toolkit that supports platform integration and portable API. In contrast, the JFace UI framework provides higher-level application constructs for supporting dialogs, wizards, actions, user preferences, and widget management.

The standard Eclipse Project in fact is an extension of Eclipse Platform. That means the project adds custom functionalities to pre-defined extension points of Eclipse Platform. Eclipse Project provides two additional set of plug-ins that are useful for plug-in development, i.e. Java Development Tools (JDT) and Plugin Development Environment (PDE). The JDT plug-ins extend the platform workbench by providing a full featured Java development environment for editing, viewing, compiling, debugging, and running Java projects while PDE supplies specialized tools that automate the creation, manipulation, debugging, and deploying of plug-ins and extensions. These tools are also themselves examples of how new tools can be added to the Eclipse Platform by building plug-ins that extend the core platform.

Similarly, an Eclipse-based IDE is an extended Eclipse Project. To develop such an IDE for a particular purpose, we usually add additional plug-ins which connect to extension points of Eclipse Platform. For example, a plug-in for certain language can be plugged into the Eclipse Platform Editor extension point. Since plug-ins are implemented in Java, JDT and PDE extensions are used as a development environment to develop and deploy custom plug-ins.

Eclipse Plug-ins

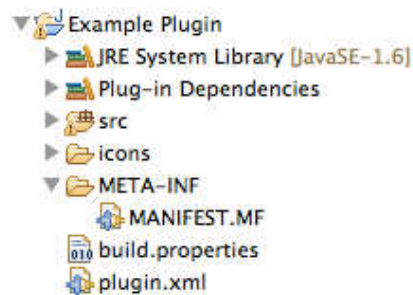


Figure 4.3: The simplest project structure of an Eclipse plug-in.

In the Eclipse plug-in development environment, every Eclipse plug-in consists of Java classes (could be packaged in JAR files after compiling) and two description files defining the plug-in's dependencies and extension points. The file `MANIFEST.MF` provides information about the version, name, required plug-ins and some other necessary configurations. The file `plugin.xml` describes contributions that the plug-in will add to the Eclipse Platform extension points or to the extension points defined in `plugin.xml` files of other plug-ins. The plug-in description file can declare any number

of extension points and any number of extensions to one or more extension points in other plug-ins. When a plug-in has no contributions to any extension points of other plug-ins, we can consider it as a normal library, thus the *plugin.xml* file is omitted.

Figure 4.3 shows the simplest structure of a plug-in in which Java classes that implement contributions or libraries are placed in the source folder (*src*). Listing 4.1 and 4.2 presents a content of the file MANIFEST.FM and an extension declaration in file *plugin.xml* for contributing to the Eclipse Platform Editor extension point, respectively.

Listing 4.1: Example Plug-in: MANIFEST.MF

```

1 Manifest-Version: 1.0
2 Bundle-ManifestVersion: 2
3 Bundle-Name: Example.Plugin
4 Bundle-SymbolicName: Example.Plugin; singleton:=true
5 Bundle-Version: 1.0.0.qualifier
6 Bundle-Activator: example_plugin.Activator
7 Require-Bundle: org.eclipse.ui, org.eclipse.core.runtime, org.eclipse.jface.text, org.eclipse.ui.
   editors
8 Bundle-ActivationPolicy: lazy

```

The MANIFEST.MF file shown in Listing 4.1 declares verbose information about the plug-in and required plug-ins to be able to use this plug-in. Since the plug-in provides an editor to the platform (see Listing 4.2) and the extension in this case is implemented using libraries provided by some other plug-ins of the platform, so we need org.eclipse.ui, org.eclipse.core.runtime, org.eclipse.jface.text, org.eclipse.ui.editors plug-ins.

Listing 4.2: Example Plug-in: *plugin.xml*

```

1 <plugin>
2   <extension point="org.eclipse.ui.editors">
3     <editor
4       name="Sample_XML_Editor"
5       extensions="xml"
6       icon="icons/sample.gif"
7       class="example_plugin.editors.XML_Editor"
8       id="example_plugin.editors.XML_Editor">
9     </editor>
10  </extension>
11 </plugin>

```

As shown in Listing 4.2, the *plugin.xml* file declares a list of extension points to which this plug-in contribute. In the example, only the org.eclipse.ui.editors extension point is declared to place an editor contribution into the UI workbench. The editor element consists of a name, file extension name and a contribution class declaration. The contribution class, in this case is example_plugin.editors.XML_Editor, is the plug-in's implementation in Java. When a user double-clicks on a XML file,

the class will be instantiated as a window for editing the file's content. However, to develop an editor, the Eclipse Platform provides some default implementation classes which support essential features for an editor, for example the library provided in the plug-in `org.eclipse.ui.editors`. Thus, all things to do is extends these classes, and customize contribution classes for particular computations, without a need of writing such classes from scratch.

Eclipse Platform-based Text Editor

In the followings, we describe briefly the principle to develop an Eclipse plug-in that provides a text editor with basic features. The contribution class for the editor will extend the concrete implementation class `TextEditor` that defines the behavior for the standard platform text editor, such as presentation of text, syntax highlighting, content assist features.

For the sake of simplicity, we will extend the Eclipse Platform with a model-independent text editor. That means the edited content is not represented by a structured resource, but directly in the buffer of the editor. Figure 4.4 gives an overview of such an extension with a plug-in named `myDSL.ui`. The plug-in describes in the file `plugin.xml` that it has a contribution to the Eclipse Platform Editor extension point.

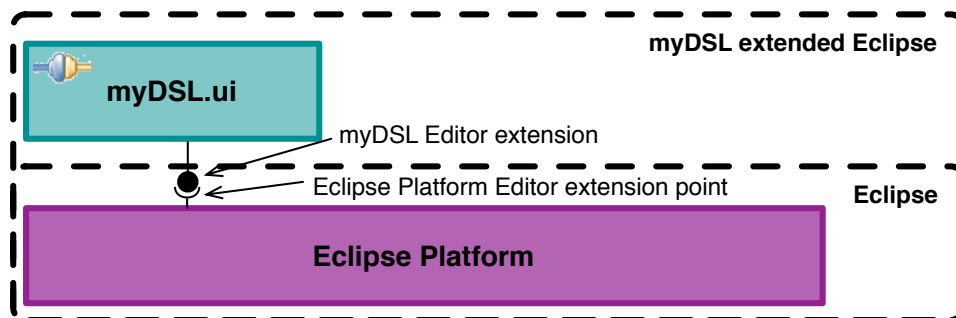


Figure 4.4: An example extending Eclipse Project by adding a text editor functionality.

Listing 4.3: `myDSL.ui` Plug-in: `plugin.xml`

```

1 <plugin>
2   <extension point="org.eclipse.ui.editors">
3     <editor
4       class="mydsl.ui.editors.myDSLEditor"
5       extensions="myDSL"
6       icon="icons/sample.gif"
7       id="mydsl.ui.editors.myDSLEditor"
8       name="My_DSL_Editor">
9     </editor>
10  </extension>
11 </plugin>

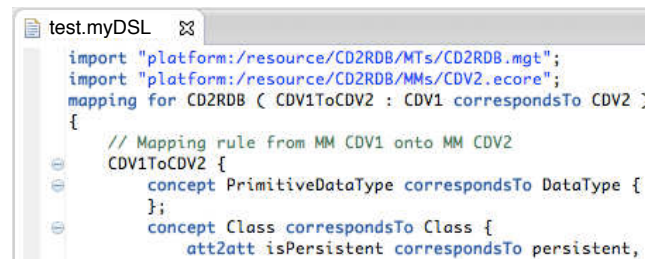
```

Listing 4.3 shows declarations in the file *plugin.xml* of the *myDSL.ui* plug-in. The description defines the name's extension of files, i.e. **.myDSL*, which can be opened by means of the editor. The contribution to the extension point is implemented in the Java file *myDSLEditor.java* in the package *mydsl.ui.editors*.

As mentioned in the previous paragraphs, the contribution class *myDSLEditor* could extend the default implementation class provided by the Eclipse Platform, i.e. class *TextEditor*, without a need to implement the required interface defined by the Eclipse Platform Editor extension point from scratch. As an example of customization, we will focus on adding the syntax highlighting (coloring) aspect to the editor. More precisely, we will explain how to implement our own contribution classes to override the default behaviors on coloring keywords of the language provided in basic classes of the platform.

As an illustration, let us take a part of the concrete syntax of the mapping language presented in Section 3.4.1 of Chapter 3 and add syntax coloring to the editor for **.myDSL* files as follows:

- Cyan for keywords of the language
- Blue for string (quoted by “ ”)
- Green for single-line comments (preceded by //)



```
test.myDSL
import "platform:/resource/CD2RDB/MTs/CD2RDB.mgt";
import "platform:/resource/CD2RDB/MTs/CDV2.ecore";
mapping for CD2RDB ( CDV1ToCDV2 : CDV1 correspondsTo CDV2 )
{
  // Mapping rule from MM CDV1 onto MM CDV2
  CDV1ToCDV2 {
    concept PrimitiveDataType correspondsTo DataType {
    };
    concept Class correspondsTo Class {
      att2att isPersistent correspondsTo persistent,

```

Figure 4.5: Syntax coloring for the *myDSL* editor.

Figure 4.5 shows the editor with such a syntax highlight as specified above. To this end, we can use Eclipse Platform APIs that help us add syntax coloring to our editor. Figure 4.6 shows the relationship of basic classes that Eclipse Platform provides and our necessary classes extending these classes for customization. The *myDSL.ui* project contains there classes *myDSLEditor*, *myDSLConfiguration* and *myDSLScanner* that extends basic classes *TextEditor*, *SourceViewerConfiguration* and *RuleBasedScanner*, respectively. These extending classes provide the following particular features for our editor:

- *myDSLEditor* - extends the default *TextEditor* class from Eclipse Platform. The *TextEditor* class has a containment relationship with the default *SourceViewerConfiguration* class. By default, the *SourceViewerConfiguration* class does not support syntax coloring. We will create our own class by extending the *SourceViewerConfiguration* class and override some methods to add the syntax coloring feature to the editor. In the *myDSLEditor* class, we change the default *SourceViewerConfiguration* class by our extending *SourceViewerCon-*

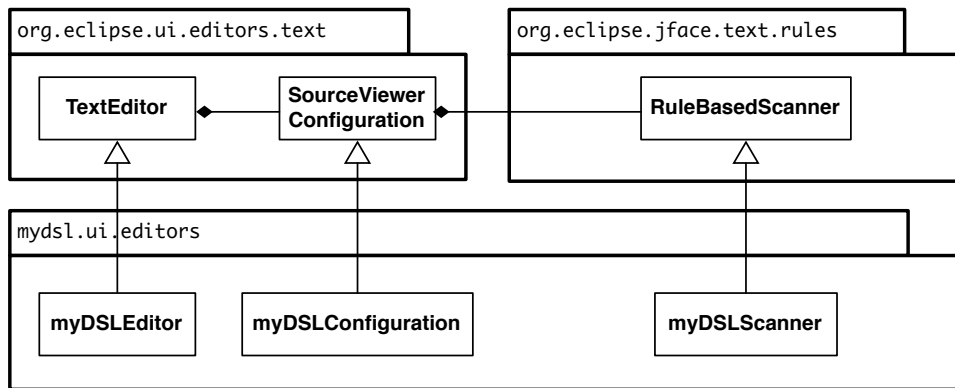


Figure 4.6: Relationship of default implementation classes.

figuration class as in Listing 4.4.

Listing 4.4: myDSL.ui Plug-in: myDSLEditor.java

```

1 public class myDSLEditor extends TextEditor {
2     public myDSLEditor() {
3         super();
4         setSourceViewerConfiguration(new myDSLConfiguration());
5     }
6 }
  
```

- myDSLConfiguration - extends the default SourceViewerConfiguration class from Eclipse Platform. Eclipse Platform uses the damage, repair, and reconcile model to manage the visual representation in the editor. To customize these models to our editor, we override the `getPresentationReconciler()` method of SourceViewerConfiguration. The overriding method configures the DefaultDamagerRepairer as shown in line 14 in Listing 4.5 by taking myDSLScanner as parameter (by invoking `getScanner()` method in lines 4 – 9). By such way, tokens that we define in myDSLScanner are passed to DefaultDamagerRepairer to repair the text that includes the color attribute.

Listing 4.5: myDSL.ui Plug-in: myDSLConfiguration.java

```
1 public class myDSLConfiguration extends SourceViewerConfiguration {
2     private myDSLScanner scanner;
3     /* Define token scanner for DefaultDamagerRepairer */
4     private ITokenScanner getScanner() {
5         if (scanner == null) {
6             scanner = new myDSLScanner();
7         }
8         return scanner;
9     }
10    /* Define reconciler for myDSLEditor */
11    public IPresentationReconciler getPresentationReconciler(ISourceViewer sourceViewer
12        )
13    {
14        PresentationReconciler reconciler = new PresentationReconciler();
15        DefaultDamagerRepairer dr = new DefaultDamagerRepairer(getScanner());
16        reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
17        reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);
18        return reconciler;
19    }
```

- myDSLScanner - extends the default RuleBasedScanner class from Eclipse Platform. The extending myDSLScanner class builds rules to detect different kinds of tokens. In our case that is strings (" "), single-line comments (//) and keywords of the languages. The constructor of the class adds rule to detect the patterns and associates color attributes with these patterns. The reconciler uses attributes stored in tokens to repair the representation in the editor. For example, in Listing 4.6 at lines 3, 10 and 17 we associate the color blue (RGB 0, 0, 255) with the token of strings. Similarly, the color green (RGB 0, 255, 0) is associated with the token of comments at lines 4, 11 and 15. Coloring keywords needs a bit more complex treatment. From line 19 to line 22, we use a WordRule to define general rule for normal text, then add keywords and tokens of keywords into the word rule.

Listing 4.6: myDSL.ui Plug-in: myDSLScanner.java

```

1 public class myDSLScanner extends RuleBasedScanner {
2     private static RGB KEYWORD = new RGB(0, 255, 255 ); /* cyan color */
3     private static RGB STRING = new RGB(0, 0, 255 ); /* blue color */
4     private static RGB COMMENT = new RGB(0, 255, 0 ); /* green color */
5     private static RGB OTHER = new RGB(0, 0 , 0 ); /* black color */
6
7     private static String[] fgKeywords = { "import", "mapping", "for", "correspondsTo",
8         , "concept", "att2att" };
9     public myDSLScanner() {
10         IToken keyword= new Token(new TextAttribute(new Color(Display.getCurrent(),
11             KEYWORD)));
12         IToken string= new Token(new TextAttribute(new Color(Display.getCurrent(),
13             STRING)));
14         IToken comment= new Token(new TextAttribute(new Color(Display.getCurrent(),
15             COMMENT)));
16         IToken other= new Token(new TextAttribute(new Color(Display.getCurrent(),
17             OTHER)));
18         List rules= new ArrayList();
19         /* Add rule for single line comments */
20         rules.add(new EndOfLineRule("//", comment));
21         /* Add rule for strings */
22         rules.add(new SingleLineRule("\"\"", "\"\"", string));
23         /* Add word rule for keywords */
24         WordRule wordRule= new WordRule(new WordDetector(), other);
25         for (int i= 0; i < fgKeywords.length; i++)
26             wordRule.addWord(fgKeywords[i], keyword);
27         rules.add(wordRule);
28         /* Set up pattern list */
29         IRule[] result= new IRule[rules.size()];
30         rules.toArray(result);
31         setRules(result);
32     }
33 }

```

At this moment, we have presented how to add an UI-related functionality of an IDE extension into Eclipse Platform. Particularly, we demonstrate how to add the syntax highlighting feature to the editor of a specific language by configuring the `SourceViewerConfiguration` class through the overriding mechanism. Other custom features can be plugged into an editor via the central hub `SourceViewerConfiguration`. However, developing an IDE from Eclipse Platform's APIs demands a lot of effort into customizing functionalities. In the next section, we will present another solution by means of XTEXT, a DSL framework tool for developing textual languages. Since XTEXT provides APIs at a higher level for well-defined features in modern text editors

and generators, using XTEXT reduces the time to develop IDEs.

4.2.2 Language Development with Xtext

This section gives an overview of XTEXT [Xtext 08], its architectural supports for automating the development of rich editors of textual languages integrated in the Eclipse platform. XTEXT is a complete environment for development of programming languages and domain specific languages that can be integrated in Eclipse Platform. XTEXT itself is implemented in Java and is based on Eclipse, EMF, and Antlr.

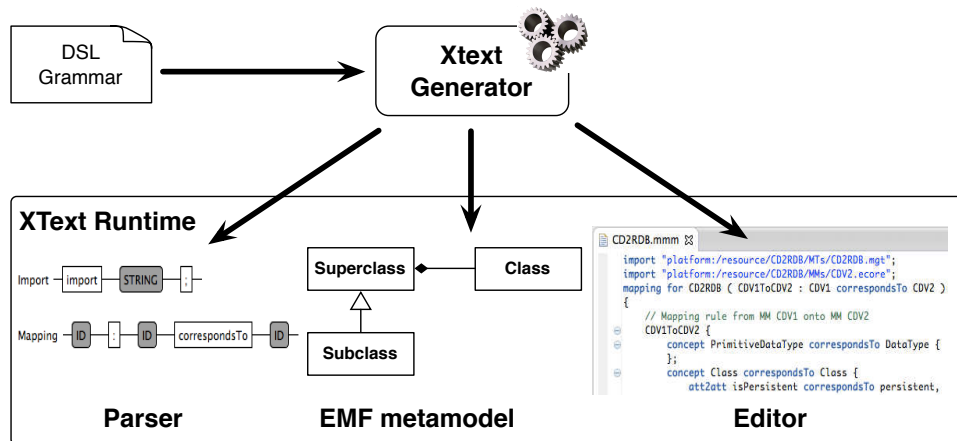


Figure 4.7: Overview of XTEXT

Figure 4.7 gives an overview on the XTEXT framework. The framework is composed of a generator and a run-time environment. By using XTEXT, developers can describe their own languages in a simple EBNF-like grammar and the generator will automatically create a parser, an EMF metamodel and a full-featured Eclipse Platform text editor. After having generated artifacts, the developer can easily add new features to a generated editor to extend its functionalities for some particular purposes. The APIs of the language and its specific editor are generated as Eclipse plugins; that means the generator contributes to create MANIFEST.MF files, *plugin.xml* files and contributions classes in respecting the requirements of Eclipse Platform extension mechanism. Based on this feature, every tool developed by using XTEXT can be fully integrated in Eclipse Project. To manage generated artifacts, XTEXT allows developers to configure the generator using the Modeling Workflow Engine (MWE2). A configuration file consumed by the XTEXT generator is composed by fragments (shown in Table 4.1) for generating artifacts such as parsers, serializers, the EMF code of metamodel, content assist contribution, etc.

Dependency Management in Xtext

All IDE components created by the XTEXT generator for a particular language are assembled by means of the lightweight Dependency Injection (DI) framework Google Guice. In XTEXT, most of generated artifacts or built-in libraries are implemented

Class	Generated Artifacts
EcoreGeneratorFragment	EMF code for generated models
XtextAntlrGeneratorFragment	ANTLR grammar, parser, lexer
GrammarAccessFragment	Access to the grammar
ResourceFactoryFragment	EMF resource factory
ParseTreeConstructorFragment	Model-to-text serialization
JavaScopingFragment	Scoping
JavaValidatorFragment	Model validation
FormatterFragment	Code formatter
LabelProviderFragment	Label provider
OutlineTreeProviderFragment	Outline view configuration
JavaBasedContentAssistFragment	Java-based content assist
XtextAntlrUiGeneratorFragment	ANTLR-based Content assist helper

Table 4.1: XTEXT standard fragments and artifacts

as services. A service is an object which implements a certain interface and XTEXT uses Guice to instantiate services and supply them to dependent ones. This means basically that whenever some code is in need for functionality (or state) from another component, all we need to do is to declare the dependency rather than stating how to resolve it, i.e. obtaining that component without instantiating or binding the component in an explicit manner.

For instance, the linking feature shipped within XTEXT is implemented as a service which allows to perform cross-links which are declared in an XTEXT grammar. This service requires a specification of linking semantics (usually provided via the scoping API, i.e. `IScopeProvider`). To this end, the linking service declares a field of type `IScopeProvider` (or in method or constructor), see Listing 4.7, and adds the `@Inject` annotation:

Listing 4.7: XTEXT: dependency declaration

```

1 public class DefaultLinkingService extends AbstractLinkingService {
2     @Inject
3     private IScopeProvider scopeProvider;
4 }

```

By such mechanism, any customization of the linking behavior can be described by implementing another service implementing the `IScopeProvider` interface. It is not the duty of the client code in somewhere to care about where the *scopeProvider* comes from or how it is created. When class `DefaultLinkingService` is instantiated, Guice sees that this instance requires an instance of `IScopeProvider` and assigns an object to the specified field or method parameter. Of course, the object itself is created by Guice. To this end, Guice needs to know how to instantiate real objects for declared dependencies. This is done by means of so-called Modules. A Module defines a set of mapping from types to their existing instance, instance providers or concrete classes.

Listing 4.8: XTEXT: configuration of components

```
1 public class MyDSLUIModule implements Module {
2     @Override
3     public void configure(Binder binder) {
4         binder.bind(IScopeProvider.class).to(MyDSLScopeProvider.class);
5     }
6 }
```

Listing 4.8 shows an example on configuring a scoping component to the scoping API. This configuration is described in a Module that can be generated by XTEXT generator when using the `JavaScopingFragment` class in Table 4.1. The generated class `MyDSLUIModule` implements the Guice Module interface which requires an implementation of a method called *configure* and gets a `Binder` passed in. That binder provides APIs to define the mentioned mappings. In this case, we declare a mapping from a type (i.e. the `IScopeProvider` interface) to a concrete class (i.e. the `MyDSLScopeProvider` class) by means of APIs `bind()` and `to()` provided in the Guice `Binder` interface.

To wire up an application with Guice, it is necessary to initialize an `Injector` using the declared Guice modules. The root instance of the whole application should be created by means of that injector. In a standalone mode, this is usually done in the main method of the entry class. Listing 4.9 is an example of initialization of a Guice-based application, the application is instantiated via the injector before running provided functionalities.

Listing 4.9: Wiring up an application in standalone mode

```
1 public static void main(String[] args) {
2     /* Create the injector by passing an own Guice module */
3     Injector injector = Guice.createInjector(new MyDSLUIModule());
4     /* Create an application by using the injector to wire up all dependencies */
5     MyDSLApp app = injector.getInstance(MyDSLApp.class);
6     /* Now ready to use the application */
7     app.run();
8 }
```

However, in XTEXT the instantiation of the injector is generated automatically for each language by the XTEXT generator. All a developer needs to do is to define the grammar of a language, then configure fragments in order to create essential artifacts for the editor. The developer can then customize default behaviors by adding or overriding methods in generated artifacts without caring about how to manage dependencies between them. This architecture makes XTEXT artifacts easy to extend, test and compose, thus reduces time for development of toolkits. In the following paragraphs, we will explain the mechanism to integrate a XTEXT-based editor application into Eclipse Platform when using Guice for dependency management.

Integrating a Xtext-based Editor into Eclipse

As aforementioned, the injector used to wire up a XTEXT-based editor application is generated automatically by the XTEXT generator. The instruction which creates a Guice Injector using the modules in line 3 of Listing 4.9 is placed in the generated Activator file of the plug-in. In addition, the creation of the editor application for each language (corresponding the instruction in line 5 of Listing 4.9) is configured in an `IExecutableExtensionFactory`, which is used to create `IExecutableExtensions` plugged into Eclipse Platform. By such way, every extension application, which is created via extension points is managed by Guice, i.e. dependencies are declared and injected upon creation of an extension.

As a language development framework, XTEXT has provided a text editor default implementation that extends the `TextEditor` class of Eclipse Platform, i.e. `XtextEditor`. The `XtextEditor` class and all run-time infrastructures of XTEXT have declared all necessary APIs and dependencies between APIs that can be wired up by Guice. A language developer does not need to redefine a new editor at the Eclipse Platform level. Instead, the developer customizes his own editor by adding or overriding methods in generated artifacts which will be wired up later by Guice. The only thing the developer has to do in order to set up the editor plug-in is to prefix the `XtextEditor` class with the generated factory's name followed by a colon in the *plugin.xml* file as in Listing 4.10.

Listing 4.10: XTEXT-based myDSL.ui Plug-in: *plugin.xml*

```
1 <plugin>
2   <extension point="org.eclipse.ui.editors">
3     <editor
4       class="MyDslExecutableExtensionFactory:mydsl.ui.editors.XtextEditor"
5       extensions="myDSL"
6       icon="icons/sample.gif"
7       id="mydsl.ui.editors.myDSL"
8       name="MyDSL_Editor">
9     </editor>
10  </extension>
11 </plugin>
```

In practice, the description file *plugin.xml* is generated by the XTEXT generator. In addition to the contribution extension to the Eclipse Platform Editor extension point, the XTEXT generator also generates many other necessary extensions that are required for an advance IDE. In this dissertation, we focus only on the text editor extension and how to integrate the theoretic parts of our approach into this extension. For other generated extensions by XTEXT, we do not customize their default implementation.

Language Editor Customization in Xtext

The development of a textual language and its editor in using XTEXT pass through

several stages starting with the definition of a grammar file to the customization of generated artifacts of a fully eclipse integrated text editor as you can see in Figure 4.7 at page 134. These stages are as follows:

- Defining a grammar, i.e. the concrete syntax of the language.
- Configuring generator fragments for creating XTEXT artifacts.
- Customizing generated artifacts for run-time/IDE concepts, e.g. adding or overriding rules for scoping and validating APIs.
- Developing a generator to translate the language to other languages or notations.

In order to develop a language, it is necessary to define a grammar for that language. XTEXT uses a simple EBNF-like grammar language for the description of textual languages. The main idea is to describe the concrete syntax and XTEXT will generate an ANTLR parser to map the text (or a persistent input file) written in the corresponding editor to an in-memory representation - the semantic model. The next stage consists in defining fragments of the code generator for generating XTEXT artifacts, e.g. Scoping or Model validation artifact and others as listed in Table 4.1. Generated artifacts usually extend default implementations shipped with APIs of the XTEXT framework, thus a language developer should customize these artifacts to enrich the semantic behavior of the editor by adding or overriding methods in respecting XTEXT APIs. Finally, an indispensable stage is to develop a generator to interpret the semantic of an input by translating it into other languages or notations.

In what follows we focus on the customization of generated XTEXT artifacts and the development of a generator. In particular, we will mention the two most important artifacts that concern the integration of the theoretic parts of our approach, i.e. the Scoping and Model Validation artifacts which are generated by `JavaScopingFragment` and `JavaValidatorFragment` and the Generator artifact.

JavaScopingFragment – Scoping: The generated Scoping artifact, i.e. a scope provider transitively implements the `IScopeProvider` API by extending the `AbstractDeclarativeScopeProvider` class, gives all operations needed for the semantic and syntactical expressions defined in the grammar except for the cross-reference assignments. Although XTEXT allows to declare a cross-link in a grammar file by stating the type of a referred element, it is not enough for the linking service (see Listing 4.7) in a particular case where one needs to limit the domain of referable elements instead of only declaring their type. For example, when defining a mapping in our approach (see Listing 3.4 in Chapter 3 for the example) we should limit referable Ecore elements in taking into account a certain context, e.g. we allow users to only refer to real used Ecore elements in a particular transformation – the transformation typing mechanism. Such a sophisticated semantics can be described by adding declarative methods in the generated scope provider, without this customization one can possibly write inconsistent assignments.

JavaValidatorFragment – Model validation: The generated Model validation artifact, i.e. a custom validator extends the library class `AbstractDeclarativeVal-`

idator, allows user to specify additional constraints specific for models (in-memory representations). Static analysis or validation is an indispensable aspect when developing a modern editor for a language. It provides informative feedback as the users of the language type a model definition. By default, XTEXT provides several levels of validation for a defined language. The automatic validation level regards the syntactical validation done by the parser, the cross-link validation done by a linker, which is often customized through the generated Scoping artifact, the concrete syntax validation done by a serializer that validates all constraints that are implied by the defined grammar. The custom validation level is described manually by the developer by adding methods annotated with the @Check annotation into a custom validator class inheriting the AbstractDeclarativeValidator class. Based on the reflective implementation of AbstractDeclarativeValidator, XTEXT allows developers to write constraints in a declarative manner. That means annotated methods will be invoked automatically when validation takes place. This is one of the most interesting features when we need to add more semantically checks into some point of the model without writing exhaustive if-else constructs. For example, the sub-typing mechanism proposed in Section 3.3.3 of Chapter 3 can be integrated into the MetaModMap editor by defining a validation rule in the generated Model validation artifact.

All these levels of validation are performed in an on-the-fly mode while a language's user types the model. If there is an error in the model, the validator reports an error message in the error-log/problem views of eclipse and the point in which the error occurs is marked in the language editor.

Code generator: The run-time framework of XTEXT provides a plug-in called Xtext Builder (in bundle org.eclipse.xtext.builder) that declares an extension point for building projects, i.e. the org.eclipse.xtext.builder.participant extension point. This extension point defines the interface IXtextBuilderParticipant that is shipped with a default implementation, i.e. the BuilderParticipant class. This class declares a dependency via the IGenerator interface that can be implemented by the generated Generator artifact. The dependency between BuilderParticipant and the generated Generator artifact is also managed by Guice, thus it is necessary to declare a concrete binding in the Guice configuration module of the language. A language developer can use the BuilderParticipant class as a contribution that is plugged into the Xtext Builder plug-in. By default, the generated Generator artifact is called by Xtext Builder when saving (committing) the edited content in the editor to the file. When invoking the generated Generator artifact, Xtext Builder passes the model (using the generated ANTLR parser to obtain the in-memory representation of the textual content) to the generator if no validation rule on the model is violated. It is an interesting feature in order to realize a non-trivial scenario from *model checking* to *model interpreting* in which *model checking* is customized in Scoping and Model Validation artifacts while *model interpreting* is implemented in a generator.

As an example, the transformation adaptation algorithm (known as a HOT algorithm) proposed in Section 3.4.2 of Chapter 3 can be realized in the custom implementation of the generated Generator artifact since the algorithm takes the model of

a mapping definition and one of an original transformation definition as inputs. The output of the algorithm is a model of an adapted transformation definition, thus can be persisted to disk by means of the default implementation Xtext Serializer artifact.

Summing up, in this section we have introduced first the architecture to develop a plain Eclipse editor as plug-ins. Next, language development in using XTEXT with an introduction for customizations has been presented. We particularly focus on aspects that are related to our theoretic parts such as scoping, validating and generator. In the next section, we will present the concrete implementation for a prototype of the MetaModMap language. Based on focused aspects, the integration of our model/transformation typing approach into scoping, validating aspect of the MetaModMap editor will be explained first, and then the implementation of the generator for a Higher-Order Transformation.

4.3 Realization of MetaModMap Toolkit

As aforementioned in the previous section, the MetaModMap toolkit is designed to work with the MOMENT2-GT framework within the Eclipse platform. Thus, one can consider MetaModMap as an additional set of plug-ins that extends MOMENT2-GT IDE. These plug-ins could be integrated directly into Eclipse Platform via certain well-defined interfaces in the platform's extension points. Furthermore, the provided functionality of MetaModMap is to migrate transformations written in MOMENT2-GT language, thus the toolkit needs to understand the infrastructure APIs of the language, e.g. the metamodel, the parser and much more. To this end, it is necessary to declare this dependency between MetaModMap's plug-ins and MOMENT2-GT's plug-ins.

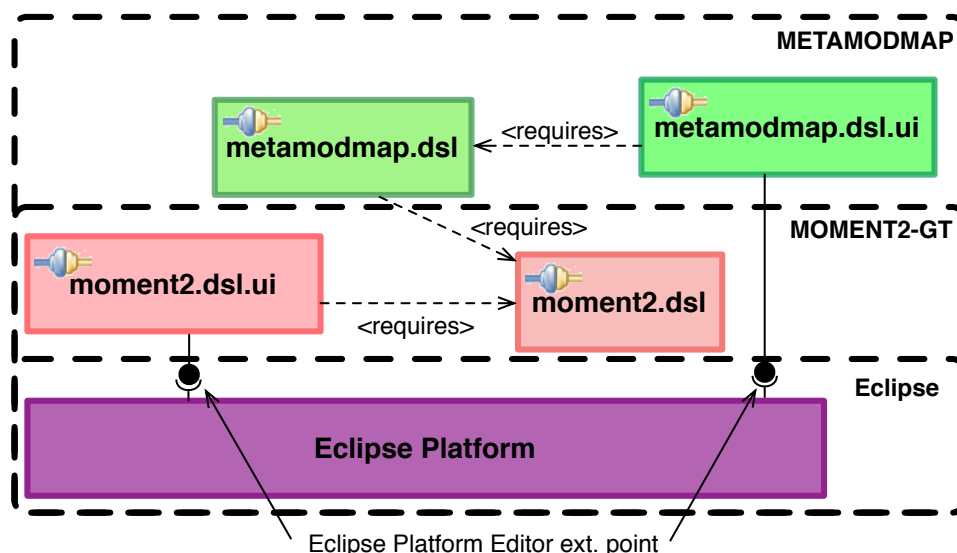


Figure 4.8: Integration of MetaModMap editor into MOMENT2-GT.

Figure 4.8 gives an overview of the integration of the MetaModMap toolkit into

the existing IDE MOMENT2-GT. In this figure, the MetaModMap language resides logically over the MOMENT2-GT language. These two languages have inter-language-cross-references, more precisely, a mapping definition can refer to a transformation definition to declare that the mapping is dedicated for a specific usage context, i.e. the transformation. This feature is possibly realized with the XTEXT's supports in cross-grammar definitions. Another side related to the structure of projects, XTEXT separates a generated language into two main plug-ins for modularity: an UI-related plug-in that requires an infrastructure-related plug-in. As shown in Figure 4.8, in the MOMENT2-GT layer the generated `moment2.dsl.ui` plug-in integrated into Eclipse Platform via the Editor extension point requires the generated `moment2.dsl` plug-in. The generated `moment2.dsl` plug-in contains most generated infrastructural artifacts (see Table 4.1) that can be customized by the language developer. Similarly, the METAMODMAP layer has the `metamodmap.dsl.ui` plug-in that requires the `metamodmap.dsl` plug-in. The cross-reference between two languages is supported by the requirement declaration between the `metamodmap.dsl` plug-in and the `moment2.dsl` plug-in.

In what follows we will present customization on considered artifacts in the `metamodmap.dsl` plug-in that are related to our theoretic parts presented in Chapter 3. More precisely, the customization focuses on scoping and validation aspects in order to support informative feedback from the editor to users. In addition, the implementation of an Higher-Order Transformation for the code generator interface will be also explained in detail.

4.3.1 Xtext-based Editor for MetaModMap

Developing a new language with XTEXT needs to define a grammar, then to use a configured XTEXT generator in order to create all artifacts for the language, see Figure 4.7 for the XTEXT workflow. As in the theoretic part presented in Chapter 3, the MetaModMap language is designed to declare semantic correspondences between Ecore-defined metamodels' elements in taking a MOMENT2-GT transformation into account, thus the grammar of MetaModMap requires references to the Ecore metamodel and the metamodel of MOMENT2-GT. This is done as in Listing 4.11 with provided supports of XTEXT:

Listing 4.11: An excerpt of MetaModMap XTEXT Grammar

```

1 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
2 import "http://www.ac.uk/le/cs/moment2/Moment2transformation" as moment2
3 MetaModMapRoot:
4     "import" importURI=STRING ";"
5     (imports+=Import)+
6     "mapping_for" transformation=[moment2::Transformation]
7     "(" mmaps+=Mapping (";" mmaps+=Mapping)* ")"
8     "{"
9     (mapRules+=MapRule)+
10    "}"

```

11 ;

In the grammar, the Ecore meta-metamodel and the metamodel of MOMENT2-GT are imported at the header and aliased as `ecore` and `moment2`, respectively. Having these importations, every type defined in `ecore` and `moment2` is available and can be used to define cross references in our language. For example, to declare the usage context (i.e. the transformation) for a mapping, line 6 in Listing 4.11 defines the cross-reference “`transformation=[moment2::Transformation]`” for the type `MetaModMapRoot`. One can see Figure 3.12 and Listings 3.2, 3.3 in Chapter 3 for the whole abstract syntax and concrete syntax of the MetaModMap language, respectively.

After generating the language from the grammar with the generator, we obtain the language infrastructure and all customizable artifacts which are located in the `metamodmap.dsl` plug-in, cf. Figure 4.8. We particularly consider the following customizable artifacts that are related to scoping and validation aspects of the being-developed editor:

- *The Scoping artifact*: the `MetamodmapScopeProvider` class extends the `AbstractDeclarativeScopeProvider` class which implements interface `IScopeProvider`. `AbstractDeclarativeScopeProvider` provides a mechanism which allows a language developer to declare custom scoping rules in a declarative manner when extending this class. `MetamodmapScopeProvider` is managed and wired up into other parts by means of the following configuration in a Guice module:

Listing 4.12: Configuration for the custom scoping artifact

```

1 public class AbstractMetamodmapRuntimeModule extends DefaultRuntimeModule {
2     /* contributed by AbstractScopingFragment */
3     public Class<? extends IScopeProvider> bindIScopeProvider() {
4         return MetamodmapScopeProvider.class;
5     }
6 }

```

- *The Validation artifact*: the `MetamodmapJavaValidator` class extends transitively the `AbstractDeclarativeValidator` class which implements interface EMF’s `EValidator`. `AbstractDeclarativeValidator` provides a mechanism which allows to write constraints in a declarative manner with the `@Check` annotation when extending this class. `MetamodmapJavaValidator` can be enable by adding the following configuration in a Guice module of the language:

Listing 4.13: Configuration for the custom validation artifact

```

1 /* contributed by JavaValidatorFragment */
2 @SingletonBinding(eager=true)
3 public Class<? extends MetamodmapJavaValidator> bindMetamodmapJavaValidator()

```

```
4 { return MetamodmapJavaValidator.class; }
```

In practice, all Guice configurations are also generated automatically for the corresponding generator's fragments. A language developer does not need to configure manually dependencies, he just adds his own semantics to customizable generated artifacts of the language.

Scoping Customization

The *scoping check semantics* presented in Chapter 3 (cf. page 106) is implemented in a declarative manner in the generated scoping artifact, i.e. the `MetamodmapScopeProvider` class that extends the `AbstractDeclarativeScopeProvider` class. The base class `AbstractDeclarativeScopeProvider` allows to declare methods in the following two signatures:

Listing 4.14: Two signatures of methods for scoping customization

```
1 IScope scope_<RefDeclaringType>_<Reference>(<ContextType> ctx, EReference ref)
2 IScope scope_<TypeToReturn>(<ContextType> ctx, EReference ref)
```

The first method signature is used when evaluating the scope (the set of candidates) for a specific cross-reference between types and the second one is used when computing the scope for a given type and is applicable to all cross-references of that type. The returned `IScope` should contain all candidate elements for the cross-reference. A scope, which is nested in a single `IScope` contains values, is abstract descriptions of real `EObjects`. One can see more details on using scoping of `XTEXT` at [Xtext 12].

We have implemented the scoping check semantics in using the two above method signatures for cross-references of the `MetaModMap` language. For demonstration purpose, we present here two examples of methods that are related to the integration of the model typing algorithm (cf. Algorithm 1) and the model transformation typing algorithm (cf. Algorithm 3) for filtering cross-references.

Listing 4.15: Method for the effective TGM model type inference

```
1 IScope scope_Mapping(MetaModMapRoot ctx, EReference ref)
2 {
3   Transformation transformation = ctx.getTransformation();
4   Trace<EObject, EObject> trace = TraceImpl.getDefault();
5   for (Mapping map : context.getMmaps())
6   {
7     /* infer TGM type of source MM */
8     EPackage mmSource = map.getModelTypeSource();
9     TypeChecker.modelTypingFunction(mmSource,trace);
10    /* Typing original MM by taking into account the transformation */
11    TypeChecker.transfTypingFunction(mmSource,transformation,trace);
12    /* infer TGM type of target MM */
```

```

13   EPackage mmTarget = map.getModelTypeTarget();
14   TypeChecker.modelTypingFunction(mmTarget,trace);
15   }
16   /* return the default scope based on naming convention computed by \textsc{Xtext} */
17   return super.delegateGetScope(ctx, ref);
18   }

```

The first example method performs the effective TGM model type inference when an user begins defining inside details of a mapping. Listing 4.15 shows a simplification of the method. The method *scope_Mapping()* first adds typing functions, cf. from line 3 to line 15, and then reuses the name-based scope computed by XTEXT, cf. line 17 in the listing. This method is invoked when the user types on the editor to define a reference to an existing instance object of Mapping which has declared the source metamodel and the target metamodel. For the sake of simplicity, we regroup all methods related to our typing approach in the static class TypeChecker. The *modelTypingFunction()* method, cf. Algorithm 1, is used in lines 9 and 14 to perform the inference of TGM model types for metamodels. The *transfTypingFunction()* method, cf. Algorithm 3, is invoked in line 11 to infer the *effective* TGM model type for a source metamodel in taking its usage context, i.e. the transformation in consideration, into account. In order to keep corresponding links between metamodel's elements and elements of TGM model type, we use a shared trace object as declared in line 4 of the listing. The implementation of the TraceImpl class is in Appendix 2.1. The on-the-fly invocation of typing functions when referring a mapping is shown in Figure 4.9.

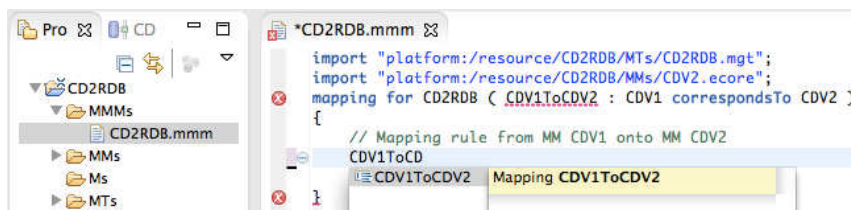


Figure 4.9: Invocation of typing functions when referring to a mapping.

The second example method is declared in using the first signature in Listing 4.14 in order to compute the set of EClass objects that are referable by the *conceptSource* reference of type ConceptMap. The method is implemented as shown in Listing 4.16.

Cross-references stated in a grammar by indicating the type of references is usually insufficient to specify the complete linking semantics. For example, the declaration “ConceptMap : conceptSource=[ecore::EClass]” in the MetaModMap grammar, cf. Listing 3.3, states that for the reference *conceptSource* only instances of the type *ecore:EClass* are allowed. However, this declaration does not say where to find the classes. That is the duty of the scoping method *scope_ConceptMap_conceptSource()*. In that method we build the set of candidate classes from those in the source metamodel, and then select only classes that have at least a corresponding link to the corresponding TGM model type, cf. from line 3 to line 13 in Listing 4.16. Finally,

Listing 4.16: Scope computation for the *conceptSource* reference of ConceptMap

```

1 IScope scope_ConceptMap_conceptSource(ConceptMap ctx, EReference ref)
2 {
3   MapRule superCtx = (MapRule) ctx.eContainer();
4   /* get EClass objects contained in the corresponding modelType */
5   EPackage modelType = superCtx.getDomain().getModelTypeSource();
6   EList<EClass> eClassList = new BasicEList<EClass>();
7   for(EClassifier cl : modelType.getEClassifiers()) {
8     if (cl instanceof EClass) {
9       if (TraceImpl.getDefault().getTargetElems(cl) != null) {
10        eClassList.add((EClass)cl); /* add a real used EClass in the candidate list */
11      }
12    }
13  }
14  /* create the scope of candidates */
15  Iterable<IEObjectDescription> scopedElems = Scopes.scopedElementsFor(eClassList);
16  return new SimpleScope(IScope.NULLSCOPE, scopedElems);
17 }

```

the computed candidate list is used to create the scope for the reference *conceptSource*.

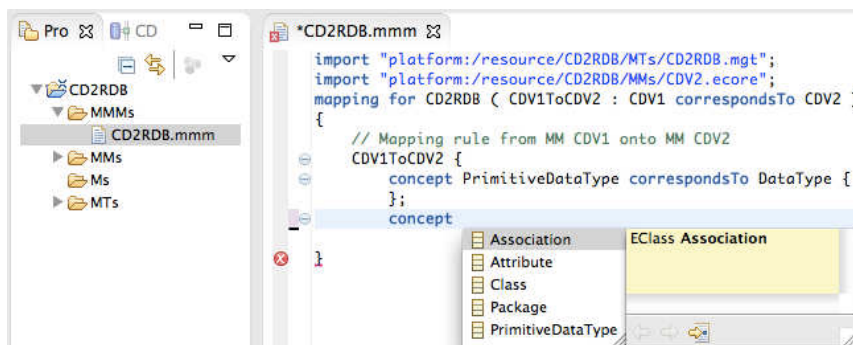


Figure 4.10: Scoping when defining a mapping between two concepts.

Figure 4.10 shows the computed scope popped up by means of the content assist feature when we define a concept mapping. As you see in the figure, the proposal contains only real used classes by the transformation. The unused class *Operation* defined in metamodel *CDV1* is not in the proposal list. Using such a scoping mechanism can avoid writing inconsistent assignments.

Validation Customization

The *validation check semantics* presented in Chapter 3 (cf. page 107) is implemented in the generated validation artifact, i.e. the *MetamodmapJavaValidator* class that extends transitively the *AbstractDeclarativeValidator* class. The base class *AbstractDeclarativeValidator* allows to declare methods in order to add constraints into a language with the *@Check* annotation. Recall that the *model type compatibility*

validation check semantics is performed at a coarse-grained level, i.e. between two metamodels referred in the instance object of Mapping, thus we add the following method into the `MetamodmapJavaValidator` class:

Listing 4.17: *Model type compatibility* validation check method

```

1 @Check
2 public void isModelTypeCompatible(Mapping mapping)
3 {
4     if (!TypeChecker.isTGMCompatible(mapping)) {
5         error("Model types are not compatible through the mapping",
6             MetamodmapPackage.Literals.MAPPING_NAME);
7     }
8     else {
9         info("Model types are compatible through the mapping",
10            MetamodmapPackage.Literals.MAPPING_NAME);
11     }
12 }

```

The validator acts as a visitor. More precisely, when an instance object of Mapping is reached, the validator looks up all methods annotated with `@Check` and invoke methods that declare a signature of type Mapping. The `isModelTypeCompatible()` method in Listing 4.17 just forwards the call to the `isTGMCompatible()` method, which is implemented in the type checker. The `isTGMCompatible()` method takes the entry object, i.e. the reached instance object of Mapping, then performs mapping operations for the *effective* TGM model types of source metamodels according to the details in the mapping definition. To the end, the TGM sub-graph relation, cf. Algorithm 2, is verified between the *mapped* effective TGM model types and those corresponding to new metamodels.

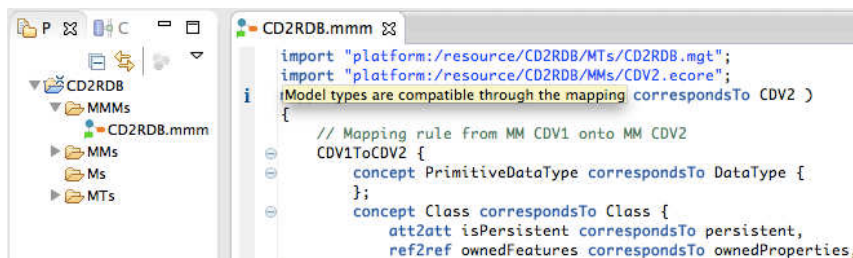


Figure 4.11: *Model type compatibility* validation at coarse-grained level.

Figure 4.11 shows how the MetaModMap validator checks the model type compatibility. When the validator is activated, it should be invoked if there is any change on the user-input. In this figure, the pop-up notification says that the user-defined mapping makes two metamodels become compatible at model type level, thus the mapping definition is valid before passing it to the transformation adaptation phase.

4.3.2 Higher-Order Transformation

As aforementioned in Section 4.2.2, XTEXT provides a mechanism which allows to integrate an on-the-fly interpreter (or compiler) into the editor of a language by means of the Xtext Builder Participant extension point. In our case, since we adapt a transformation itself based on provided directives in a mapping definition, the interpreter is thus known as a Higher-Order Transformation or HOT. To extend this extension point, it is necessary to register a builder participant in the *plugin.xml* file. By default, the XTEXT generator will create this registration in the UI-related project for the MetaModMap language as in Listing 4.18.

Listing 4.18: metamodemap.dsl.ui Plug-in: *plugin.xml*

```
1 <plugin>
2   <extension point="org.eclipse.xtext.builder.participant">
3     <participant
4       class="MetamodmapExecutableExtensionFactory:IXtextBuilderParticipant">
5     </participant>
6   </extension>
7 </plugin>
```

In Listing 4.18, the generated factory is used to instantiate the participant for the language. XTEXT also provides a default implementation for the IXtextBuilderParticipant interface, i.e. the BuilderParticipant class which declares a dependency of type IGenerator. Since the factory uses Guice to manage instantiations of dependencies, the concrete bindings should be declared in the Guice modules of the language as in Listing 4.19.

Listing 4.19: Concrete bindings in Guice modules of MetaModMap

```
1 /* contributed by GeneratorFragment */
2 public Class<? extends IXtextBuilderParticipant> bindIXtextBuilderParticipant() {
3   return org.eclipse.xtext.builder.BuilderParticipant.class;
4 }
5
6 public Class<? extends org.eclipse.xtext.generator.IGenerator> bindIGenerator() {
7   return MetamodmapGenerator.class;
8 }
```

The MetamodmapGenerator class is the Generator artifact, which is generated automatically by the XTEXT generator. The customization of this class is realized by adding an interpreter into the API method *doGenerate()* declared in type IGenerator. Listing 4.20 shows the implementation of that method.

Listing 4.20: Integration of a HOT interpreter for MetaModMap language

```
1 public class MetamodmapGenerator implements IGenerator {
2     public void doGenerate(Resource resource, IFileSystemAccess fsa) {
3         MetaModMapRoot mmmRoot = (MetaModMapRoot) resource.getContents().get(0);
4         Transformation transformation = mmmRoot.getTransformation();
5         /* adapt incrementally the model of transformation */
6         for (MapRule mapRule : mmmRoot.getMapRules()) {
7             Interpreter.doRebind(mapRule, transformation);
8         }
9         /* TODO : serialize the adapted model of transformation */
10        ...
11    }
12 }
```

The XTEXT builder participant has defined an usage scenario for a generator. In that scenario, the method *doGenerate()* will be invoked when an user commits (saves) a valid mapping definition (i.e. a mapping without errors spawned during the validating process). The caller (i.e. the builder participant) passes the whole model (i.e. parsed editing content) under the structure of EMF Resource to the receiver (i.e. the custom generator). From the passed resource, the root object of the model can be achieved as in line 3 of Listing 4.20, and so on for the original transformation as in line 4. Once having the transformation model, we iterate all mapping rules defined for each pair of metamodels, then invoke the *doRebind()* method, cf. Algorithm 5 in Chapter 3, to perform transformation adaptation in an incremental manner. Note that for the sake of simplicity, we implemented the HOT method (*doRebind()*) as static in the Interpreter class. The final step is to serialize the adapted model of transformation into a persistent file respecting the concrete syntax of the MOMENT2-GT language. This can be done by means of Xtext Serializer and, by default the adapted transformation file is saved in the same location than the original transformation file.

4.4 Transformation Adaptation in Action

The complete proposed solution has been implemented into a prototype² to support our approach to transformation adaptation.

2. Source code of the prototype, screen shot, metamodels, transformations of the motivating example and a demonstration video are available to download at <http://perso.telecom-bretagne.eu/quyetpham/software/>.

4.4.1 MetaModMap Prototype

The current version of the prototype is integrated into Eclipse as plug-ins to work with framework MOMENT2-GT with an additional textual editor (cf. Fig. 4.12, the DSL is developed by means of XText) which allows users to define intentional semantic correspondences between alternative metamodels.

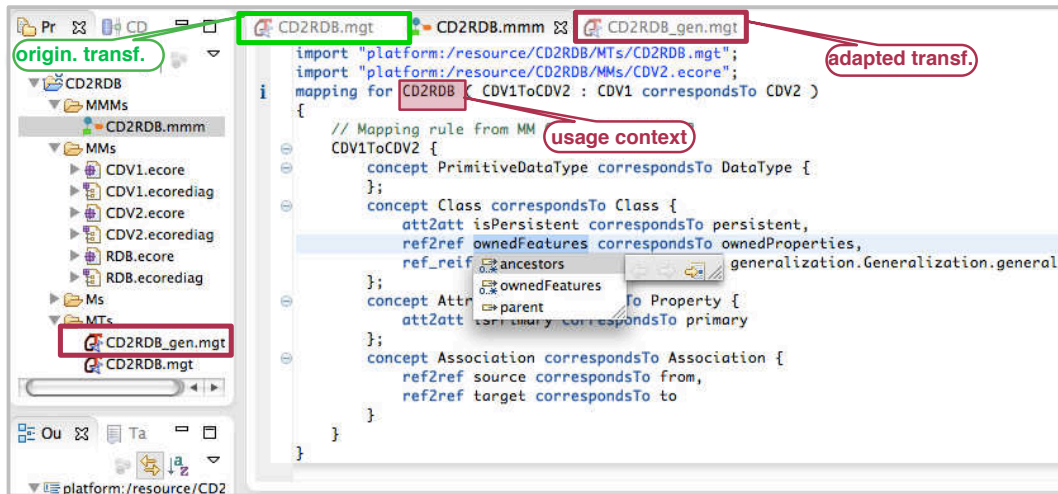


Figure 4.12: MetaModMap editor.

The prototype provides two main functionalities. First, an interactive warning-suggestion system is provided to give informative feedback to users when they type on the editor. This system implements the *model type compatibility check semantics* that is theorized in the Chapter 3. Second, an *interpreter* is injected within the editor to perform on-the-fly transformation adaptations (or HOTs). That is with a valid mapping definition, a legacy transformation definition written in the MOMENT2-GT language can be adapted to a new one for permanent use with another variant of source metamodel. These functionalities are the most essential requirements in nowadays modern editors.

4.4.2 Customer Management Case Study

Figure. 4.13 illustrates a scenario in which we need a transformation adaptation toolkit to obtain target models from source models conforming to two metamodels of the same domain (the same viewpoint of application systems). In this scenario we assume that a software company, i.e. Company A, uses a design tool for a certain viewpoint in the application development process. The design tool is based on a particular metamodel — is called the original source metamodel, i.e. models produced by means of that tool is persisted in respect of the format defined by the metamodel, cf. step (1) in the figure. The development process of the company is automated by means of the execution of a transformation definition to obtain target models that represent another viewpoint of systems. Although working on the same

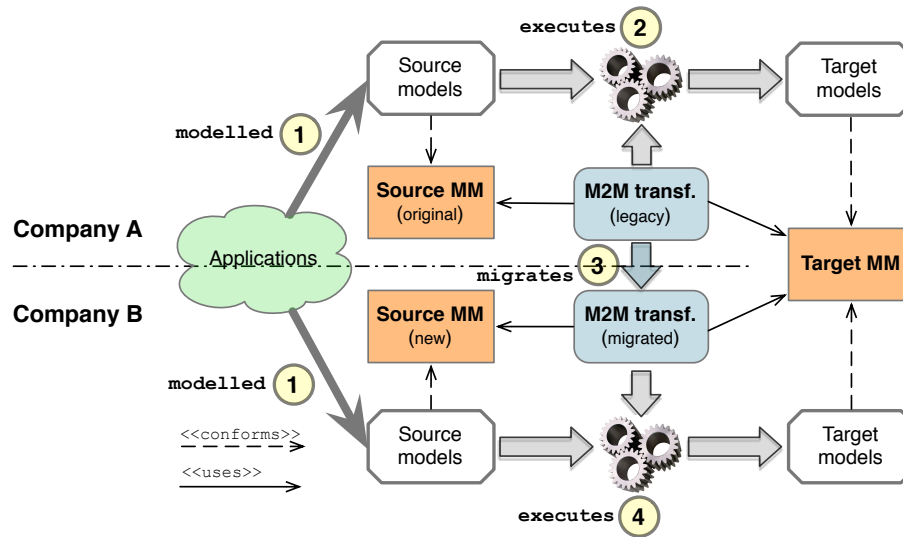


Figure 4.13: A scenario shows the need of a transformation adaptation toolkit.

source viewpoint, another company, i.e. Company B, might use a different design tool based on a similar metamodel — we call the new source metamodel. Due to the difference in the format defined by two metamodels, the legacy transformation definition cannot be used directly for Company B’s own source models. In order to reuse knowledge that has been encoded in that transformation definition, an adaptation for the transformation, cf. step (3) in the figure, can be carried out by means of the MetaModMap toolkit. In the following, we will demonstrate this scenario through the *Customer Management* example inspired by [Bézivin 05a] as a validation of our toolkit prototype.

To validate the MetaModMap toolkit prototype, one can imagine the class diagram view of the simple *Customer Management* application are modelled by means of different CASE tools which are based on different metamodels. However, for the sake of simplicity, these models are designed by using the same reflective model editor tool, i.e. Reflective Ecore Model Diagram Editor, thus they are persisted in the portable format XML. The resulting two models are shown partially in Fig. 4.14 (at middle-left and middle-right), one can see their complete representation in the UML object diagram notation in Figure 2.15a and Figure 2.18 of Section 2.3.1 in Chapter 2. The input model in the first representation is transformed into the corresponding relation database model through a transformation definition, cf. Appendix 1.1, for some further processing. We expect, when the design tool is switched to another one, the second representation is safely used to output the same identical relation database model as the output of the legacy transformation.

However, the same *intentional input model*, cf. Fig. 4.14 (at top), has two different representations with renamings such as $\{(PrimitiveDataType/DataType)\}$, $\{(Attribute/Property)\}$, $\{(isPrimary/primary)\}$, $\{(isPersistent/persistent)\}$, $\{(ownedFeatures/ownedProperties)\}$, $\{(source/from)\}$, $\{(target,to)\}$ and a relation reification $\{(parent/generalization.Generalization.general)\}$. These changes make the transformation

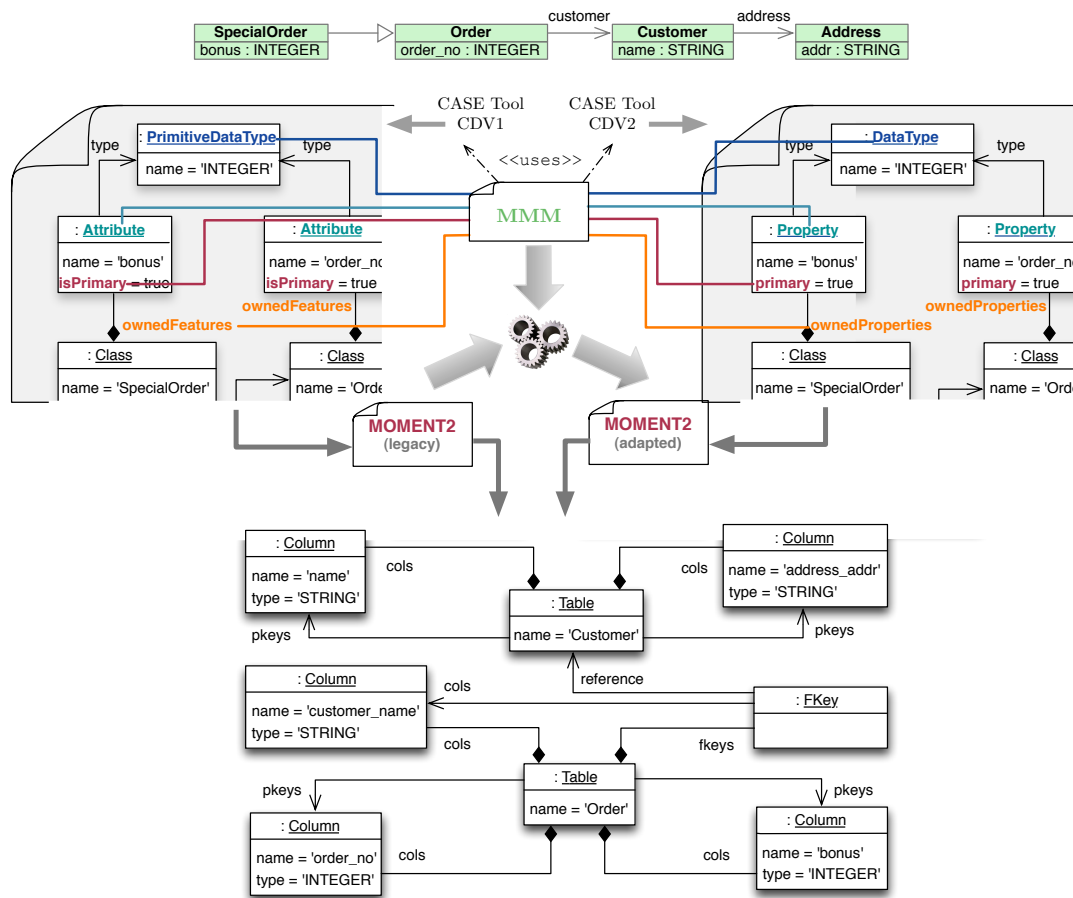


Figure 4.14: CD2RDB transformation's in/out models.

engine cannot access to the model represented in the new format to achieve and then compute values in slots of instance objects if the engine uses directly the legacy transformation definition. Due to these obstacles, the legacy transformation needs to be adapted. This work can be done by using the MetaModMap toolkit. Given a MetaModMap definition, as shown in Fig. 4.12, the new transformation can be generated automatically by the MetaModMap interpreter, cf. Appendix 1.2 for the adapted transformation definition. We obtain, as expected, *the same output model* shown in Fig. 4.14 (at bottom) for both two representations of the intentional input model, applying respectively the legacy transformation and the generated one.

4.5 Comparison to Transformation Reuse Approaches

Transformation reuse through adaptation in response to metamodel changes had been studied with two main different strategies. In our publication [Pham 12], we adopted in the first one that considers transformations as white-boxes and transforms them. The second strategy regards transformations as black-boxes and generates

adaptors for the transformation without modifying the original one.

Within the first strategy, following the *concept* mechanism presented in [de Lara 10], i.e. a specific transformation for a particular metamodel can be generated from a generic transformation defined on a conceptual metamodel and a strict structural binding between the particular and the conceptual metamodels, Sanchez Cuadrado *et al.* [Cuadrado 11] and Wimmer *et al.* [Wimmer 11] have extended the binding DSL for structural heterogeneities in independent metamodels. The adaptations in [Wimmer 11] are automatically either generated under the form of *helpers* provided by ATL or directly adapted in the specific transformation from the binding. Due to the lack of *helpers* in MOMENT2-GT, only model patterns must be adapted themselves in our approach. Nevertheless, with the support of variation points in model patterns, the adaptation process does not need to take into account *meta-attributes* on Ecore-based relations between types such as: *ordered*; *lowerBound*, except *upperBound* which is needed a special treatment. Related to the expressiveness, the number of binding/correspondence patterns provided in [Wimmer 11] is more than our current DSL version (actually, other patterns have been being developed), however, the abstraction level is actually the same. In comparison with these works, the substitution between metamodels is defined clearly (and formally) in our approach by simple graph structure TGM coupled with its effective inference mechanism on transformations. This graph structure is used to implement the *scoping and validating aspect* of the DSL editor, which provides a more interactive warning-suggestion system to users when they define correspondences.

In contrast to our approach and those proposed in [Cuadrado 11, Wimmer 11], i.e. transformation reuse for metamodels with different history, approaches presented in [Mendez 10, Levendovszky 10, Ruscio 11] focus on metamodel/transformation co-evolution. In [Mendez 10], Mendez *et al.* present a classification of typical changes and their impact on transformations when metamodels evolve while Levendovszky *et al.* [Levendovszky 10] and Di Ruscio *et al.* [Ruscio 11] propose adaptation solutions against these changes for particular transformation languages. The approach proposed in [Levendovszky 10] is also dedicated for graph rewriting-based transformation paradigm, implemented in GReAT³. However, the principle used for transformation reuse is not based on the isomorphism of type graphs as ours, but on an analysis of minor evolutions in metamodel. Therefore, when a relation and a type representing the same modeling concept as the pair *parent/generalization.Generalization.general* in our example, the corresponding original model pattern cannot be evolved automatically by their evolver tool, by opposition with our treatment which performs adaptation since the adapted model pattern still preserves, at some level, the same graph topology than the original one.

Authors in [Ruscio 11] propose EMFMigrate⁴ as a unified approach to the metamodel co-evolution problem with dependent artifacts such as models, transformations and tools. For ATL transformations, they introduced a coupled DSLs which encompasses the specification of metamodel differencing (EDelta) and (customizable) mi-

3. <http://www.isis.vanderbilt.edu/tools/GReAT>

4. <http://www.emfmigrate.org/>

gration rules (ATLMigrate). EDelta models are actually the recovered trace input by users to specify step-by-step metamodel evolution. Migration rules might be applied if conditional patterns are matched on the difference model. Then, the information extracted from the matched data will be used in rewriting rules to migrate transformations. In comparison with their work, our approach is not based on metamodel evolution traces, but on the observation of metamodel representation differences.

The second strategy considers a transformation as a black-box. Kerboeuf *et al.* [Kerboeuf 11] present an adaptation DSL which handles copy or deletion of elements from a model conforming to a new metamodel to make it become an instance model of the input metamodel of an endogenous transformation tool. The adaptation transformation also provides a trace to recover the output result of the transformation to an instance of the new metamodel.

Beside to above strategies, Sen *et al.* proposed in [Sen 10] another approach to make a transformation reusable by extending a metamodel based on the aspect weaving feature provided in Kermeta⁵. As opposed to our approach allowing users to define metamodel correspondences at a high-level and performing automatically adaptations, the adaptations in this approach are added by users at the transformation language level. This approach uses a metamodel pruning mechanism [Sen 09] to reduce adaptation requirements via aspect weaving for satisfying the model typing principle in [Steel 07]. The whole approach is similar to our solution, however, in the type point of view, we raise the substitution relation between metamodels up to TGM graphs with a bit different treatment in multiplicity depending on the way model patterns are used in MOMENT2-GT.

To facilitate comparing our contributions with these approaches, we take the summary table of related work from Section 2.4 at the end of Chapter 2 and extend a row with our published approach. The comparison is given in Table 4.2.

5. www.kermeta.org

	Reusing Model Transf. across Heterogen. MMs [Wimmer 11] [Cuadrado 11]	Semi-automation Evol. of DSML Model Transf. [Levendovszky 10] [Narayanan 09]	Managing Coevolution in MDE [Ruscio 11]	A DSML for Reversible Transformations [Kerboeuf 11] [Babau 11]	Model Adaptation by Precise Detect. of MM Changes [Garcés 09]	Reusable Model Transformations [Sen 10] [Steel 07]	Adapt. of Transf. based on Type Graph with Multi. [Pham 12]
Adaptation Strategy							
<i>transf.</i>	✓	✓	✓	NO	NO	NO	✓
<i>models</i>	NO	NO	NO	✓	✓	NO	NO
<i>metamodels</i>	NO	NO	NO	NO	NO	✓	NO
Mapping							
<i>Cardinality</i>	1:1 1:n	1:1 1:n	1:1 n:1	1:1	1:1	1:1	1:1
<i>Total vs. Partial</i>	Total	Total	Total	Total	Total	Partial	Partial
<i>Element vs. Structure</i>	Element and several structural supports	Element supports	Element supports	Element and several structural supports	Element and structural supports	Element and several structural supports	Element and several structural supports
<i>Constraint</i>	Multiplicities contraction, properties filtering	Without multiplicities contraction support	Without multiplicities contraction support	Multiplicities contraction (lost information)	Without multiplicities contraction support	Without multiplicities contraction support	Multiplicities contraction (depend on usage context)
Transformation Language							
<i>Paradigm</i>	Hybrid (decl.& imp.)	Graph rewriting-based	Hybrid (decl.& imp.)	Any	Any	Imperative with AOP supports	Graph rewriting-based
<i>Language</i>	ATL	GReAT	ATL	–	–	Kermeta	MOMENT2-GT
Application Scenarios							
<i>M2M Transf.</i>	✓ (Source MM change)	✓ (Source MM change)	✓ (Source MM change)	✓ (Source MM change)	✓ (Source MM change)	✓ (Source, Target MM change)	✓ (Source MM change)
<i>In-place Up.</i>	NO	NO	NO	✓	NO	✓	NO
Declaration Automation							
<i>explicit</i>	✓ (without checking)	✓ (without checking due to semi-automation)	✓ (without checking due to semi-automation)	✓ (without checking)	✓ (user intervention but without checking)	✓ (a priori checking)	✓ (a priori checking)
<i>implicit</i>	NO	NO	NO	NO	✓ (heuristic infer.)	NO	NO

Table 4.2: Comparison to other transformation reuse approaches

4.6 Prototype Summary and Discussion

In this chapter, we have described a prototype of a transformation migration toolkit, named MetaModMap, which provides supports for transformation reuse through adaptation with user-defined directives. The toolkit has been developed to work with model-to-model transformations written in the MOMENT2-GT language.

The toolkit was implemented as Eclipsed-based plug-ins that includes 1) a *textual editor* used to define semantic correspondences between metamodels written in a DSL language, i.e. MetaModMap, 2) a *Higher-Order-Transformation interpreter* used for automatically migrating an existing transformation definition according to a valid mapping definition defined in the provided editor. For the MetaModMap editor, we have presented how to develop a modern editor that allows users to work with a language at the abstract syntax representation level by using the XTEXT language development framework. Such an editor can provide on-the-fly features such as *scoping* and *validation* which are related to the integration of our graph-based typing approach presented in Chapter 3 into practice. For the interpreter of the language, we have shown how an on-the-fly transformation adaptation engine can be integrated into the editor by means of the code generator API provided by XTEXT. By such integrations, end-users are provided with an easy to use transformation migration toolkit within Eclipse. Thereby, they can use our toolkit for improving transformation re-usability in incorporating others EMF facilities for creating, managing models, metamodels, and specially with MOMENT2-GT framework for defining and running transformations as a complete MDE ecology.

At the end of this chapter, the use of the provided toolkit is illustrated with a case study. As an experiment validation, we have shown a scenario in which our toolkit is applicable. In the scenario, a transformation written to transformation class models into relational database models should be adapted when changing design tools based on metamodel variants for input-models. This task can be done by our toolkit that is able to be aware of how to perform adaptations on the legacy transformation, from very simple directives input by the user.

Chapter 5

Conclusion

Contents

5.1	Contribution Summary	157
5.2	Limitations	160
5.3	Perspectives	161

In this thesis we investigated an automatic transformation adaptation approach to improve the reuse of knowledge encoded in existing metamodels-based transformation definitions, in response to differences occurred when changing input metamodels. Our approach is developed upon a theory of graph-based model typing to address the transformation reuse problem without adaptation and extends with a DSL that acts as a higher-order transformation for overcoming strict model subtypes defined in the proposed theory. The following sections in this chapter will summarize contributions in our work, discuss the current limitations and propose some further investigations.

5.1 Contribution Summary

In most current MDE ecologies, everything is a model and model transformation is a crucial activity for automatically evolving or translating models among modelling languages. In such MDE environments, the abstract syntax of modelling languages is often chosen as basis to specify transformations, also called *metamodel-based transformation* approach [Levendovszky 02]. Using such approach permits specification of transformation rules focused on the structural (syntactical) translation. Thus a transformation can be applied to models that structurally conform to the base metamodels. However, since transformation definitions are tightly coupled with metamodels which are currently often used to denote some kind of “model type”, they become brittle when a base metamodel evolves or is replaced by another one because of the strength of this definition of “type”. In short, using metamodels directly as model types and the conformance relation between models and metamodels hinder the transformation reuse.

In order to address this issue, many approaches have recently been proposed to introduce a typing principle to models and also dependent artifacts. For the particular interest to this dissertation, i.e. transformation reuse, we propose a graph-based typing approach of models that derives benefits of the natural graph form of models and solves the multiplicity issue of associations. From this definition, we define the type of model transformations. That means a transformation should be *model type*-inferred based on rules which make up the transformation. As an example of the use of this type system, we propose a metamodel mapping language to improve the scope of transformation reuse in case of metamodel heterogeneities. Concretely, our work has following contributions in both theoretical and practical aspects.

Theoretical Contribution

From a theoretical point of view, our work constitutes a *type system*, in which a new notion of *model type* based on the graph topology of metamodels is at the core of the solution for the transformation reuse problem. More precisely, for a MOF-like metamodeling approach, i.e. EMF Ecore, we have proposed a graph structure representation, so-called Type Graph with Multiplicity — TGM, as another abstraction of Ecore-defined metamodels.

The main use of TGMs is to exhibit the graph topology of metamodels by flattening the *inheritance* relation between type concepts. This notion of model type, as its name, takes into account the *upperBound* meta-attribute that is defined on relations between type concepts. However, this consideration is derived upto a higher level of abstraction for multiplicity property, i.e. a higher graph topology point of view with three values: $Multiplicity \equiv \{single_valued, multi_valued, undefined_valued\}$. As explained in Section 3.3.1, the selection of the *upperBound* meta-attribute is only an example of interesting semantics variation points that we want to add into the model type definition, depending on what level of abstraction that we want to develop for a transformation language. Every metamodel in the Ecore space is leveraged into the more abstract space TGM by means of a *model typing* function, i.e. a particular exogenous transformation, provided in our type system.

From the graph-based model type, existing transformations themselves can be typed by means of our *transformation typing* function. Such typing function infers user-defined graph patterns written in MOMENT2-GT language to select real used parts of declared model types, and changes TGM-Multiplicity values if a semantics variation point is found. Therefore, the knowledge encoded in the transformation can be captured from the outputs of the type inference process, i.e. *effective* TGM model types.

Substitutability between metamodels is defined based on TGM model types, thanks to a name-based graph inclusion. In taking the usage context of metamodels into account, i.e. the transformations defined upon metamodels, the effective TGM model types are used to compare with TGM model types of new metamodels. As a result, all metamodels whose TGM view is a super-graph of the effective TGM of a declared metamodel could be replaced in the transformation without a need of

adaptation. These cases are in the *isomorphic* transformation reuse sub-problem.

In order to extend the re-usability in cases of *non-isomorphic* transformation reuse, i.e. metamodels are more heterogeneous such as elements named differently or partial difference in topology, we provided a simple mapping DSL language, namely MetaModMap. The language is dedicated to identify semantic correspondences between metamodel elements. Moreover, the language also supports several simple combinations of elements in case of partial differences in topology. The operational semantics of the language acts as a Higher-Order Transformation to adapt the original transformations, certainly in the scope of our supports. The proposed type system is integrated as a part of the language when it is used as primary premises to validate a mapping definition before the execution of a transformation adaptation. In using a mapping definition, it yields more efficient reuse of transformations as well as knowledge that is encoded in those transformations. Nevertheless, mapping definitions have to be provided by the experts who develop metamodels and domain-specific languages.

In summary, in our theoretical contribution, we have given a general solution for the problem of transformation reuse, with and without adaptation. The type system and the mapping DSL language have been realized and integrated in a concrete MDE environment, i.e. MOMENT2-GT framework, which is based on EMF Eclipse.

Practical Contribution

From a practical point of view, we have extended the MOMENT2-GT framework by providing a prototype of a transformation migration toolkit, i.e. an implementation of our model typing approach and the MetaModMap language. The toolkit has been developed to provide support for reusing transformations written in the MOMENT2-GT language.

The toolkit constitutes of Eclipse-based plug-ins that implement:

- A *textual editor*, used to define semantic correspondences between alternative metamodels.
- A *Higher-Order Transformation interpreter*, injected into the editor which is used for automatically migrating a transformation according to a mapping.

For the MetaModMap editor, we have developed it as a modern editor that allows users to work at the abstract syntax representation of the language. To this end, we have used technologies provided in the XTEXT workflow to develop the language and shown how to use them in this dissertation. Our editor is fully integrated with the proposed type system with on-the-fly features, i.e. *scoping* and *validation*, to detect early inconsistent mappings. This integration shows the usability of our typing approach to give informative feedback to users when they provide additional directives in the editor of the mapping language.

For the interpreter, we have developed and integrated it into the editor as an on-the-fly transformation adaptation engine. This has been done by using the code generator API provided by XTEXT. Such integration allows users to easily use our

toolkit within Eclipse without a need of performing a complex running step, just typing a mapping and obtaining an adapted transformation when the mapping is committed.

Summing up, by extending the MOMENT2-GT framework, which is developed upon EMF, we provide an additional functionality for an MDE ecology in which models, metamodels are created, managed by EMF facilities, transformations are defined and executed by MOMENT2-GT and in addition these transformations could be adapted by our toolkit for improving their re-usability.

5.2 Limitations

Our solution for the transformation reuse problem has several limitations. First, our proposed type system is currently limited to cases in which *source* metamodels of an exogenous transformation are replaced. Second, the proposed mapping DSL is not expressive enough for complex heterogeneities between metamodels' elements and currently supports only one-to-one mappings and has several simple correspondences between combinations of elements. Third, the whole approach is dependent on a specific transformation paradigm, i.e. graph rewriting-based transformation languages without any supports of navigation helpers.

Reusing transformation when changing source metamodels

The approach presented in this work especially focuses on the reuse of transformations when their source metamodels are changed. In respect to this objective, the centric notion of model type, i.e. TGM, as well as typing functions in the type system are constituted by eliminating non-affected meta-attributes when changing source metamodels. This strategy could be realized with an assumption that there are no object creations in models of the domain defined by a source metamodel, as shown in this dissertation. Other cases in which the target metamodel or both metamodels change need further investigation. Nevertheless, we believe that the same reuse principle, i.e. abstracting metamodels; inferring effective model types; using a mapping DSL; adapting transformations, could be used.

Expressivity of the mapping DSL

Based on the graph similarity, the mapping DSL language in this work provides some simple correspondence patterns. These are one-to-one mappings for naming differences of classes, attributes and references. Some patterns for a bit complex cases are also supported, e.g. attribute and reference reification by adding intermediate classes. However, the real world is not simple like that. In industrial case studies, metamodels are more heterogeneous in the graph topology point of view. Therefore, these patterns as well as their adaptation semantics proposed in the current mapping language are not expressive enough to employ in the real world. Thus, the applicability of the language is limited.

Dependency on the graph rewriting-based transformation paradigm

The graph rewriting-based transformation paradigm, concretely, MOMENT2-GT the language used in this dissertation, allows to make transformation descriptions more declarative and representative. More precisely, transformation writers are equipped with adequate constructs to describe a transformation by thinking about the graph topology of metamodels, which is expressed in an concrete object-oriented syntax, e.g. the UML class diagram notation. This characteristic leads us to propose a notion of metamodel similarity based graph similarity, then a substitution possibility between similar metamodels. Instead, all elements in our approach must conform to the graph notion, from the abstract model types, model subtype relation to typing functions as well as the mapping language and its adaptation semantics. This yields that our approach is dependent on graph-related constructs provided by a transformation language, which is based on the graph rewriting-based transformation paradigm.

5.3 Perspectives

In this dissertation, we have presented an approach for reusing a transformation that has been written upon a certain source metamodel, for a similar source metamodel. The approach can automatically migrate the transformation accordingly correspondences between the two source metamodels that user needs to specify. The approach is based on a theory of graph-based model typing. On critically reflecting our approach, two main points remain for future work.

Making the proposed model typing approach more complete

Chapter 3 presents a graph-based model typing approach to support the definition of metamodel similarity as graph-based model type similarity. We then show that for the MOMENT2-GT model transformation language, the effective model type for a source metamodel can be derived, which can be used instead of the source metamodel to determine if the transformation can be applied to a given model. This directly increases the re-usability of the model transformation, but only for strict subtypes between source metamodels.

As aforementioned in the section on limitations of our approach, the proposed type system needs to be enriched to support the change of target metamodels and those, which play a role as both source and target, e.g. an endogenous/in-place transformation. To this end, the information, which is included in the TGM model types must be enriched regarding the containment relation between type concepts defined in metamodels, i.e. taking the boolean value of the *containment* meta-attribute which is defined by a designer on an Ecore reference definition into account. This consideration is due to the fact that MOMENT2-GT does not allow an object creation outside the containment relation between two type concepts. As a consequence, the effective model type inference for target metamodels needs to be aligned the new

modified notion of model types accordingly. However, such a modification on the notion of model types makes the type notion less abstract, hence the algorithm for inferring effective model type become more complex.

Improving the expressiveness of the mapping DSL

The proposed type system becomes more powerful on the transformation reuse problem when only it is coupled with a mapping DSL that performs automatic adaptation on transformations. Actually, a mapping DSL for transformation migration acts as an interpreter's parts that perform dynamic bindings from a generic type to a concrete type in advanced programming language. That is why such a mapping DSL is also called as a Higher-Order Transformation (HOT) language. However, the dynamic binding semantics of HOT support more complex bindings in terms of user-defined directives than in programming languages. Hence, more patterns for user-defined directives makes the mapping language more expressive from the end-user point of view.

From above argument, the expressiveness of the mapping DSL, i.e. MetaModMap, is necessary to be improved by extending our set of correspondence patterns to support other kinds of metamodel heterogeneity, e.g. splitting type concepts, based on our graph-based typing approach. That is, correspondences defined by users in our DSL are considered as zooms (in/out) on single nodes, single edges or minor connected sub-graphs of the effective TGM as long as the isomorphism between TGM views of metamodels is warranted.

Bibliography

- [Agrawal 03] Aditya Agrawal, Gabor Karsai & Feng Shi. *Graph Transformations on Domain-Specific Models*. Technical Report Mic, Institute for Software Integrated Systems, Vanderbilt University, Nashville, 2003. 31
- [Akehurst 02] David Akehurst & Stuart Kent. *A Relational Approach to Defining Transformations in a Metamodel*. In Jean-Marc Jézéquel, Heinrich Hussmann & Stephen Cook, editors, Proceedings of the 5th International Conference on the Unified Modeling Language, volume pp of LNCS, pages 243–258. Springer-Verlag, 2002. 31
- [Akehurst 10] D H Akehurst, W G J Howells & K D McDonald-Maier. *Kent Model Transformation Language*. Science of Computer Programming, vol. 68, no. 3, pages 214–234, 2010. 31
- [Ambler 12] Scott W. Ambler. *Agile Modeling (AM) Home Page: Effective Practices for Modeling and Documentation*, 2012. 1
- [Andries 99] Marc Andries, Gregor Engels, Annegret Habel & Berthold Hoffmann. *Graph transformation for specification and programming*. Computer Programming, vol. 6423, no. 98, 1999. 31
- [AndroMDA 04] AndroMDA. *AndroMDA*, 2004. 33
- [Babau 11] Jean-philippe Babau & Mickaël Kerboeuf. *Domain Specific Language Modeling Facilities*. In 5th MoDELS workshop on Models and Evolution, pages 33–43, 2011. 57, 61, 62
- [Bast 05] Wim Bast, Mariano Belaunde, Xavier Blanc, Keith Duddy, Catherine Griffin, Simon Helsen, Michael Lawley, Michael Murpree, Sreedhar Reddy, Shane Sendall, Jim Steel, Laurence Tratt, R Venkatesh & Didier Vojtisek. *Mof qvt final adopted specification*. Object Management Group (OMG), 2005. 28, 31, 32
- [Bezevin 05] Jean Bezevin. *On the Unification Power of Models*. Software and Systems Modeling, pages 1–31, 2005. 2, 17, 20, 24
- [Bézivin 03a] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette & Jamal Eddine Rougui. *First experiments with the ATL model transformation language: Transforming XSLT into XQuery*. In 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture, numéro 1, page 50. Citeseer, Citeseer, 2003. 28, 32

- [Bézivin 03b] Jean Bézivin, Nicolas Farcet & Damien Pollet. *Reflective Model Driven Engineering*. pages 175–189, 2003. 8
- [Bézivin 04] Jean Bézivin. *In Search of a Basic Principle for Model Driven Engineering*. vol. V, no. 2, pages 1–5, 2004. 16
- [Bézivin 05a] J. Bézivin, Bernhard Rumpe, A. Schurr & Laurence Tratt. *Model transformation in practice workshop announcement*. In Satellite Events at the MoDELS 2005 Conference, pages 120–127, 2005. 12, 150
- [Bézivin 05b] Jean Bézivin, Frédéric Jouault, Peter Rosenthal & Patrick Valduriez. *Modeling in the Large and Modeling in the Small*. In Uwe Aßmann, Mehmet Aksit & Arend Rensink, editors, Proceedings of the European MDA Workshops: Foundations and Applications, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer, 2005. 31
- [Booch 94] G Booch. *Object oriented analysis and design with applications*. 1994. 9
- [Booch 04] Grady Booch, James Rumbaugh & Bran Selic. *An MDA Manifesto*. MDA Journal, vol. 2004, no. April, pages 1–9, 2004. 19
- [Boronat 07] Artur Boronat. *MOMENT: A Formal Framework for MOdel management*. PhD thesis, 2007. 43, 174
- [Boronat 09] A. Boronat, R. Heckel & J. Meseguer. *Rewriting logic semantics and verification of model transformations*. *Fundamental Approaches to Software Engineering*, pages 18–33, 2009. 3, 5, 40, 41, 42, 44
- [Braun 03] Peter Braun & Frank Marschall. *BOTL-the bidirectional object oriented transformation language*. Technical report, Technische Universität München 85748, München, Germany, 2003. 31
- [Bull 05] RI Bull & Jean-Marie Favre. *Visualization in the context of model driven engineering*. In Proceedings of the MoDELS’05 Workshop on Model Driven Development of Advanced User Interfaces, 2005. 23
- [Chen 89] Minder Chen, J.F. Nunamaker Jr & E.S. Weber. *Computer-aided software engineering: present status and future directions*. *ACM SIGMIS Database*, vol. 20, no. 1, pages 7–13, 1989. 9
- [Compuware 05] Compuware. *OptimalJ 4.0, User’s Guide*, 2005. 30, 33
- [Cuadrado 11] Jesu Sanchez Cuadrado, Esther Guerra & Juan de Lara. *Generic model transformations: write once, reuse everywhere*. In Proceeding of the 4th International Conference on Model Transformation, pages 62–77, 2011. 13, 57, 58, 152
- [Czarnecki 00] Krzysztof Czarnecki & Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley, 2000. 10
- [Czarnecki 03] Krzysztof Czarnecki & Simon Helsen. *Classification of Model Transformation Approaches*. *Architecture*, pages 1–17, 2003. 23, 27

- [Czarnecki 05] Krzysztof Czarnecki & Michal Antkiewicz. *Mapping Features to Models : A Template Approach Based on Superimposed Variants Background : Feature Modeling*. In Generative Programming and Component Engineering, pages 422–437. Springer, 2005. 31
- [Czarnecki 06] Krzysztof Czarnecki. *Feature-based survey of model transformation approaches*. IBM Systems Journal, vol. 45, no. 3, pages 621–645, 2006. 22, 23, 24, 25, 26, 27, 29, 30, 174
- [de Lara 10] J. de Lara & Esther Guerra. *Generic meta-modelling with concepts, templates and mixin layers*. Model Driven Engineering Languages and Systems, pages 16–30, 2010. 13, 57, 58, 152
- [Ehrig 06] H. Ehrig, K. Ehrig, U. Prange & G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. March 2006. 41, 50
- [EMF 04] EMF. *Eclipse Modeling Framework (EMF)*, 2004. 22, 36
- [Favre 04] J.-M. Favre. *CaCOphoNy: metamodel-driven software architecture reconstruction*. In 11th Working Conference on Reverse Engineering, pages 204–213. IEEE Comput. Soc, 2004. 23
- [Forward 08] Andrew Forward. *Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals*. workshop on Models in software, pages 27–32, 2008. 1
- [France 07a] Robert France & Bernhard Rumpe. *Model-driven Development of Complex Software: A Research Roadmap*. Future of Software Engineering (FOSE '07), pages 37–54, May 2007. 1
- [France 07b] Robert France & Bernhard Rumpe. *Model-driven Development of Complex Software: A Research Roadmap*. Future of Software Engineering (FOSE 07), pages 37–54, May 2007. 9, 10
- [Fujaba 97] Fujaba. *Fujaba Tool Suite*, 1997. 31
- [Garcés 09] Kelly Garcés, Frédéric Jouault, Pierre Cointe & Jean Bézivin. *Managing model adaptation by precise detection of metamodel changes*. Model Driven Architecture - Foundations and Applications, vol. 2, pages 34–49, 2009. 62
- [Gardner 03] Tracy Gardner, Catherine Griffin & Jana Koehler. *A review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard*. In MetaModelling for MDA, pages 178–197, 2003. 23, 24, 33
- [Gerber 02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel & Andrew Wood. *Transformation : The Missing Link of MDA*. In A Corradini, H Ehrig, H Kreowski & G Rozenberg, editors, Proceedings of the 1st International Conference on Graph Transformation, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2002. 31
- [GMF 11] GMF. *GMF - Graphical Modeling Framework* <http://www.eclipse.org/modeling/gmp/>, 2011. 10

- [Gray 06] J Gray, Y Lin & J Zhang. *Automating change evolution in model-driven engineering*. IEEE Computer, vol. 39, no. 2, pages 51–58, 2006. 31
- [Greenfield 03] Jack Greenfield & Keith Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools*. In Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 16–27, 2003. 1, 17, 19, 20, 22, 35, 173
- [Guy 12] Clément Guy, Benoit Combemale, Steven Derrien, Jim R H Steel & Jean-marc Jézéquel. *On Model Subtyping*. In 8th European Conference on Modelling Foundations and Applications, 2012. 96
- [Hevner 04] Alan Raymond Hevner, Salvatore T. March, Jinsoo Park & Sudha Ram. *Design science in information systems research*. MIS Quarterly, vol. 28, no. 1, pages 75–105, 2004. 3
- [IBM 05] IBM. *Model Transformation with the IBM Model Transformation Framework*, May 2005. 29, 31
- [Ivkovic 04] Igor Ivkovic & Kostas Kontogiannis. *Tracing Evolution Changes of Software Artifacts through Model Synchronization*. In ICSM '04 Proceedings of the 20th IEEE International Conference on Software Maintenance, pages 252–261. IEEE Computer Society, September 2004. 23
- [JAMDA 03] JAMDA. *Java Model Driven Architecture 0.2*, 2003. 33
- [JMI 02] JMI. *Java Metadata Interface (JMI)*, 2002. 30
- [Jouault 06] Frédéric Jouault & Ivan Kurtev. *Transforming Models with ATL*. In Jean-Michel Bruel, editor, Satellite Events at the MoDELS 2005 Conference, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2006. 28, 32
- [Kalnins 05] Audris Kalnins, Janis Barzdins & Edgars Celms. *Model transformation language MOLA*. In Proceedings of Model Driven Architecture: Foundations and Applications, volume 31, pages 62–76. Springer, 2005. 31
- [Kang 90] KC Kang, SG Cohen, JA Hess & WE Novak. *Feature-oriented domain analysis (FODA) feasibility study*. Technical report, echnical Report CMU/SEI-90- TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1990. 27
- [Karagiannis 06] Dimitris Karagiannis. *Metamodels in action: An overview*. IC-SOFT, 2006. 2
- [Kent 02] Stuart Kent. *Model Driven Engineering*. Engineering, pages 286–298, 2002. 2
- [Kerboeuf 11] Mickaël Kerboeuf & J.P. Babau. *A DSML for reversible transformations*. In 11th OOPSLA Workshop on Domain-Specific Modeling, pages 1–6, 2011. 1, 14, 4, 57, 61, 62, 153

- [Kleppe 03] Anneke Kleppe, Jos Warmer & Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley, 2003. 17, 23
- [Kuhne 06] Thomas Kuhne. *Matters of (Meta-) Modeling*. *Software and Systems Modeling*, vol. 5, no. 4, pages 369–385, July 2006. 2, 10, 11, 12, 16
- [Lara 02] Juan De Lara & Hans Vangheluwe. *AToM3: A Tool for Multi-formalism and Meta-modelling*. In Ralf-Detlef Kutsche & Herbert Weber, editors, *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin / Heidelberg, 2002. 31
- [Lawley 06] Michael Lawley & Jim Steel. *Practical Declarative Model Transformation with Tefkat*. In *Proceedings of Model Transformations in Practice Workshop, MoDELS Conference*, pages 139–150. Springer, Springer-Verlag Berlin Heidelberg, 2006. 31
- [Levendovszky 02] Tihamer Levendovszky, Gabor Karsai & Miklos Maroti. *Model reuse with metamodel-based transformations*. *Software Reuse*, 2002. 2, 157
- [Levendovszky 10] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan & Gabor Karsai. *A novel approach to semi-automated evolution of DSML model transformation*. *Software Language Engineering*, pages 23–41, 2010. 1, 14, 4, 57, 59, 60, 152
- [Marschall 03] Frank Marschall & Peter Braun. *Model Transformations for the MDA with BOTL*. In *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, volume *Enschede*, 2003. 31
- [Martí-Oliet 07] Narciso Martí-Oliet, José Meseguer, David Hutchison & John C Mitchell. *All About Maude - A High-Performance Logical Framework How to Specify Program and Verify*. 2007. 42
- [MDA 03] MDA. *Model-Driven Architecture Guide V1.0.1 03-06-01*, 2003. 1, 10, 17, 18, 19, 22, 173
- [Mendez 10] David Mendez, Anne Etien, Alexis Muller & Rubby Casallas. *Towards Transformation Migration After Metamodel Evolution*. In *International Workshop on Models and Evolution*, pages 84–89, 2010. 4, 152
- [Mens 06] Tom Mens & Pieter Van Gorp. *A Taxonomy of Model Transformation*. *Electronic Notes in Theoretical Computer Science*, vol. 152, pages 125–142, March 2006. 2, 5, 23, 24, 25, 26, 123, 177
- [MetaCase 00] MetaCase. *MetaCase - Domain-Specific Modeling with MetaEdit+*, 2000. 33
- [MetaCase 11] MetaCase. *MetaCase - Domain-Specific Modeling with MetaEdit+* <http://www.metacase.com/>, 2011. 10

- [MetaModMap 11] MetaModMap. <http://perso.telecom-bretagne.eu/quyetpham/software/>, 2011. 124
- [Microsoft 05] Microsoft. *Visual Studio 2005 Team System Modeling Strategy and FAQ*, 2005. 21
- [MOF 06] MOF. *Meta Object Facility (MOF) 2.0 Core Specification*, 2006. 10, 14, 19, 21
- [Moreira 04] Ana Moreira & Gustavo Rossi. *An Introduction to UML Profiles*. vol. V, no. 2, 2004. 19, 21
- [Muller 05] Pierre-Alain Muller, Franck Fleurey & Jean-Marc Jézéquel. *Weaving Executability into Object-Oriented Meta-Languages*. In Lionel C Briand & Clay Williams, editors, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, volume 3713 of *LNCS*, pages 264–278. MODELS/UML'2005, Springer, 2005. 31
- [MySQL 11] MySQL. *MySQL Database* <http://www.mysql.com>, 2011. 16
- [Narayanan 09] Anantha Narayanan, Tihamer Levendovszky & Daniel Balasubramanian. *Automatic Domain Model Migration to Manage Metamodel Evolution*. pages 706–711, 2009. 59
- [OCL 06] OCL. *Object Constraint Language OCL 2.0* <http://www.omg.org/spec/OCL/2.0/>, 2006. 14, 27, 31
- [OMG 00] OMG. *Object Management Group* <http://www.omg.org/>, 2000. 10, 14
- [OMG 05] OMG. *MOF 2.0/XMI Mapping Specification, v2.1*. Version 2.1. formal/05-09-01, no. September, 2005. 32
- [Oracle 11] Oracle. *Oracle Database* <http://www.oracle.com>, 2011. 16
- [Patrascoiu 04] Octavian Patrascoiu. *YATL: Yet Another Transformation Language*. In Proceedings of the 1st European MDA Workshop, pages 83–90. Citeseer, 2004. 32
- [Pham 12] Quyet-thang Pham & Antoine Beugnard. *Automatic Adaptation of Transformations based on Type Graph with Multiplicity*. In 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), volume 1, 2012. 151
- [Popma 05] Remko Popma. *Java Emitter Templates (JET) Tutorial*, 2005. 33
- [QVT-RFP 05] QVT-RFP. *Revised submission for MOF 2.0 Query/View/Transformation RFP*, 2005. 24
- [QVT 08] QVT. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, 2008. 10, 41
- [Rahm 01] Erhard Rahm & Philip a. Bernstein. *A survey of approaches to automatic schema matching*. The VLDB Journal, vol. 10, no. 4, pages 334–350, December 2001. 57
- [Rothenberg 89] Jeff Rothenberg. *The nature of modeling*. John Wiley & Sons, Inc., artificial edition, 1989. 10

- [Rumbaugh 99] J. Rumbaugh, I. Jacobson & G. Booch. The unified modeling language reference manual. Addison-Wesley Professional, 1999. 12
- [Ruscio 11] Davide Di Ruscio, Ludovico Iovino, Alfonso Pierantonio & Dipartimento Informatica. *What is needed for managing co-evolution in MDE ?* In Proceedings of the 2nd International Workshop on Model Comparison in Practice (IWMCP 11), pages 30–38, 2011. 14, 57, 60, 61, 152
- [Schmidt 06] Douglas C Schmidt. *Model-Driven Engineering*. Computer, no. February, pages 25–31, 2006. 9, 10, 17
- [Seidewitz Ed 03] Seidewitz Ed. *What Models Mean*. Ieee Software, pages 26–32, 2003. 2, 12, 13, 16
- [Sen 09] Sagar Sen, Naouel Moha, Benoit Baudry & Jean-Marc Jézéquel. *Meta-model pruning*. Model Driven Engineering Languages and Systems, vol. 215483, pages 32–46, 2009. 14, 63, 88, 153
- [Sen 10] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry & Jean-Marc Jézéquel. *Reusable model transformations*. Software & Systems Modeling, vol. 11, no. 1, pages 111–125, November 2010. 14, 57, 63, 153
- [Sendall 03] Shane Sendall & Wojtek Kozaczynski. *Model Transformation - The Heart and Soul of Model-Driven Software Development*. Development, no. August 2003, pages 1–12, 2003. 2, 23, 24
- [Solberg 05] Arnor Solberg & Robert France. *Navigating the MetaMuddle*. In Proceedings of the 4th Workshop in Software Model Engineering, numéro Mdd, 2005. 23
- [Stahl 06] Thomas Stahl, Markus Voelter & Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. May 2006. 22, 30
- [Steel 07] Jim Steel & Jean-Marc Jézéquel. *On model typing*. Software & Systems Modeling, vol. 6, no. 4, pages 401–413, January 2007. 14, 57, 63, 153
- [Sunye 01] Gerson Sunye, Damien Pollet, Yves Le Traon & Jean-Marc Jezequel. *Refactoring UML Models*. In Proceedings of the 4th International Conference, Unified Modeling Language Conference, Toronto, Canada, pages 134–148, 2001. 23
- [Swithinbank 05] Peter Swithinbank, Mandy Chessell, Tracy Gardner, Catherine Griffin, Jessica Man, Helen Wylie & Larry Yusuf. *Model-Driven Development Using IBM Rational Software Architecture*. IBM Red-Book, 2005. 17, 21
- [Sztipanovits 97] Janos Sztipanovits & Gabor Karsai. *Model-Integrated Computing*. Computer, pages 110–111, 1997. 1
- [Taentzer 04] Gabriele Taentzer. *AGG: A Graph Transformation Environment for Modeling and Validation of Software*. In John L Pfaltz, Manfred

- Nagl & Boris Böhlen, editors, Applications of Graph Transformations with Industrial Relevance, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2004. 28, 31
- [Taentzer 05] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan De Lara, Tihamer Levendovszky, Ulrike Prange & Daniel Varro. *Model Transformation by Graph Transformation: A Comparative Study*. In Proc. Workshop Model Transformation in Practice, Montego Bay, Jamaica, 2005. 31, 41
- [Thomas 03] Dave Thomas & Brian M. Barry. *Model driven development*. In Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '03, page 2, New York, New York, USA, October 2003. ACM Press. 1
- [Tratt 06] Laurence Tratt. *The MT model transformation language*. Proceedings of the 2006 ACM symposium on Applied computing SAC 06, vol. Dijon, Fra, page 1296, 2006. 32
- [UML 05] UML. *Unified Modeling Language 2.0* <http://www.omg.org/spec/UML/>, 2005. 14, 15, 21
- [Varró 02] D Varró, G Varró & A Pataricza. *Designing the automatic transformation of visual languages*. Science of Computer Programming, vol. 44, no. 2, pages 205–227, 2002. 28, 31
- [Varró 04] Dániel Varró & András Pataricza. *Generic and meta-transformations for model transformation engineering*. THE UNIFIED MODELING LANGUAGE. MODELLING LANGUAGES AND APPLICATIONS, pages 290—304, 2004. 28, 31
- [Visser 01] Eelco Visser. *A Survey of Rewriting Strategies in Program Transformation Systems*. Electronic Notes in Theoretical Computer Science, vol. 57, no. 2, pages 109–143, December 2001. 25
- [Vojtisek 04] Didier Vojtisek & Jean-Marc Jézequél. *MTL and Umlaut NG - Engine and Framework for Model Transformation*. ERCIM News, no. 58, 2004. 31
- [W3C 99] W3C. *XSL Transformations (XSLT)*, 1999. 32
- [Willink 03] Edward D Willink. *UMLX: A graphical transformation language for MDA*. In Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, volume TR-CTIT-03 of *Technical Report*, pages 1–12. CTIT Technology University of Twente, 2003. 31
- [Wimmer 11] Manuel Wimmer, Angelika Kusel, Werner Retschitzegger & Johannes Sch. *Reusing Model Transformations across Heterogeneous Metamodels*. 5th International Workshop on Multi-Paradigm Modeling, vol. 42, 2011. 13, 57, 58, 60, 152
- [Xtend 08] Xtend. *Xtend*, 2008. 33
- [Xtext 08] Xtext. *Xtext*, 2008. 44, 134

- [Xtext 12] Xtext. *Xtext - Documentation*, 2012. 143
- [Zhang 05] Jing Zhang & Yuehua Lin. *Generic and domain-specific model refactoring using a model transformation engine*. Model-driven Software Development, 2005. 23

List of Figures

1	Métamodèles de la transformation CD2RDB.	2
2	Extrait du fichier CD2RDB.mgt.	3
3	Métamodèle de TGMs.	4
4	TGM du nouveau MM CDV2.	5
5	Approche de sous-typage.	6
6	TGM effectif de CDV1.	7
7	TGM effectif <i>cartographié</i>	9
8	Extrait de la représentation textuelle et en mémoire des patterns de graphe (modèle) dans le processus d'adaptation.	11
9	Editeur de MetaModMap.	12
10	Modèles d'entrée et sortie de la transformation CD2RDB.	13
1.1	Objective overview in different scenarios.	3
2.1	Software development methodology roadmap.	8
2.2	An example model of a library.	11
2.3	Different kinds of model roles.	12
2.4	A metamodel to model the Class aspect of the Library system.	13
2.5	Illustration of a reflexive metamodel.	14
2.6	MOF four-layer metamodeling architecture.	15
2.7	Excerpt of the EMOF meta-metamodel.	16
2.8	Transformation of a PIM to PSM (from [MDA 03]).	18
2.9	A grid for categorizing development artifacts (from [Greenfield 03]).	20
2.10	DSML definition approaches proposed by OMG-MDA.	21
2.11	Basic concepts of M2M transformation frameworks.	23

2.12	An excerpt of the feature diagram of model transformation languages (for more details, see [Czarnecki 06]).	27
2.13	Categorization of model transformation approaches.	30
2.14	Class models to relational database models transformation's metamodels.	35
2.15	Sample input/output models.	37
2.16	New source metamodel CDV2 in Ecore-defined representation.	39
2.17	Sample intentional input model designed by a CASE tool.	40
2.18	Sample input model in CDV2-defined representation.	40
2.19	Steps performed in MOMENT2-GT (adapted from [Boronat 07]).	43
2.20	MOMENT2-GT metamodel: transformation and model types.	44
2.21	MOMENT2-GT metamodel: rules and domain patterns.	46
2.22	MOMENT2-GT metamodel: graph patterns.	48
2.23	Similar concepts between metamodels.	52
2.24	Similar type concepts (1) with different reifications in Ecore.	53
2.25	Similar attribute concepts (2) with different reifications in Ecore.	53
2.26	Similar reference concepts (3) with different reifications in Ecore.	54
2.27	Similar concept groups (4) with different reifications in Ecore.	55
3.1	The same graph topology between metamodels.	70
3.2	The graph topology in transformations.	72
3.3	Metamodel of TGMs.	75
3.4	An example of transforming a metamodel to its derived TGM.	77
3.5	TGMs of metamodels CDV1 and CDV2.	82
3.6	Models of different metamodels with the same model type.	83
3.7	Sub-typing approach.	87
3.8	Transformation typing overview.	89
3.9	Inferred effective TGM for CDV1 (at the bottom).	95
3.10	Overview of the non-isomorphic transformation reuse approach.	97
3.11	Abstract syntax of transformation language MOMENT2-GT.	99
3.12	Abstract syntax of correspondence language MetaModMap.	100
3.13	Attribute reification example.	104
3.14	Reference reification example.	105
3.15	From <i>effective</i> TGM to <i>mapped</i> effective TGM.	108

3.16	Overview of transformation adaptation algorithm.	111
3.17	Rebinding example for a type concept mapping.	114
3.18	Rebinding example for a single attribute mapping.	115
3.19	Rebinding example for a single reference mapping.	115
3.20	Rebinding example for a composite attribute mapping.	116
3.21	Rebinding example for a composite reference mapping.	117
3.22	Excerpt of textual and in-memory representation of (model) graph patterns in adaptation process.	119
4.1	MetaModMap architecture at a glance.	125
4.2	Eclipse plug-in based platform architecture.	126
4.3	The simplest project structure of an Eclipse plug-in.	127
4.4	An example extending Eclipse Project by adding a text editor func- tionality.	129
4.5	Syntax coloring for the <i>myDSL</i> editor.	130
4.6	Relationship of default implementation classes.	131
4.7	Overview of XTEXT	134
4.8	Integration of MetaModMap editor into MOMENT2-GT.	140
4.9	Invocation of typing functions when referring to a mapping.	144
4.10	Scoping when defining a mapping between two concepts.	145
4.11	<i>Model type compatibility</i> validation at coarse-grained level.	146
4.12	MetaModMap editor.	149
4.13	A scenario shows the need of a transformation adaptation toolkit.	150
4.14	CD2RDB transformation's in/out models.	151

List of Tables

2.1	Two dimensions of transformations with examples [Mens 06].	26
2.2	Related approaches to transformation reuse.	64
4.1	XTEXT standard fragments and artifacts	135
4.2	Comparison to other transformation reuse approaches	154

List of Listings

1	Une definition en MetaModMap : CDV1 vs. CDV2	8
2.1	MOMENT2-GT XTEXT Grammar: transformation and model types	45
2.2	Transformation and model type declarations	45
2.3	MOMENT2GT XTEXT Grammar: rules and domain patterns	46
2.4	Rule and domain pattern declarations	47
2.5	MOMENT2-GT XTEXT Grammar: graph patterns	48
2.6	Graph patterns declarations	49
3.1	Model typing function: the Trace interface in the Java language	78
3.2	MetaModMap XTEXT Grammar : mappings and mapping rules	101
3.3	MetaModMap XTEXT Grammar : concept and property mappings	102
3.4	MetaModMap definition: CDV1 vs. CDV2	103
3.5	Mapping illustration for attribute reification.	104
3.6	Mapping illustration for reference reification.	105
4.1	Example Plug-in: MANIFEST.MF	128
4.2	Example Plug-in: <i>plugin.xml</i>	128
4.3	myDSL.ui Plug-in: <i>plugin.xml</i>	129
4.4	myDSL.ui Plug-in: myDSLEditor.java	131
4.5	myDSL.ui Plug-in: myDSLConfiguration.java	132
4.6	myDSL.ui Plug-in: myDSLScanner.java	133
4.7	XTEXT: dependency declaration	135
4.8	XTEXT: configuration of components	136
4.9	Wiring up an application in standalone mode	136
4.10	XTEXT-based myDSL.ui Plug-in: <i>plugin.xml</i>	137
4.11	An excerpt of MetaModMap XTEXT Grammar	141
4.12	Configuration for the custom scoping artifact	142

4.13	Configuration for the custom validation artifact	142
4.14	Two signatures of methods for scoping customization	143
4.15	Method for the effective TGM model type inference	143
4.16	Scope computation for the <i>conceptSource</i> reference of <i>ConceptMap</i> . .	145
4.17	<i>Model type compatibility</i> validation check method	146
4.18	metamodemap.dsl.ui Plug-in: <i>plugin.xml</i>	147
4.19	Concrete bindings in Guice modules of <i>MetaModMap</i>	147
4.20	Integration of a HOT interpreter for <i>MetaModMap</i> language	148

Appendices

Appendix A

Transformation Definitions

1.1 Transformation for CDV1 and RDB metamodels

```
1 import "platform:/resource/CD2RDB/metamodels/CDV1.ecore";
2 import "platform:/resource/CD2RDB/metamodels/RDB.ecore";
3
4 transformation CD2RDB (cd : CDV1; rdb : RDB) {
5
6     rl InitAncestor {
7
8         nac cd noAncestor {
9             c1 : Class {
10                 ancestors = c2 : Class {
11                     }
12                 }
13             };
14
15             lhs {
16                 cd {
17                     c1 : Class {
18                         parent = c2 : Class {
19                             }
20                     }
21                 }
22             };
23
24             rhs {
25                 cd {
26                     c1 : Class {
27                         parent = c2 : Class {
28                             },
29                         ancestors = c2 : Class {
30                             }
31                     }
32                 }
33             };
34         }
35
36         rl ComputeTransitiveAncestor {
37
38             nac cd noAncestorRelation {
39                 c1 : Class {
40                     ancestors = c3 : Class {
41                         }
42                     }
43             };

```

```
44
45     lhs {
46         cd {
47             c1 : Class {
48                 ancestors = c2 : Class {
49                     ancestors = c3 : Class {
50                         }
51                     }
52             }
53         }
54     };
55
56     rhs {
57         cd {
58             c1 : Class {
59                 ancestors = c2 : Class {
60                     ancestors = c3 : Class {
61                         }
62                     },
63                 ancestors = c3 : Class {
64                     }
65             }
66         }
67     };
68 }
69
70 rl Package2Schema {
71
72     nac rdb noSchema {
73         s1 : Schema {
74             }
75     };
76
77     lhs {
78         cd {
79             p : Package {
80                 name = pname
81             }
82         }
83         rdb {
84             }
85     };
86
87     rhs {
88         cd {
89             p : Package {
90                 name = pname
91             }
92         }
93         rdb {
94             s : Schema {
95                 name = "_" + pname
96             }
97         }
98     };
99 }
```

```
100
101 rl Class2Table {
102
103     nac rdb noTable {
104         t1 : Table {
105             name = cname
106         }
107     };
108
109     nac cd noParent {
110         c : Class {
111             parent = pClass : Class {
112             }
113         }
114     };
115
116     lhs {
117         cd {
118             p : Package {
119                 ownedTypes = c : Class {
120                     isPersistent = true,
121                     name = cname
122                 }
123             }
124         }
125         rdb {
126             s : Schema {
127             }
128         }
129     };
130
131     rhs {
132         cd {
133             p : Package {
134                 ownedTypes = c : Class {
135                     isPersistent = true,
136                     name = cname
137                 }
138             }
139         }
140         rdb {
141             s : Schema {
142                 ownedTables = t : Table {
143                     name = cname
144                 }
145             }
146         }
147     };
148 }
149
150 rl DataTypeAttributeOfTopClass2Column {
151
152     nac rdb noColumn {
153         t : Table {
154             cols = c10 : Column {
155                 name = aname
156             }
157         }
158     };
```

```

159
160     lhs {
161         cd {
162             c : Class {
163                 name = cname,
164                 ownedFeatures = a : Attribute {
165                     name = aname,
166                     type = p : PrimitiveDataType {
167                         name = pname
168                     }
169                 }
170             }
171         }
172         rdb {
173             t : Table {
174                 name = cname
175             }
176         }
177     };
178
179     rhs {
180         cd {
181             c : Class {
182                 name = cname,
183                 ownedFeatures = a : Attribute {
184                     name = aname,
185                     type = p : PrimitiveDataType {
186                         name = pname
187                     }
188                 }
189             }
190         }
191         rdb {
192             t : Table {
193                 name = cname,
194                 cols = cl : Column {
195                     name = aname,
196                     type = pname
197                 }
198             }
199         }
200     };
201 }
202
203 rl DataTypeAttributeOfChildClass2Column {
204
205     nac rdb noColumn {
206         t : Table {
207             cols = cl0 : Column {
208                 name = aname
209             }
210         }
211     };
212
213     lhs {
214         cd {
215             c : Class {
216                 ancestors = anc : Class {
217                     name = cname
218                 },
219                 ownedFeatures = a : Attribute {
220                     name = aname,
221                     type = p : PrimitiveDataType {
222                         name = pname
223                     }
224                 }
225             }
226         }

```

```

227     rdb {
228         t : Table {
229             name = cname
230         }
231     }
232 };
233
234 rhs {
235     cd {
236         c : Class {
237             ancestors = anc : Class {
238                 name = cname
239             },
240             ownedFeatures = a : Attribute {
241                 name = aname,
242                 type = p : PrimitiveDataType {
243                     name = pname
244                 }
245             }
246         }
247     }
248     rdb {
249         t : Table {
250             name = cname,
251             cols = cl : Column {
252                 name = aname ,
253                 type = pname
254             }
255         }
256     }
257 };
258 }
259
260 rl SetPrimaryKeyOfTopClass {
261
262     nac rdb noPK {
263         t : Table {
264             pkeys = cl : Column {
265             }
266         }
267     };
268
269     lhs {
270         cd {
271             c : Class {
272                 name = cname,
273                 ownedFeatures = a : Attribute {
274                     name = aname,
275                     isPrimary = true
276                 }
277             }
278         }
279         rdb {
280             t : Table {
281                 name = cname,
282                 cols = cl : Column {
283                     name = aname
284                 }
285             }
286         }
287     };

```



```

288
289     rhs {
290         cd {
291             c : Class {
292                 name = cname,
293                 ownedFeatures = a : Attribute {
294                     name = aname,
295                     isPrimary = true
296                 }
297             }
298         }
299         rdb {
300             t : Table {
301                 name = cname,
302                 cols = cl : Column {
303                     name = aname
304                 },
305                 pkeys = cl : Column {
306                 }
307             }
308         }
309     };
310 }
311
312 rl setPrimaryKeyOfChildClass {
313
314     nac rdb noPK {
315         t : Table {
316             pkeys = cl : Column {
317             }
318         }
319     };
320
321     lhs {
322         cd {
323             c : Class {
324                 ancestors = anc : Class {
325                     name = cname
326                 },
327                 ownedFeatures = a : Attribute {
328                     name = aname,
329                     isPrimary = true
330                 }
331             }
332         }
333         rdb {
334             t : Table {
335                 name = cname,
336                 cols = cl : Column {
337                     name = aname
338                 }
339             }
340         }
341     };
342
343     rhs {
344         cd {
345             c : Class {
346                 ancestors = anc : Class {
347                     name = cname
348                 },
349                 ownedFeatures = a : Attribute {
350                     name = aname,
351                     isPrimary = true
352                 }
353             }
354         }

```

```

355         rdb {
356             t : Table {
357                 name = cname,
358                 cols = cl : Column {
359                     name = aname
360                 },
361                 pkeys = cl : Column {
362                 }
363             }
364         };
365     };
366 }
367
368 rl AssociationToPersClass2FKey {
369
370     nac rdb noAssColumn {
371         t1 : Table {
372             cols = cl : Column {
373                 name = assname + "_" + cname,
374                 type = pname
375             }
376         }
377     };
378
379     lhs {
380         cd {
381             ass : Association {
382                 name = assname,
383                 source = c1 : Class {
384                     name = c1name
385                 },
386                 target = c2 : Class {
387                     name = c2name
388                 }
389             }
390         }
391         rdb {
392             t1 : Table {
393                 name = c1name
394             }
395             t2 : Table {
396                 name = c2name,
397                 pkeys = pk : Column {
398                     name = cname,
399                     type = pname
400                 }
401             }
402         }
403     };
404
405     rhs {
406         cd {
407             ass : Association {
408                 name = assname,
409                 source = c1 : Class {
410                     name = c1name
411                 },
412                 target = c2 : Class {
413                     name = c2name
414                 }
415             }
416         }

```

```

417     rdb {
418         t1 : Table {
419             name = c1name,
420             cols = cl : Column {
421                 name = assname + "_" + c1name,
422                 type = pname
423             },
424             fkeys = fk : FKKey {
425                 cols = cl : Column {
426                     },
427                 reference = t2 : Table {
428                     }
429             }
430         }
431         t2 : Table {
432             name = c2name,
433             pkeys = pk : Column {
434                 name = c1name,
435                 type = pname
436             }
437         }
438     };
439 };
440 }
441
442 rl AssociationToNonPersClass2FKKey {
443
444     nac rdb noAssColumn {
445         cl : Column {
446             name = assname + "_" + aname,
447             type = pname
448         }
449     };
450
451     lhs {
452         cd {
453             ass : Association {
454                 name = assname,
455                 source = c1 : Class {
456                     name = c1name
457                 },
458                 target = c2 : Class {
459                     isPersistent = false,
460                     ownedFeatures = a : Attribute {
461                         name = aname,
462                         type = p : PrimitiveDataType {
463                             name = pname
464                         }
465                     }
466                 }
467             }
468         }
469         rdb {
470             t1 : Table {
471                 name = c1name
472             }
473         }
474     };

```

```
475
476   rhs {
477     cd {
478       ass : Association {
479         name = assname,
480         source = c1 : Class {
481           name = c1name
482         },
483         target = c2 : Class {
484           isPersistent = false,
485           ownedFeatures = a : Attribute {
486             name = aname,
487             type = p : PrimitiveDataType {
488               name = pname
489             }
490           }
491         }
492       }
493     }
494     rdb {
495       t1 : Table {
496         name = c1name,
497         cols = cl : Column {
498           name = assname + "_" + aname,
499           type = pname
500         },
501         pkeys = cl : Column {
502           }
503       }
504     }
505   };
506 }
507 }
```



```

48
49     lhs {
50         cd {
51             c1 : Class {
52                 ancestors = c2 : Class {
53                     ancestors = c3 : Class {
54                         }
55                     }
56             }
57         }
58     };
59
60     rhs {
61         cd {
62             c1 : Class {
63                 ancestors = c2 : Class {
64                     ancestors = c3 : Class {
65                         }
66                     },
67                 ancestors = c3 : Class {
68                     }
69             }
70         }
71     };
72 }
73
74 rl Package2Schema {
75
76     nac rdb noSchema {
77         s1 : Schema {
78             }
79     };
80
81     lhs {
82         cd {
83             p : Package {
84                 name = pname
85             }
86         }
87         rdb {
88             }
89     };
90
91     rhs {
92         cd {
93             p : Package {
94                 name = pname
95             }
96         }
97         rdb {
98             s : Schema {
99                 name = pname
100             }
101         }
102     };
103 }
104
105 rl Class2Table {
106
107     nac rdb noTable {
108         t1 : Table {
109             name = cname
110         }
111     };

```

```

112
113     nac cd noParent {
114         c : Class {
115             generalization = _newpClass : Generalization {
116                 general = pClass : Class {
117                     }
118                 }
119             }
120         };
121
122     lhs {
123         cd {
124             p : Package {
125                 ownedTypes = c : Class {
126                     persistent = true,
127                     name = cname
128                 }
129             }
130         }
131         rdb {
132             s : Schema {
133             }
134         }
135     };
136
137     rhs {
138         cd {
139             p : Package {
140                 ownedTypes = c : Class {
141                     persistent = true,
142                     name = cname
143                 }
144             }
145         }
146         rdb {
147             s : Schema {
148                 ownedTables = t : Table {
149                     name = cname
150                 }
151             }
152         }
153     };
154 }
155
156 rl DataTypeAttributeOfTopClass2Column {
157
158     nac rdb noColumn {
159         t : Table {
160             cols = cl0 : Column {
161                 name = aname
162             }
163         }
164     };
165
166     lhs {
167         cd {
168             c : Class {
169                 name = cname,
170                 ownedProperties = a : Property {
171                     name = aname,
172                     type = p : DataType {
173                         name = pname
174                     }
175                 }
176             }
177         }

```

```

178         rdb {
179             t : Table {
180                 name = cname
181             }
182         }
183     };
184
185     rhs {
186         cd {
187             c : Class {
188                 name = cname,
189                 ownedProperties = a : Property {
190                     name = aname,
191                     type = p : DataType {
192                         name = pname
193                     }
194                 }
195             }
196         }
197         rdb {
198             t : Table {
199                 name = cname,
200                 cols = cl : Column {
201                     name = aname,
202                     type = pname
203                 }
204             }
205         }
206     };
207 }
208
209 rl DataTypeAttributeOfChildClass2Column {
210
211     nac rdb noColumn {
212         t : Table {
213             cols = cl0 : Column {
214                 name = aname
215             }
216         }
217     };
218
219     lhs {
220         cd {
221             c : Class {
222                 ancestors = anc : Class {
223                     name = cname
224                 },
225                 ownedProperties = a : Property {
226                     name = aname,
227                     type = p : DataType {
228                         name = pname
229                     }
230                 }
231             }
232         }
233         rdb {
234             t : Table {
235                 name = cname
236             }
237         }
238     };

```



```

239
240     rhs {
241         cd {
242             c : Class {
243                 ancestors = anc : Class {
244                     name = cname
245                 },
246                 ownedProperties = a : Property {
247                     name = aname,
248                     type = p : DataType {
249                         name = pname
250                     }
251                 }
252             }
253         }
254         rdb {
255             t : Table {
256                 name = cname,
257                 cols = cl : Column {
258                     name = aname,
259                     type = pname
260                 }
261             }
262         }
263     };
264 }
265
266 rl SetPrimaryKeyOfTopClass {
267
268     nac rdb noPK {
269         t : Table {
270             pkeys = cl : Column {
271             }
272         }
273     };
274
275     lhs {
276         cd {
277             c : Class {
278                 name = cname,
279                 ownedProperties = a : Property {
280                     name = aname,
281                     primary = true
282                 }
283             }
284         }
285         rdb {
286             t : Table {
287                 name = cname,
288                 cols = cl : Column {
289                     name = aname
290                 }
291             }
292         }
293     };
294
295     rhs {
296         cd {
297             c : Class {
298                 name = cname,
299                 ownedProperties = a : Property {
300                     name = aname,
301                     primary = true
302                 }
303             }
304         }

```

```

305         rdb {
306             t : Table {
307                 name = cname,
308                 cols = cl : Column {
309                     name = aname
310                 },
311                 pkeys = cl : Column {
312                 }
313             }
314         }
315     };
316 }
317
318 rl setPrimaryKeyOfChildClass {
319
320     nac rdb noPK {
321         t : Table {
322             pkeys = cl : Column {
323             }
324         }
325     };
326
327     lhs {
328         cd {
329             c : Class {
330                 ancestors = anc : Class {
331                     name = cname
332                 },
333                 ownedProperties = a : Property {
334                     name = aname,
335                     primary = true
336                 }
337             }
338         }
339         rdb {
340             t : Table {
341                 name = cname,
342                 cols = cl : Column {
343                     name = aname
344                 }
345             }
346         }
347     };
348
349     rhs {
350         cd {
351             c : Class {
352                 ancestors = anc : Class {
353                     name = cname
354                 },
355                 ownedProperties = a : Property {
356                     name = aname,
357                     primary = true
358                 }
359             }
360         }
361         rdb {
362             t : Table {
363                 name = cname,
364                 cols = cl : Column {
365                     name = aname
366                 },
367                 pkeys = cl : Column {
368                 }
369             }
370         }
371     };
372 }

```

```

373
374 rl AssociationToPersClass2FKKey {
375
376   nac rdb noAssColumn {
377     t1 : Table {
378       cols = cl : Column {
379         name = assname + "_" + cname,
380         type = pname
381       }
382     }
383   };
384
385   lhs {
386     cd {
387       ass : Association {
388         name = assname,
389         from = c1 : Class {
390           name = c1name
391         },
392         to = c2 : Class {
393           name = c2name
394         }
395       }
396     }
397     rdb {
398       t1 : Table {
399         name = c1name
400       }
401       t2 : Table {
402         name = c2name,
403         pkeys = pk : Column {
404           name = cname,
405           type = pname
406         }
407       }
408     }
409   };
410
411   rhs {
412     cd {
413       ass : Association {
414         name = assname,
415         from = c1 : Class {
416           name = c1name
417         },
418         to = c2 : Class {
419           name = c2name
420         }
421       }
422     }
423     rdb {
424       t1 : Table {
425         name = c1name,
426         cols = cl : Column {
427           name = assname + "_" + cname,
428           type = pname
429         },
430         fkeys = fk : FKKey {
431           cols = cl : Column {
432             },
433           reference = t2 : Table {
434             }
435         }
436     }

```

```

437         t2 : Table {
438             name = c2name,
439             pkeys = pk : Column {
440                 name = cname,
441                 type = pname
442             }
443         }
444     };
445 };
446 }
447
448 rl AssociationToNonPersClass2FKKey {
449
450     nac rdb noAssColumn {
451         cl : Column {
452             name = asname + "_" + aname,
453             type = pname
454         }
455     };
456
457     lhs {
458         cd {
459             ass : Association {
460                 name = asname,
461                 from = c1 : Class {
462                     name = cname
463                 },
464                 to = c2 : Class {
465                     persistent = false,
466                     ownedProperties = a : Property {
467                         name = aname,
468                         type = p : DataType {
469                             name = pname
470                         }
471                     }
472                 }
473             }
474         }
475         rdb {
476             t1 : Table {
477                 name = cname
478             }
479         }
480     };
481
482     rhs {
483         cd {
484             ass : Association {
485                 name = asname,
486                 from = c1 : Class {
487                     name = cname
488                 },
489                 to = c2 : Class {
490                     persistent = false,
491                     ownedProperties = a : Property {
492                         name = aname,
493                         type = p : DataType {
494                             name = pname
495                         }
496                     }
497                 }
498             }
499         }

```

```
537         rdb {
538             t1 : Table {
539                 name = c1name,
540                 cols = cl : Column {
541                     name = assname + "_" + aname,
542                     type = pname
543                 },
544                 pkeys = cl : Column {
545                     }
546             }
547         };
548     };
549 }
550 }
```

Appendix B

MetaModMap Implementation Detail

This appendix includes some implementation details.

2.1 A Java implementation for the *Trace* interface

```
1 package typesystem.trace;
2
3 import java.util.HashMap;
4 import java.util.List;
5
6 import org.eclipse.emf.common.util.BasicEList;
7 import org.eclipse.emf.ecore.EObject;
8
9 public class TraceImpl<SRC extends EObject, TGT extends EObject > implements
    Trace <SRC,TGT> {
10
11     private HashMap<SRC, List <TGT> > src2tgt;
12     private HashMap<TGT,SRC> tgt2src;
13
14     /* The shared global trace */
15     private static TraceImpl<EObject, EObject> sharedTrace;
16
17     /* get the global trace */
18     public static TraceImpl<EObject, EObject> getDefault()
19     {
20         if (sharedTrace == null)
21         {
22             sharedTrace = new TraceImpl<EObject, EObject>();
23         }
24         return sharedTrace;
25     }
```

```
26
27  /* constructor to init maps */
28  public TraceImpl(){
29      src2tgt = new HashMap<SRC, List <TGT> >();
30      tgt2src = new HashMap<TGT,SRC>();
31  }
32
33  /* implementation to get a source element */
34  public SRC getSourceElem(TGT tgt) {
35      return tgt2src.get(tgt);
36  }
37
38  /* implementation to get a list of target element */
39  public List<TGT> getTargetElems(SRC src) {
40      return src2tgt.get(src);
41  }
42
43  /* implementation to store a trace */
44  public void storeTrace(SRC src, TGT tgt)
45  {
46      List < TGT> listTGT = src2tgt.get(src);
47
48      if (listTGT !=null ) {
49          listTGT.add(tgt);
50      }
51      else {
52          listTGT = new BasicEList<TGT>();
53          listTGT.add(tgt);
54          src2tgt.put(src,listTGT);
55      }
56
57      tgt2src.put(tgt, src);
58  }
59
60  /* implementation to remove a trace */
61  public void removeTrace(SRC src, TGT tgt)
62  {
63      List < TGT> listTGT = src2tgt.get(src);
64
65      if (listTGT != null)
66      {
67          listTGT.remove(tgt);
68          if (listTGT.size() == 0)
69          {
70              src2tgt.remove(src);
71          }
72      }
73
74      tgt2src.remove(tgt);
75  }
76 }
```

