Table des matières

Liste des figures **IX**

Liste des tableaux **XI**

Acronymes **1**

Introduction Générale **3**

Chapitre 1 Ingénierie Dirigée par les modèles **6**

1.1 Introduction 6

1.2 Notions générales de L'IDM 7

1.3 Métamodélisation 8

1.3.1 IDM et Langage 8

1.3.1.1 Aspect syntaxique 9

1.3.1.2 Modèle, métamodèle et langage 10

1.3.1.3 Sémantique 10

1.3.2 Architecture de métamodélisation 11

1.3.3 Formalismes de métamodélisation 12

1.3.3.1 Les langages de métamodélisation 13

1.3.3.2 Les Profils UML 14

1.4 Transformation de modèles 14

1.4.1 Typologie de transformation 15

1.4.2 Pourquoi la transformation de modèles 16

1.4.3 Propriétés de transformation de modèles 18

1.4.4 Approches de transformation de modèles 19

1.4.4.1 Model to Text 20
1.4.4.2 Model to Model 20
1.4.5 Les outils de transformation 22
1.4.5.1 Outils génériques 22
1.4.5.2 Outils intégrés aux AGLs 23
1.4.5.3 Langages spécifiques 23
1.4.5.4 Outils de métamodélisation 23
1.5 Conclusion 24

Chapitre 2 Grammaires de graphes 25
2.1 Introduction 25
2.2 Notions sur les graphes 26
2.2.1 Définition d'un graphe 26
2.2.2 propriété d'un graphe 26
2.3 Grammaire de graphes 29
2.3.1 Définition formelle 29
2.3.2 Grammaire de graphe et langage 30
2.3.3 Dérivation 30
2.3.4 Transformation de grammaire 33
2.4 Principe de transformation de graphes 34
2.5 Approches de transformation de graphes 36
2.5.1 Approche algèbrique 36
2.5.2 TGG 37
2.6 Conclusion 38

Chapitre 3 Outils de transformation de graphes 39
3.1 Introduction 39
3.2 Outils de transformation 39
3.2.1 AGG 40
3.2.2 ATOM3 40
3.2.3 GROOVE 40
3.2.4 GrGen 41

3.2.5 Fujaba 41
3.2.6 EMF Tiger 41
3.2.7 Viatra 42
3.3 Etude comparative sur les outils de transformation de graphes . . . 42
3.3.1 Critères de comparaison 42
3.3.1.1 Nombre de source et cible 43
3.3.1.2 Type de transformation 43
3.3.1.3 Approche utilisée 43
3.3.1.4 L'intégrité de contrainte 44
3.3.1.5 Editeur 44
3.3.1.6 Degré d'automatisation 44
3.3.1.7 Processus de transformation 45
3.3.1.8 Réutilisabilité 45
3.3.1.9 Vérification 45
3.3.1.10 Validation 46
3.3.1.11 Composition de transformation 46
3.3.1.12 Complexité de transformation 47
3.3.1.13 Standardisation 47
3.3.1.14 Type d'utilisation des outils 47
3.3.1.15 Typage 48
3.3.1.16 Le contrôle de fonctionnalité des outils 48
3.3.1.17 Exploration 48
3.3.2 Tableaux de comparaison 48
3.4 Conclusion 50

Chapitre 4 Spécification des transformations de modèles avec le formalisme TGG 51
4.1 Introduction 51
4.2 Fondements théoriques de TGG 52
4.3 Processus de transformation de modèles avec TGG 54
4.3.1 Exemple de Session 54
4.3.2 Spécification des transformations 55

4.3.2.1 Spécification des Métamodèles 57
4.3.2.2 Spécification de règles de transformation 61
4.3.2.3 Transformation du modèle 68
4.4 Conclusion 71

Chapitre 5 Mise en œuvre de la transformation de modèles 72
5.1 Introduction 72
5.2 Environnement d'implémentation 72
5.2.1 Eclipse 73
5.2.1.1 Eclipse Modeling Framework 73
5.2.1.2 Ecore 74
5.2.1.3 TGG interpreter 74
5.2.2 Architecture d'implémentation de transformation de modèles 76
5.3 Études de cas 77
5.3.1 Transformation d'un diagramme de classes en base de données 77
5.3.1.1 Les métamodèles 78
5.3.1.2 Définition de règles de TGG" 80
5.3.1.3 L'exécution 84
5.3.2 Transformation d'un diagramme d'activité UML en réseau de Pétri 86
5.3.2.1 Les métamodèles 86
5.3.2.2 Définir la relation "les règles TGG" 87
5.3.2.3 L'exécution 94
5.4 Conclusion 95

Conclusion Générale 96

Bibliographie 98

Appendix A Annexe 103
A.1 Les règles de transformation 103
A.1.1 Transformation de "MergeNode" en "place" 103
A.1.2 Transformation d'un "DecisionNode" en "transition" 105

A.1.3 Transformation de "ForkNode" en transition 107
A.1.4 Transformation de "JoinNode" en "transition" 108

Liste des figures

1.1 L'architecture de métamodélisation 12
1.2 Concepts de base de transformation de modèles 15

2.1 Graphe non orienté 26
2.2 (a) Graphe G (b) Sous graphe de G 27
2.3 Graphe orienté 27
2.4 Graphe étiqueté 28
2.5 Une grammaire déterministe de graphes 31
2.6 Réécriture pour la grammaire de la figure 2.5 32
2.7 Réécritures parallèles pour la grammaire de la figure 2.5 32
2.8 Graphe engendré par la grammaire dans la figure 2.5 33
2.9 Une grammaire de graphe 34
2.10 Principe de mise en œuvre de transformation de modèles 35
2.11 Exemple d'application d'une règle selon DPO 36
2.12 Exemple d'application d'une règle selon SPO 37

4.1 Plan de construction d'un train jouet 55
4.2 Transformation dans les espaces techniques IDM et TGG 56
4.3 Code source d'une règle de transformation en TGG 57
4.4 Réseau de Pétri 58
4.5 Métamodèle du réseau de Pétri 59
4.6 Métamodèle d'un projet jouet 59
4.7 Diagramme d'objets: syntaxe abstraite d'un réseau de Pétri 60
4.8 Différentes nœuds dans une règle de transformation TGG 62
4.9 Plan de construction d'un jouet train 65
4.10 Diagramme d'objet d'un Track et son réseau de Pétri 65
4.11 Règle TGG pour un Track 66
4.12 Règle TGG pour une connexion 67
4.13 L'axiome 67
4.14 Un modèle source et son modèle cible "syntaxe concrète" 68
4.15 Diagramme d'objets d'un projet 69

4.16 Transformation à l'avant après deux applications de la règle Track . 70
4.17 Transformation à l'avant après l'application de la règle de connexion 70

5.1 Principe de mise en œuvre de transformation de modèles 73
5.2 Éditeur de Règle TGG de TGG Interpreter 75
5.3 L'architecture de TGG Interpreter 77
5.4 Métamodèle diagramme de classes 78
5.5 Métamodèle de base de données 79
5.6 Métamodèle de correspondances 79
5.7 Présentation graphique de l'axiome 80
5.8 Présentation graphique de la règle ClassetoTable 81
5.9 Présentation graphique de la règle SubClassetoTable 82
5.10 Présentation graphique de la règle AttributeToColumn 83
5.11 Présentation graphique de la règle PrimaryAttributetoPkey 83
5.12 Présentation graphique de la règle AssociationToFkey 84
5.13 Exemple de modèle de diagramme de classes 85
5.14 Exemple de modèle de base de données 85
5.15 Exemple de modèle de diagramme de classes 87
5.16 L'axiome 88
5.17 Transformation d'un ActivityEdge 88
5.18 Règle TGG pour un activityEdge 89
5.19 Transformation d'un nœud initial 90
5.20 Règle TGG pour un InitialNode-a 90
5.21 Règle TGG pour un InitialNode-b 91
5.22 Transformation d'une Action 91
5.23 Transformation d'une Action 92
5.24 Transformation d'un FinalNode 92
5.25 Transformation d'un FinalNode 93
5.26 Transformation d'un FinalNode 94
5.27 Exemple de modèle de diagramme d'activités 94
5.28 Exemple de modèle de réseau de Pétri 95

A.1 Transformation du nœud merge 103
A.2 La règle de transformation d'un nœud merge 104
A.3 la règle de transformation 105
A.4 présentation graphique de la règle AssociationToFkey 106
A.5 présentation graphique de la règle AssociationToFkey 106
A.6 présentation graphique de la règle AttributeToColumn 107
A.7 présentation graphique de la règle PrimaryAttributetoPkey 108
A.8 présentation graphique de la règle PrimaryAttributetoPkey 109
A.9 présentation graphique de la règle PrimaryAttributetoPkey 110

Liste des tableaux

1.1 Synthèse des transformations de modèles 16
1.2 Les approches de transformation de modèles 19

3.1 Tableau de comparaison 49

Acronymes

[AGG] **A**ttributed **G**raph **G**rammar

[API] **A**pplication **P**rogramming **I**nterface

[$AToM^3$] **A** **To**ol for **M**ulti-formalism and Meta-Modeling

[DPO] **D**ouble **P**ush**O**ut

[EMF] **E**clipse **M**odeling **F**ramework

[EMF Tiger] **T**ransformat**i**on **ge**ne**r**ation

[FUJABA] **F**rom **U**ML to **J**ava **A**nd **B**ack

[GrGen] **Gr**aph Rewrite **Gen**erator

[GROOVE] **GR**aphs for **O**bject-**O**riented **VE**rification

[GXL] **G**raph e**X**change **L**anguage

[IDM] **I**ngénierie **D**irigée par les **M**odèles

[MDA] **M**odel **D**riven **A**rchitecture

[MOF] **M**eta **O**bject **F**acility

[OMG] **O**bject **M**anagement **G**roup

[NAC] **N**egative **A**pplication **C**ondition

[SPO] **S**ingle Push**O**ut

[TG] **T**riple **G**raph.

[TGG] **T**riple **G**raph **G**rammar.

[UML] **U**nified **M**odeling **L**anguage

[VIATRA] **VI**sual **A**utomated model **TRA**nsformations

[XMI] **X**ML **M**etadata **I**nterchange

[XML] e**X**tended **M**arkup **L**anguage

Clicours.COM

Introduction Générale

Depuis une douzaine d'années, en termes d'interopérabilité et d'évolutivité et de réutilisabilité, et grâce à l'Ingénierie Dirigée par les Modèles (IDM) où la philosophie IDM ne ressemble pas à celle des pratiques classiques de développement, le code n'est vu pas comme l'aboutissement final d'un projet mais comme un simple aspect de celui-ci. Les projets sont désormais centrés sur un élément fondamental : "le modèle". Celui-ci est spécifié au moyen d'un formalisme de modélisation, mais de manière indépendante de toute plateforme et à haut niveau d'abstraction où on se concentre sur les concepts plutôt que sur l'implémentation. Ceci nous permet, de conserver le modèle, de le transporter, puis de le transformer afin d'obtenir du code. Lors du changement de plateforme par exemple, on cherche à générer ce code par raffinements successifs des modèles, de la manière la plus automatisée qui soit. On parle de passage du stade «contemplatif» au stade «productif», stade auquel le modèle est, très souvent disponible sous forme de code.

Il existe différentes approches de transformation de modèles qui sont proposées dans la littérature en se basant essentiellement sur la structure du modèle à transformer : arbre ou graphe. A ce niveau, une transformation d'un modèle utilise la structure du modèle pour définir le processus opérationnel de la transformation. Donc, deux types de transformations de modèles émergent :

- Transformation d'arbres : ces méthodes permettent le parcours d'un arbre d'entrée au cours duquel sont générés les fragments de l'arbre de sortie. Ces méthodes se basent généralement sur des documents au format XML.
- Les approches basées sur la transformation de graphes : les modèles possè-

dent souvent une représentation graphique apparentée à un graphe. Dans ce cas, Les techniques de grammaire de graphes et de transformation de graphes peuvent être appliquées pour des transformations de modèles. Cette catégorie d'approches est mise en œuvre dans différents outils tels que VIATRA, ATOM3, FUJABA et AGG.

La majorité des travaux de la communauté d'IDM se concentre sur la mise en œuvre d'un moteur de transformation de modèles utilisant la grammaire de graphes pour les raisons suivantes : Généralement les modèles proposent une variété de notations visuelles y compris les diagrammes d'états, diagrammes d'activités, diagrammes de collaboration, les diagrammes de classes et les réseaux de . En conséquence les graphes semblent être la meilleure présentation de modèles. Le deuxième avantage de l'approche de transformation de graphes réside dans le fait qu'elle est une approche déclarative et très puissante. Elle prouve l'exactitude et les propriétés de convergence et elle permet d'exprimer les modèles à un très haut niveau d'abstraction.

Dans le cadre de nos recherches, notre choix s'est porté naturellement sur la grammaire de graphes comme formalisme sous-jacent pour exprimer la transformation de modèles. Un graphe dans l'approche des grammaires de graphes est dérivé d'un autre graphe en appliquant une règle après l'autre. L'application d'une règle à un graphe définit une relation binaire qui peut être itérée arbitrairement, ce qui donne un processus de dérivation. Par conséquent, l'idée principale de grammaire de graphes est de spécifier comment les éléments du graphe source sont transformés en éléments de graphe cible en utilisant un ensemble de règles de transformation.

La transformation de modèles est un concept clé pour l'ingénierie dirigée par les modèles. Dans ce contexte, les grammaires TGG (Triple Graph Grammar) ont été étudiées et appliquées à plusieurs études de cas et elles montrent une combinaison commode de spécification formelle et de capacités intuitives. Surtout la dérivation automatique des transformations avant et arrière sur un seul ensemble de règles spécifiques pour le modèle intégré simplifie la spécification et améliore la facilité d'utilisation ainsi que la cohérence.

Dans le reste, nous détaillons ces idées à travers le plan du présent manuscrit :

- Le **premier chapitre** présente les notions de base de l'ingénierie dirigée par les modèles en présentant les deux axes principaux de l'IDM qui sont la métamodélisation et la transformation de modèles.
- Le **deuxième chapitre** décrit particulièrement les grammaires de graphes où nous présentons les définitions de base, les différentes approches basées sur l'approche algébrique et la transformation de graphes.
- Le **troisième chapitre** présente plusieurs outils de transformation de graphes en mettant l'accent sur une étude comparative de ces outils selon différents critères.
- Le **quatrième chapitre** présente la conception et formalisme de l'approche des grammaires de graphes triples (TGG).
- Le **cinquième chapitre** explicite l'environnement TGG et l'implémentation réalisée dans le cadre de ce projet. .

Et nous concluons ce mémoire par des perspectives qui permettront de donner une suite à cette recherche.

Chapitre 1

Ingénierie Dirigée par les modèles

1.1 Introduction

Ces dernières années ont été témoins d'une complexité croissante des systèmes et des technologies logicielles. Un certain nombre de plates-formes (par exemple CORBA, J2EE, .NET) ont été introduites, qui venait souvent avec leur propre langage de programmation (par exemple C ++, Java). Afin de faire face à ces problèmes, l'Ingénierie Dirigée par les Modèles "IDM" a été proposée visant à préserver les investissements dans la construction de systèmes logiciels complexes contre les solutions technologiques en évolution rapide. Les principales difficultés avec les langages de modélisation actuels, y compris UML, est qu'ils ne sont généralement pas utilisés pour fournir de manière intégrée les spécifications pour un programme dans une collection mathématiquement correctes des documents. À cet égard, IDM propose d'étendre l'utilisation officielle des langages de modélisation de plusieurs façons intéressantes en exploitant le principe de "tout est un modèle". En particulier, il prescrit la manière dont la conception doit être mis en œuvre par le découplage des fonctionnalités du système à partir des décisions spécifiques de la plate-forme sur laquelle la mise en œuvre est développé. Au-delà de l'aide de cette information pour la génération de code, les sites peuvent l'employer pour l'entretien, ainsi que pour des considérations évolutives telles que le portage vers de nouvelles plates-formes. En résumé, IDM couvre le cycle de vie complet de l'application.

Dans ce chapitre, nous présentons les fondements de base de l'IDM ainsi que ses

deux principaux concepts : la métamodélisation et la transformation de modèles qui sont toujours des domaines de recherche.

1.2 Notions générales de L'IDM

Système : Un système est un ensemble d'éléments liés entre eux par des relations dont si un élément soit modifié tout l'ensemble d'éléments sera modifié. Pour mieux connaître et étudier un système, on a besoin de le simplifier et le modéliser et avoir une abstraction du système qui est un modèle[14].

Modèle : Un «modèle» est une représentation simplifiée d'un système où il est construit avec l'intention de décrire le système et répondre aux questions que l'on se pose sur lui. Donc un système est modélisé avec un ensemble de modèles où chacun capture un aspect particulier. Par analogie avec les langages de programmation, un programme exécutable représente le système alors que le code source de ce programme représente le modèle[41].

Chaque modèle est exprimé à partir d'un langage de modélisation qui doit être clairement défini, et comme l'IDM prend la définition de tout est modèle donc le langage de modélisation prend la forme d'un modèle, appelé métamodèle, [54]

Métamodèle : Un métamodèle est un langage d'expression (langage de modélisation) d'un modèle, ou un métamodèle représente une spécification formelle d'une abstraction. Le concept de métamodèle permet également d'aider, pour un système donné, à la construction d'un modèle. Par conséquent, un modèle est une instance d'un métamodèle et un modèle est lié à son métamodèle par une relation de conformité.

Un métamodèle est exprimé à son tour avec un langage de métamodélisation spécifié par un méta-métamodèle.

Méta-métamodèle : Comme l'IDM prend la définition de "tout est modèle", un méta-métamodèle permet de décrire un modèle de métamodèles. Il est un langage de description de métamodèles. Il permet d'exprimer les règles de conformité qui lient les entités du niveau modèle à celles du niveau métamodèle.

Un méta-métamodèle est conçu avec la propriété de méta-circularité, c'est à dire la capacité de se décrire lui-même. ça veut dire que le langage utilisé au niveau du méta-méta-modèle doit être suffisamment puissant pour spécifier sa propre syntaxe abstraite et ce niveau d'abstraction demeure largement suffisant. Le méta-métamodèle MOF est un exemple proposé par l'OMG [51].

1.3 Métamodélisation

La métamodélisation est une activité qui consiste à définir le métamodèle d'un langage de modélisation. Il s'agit non seulement de produire des métamodèles mais aussi de définir la sémantique du langage, de mettre en œuvre des analyseurs, des compilateurs, des générateurs de code et plus généralement, à construire un ensemble d'outils exploitant les métamodèles.

1.3.1 IDM et Langage

Dans le contexte de l'IDM, les modèles sont traités via l'utilisation de langages. Un langage est présenté soit graphiquement, soit textuellement, soit par les deux (composé de texte et de symboles graphiques). Cependant les symboles graphiques ou les mots d'un langage textuel ne peuvent pas être exploités sauf si on leur donne une interprétation par rapport aux éléments du système modélisé et pour comprendre et manipuler un élément de modèle, il faut qu'il puisse être interprété. Un système peut être décrit par un modèle où les deux peuvent être considérés comme deux ensembles associés par une relation d'interprétation.
L'interprétation établie un lien entre un ou plusieurs éléments du modèle et un ou plusieurs éléments du système modélisé. Pour attribuer un sens aux éléments d'un modèle, la définition explicite d'une sémantique, qu'elle soit formelle ou non, est indispensable pour relier les éléments de modèles aux informations. L'ensemble des informations décrites constitue le domaine sémantique. La ou les données d'un

modèle qui décrivent une information du domaine sémantique sont appelées des méta-informations. [54]

1.3.1.1 Aspect syntaxique

La syntaxe d'un langage est définie par un ensemble d'expressions (des mots, des phrases, des instructions, ou des diagrammes). Ces expressions peuvent être de deux types : les expressions simples et les expressions construites à partir d'autres expressions. Ainsi, pour exprimer une syntaxe d'un langage, on trouve plusieurs approches où les deux les plus utilisées sont l'approche par métamodélisation et l'approche grammaticale. La syntaxe d'un langage peut être l'un des deux types suivants :

- *Syntaxe concrète* ou syntaxe statique est destinée à être manipulée par l'utilisateur. Elle est spécifiée par des règles formelles qui permettent la construction des analyseurs. Les syntaxes concrètes (CS) d'un langage fournissent à l'utilisateur un ou plusieurs formalismes, graphiques et/ou textuels, pour manipuler les concepts de la syntaxe abstraite et ainsi en créer des «instances». Le modèle ainsi obtenu sera conforme à la structure définie par la syntaxe abstraite. La définition d'une syntaxe concrète consiste à définir des mappings et permet aussi «d'annoter» chaque construction du langage de modélisation définie dans la syntaxe abstraite.[14]

- *Syntaxe abstraite* est destinée à être manipulée par l'ordinateur. C'est une dérivée de la syntaxe concrète où elle exprime, de manière structurelle, l'ensemble des concepts manipulés au travers du langage. La syntaxe abstraite (AS) d'un langage de modélisation exprime l'ensemble de ses concepts et leurs relations. Les langages de métamodélisation tels que le standard MOF, offrent les concepts et les relations élémentaires qui permettent de décrire un métamodèle représentant la syntaxe abstraite d'un langage de modélisation.[14]

D'une autre façon, un modèle est une représentation abstraite d'un système, défini dans un langage de modélisation. Ce dernier se compose d'une syntaxe abstraite, d'une syntaxe concrète et d'une sémantique. La syntaxe abstraite décrit le vocabulaire des concepts fournis par le langage de modélisation et comment

ils peuvent être reliés pour créer des modèles. La syntaxe concrète présente un modèle soit sous une forme de schéma (syntaxe visuelle) ou sous une forme textuelle structurée (syntaxe textuelle). La sémantique d'un langage de modélisation est l'information supplémentaire pour expliquer ce que la syntaxe abstraite signifie réellement [10].

1.3.1.2 Modèle, métamodèle et langage

Dans le contexte de l'IDM, un métamodèle est la syntaxe abstraite du langage qui sert à exprimer les modèles. Donc, la syntaxe abstraite décrit comment un modèle doit être constitué et qu'elles sont les hypothèses de validité des constituants du modèle. Il s'agit tout d'abord d'une spécification purement structurelle le contenant qui indique comment les éléments de modèles sont organisés et quelle sont les informations contenues. Cette spécification structurelle peut être enrichie de contraintes. Ainsi, un modèle est alors considéré comme bien formé si les règles structurelles sont respectées et si l'évaluation de toutes les contraintes renvoie vrais.
Typiquement, l'analyse syntaxique vérifie si des modèles sont structurellement bien formés par construction. Après cette étape, l'évaluation des contraintes permet de garantir qu'un modèle est valide par rapport aux règles exprimées dans la syntaxe abstraite.

1.3.1.3 Sémantique

Alain Plantec [54] introduit la notion de la sémantique dans le contexte de l'IDM et la relie au raisonnement et la prise de décision. Il argumente ses propos dans ce qui suit : *Le raisonnement et la prise de décision implique l'existence d'une sémantique quelle soit formelle ou non. La sémantique d'un langage permet d'associer un sens aux éléments du langage utilisé pour le raisonnement. Le sens donné à un élément de modèle comporte toujours une part sociologique dépendante de la communauté qui manipule le modèle.*
Les modèles conceptuels et les métamodèles s'appuient sur une formalisation, la nature des informations représentées est prescrite et imposée par le modèle pour une application particulière. La sémantique peut-être exprimée suivant différentes approches, informelle, en langage courant ou formellement par une sémantique

dénotationnelle, opérationnelle, axiomatique ou algébrique. Si le modèle est un automate ou un programme, la sémantique comprend son activité ou son comportement.
Pour un modèle comprenant des instructions logiques, la sémantique inclut l'évaluation des instructions logiques (par interprétation de fonctions ou des symboles des prédicats). Mais le terme "sens" inclut aussi une description informelle de comment les éléments du modèles "fonctionnent". L'objectif est d'exprimer le lien entre les éléments de modèles construits et un domaine sémantique.

1.3.2 Architecture de métamodélisation

L'OMG a défini une architecture de métamodélisation à quatre niveaux, connue sous l'appellation de l'architecture dirigée par les modèles (Model-Driven Architecture : MDA) :

1. *Niveau M0* : c'est la couche la plus basse qui présente les données du monde réel (concrète). Les éléments de cette couche sont une instance de la couche supérieure qui est M1.

2. *Niveau M1* : c'est le niveau modèle où chaque élément du monde réel est représenté par un modèle. Les modèles de cette couche sont décrits par un métamodèle de la couche M2.

3. *Niveau M2* : dans ce niveau sont représentés les métamodèles pour décrire les langages de modélisation.

4. *Niveau M3* : c'est le plus haut niveau, c'est le niveau de méta-métamodèle qui permet de définir la structure des métamodèles du niveau M2, et comme le méta-métamodèle est auto-descriptif, il permet de décrire sa propre sémantique.

La figure 1.1 illustre les quatre niveaux de l'architecture de métamodélisation.

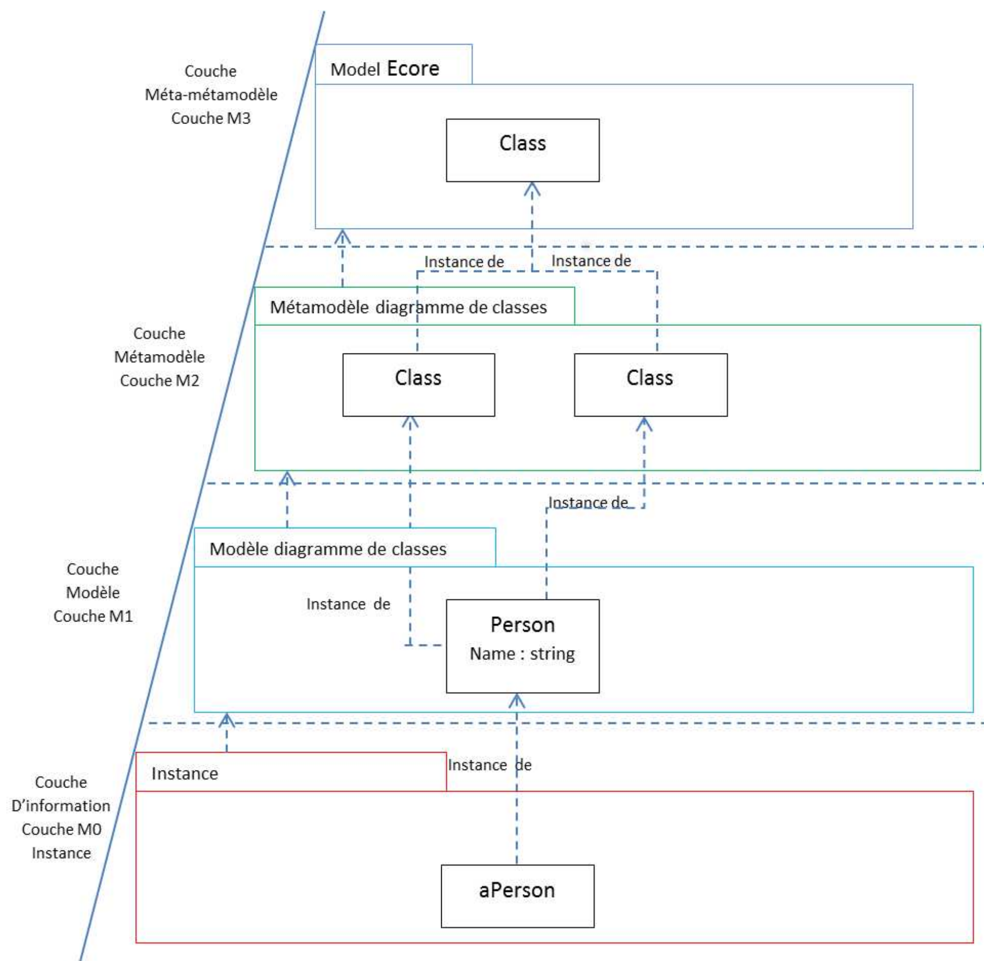

Figure 1.1: L'architecture de métamodélisation

1.3.3 Formalismes de métamodélisation

On a vu que l'architecture de métamodélisation est composée de quatre niveaux : le niveau M0 (les éléments à modéliser), le niveau M1 (les modèles), le niveau M2 (les métamodèles) et le plus haut niveau M3 (les méta-métamodèles). Les langages de métamodélisation se situent au niveau M3 et ils permettent la spécification des métamodèles du niveau M2. Dans l'approche MDA, le langage MOF incarne le socle architectural de la métamodélisation. Dans l'environnement Eclipse, le langage Ecore joue le même rôle de MOF au niveau applicatif [31].

1.3.3.1 Les langages de métamodélisation

Tous les langages de métamodélisation reposent sur les mêmes constructions élémentaires de l'approche orientée objet où le concept de classe est central. Une classe est composée de propriétés qui la caractérisent où une propriété est appelée référence lorsqu'elle est typée par une autre classe, et attribut lorsqu'elle est typée par un type de donnée (ex: booléen, chaîne de caractère et entier).

Le langage de métamodélisation MOF

Le langage MOF (Meta Object Facility) [51] se situe au sommet dans l'architecture à quatre niveaux de l'OMG. C'est un méta-formalisme, pour établir des langages de modélisation permettant eux-mêmes d'exprimer des modèles. Dans sa version 2.0, le méta-métamodèle MOF est constitué de deux parties : EMOF (Essential MOF), pour l'élaboration des métamodèles sans association, et CMOF (Complete MOF) pour l'élaboration des métamodèles avec associations. Il faut souligner que MOF s'auto-décrit pour pouvoir limiter l'architecture de l'OMG à quatre niveaux. Aussi, le langage MOF emprunte la syntaxe du langage de modélisation UML.

Le langage de métamodélisation Ecore

Le langage Ecore a été présenté par la fondation Eclipse dans le cadre du projet EMF (Eclipse Modeling Framework). Le projet EMF propose un framework pour le développement des applications basées sur l'ingénierie dirigée par les modèles. Ce framework permet de générer automatiquement des interfaces Java à partir des métamodèles Ecore.

Le langage de métamodélisation Kermeta

Il est défini comme un langage de métamodélisation exécutable ou de méta-programmation objet, ce qui signifie qu'il permet de décrire des métamodèles dont les modèles sont exécutables. Kermeta est basé sur le langage de métamodélisation EMOF et est intégré à l'IDE Eclipse sous la forme d'un plugin. Il a été conçu comme un tissage entre un métamodèle comportemental et EMOF. Le métamodèle de Kermeta est donc composé de deux packages : Core et Behavior. Le premier correspond à EMOF et le second est une hiérarchie de métaclasses représentant des expressions impératives.

1.3.3.2 Les Profils UML

Les Profils supportent l'adaptation ou l'extension d'UML pour s'adapter à des domaines professionnels ou techniques. On pourrait dire aussi qu'UML n'est pas un seul langage, mais une famille de langages: dans ce cas, les profils UML sont des éléments de "langages concrètes" de cette famille. L'objectif est que les outils UML et les générateurs de diagrammes peuvent traiter les profils comme des plugins. A cet effet, les Profils UML sont dédiés à l'intention stratégique de formaliser et de soutenir le développement d'applications avec UML. Ainsi lorsque UML est incapable de représenter un système ou une application, un profil UML peut donner des mécanismes supplémentaires pour le faire [63].

Exemples de profils UML:
Plusieurs profils UML existent actuellement tels que :

- SysML : SysML est un profil d'UML 2.0 orienté système. La majorité des concepts présents dans SysML sont issus d'UML. Les concepteurs de ce langage insistent sur le fait que SysML (tout comme UML) n'est pas une méthode de modélisation mais bien un langage de modélisation graphique [20].

- Profil UML pour CORBA: CORBA permet aux applications de communiquer entre eux indépendamment de l'emplacement et de la conception, le profil UML pour CORBA fournit une norme pour exprimer la sémantique de CORBA IDL utilisant des artefacts UML, qui permettent l'expression de ces objets avec des outils UML.[50]

- Profil MARTE (*UML profile for Modeling and Analysis of Real Time and Embedded systems*) a pour objectif d'étendre UML pour l'utiliser dans les applications à temps réel et les systèmes embarqués. Il fournit des supports pour les étapes de spécification, de conception et de vérification/validation [4].

1.4 Transformation de modèles

La transformation de modèles est au cœur de l'IDM et y joue un rôle important. Elle peut être définie comme une génération d'un modèle cible à partir d'un modèle

source, suivant une définition de transformation qui est un ensemble de règles de transformation qui décrivent comment un modèle dans un langage source peut être transformé en un modèle dans un langage cible. Une règle de transformation est une description de la manière d'une ou de plusieurs structures du langage source peuvent être transformées à une ou à plusieurs structures du langage cible.
Pour que la transformation soit appliquée à maintes reprises, elle doit être appliquée indépendamment du modèle source [19]. Les concepts de base de transformation sont présentés dans la figure 1.2.

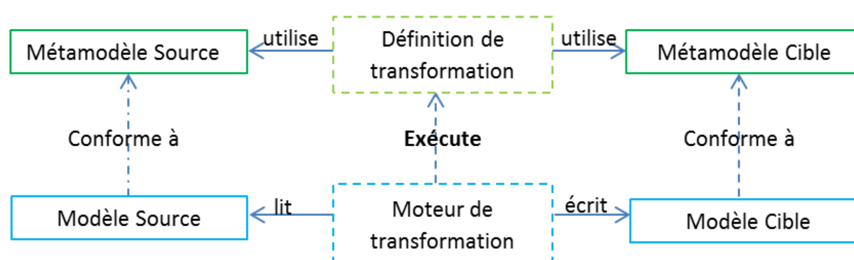

Figure 1.2: Concepts de base de transformation de modèles

1.4.1 Typologie de transformation

Une transformation de modèle génère un modèle cible à partir d'un modèle source. ces deux modèles source et cible sont conformément aux même métamodèle ou à deux métamodèles différents aussi les modèles sources et cibles peuvent appartenir au même niveau d'abstraction (raffinement) ou à deux niveaux d'abstractions différents [22].

Endogène et Exogène

Une transformation de modèles endogène lorsque les modèles source et cible sont conforme au même métamodèle. Donc les règles de transformation sont présentées sur un seul métamodèle. On utilise les transformations endogènes dans le cadre d'optimisation de modèle ou d'une simplification d'un modèle ou dans un refactoring.
Une transformation de modèles exogène lorsque le métamodèle du modèle source est différent du métamodèle du modèle cible. Donc le formalisme qui va exprimer

le modèle source est différent du formalisme qui va exprimer le modèle cible. On utilise les transformations exogènes dans le cadre de migration de modèle d'une plateforme à une autre en restant au même niveau d'abstraction, ou pour la génération de code et aussi les opérations de rétro-ingénierie ou rétro-conception.

Verticalité et horizontalité

Une transformation de modèle peut aussi être classé selon un autre critère qui est le niveau d'abstraction si les modèles cible et source appartiennent au même niveau d'abstraction donc on a une transformation horizontale et si les modèles cible et source appartiennent à deux niveaux d'abstractions différents donc on a une transformation verticale. Par exemple, une génération de code est une transformation verticale.

Le tableau 1.1 présente la synthèse des transformations possibles dans l'IDM

Transformation	Horizontale	Verticale
Endogène	Simplification Restructuration Refactoring	Raffinement
Exogène	Migration de langage	Génération de code Rétro-ingénierie

Tableau 1.1: Synthèse des transformations de modèles

1.4.2 Pourquoi la transformation de modèles

L'objectif principal et final de la transformation de modèles dans le contexte de l'IDM est la génération de code, mais la transformation de modèles couvre plusieurs activités qui sont :

Extraction de modèle

L'extraction de modèle est une transformation de modèle exogène et verticale. L'extraction est l'inverse de la génération de code et c'est l'une des activités essentielles dans l'ingénierie inverse et de la compréhension du programme. Elle

permet de construire un modèle visuel à un niveau d'abstraction supérieur qui permettra de connaître la structure du code.

Traduction de modèle

La traduction de modèle est une transformation de modèle de type horizontale et exogène qui est utilisée dans la recherche académique. Son objectif est de transformer un modèle représenté par un langage de modélisation en un modèle d'un autre langage de modélisation.

Simulation et exécution de modèle

La simulation permet d'avoir une vision sur l'exécution d'un modèle sans l'avoir exécuté réellement. Les simulations sont faites pour des raisons de coûts, de temps et de ressources.

Vérification de modèle

La vérification d'un modèle est le synonyme de modèle correct, c'est à dire le vérifier syntaxiquement et sémantiquement. L'analyse syntaxique vérifie si le modèle est conforme à un métamodèle et s'il respecte les conditions "contraintes" imposées par le langage tandis que l'analyse sémantique vérifie les propriétés dynamiques d'un domaine.

Validation de modèle

La validation est de vérifier si un modèle répond aux exigences, ou d'une autre façon de vérifier si un modèle est fiable. La simulation et le test de modèle sont utilisés pour vérifier si une modèle est valide.

Amélioration de la qualité de modèle

Un type particulier de l'évolution du modèle pour lequel les transformations sont particulièrement utiles est l'amélioration de la qualité du modèle. Les modèles peuvent avoir différents types de critères de qualité qui doivent être satisfaits, mais cette qualité a tendance à se dégrader au fil du temps en raison de nombreux changements qui sont apportés au modèle au cours de sa vie.

Migration et coévolution de modèle

Les transformations de modèles sont également essentielles pour faire face à l'inévitable évolution des modèles. Dans ce contexte, un problème supplémentaire se pose : non seulement les modèles évoluent, mais aussi les langages de modélisation dans lesquels ces modèles sont exprimés. Avec tout changement dans le langage de modélisation (par exemple, une nouvelle version d'UML est présentée), les concepteurs du modèle sont confrontés à la nécessité de migrer leurs modèles pour cette nouvelle version, ou courir le risque que leurs modèles deviennent obsolètes ou incompatibles.

La gestion d'incohérence de modèle
La gestion d'incompatibilité de modèle est également bien adaptée pour être prise en charge par la transformation de modèles. En raison du fait que les modèles sont généralement exprimés en utilisant les points de vue multiples, ils sont en évolution constante, et sont souvent développés dans un cadre de collaboration. Donc les incohérences dans les modèles ne peuvent pas être évitées. Par conséquent, nous avons besoin de techniques basées sur la transformation de modèle pour réparer les incohérences.

1.4.3 Propriétés de transformation de modèles

Les principales propriétés qui caractérisent les transformations de modèles sont :

- **Traçabilité :** La traçabilité est la propriété d'avoir un dossier de liens entre les éléments de modèle source et ceux du modèle cible ainsi que les différentes étapes du processus de transformation. Les liens de traçabilité peuvent être stockés soit dans le modèle source, soit dans le modèle cible ou dans un modèle à part.
- **Directivité ou réversibilité :** une transformation est dite réversible si elle peut se faire dans les deux sens. Exemple : Model to text et text to Model. On dit en outre qu'une transformation est réversible s'il existe une transformation permettant de retrouver le modèle source à partir du modèle cible.
- **Réutilisabilité :** la Réutilisabilité peut être mesurée avec la possibilité d'adapter et de réutiliser les règles d'une transformation de modèle dans

Clicours.COM

d'autres transformations. L'identification de patrons de transformation est un moyen pour mettre en œuvre cette réutilisabilité.

- **Ordonnancement :** l'ordonnancement consiste à représenter la suite des règles à exécuter lors d'une transformation. En effet, les règles de transformation peuvent déclencher d'autres règles.

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

1.4.4 Approches de transformation de modèles

Les approches de transformation de modèles sont classées en deux grandes familles : les approches de modèle au code (Model to Text M2T) où le code est considéré comme un modèle spécifique et les approches de modèle à modèle (Model to Model M2M).

Le tableau 1.2 ci-dessous, illustre les approches de transformation de modèles.

<table>
<tr><th colspan="2">Approches de transformation de modèles</th></tr>
<tr><td>M2M</td><td>M2T</td></tr>
<tr><td>Dirigée par la structure</td><td rowspan="2">Parcours de modèles (programmation)</td></tr>
<tr><td>Manipulation directe</td></tr>
<tr><td>Approche relationnelle</td><td rowspan="3">Template</td></tr>
<tr><td>Transformation de graphes</td></tr>
<tr><td>Hybride</td></tr>
</table>

Tableau 1.2: Les approches de transformation de modèles

clicours.com

1.4.4.1 Model to Text

Cette classe est caractérisée par la méthode de génération du code source qui se base soit sur le parcours du modèle soit sur l'identification de canevas (templates).

Génération de code par parcours de modèle
C'est une approche de génération de code la plus basique qui consiste à parcourir la représentation du modèle en entrée et a écrire un code en sortie. L'exemple le plus connu avec cette approche est l'environnement orienté objet Jamda [34] dont les modèles UML sont représentés par des classes où une API manipule les modèles et un outil est utilisé pour la génération de code.

Génération de code par template
C'est l'une des approches la plus utilisée pour la génération de code. Cette approche consiste à définir des canevas de modèles cibles. Ces canevas sont des modèles cibles paramétrés ou modèles templates. L'exécution d'une transformation consiste à prendre un modèle template et à remplacer ses paramètres par les valeurs d'un modèle source. D'une autre façon , elle consiste à utiliser comme RHS *Right Hand Side* (le côté droit de la règle) un texte fixe dans lequel certaines parties sont variables : elles sont renseignées en fonction des informations récupérées dans le modèle source LHS *Left Hand Side* (le côté gauche de la règle), et peuvent notamment faire l'objet d'itérations. Parmi les langages les plus utilisées, on a Xpand [64], Acceleo [1].

1.4.4.2 Model to Model

Cette catégorie porte sur la transformation de modèle à modèle. Les méthodes utilisées sont variées qu'on peut résumer comme suit :

Approche par manipulation directe
Cette approche offre une représentation interne d'un modèle plus une API pour le manipuler. ils sont généralement implémentés dans un environnement orienté objet, qui peut également fournir une infrastructure minimale pour organiser les transformations. Mais les utilisateurs doivent implémenter les règles de transformations à partir de zéro en utilisant un langage de programmation comme

Java. Des exemples de cette approche comprennent Jamda [34] et JMI(Java Metadata Interface est une API basée sur le MOF). L'approche par manipulation directe est de type impératif, elle est située à un bas niveau d'abstraction.

Approche relationnelle

Elle se base sur les transformations de type déclaratif reposant sur les relations mathématiques où l'idée de base est d'indiquer les types d'éléments de la source et de la cible dans une relation et de les spécifier à l'aide des contraintes. Par définition, une telle transformation n'est pas exécutable telle quelle, mais son exécution est possible par l'adjonction d'une sémantique exécutable, par exemple au moyen de la programmation logique. Il n'y a pas d'effets de bord avec l'approche relationnelle, contrairement à l'approche par manipulation directe. La plupart des transformations basées sur l'approche relationnelle permettent la multidirectionnalité des règles, et fournissent des liens de traçabilité. Elles n'autorisent pas la transformation sur place (source et cible doivent être distinctes). Des exemples sur cette approche sont RFP [30], MOF 2.0 [51], QVT [52].

Approche dirigée par la structure

Cette approche comporte deux phases distinctes :

- La première phase concerne la création de la structure hiérarchique du modèle cible.
- la deuxième phase définit les attributs et les références du modèle cible.

L'environnement de transformation détermine la stratégie d'application et d'ordonnancement des règles. Les utilisateurs ne sont concernés que par fournir les règles de transformation. Une des particularités de cette approche est que l'organisation des règles se fait en fonction de la cible : pour chaque type d'élément cible, il existe une règle. Ainsi, l'imbrication des règles correspond à la structure du métamodèle cible. Un exemple de cette approche est fourni dans OptimalJ [53]. C'est un environnement implémenté en Java où la transformation s'obtient en créant des sous-classes qui héritent de classes de transformation génériques.

Approche par transformation de graphes
Cette catégorie d'approches de transformation de modèles s'appuie sur la théorie de graphes. Une approche par transformation de graphes utilise usuellement des graphes typés, orientés et étiquetés où les règles de transformation de graphes possèdent un schéma LHS et un schéma RHS : le schéma LHS est repéré dans le modèle source et remplacé par le schéma RHS dans le modèle cible.

Il existe de nombreuses approches M2M basées sur la transformation de graphes dont certaines sont intégrées dans des environnements. Les plus connues sont : AGG (Attributed Graph Grammar) [2], l'environnement VMTS (Visual Modeling and Transformation System) [62], l'environnement AToM3 [7], BOTL (Bidirectional Object-oriented Transformation Language) [45], l'outils Fujaba (From UML to Java And Back Again)[21], GReAT (Graph REwriting And Transformation) [8], MOLA (MOdel transformation LAnguage)[35] [36], VIATRA (Visual Automated model TRAnsformations) [61] et le langage de transformations graphique UMLX [60].

Approche hybride
Les approches hybrides combinent différentes techniques des approches précédentes, notamment grâce à leur puissance d'expression. Elles permettent, selon le contexte, de spécifier des règles de manière impérative, déclarative, ou bien de mélanger les deux. Par conséquent, la stratégie d'application de ces règles peut être soit implicite (gérée par l'environnement de transformation) soit explicite (gérée par l'utilisateur). Les approches hybrides se situent sur plusieurs approches que nous venons de voir et permettent de combiner leurs avantages.

1.4.5 Les outils de transformation

On peut d'une manière générale distinguer quatre catégories d'outils :

1.4.5.1 Outils génériques

Dans cette catégorie, on trouve les outils de la famille XML, comme XSLT ou Xquery , et les outils de transformation de graphes. Les outils de la famille XML ont été déjà largement utilisés dans le monde XML, ce qui leur a permis d'atteindre

un certain niveau de maturité du fait d'une large diffusion dans le monde XML. En revanche, des retours d'expérience montrent que le langage XSLT est assez mal adapté pour des transformations de modèles complexes. En effet, selon Muller [49] les transformations XSLT ne permettent pas de travailler au niveau de la sémantique des modèles manipulés mais seulement au niveau d'un arbre couvrant le graphe de la syntaxe abstraite d'un modèle, ce qui rend ce type de transformation fastidieuse dans le cas des modèles complexes.

1.4.5.2 Outils intégrés aux AGLs

Dans cette catégorie, on va trouver une famille d'outils de transformation de modèles proposés par des ateliers de génie logiciel. Par exemple, l'outil Arcstyler [40] de Interactive Objects propose la notion de MDA-Cartridge qui encapsule une transformation de modèles écrite en JPython (langage de script construit à l'aide de Python et de Java).
L'intérêt de cette catégorie d'outils est leur maturité et leur excellente intégration dans l'atelier qui les héberge. L'inconvénient en est que la majeure partie d'entre eux intègre des langages de transformation propriétaires qui s'adaptent difficilement à l'évolution de l'IDM. Ces outils montrent aussi leur limite lorsque les transformations de modèles deviennent complexes et qu'il faut les gérer, les faire évoluer et les maintenir sur une longue période

1.4.5.3 Langages spécifiques

Dans cette catégorie, on a les outils conçus spécifiquement pour faire de la transformation de modèles et prévus pour être plus ou moins intégrables dans les environnements de développement standards. On trouve par exemple, Mia-Transformation [48] de Mia-Software [MS] qui est un outil qui exécute des transformations de modèles prenant en charge différents formats d'entrée et de sortie (XMI, tout autre format de fichiers, API).

1.4.5.4 Outils de métamodélisation

La dernière catégorie d'outils de transformation de modèles est celle des outils de métamodélisation dans lesquels la transformation de modèles revient à l'exécution

d'un méta programme. Le plus ancien et le plus connu de ces outils est certainement Kermeta de TOPCASED. Les outils de métamodélisation proposent des technologies qui permettent de faire des transformations impératives (cas de Kermeta), qui engendrent la syntaxe concrète d'un langage à partir de sa syntaxe abstraite (c'est le cas de TOPCASED), et qui permettent l'interopérabilité entre les outils IDM (cas de l'environnement de modélisation EMF).

1.5 Conclusion

Dans ce chapitre, nous avons présenté les notions de base de l'ingénierie dirigée par les modèles où nous avons mis l'accent sur la transformation de modèles. Nous avons aussi décrit le rôle principal de la transformation dans le cadre de l'approche IDM et ses activités. L'objectif visé dans notre projet est d'intégrer la notions des grammaires de graphes dans le processus de transformation de modèles. Le concept de grammaire de graphes est devenu très important pour simplifier l'utilisation des langages et de logiciels de l'ingénierie dirigée par les modèles. Le chapitre suivant explicitera plus particulièrement le concept de la transformation de graphes.

Chapitre 2

Grammaires de graphes

2.1 Introduction

Dans ce chapitre, nous examinons les concepts de base des grammaires de graphes qui ont été choisies comme formalisme sous-jacent pour exprimer la transformation de modèles. L'approche de grammaires de graphes dérive un graphe en un autre graphe en appliquant des règles de production de la grammaire l'une après l'autre. Chaque règle appliquée transforme un graphe par l'ajout, la suppression ou le remplacement d'une partie d'un graphe dans un autre graphe.

Les transformations de graphes ont évolué dans l'expressivité des approches de grammaires classiques comme les grammaires de Chomsky en termes de s'y prendre avec les structures non linéaires.
Une transformation de graphe consiste en l'application d'une règle à un graphe et d'itérer ce processus. Chaque application de règle transforme un graphe par le remplacement d'une de ses parties par un autre graphe. Autrement dit, la transformation de graphe est le processus de choisir une règle parmi un ensemble de règles, appliquer cette règle à un graphe et réitérer le processus jusqu'à ce qu'aucune règle ne puisse être appliquée.

2.2 Notions sur les graphes

Avant d'expliciter les fondements de la transformation de graphe, nous devons présenter les notions de base d'un graphe: graphe, sous graphe et morphisme de graphe....

2.2.1 Définition d'un graphe

Un graphe est constitué d'un ensemble de sommets reliés par des arêtes et où deux sommets reliés par une arête sont adjacents.
L'ordre d'un graphe est le nombre de ses sommets.
Le degré d'un sommet est le nombre d'arêtes dont le sommet est une extrémité.

La figure 2.1 illustre un exemple de graphe non orienté, d'ordre 5.

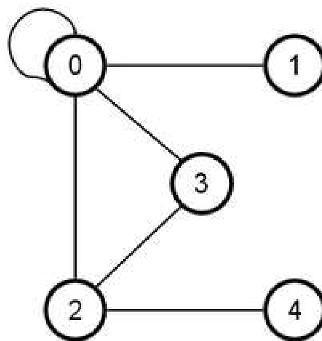

Figure 2.1: Graphe non orienté

2.2.2 propriété d'un graphe

Sous-graphe
Un sous-graphe d'un graphe G est un graphe G' composé de certains sommets de G, ainsi que toutes les arêtes qui relient ces sommets. La figure 2.2 illustre un exemple où le graphe (b) est un sous-graphe du graphe (a).

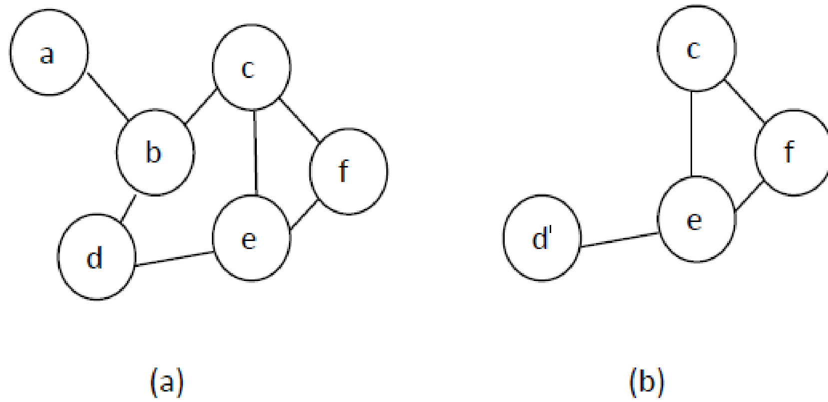

Figure 2.2: (a) Graphe G (b) Sous graphe de G

Graphe orienté

Un graphe orienté est un graphe dont les arêtes sont munies de directions : nous parlons alors de l'extrémité d'une arête. Dans un graphe orienté, une arête est appelée "arc".
La figure 2.3 illustre un exemple de graphe orienté.

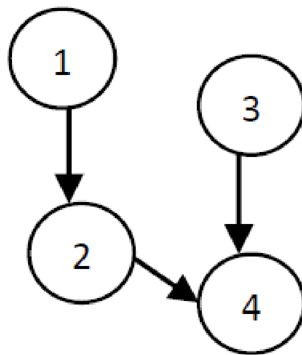

Figure 2.3: Graphe orienté

Graphe étiqueté

Un graphe étiqueté est un graphe orienté, dont les arcs sont munis d'étiquettes. Si toutes les étiquettes sont des nombres positifs, on parle de graphe pondéré. La figure 2.4 illustre un exemple de graphe orienté étiqueté.

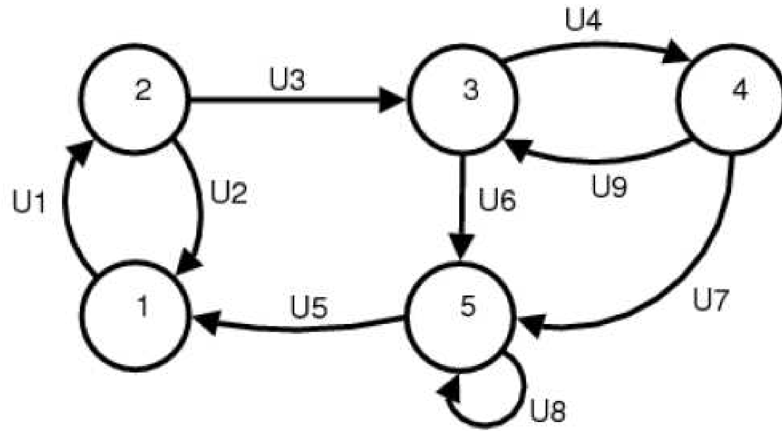

Figure 2.4: Graphe étiqueté

Morphisme de graphe

C'est une paire de fonctions qui transforment un graphe en un autre, en conservant la structure algébrique (les nœuds et les arrêtes).
Soient G et H deux graphes. Un morphisme de graphe m : G → H est une paire de fonctions $m_V : v_G \rightarrow v_H$ et $m_E : E_G \rightarrow E_H$ telles que :

- $s_H \circ m_E = m_V \circ sG$ et $tH \circ m_E = m_V \circ t_G$ (préservent les nœuds source et cible des arrêtes).
- $l_H \circ m_V = l_G$ et $l_H \circ m_E = l_G$ (préservent les étiquettes des nœuds et des arrêtes).

Pushout

Étant donné deux morphismes p : L → R et m : L → G dans une catégorie C, un pushout (H, p', m') sur p et m est défini par :
Un objet (graphe) H et,
Un paire de morphismes p' : G → H et m' : R → H tels que p' ∘ m = m' ∘ p.
Le pushout vérifie la propriété : pour tout objet X et morphismes h : R → X et, k : G → X tels que k ∘ m=h ∘ p, alors il existe un unique morphisme x : H → X tel que : x ∘ m' = x ∘ p

2.3 Grammaire de graphes

Une grammaire de graphes distingue les graphes non terminaux, qui sont les résultats intermédiaires sur lesquels les règles sont appliquées, des graphes terminaux dont on ne peut plus appliquer de règles. Ces derniers sont dans le langage engendré par la grammaire de graphe, pour vérifier si un graphe G est dans les langages engendrés par une grammaire de graphe, il doit être analysé. Le processus d'analyse va déterminer une séquence de règles dérivant G [12].

2.3.1 Définition formelle

Une grammaire de graphes est un quadruplet R = (T,N, V, P) où

- T est un alphabet d'étiquettes terminales,
- N est un alphabet d'étiquettes non-terminales ou variables,
- V est un ensemble quelconque de sommets,
- P est un sous-ensemble fini de règles (x,H) ou le membre gauche $x \in NV*$ est un hyperarc non-terminal et $H \subseteq NV * \to TVV$ est un hypergraphe fini d'hyperarcs non-terminaux et d'arcs terminaux.

Une règle est définie par une paire de graphes :

- LHS (Left Hand Side) est un graphe de côté gauche présentant la précondition de la règle et doit être un sous graphe de G.
- RHS (Right Hand Side) est un graphe de côté droit présentant la post-condition de la règle et doit remplacer LHS dans G.

Une grammaire de graphes est déterministe si deux membres gauches ont des étiquettes différentes.

2.3.2 Grammaire de graphe et langage

On peut associer un langage à un graphe en considérant l'ensemble des mots qui étiquettent ses chemins entre deux ensembles déterminés de sommets.
Formellement,on appelle ensemble de traces d'un graphe G à étiquettes dans $\sum$ entre les ensembles de sommets I et F, ou simplement langage de G.

$$\text{L(G,I,F)} = \{u \in \sum * \mid \exists s \in I, \exists t \in F, (s \underset{\text{G}}{\overset{\text{u}}{\rightarrow}} t)\}.$$

La spécification des ensembles des sommets initiaux et finaux autorisés se révèle importante pour les traces d'un graphe ou d'une famille de graphes. Par exemple, cela induit des différences importantes de ne considérer que des ensembles finis plutôt que des ensembles réguliers ou algébriques. Nous dirons que deux familles de graphes F_1 et F_2 sont trace-équivalentes par rapport à deux familles K_I et K_F de sous-ensembles de V si, pour tous G $\in F_i$, I $\in K_I$, F $\in K_F$, il existe un graphe H $\in F_{3?i}$ et deux ensembles I'$\in$ K $_I$ et F'$\in K_F$ tels que L(G,I,F) = L(H,I',F').

2.3.3 Dérivation

Une correspondance p: L $\underset{r}{\longrightarrow}$ R dans un graphe G et un morphisme de graphe m: L $\rightarrow$ G.
Étant donné un p une production et un m une correspondance pour p dans le graphe G, la dérivation directe de G avec p à m, écrite, G $\underset{p,m}{\longrightarrow} *$ H , se fait comme suit:

- En utilisant le morphisme m, supprimer les sommets et les arêtes de G qui apparaissent dans L et n'apparaissent pas dans R.
- Ajouter à G tous les sommets et les arêtes qui apparaissent dans R mais n'apparaissent pas en L.
- Supprimer tous les arêtes qui n'ont pas un nœud source ou cible dans G.

Intuitivement, l'application d'une règle graphique p: L $\underset{r}{\longrightarrow}$ R à un graphe G fonctionne comme suit: remplacer l'occurrence de L dans G par R. Supprimer les

arêtes dont le nœud source ou cible sont supprimés.

Nous nous limitons à des grammaires pour lesquelles chaque membre gauche est d'arc sortant de 2, ni d'arc entrant en 1.

Une grammaire R est un ensemble fini de règles de la forme :

$$(1, A, 2) \rightarrow H$$

où (1, A, 2) est un arc étiqueté par A de source 1, de but 2, et H est un graphe fini.

A partir de n'importe quel graphe axiome, nous voulons une grammaire de graphes pour engendrer un graphe unique à isomorphisme près. Nous devons donc nous limiter aux grammaires déterministes pour lesquelles deux règles ont des membres gauches distincts :

$$((1, A, 2),H) , ((1, A, 2),K) \in R \Rightarrow H = K.$$

Figure 2.5: Une grammaire déterministe de graphes

A partir d'un graphe, cette grammaire engendre un graphe unique obtenu en appliquant indéfiniment des réécritures parallèles. Pour toute grammaire R, la réécriture $\underset{R}{\longrightarrow}$ est la relation binaire entre graphes définie par M $\underset{R}{\longrightarrow}$ N si nous pouvons choisir un arc non-terminal X = (s, A, t) dans M et une partie droite H de A dans R pour remplacer X par H dans M.

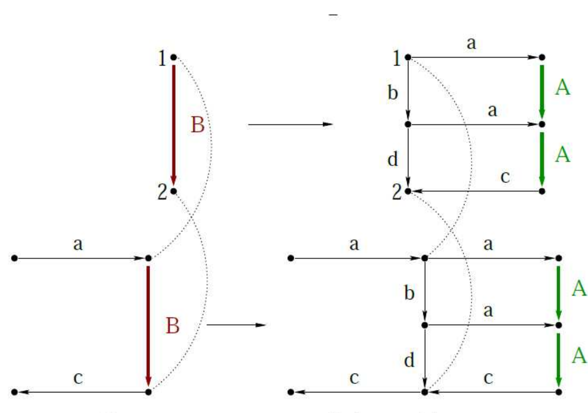

Figure 2.6: Réécriture pour la grammaire de la figure 2.5

pour une injection h qui associe le sommet 1 à s, le sommet 2 à t, et les autres sommets de H à des sommets autres que ceux de M. Cette réécriture est notée par $M \underset{R,X}{\longrightarrow} N$.

La réécriture $\underset{R,X}{\longrightarrow}$ d'un arc non-terminal X est étendue de manière évidente à la réécriture $\underset{R,E}{\longrightarrow}$ de tout sous-ensemble E d'arcs non-terminaux. La réécriture parallèle $\underset{R}{\longrightarrow}$ est la réécriture de l'ensemble des arcs non-terminaux : $M \underset{R}{\longrightarrow} N$ si $M \underset{R,E}{\longrightarrow} N$ où E est l'ensemble des arcs non-terminaux de M

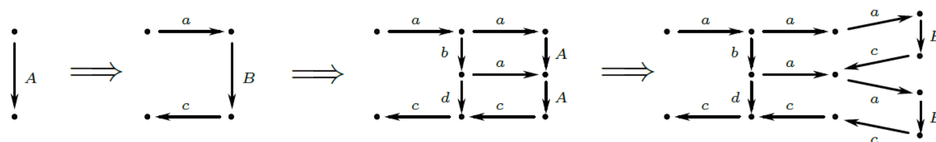

Figure 2.7: Réécritures parallèles pour la grammaire de la figure 2.5

Grâce aux deux arcs non-terminaux dans la partie droite de B pour la grammaire de la figure 2.5 , nous obtenons après n réécritures parallèles appliquées à partir de H0 = (1, A, 2), un graphe Hn ayant un nombre exponentiel d'arcs.
La dérivation $\underset{R}{\longrightarrow} *$ est la fermeture réflexive et transitive pour la composition de la réécriture parallèle $\underset{R}{\longrightarrow}$ i.e. $G \underset{R}{\longrightarrow} * H$ si H est obtenu de G par une suite

consécutive de réécritures parallèles.

Par exemple, en itérant indéfiniment la dérivation de la figure 2.7, nous obtenons le graphe infini ci-dessous.

Figure 2.8: Graphe engendré par la grammaire dans la figure 2.5

2.3.4 Transformation de grammaire

On peut transformer toute grammaire algébrique G = (T, N, P) en une grammaire de graphes R = (T,N,V,P') ou P'= (A1,2,HA) | A ∈ N de sorte que pour tout non terminal A ∈ N, $L_G(A) = L_R(A)$.

Par exemple, on considère la grammaire algébrique G = (T, N, P) où T = a, b, c, d, N = A,B et où les règles sont :
A → Bb
A → aAB
B → c
B → BBd

La construction de la proposition 1 associe à la grammaire algébrique G la grammaire de graphes R constituée des deux règles comme la figure

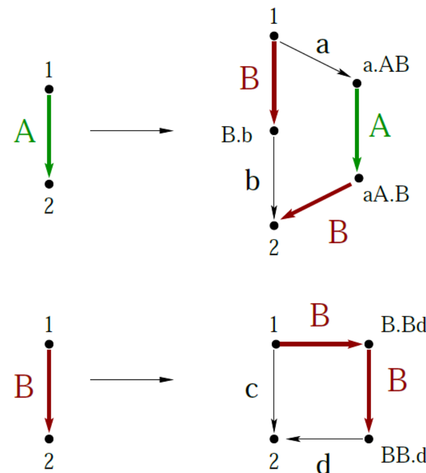

Figure 2.9: Une grammaire de graphe

La réciproque de la proposition 1 est vraie.

transformation d'une grammaire graphique en une grammaire textuelle
On peut transformer toute grammaire R de graphes en une grammaire algébrique G telle que $L_R(A) = L_G(A)$ pour tout $A \in N_R$.

Par exemple, la grammaire de graphes de la figure 2.5 est transformée en la grammaire:

$A = aC$; $B = aF + bE$; $C = BD$
$D = c$; $E = aG + d$; $F = AG$
$G = AH$; $H = c$

2.4 Principe de transformation de graphes

Les approches de réécriture classiques comme les grammaires de Chomsky ou la réécriture de termes manquent d'expressivité. Les transformations de graphes ont évolué pour y remédier. Le processus de transformation de graphes consiste

Clicours.COM

en l'application itérative d'une règle à un graphe. Chaque application de règle remplace une partie du graphe par une autre suivant la définition de la règle. La mécanique de la transformation de graphe fonctionne de la façon suivante : sélectionner une règle applicable de l'ensemble des règles; appliquer cette règle au graphe d'entrée; rechercher une autre règle applicable (réitération) jusqu'à ce qu'aucune règle ne puisse plus être appliquée. Cette opération est basée sur un ensemble de règles respectant une syntaxe particulière, appelé modèle de grammaire de graphe.

Un modèle de grammaire de graphe est une généralisation des grammaires de Chomsky. C'est une composition de règles où chaque règle possède deux parties : une partie gauche (LHS : Left Hand Side) et une partie droite (RHS : Right Hand Side). La partie gauche LHS correspond au graphe d'entrée (appelé aussi Host Graph). La règle compare à chaque fois qu'elle est invoquée son LHS avec le graphe sur lequel nous appliquons la transformation. La règle remplace l'équivalent de son LHS dans le graphe à transformer par sa partie droite RHS. Le RHS décrit la modification du host graph.

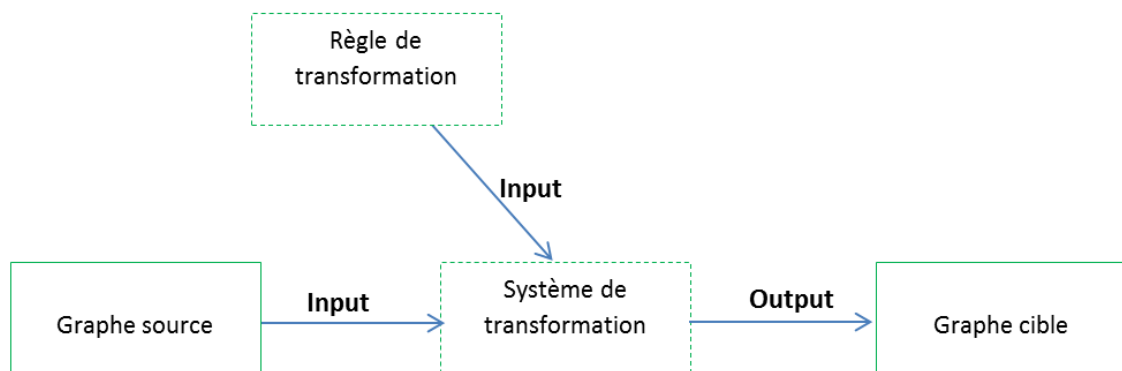

Figure 2.10: Principe de mise en œuvre de transformation de modèles

2.5 Approches de transformation de graphes

Il existe plusieurs approches de transformation de graphes, ou chaque approche a sa propre définition de règle. Ces dernières sont présentées dans les paragraphes ci-dessous.

2.5.1 Approche algèbrique

L'approche algébrique est basée sur la théorie des catégories. l'appellation algébrique a comme origine le collage qui se fait par un 'pushout' dans une catégorie appropriée. Cette approche est divisée, principalement en deux sous approches qui sont SPO (Single PushOut)[44] et DPO (Double PushOut) [15].

- **Double PushOut:**
  Une règle de production est donnée par p = (L, K, R), où L et R sont les parties gauche et droite de la règle et K est l'interface commune entre L et R, qui n'est autre que leur intersection. La partie gauche L représente les pré-conditions de la règle, tandis que la partie droite R décrit les post-conditions. K décrit une partie du graphe qui doit exister pour appliquer la règle, mais qui n'est pas modifiée. K $\rightarrow$ L est un morphisme de graphe décrivant la partie qui va être supprimée, et K $\rightarrow$ R est un morphisme de graphe décrivant la partie qui va être créée.

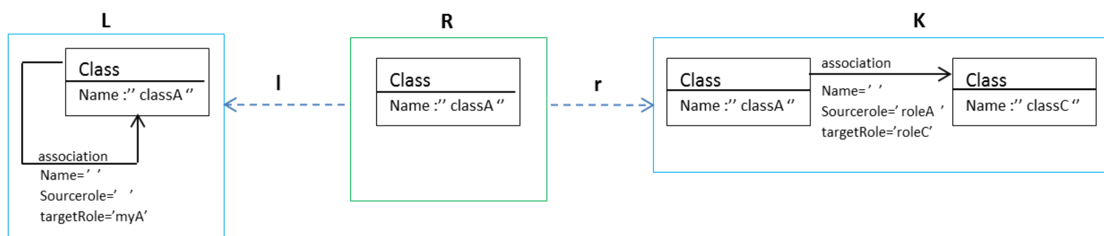

Figure 2.11: Exemple d'application d'une règle selon DPO

- **Single PushOut:**
  La réécriture d'un graphe G en un graphe G' revient à remplacer un sous graphe L de G par un graphe R. G' est le graphe résultant de ces deux opérations. G est appelé graphe hôte, L est appelé graphe mère et R est appelé graphe fille. Une règle de réécriture est sou forme p : L → R. ce type de règle est applicable à un graphe G s'il existe une occurrence de L dans G [18].

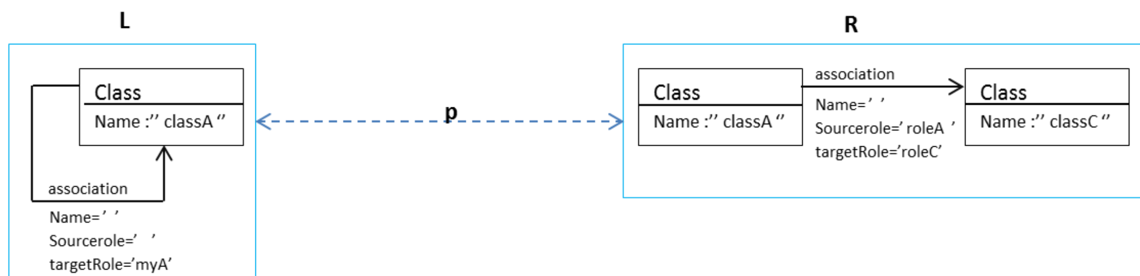

Figure 2.12: Exemple d'application d'une règle selon SPO

- **Différence entre SPO et DPO**
  - DPO est plus complexe que SPO.
  - Des fonctions partielles comme morphismes au lieu des fonctions totales.
  - Un seul pushout.
  - Pas de réversibilité des règles.
  - Peut être seulement faible pushout (weak pushout).
  - Plus général que le double pushout (plus de règles possibles).

2.5.2 TGG

Dans son travail, Andy Schurr [56] a essayé de prolonger les "pair graph grammar" en "Triple Graph Grammar". Contrairement à pair graph grammar, Triple Graph Grammar soutiennent les productions contextuelles avec des côtés assez complexes à gauche et à droite. Généralement, la séparation des objets de correspondance

permet la modélisation de relations m-à-n entre les côtés connexes. L'approche TGG fait une distinction claire entre les modèles source et cible, il garde aussi les liens supplémentaires nécessaires pour spécifier les transformations comme une spécification distincte. [37]

L'avantage de triple graph grammar sur les autres approches réside dans la définition des relations inter-graphe, qui offrent la souplesse nécessaire pour utiliser les productions à la fois pour la transformation vers l'avant et vers l'arrière et l'analyse de la correspondance. triple graph grammar, comme une définition déclarative de la correspondance entre les deux graphes, peut être utilisé pour la mise en œuvre d'un traducteur dans les deux sens. Pour plus de details voir le chapitre 4

2.6 Conclusion

Dans ce chapitre, nous avons présenté les grammaires de graphes et le principe de transformation de graphe. En outre, nous avons introduit dans ce chapitre quelques-unes des définitions formelles de grammaires de graphes. Les mécanismes de transformation de graphe comme SPO ou DPO et TGG sont également illustrés dans ce chapitre. Un des aspects essentiels de manipulation des approches de transformation de graphe consiste à automatiser la transformation grâce à l'utilisation de langages et d'outils de transformation implémentant ces approches. Au-delà du principe et des concepts de transformation de graphes, les outils de transformation varient beaucoup, surtout dans la mise en œuvre. Donc, il est nécessaire de les classifier et les comparer. Ceci est fait dans le chapitre suivant.

Chapitre 3

Outils de transformation de graphes

3.1 Introduction

L'utilisation généralisée de n'importe quelle technologie ou approche dépend fortement de la disponibilité de ses outils. Pratiquement la transformation de modèles basée sur l'utilisation de la transformation de graphes ne fait pas exception. Depuis un certain temps, une variété d'outils de transformation de graphes sont disponibles et ont considérablement évolué. Afin d'utiliser un de ses outils dans notre projet de magister, un besoin de leur classification s'est imposé pour apporter une aide de choix d'utilisation et justifier ce dernier.

Le chapitre sera organisé comme suit : nous présentons dans la section 2 les outils de transformation de graphes. La section 3 présente une étude comparative sur les outils étudiés selon différents critères. Dans la section 4, nous concluons ce chapitre en motivant notre choix de l'outil à utiliser dans notre projet.

3.2 Outils de transformation

Les outils de transformation présentés dans cette section sont orientés vers la transformation de graphes basée sur les grammaires de graphes. Il est important de noter que les outils choisis pour l'étude comparative sont des outils "open-source".

clicours.com

3.2.1 AGG

AGG " Algebraic Graph Grammar" est un environnement à usage général pour le développement des systèmes de transformation de graphes attribués. Il est basé sur l'approche algébrique pour la transformation de graphes. Il vise à la spécification et le prototypage rapide d'applications complexes tel que le graphe de données structurées. Puisque la transformation de graphe peut être appliquée à des niveaux d'abstraction très différents, elle peut être attribuée ou non par des calculs simples ou par des processus complexes, selon le niveau d'abstraction. En raison de sa base formelle, AGG offre un support de validation, de vérification de la cohérence des graphes en fonction des contraintes graphiques. [57]

3.2.2 ATOM3

ATOM3 "Domain Specific Visual Languages" [17] est un outil pour la conception de Domain Specific Visual Languages (DSML). Il permet de définir la syntaxe abstraite et concrète d'un langage visuel au moyen de la métamodélisation et d'exprimer la manipulation du modèle en utilisant la transformation de graphe. Avec les informations de métamodèle, ATOM3 génère un environnement de modélisation personnalisé pour le langage décrit. Récemment, ATOM3 a été étendu avec des fonctionnalités pour générer des environnements avec des vues multiples de langages visuels (tels que UML) et Triple Graph Grammar (TGG) [56]. Ce dernier est utile pour exprimer l'évolution de deux modèles différents, liés par un modèle intermédiaire.

3.2.3 GROOVE

GROOVE "GRaph of Object Oriented VErification" est un outil d'usage général basé sur la transformation de graphes pour modéliser la conception, la compilation et la structure d'exécution des systèmes orienté objet. La transformation de graphes avec GROOVE se base sur la transformation de modèles et la sémantique opérationnelle. Cela implique l'usage d'un fondement formel de transformation de modèles et de sémantique dynamique, et aussi la capacité de vérifier la transformation de modèle en utilisant le model checking. L'outil GROOVE a un éditeur pour la création des règles de production graphiques, un simulateur pour calculer

visuellement les transformations de graphe induit par un ensemble de règles de production graphique. [23] [29]

3.2.4 GrGen

GrGen est un outil de transformation de graphes utilisé dans différents domaines tels que la transformation de données graphiques complexes, la linguistique informatique, ou la construction de compilateur moderne. GrGen permet de travailler à un niveau très abstrait en utilisant le modèle déclaratif. Le code généré par GrGen s'exécute très rapidement. En termes de performance par rapport aux autres outils GrGen permet la manipulation et la transformation des systèmes complexes. Le système GrGen est écrit en Java et C. Son noyau est un générateur de grammaires de graphe [28].

3.2.5 Fujaba

L'outil FUJABA (**F**rom **U**ML to **J**ava **A**nd **B**ack **A**gain) utilise les classes UML et les diagrammes pour produire du code Java, afin de fournir uns conception de systèmes formels et de langages spécifiques. Les fonctionnalités de la rétro-ingénierie (reverse engineering) y compris la conception de reconnaissance de formes sont également disponibles. Le métamodèle du FUJABA peut être étendu par héritage à la place de l'instanciation [43].

3.2.6 EMF Tiger

EMF Tiger (**T**ransformat**i**on **ge**ne**r**ation) est un environnement pour la transformation de modèles EMF. Les transformations sont définies et exécutées directement sur des modèles EMF soit à l'aide d'un code généré ou soit avec un interpréteur afin d'assurer la sécurité et l'efficacité de types d'artifacts.

Les règles basée sur Le langage de transformation EMF Tiger est inspiré par les concepts de transformation de graphes et combinent les deux aspects déclaratif et procédural. En particulier, les règles de transformation sont déclaratives dans le sens où un modèle structurel est utilisé pour définir les pré-condition d'une règle. Le concept procédural est représenté par la structure de contrôle de flux tels que

les couches et les boucles peuvent être utilisés pour appliquer un ensemble de règles de transformation d'une manière contrôlée. [11]

3.2.7 Viatra

Viatra est un outil basé sur Eclipse pour un usage général de l'ingénierie de transformation de modèles. C'est un framework qui supporte un cycle de vie complet pour la spécification, la conception, l'exécution, la validation et la maintenance des transformations entre les différentes langages et les domaines de modélisation. VIATRA2 est apte à coopérer avec un système externe arbitraire et exécuter la transformation avec une transformation de modèle natif (plugin), qui est généré par VIATRA2. Le langage de spécification des règles combine la transformation de graphes et les machines d'état abstraites en un seul paradigme. Essentiellement, les étapes de transformation élémentaires sont capturés par des règles de transformation graphiques, tandis que des transformations complexes sont assemblés à partir des étapes élémentaires en utilisant les règles de machine d'états abstraites pour la spécification du flux de contrôle [58].

3.3 Etude comparative sur les outils de transformation de graphes

Afin de présenter notre étude comparative sur les outils de transformation qui se basent sur l'utilisation de la théorie des grammaires de graphes, nous établissons un ensemble de critères qui serviront comme éléments de comparaison.

3.3.1 Critères de comparaison

Il existe dans la littérature des études comparatives d'outils de transformation. Nous pouvons citer les travaux de G. Rozenberg [46] et R. Bardhol [9] dont l'étude de comparaison repose sur le domaine d'application des outils de transformation de graphes. Dans [3], A. Agrawal traite l'aspect d'infrastructure à offrir pour éditer les modèles et exécuter les transformations. D'autres comparaisons plus récentes se concentrent sur des aspects polyvalents tels que la comparaison de T. Mens [47], et celle de G. Taentzer et al. [58]. chaque étude prend quelques outils, on a essayé

de prendre tous les outils opensource pour les transformation de graphes et faire une comparaison. L'idée de base de notre étude comparative est d'analyser et de faire ressortir tous les critères pertinents des comparaisons proposées dans [46], [9], [3], [47], [58], [5]. Dans notre étude, nous ajoutons quelques critères comme l'exploration et le contrôle et on a . Ces aspects sont développés ci-dessous.

3.3.1.1 Nombre de source et cible

Cet critère présente le nombre de graphes sources et le nombre de graphes cibles disponibles après la transformation.

3.3.1.2 Type de transformation

Les transformations exogènes où le modèle source et le modèle cible ont des métamodèles différents, peuvent être exprimées en utilisant des transformations de graphes à condition qu'un type de graphe différent peut être utilisé pour le modèle source et le modèle cible. Ce type de transformation est requis par GrGen et EMF Tiger. Dans les transformations endogènes, les modèles source et cible sont des instances du même métamodèle. Ce type de transformation est requis par l'outil AGG, ATOM3, GROOVE et FUJABA, où un seul métamodèle peut être spécifié pour un système de transformation de graphe donné. Pour Viatra, les deux types de transformation exogène et endogène peuvent être utilisées.

3.3.1.3 Approche utilisée

Dans le chapitre précèdent, on a définie les différentes approches de transformation de graphe. A cet effet, la plupart des outils discutés utilisent l'approche algébrique avec l'un de ses sous approches DPO ou SPO. Généralement ces approches ne sont pas utilisées telles qu'elles sont dans la théorie mais augmentées par des contraintes.
AGG et GROOVE mettent en œuvre l'approche SPO en ajoutant la condition d'application négative. GrGen supporte les deux approches SPO et DPO, tandis que les transformations dans AToM3 sont exprimées comme des modèles de grammaires de graphes. Et elles sont exprimées avec TGG dans FUJABA et avec l'approche SPO dans EMF Tiger.

3.3.1.4 L'intégrité de contrainte

Les contraintes d'intégrité sont utilisées pour interdire certaines structures dans le graphe initial. Elles sont vérifiées à l'exécution. Généralement les contraintes d'intégrité sont présentées par les conditions d'application négative "NAC". Ceci illustre qu'une règle de transformation ne peut être appliquée dans le cas d'une structure ne devant pas apparaître dans le graphe initial.
AGG, GROOVE, GrGen et Tiger spécifient les contraintes par l'utilisation des conditions d'application négative "NAC". Ces contraintes sont spécifiées dans AToM3 textuellement via un code Python ou OCL. Aussi, elles sont spécifiées en OCL dans FUJABA, par contre dans Viatra on n'a pas besoin d'intégrer les contraintes car il supporte les contraintes pour le graphe pattern.

3.3.1.5 Editeur

Les outils de transformation de graphes utilisent deux modes de représentation : mode graphique via une interface graphique intégrée pour spécifier et exécuter des transformations de graphes visuellement et le mode textuel via une API pour écrire des programmes qui utilisent un moteur de transformation de graphes. Ce dernier mode est plus complexe à mettre en œuvre car il faut écrire le code de transformation. Le mode graphique est supportée par GROOVE, AToM3, FUJABA, Viatra et EMF Tiger via des interfaces graphiques contrairement à l'outil textuel GrGen qui utilise son propre API appelée LibGr. Aussi, l'outil AGG utilise les deux modes graphique et textuel via une API Java.

3.3.1.6 Degré d'automatisation

La théorie de la transformation du graphes se base sur l'utilisation : grammaires de graphes et la transformation de graphe programmée. L'approche des grammaires de graphes est prise en charge par AGG, ATOM3 et GROOVE. A partir d'un graphe initial, tous les possibles productions de graphe applicables sont utilisées à plusieurs reprises, et en parallèle. Ce processus itératif est répété aussi longtemps que possible. L'ensemble des graphes résultats obtenus par ce procédé est appelé le langage engendré par la grammaire de graphes. La transformation de graphe programmée est prise en charge par Fujaba, VIATRA. Dans cette approche, les règles sont précisées pour contrôler l'ordre des transformations de graphes. Fujaba

utilise ce qu'on appelle la modélisation historique axée, où un diagramme d'activité est utilisé pour spécifier l'ordre dans lequel les transformations de graphes doivent être appliquées. En VIATRA, les transformations de graphes sont spécifiées par des machines abstraites d'états.

3.3.1.7 Processus de transformation

Un processus de transformation de modèles est dit semi-automatique ou automatique selon l'intervention de l'utilisateur dans la sélection de règle de transformation de départ ou non. Dans notre étude tous les outils sont automatique sauf FUJABA est semi-automatique.

3.3.1.8 Réutilisabilité

la réutilisation est un critère qui permet de réutiliser certain composants dans les transformation. Une façon simple mais crucial pour réutiliser les transformations de graphes est par des moyens de paramétrage. Une production graphique paramétré représente un ensemble infini de productions graphiques possibles, chacun obtenu en fournissant des valeurs concrètes pour les paramètres. Certains outils de transformation de graphes sont intégrés dans un environnement de développement orienté objet, permettant ainsi d'exploiter les mécanismes orientés objets bien connus tels que l'héritage pour permettre la réutilisation. Par exemple, dans Fujaba, les productions graphiques sont utilisés comme des spécifications de méthodes, et l'héritage peuvent être utilisés pour réutiliser ces méthodes dans les sous-classes. Dans VIATRA, une transformation de graphe peut être construit à partir de modèles prédéfinis, qui sont réutilisables dans les transformations.

3.3.1.9 Vérification

La vérification de graphe analyse si ce dernier est correct tant au niveau syntaxique qu'au niveau sémantique. L'analyse syntaxique dans la transformation de graphe est appliquée par des moyens graphiques tels que le type, la cardinalité de nœuds et d'arêtes et les contraintes de graphes. L'analyse sémantique peut être faite pour vérifier les propriétés dynamiques ou comportementales du graphe.

La vérification est exprimée par les couches de règles dans AGG, par la priorité de règles dans AToM3 et GROOVE tandis que GrGen utilise une application appelée "Graph Rewrite Sequences" (GRS) pour le contrôle des règles de transformation. Au niveau de la validation qui aide les outils de transformation à contrôler si les contraintes sont vérifiées après la transformation, AGG supporte les paires critiques et GrGen contrôle les règles de réécriture contenant les tests arbitraires.

3.3.1.10 Validation

La validation du graphe correspond à vérifier si le graphe est conforme aux exigences. Des approches telles que la simulation du graphe et le test de graphe peuvent être utilisées pour vérifier ces points.

L'ensemble des paires critiques représente précisément tous les conflits potentiels. Il existe une paire critique si et seulement si, la règle p1 peut désactiver la règle p2 ou, inversement, les conflits donc peuvent être de types suivants:

- L'application de p1 supprime un objet de graphe utilisé pour correspondre à p2.
- L'application de p1 produit une structure de graphe qui interdit p2.
- L'application de p1 change une valeur d'attribut utilisé pour correspondre à p2.

3.3.1.11 Composition de transformation

La composition de transformation de graphes peut être réalisée en utilisant la transformation de graphe contrôlée ou programmée, à savoir un ensemble de mécanismes de contrôle pour régir l'ordre d'exécution des règles. Les mécanismes de contrôle typiques sont le séquençage, la jointure et le bouclage. Ils sont soutenus dans Fujaba au moyen de ce qu'on appelle les diagrammes historiques, et dans VIATRA au moyen de machines abstraites d'états qui animent les transformations. En outre, dans Fujaba, les transformations sont appliquées en tant que corps de la méthode, de manière que la composition des transformations peut être réalisée en effectuant des appels de méthodes. En VIATRA, les transformations de graphes peuvent être composées de motifs plus les primitifs, mais les modèles

récursifs ne sont pas encore pris en charge. Un autre mécanisme qui a été proposé pour grouper les transformations de graphes est le mécanisme de structuration des unités de transformation de graphes. Par exemple, AGG utilise des mécanismes de structuration pour les transformations.
ATOM3 met l'accent sur la modélisation des systèmes complexes composés de différents formalismes et permet de les transformer en un seul formalisme commun basé sur la transformation de graphe.

TIGER permet d'implémenter sa propre stratégie dans un programme principal Java, en se basant sur l'API générée à partir des règles de transformation.

3.3.1.12 Complexité de transformation

Les transformations complexes nécessitent des mécanismes de contrôle plus complexes à diriger l'ordre d'exécution des règles. Les langages de transformation de graphes peuvent varier de manière significative dans différents niveaux. AGG s'appuie sur les grammaires de graphes en imposant un certain ordre sur les règles de production graphique à être appliquées. Fujaba utilise les diagrammes historiques (une sorte de diagrammes d'activités) pour contrôler l'application de transformations de graphes.

3.3.1.13 Standardisation

L'outil de transformation doit prendre en charge des formalismes conformes aux normes standard (tels que XML, MOF, UML). Par exemple, l'outil doit nécessairement accepter le formalisme XMI pour importer ou exporter les modèles source ou cible d'une transformation.

3.3.1.14 Type d'utilisation des outils

Le type d'utilisation permet de classifier les outils. Nous avons trois différentes catégories: usage général, transformation de modèles, et haute performance. Nous situons un outil de haute performance s'il intègre les décisions de conception qui améliorent les performances, peut-être au détriment de la généralité; par exemple en limitant les structures de graphes autorisés à refléter les structures de données programmables.

3.3.1.15 Typage

Tous les outils prennent en charge les graphes typés ou métamodèles et les graphes typées sont obligatoires dans tous les outils sauf GROOVE, c'est à dire, les modèles doivent avoir un graphe typé. GROOVE est le seul outil dont l'utilisation du typage est optionnelle.
GROOVE peut travailler dans un mode non typé ou dans un mode typé. En mode non typé, les graphes peuvent être arbitraires : il n'y a pas de contraintes sur les combinaisons autorisées de nœud typés, des arêtes. Dans le mode typé, les graphes et les règles doivent être bien typés, qu'ils peuvent être mappés dans un graphe de type spécial. Cela est vérifié statiquement pour le graphe de démarrage et les règles.

3.3.1.16 Le contrôle de fonctionnalité des outils

Dans la plupart des outils, l'application de règles peut être contrôlée en utilisant un langage impératif. Dans VIATRA, FUJABA. AGG et ATOM3 supporte uniquement la priorité. GROOVE n'est pas aussi avancé que d'autres outils, mais GROOVE supporte les priorités pour les règles. En outre, GROOVE a des fonctionnalités avancées de quantification.

3.3.1.17 Exploration

La capacité des outils par rapport aux règles d'application (exploration) des stratégies. La plupart des outils soutiennent plus d'une stratégie, comme au hasard, manuel, basé sur la priorité de règle, ou personnalisé grâce à l'utilisation du langage de contrôle. Cependant, toutes ces stratégies explorent une trace linéaire d'applications de règles. GROOVE, cependant, peut explorer tout l'espace d'état généré par différentes séquences d'application de la règle. En fait, il prend en charge plusieurs stratégies d'exploration.

3.3.2 Tableaux de comparaison

Le tableau suivant 3.1 présente une étude comparative d'un ensemble d'outils basée sur les critères introduits précédemment.

Critère	AGG	ATOM3	GROOVE	GrGen	Fujaba	EMF Tiger	Viatra
Nombre de source et cible	One to one	One to one	One to one	One to one	Many to many	One to one	Many to many
Type de transformation	Endogène	Endogène	Endogène	Exogène	Endogène	Exogène	Endogène et exogène
Approche utilisé	SPO et NAC	Grammaire de graphe	SPO et NAC	SPO et DPO	TGG	SPO	TGG
Intégrité de contrainte	Non supporté Présenté par NAC ou java	Non supporté Présenté par OCL ou Python	Non supporté Présenté par NAC	Non supporté Présenté par NAC	Non supporté Présenté par OCL	Non supporté Présenté par NAC	Supporté Graph pattern
Editeur	Graphique et textuel	Graphique	Graphique	Textuel	graphique	graphique	Graphique pour les modèles et textuel pour les règles
Degré d'automatisation	Grammaire de graphe	Grammaire de graphe	Grammaire de graphe	Optimisation automatique de règles de transformation	histoire axée modélisation	Simulation automatique	transformations entraînées par les machines d'état abstraites
Processus de transformation	Automatique	Automatique	Automatique	Automatique	Semi-automatique	Automatique	Automatique
Réutilisabilité	transformations paramétrées				transformations paramétrées/héritage de transformations	Transformation configurable	réutilisation des motifs prédéfinis
vérification	exprimé par les couches de règles	priorité de règles	priorité de règles	(GRS) pour le contrôle des règles de transformation	Analyse automatique et une vérification cohérente	contrôles de fin de contrat et l'unicité des résultats de la transformation.	vérification à part entière
Validation	L'analyse des paires critiques, la résiliation, la préservation de contraintes graphiques	Typage correct du modèle cible	Valider avec le model checking de groove	contrôle les règles de réécriture contenant les tests arbitraires	techniques de tests unitaires sont utilisées	confluence, terminaison, analyse des paires critiques de règles	"Typage correct + Préservation des modèles de graphes" valider les transformations avec des techniques de model checking
Composition	niveaux de priorités (layers) règle	Priorité de règle	Priorité de règle	Graphe Rewrite Sequence (GRS)	appel aux méthodes de modélisation de l'histoire conduit 'story diagrams'	Niveaux de priorité Procédural	composition non récursive de modèle dans les règles 'machine d'états'
Complexité	Niveaux de grammaires	Des couches avec priorités, le séquençage par priorité. l'exécution en parallèle des matches non chevauchés			transformations de graphe contrôlées	respecter les contraintes induites par la sémantique d'agrégation	d'ordre supérieur et méta transformations
standardisation	GXL, GTXL	XMI	GXL	XMI	UML, Java, XMI, MOF	ECORE	XMI, MOF, UML
type d'utilisation	usage général	Transformation de modèle	usage général	haute performance	haute performance	Transformation de modèle	Transformation de modèle
typage	requis	requis	optionnel	requis	requis	requis	requis
Contrôle	priorité	priorité	Impératif/priorité	impératif	impératif	priorité	impératif
Exploration	Linéaire	Linéaire	Multiple	Linéaire	Linéaire	linéaire	linéaire

Tableau 3.1: Tableau de comparaison

3.4 Conclusion

Dans ce chapitre, nous avons présenté une étude comparative sur un échantillon d'outils de transformation de graphes pour aider à identifier et à évaluer les performances de chaque outil.

A travers cette étude comparative d'outils et en se basant sur les différentes approches de transformations de graphes, l'approche de Triple graph grammar s'impose comme une technique la plus pertinente au niveau de la définition de relation inter-graph et la souplesse d'utilisation de règles. En plus, TGG est plus appropriée pour une transformation bidirectionnelle. Ceci nous a emmené à opter pour un outil de ce type.

Pour mieux comprendre l'approche TGG, nous détaillerons le formalisme TGG dans le chapitre suivant.

Clicours.COM

Chapitre 4

Spécification des transformations de modèles avec le formalisme TGG

4.1 Introduction

Travailler avec la représentation visuelle présente certains avantages par rapport à traiter l'information écrite. Ce qui a conduit au développement de divers langages visuels dans tous les domaines de la science moderne. Seulement en informatique il y a d'innombrables types de diagrammes, utilisés pour représenter (les flux de données, stockage de données, les interactions des utilisateurs, des séquences d'instructions, les niveaux du système d'exploitation, les restrictions du système, etc). Pourtant, en dépit de toutes les possibilités que nous gagnons en utilisant les Séquences, Entité/Relation, diagramme de classes, des fractions ou des données graphiques, etc), nous sommes limités par la syntaxe de la méthode de visualisation choisi.

L'approche des grammaires de graphes triples (Triple Graph Grammar : TGG), introduite par Andy Schürr [56]) est une tentative de créer une méthode pour connecter différents systèmes/modèles par rapport à certains règles/critères prédéfinis, de sorte que les changements dans un système/modèle conduirait inévitablement à des changements dans l'autre. TGG peut être utilisé dans différents transformation de modèles et les scénarios de synchronisation. TGG est spécifié pour les transformations bidirectionnelles (transformation à l'avant et en arrière).

TGG est un triplet de grammaires de graphes où on a une grammaire de graphe source, une grammaire de graphe cible et une grammaire de graphe de correspondance ainsi que deux morphismes le premier relie la grammaire de graphe source et de correspondance et le deuxième relie la grammaire de graphe de correspondance et cible.
Dans ce chapitre nous présentons les fondements Théoriques de TGG et le processus de transformation de modèles avec TGG.

4.2 Fondements théoriques de TGG

A fin de mieux comprendre TGG, nous explicitons à partir de [16], les notions de base pour TGG.

- Un **Graphe** est défini comme G=(V,E,s,t) ou V ensemble de sommets (nœuds) et E ensemble des Arêtes et s,t: $E \longrightarrow V$ sont des fonctions qui attribuent les sommets source et cible aux arêtes.
- Un **morphisme de graphe** de G à G' est une paire h =(h_V, h_E), ou h_V= V→V', h_E= E→E' sont définis de telle sorte qu'ils "préservent" les nœuds sources et cibles.
- **Une Grammaire de graphe** est une paire G=(G_0, **P**), ou $G_0 \in$ G est l'axiome et **P** ={p: $(L \xrightarrow{l} K \xrightarrow{r} R)$} est un ensemble de règles de transformation.
- **Une production de graphe monotone** est une paire de graphes p= (L,R), ou $L \subset R$.
- **Une production de graphe** p est applicable à un graphe G s'il existe un morphisme h: L→G.
- **Un triple de graphes** G= $(SG \xrightarrow{h_{SG}} CG \xrightarrow{h_{TG}} TG)$, ou SG graphe source, CG graphe de correspondance et TG graphe cible, h_{SG} est un morphisme de graphe entre les graphes SG ET CG et h_{TG} est morphisme de graphe entre les graphes CG et TG.

- Un triple de productions est une structure p=(sp$\xrightarrow{h_{sp}}$cp$\xrightarrow{h_{tp}}$tp), ou sp production source, cp production de correspondance, tp production cible, h_{sp} est un morphisme de graphe entre production source et de correspondance et h_{tp} est un morphisme de graphe entre productions de correspondance et cible.

- **Une triple de production** p est applicable à un triple de graphe G si les composants de production sont applicables sur les composants de graphe. L'application de p donne un triple de graphe G' de telle sorte que les composants de productions sont appliquées sur les composants de graphe.

- **Un triple de production donné**:
  p=((SL,SR)$\xrightarrow{h_{sp}}$(CL,CR)$\xrightarrow{h_{tp}}$(TL,TR))
  peut être devenir une production de source local
  p_{sl}=((SL,SR) et source-cible production p_{st}=((SR,SR)$\xrightarrow{h_{sp}}$(CL,CR)$\xrightarrow{h_{tp}}$(TL,TR))
  ou p=$p_{sl}p_{st}$

  Une séquence d'application d'un triple de production p_1......p_n est équivalent à l'application de tous les triples de production source $p_{1_{sl}}$.....$p_{n_{sl}}$, suivie par l'application de tous les triple de production source-cible $p_{1_{st}}$......$p_{n_{st}}$.

TGG est utilisé pour la spécification des outils basés sur des graphes qui peuvent être classés comme suit:

- **La synchronisation** entre le graphe source , le graphe cible et le graphe de correspondance permet la modification d'une manière synchrone en appliquant les triples de production.

- **La transformation source-cible** donne un graphe source SG et une sequence de triple de production source, en appliquant les productions source-cible, on aura un graphe de correspondance CG et un graphe cible TG.

- **Incrémentation de propagation de modification**, à partir d'un triple de G=(SG,CG,TG), on applique d'abord une séquence de triple de production source sur SG puis on propage les modifications à CG et TG en appliquant les productions source-cible.

4.3 Processus de transformation de modèles avec TGG

Le formalisme TGG a été conçu pour générer et transformer des paires de graphes connexes (généralement appelés graphes source et cible) en vertu d'un mécanisme de synchronisation bien formé (par exemple, un graphe de correspondance) qui permettrait de préserver les relations dans les graphes mis en correspondance après l'application d'une transformation. La motivation initiale du formalisme était de fournir une modélisation et une spécification d'outil graphique de haut niveau pour les problèmes impliquant des diagrammes connexes (arbres de syntaxe, diagrammes de contrôle) et les structures d'information (les exigences, les documents de conception et la traçabilité). Plus récemment, les concepts clés du formalisme TGG ont été utilisés dans le langage de transformation standard de l'OMG, pour spécifier les "correspondances" d'une transformation. En outre, les applications de TGG dans le domaine des spécifications de transformations bidirectionnelles de modèles sont de plus en plus fréquentes.

Pour bien comprendre les étapes de transformation avec le formalisme TGG, nous utilisons un exemple simple de jouet train emprunté de [59] afin d'illustrer le processus de transformation.

4.3.1 Exemple de Session

L'exemple de session présente un plan de construction de voies pour un train jouet qui est transformé en un réseau de Pétri illustrant le comportement dynamique du jouet train. Le plan se compose de plusieurs pistes et des commutateurs. Les pistes ou voies ferrées (en anglais : track) possèdent des points de liaison (en anglais : port). Une voie a un seul In-port et un seul Out-port. La figure 4.9 montre un exemple simple de la représentation graphique d'un plan de construction, que nous appelons un projet. Les différentes composants sont représentés par des cases, les In-ports d'un composant sont indiqués par de petits cercles attachés à ces boîtes, et les Out-ports sont représentés par de petits carrés. La connexion d'un In-port et un Out-port est représenté comme une flèche de l'In-Port à l'Out-port.

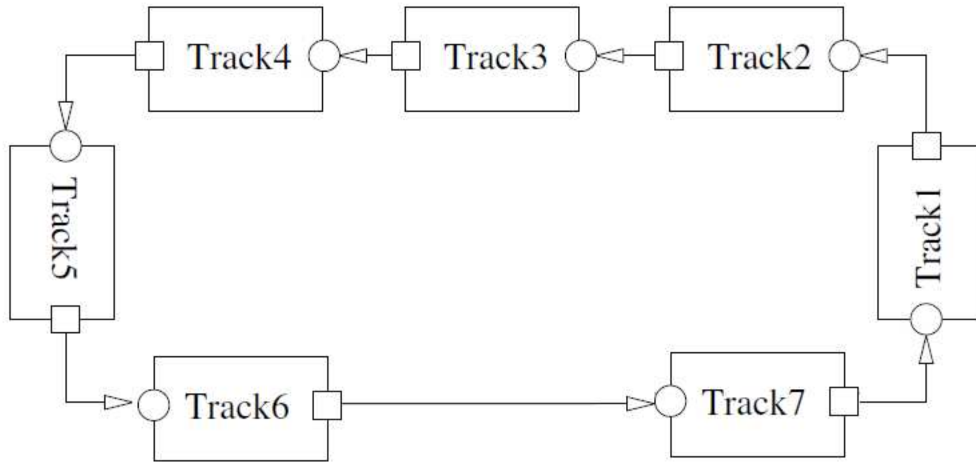

Figure 4.1: Plan de construction d'un train jouet

4.3.2 Spécification des transformations

Modelware est l'espace technique des modèles, des métamodèles et leurs transformations dans l'IDM. En revanche, Grammarware est l'espace technique des grammaires. Le pont technologique sert à trouver une communication ou une connexion entre l'espace technique des modèles et l'espace technique des grammaires, comme par exemple la traduction des éléments de Modelware en éléments de Grammarware. Il existe une simultanéité entre une grammaire et un métamodèle. La définition linguistique d'un métamodèle a une relation avec le concept d'une grammaire où un modèle conforme à un métamodèle est comme une chaîne reconnue par une grammaire. En outre, les propriétés syntaxiques et sémantiques en relation avec la conformité d'un modèle à un métamodèle, peuvent être prises en considération dans la sémantique d'une grammaire.
Notre pont technologique de la figure 4.2 sert à lier l'espace technique de l'IDM et l'espace technique des grammaires où la transformation d'un métamodèle en une grammaire se fait d'une façon automatique. Les deux métamodèles source et cible sont transformé automatiquement en grammaires de graphes qui permettent de créer d'une manière incrémentale les règles de transformation et aussi la grammaire de correspondance. A partir des correspondances sera généré un moteur transformation qui lit en entrée un modèle sorce et produit en sortie un modèle

cible.

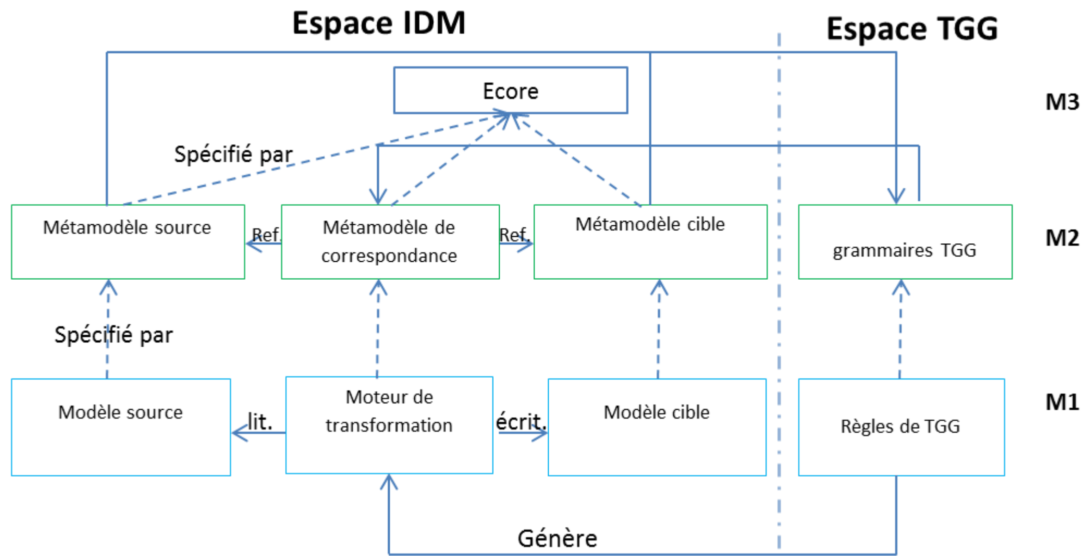

Figure 4.2: Transformation dans les espaces techniques IDM et TGG

La transformation d'un métamodèle en une grammaire de graphes nous aide à écrire les règles de transformation. La figure 4.3 présente le code source XMI d'une règle de transformation en TGG.

```
<?xml version="1.0" encoding="UTF-8"?>
<notation:Diagram xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.or
  <children xmi:type="notation:Node" xmi:id="_nE894FpfEeWR57hOsHkymA" type="2005">
    <children xmi:type="notation:DecorationNode" xmi:id="_nE9k8FpfEeWR57hOsHkymA" type="5004"/>
    <styles xmi:type="notation:DescriptionStyle" xmi:id="_nE894VpfEeWR57hOsHkymA"/>
    <styles xmi:type="notation:FontStyle" xmi:id="_nE894lpfEeWR57hOsHkymA" fontName="Ubuntu"/>
    <element xmi:type="de.upb.swt.qvt.tgg:DomainGraphPattern" href="ctoolstopnet.tgg#_nE4scFpfEeWR
    <layoutConstraint xmi:type="notation:Bounds" xmi:id="_nE8941pfEeWR57hOsHkymA" x="75" y="60"/>
  </children>
  <children xmi:type="notation:Node" xmi:id="_nQqeoFpfEeWR57hOsHkymA" type="2005">
    <children xmi:type="notation:DecorationNode" xmi:id="_nQqepFpfEeWR57hOsHkymA" type="5004"/>
    <styles xmi:type="notation:DescriptionStyle" xmi:id="_nQqeoVpfEeWR57hOsHkymA"/>
    <styles xmi:type="notation:FontStyle" xmi:id="_nQqeolpfEeWR57hOsHkymA" fontName="Ubuntu"/>
    <element xmi:type="de.upb.swt.qvt.tgg:DomainGraphPattern" href="ctoolstopnet.tgg#_nQopcFpfEeWR
    <layoutConstraint xmi:type="notation:Bounds" xmi:id="_nQqeo1pfEeWR57hOsHkymA" x="305" y="65"/>
  </children>
  <children xmi:type="notation:Node" xmi:id="_nnr3gFpfEeWR57hOsHkymA" type="2005">
    <children xmi:type="notation:DecorationNode" xmi:id="_nnr3hFpfEeWR57hOsHkymA" type="5004"/>
    <styles xmi:type="notation:DescriptionStyle" xmi:id="_nnr3gVpfEeWR57hOsHkymA"/>
    <styles xmi:type="notation:FontStyle" xmi:id="_nnr3glpfEeWR57hOsHkymA" fontName="Ubuntu"/>
    <element xmi:type="de.upb.swt.qvt.tgg:DomainGraphPattern" href="ctoolstopnet.tgg#_nnrQcFpfEeWR
    <layoutConstraint xmi:type="notation:Bounds" xmi:id="_nnr3g1pfEeWR57hOsHkymA" x="600" y="65"/>
  </children>
  <children xmi:type="notation:Node" xmi:id="_0ORkkFpfEeWR57hOsHkymA" type="2002">
    <children xmi:type="notation:DecorationNode" xmi:id="_0ORklFpfEeWR57hOsHkymA" type="5002"/>
    <styles xmi:type="notation:DescriptionStyle" xmi:id="_0ORkkVpfEeWR57hOsHkymA"/>
    <styles xmi:type="notation:FontStyle" xmi:id="_0ORkklpfEeWR57hOsHkymA" fontName="Ubuntu"/>
    <element xmi:type="de.upb.swt.qvt.tgg:Node" href="ctoolstopnet.tgg#_0N8NYFpfEeWR57hOsHkymA"/>
    <layoutConstraint xmi:type="notation:Bounds" xmi:id="_0ORkk1pfEeWR57hOsHkymA" x="65" y="260"/>
  </children>
  <children xmi:type="notation:Node" xmi:id="_1k8ckFpfEeWR57hOsHkymA" type="2002">
    <children xmi:type="notation:DecorationNode" xmi:id="_1k8clFpfEeWR57hOsHkymA" type="5002"/>
    <styles xmi:type="notation:DescriptionStyle" xmi:id="_1k8ckVpfEeWR57hOsHkymA"/>
    <styles xmi:type="notation:FontStyle" xmi:id="_1k8cklpfEeWR57hOsHkymA" fontName="Ubuntu"/>
```

Figure 4.3: Code source d'une règle de transformation en TGG

Afin de créer le métamodèle de correspondance, on doit suivre les étapes suivantes du processus de transformation:

4.3.2.1 Spécification des Métamodèles

Généralement, la pratique de la méta-modélisation consiste à définir les métamodèles qui reflètent une partie de la sémantique des modèles. Les langages de métamodélisation offrent les concepts et les relations élémentaires en termes desquels il est possible de définir les métamodèles et qui peuvent être utilisés dans la spécification des règles de transformation de modèles.
Les métamodèles de notre étude sont mis en œuvre dans la plateforme Eclipse, en utilisant en particulier les modèles EMF Ecore.

Avec les méta-modèles source et cible fournis, les mesures nécessaires pour spécifier la transformation sont constitués de:

a. définir le métamodèle de correspondance.

b. la mise en place des règles de transformation TGG.

Un modèle est conforme à un métamodèle, ou le modèle est représenté sous forme de diagramme d'objets et le métamodèle sous forme d'un diagramme de classes. comme exemple d'un modèle, on prend un modèle simple de réseau de petri.La figure 4.4 montre un réseau de Pétri dans la syntaxe concrète. Il se compose de places, de transitions et d'arcs où les places sont représentés graphiquement par des cercles, les transitions par des carrés et les arcs par des flèches.

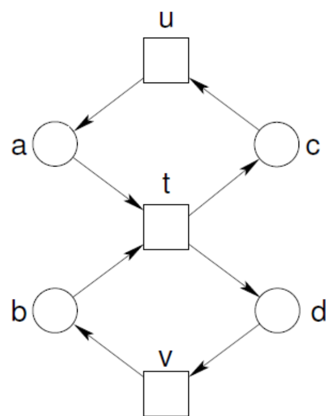

Figure 4.4: Réseau de Pétri

Cette structure des réseaux de Pétri est formalisée par son métamodèle représenté graphiquement par le diagramme de classes UML dans la figure 4.5.

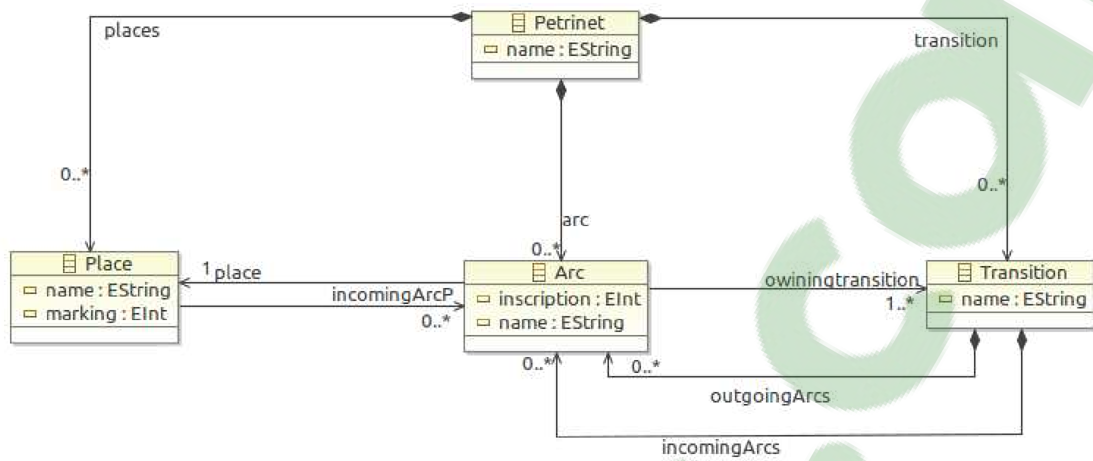

Figure 4.5: Métamodèle du réseau de Pétri

Le métamodèle pour d'un plan de construction d'un jouet train est représenté dans la figure 4.6.

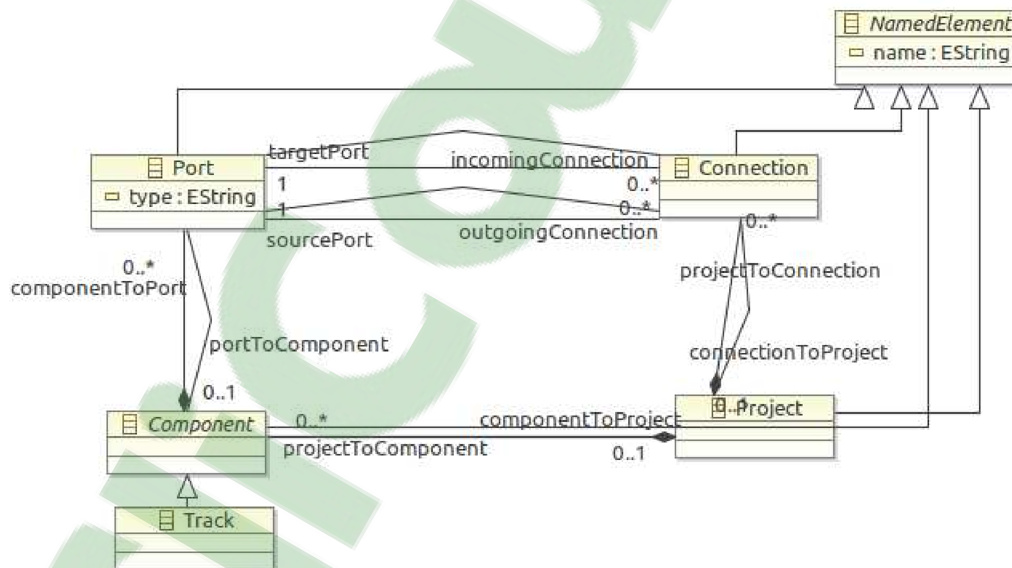

Figure 4.6: Métamodèle d'un projet jouet

Tous les concepts du métamodèle d'un réseau de Pétri sont représentés par des classes. Dans la figure 4.5, un arc relie une place et une transition où il est interdit d'avoir un arc entre deux places ou entre deux transitions; cette condition est exprimée par une contrainte OCL. le réseau de Pétri représenté dans la figure 4.4 est un modèle et une instance du méta-modèle de la figure 4.5.

En fait, La figure 4.4 illustre un réseau de Pétri dans sa représentation graphique, qui est souvent appelé *"syntaxe concrète"*. Dans le contexte d'IDM, un modèle peut être illustré par un diagramme d'objets qui est une instance de métamodèle. Le diagramme d'objets correspondant au réseau de Pétri de la figure 4.4 est représenté par la figure 4.7.

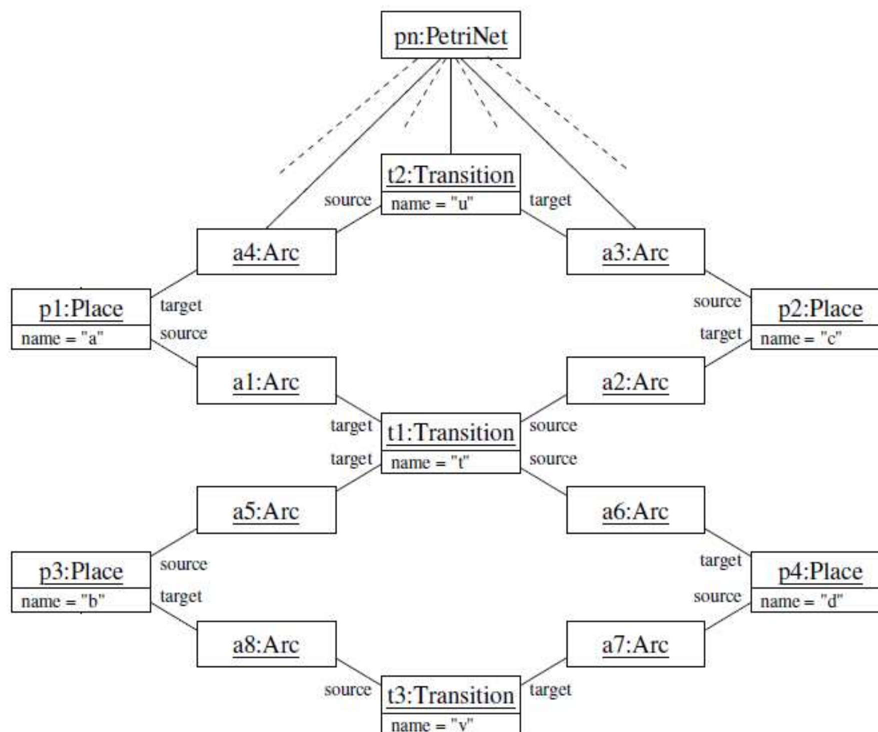

Figure 4.7: Diagramme d'objets: syntaxe abstraite d'un réseau de Pétri

Bien sûr, cela n'est pas très lisible vu que les arcs sont maintenant explicitement présentés comme des objets. Le type de chaque objet est indiquée par le nom de la classe. Le rapport à d'autres objets est indiqué par des liens. Ce type

de représentation d'un modèle de réseau de Pétri est appelé *"syntaxe abstraite"*.

Les nœuds de correspondance des règles TGG fournissent la correspondance entre les éléments des domaines source et cible. On va mettre un accent particulier sur la définition du métamodèle de correspondance, car il sert de base pour la création des règles de transformation. Le métamodèle de correspondance devrait [55] :

- Contenir des éléments de correspondance entre les modèles source et cible (ces éléments sont référencés à leurs éléments connexes dans les modèles source cible);
- Fournir une représentation, où les modèles de source sont facilement reconnaissables, et contenir des éléments de cible.

Afin de créer le métamodèle de correspondance, on doit suivre les étapes suivantes pour notre transformation :

1. Intégration des métamodèles source et cibles. Cela implique l'évaluation des relations sémantiques entre les éléments du modèle. Par exemple, ce moyen de trouver des constructions sémantiquement équivalentes des modèles concernés.
2. Ajouter des éléments représentant le modèle d'un plan de jouet de train. Ainsi, que ces éléments peuvent être facilement reconnus après une transformation et plus important, accueillir ces éléments jouent un rôle majeur dans la définition des règles de transformation.
3. Ajouter des éléments de correspondance pour avoir des éléments de réseau de Pétri comme résultat de transformation.

4.3.2.2 Spécification de règles de transformation

La transformation de modèles se base sur la définition des règles de transformation où une bonne transformation de modèles dépend d'une bonne définition des règles de transformation.
TGG se compose de trois graphes ou chaque graphe appartient à un domaine.

Chaque domaine contient une règle de grammaire de graphe. Habituellement, le domaine sur la gauche est appelé source et le domaine sur la droite est appelé cible. Le domaine de correspondance relie le domaine source et cible.
Chaque domaine est attribué à un métamodèle. Les nœuds de ce domaine sont les classes de ce métamodèle (on peut considérer le domaine comme une grammaire du métamodèle).

Avant de spécifier les règles, on doit définir les différents nœuds qu'on va rencontrer dans les règles. Il existe différents nœuds de règle de transformation: nœud de contexte , nœud de production, nœud réutilisable et les contraintes.

- **nœud de contexte** Parfois, seule une partie d'un modèle est pertinente et doit être transformée. Ensuite, les règles de TGG devraient être conçues pour les parties pertinentes de la transformation. Si les pièces d'un modèle moins pertinents influencent la transformation, ils peuvent apparaître comme des nœuds de contexte dans les règles de TGG. Le numéro 2 dans La figure 4.8 est la présentation graphique d'un nœud de contexte où il est présenté d'une case avec un contour noire.

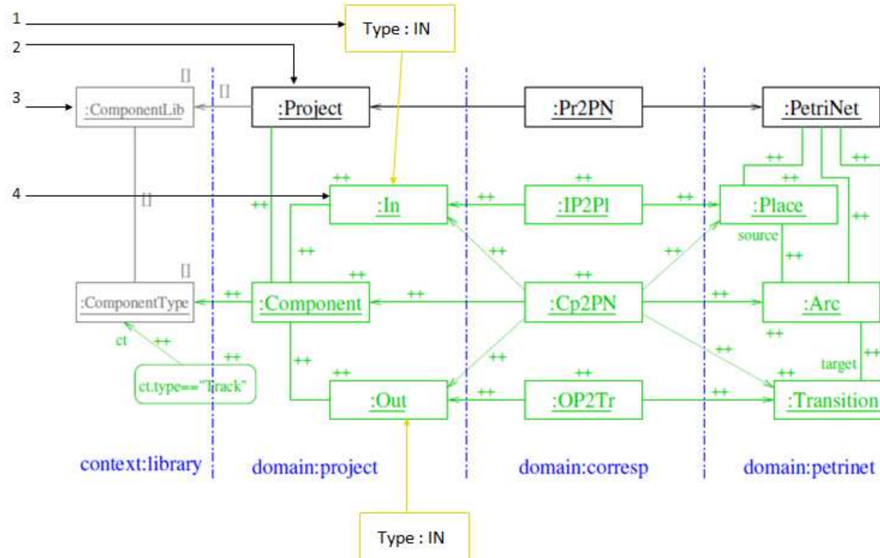

Figure 4.8: Différentes nœuds dans une règle de transformation TGG

Les nœuds de contexte doivent être appariés avec les objets du modèle précédemment traités. Cela signifie que le TGG, tel qu'il est présenté à ce jour, besoin de spécifier une grammaire complète pour toutes les parties de modèle. En revanche,TGG permet également de formuler une grammaire partielle sur un modèle. Là, les nœuds de contexte ne doivent pas nécessairement être préalablement traitées par une autre règle [25].

- **Nœud de production**

  Le nœuds de production sont affichés sous forme de boîtes vertes avec une bordure verte et une label "++". Le numéro 4 dans la figure 4.8 présente un nœud de production. Les arrêtes de production sont affichés sous forme de flèches de couleur vert foncé avec une étiquette "++".

  Les nœuds de production ne doivent pas correspondre à nœud qui existe déjà. Si on trouve ces éléments dans le domaine source, les éléments de modèle cible et de modèle de correspondance peuvent être créés selon la règle. Tous les objets créés sont liés aux nœuds de contexte de la règle. En conséquence, un objet de modèle ne peut être présenté comme un nœud de production qu'une seul fois. Nous appelons cela la sémantique bind-only-once pour les de nœuds de productions [27].

- **Nœud réutilisable**

  A noter que les types de composants ne doivent pas être générés, mais peuvent être réutilisés encore et encore depuis des nœuds déjà existants avec les propriétés requises. Par conséquent, nous appelons ces nœuds "les nœuds réutilisables". Graphiquement, ces nœuds sont représentés en gris et avec [] . Le numéro 3 dans la figure 4.8 est la présentation graphique d'un nœud réutilisable. La sémantique des nœuds réutilisables est qu'ils peuvent être nouvellement générées ou réutilisés de façon arbitraire. La couleur grise reflète le fait que, sémantiquement, chaque nœud réutilisable pourrait être soit en noir (un nœud du côté gauche de la règle de TGG, à savoir qu'il est réutilisé) ou en vert (un nœud du côté droit de la règle de TGG , à savoir qu'il est nouvellement généré), qui peut être choisi à chaque fois que

la règle est appliquée. Cette interprétation montre en fait que, les nœuds réutilisables ne sont pas strictement nécessaire. Nous pourrions remplacer par un nombre exponentiel de règles de TGG.

Mais, le concept de nœuds réutilisables permet de réduire le nombre et la complexité des règles de TGG. En plus si l'exemple est complexe, les règles sans nœuds réutilisables peuvent générer un désordre. Les nœuds réutilisables nous permettre d'avoir plus de règles simples et de concentrer sur l'essentiel des modèles pertinents [39].

- **Les contraintes** la contrainte réelle dans un nœud de contrainte peut être toute expression OCL qui se réfère à des objets qu'il est attaché. Les restrictions sont seulement nécessaires pour les mettre en œuvre des transformations ou des synchronisations plus efficace.

  Le numéro 1 dans la figure 4.8 est la présentation graphique d'une contrainte OCL représentée avec la couleur jaune. Une contrainte OCL doit être attachée avec un autre nœud.

Théoriquement, les grammaires de graphes peuvent être utilisées pour définir l'évolution dynamique d'un modèle unique. Les TGGs nous permettent de définir la relation entre différents domaines et pour bien comprendre, nous allons examiner un exemple simplifié d'un jouet de train à partir [38].

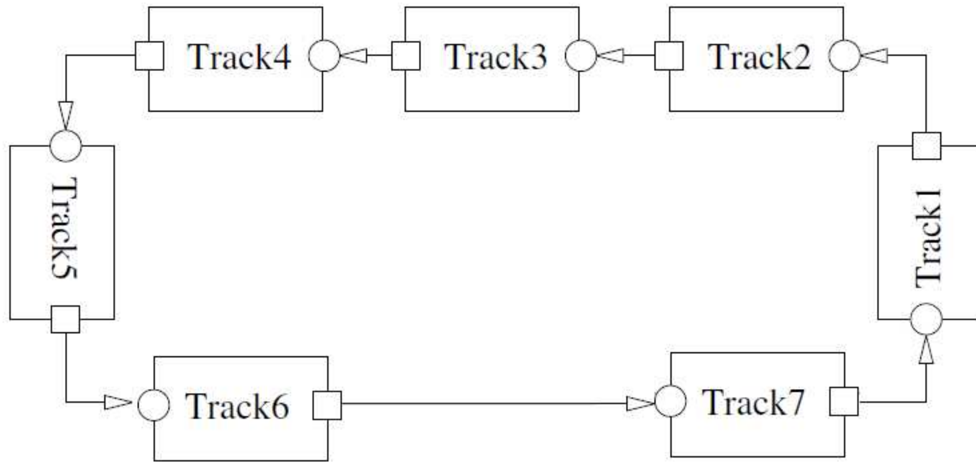

Figure 4.9: Plan de construction d'un jouet train

La première étape pour la spécification d'une règle TGG est d'avoir la syntaxe abstraite du composant. La figure 4.10 montre un composant Track et son réseau de Pétri correspondant avec le formalisme de syntaxe abstraite.

Figure 4.10: Diagramme d'objet d'un Track et son réseau de Pétri

La figure 4.11 montre une règle TGG complète de cette relation : il définit comment un composant Track correspond à un réseau de Pétri. D'un premier coup d'œil, cette règle ressemble à une règle de grammaire de graphe. La seule différence est que, maintenant, il y a trois voies qui définissent les différents domaines.

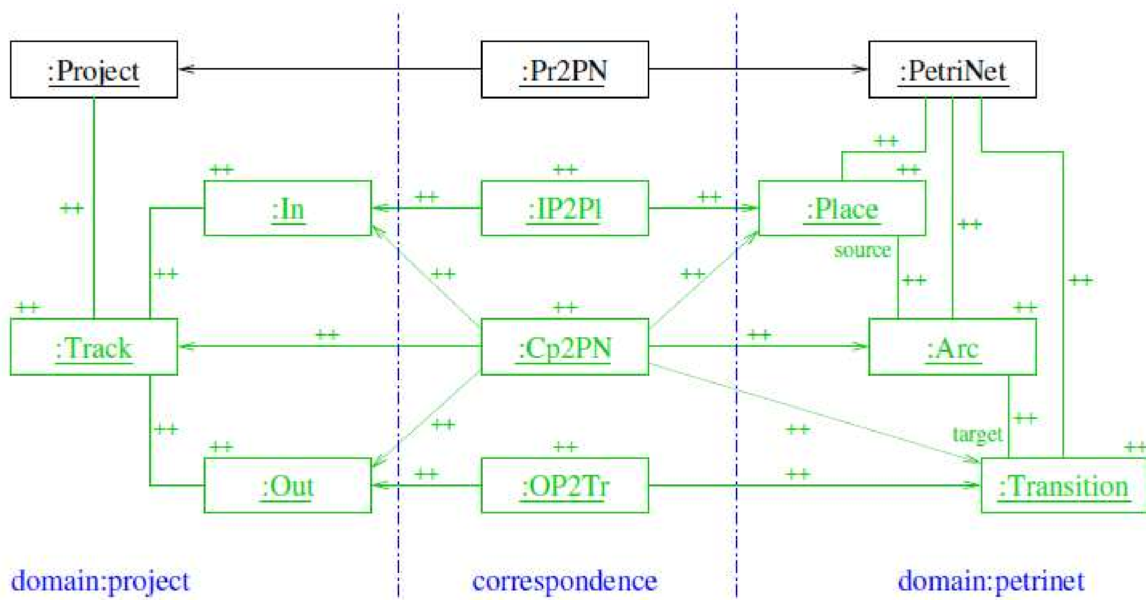

Figure 4.11: Règle TGG pour un Track

Le côté gauche de la précédente figure représente le domaine pour le projet du jouet train et le côté droit est le domaine pour le réseau de Pétri, et au milieu représente la partie qui définisse les correspondances entre les éléments de modèle source et cible. Ces objets sont appelés nœuds de correspondance. La signification de cette règle TGG est la suivante : Les parties noires, qui ne sont pas marqués par ++, représentant un projet de train jouet mis en relation avec un réseau de Pétri via un nœud de correspondance Pr2Pn. Les parties vertes, également marqués par ++, insérer un composant Track avec ses deux ports pour le projet, ainsi que la place, la transition et l'Arc qui lui correspond au réseau de Pétri. Cela définit, comment un composant Track inséré au projet correspond à l'insertion d'un fragment au réseau Pétri.

Clicours.COM

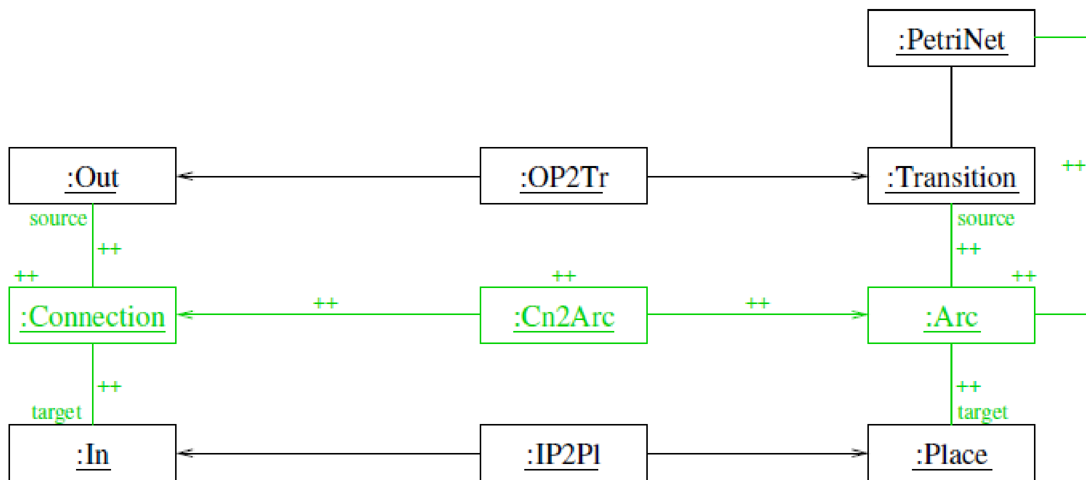

Figure 4.12: Règle TGG pour une connexion

Cette interprétation a les mêmes concepts que l'interprétation des grammaires de graphe. Ceci illustre comment le projet d'un jouet train et le réseau de Pétri peuvent être générés en parallèle. La seule différence est que les règles TGG sont considérées pour générer deux modèles en parallèle.
De cette façon, ils définissent les correspondances entre les modèles de différents types. Avant de discuter ceci en détail, nous présentons la règle TGG pour les connexions. La règle TGG pour les connexions, représentée sur la figure 4.12, illustre une connexion qui va être insérer entre un Out-port et un In-port où le Out-port correspond à une transition qui existe déjà et le In-port correspond à une place qui aussi existe déjà. Donc notre règle va produire un arc qui va correspondre à la connexion ce qui correspond à une transition et une place existants déjà, un arc correspondant à la connexion sera inséré entre cette transition et cette place.
La règle suivante présente l'axiome de la transformation.

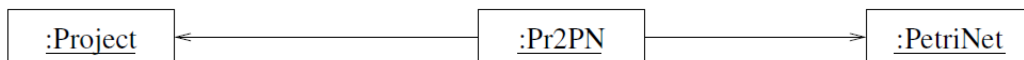

Figure 4.13: L'axiome

4.3.2.3 Transformation du modèle

L'application du scénario le plus évident de TGGs est la transformation d'un modèle à un autre modèle. Dans ce scénario, l'un des deux modèles existe déjà, et l'objectif est de générer un modèle correspondant. Par exemple, il pourrait y avoir un projet et on veut générer le réseau de Pétri correspondant.

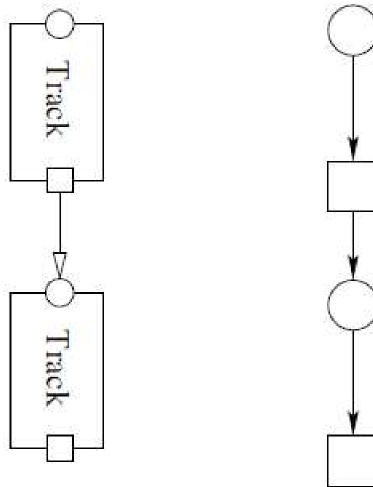

Figure 4.14: Un modèle source et son modèle cible "syntaxe concrète"

Afin de faire cette transformation, on part d'un schéma d'objet existant d'un projet (figure 4.15 "le côté gauche de la figure 4.14"). Ensuite, on essaye de faire correspondre le domaine source des règles TGG au modèle projet existant, et ajouter les nœuds de correspondance et les nœuds de réseau de Pétri généré par cette règle au modèle de correspondance et au modèle de réseau de Pétri.

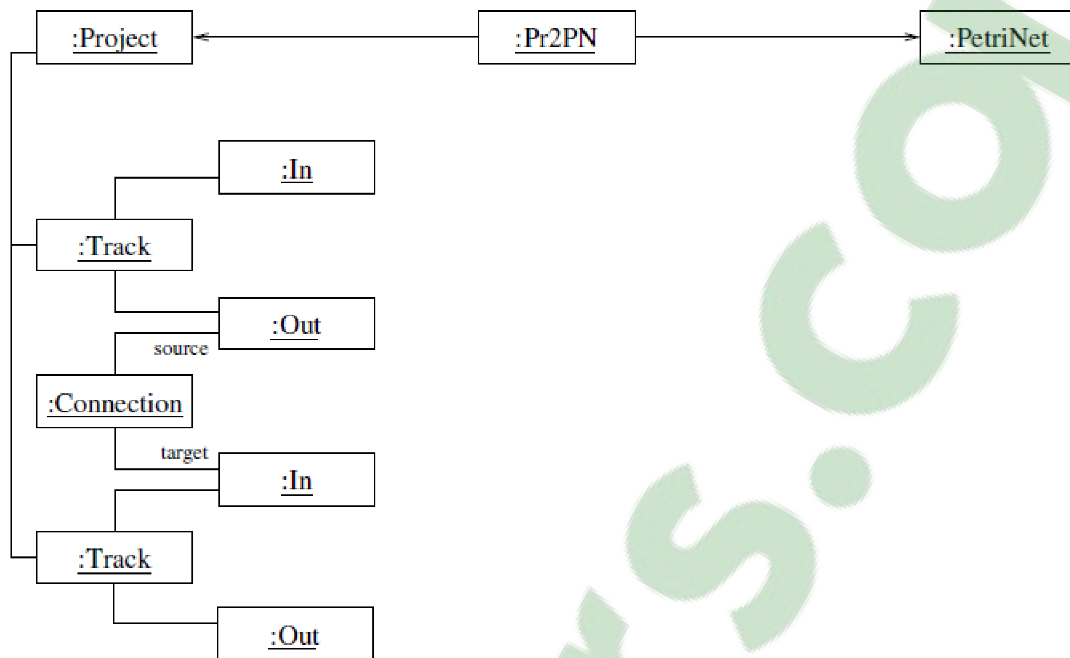

Figure 4.15: Diagramme d'objets d'un projet

La figure 4.16 montre la situation après l'application de la règle TGG pour les Track. Dans cette exemple, on exécute la règle deux fois.

la figure 4.17 montre l'application de la règle TGG pour les connexions. Une fois que nous avons entièrement adapté le modèle de projet, nous avons généré le modèle réseau de Pétri correspondant. Le résultat obtenu après la transformation et le côté droit de la figure 4.14.

clicours.com

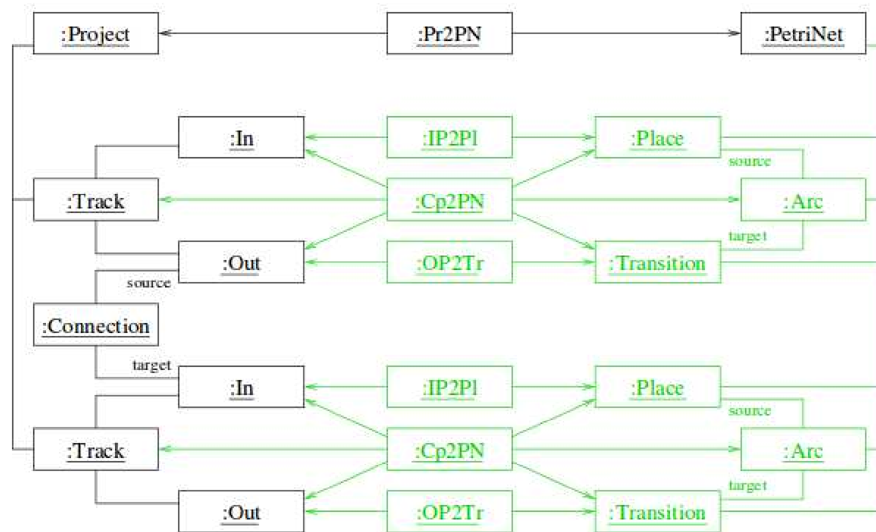

Figure 4.16: Transformation à l'avant après deux applications de la règle Track

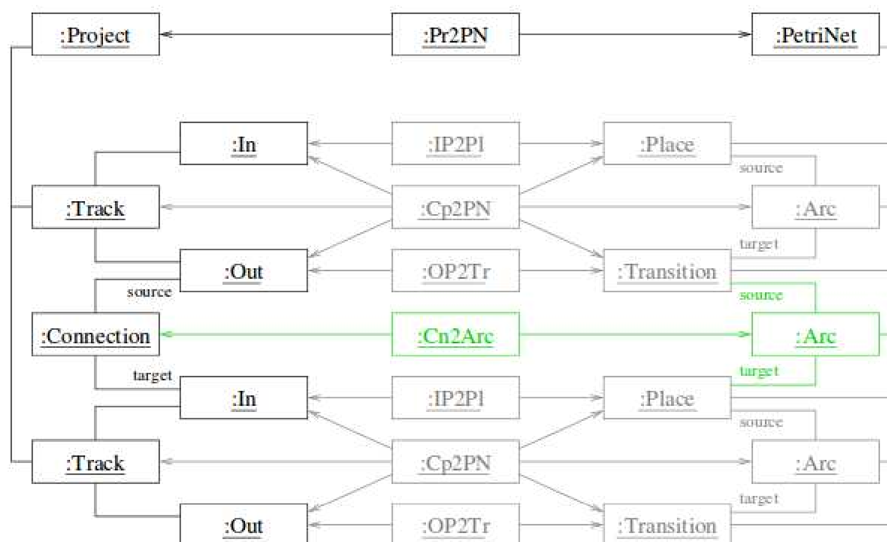

Figure 4.17: Transformation à l'avant après l'application de la règle de connexion

En principe, la transformation fonctionne également dans l'autre sens, un

Clicours.COM

réseau de Pétri pourrait être transformé en projet. Ceci est souvent appelé une transformation vers l'arrière. En fait, TGG est neutre par rapport à la direction de la transformation où il ne dépend que de ce qui est appelé le modèle source ou le modèle cible. Parfois, il est même pas claire, ce qui devrait être la source et ce qui devrait être la cible, de sorte que nous appelons simplement les domaines. Ensuite, on doit indiquer quel domaine doit être utilisé en tant que source et cible pour une transformation.

4.4 Conclusion

Nous avons présenté dans ce chapitre les différentes étapes d'un processus de transformation TGG. Puisque l'hybridation a montré son épreuve sur le plan théorique qu'on a introduit à travers d'un exemple de session, il est nécessaire de montrer ces résultats à travers des études de cas sur le plan d'implémentation. Ceci fera l'objet du chapitre suivant.

Chapitre 5

Mise en œuvre de la transformation de modèles

5.1 Introduction

Le formalisme TGG permet de spécifier des transformations de modèles bidirectionnelles. Il existe de multiples outils TGG qui diffèrent considérablement concernant l'expressivité, l'applicabilité, l'efficacité, et l'algorithme de traduction sous-jacent. Dans ce contexte, TGG Interpreter [33] est considéré comme le meilleur choix pour les transformations de graphes.

Dans ce chapitre, nous présentons l'environnement d'implémentation des transformations de modèles ainsi que deux études de cas.

5.2 Environnement d'implémentation

L'implémentation des transformations de modèles met en jeu trois étapes : la première étape permet la transformation des métamodèles de l'espace technique EMF en grammaires de graphes dans l'espace technique TGG , la seconde étape consiste à spécifier les règles de transformation et la grammaire de graphe de correspondance, la troisième étape sert à générer le moteur de transformation et exécuter ce dernier sur un modèle source pour avoir un modèle cible. La figure 5.1 illustre ces trois étapes. Dans ce qui suit, nous présenterons l'environnement d'implémentation que nous avons utilisé pour notre recherche.

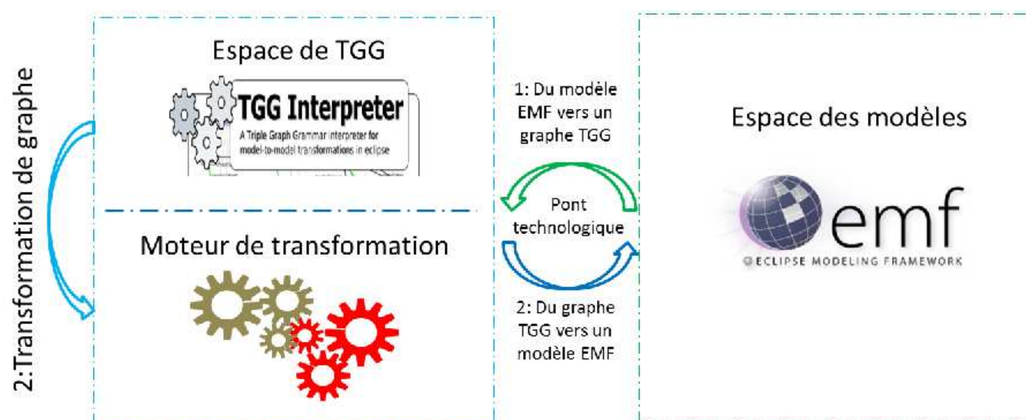

Figure 5.1: Principe de mise en œuvre de transformation de modèles

5.2.1 Eclipse

Eclipse est une plateforme universelle d'intégration d'outils de développement. Il s'agit en effet d'un IDE (Environnement de Développement Intégré) ouvert, facilement extensible et qui n'est pas lié spécifiquement à un langage de programmation. Cette plateforme fournit à la base, un ensemble de services pouvant être éventuellement enrichis par le biais de plugins.

5.2.1.1 Eclipse Modeling Framework

Eclipe Modeling Framework (EMF) est un framework Java de modélisation et un outil de génération de code pour construire des applications basées sur des modèles. Depuis un modèle de spécifications décrit en XMI, EMF fournit des outils et un support de moteur d'exécution pour produire des classes Java. De plus EMF permet de stocker les modèles sous forme de plusieurs fichiers reliées. Les modèles peuvent être spécifiés en utilisant des documents Java, UML, XSD, XML, puis sont importés dans EMF. Par contre EMF ne propose pas d'éditeur graphique pour la modélisation. Le plus important est qu'EMF fournit les fondements à l'interopérabilité avec d'autres outils ou applications basés sur EMF. EMF utilise le langage Ecore qui est un langage de métamodélisation graphique et textuelle défini par IBM.

5.2.1.2 Ecore

Le langage Ecore [16] est un langage de méta-modélisation graphique et textuelle défini par IBM et utilisé dans le framework de modélisation d'Eclipse EMF. Il est utilisé pour définir les métamodèles et le langage OCL est intégré pour d'écrire les contraintes dans le but de vérifier la conformité des modèles à leurs métamodèles. Nous choisissons d'utiliser ce langage à base des critères suivants :

- Le langage Ecore [13] est un langage graphique et textuel. La réprésentation graphique du langage permet de manipuler directement les concepts, sans passer par une formalisation textuelle abstraite et difficile à acquérir.

- Ecore intègre le langage formel OCL qui facilite l'implémentation et permet de d'écrire les contraintes. Une contrainte est une expression que l'on peut attacher à n'importe quel élément du métamodèle.

5.2.1.3 TGG interpreter

Le TGG Interpreter [33] a été développé pour la transformation de modèles TGG et est un outil de mise à jour incrémentale résultant de la comparaison de TGG et de la norme de l'OMG bidirectionnel QVT (Query/View/Transformation)[24], [26] pour les transformations de modèles.

Ergonomie: Le TGG interpreter est basé sur Eclipse et peut être installé via l'Update Manager Eclipse. La figure 5.2 représente un capture d'écran de l'éditeur de règles TGG montrant une syntaxe concrète visuelle semblable à tous les autres outils.

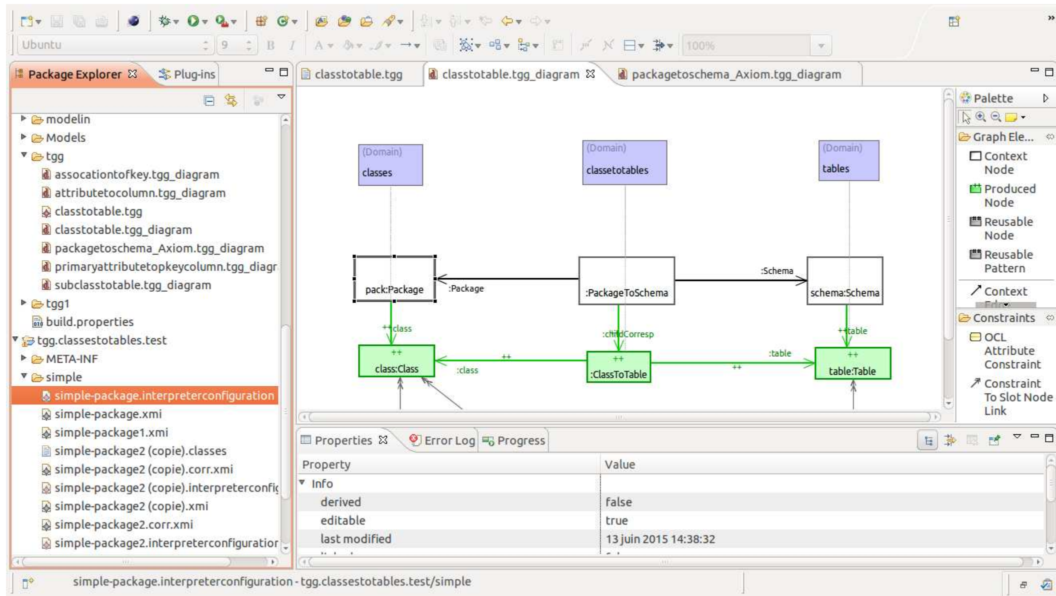

Figure 5.2: Éditeur de Règle TGG de TGG Interpreter

Plusieurs vérifications (par exemple, nœud/arc conformité de type, validité de règle d'héritage , contrôles de syntaxe OCL, etc.) sont réalisées statiquement pour éviter les erreurs de modélisation. Cependant, il n'y a pas de soutien pour l'analyse de paire critique afin d'identifier les règles qui peuvent être contradictoires. Un éditeur GMF est prévu pour l'édition de règles TGG ainsi que la fonctionnalité de commodité, par exemple, la création de nouvelles règles fondées sur des modèles dans d'autres règles, et la création des nœuds de correspondance sur le champs pour les nœuds de correspondance. TGG Interpreter exécute directement les spécifications TGG sans autre traitement. Il s'intègre dans Eclipse GUI permettant à des transformations sur scène par un clic droit sur un fichier de modèle ou via Eclipse Run configuration. Les transformations peuvent également être exécutées via un appel d'API.

Expressivité : des conditions d'application et de contraintes d'attributs peuvent être formulés en OCL, mais les relations bidirectionnelles sur les valeurs d'attributs doivent être exprimé en des affectations dans la direction vers l'avant et vers l'arrière. Pas de fortes restrictions sont imposées sur la structure de règles

TGG, à savoir, les modèles ne doivent être faiblement connectés, des arcs peuvent être créés entre les nœuds de contexte, et des nœuds de correspondance peuvent être connectés aux éléments de source et aux éléments de cible. TGG Interpreter soutient des concepts avancés tels que l'héritage de règle, les stéréotypes dans les domaines de UML, et des nœuds et des modèles réutilisables pour conserver les informations dans le cas incrémentale [32].

Propriétés formelles : TGG Interpreter supporte des fonctionnalités avancées telles que les liaisons de bords explicites. En outre, TGG Interpreter est interprété pour effectuer la transformation, il peut être mis à jour à la volée lors d'une transformation sans avoir recompiler ou calculer quoi que ce soit. Il prend en charge les mises à jour incrémentielles. En ce qui concerne l'algorithme de contrôle. TGG Interpreter nécessite un nœud de départ désigné, il ne permet qu'un seul axiome dans le TGG, et nécessite un comportement fonctionnel pour éviter le retour en arrière.Le modèle d'entrée est traversé par un parcours de l'ensemble des nœuds traduits ,en examinant tous les éléments qui peuvent être par l'intermédiaire d'un seul "edge" . Seuls ces éléments sont pris en compte dans la prochaine étape limitant la recherche de règles applicables à seulement ceux qui ont besoin de ces éléments. Le processus se termine lorsque il n'y aura plus de règles qui peut être exécuté.[42]

5.2.2 Architecture d'implémentation de transformation de modèles

La transformation de modèles tel que définie dans l'IDM nécessite un environnement de manipulation tel qu'EMF. Afin de réaliser une transformation de modèle basée sur les graphes nous avons été amenés à utiliser l'outil TGG Interpreter. Notre application est chargée de la construction des grammaires (source, cible et de correspondance) de TGG à partir des métamodèles EMF et une construction d'un modèle EMF cible à partir d'un modèle EMF source en utilisant un moteur de recherche TGG.Ce mécanisme est présenté dans la figure 5.3 sous forme de deux Espaces EMF et TGG. Dans l'espace EMF, on a les métamodèles qui sont présenté sous forme de fichiers .ecore et les modèles qui sont présentés en fichier

.xmi. Les métamodèles seront transformés en grammaires dans l'environnement TGG et ainsi des règles de production. De ces règles de production, un moteur de transformation sera déduit.

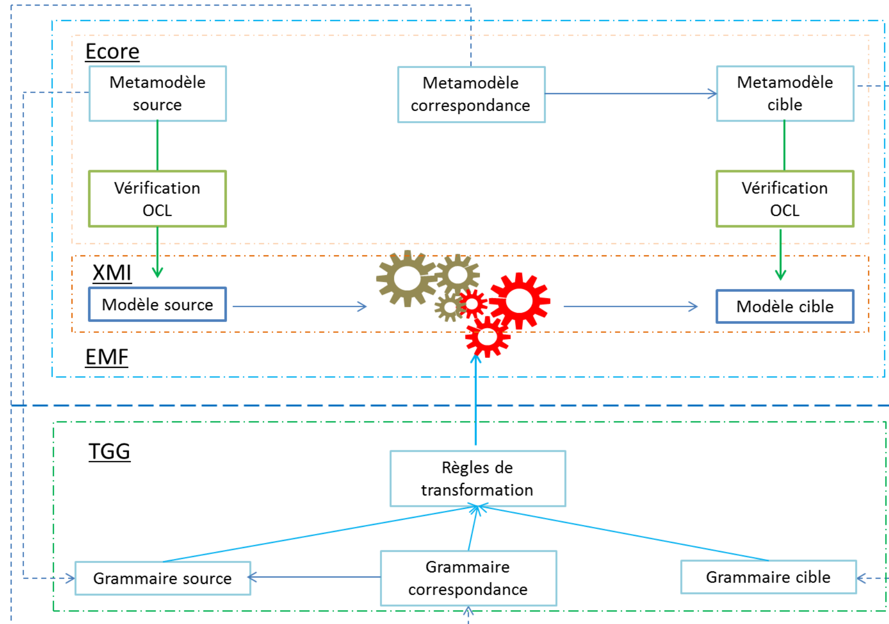

Figure 5.3: L'architecture de TGG Interpreter

5.3 Études de cas

Afin de bien comprendre notre contribution sur la transformation de modèles basée sur les grammaires de graphes, nous présentons deux études de cas qui illustrent le processus de spécification des transformations et leur mise en œuvre.

5.3.1 Transformation d'un diagramme de classes en base de données

Pour montrer l'utilité de TGG dans le contexte des transformations de modèles, nous utilisons un étude de cas bien connu de la transformation d'un modèle d'un diagramme de classe en un modèle de base de données relationnelle. Cette transformation est nécessaire si une application a besoin de stocker un ensemble d'objets

persistant dans une base de données. Cette transformation a été proposée dans différents langages de transformation comme QVT [6].

5.3.1.1 Les métamodèles

Dans ce qui suit, nous présentons les métamodèles source et cible utilisés ainsi celui des correspondances.

Métamodèle de diagramme de classes

La figure 5.4 montre le métamodèle d'un diagramme de classes. Le métamodèle indique qu'un diagramme de classes se compose de *Package* qui contient des *Class* ou des *PrimitiveDataTypes*, Chaque *Class* a ces propres *Attribute*, et peuvent être associés les uns aux autres par des *association*. En outre, les *Attribute* ont un type qui est un *PrimitiveDataTypes*.

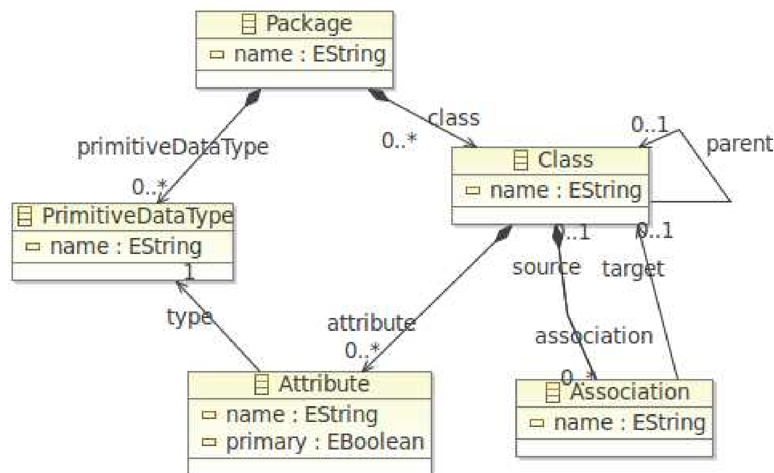

Figure 5.4: Métamodèle diagramme de classes

Métamodèle de Base de données

la Figure 5.5 présente le métamodèle donné pour un *schema* de base de données résultant de la transformation. Un *Schema* de base de données se compose d'un ensemble de *Table* qui possède des *Column* et *FKey* (foreign keys ou clé étrangère). En outre, chaque *Table* contient un ensemble désigné de *Column* qui

constituent les clés primaires de le *Table*. Enfin, chaque *FKey* fait référence à une *Table*, chaque *Table* a son propre nom et chaque *Column* a un nom et un type.

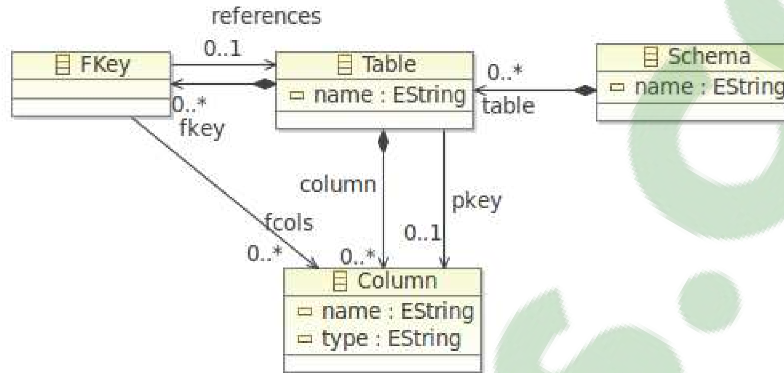

Figure 5.5: Métamodèle de base de données

Métamodèle des correspondances

La figure 5.6 illustre le métamodèle de correspondances de l'exemple étudié où *PackageToSchema* présente la correspondance entre un package et schéma. *ClasseToTable* représente la correspondance entre une classe et une table, ..etc

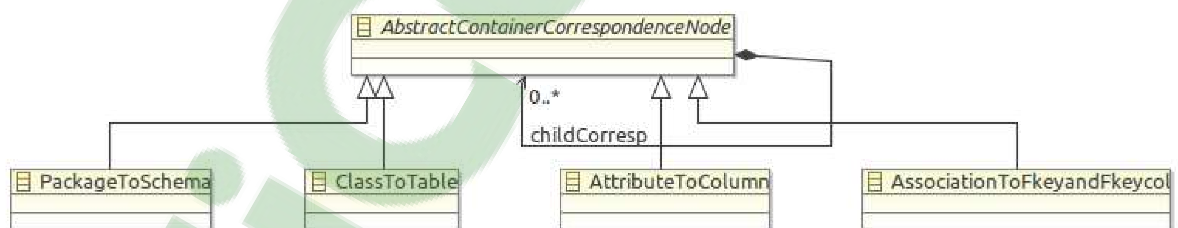

Figure 5.6: Métamodèle de correspondances

5.3.1.2 Définition de règles de TGG"

L'idée générale dans la définition des règles de transformation est de spécifier des règles TGG qui transforment chaque classe du métamodèle source en une classe du métamodèle cible.

L'axiome

La plus insignifiante relation entre un diagramme de classe et une base de données est que un Package correspond à un Schéma, ça veut dire les deux objets racine correspondent les uns aux autres. Ceci est aussi le point de départ de toute transformation et qui se nomme l'axiome dans TGG. Cet axiome est illustré dans la figure 5.7. Sur le côté gauche, il montre un objet racine d'un diagramme de classes, sur le coté droit, il montre un objet racine d'une base de données, et dans la partie médiane, il montre un nœud de correspondance relative des deux. Comme les règles TGG n'ont pas de sens et car ils sont graphique donc on est obligé de préciser les domaines ou les nœuds de chaque métamodel soient relié, pour le diagramme de classes on a le domaine *classes* et pour la base de données on a le domaine *tables* et on a le domaine *classestotables* pour les objets de correspondance.

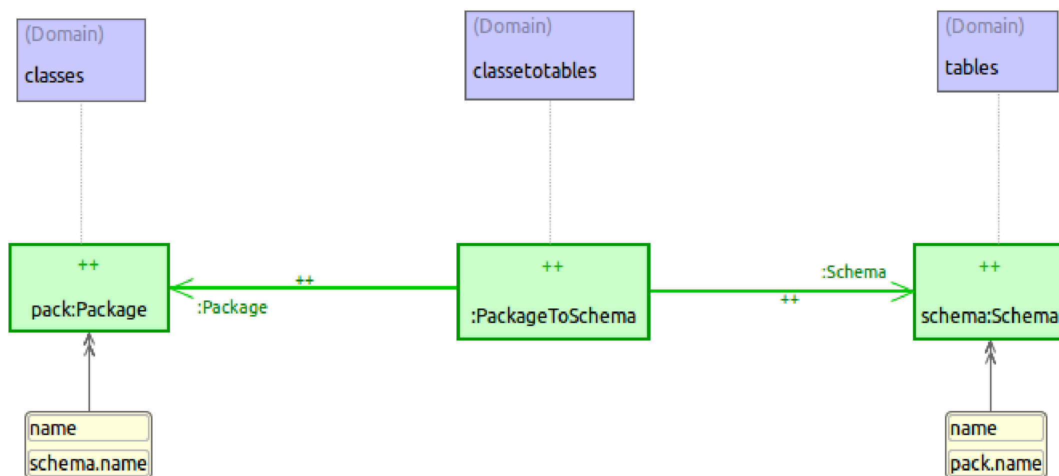

Figure 5.7: Présentation graphique de l'axiome

A partir de cet axiome, nous discutons maintenant les autres constructions qui se produisent dans les diagrammes de classes. Nous montrons comment les états correspondants sont créés par conséquence dans la base de données.

Transformation de "classe" en "table" L'idée de base de la transformation est que chaque classe d'un diagramme de classes se traduit à une table de base de données. Maintenant, on suppose que cette classe vient d'être ajouté au package de classes, qui est indiqué par la couleur verte et l'étiquette supplémentaire ++, on indique également, sur le côté droit, que l'équation correspondante doit être ajouté à la partie de base de données.

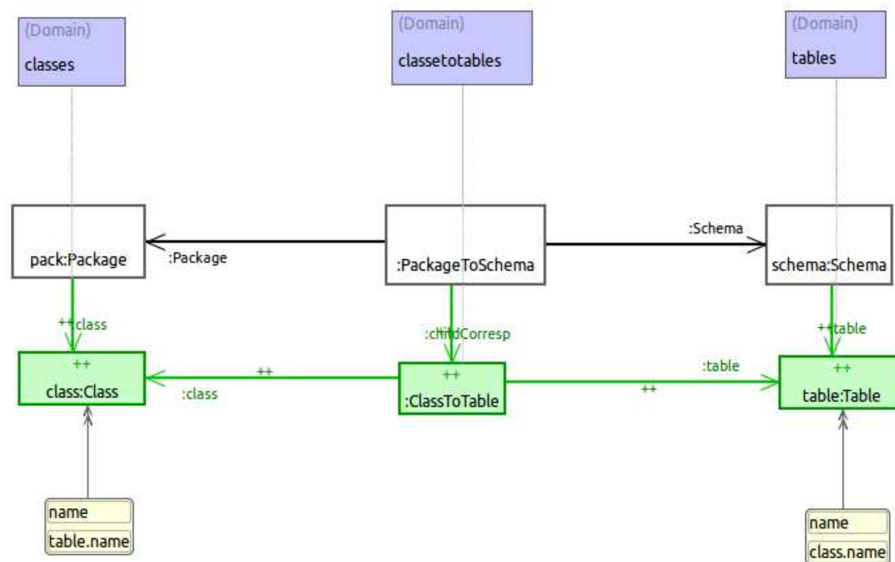

Figure 5.8: Présentation graphique de la règle ClassetoTable

Cette idée peut être exprimée par une règle TGG qui est montré dans la figure 5.8, la conversion de la syntaxe concrète et la syntaxe abstraite selon les métamodèles pour le diagramme de classes et la base de données . Dans la figure , il y a un pair de correspondance d'un Package et un Schéma, qu'il suppose qu'ils existent déjà. Par conséquent, ils sont présentés comme des nœuds de contexte noirs (et ne sont pas étiquetés avec ++). En dessous, il montre les pièces nouvellement ajoutées, indiquées en vert et marquées avec ++: Pour le domaine

diagramme de classes , cela est *Classes*, pour le domaine de base de données, il est le nouveau *Tables*. En outre, il y a un nouveau nœud de correspondance *ClassToTable* comme une manifestation de la relation entre ces concepts. Le nœud avec les bords arrondis est une contrainte supplémentaire qui garantit que les valeurs pour les noms de la classe et la table sont les mêmes.

Transformation des sous-classes en tables

la figure 5.9 présente la règle *SubClassToTable* selon laquelle si une classe à une classe parent, elle sera simplement liée à la même table déjà existante qui est lié à la classe parent.

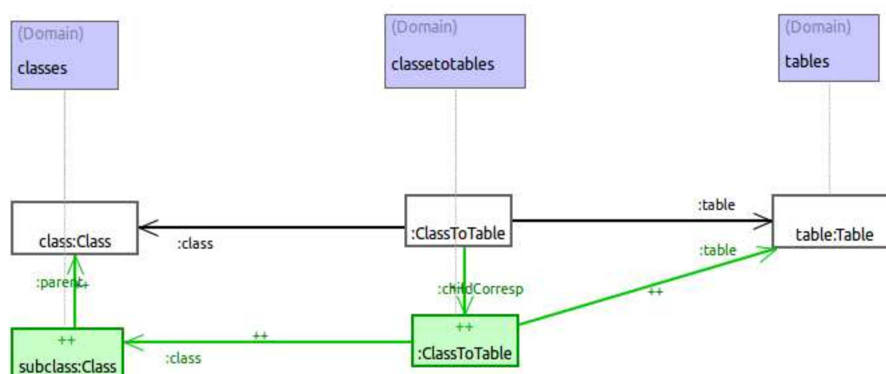

Figure 5.9: Présentation graphique de la règle SubClassetoTable

Transformation d'attribut en colonne

La figure présente la règle *AttributeToColumn* qui transforme un attribut d'une classe à une colonne d'une table avec le même nom. La colonne sera ajoutée à la table qui a été liée à la classe qui contient l'attribut, dans cette règle on va crée un nouveau nœud de correspondance *AttributeToColumn* qui va correspondre un attribut à une colonne.

Clicours.COM

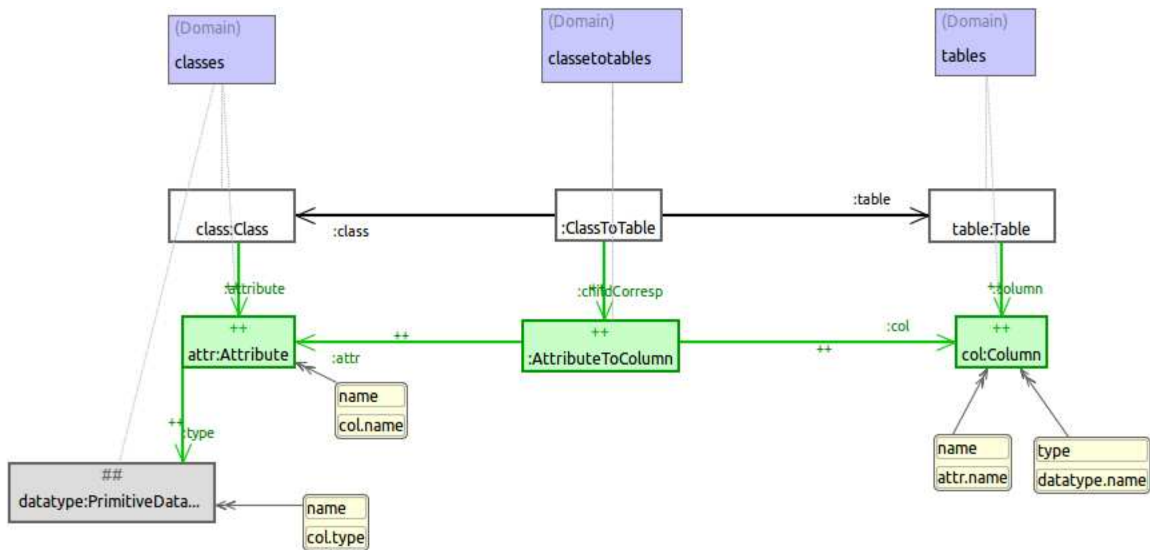

Figure 5.10: Présentation graphique de la règle AttributeToColumn

Transformation d'attribut primaire en clé primaire

La figure 5.11 présente la règle *PrimaryAttributeToPkey*, cette règle vérifié si un attribut est primaire donc il va crée un lien Pkey a la colonne qui correspond a cet attribut.

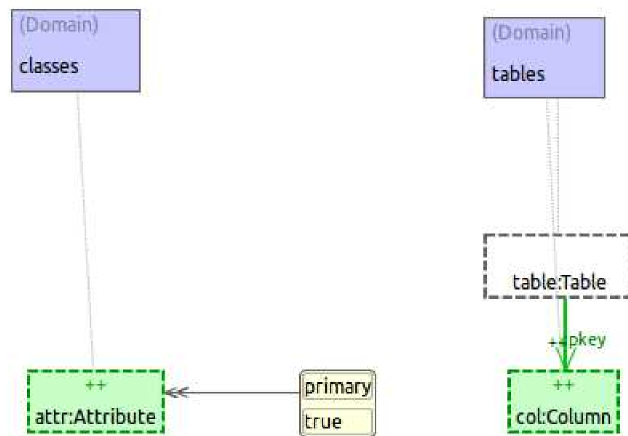

Figure 5.11: Présentation graphique de la règle PrimaryAttributetoPkey

Transformation d'association en clé étrangère

la Figure 5.12 présente la règle *AssociationToFkey*, cette règle transforme un association en une clé étrangère qui va être ajoutée à la table de la première classe, en créant aussi une nouvelle colonne dans la même table et elle va avoir une colonne de la deuxième table et qui sera une clé primaire.

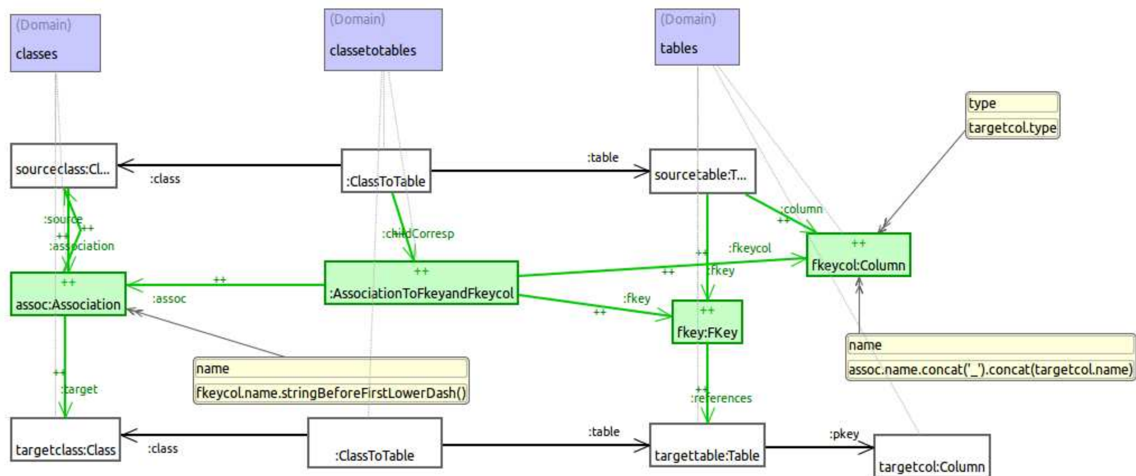

Figure 5.12: Présentation graphique de la règle AssociationToFkey

5.3.1.3 L'exécution

La phase d'exécution dans TGG Interpreter se résume à crée une configuration du programme de transformation. Une fois ce dernier est créé, il est exécuté sur le modèle source afin d'obtenir le modèle cible et une trace de l'exécution des règles est générée.

Le modèle d'entrée

on a prie un exemple simple d'un diagramme de classes ou on a une classe *company* qui a trois attributs: *name, street, city* qui sont de type String et elle a une a association avec la classe *Person*, la classe Person a trois attributs: *id number* de type Integer et qui est une clé primaire et *familyname, firstname* de

type string. la figure 5.13 présente la syntaxe concrète de cet exemple.

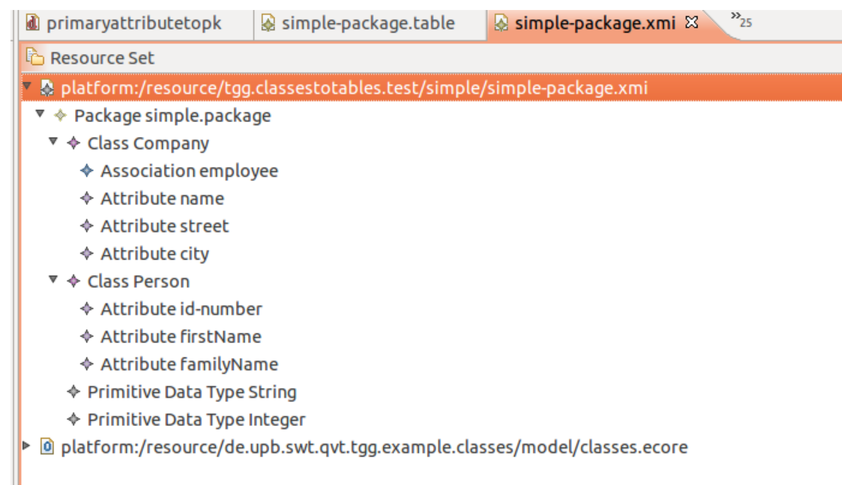

Figure 5.13: Exemple de modèle de diagramme de classes

Le modèle de sortie

Après avoir exécuter notre moteur de transformation on a eu un schéma qui contient une table *company* qui a quatre colonne, les trois premier colonnes sont *name, street, city* et la quatrième une colonne *employee id-number* qui correspond à la transformation d'une association et une clé étrangère qui réfère a la table *Person* qui a trois attributs *familyname, firstname, id-number.* la figure 5.14 présente le modèle cible de la transformation.

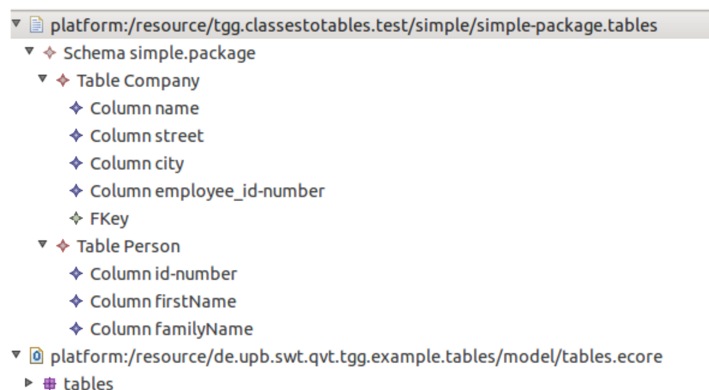

Figure 5.14: Exemple de modèle de base de données

5.3.2 Transformation d'un diagramme d'activité UML en réseau de Pétri

La motivation pour la transformation d'un diagramme d'activités en réseaux de Pétri est que les diagrammes d'activités sont basés sémantiquement sur les réseaux Pétri selon la spécification de la superstructure d'UML. Les diagrammes d'activités ont un niveau d'abstraction plus élevé. Les diagrammes d'activités partagent des propriétés communes avec les réseaux de Pétri. Différentes approches de transformation de diagrammes d'activités en réseaux de Pétri ont été suggérés. Certaines sont informelles, d'autres sont semi-formelles ou complètement formelles. Certaines de ces approches sont assez complexes. Nous illustrons ce cas à travers l'approche TGG proposée.

5.3.2.1 Les métamodèles

Métamodèle de diagrammes d'activités UML

La figure 5.15 présente le métamodèle d'un diagrammes d'activités UML. Il indique qu'un ActivityDiagram se compose d'ActivityEdges et d'ActivityNodes. On distingue en outre sept types différents d'ActivityNodes: InitialNode, Action, ForkNode, etc. Ces nœuds sont représentés comme les classes dérivées de la classe ActivityNode. EMF a besoin d'un objet que l'on appelle la racine à partir de laquelle tous les objets de certains schéma sont accessibles par des compositions dans notre métamodèle où la racine est ActivityDiagram.

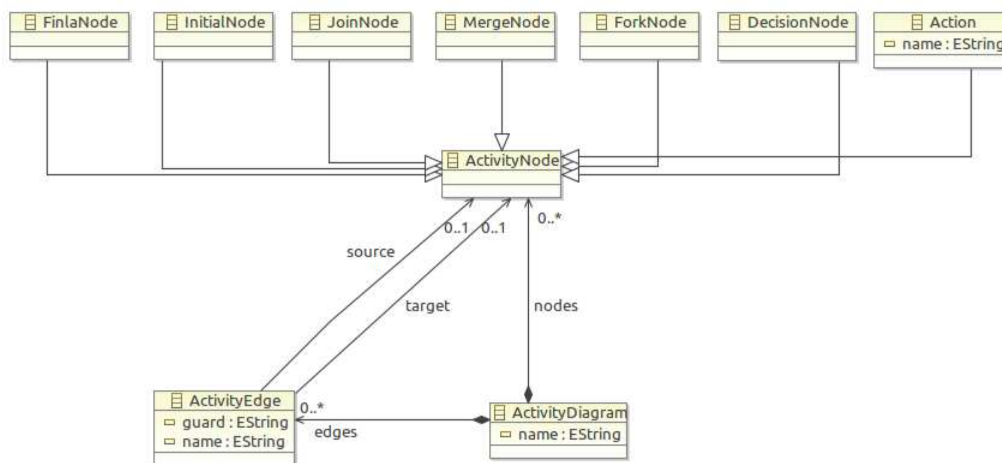

Figure 5.15: Exemple de modèle de diagramme de classes

Métamodèle de réseau de Pétri

La figure 4.7 montre le métamodèle d'un réseau de Pétri. Il montre qu'un réseau de Pétri se compose de de places, transitions et arcs. pour plus de détails" a revoir la section

5.3.2.2 Définir la relation "les règles TGG"

Dans cette section, nous définissons la relation réelle entre les diagrammes d'activité et les réseaux de Pétri.

L'axiome

la figure 5.16 présente l'axiome de la transformation d'un diagramme d'activités en réseau de Pétri, Sur le côté gauche, il montre un objet racine d'un diagramme d'activités, sur le coté droit, il montre un objet racine d'un réseau de Pétri, et dans la partie médiane, il montre un nœud de correspondance relative des deux.

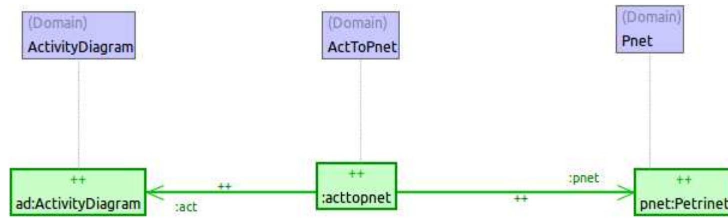

Figure 5.16: L'axiome

A partir de cet axiome, nous discutons maintenant les autres constructions qui se produisent dans le diagramme d'activités et montrons comment les états correspondants sont créés dans le réseau de Pétri.

Transformation d'un "ActivityEdge" à un "arc"

L'idée de base de cette règle de transformation est que chaque ACtivityEdge d'un diagramme d'activités se transforme à un arc dans le réseau de Pétri. Cette idée est illustrée dans la figure 5.17. Sur le côté gauche, il montre un ActivityEdge du diagramme, la droite montre l'arc qui lui correspond de le réseau de Pétri, comme indiqué par le point. Cette partie sera rempli par d'autres règles plus tard. Maintenant, nous supposons que ce ActivityEdge vient d'être ajouté au diagramme d'activités, qui est indiqué par la couleur verte et l'étiquette supplémentaire ++; Nous indiquons également, sur le côté droit, que l'arc correspondant doit être ajouté à la partie de réseau de Pétri.

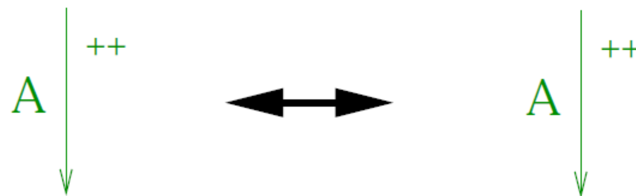

Figure 5.17: Transformation d'un ActivityEdge

Cette idée peut être exprimée par une règle TGG qui est montré dans la figure 5.18. Ça ne dépasse pas une traduction de la règle informelle de la figure 5.17, une conversion de la syntaxe concrète à la syntaxe abstraite selon les méta-modèles

pour les diagrammes d'activité et les réseaux de Pétri . En dessus de la figure, il montre un pair de diagramme d'activité et de réseau de Pétri, qui sont considéré comme des éléments qui existe déjà. Par conséquent, ils sont présentés comme nœuds de contexte noirs. En dessous, il montre les pièces nouvellement ajoutées, indiquées en vert et marquées avec ++: Pour le domaine de diagramme d'activités , on a l'ActivityEdge, pour le domaine de réseau de Pétri, on aura un nouveau arc . En outre, il y a un nouveau nœud de correspondance qui relie ces deux éléments. Le nœud avec les bords arrondis est une contrainte supplémentaire qui garantit que le nom de l'arc est le même que l'ActivityEdge.

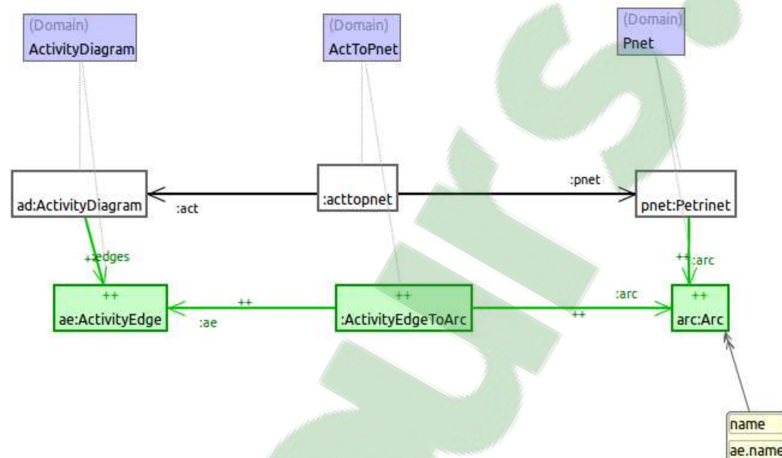

Figure 5.18: Règle TGG pour un activityEdge

Transformation d'un "InitialNode" en "place"

Maintenant , on va commencer les règles pour les nœuds la première règle est pour le nœud initial. La figure 5.22 montre l'idée de cette transformation. Nous supposons que l'ActivityEdge existe déjà (représenté en noir) et maintenant le nœud initial est ajouté en tant que nœud de source du edge (en vert et étiqueté avec ++).

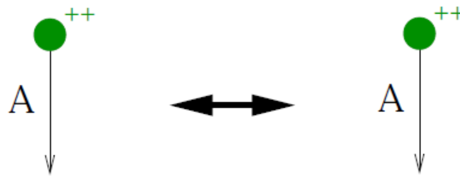

Figure 5.19: Transformation d'un nœud initial

Dans ce cas, on aura une place qui n'a pas d'arc entrants. La figure 5.20 montre cette idée en termes d'une règle TGG. L'InitialNode nouvellement inséré correspond à une place dans le réseau de Pétri.

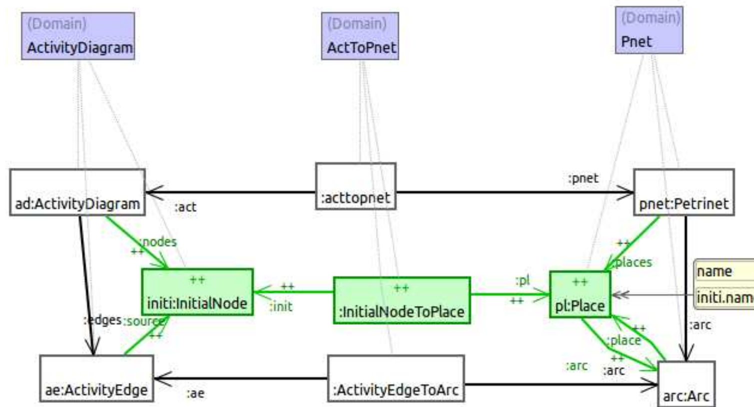

Figure 5.20: Règle TGG pour un InitialNode-a

Pour l'InitialNode, on va avoir une autre règle "présenté dans la figure 5.21", car un nœud initial peut avoir plusieurs edges sortants, donc on a besoin d'une règle qui relie tous les edges transformé en arcs avec la place nouvellement crée.

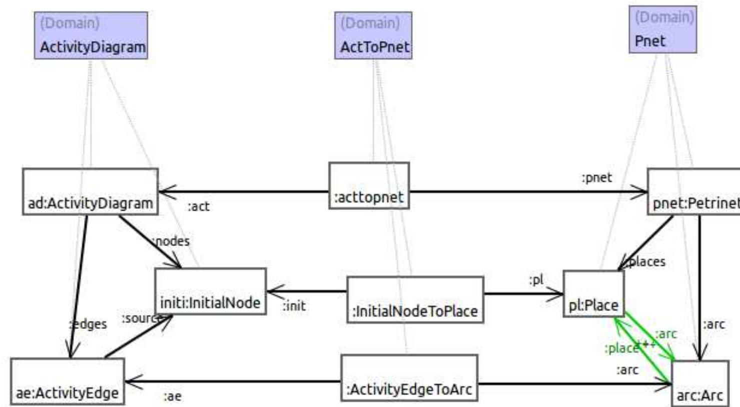

Figure 5.21: Règle TGG pour un InitialNode-b

Transformation d'une "Action" en "transition"
L'idée de transformer une action est très similaire. Il est montré dans la figure 5.22. Nous supposons que les edges A et B existent déjà et maintenant une action est ajouté entre eux (à nouveau en vert et étiqueté avec ++). Maintenant, on peut ajouté une transition entre les arcs A et B.

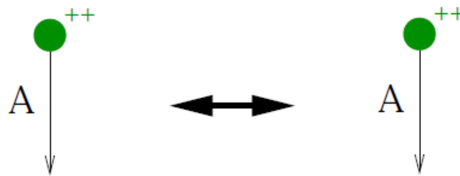

Figure 5.22: Transformation d'une Action

La règle TGG correspondant est illustré à la figure 5.23. Sur le côté gauche (dans le domaine de diagramme d'activités), nous voyons l'action ajoutée entre les deux edges. Sur le côté droit (dans le domaine de réseau de Pétri), nous voyons une nouvelle transition qui à l'arc A comme arc entrant et l'arc B comme un arc sortant. La contrainte sur la transition garantit que le nom de la transition soit le même que l'action.

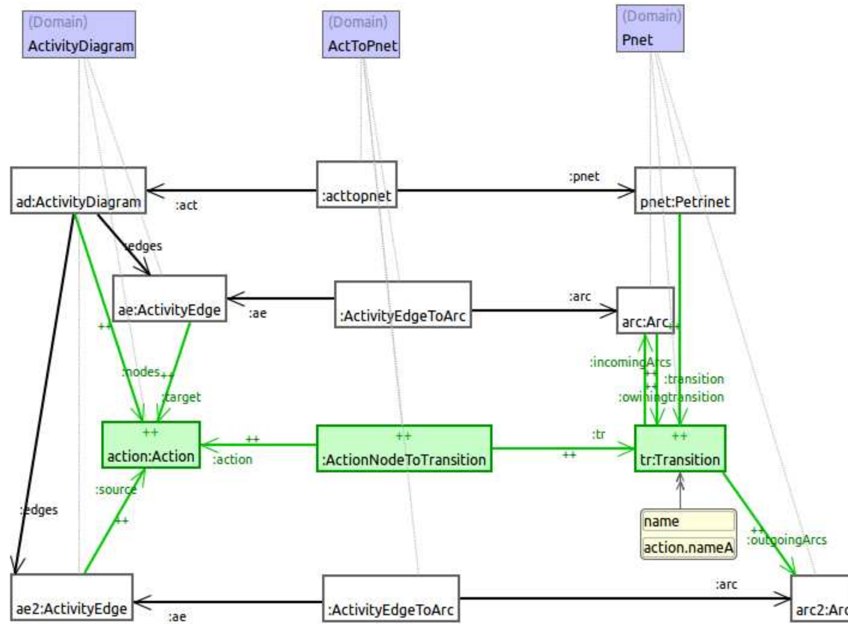

Figure 5.23: Transformation d'une Action

transformation d'un "FinalNode" en "place"

L'idée de la transformation du nœud final est montré dans la figure 5.24. le nœud final sera remplacé par une place qui n'a pas des arcs sortant.

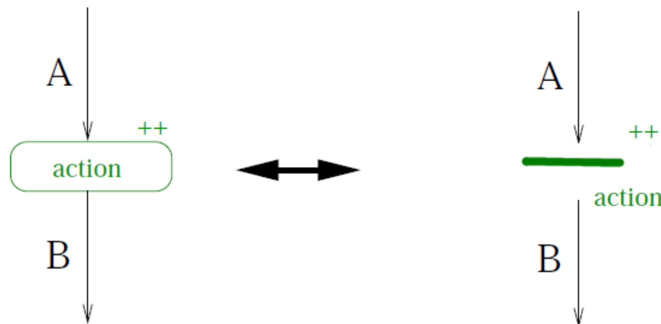

Figure 5.24: Transformation d'un FinalNode

La règle TGG correspondant est illustrée dans la figure 5.25.

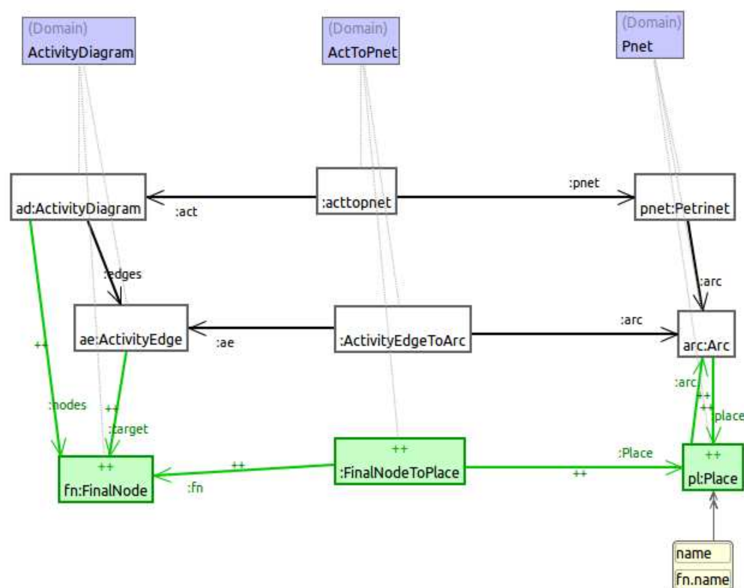

Figure 5.25: Transformation d'un FinalNode

Il peut y arriver qu'un nœud final à plusieurs edges entrants. Dans ce cas, la règle ci-dessus ne fonctionne pas. Depuis le nœud final est utilisé comme nœud de production pour la première règle. Dans le second, il est plus nouveau. Par conséquent, nous avons besoin d'une règle supplémentaire qui utilise seulement un "nouvelle connexion" d'un edge vers le nœud final et affecte l'arc à la place qui le correspond. Cette règle est illustrée dans la figure 5.26. Il est le même que ci-dessus, sauf que le nœud final est un nœud de contexte.

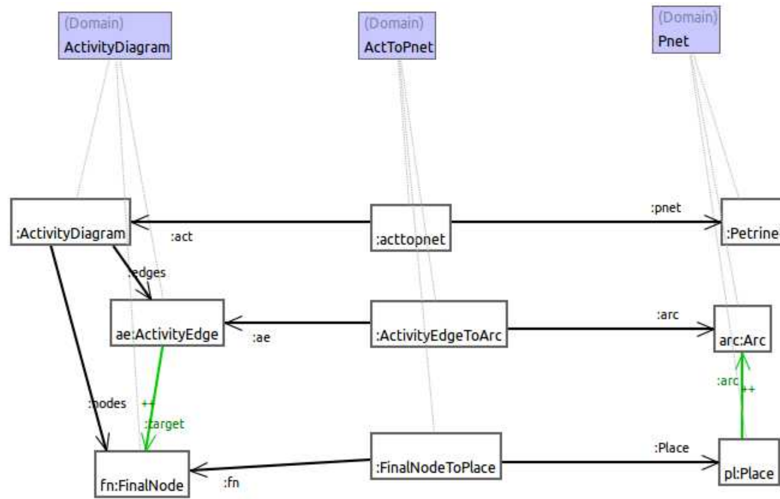

Figure 5.26: Transformation d'un FinalNode

5.3.2.3 L'exécution

Le modèle d'entrée

on a prie un exemple simple d'un diagramme d'activités avec un nœud initial et nœud final et un nœud de décision. la figure 5.27 présente la syntaxe concrète de cet exemple.

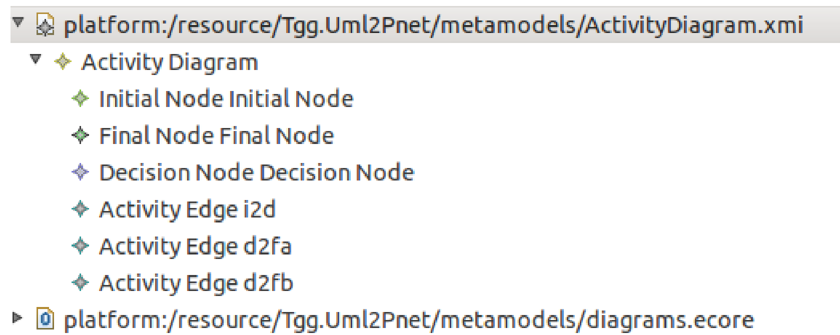

Figure 5.27: Exemple de modèle de diagramme d'activités

Le modèle de sortie

Après avoir exécuter notre moteur de transformation on a eu un réseau de Pétri

avec deux places et une transition. La figure 5.28 présente le modèle cible de la transforcesmation.

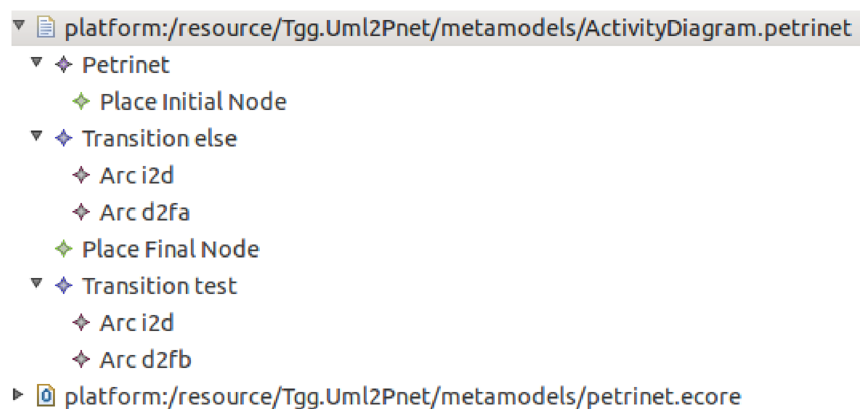

Figure 5.28: Exemple de modèle de réseau de Pétri

5.4 Conclusion

Dans ce chapitre, nous avons présenté la mise en œuvre de la transformation de modèles basée sur l'utilisation des grammaires de graphes. Notre travail s'est axé spécialement sur la spécification des transformations avec le formalisme TGG et leur implémentation avec l'outil TGG Interpreter intégré dans l'environnement EMF.

Conclusion Générale

L'ingénierie dirigée par des modèles permet le développement logiciel et elle a apporté plusieurs avantages dans la production et la maintenance des systèmes. L'IDM n'est intéressante que si la transformation de modèles qui occupe une place très importante, est partiellement ou complètement automatisée dans le processus de développement.
La notion de transformation se base sur un ensemble de techniques, langages et d'outils dont l'objectif final est de faciliter le concept de programmation et de minimiser le coût du développement.

Notre recherche de magister s'est portée sur l'étude de la théorie des grammaires de graphes et sur l'intégration de cette dernière avec la métamodélisation pour transformer des modèles selon le point de vue de l'IDM. Plus spécifiquement, nous avons mis en œuvre un pont technologique entre l'IDM et les grammaires de graphes triples.
Dans nos travaux, nous nous sommes intéressées à la transformation de modèles avec l'outil TGG interpreter qui est intégré dans la plateforme EMF/Eclipse.

Les travaux menés au cours de cette thèse ouvrent un certain nombre de perspectives à la fois pour la transformation de modèles basée sur les graphes et pour les outils de transformation de graphes.

Nous pourrons proposer quelques pistes de développement pour le futur :

- La principale mission de l'environnement EMF est la génération automatique du code java. Ce code peut être généré avec l'outil TGG interpreter si la transformation de graphe fusionné est améliorée pour la génération automa-

tique de code java.

- L'incompatibilité entre les formats de données de différents outils de transformation nécessite la standardisation de la représentation des données. L'approfondissement de cet aspect peut conduire à étudier et à proposer un pont technologique générique utilisé par différents outils de transformation de graphes et EMF.

- Finalement, l'utilisation des autres approches de transformation de graphes présentées dans ce document et autres, peuvent faire l'objet d'expérimentation dans le contexte de transformation de modèles afin de répondre à la question suivante qui ne cesse d'être posée par les chercheurs : quel type d'approche graphique ou textuelle doit-on choisir pour la transformation de modèles?

Bibliographie

[1] acceleo. Homepage. http://wiki.eclipse.org/Acceleo.

[2] AGG. Homepage. http://tfs.cs.tu-berlin.de/agg.

[3] Aditya Agrawal, Gabor Karsai, Janos Sztipanovits, Doug Schmidt, Jeremy Spinrad, Mark Ellingham, and Gautam Biswas. Model based software engineering, graph grammars and graph transformations. *Area paper, EECS at Vanderbilt University*, 2004.

[4] Charles André. Le temps dans le profil uml marte. *Université Nice Sophia Antipolis*, 2007.

[5] A. Aouat, A.Deba, and F. Bendella. Tools of model transformation by graph transformation. *IEEE ICSESS*, 2012.

[6] Biju K Appukuttan, Tony Clark, Laurence Tratt, Sreedhar Reddy, R Venkatesh, Paul Sammut, Andy Evans, and James Willans. Revised submission for mof 2.0 query/views/transformations rfp. 2003.

[7] ATOM3. Homepage. http://atom3.cs.mcgill.ca/.

[8] Daniel Balasubramanian, Anantha Narayanan, Christopher van Buskirk, and Gabor Karsai. The graph rewriting and transformation language: Great. *Electronic Communications of the EASST*, 1, 2006.

[9] Roswitha Bardohl, Mark Minas, Andy Schürr, and Gabi Taentzer. Application of graph transformation to visual languages. *Handbook of graph grammars and computing by graph transformation*, 2:105–180, 1999.

[10] Fernando Berzal, Francisco J Cortijo, Juan-Carlos Cubero, and Luis Quesada. The modelcc model-driven parser generator. *arXiv preprint arXiv:1501.02038*, 2015.

Clicours.COM

[11] Enrico Biermann, Claudia Ermel, Leen Lambers, Ulrike Prange, Olga Runge, and Gabriele Taentzer. Introduction to agg and emf tiger by modeling a conference scheduling system. *International Journal on Software Tools for Technology Transfer*, 12(3-4):245–261, 2010.

[12] Mouna Bouarioua. *Une approche basée transformation de graphes pour la génération de modèles de réseaux de Petri analysables à partir de diagrammes UML*. Thèse de doctorat, Université de CONSTANTINE 2, 2013.

[13] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[14] Benoit Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle–Application à l'ingénierie des procédés*. Thèse de doctorat, Institut National Polytechnique de Toulouse-INPT, 2008.

[15] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation-part i: Basic concepts and double pushout approach. In *Handbook of Graph Grammars*, pages 163–246. World Scientific, 1997.

[16] Francisco de la Parra. Application of graph grammars to model transformations. *Technical Report: 2013-604*, 2013.

[17] Juan De Lara and Hans Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In *Fundamental approaches to software engineering*, pages 174–188. Springer, 2002.

[18] Sana Fathallah. *Résolution des interférences pour la composition dynamique de services en informatique ambiante*. Thèse de doctorat, Université Nice Sophia Antipolis, 2013.

[19] Cyril Faucher. *Modélisation d'événements composites répétitifs, propriétés et relations temporelles*. Thèse de doctorat, Université de La Rochelle, 2012.

[20] Damien Foures. Transformation des diagrammes d'activités sysml1. 2 vers les réseaux de petri dans un cadre mde. *rapport LAAS n 11864, apport de stage, HAL Id: hal-00761053*, 2012.

[21] Fujaba. site web. http://www.fujaba.de/no_cache/publications.html.

[22] Stéphane Garredu. *Approche de méta-modélisation et transformations de modèles dans le contexte de la modélisation et simulation à évènements discrets : application au formalisme DEVS*. Thèse de doctorat, Université Pascal Paoli, July 2013.

[23] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*, 14(1):15–40, 2012.

[24] Holger Giese, Stephan Hildebrandt, and Stefan Neumann. Model synchronization at work: keeping sysml and autosar models consistent. In *Graph transformations and model-driven engineering*, pages 555–579. Springer, 2010.

[25] Joel Greenyer. A study of model transformation technologies: Reconciling tggs with qvt. *Master's thesis, University of Paderborn*, 2006.

[26] Joel Greenyer and Ekkart Kindler. Reconciling tggs with qvt. In *Model Driven Engineering Languages and Systems*, pages 16–30. Springer, 2007.

[27] Joel Greenyer and Jan Rieke. Applying advanced tgg concepts for a complex transformation of sequence diagram specifications to timed game automata. In *Applications of Graph Transformations with Industrial Relevance*, pages 222–237. Springer, 2012.

[28] GrGen. Homepage. http://www.info.uni-karlsruhe.de/software/grgen/.

[29] GROOVE. Homepage. http://groove.cs.utwente.nl/about/.

[30] Object Management Group. Request for Proposal: MOF 2.0 Query / Views / Transformations RFP. http://www.omg.org/cgi-bin/doc?ad/2002-4-10.

[31] Slimane HAMMOUDI. *Contribution à l'étude et à l'application de l'ingénierie dirigée par les modèles*. Thèse de doctorat, université de DANGERS, 2010.

[32] Stephan Hildebrandt, Leen Lambers, Holger Giese, Dominic Petrick, and Ingo Richter. Automatic conformance testing of optimized triple graph grammar implementations. In *Applications of Graph Transformations with Industrial Relevance*, pages 238–253. Springer, 2012.

[33] TGG interpreter. HomePage. http://www.cs.uni-paderborn.de/index.php?id=tgg-interpre

[34] Jamda. site web. http://jamda.sourceforge.net/.

[35] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language MOLA. In *Model Driven Architecture*, pages 62–76. Springer, 2005.

[36] Audris Kalnins, Edgars Celms, and Agris Sostaks. Model transformation approach based on mola. In *Model Transformations in Practice Workshop at MoDELS*, pages 83–90. Citeseer, 2005.

[37] Ekkart Kindler, Vladimir Rubin, and Robert Wagner. An adaptable tgg interpreter for in-memory model transformation. *Proc. of the FUJABA Days*, pages 35–38, 2004.

[38] Ekkart Kindler, Vladimir Rubin, and Robert Wagner. Component tools: Integrating petri nets with other formal methods. In *Petri Nets and Other Models of Concurrency-ICATPN 2006*, pages 37–56. Springer, 2006.

[39] Ekkart Kindler and Robert Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. *University of Paderborn*, 2007.

[40] Anneke G Kleppe, Jos Warmer, Wim Bast, and MDA Explained. The model driven architecture: practice and promise, 2003.

[41] Amine Lajmi. *Usine logicielle de composants de simulation de procédés CAPE-OPEN*. Thèse de doctorat, Paris 6, 2010.

[42] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A comparison of incremental triple graph grammar tools. *Electronic Communications of the EASST*, 67, 2014.

[43] Tihamér Levendovszky and Hassan Charaf. *Applying Metamodels In Software Model Transformation Methods*. PhD thesis, Budapest, 2005.

[44] Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1):181–224, 1993.

[45] Frank Marschall and Peter Braun. Model Transformations for the MDA with BOTL. In *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, pages 25–36, 2003.

[46] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[47] Tom Mens, Pieter Van Gorp, Dániel Varró, and Gabor Karsai. Applying a model transformation taxonomy to graph transformation technology. *Electronic Notes in Theoretical Computer Science*, 152:143–159, 2006.

[48] Mia-software. site web. http://www.mia-software.com/.

[49] Pierre-Alain Muller. *De la modélisation objet des logiciels à la metamodélisation des langages informatiques*. Thèse de doctorat, Université Rennes 1, November 2006.

[50] OMG. UML Profile for CORBA, v1.0. 2002.

[51] omg. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[52] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0. http://www.omg.org/spec/QVT/1.0/PDF/, April 2008.

[53] optimalJ. Homepage. http://www.compuware.com/products/optimalj.

[54] Alain Plantec. Terminologie documentée de l'ingénierie dirigée par le modèles. *Technical Report, Université de Bretagne Occidentale*, 2013.

[55] Jan Dominik Rieke, Wilhelm Schafer, Jürgen Gausemeier, and Gerti Kappel. *Model consistency management for systems engineering*. PhD thesis, Paderborn University, 2015.

[56] Andy Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1995.

[57] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In *Applications of Graph Transformations with Industrial Relevance*, pages 481–488. Springer, 2000.

[58] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan De Lara, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Daniel Varro, and Szilvia Varro-Gyapay. Model transformation by graph transformation: A comparative study. In *Proc. Workshop Model Transformation in Practice, Montego Bay, Jamaica*, 2005.

[59] Susanna Donatelli PS Thiagarajan. Petri nets and other models of concurrency–icatpn 2006. 2006.

[60] UMLx. site web. http://www.eclipse.org/gmt/umlx/.

[61] Viatra2. Homepage. http://www.eclipse.org/viatra2/.

[62] VMTS. site web. http://avalon.aut.bme.hu/?tihamer/research/vmts.

[63] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[64] Xpand. Homepage. http://wiki.eclipse.org/Xpand.

Appendix A

Annexe

Cette annexe apporte des détails sur les spécifications des règles de transformation pour le deuxième étude de cas présenté dans le chapitre 5.

A.1 Les règles de transformation

A.1.1 Transformation de "MergeNode" en "place"

L'idée pour le nœud de merge (fusion) est illustré dans la figure A.1, ou le nœud merge se transforme en place. Pour chaque edge entrant, nous introduisons un arc entrant a la place qui remplace le nœud merge (fusion).

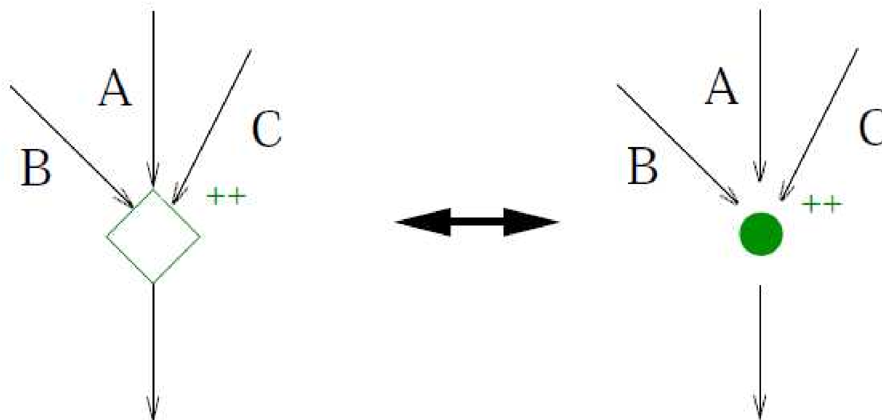

Figure A.1: Transformation du nœud merge

La figure A.2 montre la règle TGG correspondant à cette transformation. Mais, comme pour le nœud final, un nœud de fusion peut avoir plusieurs arcs entrants.

Par conséquent, nous avons besoin d'une règle supplémentaire qui traite tous les autres arcs (lorsque le nœud de fusion est déjà là). Cette règle est illustrée à la figure A.3.

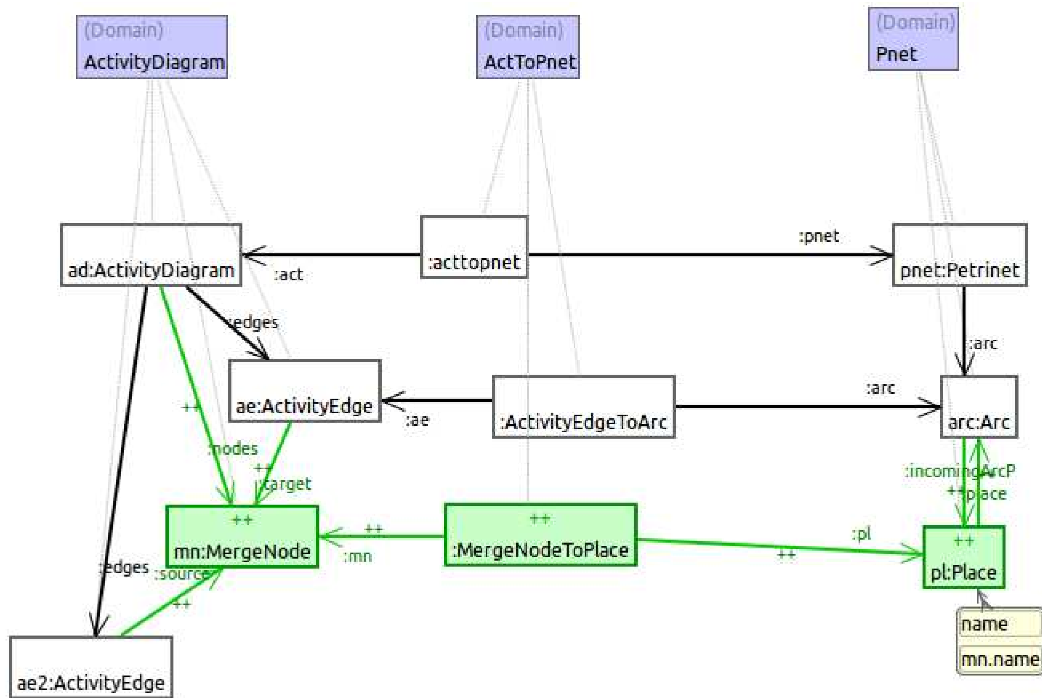

Figure A.2: La règle de transformation d'un nœud merge

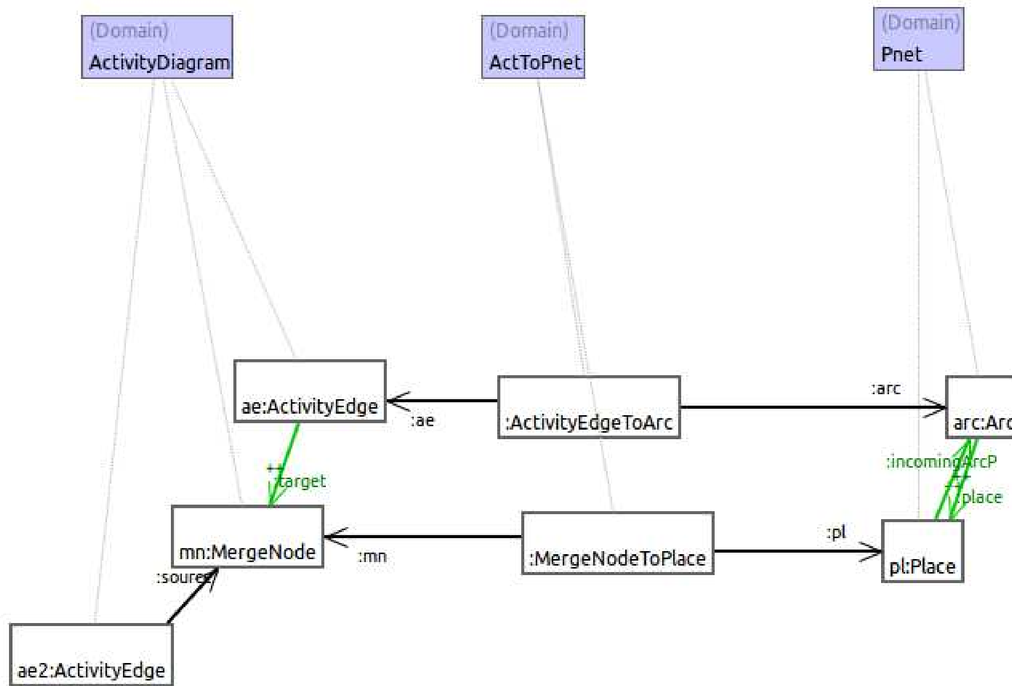

Figure A.3: la règle de transformation

A.1.2 Transformation d'un "DecisionNode" en "transition"

la Figure A.4 présente la règle de transformation d'un nœud de décision, cette règle transforme un nœud de décision en transition et la figure A.5 est la deuxième règle qui nous donne la deuxième transition.

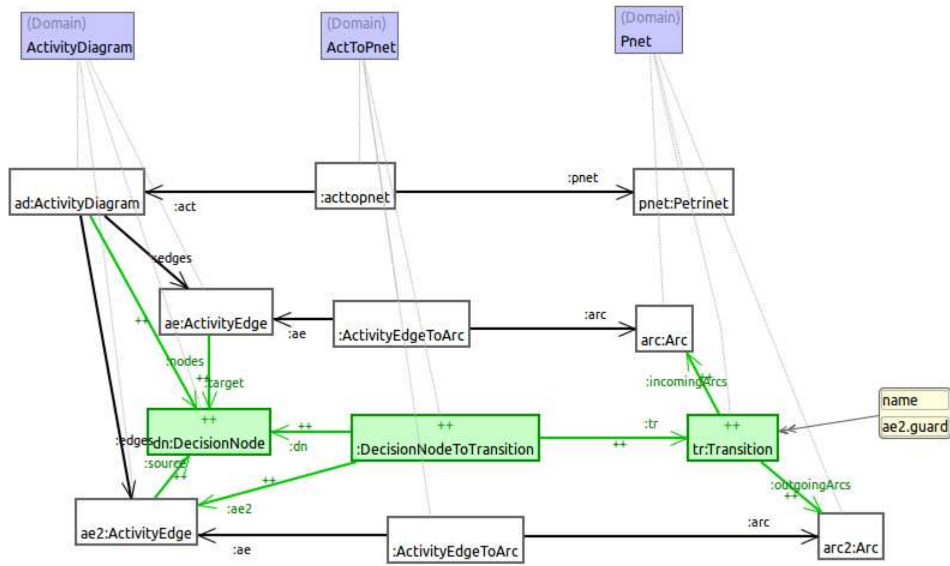

Figure A.4: présentation graphique de la règle AssociationToFkey

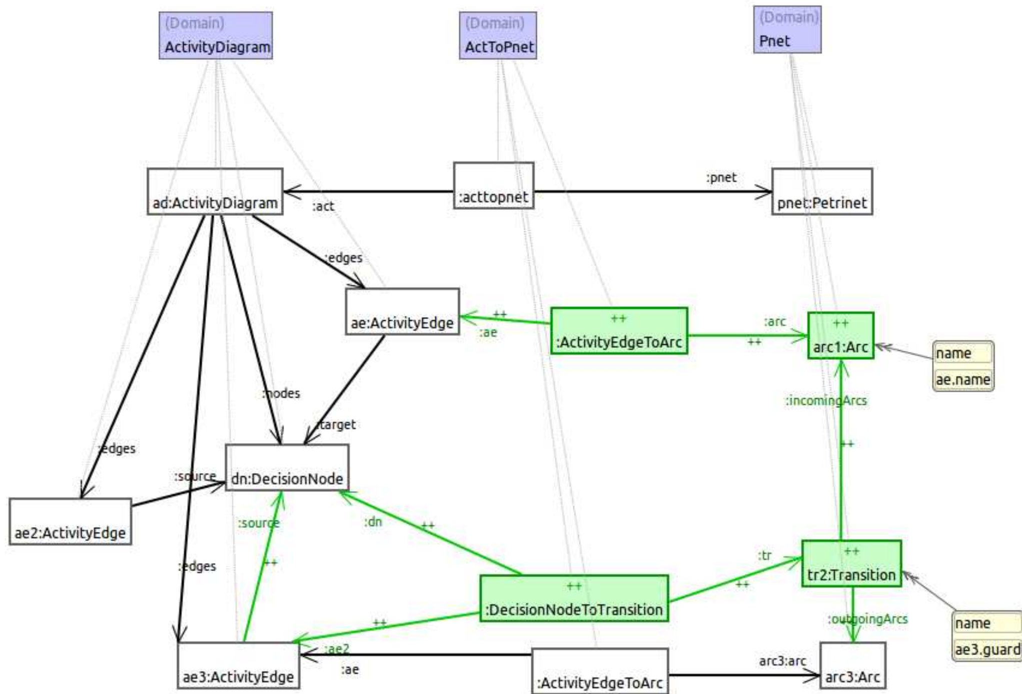

Figure A.5: présentation graphique de la règle AssociationToFkey

A.1.3 Transformation de "ForkNode" en transition

Le même principe que le nœud de fusion mais le nœud Fork sera remplacé par une transition au lieu d'un place. Sachant que le nœud fork à un arc entrant et plusieurs arcs sortants. La figure A.6 presente la transformation d'un nœud Fork en transition.

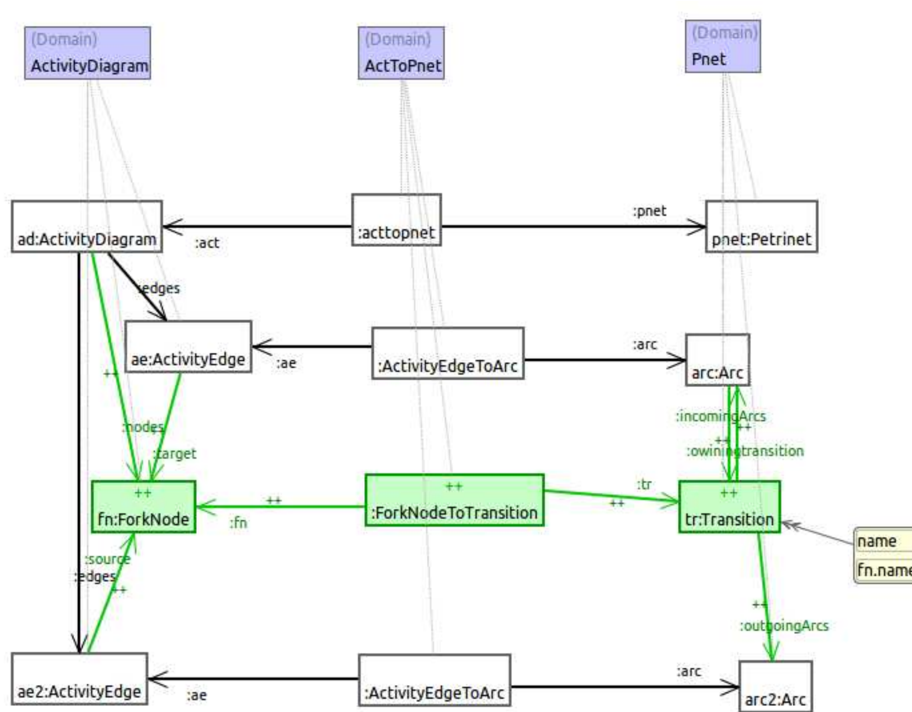

Figure A.6: présentation graphique de la règle AttributeToColumn

la figure A.7 présente le traitement du reste d'arcs sortants du nœud Fork.

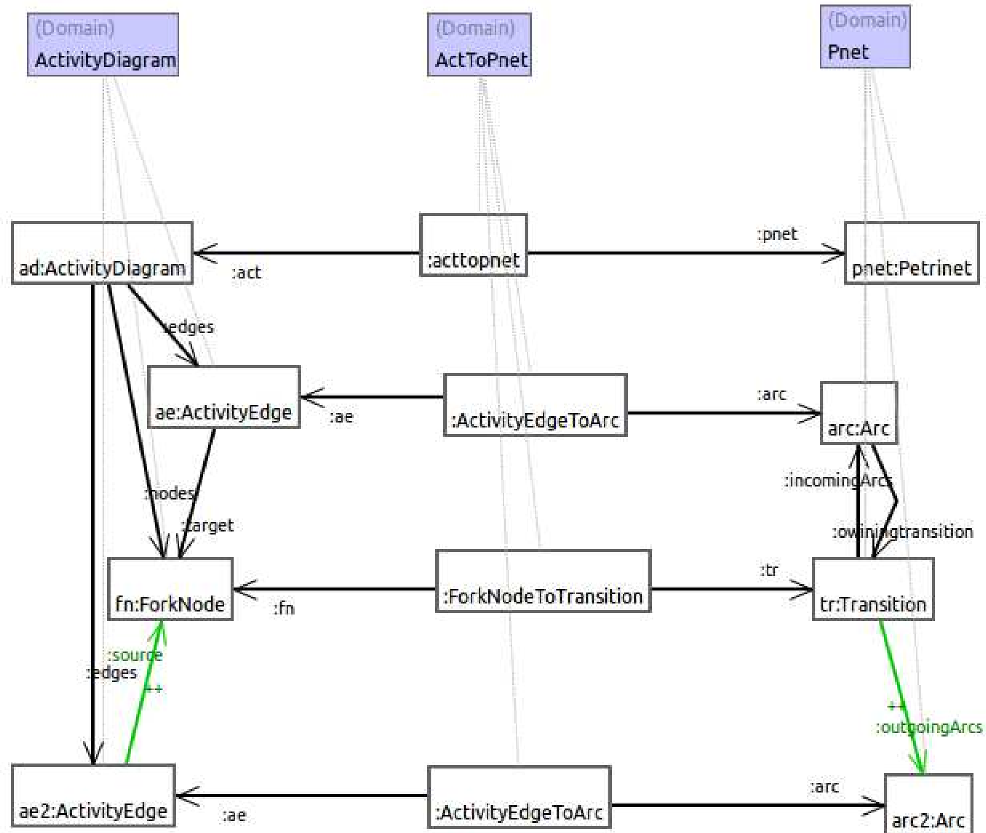

Figure A.7: présentation graphique de la règle PrimaryAttributetoPkey

A.1.4 Transformation de "JoinNode" en "transition"

Le même principe que le nœud de fusion mais le nœud Join sera remplacé par une transition au lieu d'un place. sachant que un nœud join à plusieurs arc entrants et un arc sortant.
La figure A.8 présente la transformation d'un nœud Join en transition.

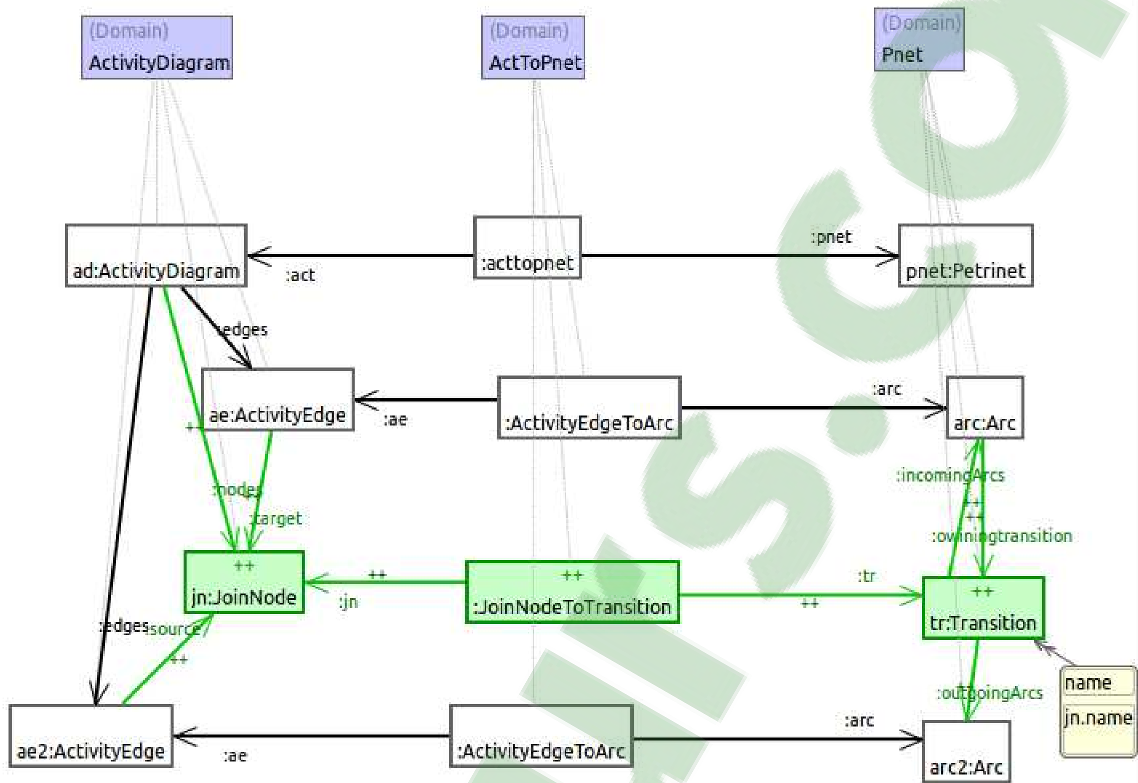

Figure A.8: présentation graphique de la règle PrimaryAttributetoPkey

La figure A.9 présente le traitement du reste d'arcs entrants du nœud Join.

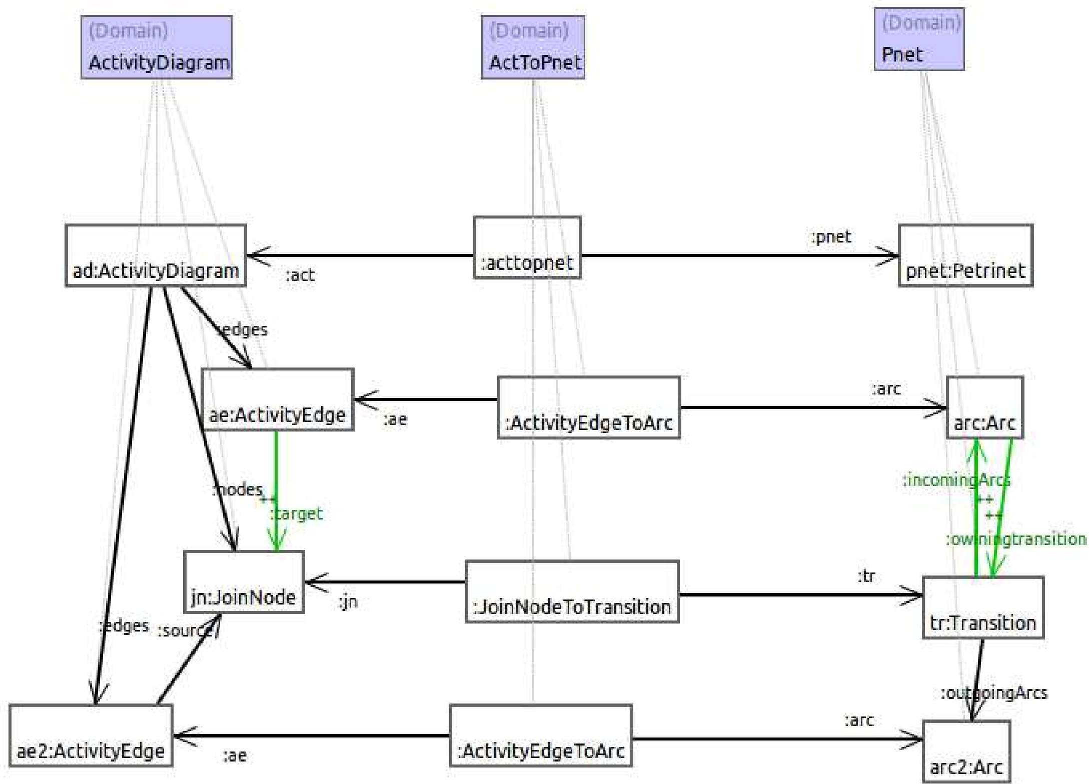

Figure A.9: présentation graphique de la règle PrimaryAttributetoPkey

Résume

L'ingénierie dirigée par des modèles joue un rôle très important dans le développement des logiciels où la transformation de modèles consiste à transformer un modèle source en un modèle cible conformément à des méta-modèles source et cible ou l'IDM vise à améliorer plusieurs problème (réutilisabilité, interopérabilité, migrer les modèles). La présentation graphique joue un rôle très important dans le développement logiciel qui nous ramène à l'intégration des graphes dans les transformations de modèles et la transformation de graphes. A cet effet, il est important de proposer des solutions pour intégrer les graphes dans le processus de transformation. L'idée de base de notre travail est de créer un pont technologique entre l'environnement de l'IDM et l'environnement de grammaires de Graphes et de faire une transformation bidirectionnelle en utilisant l'outil TGG interpréter.

Mots Clés :

Ingénierie Dirigé par les Modèles; Modèle; Métamodèle; Métamodèle de correspondances; Transformation de modèles; Moteur de transformation; Grammaire de graphe; Transformation de graphe; Formalisme TGG; TGG interpréter.

Abstract

Model Driven Engineering "MDE" plays a very important role in software development where model transformation is to transform a source model into a target model according to source and target meta-models. The MDE aims to improve several problem (reusability, interoperability, migration patterns). Graphical presentation plays a very important role in software development that brings us to the integration of graphs in model transformations and transformation graph. To this end, it is important to offer solutions to integrate graph in the transformation process. The basic idea of our work is to create a technological bridge between MDE environment and the Graph grammar environment and make a bidirectional transformation using the TGG interpreter tool.

KeyWords:

Model Driven Engineering; Model; Metamodel; Correspondence Metamodel; Model Transformation; Transformation Motor; Graph Grammar; Graph tansformation; TGG formalism; TGG interpreter.