Table des matières

1 Introduction générale 5
1.1 Langages dédiés . . . 5
1.2 Besoin de facilités d'ingénierie pour les langages de modélisation dédiés . . . 6
1.3 Contributions . . . 6
1.4 Plan . . . 7

I Ingénierie des langages dédiés 9

2 Ingénierie des langages logiciels 11
2.1 Langages logiciels . . . 11
2.1.1 Qu'est-ce-qu'un langage logiciel . . . 11
2.1.2 "Phrases", "énoncés", ou *mograms* . . . 12
2.1.3 Langages généralistes et langages dédiés . . . 12
2.2 Spécification d'un langage logiciel . . . 14
2.2.1 Syntaxes d'un langage logiciel . . . 15
2.2.2 Sémantique d'un langage logiciel . . . 16
2.2.3 Validité d'un *mogram* par rapport à la spécification d'un langage . . . 17
2.2.4 Métalangages et méta-outils . . . 18
2.3 Multiplication des langages logiciels et ingénierie des langages logiciels . . . 23
2.3.1 Multiplication des langages logiciels . . . 23
2.3.2 D'un "artisanat" à une ingénierie des langages logiciels . . . 24
2.4 Conclusion . . . 25

3 Définition de langages de modélisation dédiés 27
3.1 Spécification de langages de modélisation dédiés . . . 27
3.1.1 Métamodèle . . . 27
3.1.2 Syntaxe abstraite des langages de modélisation dédiés . . . 29
3.1.3 Outillage des langages de modélisation dédiés : transformations de modèles . . . 31
3.2 Relation de conformité . . . 33
3.2.1 La relation de conformité dans la littérature . . . 33
3.2.2 La relation de conformité dans les standards *de facto* . . . 35
3.2.3 Définition de la relation de conformité . . . 36
3.2.4 Limites de la relation de conformité . . . 36
3.3 Facilités d'ingénierie pour les langages de modélisation dédiés . . . 38
3.3.1 Modélisation *in-the-large* . . . 38
3.3.2 Réutilisation de transformations de modèles . . . 40
3.3.3 Limites des approches existantes . . . 42

3.4 Conclusion 43

4 Typage de modèles 45
4.1 Principes de base du typage objet 45
4.1.1 Objets 45
4.1.2 Types objets 46
4.1.3 Classes 46
4.1.4 Interface et implémentation 47
4.2 Réutilisation : polymorphisme et héritage 48
4.2.1 Polymorphisme de sous-type : relations de sous-typage 48
4.2.2 Héritage 52
4.2.3 Sous-typage ⇔ Héritage 52
4.2.4 Polymorphisme paramétrique : paramètres de types et variables de types 53
4.3 Polymorphisme de groupe de types et spécialisations de groupe de classes . . . 54
4.3.1 Limites du sous-typage objet pour les groupes de types 54
4.3.2 Types *chemin-dépendants* 56
4.3.3 Spécialisation de groupes de classes 56
4.3.4 Correspondance entre types objets 58
4.4 Typage de modèles 58
4.4.1 Types de modèles 59
4.4.2 Relation de sous-typage entre types de modèles 59
4.4.3 Extension aux hétérogénéités structurelles 62
4.5 Limites actuelles du typage de modèles 63
4.5.1 Définition du typage de modèles 64
4.5.2 Correspondances illégales 65
4.5.3 Conclusion 66

II Facilités de typage pour l'ingénierie des langages 67

5 Présentation de l'approche 69
5.1 Limites des approches de métamodélisation actuelles 69
5.1.1 Relation de conformité 69
5.1.2 Facilités d'ingénierie 70
5.1.3 Typage de modèles 70
5.2 METAL : un métalangage supportant une famille de systèmes de types orientés-modèle 71
5.2.1 Séparation des interfaces et des implémentations 71
5.2.2 Séparation des modélisations *in-the-small* et *in-the-large* 71
5.2.3 METAL : un MÉTAlangage pour la modélisation *in-the-LArge* 72
5.2.4 Une famille de systèmes de types orientés-modèle 74

6 Réification de concepts pour la modélisation in-the-large 77
6.1 Modélisation *in-the-small* 78
6.1.1 METAL, MOF et Ecore 79
6.1.2 Types objets 80
6.1.3 Types chemins-dépendants 81
6.1.4 Classes et *type exact* 81
6.1.5 Autres types d'éléments de modèles : types de "valeurs" 83
6.2 Modélisation *in-the-large* 83

6.2.1 Relations entre modèles : champs et opérations de modèles 83
6.2.2 Types de modèles 84
6.2.3 Métamodèles 84
6.2.4 Représentation des types de modèles et métamodèles 86
6.3 Relation de typage de modèle 87
6.4 Conclusion 87

7 Relations entre langages : sous-typage et héritage 89
7.1 Relations de sous-typage entre types de modèles 90
7.1.1 Correspondance de types objets 91
7.1.2 Correspondance de signatures d'opérations 93
7.1.3 Correspondance de champs 95
7.1.4 Correspondance de champs et de signatures d'opérations de modèles . 97
7.1.5 Correspondance de classes et instanciation 98
7.1.6 Classification des relations de sous-typage de types de modèles 99
7.1.7 Définition de quatre relations de sous-typage entre types de modèles . . 104
7.1.8 Syntaxe abstraite de METAL pour les relations de sous-typage entre types de modèles 106
7.2 Relation d'héritage entre métamodèles 107
7.2.1 Héritage, extension et redéfinition 107
7.2.2 Héritage de champs de modèles 108
7.2.3 Héritage d'opérations de modèles 109
7.2.4 Héritage de classes 109
7.2.5 Héritage non-isomorphique, héritage partiel 110
7.2.6 Héritage entre métamodèles et types chemin-dépendants 110
7.2.7 Syntaxe abstraite de METAL pour les relations d'héritage entre métamodèles 110
7.3 Conception d'un système de types orienté-modèle 110
7.3.1 Déclaration des relations de sous-typage entre types de modèles 111
7.3.2 Vérification des relations de sous-typage entre types de modèles 111
7.3.3 Une famille de systèmes de types orientés-modèle 112
7.4 Conclusion 113

III Implémentation et Validation 117

8 Implémentation d'un système de types orienté-modèle dans Kermeta 119
8.1 Présentation de Kermeta 119
8.1.1 Définition d'un langage de modélisation dédié au sein de l'environnement Kermeta 120
8.1.2 Composition des différentes préoccupations 120
8.1.3 Compilation de la composition 120
8.2 Présentation du système de types implémenté 122
8.3 Syntaxe abstraite 124
8.4 Syntaxe concrète 125
8.4.1 Métamodèles et types de modèles 126
8.4.2 Types *chemins-dépendants* 127
8.4.3 Opérations et champs de modèles 127
8.4.4 Relation de sous-typage et d'héritage 128
8.5 Sémantique 129
8.5.1 Modifications apportées au *front-end* du compilateur Kermeta 129

8.5.2 Modifications apportées au *back-end* du compilateur Kermeta : Sémantique translationnelle en Scala . . . 130
8.6 Conclusion . . . 137

9 Application au passage en forme Static Single Assignment 139
9.1 Présentation de la forme *Static Single Assignment* . . . 139
9.2 Calcul de la frontière de dominance . . . 141
9.3 Insertion de ϕ-fonctions et renommage de variables . . . 143
9.4 Réutilisation du passage en forme *Static Single Assignment* . . . 144
9.5 Adaptations pour GeCoS . . . 148
9.5.1 Graphes de flot de contrôle dans GeCoS . . . 148
9.5.2 Adaptations entre `GeCoS IR` et `SSA_Form` . . . 149
9.6 Conclusion . . . 154

IV Conclusion et perspectives 159

10 Conclusion et perspectives 161
10.1 Conclusion . . . 161
10.2 Perspectives . . . 162
10.2.1 "Fonctions de modèles" . . . 162
10.2.2 Opérateurs d'adaptation . . . 163
10.2.3 Application du typage de modèles à d'autres espaces technologiques . . . 164
10.2.4 Extension des types de modèles . . . 164
10.2.5 Évaluation expérimentale . . . 165

Bibliographie 167

Liste des figures 177

Liste des listings 179

Liste des tables 181

Liste des publications 183

Chapitre 1

Introduction générale

1.1 Langages dédiés

Le nombre et la complexité toujours croissants des préoccupations prises en compte dans les systèmes logiciels complexes (e.g., sécurité, IHM, scalabilité, préoccupations du domaine d'application) poussent les concepteurs de tels systèmes à séparer ces préoccupations afin de les traiter de manière indépendante.

Par exemple, la compilation optimisante utilise différentes représentations intermédiaires d'un programme (e.g., graphe de flot de contrôle, forme Static Single Assignment, représentation polyédrique), chacune de ces représentations étant dédiée à une préoccupation d'analyses ou d'optimisation (e.g., propagation de constante, élimination de code mort, parallélisation).

Les langages dédiés permettent de manipuler les concepts spécifiques à une préoccupation particulière tout en s'affranchissant de la complexité liée à l'apprentissage et l'utilisation de langages généralistes tels que Java ou C++.

L'ingénierie dirigée par les modèles s'appuie sur les concepts de l'orienté-objet pour modéliser et outiller des langages dédiés, permettant aux experts de séparer les différentes préoccupations métiers sur un même système en autant de langages dits de modélisation dédiés [JCV12].

Un langage de modélisation dédié est constitué d'une syntaxe abstraite, d'une syntaxe concrète et d'une sémantique. En ingénierie dirigée par les modèles, la syntaxe abstraite est généralement définie sous la forme d'un ensemble de classes et une "phrase" issue du langage est un ensemble d'objets, instances de ces classes (un modèle). La syntaxe concrète et la sémantique d'un langage de modélisation dédié sont implémentées par des transformations de modèles qui vont afficher, interpréter ou compiler les modèles.

La méthodologie et les outils développés au sein de l'ingénierie dirigée par les modèles ont facilité la création de langages de modélisation dédiés. Leur nombre n'a donc cessé de croître pour répondre aux besoins d'une population croissante d'experts métier [Kle08, HWRK11].

La compilation optimisante utilise de nombreux langages intermédiaires dédiés afin de modéliser différentes représentations intermédiaires du programme compilé. Certaines infrastructures de compilation ont fait le choix de concevoir ces langages intermédiaires sous la forme de langages de modélisation dédiés. Dans ce cas, les représentations intermédiaires des programmes compilés sont alors des modèles, transformés au cours de la compilation par différentes transformations de modèles [FYG+11].

1.2 Besoin de facilités d'ingénierie pour les langages de modélisation dédiés

La définition et l'outillage d'un langage dédié demande un effort de développement important pour un public par définition réduit. En effet, le développement de langages généralistes justifie des efforts importants de définition et d'outillage. Ce même effort se justifie plus difficilement dans le cadre de nombreux langages dédiés, qui par définition s'adressent à un nombre restreint d'utilisateurs. Ces utilisateurs ont pourtant besoin des mêmes outils que les utilisateurs de langages généralistes (éditeurs, analyseurs syntaxiques, simulateurs, compilateurs, outils de vérification, etc.).

De même que la réutilisation de briques logicielles (méthodes, objets, composants) entre différents systèmes permet de diminuer l'effort de développement d'un logiciel [BBM96, MC07], la réutilisation d'outils entre différents langages de modélisation dédiés permettrait de diminuer l'effort de développement d'un langage de modélisation dédié [KSW+13].

Certains concepts de base de l'ingénierie dirigée par les modèles sont actuellement un frein à l'établissement de facilités d'ingénierie permettant de diminuer le coût de définition et d'outillage des langages de modélisation dédiés. En particulier, la relation de conformité interdit la réutilisation par polymorphisme de sous-type au niveau des modèles et ne prend en compte que les éléments de modèle (i.e., les objets) en ignorant les transformations associées aux modèles.

Malgré la volonté de l'ingénierie dirigée par les modèles de faire des modèles des entités de première classe, placées au centre du développement logiciel, rares sont les langages et les environnements de modélisation qui offrent cette possibilité. En effet, même au sein de standards de facto comme le MOF ou Ecore, ni la notion de modèle, ni celle de métamodèle n'apparaît. Dans ces métalangages, et dans ceux qui s'appuient sur eux, un métamodèle n'est défini et manipulable qu'au travers d'un ensemble de packages.

S'abstraire de ces concepts de bas niveau sous-jacents à l'ingénierie dirigée par les modèles permettrait d'offrir aux utilisateurs des environnements de modélisation in-the-large, i.e., des environnements permettant de manipuler les modèles et leurs relations comme des entités dites de première classe, c'est à dire des concepts directement exprimables dans un langage.

1.3 Contributions

Dans les langages orientés-objet, l'abstraction et la réutilisation sont notamment fournies par le système de types et les mécanismes de sous-typage et d'héritage. Cette thèse se fonde sur des travaux préliminaires abordant le typage de modèles [SJ07, LG13] pour fournir la définition d'une famille de systèmes de types orientés-modèle. Cette définition est un socle formel permettant d'implémenter un système de types orienté-modèle supportant diverses facilités de typage au sein d'un environnement de développement de langages de modélisation dédiés. Dans ce but nous définissons :

- Une relation de typage entre les modèles et les langages de modélisation dédiés au travers des types de modèles. Les types de modèles exposent les éléments de modèles et les transformations de modèles associés à un langage de modélisation dédié et permettent de considérer les modèles comme des entités de première classe.
- Des relations de sous-typage et d'héritage entre langages de modélisation dédiés permettant la réutilisation de la structure (i.e., de la syntaxe abstraite) et du comportement (i.e., des transformations de modèles) de ces langages.

Nous avons implémenté au sein du langage Kermeta un système de types s'appuyant sur ces relations. Ce système de types supporte une relation de sous-typage et une relation d'héritage qui offrent des mécanismes de réutilisation de la structure et du comportement entre langages de modélisation dédiés. Nous utilisons ces relations pour réutiliser la structure de représentations intermédiaires issues de la compilation optimisante et des passes d'optimisation définies comme des transformations de modèles sur cette structure.

1.4 Plan

Ce document est organisé comme suit.

La Partie I détaille le contexte de cette thèse et l'état de l'art. Le Chapitre 2 introduit l'ingénierie des langages logiciels, qui s'intéresse aux approches systématiques de définition, d'outillage et de maintenance des différents langages logiciels. Le Chapitre 3 présente plus en détails l'ingénierie dirigée par les modèles (IDM) qui s'intéresse à la définition et l'outillage de langages de modélisation dédiés. La méthodologie et les outils que celle-ci propose, ainsi que leurs limites, sont discutés. Enfin, le Chapitre 4 présente les travaux existants sur l'intégration d'une sémantique de typage dans l'ingénierie dirigée par les modèles pour lever ces limites.

La Partie II présente les contributions de cette thèse pour la définition d'une famille de systèmes de types orientés-modèle. Le Chapitre 5 donne une présentation générale des contributions principales présentées dans les deux chapitres suivants. Le Chapitre 6 présente la réification de concepts et de relations de l'ingénierie dirigée par les modèles pour la modélisation in-the-large et le Chapitre 7 définit des relations de sous-typage et d'héritage entre langages de modélisation dédiés pour la réutilisation de leur structure et de leur comportement.

La Partie III présente la validation de nos travaux au travers de l'implémentation d'un système de types orienté-modèle au sein de l'environnement Kermeta (Chapitre 8) et de son application à un cas d'utilisation provenant de la compilation optimisante (Chapitre 9).

Finalement, la Partie IV conclut ce document par un résumé des contributions de cette thèse et par la présentation de perspectives de recherche, notamment au sujet des opérateurs d'adaptation entre deux types de modèles.

Première partie

Ingénierie des langages dédiés

Chapitre 2

Ingénierie des langages logiciels

Les systèmes logiciels croissent en complexité et en taille, et leurs concepteurs doivent prendre de plus en plus de préoccupations différentes en compte : sécurité, IHM, scalabilité, interopérabilité, portabilité, mais également préoccupations des domaines d'applications de plus en plus nombreux (calcul scientifique, avionique, génomique, domotique, etc.). Afin de faire face à la complexité et au nombre toujours croissants de ces préoccupations, les concepteurs de logiciels complexes cherchent à les séparer pour les traiter indépendamment [DK75, Mit90]. Une façon de traiter séparément les différentes préoccupations dans la conception d'un système logiciel est d'utiliser un langage dédié à chacune d'elle. Cette approche permet aux concepteurs de se concentrer sur une préoccupation, sur un domaine, et d'aborder les problèmes à l'aide d'un langage proche de ce domaine. Cependant l'utilisation de plusieurs langages dédiés plutôt que celle d'un langage généraliste amène une multiplication des langages logiciels, i.e., des langages artificiels impliqués dans le développement logiciel.

Dans ce chapitre, nous abordons l'ingénierie des langages logiciels, qui vise à faciliter la définition, l'outillage et la maintenance des langages logiciels par des approches systématiques et mesurables. Nous commençons par présenter les langages logiciels (Section 2.1) et la façon dont ces langages sont définis et outillés (Section 2.2). Enfin, nous présentons les motivations de l'ingénierie des langages logiciels et les outils et méthodes qu'elle propose afin de faciliter la définition et l'outillage des langages logiciels (Section 2.3).

2.1 Langages logiciels

Le terme langage logiciel englobe tous les langages artificiels impliqués dans le développement de systèmes logiciels. Les "phrases" issues de ces langages sont utilisées pour décrire, spécifier, implémenter ou manipuler des systèmes logiciels. Dans cette première section, nous présentons le concept de langages logiciels ainsi que celui de mogram (i.e., "phrase") issu d'un langage logiciel.

2.1.1 Qu'est-ce-qu'un langage logiciel

Un langage est généralement défini comme un ensemble (potentiellement) infini de phrases ou d'énoncés [Fav04a, Kle08]. Les "phrases" ou "énoncés" d'un langage sont toutes les phrases valides qui peuvent être écrites dans ce langage.

Définition 2.1. ***(Langage)*** Un langage est un *ensemble* de phrases.

Au-delà de cette définition très large, qui englobe également les langages naturels, le terme langage logiciel se réfère à tous les langages artificiels qui sont impliqués dans le développement de systèmes logiciels [FGLW09].

Définition 2.2. ***(Langage logiciel)*** Un langage logiciel est un *langage artificiel* impliqué dans le développement de *systèmes logiciels*.

Les langages logiciels englobent donc les langages de programmation aussi bien génériques (e.g., Java) que dédiés (e.g., Logo [1]) ; mais aussi les langages de modélisation : d'exigences (e.g., KAOS [DDMvL97] ou i* [Yu97]), de la variabilité logicielle (e.g., les langages de feature model [SHT06]), de processus de développement (e.g., SPEM [OMG08]), ou génériques (e.g., UML [OMG11]) ; les langages de requête (e.g., SQL [ISO11b]) ; les ontologies (e.g., Cyc [Len95]) ; ou les langages de stockage et d'échange de données (e.g., XML [W3Ca]).

2.1.2 "Phrases", "énoncés", ou *mograms*

Si les termes "phrases" et "énoncés" sont appropriés pour désigner les éléments de langages naturels, ils peuvent sembler moins intuitifs lorsqu'il s'agit de désigner d'une manière générale les éléments de langages logiciels, qui incluent non seulement des langages textuels, mais aussi des langages graphiques (comme UML). Les termes "programme", "modèle", "diagramme", "requête" ou "document", qui sont habituellement employés dans le cadre de certains langages logiciels sont inadéquats à désigner les éléments de l'ensemble des langages logiciels. Il semble en effet difficile de parler d'un programme XML ou d'un modèle Java. Enfin le terme "élément" est trop large et peut prêter à confusion dans certains contextes (en particulier dans celui de l'ingénierie dirigée par les modèles où il est employé pour désigner les éléments de modèles). C'est pourquoi nous utiliserons pour désigner les éléments d'un langage logiciel le mot-valise mogram, contraction de modèle et de programme, proposé par Kleppe [Kle08, Chap. 3].

2.1.3 Langages généralistes et langages dédiés

Une solution pour aborder le nombre et la complexité croissants de préoccupations à prendre en compte dans le développement de systèmes logiciels complexes est de séparer ces préoccupations afin de les traiter indépendamment. Une manière de séparer ces préoccupations est de modulariser le système à l'aide de différentes abstractions (procédures, modules, objets, composants, aspects, etc.). Il est possible de pousser plus loin la séparation des préoccupations en séparant non seulement les artefacts de conception et de développement mais également les langages utilisés pour décrire ces artefacts. Les langages dédiés se concentrent sur une préoccupation donnée et cherchent à faciliter l'expression de solutions aux experts de cette préoccupation.

Un langage dédié est donc un langage conçu spécifiquement pour répondre à une préoccupation, pour aborder un domaine donné. La définition même de domaine étant lâche, il n'existe pas de limite claire entre les langages dédiés d'une part et les langages généralistes d'autre part [2] [3]. Cependant, il est possible d'énoncer des caractéristiques des langages dédiés généralement admises :

- les langages dédiés cherchent à capitaliser le savoir et le savoir-faire d'un domaine ;
- les langages dédiés fournissent généralement un nombre réduit de concepts ;

1. http://el.media.mit.edu/logo-foundation/index.html
2. "Our definition inherits the vagueness of one of its defining terms : *problem domain*." [vDKV00]
3. "We will not give a definition of what constitutes an application domain and what does not." [MHS05]

– les langages dédiés permettent d'exprimer des solutions au niveau d'abstraction du domaine.

Ces caractéristiques sont à la fois la source de bénéfices et de désavantages pour les concepteurs et les utilisateurs de langages dédiés [vDKV00, MHS05].

En capitalisant le savoir d'un domaine au sein des concepts dédiés, les langages dédiés en facilitent la réutilisation, ainsi que la capitalisation du savoir-faire du domaine au travers d'outils spécialisés. Les langages dédiés offrent de plus des possibilités d'analyse, de vérification et d'optimisation spécifiques aux domaines. En effet, les solutions exprimées à l'aide de langages généralistes peuvent impliquer des motifs complexes, qui peuvent impliquer des analyses sémantiques poussées et dont il est plus difficile d'extraire les informations nécessaires à des outils d'analyse, de vérification ou d'optimisation, alors que ces informations sont par définition directement présentes dans les mograms d'un langage dédié.

La taille réduite du langage et la possibilité d'exprimer des solutions à un niveau d'abstraction approprié facilitent l'accès aux langages dédiés. En effet, le langage possède un nombre réduit de concepts, qui sont de plus familiers aux experts du domaine, ce qui facilite l'apprentissage. De plus, ces experts peuvent exprimer à l'aide d'un langage dédié des solutions d'une manière qui leur est plus naturelle, car plus proche de leur domaine d'expertise.

L'expressivité des langages dédiés permet en effet d'exprimer les mêmes solutions qu'avec un langage généraliste, mais souvent d'une manière plus simple et plus concise. De plus, les concepts utilisés sont moins génériques et donc plus parlants (au moins pour les experts du domaine), ce qui participe directement à la documentation du mogram.

À l'inverse, les langages dédiés présentent plusieurs désavantages, directement liés à leur spécificité.

Alors que le développement de quelques langages généralistes s'adressant à un très grand nombre d'utilisateurs justifie des efforts importants de définition et d'outillage, il est difficile de fournir le même effort dans le cadre de langages dédiés, qui par définition s'adressent à un nombre d'utilisateurs restreint. Les coûts de spécification, d'outillage et de maintenance d'un langage dédié sont d'autant plus importants à cause de la faible disponibilité des langages dédiés, destinés à un usage restreint, voire confidentiel. En effet, cette confidentialité rend la réutilisation de tout ou partie des langages dédiés existants difficile.

La formation des utilisateurs à un nouveau langage a également un coût, qui peut être multiplié par le nombre de langages dédiés mis en œuvre. Selon le niveau d'expertise des utilisateurs dans le domaine ciblé par le langage dédié, l'apprentissage peut être plus ou moins coûteux.

Enfin, un langage dédié peut entraîner une perte de performance par rapport à des langages généralistes donnant la possibilité à l'utilisateur de manipuler des informations de "bas niveau" (e.g., mémoire ou parallélisation).

Le choix d'utiliser un langage dédié ou un langage généraliste dépendra donc de plusieurs facteurs : l'expérience des utilisateurs dans des langages généralistes et dans le domaine ciblé par le langage dédié, des possibilités ouvertes par le langage dédié (e.g., nouvelles optimisations) et du coût de conception et d'outillage du langage dédié.

Afin de faire basculer ce compromis en faveur des langages dédiés, l'ingénierie des langages logiciels cherche à offrir des solutions facilitant la définition et l'outillage de langages. En effet, l'expertise des utilisateurs et les possibilités ouvertes par un langage dédié sont spécifiques à chaque cas d'utilisation, il n'est donc pas possible de fournir des solutions génériques permettant de diminuer les coûts qui en découlent. À l'inverse, il est possible d'offrir des méthodes et des outils qui facilitent la définition et l'outillage des langages logiciels de manière générale.

2.2 Spécification d'un langage logiciel

Un langage est par nature immatériel. Cependant, nous voulons créer et manipuler les mograms qui appartiennent à un langage. Il nous faut donc connaître la structure d'un mogram, afin d'être capable de le créer et de savoir de quelle manière il est possible de le manipuler. C'est pourquoi il est nécessaire de caractériser un langage logiciel au travers de sa spécification [Kle08, Chap. 4].

La spécification d'un langage logiciel définit à la fois les formes concrètes sous lesquelles un mogram se présente à l'utilisateur (e.g., un texte ou un diagramme) et la forme abstraite, sous laquelle un mogram est traité par un ordinateur. Les premières, les formes concrètes du mogram permettent à l'utilisateur de créer, de manipuler, et de modifier le mogram au travers d'outils (e.g., un éditeur textuel). La seconde, la forme abstraite du mogram est la forme stockée et manipulée en mémoire par la machine. La syntaxe abstraite d'un langage exprime la forme abstraite des mograms de ce langage, tandis que les syntaxes concrètes en expriment les différentes formes concrètes possibles.

Les syntaxes abstraite et concrète sont liées, l'une donnant une vue sur l'autre. Il existe donc un mapping entre les constructions, i.e., les éléments, de la syntaxe abstraite et les constructions de la syntaxe concrète. Suivre le mapping de la syntaxe concrète vers la syntaxe abstraite permet de construire la représentation abstraite d'un mogram à partir de sa représentation concrète. Dans le sens inverse, le mapping permet d'afficher un mogram contenu en mémoire sous sa représentation abstraite.

La spécification d'un langage définit également le sens associé aux mograms du langage : la sémantique du langage. La sémantique d'un langage peut aussi bien être la façon dont un mogram doit être exécuté (e.g., pour les langages de programmation) que la relation d'un mogram à une partie du monde réel (e.g., dans le cas d'un langage de modélisation descriptif, non-exécutable).

La sémantique est exprimée dans un ou plusieurs domaines sémantiques auxquels les constructions de la syntaxe (concrète ou abstraite) du langage sont reliées. C'est le mapping entre les constructions de la syntaxe du langage et les concepts d'un domaine sémantique qui donne leur sens aux mograms du langage. Dans le cas d'un langage de programmation, l'un des domaines sémantiques qu'il est possible d'utiliser est l'ensemble des instructions machines du processeur. Dans ce cas le mapping peut être exprimé au sein d'un compilateur qui traduit un programme dans le jeu d'instructions du processeur.

Définition 2.3. ***(Spécification d'un langage logiciel)*** La spécification $Spec_L = \langle AS_L, CS_L^*, M_{CS_L}*, SD_L^*, M_{SD_L}* \rangle$ d'un langage logiciel L est constitué de :

- une syntaxe abstraite AS_L, décrivant les concepts manipulés par la machine ;
- une ou plusieurs syntaxes concrètes CS_L, décrivant les concepts manipulés par l'utilisateur ;
- un *mapping* M_{CS_L} entre la syntaxe abstraite et chaque syntaxe concrète ;
- un ou plusieurs domaines sémantiques SD_L ;
- un *mapping* M_{SD_L} entre une syntaxe et chaque domaine sémantique.

Nous détaillons dans la suite la spécification d'un langage logiciel et l'illustrons sur un exemple issu de la compilation optimisante : les graphes de flot de contrôle.

Les graphes de flot de contrôle sont une forme de représentation intermédiaire (i.e., une représentation du programme compilé interne au compilateur) largement utilisée dans les compilateurs, permettant de nombreuses analyses statiques et optimisations sur le programme [ALSU06]. Le graphe de flot de contrôle est une représentation d'un programme,

Listing 2.1 – Programme calculant et affichant la puissance de x par y.

```
1 var x, y
2 x := read()
3 y := read()
4 while (y > 1) do
5   x := x * x
6   y := y - 1
7 end
8 print(x)
```

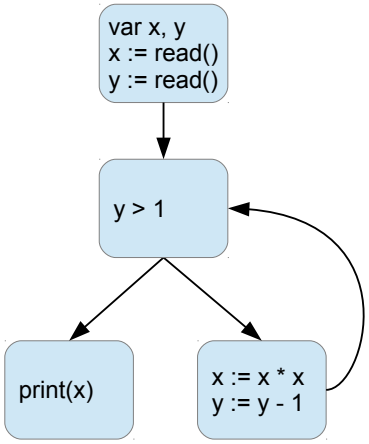

FIGURE 2.1 – Graphe de flot de contrôle du programme présenté en Listing 2.1.

dans laquelle chaque nœud représente un bloc de base (i.e., une séquence d'instructions avec un seul point d'entrée et un seul point de sortie) et où les arcs représentent les sauts d'un bloc de base à un autre. Les instructions qui forment les points d'entrée des blocs de base sont : la première instruction du programme, les instructions ciblées par une instruction conditionnelle ou une instruction de saut, et les instructions qui suivent immédiatement une instruction conditionnelle ou une instruction de saut. Ainsi, un programme sans branchement (i.e., sans boucle, conditionnelle ou saut) sera représenté par un seul bloc de base contenant l'ensemble des instructions du programme ordonnées. Le point d'entrée de ce bloc de base sera la première instruction du programme.

Le Listing 2.1 présente un court programme en pseudocode calculant et affichant la puissance d'une valeur `x` par une valeur `y`, `x` et `y` étant lues au clavier. Ce programme contient quatre instructions qui définissent un point d'entrée d'un bloc de base :

- l'instruction de déclaration des variables `x` et `y` (ligne 1) est la première instruction du programme, elle est donc le point d'entrée d'un bloc de base ;
- l'instruction de comparaison `y > 1` (ligne 4) est la cible d'un saut inconditionnel (à la fin d'un tour de boucle, la condition est réévaluée), elle est donc le point d'entrée d'un bloc de base ;
- l'instruction de multiplication `x * x` (ligne 5) est la cible d'une instruction conditionnelle (le `while`), elle est donc le point d'entrée d'un bloc de base ;
- l'instruction d'affichage `print(x)` (ligne 8) suit directement une instruction conditionnelle (le `while`), elle est donc le point d'entrée d'un bloc de base.

La Figure 2.1 présente le graphe de flot de contrôle correspondant à ce programme. Le graphe contient quatre blocs de base, donc les points d'entrée sont les instructions présentées ci-dessus. Les arcs représentent les sauts (conditionnels ou non) entre les blocs de base.

2.2.1 Syntaxes d'un langage logiciel

À la différence des langages naturels, un langage logiciel possède plusieurs syntaxes : une syntaxe abstraite et une ou plusieurs syntaxes concrètes (cf. Définition 2.3).

La syntaxe abstraite est la syntaxe qui décrit les concepts manipulables par un ordinateur, les concepts qui vont être traités de manière automatique dans différents buts : exécution (c'est le cas classique d'un langage de programmation), vérification, simulation, etc. Dans le cas d'un langage de programmation orienté-objet, la syntaxe abstraite sera par exemple l'ensemble des concepts objets traités par le compilateur ou l'interpréteur : classes, méthodes, associations, relations d'héritage, expressions (e.g., appel de méthode, déclaration de variable ou assignation), etc.

Dans le cas d'un graphe de flot de contrôle, ces concepts sont les blocs, les instructions (dont la liste dépend du langage source du compilateur) et les sauts entre blocs.

La syntaxe concrète décrit les concepts manipulables par l'utilisateur, une représentation compréhensible et manipulable par l'humain. La syntaxe concrète peut se présenter sous forme textuelle ou graphique. Dans le cas d'un langage de programmation orienté-objet, il s'agira par exemple des mots-clés et des séparateurs à disposition de l'utilisateur : `class`, `package`, `if`, `;`, `{`, etc.

Bien qu'un graphe de flot de contrôle soit par définition une représentation interne à un compilateur, il peut être utile de lui donner une syntaxe concrète, e.g., dans des buts de test ou de débogage. On peut par exemple imaginer une représentation graphique à gros grain, présentant les blocs et les sauts entre les blocs sous la forme d'un graphe dirigé. On peut également imaginer une représentation textuelle de ce même graphe dirigé dans un format proche du langage DOT [4], où les nœuds portent un identifiant et où les arcs sont donnés par une suite de triplets `identifiant "->" identifiant`.

Les syntaxes abstraite et concrète sont différentes mais sont liées. Afin de pouvoir passer d'une syntaxe à l'autre, il existe en effet un mapping reliant les concepts de la syntaxe abstraite à ceux de la syntaxe concrète (M_{CS_L} dans la Définition 2.3). Ce mapping peut être arbitrairement complexe, puisque la syntaxe abstraite peut contenir des concepts sans correspondance directe avec ceux de la syntaxe concrète et inversement.

En effet, certains concepts présents dans la syntaxe abstraite ne sont pas directement manipulables par l'utilisateur, mais sont extraits automatiquement des données (il peut s'agir par exemple, d'artefacts utiles à la compilation). À l'inverse, certains éléments de la syntaxe concrète peuvent être simplement ignorés dans la syntaxe abstraite (c'est le cas des séparateurs, par exemple), ou différents concepts manipulés par l'utilisateur peuvent être traduits vers un même concept de la syntaxe abstraite.

Le passage de la syntaxe concrète à la syntaxe abstraite se fait grâce à ce mapping. Un outil va lire les fichiers contenant les données entrées par l'utilisateur à l'aide de la syntaxe concrète et les traduire dans les concepts correspondants de la syntaxe abstraite, les rendant manipulables par la machine. Dans le cas d'un langage textuel, cette étape est nommée analyse syntaxique. À l'inverse, le mapping peut servir à afficher un mogram dans sa représentation concrète depuis sa représentation abstraite.

2.2.2 Sémantique d'un langage logiciel

La sémantique d'un langage définit la signification des différentes constructions du langage et donne ainsi un sens aux mograms de ce langage. Selon ses usages, la sémantique peut être spécifiée de manière complète et non-ambigüe, ou de manière moins précise.

Si les mograms du langage doivent être interprétés ou manipulés par une machine, la sémantique du langage doit être complète et non-ambigüe. Par exemple, un langage exécutable

4. http://www.graphviz.org/doc/info/lang.html

(e.g., un langage de programmation) doit posséder une sémantique exécutable qui donne un unique sens à chaque mogram du langage. Elle doit donc être exprimée de manière complète (elle doit définir une signification pour chaque construction du langage) et non-ambigüe (une construction du langage ne peut pas avoir plusieurs significations possibles).

Si les mograms du langage sont interprétés et manipulés uniquement par des humains, les contraintes peuvent être moins importantes. Par exemple, la sémantique d'un langage de description graphique, sans outillage informatique, peut rester informelle (e.g., sous la forme de langage naturel). Elle peut être exprimée de manière complète et non-ambigüe à l'aide de langages ou de formalismes appropriés mais elle peut également être exprimée de manière moins précise.

La sémantique d'un langage est définie grâce à un ou plusieurs domaines sémantiques et à des mappings entre les concepts de la syntaxe abstraite ou d'une syntaxe concrète et les concepts du domaine sémantique. La sémantique d'un langage peut être exprimée dans différents domaines sémantiques (e.g., la sémantique opérationnelle structurelle [Plo81], la logique de Hoare [Hoa69], le langage naturel, d'autres langages logiciels, etc.). Il n'est pas rare qu'un langage logiciel possède plusieurs domaines sémantiques, et donc plusieurs mappings vers ces domaines sémantiques. Par exemple, la sémantique du langage C est à la fois exprimée en langage naturel et dans différents jeux d'instructions de différents processeurs. Dans le cas où le domaine sémantique est le langage naturel, le mapping est exprimé au travers de la norme ISO/IEC 9899 :2011 [ISO11a]. Dans le cas où les domaines sémantiques sont les jeux d'instructions de processeurs, le mapping est exprimé par les différents compilateurs qui génèrent les instructions du processeur. Il se pose alors la question de la cohérence entre ces différentes sémantiques, qui, si elle est un problème important, n'est pas abordée dans cette thèse.

La sémantique d'un graphe de flot de contrôle peut, par exemple, être implémentée par un générateur de code assembleur qui génère pour chaque instruction d'un bloc les instructions correspondantes en assembleur et qui génère un saut pour chaque arc. Dans ce cas le domaine sémantique est le langage assembleur, et le mapping est exprimé à travers le générateur de code.

2.2.3 Validité d'un *mogram* par rapport à la spécification d'un langage

Selon la manière dont un mogram est construit, il peut être nécessaire de vérifier qu'il appartient à un langage, i.e., qu'il est valide par rapport à la spécification du langage. Par exemple, un programme Java peut contenir des erreurs vis-à-vis de la syntaxe, ou de la sémantique du langage Java. Ainsi, avant de l'utiliser pour générer du bytecode, il est nécessaire de vérifier que ce programme est bien un programme Java valide, i.e., de vérifier que ce programme est valide par rapport à la spécification du langage Java. De la même manière que la spécification d'un langage se découpe en plusieurs parties, la vérification de la validité d'un mogram peut se faire à plusieurs niveaux : au niveau syntaxique ou au niveau sémantique.

Les vérifications syntaxiques permettent de détecter la présence d'éléments invalides dans un mogram. Par exemple, une `voiture` déclarée au sein d'un graphe de flot de contrôle. L'équivalent en langage naturel est la présence d'un mot d'une autre langue dans une phrase française.

Les vérifications syntaxiques permettent également de détecter des violations des contraintes syntaxiques impliquant plus d'un élément. Par exemple, un arc est un élément valide d'un graphe de flot de contrôle, mais un arc seul, sans nœud source ni cible, pourrait être interdit par la syntaxe. De la même manière une phrase ne commençant pas par une majuscule peut ne contenir que des éléments valides un à un, mais est invalide dans son ensemble.

Enfin des vérifications sémantiques peuvent détecter des mograms valides d'un point de vue syntaxique mais qui n'ont pas de sens. Par exemple un graphe de flot de contrôle contenant des instructions mal typées (e.g., une division d'un entier par une chaîne de caractères) est valide d'un point de vue syntaxique, mais pas d'un point de vue sémantique. Une phrase peut respecter toutes les contraintes syntaxiques (i.e., grammaticales) du français sans avoir de sens pour autant.

Ces vérifications sont généralement pratiquées par les outils développés pour le langage : éditeurs, compilateurs, interpréteurs, etc. Les vérifications syntaxiques peuvent être appliquées sur la forme concrète comme sur la forme abstraite du mogram selon la manière dont les mograms sont créés (par un utilisateur utilisant la syntaxe concrète ou par une machine utilisant directement la syntaxe abstraite).

2.2.4 Métalangages et méta-outils

Afin de spécifier un langage logiciel, il est nécessaire de décrire sa syntaxe abstraite, sa ou ses syntaxes concrètes, ainsi que les mappings entre syntaxes et vers le ou les domaines sémantiques. Ces descriptions se font elles-mêmes à l'aide de langages, dits métalangages. Ces métalangages peuvent également être outillés, soit pour fournir des facilités pour la manipulation des spécifications qui en sont issues (i.e., leurs mograms), par exemple à l'aide d'éditeurs, soit afin de générer des outils pour ces spécifications, i.e., afin de générer des outils manipulant les mograms issus des mograms de ces métalangages. De tels outils sont appelés méta-outils, qu'ils soient génératifs ou non. Au cours de l'évolution de l'informatique, de nombreux métalangages ont vu le jour au sein de différents paradigmes, ou espaces technologiques, chacun proposant de nouvelles méthodes de définition et d'outillage des langages logiciels.

2.2.4.1 Métalangages

En linguistique, un métalangage peut-être vu comme un "langage (au sujet) de langages" [5] ou comme un "langage dans le contexte des représentations et évaluations linguistiques" [6] [JCG04]. Un métalangage est donc un langage qui permet d'évaluer, de décrire et de manière plus générale de raisonner sur un langage.

En ingénierie des langages logiciels, un métalangage est un langage utilisé pour spécifier tout ou partie d'un langage. Les mograms d'un métalangage sont donc des sous-ensembles de la spécification de langages, i.e., la spécification d'une préoccupation sur un langage (e.g., syntaxe abstraite, syntaxe concrète ou sémantique). Tout langage dédié à la spécification de la syntaxe (abstraite ou concrète) ou de la sémantique d'un langage logiciel est donc un métalangage.

Par exemple, l'Extended Backus-Naur Form (EBNF) est un métalangage qui permet d'exprimer des grammaires, i.e., de spécifier des syntaxes concrètes textuelles, sous la forme de règles de production. Suivre les règles de production permet de produire n'importe quel mogram appartenant au langage, sous sa forme concrète textuelle. Un mogram de l'EBNF est donc la spécification de la syntaxe textuelle d'un langage.

Chaque règle de production est composée d'une partie gauche et d'une partie droite séparées par un caractère égal (=) et est terminée par un point virgule. La partie gauche possède un symbole et la partie droite donne la façon de produire un élément de mogram à partir de ce symbole. La partie droite peut comporter une suite de symboles (séparés par des virgules) qui peuvent être soit des appels à d'autres règles (on parle alors de symboles non-terminaux), soit des éléments textuels entre doubles guillemets (on parle alors de symboles terminaux,

5. *"language about language"*
6. *"language in the context of linguistic representations and evaluations"*

```
graph = "graph", identifier, "{", contents, "}" ;

contents = element, { element } ;

element = node | edge ;

node = identifier ;

edge = identifier, "->", identifier ;

identifier = letter , { letter | digit } ;

letter =
  "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
  "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
  "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" ;

digit =
  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

```
graph G {
  N1
  N2
  N3
  N1 -> N2
  N2 -> N1
  N2 -> N3
}
```

FIGURE 2.2 – Spécification d'une syntaxe de graphe utilisant l'*Extended Backus-Naur Form* (EBNF) et exemple d'utilisation de cette syntaxe.

i.e., ils ne mènent pas à une autre règle de production). Il est également possible d'exprimer des alternatives (deux symboles séparés par une barre verticale, `|`) ou des répétitions (entre accolades).

La figure 2.2 présente la spécification d'une syntaxe concrète textuelle pour des graphes, exprimée dans le métalangage EBNF (à gauche) et un exemple d'utilisation de cette syntaxe (à droite).

La règle `graph` (première ligne) permet de produire un mogram décrivant un graphe sous sa forme concrète textuelle. Cette forme textuelle est composée du mot-clé `graph` suivi d'un identifiant (appel à la règle `identifier`) et d'un contenu (appel à la règle `contents`) entre crochets. Le contenu d'un graphe est au moins un élément (appel à la règle `element`) qui peut être suivi d'autres éléments. Un élément de graph est un nœud ou un arc (alternative entre les règles `node` et `edge`). Un nœud est simplement représenté par un identifiant, tandis qu'un arc est représenté par une flèche (`->`) entre deux identifiants. Finalement un identifiant commence par une lettre (appel à la règle `letter`), suivie d'une suite de lettres et de chiffres.

Un métalangage est dit métacirculaire quand il est suffisant pour se spécifier lui-même, c'est à dire que tous les concepts du métalangage peuvent être exprimés en utilisant le métalangage lui-même. Le métalangage est alors l'un de ses propres mograms. La métacircularité d'un métalangage permet d'unifier la représentation d'une préoccupation sur un langage, y compris au niveau du métalangage. Cette unification, en plus d'éviter une multiplication des formalismes, offre la possibilité de réutiliser les outils définis pour les mograms d'un métalange sur le métalangage lui-même. Par exemple, l'EBNF peut être utilisé pour spécifier sa propre grammaire (cf. Listing 2.2). Les outils écrits pour traiter une grammaire EBNF (e.g., un générateur d'analyseurs syntaxiques) pourront donc être utilisés sur la grammaire spécifique de l'EBNF.

2.2.4.2 Méta-outils

Les outils définis sur un métalangage, i.e., les méta-outils, sont des outils qui s'appliquent à toutes les spécifications définies à l'aide de ce métalangage.

Au-delà des outils "classiques" pour un langage qui manipulent les spécifications issues du métalangage (e.g., éditeur, analyseur syntaxique), l'intérêt des méta-outils se trouvent éga-

Listing 2.2 – Spécification *métacirculaire* de l'*Extended Backus-Naur Form* (EBNF) [Wik13]

```
grammar = { rule } ;

rule = lhs , "=" , rhs , ";" ;

lhs = identifier ;

rhs =
  identifier | terminal | "[" , rhs , "]" | "{" , rhs , "}" |
  "(" , rhs , ")" | rhs , "|" , rhs | rhs , "," , rhs ;

terminal =
  "'" , character , { character } , "'" |
  '"' , character , { character } , '"' ;

character = letter | digit | symbol | "_" ;

identifier = letter , { letter | digit | "_" } ;

letter =
  "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
  "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
  "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

symbol =
  "[" | "]" | "{" | "}" | "(" | ")" | "<" | ">" |
  "'" | '"' | "=" | "|" | "." | "," | ";" ;
```

lement dans les méthodes génératives. Les méta-outils génératifs permettent l'outillage de langages spécifiés par le métalangage, en générant des outils pour ces langages. La figure 2.3 présente une telle situation. Les méta-outils définis sur le métalangage manipulent les spécifications de langages qui appartiennent à ce métalangage. Certains de ces méta-outils manipulent les spécifications afin de générer des outils définis pour ces langages. Les outils générés manipulent donc des mograms des langages spécifiés en utilisant les métalangages.

La figure 2.3 donne l'exemple d'Antlr [PQ95], un méta-outil qui génère des analyseurs syntaxiques à partir de la spécification d'une grammaire EBNF. Il s'agit donc d'un outil défini sur le métalangage EBNF, qui génère des outils pour les langages dont la syntaxe concrète est spécifiée grâce à l'EBNF.

Les méta-outils génératifs permettent donc d'automatiser tout ou partie des tâches d'outillage d'un langage à partir de sa spécification (e.g., génération de l'éditeur textuel ou graphique, génération de simulateur).

2.2.4.3 Espaces technologiques et langages logiciels

La notion d'espace technologique a été introduite par Kurtev et al. et désigne un contexte de travail regroupant concepts, savoir, savoir-faire et outils [KBA02]. Généralement associé à une communauté, un espace technologique peut être vu comme un ensemble de connaissances (body of knowledge en anglais) et les compétences et outils qui lui sont associés. Dans le cadre de la définition et l'outillage de langages logiciels, différents espaces technologiques ont vu apparaître des métalangages et méta-outils, développés au sein de différents domaines de recherche, en fonction de leurs spécificités et de leurs besoins.

Le domaine des grammaires formelles ou *grammarware*, formalisé par Chomsky [Cho56], est probablement l'un des plus anciens espaces technologiques de l'informatique. Au cœur de

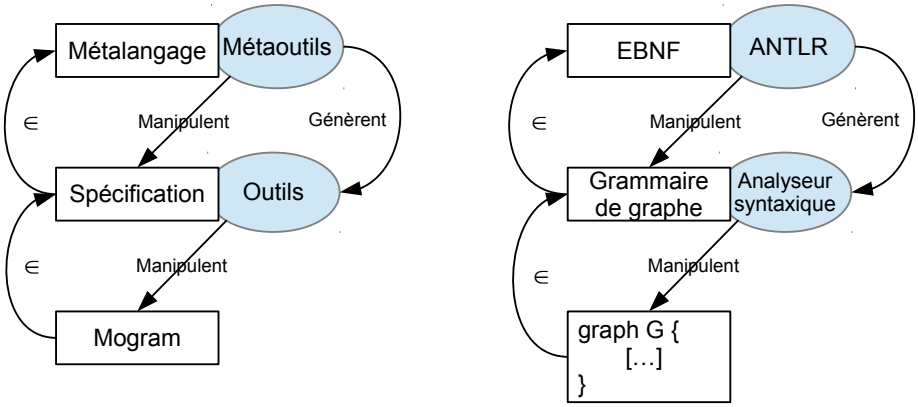

FIGURE 2.3 – Méta-outils générant des outils pour les spécifications écrites dans le métalangage sur lequel ils sont définis et exemple du métalangage EBNF et du méta-outil Antlr.

la spécification des langages de programmation, les grammaires formelles et plus particulièrement les grammaires non contextuelles permettent de décrire la syntaxe concrète textuelle. Les grammaires non-contextuelles sont des grammaires dont la partie gauche des règles de production ne peut contenir qu'un seul symbole non-terminal. Cette classe de grammaire permet de définir des algorithmes d'analyse syntaxique efficaces. A l'inverse, définir si un mogram appartient à une grammaire contextuelle (la classe plus générale de grammaire qui inclut les grammaires non-contextuelles) est un problème PSPACE-complet. La classe des grammaires non-contextuelles représente donc la classe de grammaire la plus expressive permettant de définir des algorithmes d'analyse syntaxique efficaces.

Les deux métalangages principaux utilisés pour la définition de grammaires sont la Backus-Naur Form (BNF) et l'Extended Backus-Naur Form (EBNF). Autour de ces langages ont été développés des méta-outils tels que Lex, YACC et Antlr qui permettent de générer automatiquement une syntaxe abstraite et un analyseur syntaxique à partir de la grammaire d'un langage. Grâce à cet analyseur, il est possible pour une machine de "lire" la forme concrète textuelle d'un mogram appartenant au langage et d'en générer l'arbre de syntaxe abstrait (i.e., la forme abstraite), manipulable par la machine. Certains outils vont même plus loin : CENTAUR propose de générer un environnement de programmation dédié à un langage à partir des spécifications de sa syntaxe et de sa sémantique [CIK90]. Cet environnement inclut notamment un éditeur et un interpréteur générés respectivement grâce à la spécification de la syntaxe et à la spécification de la sémantique.

De plus, de nombreuses analyses ont été développées sur les arbres de syntaxe abstrait, permettant de détecter des erreurs de typage, ou d'appels à des variables non-déclarées. C'est également à partir de l'arbre de syntaxe abstrait que sont obtenues d'autres représentations facilitant d'autres analyses et optimisations (e.g., le graphe de flot de contrôle).

Le domaine de l'échange de données structurées ou *XMLware* s'appuie sur l'Extended Markup Language (XML) [W3Ca], une spécification du World Wide Web Consortium (W3C) pour la définition de langages de balisage. Le langage XML est accepté comme un standard pour l'échange de données au travers de documents XML.

Le métalangage XML Schema Definition (XSD) [W3Cb] permet d'exprimer des règles de

validité sur les données contenues par un document XML dans un schéma XML. Un schéma définit un ensemble de concepts utilisables dans un document XML. Á l'aide de ces règles, il est possible de déterminer si un document XML donné est valide par rapport à un schéma, i.e., de déterminer si le document XML est un mogram du langage spécifié par le schéma. Il est donc possible de créer des outils spécifiques, capables de traiter les documents XML valides par rapport à un schéma XML. Un schéma XSD forme donc la spécification de la syntaxe concrète d'un langage, et les documents XML valides par rapport à ce schéma sont des mograms de ce langage.

Afin de manipuler des documents XML, le W3C a également proposé l'Extensible Spreadsheet Language Transformations (XSLT) et XQuery. XSLT est un langage déclaratif qui permet de transformer un document XML en un autre (ou en un autre format : HTML, texte). XQuery est un langage de requête utilisé pour extraire et manipuler des données de documents XML et de base de données.

L'ingénierie dirigée par les modèles ou *modelware* place les modèles au cœur du processus de développement de systèmes complexes. L'ingénierie dirigée par les modèles prône l'utilisation de multiples modèles d'un système, chacun permettant d'aborder une préoccupation particulière sur le système. De nombreuses définitions de la notion de modèle existent, au sein même de l'ingénierie dirigée par les modèles [MFBC12]. Cependant, ces définitions ne sont pas contradictoires et tendent à converger vers la même vision de ce qu'est un modèle.

Définition 2.4. ***(Modèle [JCV12])*** "Un modèle est un ensemble de faits caractérisant un aspect d'un système dans un objectif donné. Un modèle représente donc un système selon un certain point de vue, à un niveau d'abstraction facilitant par exemple la conception et la validation de cet aspect particulier du système."

Un modèle est donc une représentation d'un aspect du système. Cette représentation est construite dans un but donné et doit donc fournir une abstraction pertinente par rapport à ce but. Une abstraction pertinente (un modèle pertinent) est une abstraction suffisante et nécessaire pour répondre aux questions correspondant à ce but comme le système modélisé aurait lui-même répondu et qui abstrait les autres informations. Ces modèles sont des mograms de langage de modélisation, qu'ils soient généralistes (e.g., UML) ou dédiés. Selon l'approche choisie, il peut donc être nécessaire de non seulement modéliser le système en développement mais aussi de modéliser le domaine sous la forme d'un langage de modélisation dédié permettant de capitaliser le savoir et le savoir-faire du domaine.

De nombreux métalangages ont été proposés au sein de la communauté de l'ingénierie dirigée par les modèles afin de spécifier et d'outiller les langages de modélisation dédiés. Nous abordons en détail la spécification des langages de modélisation dédiés dans le chapitre suivant (Chapitre 3).

Certains langages et outils forment un pont entre plusieurs espaces technologiques. XText[7] et EMFText [HJK+09] sont par exemple des outils permettant de décrire le mapping entre une syntaxe abstraite décrite à l'aide d'un métalangage de l'ingénierie dirigée par les modèles (Ecore) et une syntaxe concrète décrite à l'aide d'une grammaire formelle. L'Eclipse Modeling Framework [SBPM09] génère un mapping entre une syntaxe abstraite Ecore et une syntaxe concrète XML, et fournit des équivalences entre schéma XSD et syntaxe abstraite Ecore. TOM [BBK+07] est un langage de réécriture et de manipulation d'arbres, qui peut être aussi bien utilisé pour manipuler un arbre de syntaxe abstrait créé à partir d'une grammaire formelle,

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

que des documents XML [CMR09] ou pour écrire des transformations de modèles [BCMP12]. De tels outils sont au cœur de l'ingénierie des langages logiciels, qui cherche à développer des méthodes fournissant le meilleur des différents paradigmes de définition et d'outillage de langages logiciels.

2.3 Multiplication des langages logiciels et ingénierie des langages logiciels

De plus en plus de langages logiciels sont développés, pour répondre aux besoins de nouveaux utilisateurs et pour permettre de réduire la complexité du développement des systèmes logiciels [Kle08, FGLW09, HWRK11]. Avec ce nombre grandissant de langages est né un besoin pour des approches systématiques du cycle de vie des langages logiciels : développement, utilisation, évolution. C'est à ces approches que s'intéresse l'ingénierie des langages logiciels.

2.3.1 Multiplication des langages logiciels

Le nombre de langages logiciels est en constante augmentation [Kle08, Chap.1], principalement pour deux raisons :

La première est l'éventail de domaines abordés par des systèmes logiciels de plus en plus large (e.g., avionique, téléphonie, domotique, etc). Aborder ce nombre grandissant de domaines demande de prendre en compte les spécificités de ces domaines dans les langages logiciels. Pour prendre en compte ces spécificités, de nouveaux langages, particulièrement des langages dédiés sont créés [HWRK11]. En effet, afin de concevoir des systèmes logiciels de plus en plus complexes et mettant en jeu de plus en plus de domaines, les experts de différents domaines doivent collaborer. Dans le même temps, ils ont besoin d'outils pour représenter la connaissance et capitaliser le savoir-faire de leurs propres domaines. Il faut donc créer, mais également rendre accessibles ces langages logiciels pour les utilisateurs, qui ne sont pas spécialistes de la définition et de l'outillage, ni même de l'utilisation, de langages logiciels.

La deuxième raison est la taille et la complexité sans cesse croissantes des systèmes logiciels. Pour conserver la maîtrise du développement de tels systèmes, ces systèmes sont décomposés en modules inter-connectés, permettant aux développeurs d'encapsuler une préoccupation au sein d'un module d'une part et de raisonner au niveau du système en abstrayant les détails internes aux modules d'autre part [DK75]. De nouvelles abstractions (méthodes, objets, aspects, composants, etc.) ont été introduites dans les langages de programmation pour supporter cette modularisation.

Ces deux raisons conduisent à la création continue de nouveaux langages logiciels (pour aborder un nouveau domaine ou proposer une nouvelle manière d'aborder la complexité), ainsi qu'à l'évolution des langages logiciels existants [Kle08]. Parmi tous ces langages logiciels, il n'est pas rare que certains langages partagent des caractéristiques communes (e.g., structure, outils). De la même manière l'évolution d'un langage conduit à un ensemble de versions partageant une base commune. Ces langages logiciels, qui partagent des caractéristiques communes, forment des familles de langages logiciels [LDA13].

On peut trouver un exemple de cette multiplication des langages logiciels dans le domaine de la compilation optimisante, qui utilise de nombreux langages intermédiaires dédiés afin de modéliser différentes représentations intermédiaires du programme compilé. Par exemple, il n'existe pas un seul langage de graphe de flot de contrôle utilisé comme représentation intermédiaire. Au contraire, il existe une ou plusieurs implémentations spécifiques des graphes de flot de contrôle par compilateur. En effet, les compilateurs doivent prendre en compte des

constructions de langage différentes en fonction de leur(s) langage(s) source(s). Ces constructions différentes peuvent apparaître dans le graphe de flot de contrôle (e.g., une instruction spécifique à un langage de programmation donné). De plus, différentes versions des graphes de flot de contrôle, dédiées à différentes tâches d'analyse ou d'optimisation peuvent être utilisées par un même compilateur. L'ensemble des langages de graphes de flot de contrôle forme donc une famille de langages logiciels, partageant une structure et des analyses et optimisations communes.

2.3.2 D'un "artisanat" à une ingénierie des langages logiciels

Cette augmentation du nombre de langages logiciels amène une nécessité de passer d'un "artisanat" de la définition et de l'outillage des langages logiciels, pratiqué par quelques spécialistes des langages, à une approche systématique pour le développement de langages logiciels outillés, utilisable par un grand nombre d'utilisateurs d'un autre domaine d'expertise. Il est donc nécessaire de proposer de nouveaux outils et méthodes pour le développement des langages logiciels et de mettre à portée des experts métier les outils existants (i.e., qui ne sont pas spécialistes de la définition et de l'outillage des langages).

Certains de ces outils et méthodes existent et sont utilisés depuis longtemps déjà. Par exemple, les générateurs d'analyseurs lexicaux et syntaxiques tels que Lex, YACC et Antlr permettent l'outillage automatique des langages logiciels définis à partir des métalangages BNF et EBNF. Cependant, maîtriser ces outils, pourtant bien établis, nécessite un effort qui n'est généralement pas à la portée des experts métier.

L'ingénierie des langages logiciels s'intéresse donc aux approches systématiques de gestion du cycle de vie des langages logiciels (de la conception et de la mise en œuvre à l'utilisation et l'évolution). De telles approches, mesurées et automatisées, permettent de diminuer les coûts de spécification, d'implémentation et de maintenance d'un nouveau langage logiciel. De plus, l'automatisation peut rendre accessible la définition et l'outillage de langages aux experts métiers (i.e., aux non-informaticiens ou aux informaticiens ayant développé une expertise dans un autre domaine que celui de la spécification ou l'implémentation de langages logiciels). Ainsi, ces experts métier sont non seulement capables de manipuler des langages proches de leur domaine d'application, mais également de participer à la création même de ces langages.

Ce besoin de faciliter la définition et l'outillage des langages logiciels a conduit à la création d'environnements de développement de langages (language workbenches en anglais) [Mar05]. Les environnements de développement de langages fournissent aux concepteurs de langages logiciels des (méta)langages et des (méta-)outils afin de rendre la création de langages logiciels plus facile. CENTAUR [CIK90] est probablement l'un des premiers environnements de développement de langages créés. Les approches plus récentes incluent l'Eclipse Modeling Framework [SBPM09] et les outils de son écosystème (e.g., Xtext, EMFText, Xtend [8], Kermeta [9], ATL [10]), MetaEdit+ [TR03], Monticore [GKR+08], Spoofax [KV10] et MPS [Voe11]. Ces environnements de développement de langages permettent généralement de ne spécifier que la syntaxe abstraite ou la syntaxe concrète, pour ensuite générer l'autre syntaxe et le mapping entre elles (e.g., un analyseur syntaxique). Ils proposent également de générer des outils tels que des éditeurs évolués (e.g., avec auto-complétion et coloration syntaxique).

Malgré les facilités offertes par les environnements de développement de langages, la définition et l'outillage d'un langage logiciel restent des tâches longues et coûteuses. En effet, si la spécification de la syntaxe abstraite, de la syntaxe concrète et de leur mapping bénéficient

8. http://www.eclipse.org/xtend/
9. http://www.kermeta.org/
10. http://www.eclipse.org/atl/

des facilités offertes par les métalangages et outils, ce n'est généralement pas le cas de l'expression de la sémantique d'un langage logiciel. Définir un compilateur, un interpréteur ou un vérificateur demande une connaissance approfondie du domaine du langage. Ceci rend l'automatisation de telles tâches complexe, et laisse cette charge aux mains du concepteur du langage. C'est pourquoi de nouvelles facilités d'ingénierie développées au sein de l'ingénierie des langages logiciels permettraient de diminuer les coûts de définition et d'outillage des langages logiciels, et particulièrement de faciliter l'apparition de langages dédiés.

2.4 Conclusion

Le nombre de langages logiciels, i.e., de langages artificiels impliqués dans les développements logiciels, ne cesse de croître. En effet, de nouveaux langages logiciels dédiés aux différentes préoccupations du développement logiciel voient le jour pour permettre d'aborder de manière indépendante ces préoccupations. Ces langages dédiés permettent de capitaliser le savoir et le savoir-faire liés à une préoccupation. Ils offrent de plus un moyen d'exprimer des solutions dans un langage proche du domaine de cette préoccupation, ce qui facilite leur apprentissage et leur utilisation aux experts du domaine. Bien que ces langages dédiés offrent de nombreux bénéfices, ils restent longs et coûteux à définir et outiller.

L'ingénierie des langages logiciels s'intéresse aux approches systématiques du cycle de vie des langages logiciels (conception, implémentation, maintenance, etc.). De telles approches peuvent permettre de réduire les coûts de définition et d'outillage des langages logiciels, facilitant ainsi la création de langages dédiés. Il existe déjà de nombreux outils qui permettent de spécifier la syntaxe abstraite ou la syntaxe textuelle d'un langage rapidement et facilement. De mêmes, des outils générant des mappings entre les syntaxes abstraites et concrètes, sous la forme d'outils tels que des éditeurs, analyseurs syntaxiques ou pretty-printers existent.

Cependant la mise en place de l'outillage d'un langage reste coûteuse, entre autres à cause de l'outillage lié à la sémantique du langage. Des outils complexes tels que les compilateurs, les interpréteurs ou les outils de vérifications sont encore très souvent définis à partir de zéro.

Dans le chapitre suivant, nous détaillons la métamodélisation, la méthode de spécification des langages de modélisation dédiés issue de l'ingénierie dirigée par les modèles. La métamodélisation est centrée sur la spécification de la syntaxe abstraite du langage sous la forme d'un graphe de classes, autour de laquelle s'articule le reste de la spécification du langage. Nous montrons également comment certains concepts de base freinent le développement de facilités d'ingénierie pour la définition et l'outillage des langages de modélisation dédiés, et particulièrement pour la réutilisation d'outils entre différents langages d'une même famille.

Chapitre 3

Définition de langages de modélisation dédiés

L'ingénierie dirigée par les modèles prône la séparation des préoccupations sur un système logiciel au sein de modèles, chaque modèle étant exprimé dans un langage de modélisation dédié à la préoccupation qu'il traite. Cette approche entraîne la multiplication des langages de modélisation dédiés, alors que leur définition et leur outillage restent des tâches longues et coûteuses.

Dans ce chapitre, nous présentons la métamodélisation : la méthode de définition d'un langage de modélisation dédié au sein de l'ingénierie dirigée par les modèles (Section 3.1). La métamodélisation place la syntaxe abstraite du langage au centre de la définition d'un langage de modélisation. La syntaxe concrète, la sémantique et les outils du langage sont ensuite définis à l'aide de transformations de modèles. Nous montrons également pourquoi la relation de conformité est un frein à l'établissement de facilités d'ingénierie qui permettrait de diminuer les coûts de définition et d'outillage des langages de modélisation dédiés (Section 3.2). Enfin, nous présentons des outils et méthodes développés dans la communauté de l'ingénierie dirigée par les modèles afin de faciliter la manipulation des modèles et les limites de ces approches (Section 3.3).

3.1 Spécification de langages de modélisation dédiés

Les langages de modélisation dédiés permettent de modéliser une préoccupation ou un aspect particulier d'un système dans un langage dédié fournissant les concepts spécifiques au domaine de cette préoccupation ou de cet aspect.

L'ingénierie dirigée par les modèles plaçant les modèles au cœur du développement, il n'est pas surprenant que les langages de modélisation dédiés soient eux-mêmes spécifiés à l'aide de modèles.

3.1.1 Métamodèle

Le (ou les) modèle utilisé pour spécifier un langage de modélisation dédié est appelé un métamodèle (i.e., un modèle issu d'un métalangage). La notion de métamodèle ne fait pas l'objet d'un consensus dans la communauté de l'ingénierie dirigée par les modèles. La table 3.1 présente plusieurs définitions de la notion de métamodèle provenant de la littérature.

TABLE 3.1 – Définitions de *métamodèle* dans la littérature

Bezivin [BG01]	"A meta-model defines a set of concepts and the relations between these concepts and is used as an abstraction filter in a particular modeling activity."
Favre [Fav04a]	"A metamodel is a model of a modelling language."
Bezivin *et al.* [BJT05]	"As models, metamodels are also composed of elements. The metamodel elements provide a typing scheme for model elements."
Object Management Group [OMG06]	"A metamodel is a model used to model modeling itself. A metamodel is also used to model arbitrary metadata (for example software configuration or requirements metadata). Metamodels provide a platform independent mechanism to specify the following : – The shared structure, syntax, and semantics of technology and tool frameworks as metamodels. – A shared programming model for any resultant metadata (using Java, IDL, etc.). – A shared interchange format (using XML)."
Găsevic *et al.* [GKH07]	"A metamodel is a specification model, which leads us to the conclusion that metamodels should be used to validate models represented in a specific language. That is, a metamodel makes statements about what can be expressed in the valid models of a certain modeling language."
Egea et Rusu [ER10]	"Metamodel descriptions define the structure and semantics of metadata. In a nutshell, the MOF modeling elements are classes [...] ; associations [...] ; inheritance or generalization relationships [...] ; operations [...] ; attributes [...] ;data types [...] ; and packages [...]."
Jézéquel *et al.* [JCV12]	"Un métamodèle est un modèle qui définit le langage d'expression d'un modèle [OMG06], c'est-à-dire le langage de modélisation."

Le métamodèle peut être vu comme une partie de la spécification d'un langage de modélisation dédié : la spécification du langage est alors l'ensemble des métamodèles. C'est par exemple la définition employée par Favre : "un métamodèle est le modèle d'un langage de modélisation" [Fav04a]. Un modèle caractérisant un aspect particulier du système qu'il modélise (cf. définition 2.4), un métamodèle est donc, pour Favre, la spécification d'un aspect particulier du langage de modélisation dédié. Bézivin et al. précisent encore cette vision : pour eux, un métamodèle est un ensemble de concepts [BG01] qui fournissent un "schéma de typage" aux éléments des modèles du langage [BJT05]. Cette vision d'un métamodèle est donc proche de celle de la syntaxe abstraite d'un langage : un ensemble de concepts spécifiant quels éléments peuvent appartenir à un mogram du langage.

Le métamodèle peut aussi être vu comme la spécification complète d'un langage de modélisation dédié. Il est alors composé de différents mograms issus de différents métalangages. C'est la vision défendue par l'Objet Management Group (OMG) et par Egea et Rusu qui définissent un métamodèle comme décrivant à la fois la syntaxe et la sémantique du langage [OMG06, ER10]. De même, Găsevic et al. et Jézéquel et al. définissent un métamodèle

comme un modèle de spécification d'un langage [GKH07], i.e., un modèle qui définit le langage dans lequel est exprimé un modèle [JCV12].

Nous avons choisi de considérer dans la suite de cette thèse un métamodèle comme la spécification complète d'un langage de modélisation dédié.

Définition 3.1. ***(Métamodèle d'un langage de modélisation dédié)*** Le métamodèle $MM_L = \langle AS_L, CS_L^*, M_{CS_L}*, SD_L^*, M_{SD_L}* \rangle$ d'un langage de modélisation dédié L est constitué de :

- une syntaxe abstraite AS_L, décrivant les concepts manipulés par la machine ;
- une ou plusieurs syntaxes concrètes CS_L, décrivant les concepts manipulés par l'utilisateur ;
- un *mapping* M_{CS_L} entre la syntaxe abstraite et chaque syntaxe concrète ;
- un ou plusieurs domaines sémantiques SD_L ;
- un *mapping* M_{SD_L} entre une syntaxe et chaque domaine sémantique.

Différents métalangages ont été proposés afin de spécifier les différents éléments des langages de modélisation dédiés. Chaque mogram de ces métalangages est donc une partie d'un métamodèle.

3.1.2 Syntaxe abstraite des langages de modélisation dédiés

L'ingénierie dirigée par les modèles place la syntaxe abstraite au centre de la spécification d'un langage de modélisation dédié. Les concepts exprimés au sein de la syntaxe abstraite du langage sont les concepts du domaine auquel le langage est dédié et sur lesquels s'appuie le reste de la spécification du langage.

Les métalangages permettant de définir la syntaxe abstraite d'un langage de modélisation dédié (e.g., Meta-Object Facility (MOF) [OMG06], de l'Object Management Group (OMG), ou Ecore, intégré à l'Eclipse Modeling Framework [SBPM09]) reposent sur les principes de la programmation orientée-objet. Ces métalangages orientés-objet permettent de représenter les concepts de la syntaxe abstraite au travers de classes contenues par des packages et reliées par différentes relations (associations, compositions ou spécialisations). Chaque classe représente un concept du langage de modélisation dédié, c'est à dire un concept du domaine pour lequel le langage est conçu.

L'Essential Meta-Objet Facility (EMOF) définit le cœur du métalangage MOF. EMOF permet de définir la syntaxe abstraite d'un langage sous la forme d'un graphe de classes connexes organisées en paquets. Un modèle est donc un ensemble d'objets qui sont des instances des classes contenues par la syntaxe abstraite du langage de modélisation dédié du modèle.

La figure 3.1 présente la structure d'EMOF sous la forme d'un diagramme de classes. EMOF permet de définir la syntaxe abstraite d'un langage de modélisation dédié à l'aide de classes (`Class`). Une `Class` peut être abstraite (i.e., elle ne peut pas être instanciée) et contenir des `Property` et des `Operation` qui déclarent respectivement les champs et les signatures des opérations disponibles à partir du concept modélisé par la `Class`. Une `Class` peut avoir plusieurs super-classes, dont elle hérite toutes les `Property` et `Operation`.

Un champ (`Property`) peut être composite (un objet ne peut être référencé que par une seule `Property` composite à un instant donné), dérivé (i.e., calculé à partir d'autres `Property` ou d'`Operation`) et en lecture seule (i.e., ne peut pas être modifié). Un champ peut aussi avoir un champ opposé avec lequel il forme une association bidirectionnelle.

Une `Operation` déclare le `Type` des exceptions qu'elle peut soulever et possède des `Parameter` ordonnés, qui représente les paramètres de l'opération.

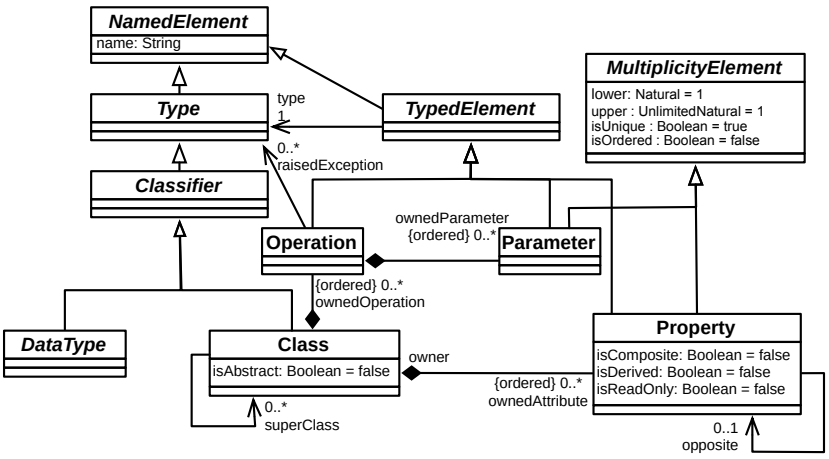

FIGURE 3.1 – Concepts au cœur d'EMOF sous la forme d'un diagramme de classes

Les `Property`, `Operation` et `Parameter` sont des `TypedElement` ; leur type peut être : un `DataType` (e.g., `Boolean`, `String`, etc.) ou une `Class`. Les `Property` et `Parameter` sont des `MultiplicityElement`. À ce titre, ils ont une multiplicité (définie par une borne inférieure et une borne supérieure qui stipulent respectivement le nombre minimal et le nombre maximal d'objets qui peuvent être passés comme arguments ou référencés), peuvent être ordonnés (les objets passés en arguments ou les instances référencées sont toujours visités dans le même ordre) et uniques (un objet ne peut être passé comme argument ou référencé qu'une fois).

Le métalangage Ecore définit un ensemble de concepts pour la définition de la syntaxe abstraite très proche de l'EMOF, et l'Eclipse Modeling Framework supporte EMOF. La syntaxe abstraite d'un langage de modélisation dédié définie avec EMOF ou Ecore est donc un graphe dont les nœuds sont des classes MOF (ou leur équivalent Ecore) et dont les arcs sont des champs.

Définition 3.2. ***(Syntaxe abstraite d'un langage de modélisation dédié)*** La syntaxe abstraite AS_L d'un langage de modélisation dédié L est un graphe de classes MOF.

Il est à noter que bien que les classes d'une syntaxe abstraite définie à l'aide du MOF ou d'Ecore soient instanciables, elles ne possèdent pas de comportement. En effet les `Operation` de ces deux métalangages ne possèdent pas de corps, mais uniquement une signature (nom, type de retour, paramètres, etc.). L'implémentation de ces opérations peut être réalisée à l'aide d'autres langages, e.g., Java (qui est alors utilisé comme un métalangage) ou Kermeta [1]. De telles implémentations font partie de la sémantique ou de l'outillage du langage de modélisation dédié (cf. Section 3.1.3), le domaine sémantique étant le langage utilisé pour décrire le corps des opérations et le mapping ayant lieu entre les signatures d'opérations de la syntaxe abstraite et le corps des opérations correspondantes dans le domaine sémantique.

La figure 3.2 présente la syntaxe abstraite d'un langage de graphes de flot de contrôle simple de deux manières différentes : un diagramme de classes et un diagramme d'objets. Le diagramme de classes est la représentation la plus courante d'une syntaxe abstraite définie à l'aide du MOF. Les concepts du langage sont représentés par des classes (`ControlFlowGraph` et `Node`) et leurs relations sont représentées par des champs : un `ControlFlowGraph` contient un

1. http://www.kermeta.org/

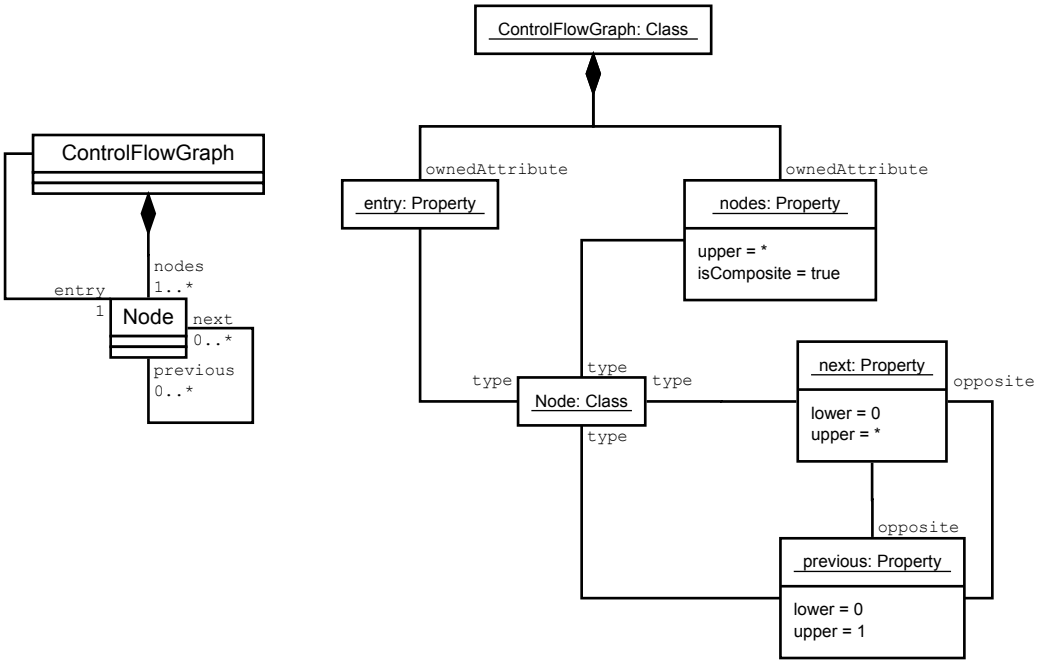

FIGURE 3.2 – Syntaxe abstraite d'un langage de graphe de flot de contrôle définie à l'aide d'EMOF, sous la forme d'un diagramme de classes et d'un diagramme d'objets.

ensemble de `Node` dont l'un est le point d'entrée (`entry`) du graphe ; les `Node` peuvent être reliés par des arcs (`next`) qui définissent les sauts entre deux ensembles d'instructions séquentiels. Le diagramme d'objets est valide vis-à-vis du diagramme de classe d'EMOF donné en Figure 3.1 [2]. Ainsi, on peut voir que chacune des deux classes de la syntaxe abstraite proposée sont instances de la classe `Class` du MOF, et que les champs entre ces classes sont instances de la classe `Property` du MOF.

La communauté de l'ingénierie dirigée par les modèles a développé de nombreux outils et métalangages pour définir la syntaxe concrète et la sémantique d'un langage de modélisation dédié à partir de sa syntaxe abstraite. En effet, il est possible de générer une syntaxe concrète ou le mapping entre la syntaxe abstraite et une syntaxe concrète à partir de la syntaxe abstraite du langage. De plus, la définition de la sémantique d'un langage de modélisation dédié s'appuie sur les classes de la syntaxe abstraite du langage. Cet outillage des langages de modélisation dédiés se fait généralement à l'aide de transformations de modèles, définies par rapport à la syntaxe abstraite du langage.

3.1.3 Outillage des langages de modélisation dédiés : transformations de modèles

Définir le graphe de classe qui forme la syntaxe abstraite d'un langage de modélisation dédié n'est pas suffisant pour obtenir un langage opérationnel. Il est possible de créer des modèles à partir de la syntaxe abstraite du langage de modélisation dédié en instanciant ses classes pour obtenir des éléments de modèles. Mais ces modèles sont au mieux descriptifs. Ils

2. Les attributs de chaque objet ne sont indiqués que lorsqu'ils n'ont pas la valeur par défaut définie dans le diagramme de classe.

ne possèdent ni comportement (i.e., ni sémantique, ni outillage), ni syntaxe concrète.

3.1.3.1 Transformations de modèles

Afin de compléter la spécification d'un langage de modélisation dédié, l'ingénierie dirigée par les modèles s'appuie sur les transformations de modèles. Les transformations de modèles regroupent tous les processus automatisés manipulant des modèles [SK03]. Elles permettent d'outiller les langages de modélisation dédiés. Les transformations de modèles peuvent fournir des outils de génération (de code, de documentation, de tests, etc.), de visualisation, de simulation, de vérification, d'analyse, de refactoring, etc. La spécificité des transformations de modèles est de traiter des mograms d'un langage de modélisation dédié. Nous définissons donc une transformation de modèles comme une fonction dont au moins un des paramètres est un modèle, ou qui retourne au moins un modèle (cf. Définition 3.3).

Une transformation qui accepte un modèle en paramètre peut être un générateur de code, qui parcourt le modèle et génère un code exécutable pour chacun de ses éléments. Une transformation retournant un modèle peut être un analyseur syntaxique qui traite la forme concrète textuelle d'un modèle pour retourner sa forme abstraite. Enfin, un exemple de transformation qui accepte un modèle en paramètre et retourne un modèle peut être un refactoring qui modifie le modèle afin qu'il respecte de bonnes propriétés, ou une transformation dans un autre langage de modélisation dédié, e.g., un langage plus approprié pour appliquer des outils de vérification.

Définition 3.3. ***(Transformation de modèles)*** Une transformation de modèles est une fonction qui prend en paramètre un ou plusieurs modèles, ou qui retourne un ou plusieurs modèles.

Il existe de nombreuses catégories de transformations de modèles : retournant un modèle, du code ou du texte ; acceptant en entrée un modèle, du code ou du texte ; endogènes (i.e., retournant un mogram du même langage que le mogram d'entrée), exogènes (i.e., retournant un mogram d'un langage différent du mogram d'entrée), etc [MVG06]. Quelles qu'elles soient, les transformations de modèles peuvent être implémentées à l'aide de nombreux langages et de nombreuses manières. Il est possible de définir une transformation de modèles en utilisant un langage de programmation généraliste, tel que Java. Il existe également de nombreux langages dédiés à la définition de transformation de modèles [CH06].

3.1.3.2 Métamodèle = Structure + Comportement

Le terme transformation de modèles regroupe donc un ensemble d'outils très large. Parmi ces outils, on peut trouver les outils qui définissent les différents mappings d'un métamodèle. En effet, un mapping entre la syntaxe abstraite et une syntaxe concrète peut être vu comme une paire de transformations de modèles, l'une prenant en paramètre un modèle et retournant la forme concrète de ce modèle (e.g., un pretty-printer), et l'autre prenant en paramètre la forme concrète d'un modèle et retournant sa forme abstraite (e.g., un analyseur syntaxique). De même, le mapping entre la syntaxe abstraite et un domaine sémantique peut être implémenté au travers d'une transformation de modèles acceptant un modèle en entrée et retournant l'ensemble des concepts correspondants du domaine sémantique (e.g., un générateur de code).

Ceci nous amène à donner une autre définition d'un métamodèle, comme la somme d'une structure et d'un comportement (cf. Définition 3.4). Il est en effet possible de concevoir la spécification d'un langage de modélisation dédié comme une structure accompagnée d'un ensemble de transformations de modèles. La structure du langage dédié est alors définie sous la forme d'un graphe de classes MOF et les transformations de modèles fournissent le comportement du langage. Ces transformations peuvent aussi bien définir la sémantique du

langage (e.g., sous la forme d'un compilateur ou d'un interpréteur) que sa (ou ses) syntaxe concrète (e.g., en implémentant un analyseur syntaxique).

Définition 3.4. ***(Métamodèle d'un langage de modélisation dédié (2))*** Le métamodèle $MM_L = \langle Str_L, \mathbf{Trans_L} \rangle$ d'un langage de modélisation dédié L est constitué de :

- une structure Str_L définie sous la forme d'un ensemble de classes MOF reliées par des associations ;
- un ensemble de transformations de modèles $\mathbf{Trans_L}$.

La Définition 3.4 n'entre pas en contradiction avec la Définition 3.1 donnée plus haut. Ces deux définitions offrent chacune une vision différente sur la spécification d'un langage de modélisation dédié.

3.2 Validité d'un modèle par rapport à la spécification d'un langage de modélisation dédié : relation de conformité

Pour déterminer si un modèle est un mogram valide d'un langage de modélisation dédié, l'ingénierie dirigée par les modèles utilise la relation de conformité entre un modèle et un métamodèle. Un modèle conforme au métamodèle d'un langage de modélisation dédié est un modèle valide de ce langage. La relation de conformité s'établit en comparant les éléments du modèle (des objets) aux éléments de la syntaxe abstraite du métamodèle (des classes) et s'appuie donc sur la relation d'instanciation.

3.2.1 La relation de conformité dans la littérature

On retrouve dans la littérature plusieurs noms donnés à cette relation : "sem", "instanciation", "conformité", et "compliance". Toutes ces relations reposent directement sur la syntaxe abstraite des langages de modélisation dédiés. Dans cette section, nous passons en revue les définitions de ces relations. Par la suite, nous n'utiliserons que le terme conformité pour désigner cette relation entre les modèles et la spécification des langages de modélisation dédiés. Le tableau 3.2 présente différentes définitions données au cours des dix dernières années à la relation de conformité dans la littérature.

Favre considère n'importe quelle représentation d'un langage, sous quelque forme que ce soit, comme un métamodèle et établit la relation de conformité entre un modèle et n'importe laquelle de ces représentations (e.g., syntaxe abstraite ou concrète, documentation). Cette définition est donc plus large et moins stricte que les autres définitions présentées ici. Cependant, si elle est suffisante pour l'étude et la compréhension de l'ingénierie dirigée par les modèles (son objectif premier), elle n'est pas assez précise pour des fins d'outillage et donc de vérification automatique de la validité d'un modèle.

À cette exception près, les définitions de la table 3.2 convergent sur deux points : la relation de conformité s'appuie sur la relation d'instanciation entre objets et classes, et un modèle est conforme à un et un seul métamodèle.

TABLE 3.2 – Définitions de la *relation de conformité* dans la littérature

Bezivin [BG01]	"Let us consider model X containing entities a and b. There exists one (and only one) meta-model Y defining the "semantics" of X. The relationship between a model and its meta-model (or between a meta-model and its meta-meta-model) is called the sem relationship. The significance of the sem relationship is as follows. All entities of model X find their definition in meta-model Y."
Atkinson et Kühne [AK02]	"In an n-level modeling architecture, M0, M1...Mn-1, every element of an Mm-level model must be an instance-of exactly one element of an Mm+1-level model, for all m < n - 1, and any relationship other than the instance-of relationship between two elements X and Y implies that level(X)=level(Y)."
Favre [Fav04a]	Favre ne donne pas une définition de la relation de conformité, mais la présente comme un raccourci de la succession de deux autres relations : "elementOf" (qui lie un élément d'un langage à ce langage) et "representationOf" (qui lie un métamodèle à un langage de modélisation).
Bezivin *et al.* [BJT05]	"A model M conforms to a metamodel MM if and only if each model element has its metaelement defined in MM."
Găsevic *et al.* [GKH07]	"metamodels and models are connected by the instanceOf relation meaning that a metamodel element (*e.g.*, the Class metaclass from the UML metamodel) is instantiated at the model level (*e.g.*, a UML class Collie)."
Rose *et al.* [RKPP10]	"A model conforms to a metamodel when the metamodel specifies every concept used in the model definition, and the model uses the metamodel concepts according to the rules specified by the metamodel. [...] For example, a conformance constraint might state that every object in the model has a corresponding non-abstract class in the metamodel."
Egea et Rusu [ER10]	"Namely, the objects of a "conformant" model are necessarily instances of the classes of the associated metamodel (possibly) related by instances of associations between the metamodel's classes."
Jézéquel *et al.* [JCV12]	"Un modèle est dit conforme à un métamodèle si chacun de ses éléments (objets et relations) est instance d'un élément du métamodèle, et s'il respecte l'ensemble des propriétés (*e.g.*, contraintes d'invariant) exprimées sur le métamodèle."

La relation de conformité est liée à la relation d'instanciation. En effet de nombreux auteurs définissent la relation de conformité en s'appuyant sur la relation d'instanciation qui lie un objet et sa classe : "every element of an Mm-level model must be an instance-of exactly one element of an Mm+1-level model" [AK02], "metamodels and models are connected by the instanceOf relation" [GKH07], "every object in the model has a corresponding non-abstract class in the metamodel" [ER10], "Un modèle est dit conforme à un métamodèle si chacun de ses éléments (objets et relations) est instance d'un élément du métamodèle" [JCV12].

Bézivin et al. ne font pas directement référence à la notion de classe mais de "définition" [BG01] ou de "méta-élement" [BJT05] (i.e., élément de métamodèle), autorisant la définition de la syntaxe abstraite d'un langage autrement que sous forme d'un graphe de classes.

Un modèle est conforme à un et un seul métamodèle. Ce point est une conséquence du précédent. En effet, un objet n'est instance que d'une classe, qui définit ses champs et ses opérations. Une classe ne peut appartenir qu'à un seul package (par la sémantique de la composition) et donc à un seul métamodèle. Ainsi, il existe un et un seul métamodèle auquel un modèle se conforme, celui qui définit toutes les classes dont sont instances les éléments du modèle ("There exists one (and only one) meta-model Y defining the "semantics" of X" [BG01]). La relation de conformité interdit donc toute forme de polymorphisme au niveau des modèles, puisqu'un modèle ne peut avoir qu'une seule "forme", celle définie par son unique métamodèle (cf. Section 3.2.4).

3.2.2 La relation de conformité dans les standards *de facto*

Le Meta-Object Facility (MOF) de l'Object Management Group (OMG) et l'Eclipse Modeling Framework (EMF) sont de facto des standards technologiques dans la communauté de l'ingénierie dirigée par les modèles. De nombreux outils et environnements de métamodélisation s'appuient sur ces standards (MOFLON [3], Kermeta, ATL [4], Henshin [5], Epsilon [6], XText [7], XTend [8]) pour outiller les langages de modélisation dédiés. Nous nous intéressons donc à la façon donc ces standards définissent la relation liant modèles et métamodèles.

Le Meta-Object Facility (MOF) , décrit dans la spécification du même nom issue de l'Object Management Group OMG) [OMG06] est un langage pour la définition de la syntaxe abstraite des langages de modélisation dédiés. Cependant, la spécification MOF ne donne aucune indication sur la relation qui lie un modèle et un métamodèle. Les métamodèles sont dits instances du MOF, mais cette relation d'instanciation n'est jamais clairement définie. La spécification MOF et la spécification UML étant fortement interdépendantes, nous pouvons également nous intéresser à la spécification UML. Dans la spécification UML, UML est dit être une instance du MOF et "chaque élément de modèle UML est une instance d'exactement un élément de modèle du MOF" [9] [OMG11, article 7]. Toutefois, cela ne donne pas une définition claire et générale de la relation qui se dresse entre le MOF et les métamodèles, ni entre les métamodèles et les modèles.

3. http://www.moflon.org/
4. http://www.eclipse.org/atl/
5. http://www.eclipse.org/henshin/
6. http://www.eclipse.org/epsilon/
7. http://www.eclipse.org/Xtext/
8. http://www.eclipse.org/xtend/
9. *"every model element of UML is an instance of exactly one model element in MOF"*

L'Eclipse Modeling Framework (EMF) [SBPM09] ne fournit pas non plus une telle définition, mais s'appuie sur deux technologies pour manipuler des modèles : les classes Java pour l'instanciation d'éléments de modèles et les schémas XML pour la sérialisation de modèles. Côté Java, une classe est générée pour chaque concept de la syntaxe abstraite et les modèles sont des ensembles d'objets instances de ces classes générées. Côté XML, les schémas XML décrivent la structure des métamodèles, permettant la sérialisation des modèles en tant que documents XML. La norme des schémas XML stipule que "la vérification de la conformité peut être considérée comme un procédé par étapes : vérifier que le contenu de l'élément racine est correct, puis vérifier que chaque sous-élément est conforme à sa description dans un schéma et ainsi de suite jusqu'à ce que la totalité du document soit vérifiée."[10] [W3Cb, partie 5.7]. Par ailleurs, "Pour vérifier la conformité d'un élément, l'outil localise d'abord la déclaration de l'élément dans un schéma."[11] [W3Cb, partie 5.7]. Ainsi, un élément d'un document XML sans déclaration correspondante dans un schéma XML n'est pas conforme, et l'ensemble du document non plus. Les métamodèles étant décrits par des schémas XML et les modèles par des documents XML, un modèle n'est conforme par rapport à un métamodèle donné que si tous ses éléments décrits dans le document XML sont déclarés dans le schéma XML du métamodèle.

3.2.3 Définition de la relation de conformité

À l'exception de celle de Favre, toutes les définitions de la littérature convergent vers une même définition : un modèle est conforme à un métamodèle si chacun des éléments du modèle est instance de l'un des éléments du métamodèle, ou de la syntaxe abstraite du langage. De même, dans le cadre de l'Eclipse Modeling Framework, un modèle (que ce soit sous la forme d'ensemble d'objets Java ou de fichier XML) ne peut pas contenir d'éléments qui ne sont pas instances des classes du métamodèle ou ne sont pas décrits par le schéma XML. Enfin, la spécification du Meta-Object Facility [OMG06] ne donne pas de définition claire de la relation de conformité, mais suggère une relation d'instanciation entre métamodèles et modèles. À partir de l'étude de la littérature et des standards technologiques et de la définition de métamodèle (définition 3.1), nous définissons donc la relation de conformité comme suit :

Définition 3.5. ***(Relation de conformité)*** Un modèle m est conforme à un métamodèle MM_L si et seulement si, $\forall o \in m$, o est une instance de la classe C telle que $C \in AS_L$.

3.2.4 Limites de la relation de conformité

La relation de conformité, qui lie un modèle à un métamodèle, permet de déterminer si un modèle est un mogram valide d'un langage de modélisation dédié. C'est donc un concept central de l'ingénierie dirigée par les modèles, qui est utilisé comme une relation de typage pour autoriser ou non le passage d'un modèle à une transformation de modèles, ou le chaînage de transformations de modèles. D'après la Définition 3.5, il n'existe qu'un métamodèle MM_L auquel se conforme un modèle m donné. Ce métamodèle est celui dont la syntaxe abstraite (AS_L) contient toutes les classes dont sont instances les différents éléments de m.

En interdisant à un modèle d'être lié à plusieurs métamodèles, la relation de conformité freine la réutilisation. En effet, puisqu'un métamodèle est utilisé pour "typer" les paramètres d'une transformation de modèles, et qu'un modèle n'est conforme qu'à un métamodèle, il n'est

10. *"Conformance (i.e., validity) checking can be thought of as proceeding in steps, first checking that the root element of the document instance has the right contents, then checking that each subelement conforms to its description in a schema, and so on until the entire document is verified."*
11. *"To check an element for conformance, the processor first locates the declaration for the element in a schema."*

pas possible de substituer un modèle conforme à un métamodèle *MM*' à un modèle conforme à un métamodèle *MM* attendu par une transformation de modèles *T*. La transformation *T* n'accepte que des modèles conformes à *MM* et n'est donc pas réutilisable entre plusieurs langages de modélisation dédiés.

La syntaxe abstraite d'un langage de graphes de flot de contrôle présentée en Figure 3.2 est un graphe de classe très simple (il ne contient que deux classes et quatre champs). Cependant, il est déjà possible de définir plusieurs outils à partir d'une représentation aussi simple :

- un éditeur (textuel ou graphique), permettant de visualiser et de modifier un graphe de flot de contrôle donné ;
- un outil permettant de sauvegarder et de charger des graphes de flot de contrôle (e.g., sous la forme d'un fichier .xmi) ;
- un outil d'analyse d'accessibilité, permettant de détecter les blocs de base inaccessibles dans un graphe de flot de contrôle donné ;
- un outil de détection des boucles à l'aide des arcs en arrière (back edge en anglais) dont la cible se trouve plus haut dans le graphe que la source ;
- un outil d'analyse de domination : un bloc de base b_1 domine un autre bloc b_2 si tous les chemins du bloc d'entrée à b_2 doivent passer par b_1. Cette propriété est utile pour un certain nombre d'analyses et d'optimisations (e.g., le passage en forme Static Single Assignment) [AP03, Chap. 18].

Cependant, les compilateurs utilisent des graphes de flot de contrôle bien plus complexes comme représentations intermédiaires, afin de véhiculer plus d'informations sur le programme. En effet, les nœuds, ou blocs, des graphes de flot de contrôle contiennent généralement des séquences d'instructions contenant des expressions qui peuvent elles-mêmes contenir ou avoir besoin de plus d'informations (symboles utilisés dans le programme, types, etc.). Certaines de ces instructions sont spécifiques à certains langages de programmation. Plusieurs compilateurs ne peuvent donc pas partager la même représentation des graphes de flot de contrôle.

La modélisation de tels graphes de flot de contrôle à l'aide de métamodèles mènera à plusieurs métamodèles, chacun contenant des informations spécifiques à un compilateur ou à un langage de programmation. Par définition (cf. Définition 3.5), un modèle conforme à l'un de ces métamodèles ne sera pas conforme à l'un des autres métamodèles, ni à celui dont la syntaxe abstraite est présentée en Figure 3.2. Ainsi, il devient impossible de réutiliser l'un des outils mentionnés ci-dessus sur un modèle conforme à l'un de nos métamodèles de graphe de flot de contrôle complexes, bien qu'ils partagent la même structure de base (un `ControlFlowGraph` composé de `Nodes` connectés).

Sans possibilité de réutilisation, les analyses et optimisations seront réécrites à partir de zéro pour chaque nouvelle représentation intermédiaire utilisant les graphes de flot de contrôle. De plus, l'impossibilité de réutiliser ces différents outils empêche également les concepteurs de compilateurs de les définir au niveau d'abstraction approprié. Plutôt que de définir l'analyse de domination sur un graphe de classes simple comme celui de la Figure 3.2, les concepteurs vont définir cette analyse sur un graphe complexe, contenant toutes les informations nécessaires pour toutes les analyses et optimisations du compilateur. À titre d'exemple, la représentation intermédiaire du compilateur GeCoS [12] est un métamodèle composé de 122 classes et plus de 1500 éléments. Un tel métamodèle est difficile à maintenir et à faire évoluer et rend la définition de nouvelles analyses et optimisations difficiles en raison de la surabondance d'informations.

12. http://gecos.gforge.inria.fr

La relation de conformité, en empêchant de réutiliser les transformations de modèles entre différents langages de modélisation dédiés, entrave donc la flexibilité de conception des langages de modélisation dédiés, conduisant à la création de métamodèles monolithiques dont on doit concevoir tous les outils de zéro, sans possibilité de réutiliser les outils existants.

3.3 Facilités d'ingénierie pour les langages de modélisation dédiés

Le choix de développer un langage de modélisation dédié répond au même compromis que celui de développer n'importe quel langage dédié : bénéfices apportés par rapport aux coûts de développement et d'apprentissage. L'ingénierie dirigée par les modèles fournit des outils pour faciliter la conception de langages de modélisation dédiés. Cependant, ces langages sont le plus souvent encore définis et outillés à partir de zéro. Afin de supporter le développement d'un nombre croissant de langages de modélisation dédiés [HWRK11] et de diminuer les coûts de définition et d'outillage de ces langages, différentes facilités d'ingénierie ont été proposées. Ces facilités incluent la définition de mégamodèles permettant de modéliser les langages de modélisation dédiés, les modèles et leurs relations et la réutilisation des transformations de modèles, malgré les limites de la relation de conformité.

3.3.1 Modélisation *in-the-large*

De nombreuses tâches de l'ingénierie dirigée par les modèles impliquent de manipuler plusieurs modèles issus de différents langages de modélisation dédiés, mais également de raisonner sur les relations entre ces modèles. Ces relations entre modèles peuvent être la relation de conformité entre un modèle et son métamodèle (lui-même étant un modèle), la transformation d'un ou plusieurs modèles en un ou plusieurs autres, les relations de cohérence ou de synchronisation à assurer entre plusieurs modèles, les relations de traçabilité entre les différents modèles créés au sein d'une chaîne de transformations de modèles, etc.

Malgré la volonté de l'ingénierie dirigée par les modèles de faire des modèles des entités de première classe, placées au centre du développement logiciel, rares sont les langages et les environnements de modélisation qui offrent cette possibilité. En effet, même au sein de standards de facto comme le MOF ou Ecore, ni la notion de modèle, ni celle de métamodèle n'apparaît. Dans ces métalangages, et dans ceux qui s'appuient sur eux, un métamodèle n'est défini et manipulable qu'au travers d'un ensemble de packages. De même, il est plus courant de passer en paramètre d'une transformation de modèles un élément de modèle (souvent l'élément racine), à partir duquel la transformation va accéder aux autres éléments, qu'un modèle.

3.3.1.1 Définition de la modélisation *in-the-large*

Bézivin et al. définissent la modélisation in-the-large comme le fait "d'établir et d'utiliser des relations au niveau des entités macroscopiques de la modélisation : modèles et métamodèles en ignorant les détails internes de ces entités" [BJRV05]. Ainsi, il est possible de fournir une vision globale sur un ensemble de modèles liés par différentes relations, à un niveau d'abstraction plus haut que celui des éléments de modèles et de leurs relations (i.e., des objets et de leurs champs). Cette notion de modélisation in-the-large est très liée à celle de modèle comme entité de première classe. En effet pour définir des relations au niveau des modèles en s'abstrayant de leurs détails internes (i.e., de leurs éléments), il est nécessaire de pouvoir appréhender les modèles directement, et pas uniquement au travers des objets qui les composent.

La modélisation in-the-large s'appuie sur la modélisation in-the-small, i.e., sur les activités de modélisation au niveau des éléments de modèles. En effet, les modèles sont composés d'éléments de modèles, et les relations entre modèles s'appuient généralement sur des relations entre éléments de modèles. Une transformation d'un modèle vers un autre établit une relation entre les deux modèles, mais s'appuie sur la façon dont des ensembles d'éléments d'un modèle sont utilisés pour créer des ensembles d'éléments de l'autre. De la même manière, établir une relation indiquant que deux modèles doivent rester cohérents l'un par rapport à l'autre nécessite de préciser quels éléments des ces modèles doivent rester cohérents.

La modélisation in-the-large (cf. Définition 3.7) est donc une activité dépendante de la modélisation in-the-small. À l'inverse, la modélisation in-the-small (cf. Définition 3.6) ne dépend pas de la modélisation in-the-large. En effet, manipuler les objets et leurs relations ne nécessite pas pas de connaître les modèles auxquels ils appartiennent.

Définition 3.6. ***(Modélisation* in-the-small*)*** La modélisation *in-the-small* regroupe les activités de modélisation au niveau des éléments de modèles et de leurs relations.

Définition 3.7. ***(Modélisation* in-the-large*)*** La modélisation *in-the-large* regroupe les activités de modélisation au niveau des modèles et de leurs relations.

3.3.1.2 Mégamodèles

Il existe plusieurs approches fournissant des facilités de modélisation in-the-large, souvent désignées sous le nom de mégamodèles. La notion de mégamodèle a été introduite par Bézivin pour désigner les modèles dont les éléments sont eux-mêmes des modèles [BGMR03]. Cette définition a ensuite été étendue pour inclure les relations entre modèles [BDFB07]. Là où un modèle "classique" (ou in-the-small) est composé d'objets et de leurs relations, un mégamodèle est donc composé de modèles et de leurs relations. Puisque un métamodèle est un modèle, il est également possible d'exprimer au sein d'un mégamodèle des relations entre métamodèles, et donc de modéliser des relations entre langages de modélisation dédiés à l'aide d'un mégamodèle.

De nombreuses approches s'intéressent aux mégamodèles [BGMR03, Fav04b, Fav04a, BDFB07, GKH07, SNG10], parfois sous un autre nom : macromodèles [SME09] ou intermodélisation (intermodelling en anglais) [GdLKP10] [13].

Favre [Fav04b, Fav04a] et Gasević et al. [GKH07] proposent un mégamodèle qui est un modèle de l'ingénierie dirigée par les modèles. Ces composants sont donc les concepts de modèle, de système et de langage, et les relations qui unissent ces concepts sont les relations de représentation (un modèle représente un système, un métamodèle représente un langage), d'appartenance (un modèle appartient à un langage) et de conformité (un modèle se conforme à un métamodèle). Ce mégamodèle a pour but l'étude théorique de l'ingénierie dirigée par les modèles. Il ne s'agit donc pas de fournir un langage de mégamodélisation permettant de définir des relations entre modèles.

À l'inverse, Barbero et al., Salay et al., Seibel et al. et Guerra et al. proposent des approches permettant de définir des mégamodèles, i.e., des modèles contenant des modèles et des relations entre ces modèles. Certaines de ces approches sont centrées sur une relation donnée, comme la traçabilité [BDFB07, SNG10] tandis que d'autres proposent un cadre plus large [SME09, GdLKP10]. Les relations entre modèles sont définies au niveau des métamodèles, de la même manière que les relations entre objets sont définies sous la forme de champs au niveau des classes.

13. pour une revue plus détaillée de certaines de ces approches, nous renvoyons le lecteur à l'article d'Hebig *et al.* [HSG10]

Guerra et al. proposent de définir des relations génériques entre deux modèles à l'aide de triple graphs. Ces relations génériques sont ensuite utilisées pour générer des outils spécifiques. Chaque triple graph est composé des deux métamodèles impliqués dans la relation (chacun sous la forme d'un graphe) et d'un graphe liant les éléments de ces métamodèles. À partir d'une même spécification s'appuyant sur les triple graphs, Guerra et al. fournissent une approche pour dériver une transformation de modèles d'un métamodèle vers l'autre, un outil de comparaison de modèles et un outil de vérification de liens de traçabilité.

Salay et al. proposent de définir des types de relations entre modèle sous la forme de métamodèles [SME09]. Ces métamodèles sont l'union des métamodèles concernés par la relation, plus des relations entre leurs éléments. Par exemple, le type de relation objectOf est défini entre le métamodèle des diagrammes d'objets et le métamodèle des diagrammes de classes. Le métamodèle représentant ce type de relation est donc l'union de ces deux métamodèles, plus des relations liant leurs éléments. Une relation entre deux modèles est donc un modèle conforme au "métamodèle union".

3.3.2 Réutilisation de transformations de modèles

Les transformations de modèles définissent le comportement d'un langage de modélisation dédié. Elles peuvent implémenter des manipulations de modèles complexes et difficiles à écrire, telles qu'un compilateur. Il n'est donc pas surprenant que de nombreuses approches aient été proposées afin de réutiliser les transformations de modèles. Certaines approches s'appuient sur la définition de transformations de modèles génériques, i.e., de transformations acceptant des modèles conformes à différents métamodèles. D'autres approches composent ou chaînent des transformations de modèles atomiques afin de créer des transformations plus complexes. Les transformations de modèles atomiques peuvent donc être réutilisées au sein de différentes chaînes.

3.3.2.1 Transformations de modèles génériques

Les transformations de modèles génériques sont des transformations de modèles qui acceptent ou retournent des modèles issus de plusieurs langages de modélisation dédiés. Pour cela, elles sont définies sur des entités différentes des métamodèles. En effet, puisqu'un modèle ne se conforme qu'à un unique métamodèle, définir les transformations sur un métamodèle les empêchent d'accepter des modèles conformes à d'autres métamodèles. Une transformation de modèles qui est définie à partir des classes d'un métamodèle ne peut manipuler que des objets instances de ces classes. Dès lors, un modèle conforme à un autre métamodèle ne peut pas être manipulé par la transformation, puisque les objets qu'il contient sont instances d'autres classes.

Pour appliquer une transformation générique définie sur une de ces entités (motifs de transformations contenant des entités variables, templates ou concepts), il faut établir une relation entre le métamodèle du modèle et ladite entité. Cette relation s'appuie sur une comparaison des structures du métamodèle et de l'entité à partir de laquelle la transformation a été définie. Ainsi, une transformation générique s'établit non pas sur un langage de modélisation dédié, mais sur une famille de langages, établie à l'aide de cette comparaison de structures.

Varrò et Pataricza ont introduit des entités variables (variable entities en anglais) dans les motifs (patterns en anglais) de règles de transformation déclaratives [VP04]. Le motif d'une règle de transformation peut être exprimé en faisant référence aux éléments d'un métamodèle ou en utilisant des entités variables qui spécifient uniquement les concepts nécessaires pour appliquer la règle (i.e., types, champs, etc). La reconnaissance de motifs (pattern matching en

anglais) peut ensuite sélectionner un modèle présentant ces concepts, indépendamment de son métamodèle.

Cuccuru et al. ont proposé la notion de points de variation sémantique dans les métamodèles [CMTG07]. Un métamodèle peut contenir des points de variation spécifiés par des classes abstraites, il définit alors un template. Une transformation de modèles écrite sur un template peut-être réutilisée sur les modèles d'autres métamodèles. Pour cela, les métamodèles doivent fixer les points de variation en les liant à des classes qui étendent les classes abstraites. Les métamodèles créés à partir d'un template forment une famille de langages de modélisation dédiés partageant une partie de leur structure et le comportement défini par les transformations écrites à partir du template.

De Lara et Guerra introduisent le mécanisme de concept inspiré de la programmation générique [DLG10]. Les concepts définissent les exigences que doit remplir un métamodèle, sous la forme d'un ensemble de classes, pour réutiliser une transformation. À partir d'une transformation définie sur un concept et d'une correspondance entre le concept et un métamodèle, établie à l'aide d'un langage dédié, une transformation spécifique à ce métamodèle est générée.

Ces approches s'appuient sur une correspondance structurelle forte entre le métamodèle et l'entité sur laquelle est définie la transformation. En effet, le métamodèle doit contenir un sous-graphe isomorphe [Coo71] de l'entité, i.e., un sous-ensemble du graphe de classes de sa syntaxe abstraite doit être identique (même noms, multiplicités, etc.) au graphe de classes défini par le motif à entités variables, le template ou le concept sur lequel est définie la transformation.

C'est pourquoi Sanchez et al. et Wimmer et al. ont étendu le mécanisme de liaison présentée pour les concepts [DLG10] pour autoriser des adaptations [SCGdL11, WKR+11]. Ces adaptations incluent le renommage, la liaison de plusieurs éléments d'un métamodèle à un seul élément d'un concept et la navigation et le filtrage de champs en utilisant des expressions de l'Object Contrainst Language (OCL) [OMG03]. Ces adaptations sont possibles grâce à une approche qui mêle adaptation des modèles à transformer et génération de transformations spécifiques pour leurs métamodèles.

3.3.2.2 Décomposition et chaînage de transformations de modèles

Un processus de développement dirigé par les modèles complet peut impliquer de nombreuses transformations de modèles. De plus, certaines transformations de modèles sont suffisamment complexes pour qu'il soit utile de les décomposer en plusieurs transformations de complexité moindre. Pour supporter cette modularisation des transformations de modèles, plusieurs approches ont été proposées pour définir des chaînes de transformations de modèles.

Les transformations atomiques peuvent être chaînées au sein de transformations composites (ou complexes) [Old05, VJBB13] ou à l'aide d'un langage d'orchestration [Kle06, PVSGB08, RRGLR+09]. Pour déterminer si deux transformations peuvent être chaînées, les environnements d'orchestration ou de définition de transformations de modèles comparent les signatures des transformations. Ces signatures sont exprimées sous la forme $T : I_1...I_n \rightarrow O_1...O_n$, où $I_1...I_n$ représentent les types des modèles d'entrée de la transformation, et $O_1...O_n$ les types des modèles de sortie de la transformation. Ces types sont exprimés sous la forme des métamodèles auxquels se conforment les modèles. Par exemple une transformation $MM_A \rightarrow MM_B$ accepte en entrée un modèle conforme au métamodèle MM_A et retourne un modèle conforme au métamodèle MM_B.

Pilgrim et al. autorisent l'utilisateur à définir des pré et post-conditions aux transformations pour contraindre les modèles d'entrée et de sortie au-delà de la conformité [PVSGB08]. Aranega, Etien et al. ont proposé des analyses statiques, qui analysent le code des transforma-

tions afin de déduire quelles actions CRUD [14] (création, lecture, modification ou suppression) peuvent être réalisées par la transformation sur une classe donnée [AEM12, EABP12]. Ces informations permettent d'obtenir une signature plus précise des transformations de modèles en annotant les classes du métamodèle.

Pour chaîner deux transformations de modèles, celles-ci doivent avoir des signatures compatibles : l'exécution de la transformation $T^{fst} : I_1^{fst}...I_n^{fst} \rightarrow O_1^{fst}..O_n^{fst}$ peut être suivie de l'exécution de la transformation $T^{snd} : I_1^{snd}...I_n^{snd} \rightarrow O_1^{snd}...O_n^{snd}$ si et seulement si $O_1^{fst}...O_n^{fst} = I_1^{snd}...I_n^{snd}$. Deux transformations ne peuvent donc être chaînées que si le métamodèle (éventuellement étendu par des informations supplémentaires) du modèle de retour de la première est le même métamodèle que le métamodèle du paramètre de la seconde.

3.3.3 Limites des approches existantes

Les approches existantes fournissent des facilités pour la définition et l'outillage des langages dédiés, et pour la manipulation des modèles. Ainsi, il est possible de définir explicitement des relations entre modèles au sein de mégamodèles, afin de de visualiser et de manipuler des ensembles de modèles liés. Il existe également des approches permettant de réutiliser des transformations de modèles entre différents langages dédiés, ou au sein de transformations plus complexes.

Cependant, ces approches restent déconnectées les unes des autres et manquent d'un cadre unifié permettant à la fois leur combinaison et leur comparaison. En effet, ces approches utilisent différents formalismes et langages pour la définition de la syntaxe abstraite (e.g., Ecore, MOF, sous-ensemble du MOF, triple graphs) et pour l'expression des transformations de modèles réutilisables (e.g., langages de transformation déclaratifs ou impératifs). Certaines approches nécessitent également d'intégrer de nouvelles entités à l'ingénierie dirigée par les modèles (e.g., templates, concepts, relations sous la forme de triple graphs).

Cette disparité rend ces différentes approches difficilement combinables. Puisque les transformations de modèles génériques s'appuient sur des entités différentes des métamodèles, il n'est pas possible de les chaîner à l'aide d'approches qui comparent la signature des transformations en termes de métamodèles. De même, il n'est pas possible de chaîner des transformations déclarées à l'aide de relations explicites in-the-large au sein de transformations plus complexes. Enfin, utiliser ces relations explicites pour déclarer des transformations génériques n'est pas non plus possible, puisque les différents mégamodèles s'appuient sur les métamodèles.

La Figure 3.3 représente cette situation : les transformations de modèles génériques sont définies sur des entités spécifiques (concepts, templates, ou motifs à entités variables), les transformations qu'il est possible de composer sont définies sur les métamodèles et les mégamodèles contiennent les transformations réifiées par la modélisation in-the-large.

Puisqu'il n'existe pas de formalisme commun à ces approches, il est également difficile de les comparer. Pour connaître leurs caractéristiques et les comparer, voir les mesurer, afin, par exemple, de choisir une méthode plutôt qu'une autre, il faudrait pouvoir exprimer ces caractéristiques d'une manière unique.

En effet, si la définition de la syntaxe abstraite sous la forme d'un graphe de classes est un standard accepté dans la communauté de l'ingénierie dirigée par les modèles, il n'en va pas de même pour le reste de la spécification des langages de modélisation dédiés. Il n'existe donc pas de cadre unifié et bien défini qui permette la comparaison et la combinaison d'outils et de facilités tels que ceux présentés plus haut.

14. *Create, Read, Update, Delete* en anglais

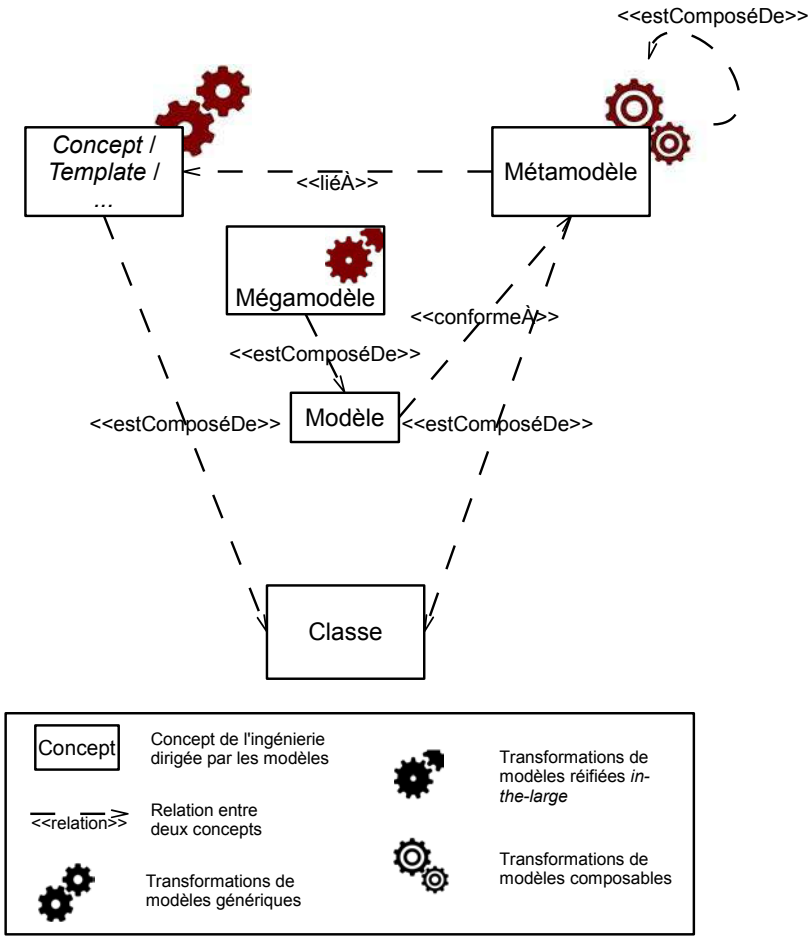

FIGURE 3.3 – Facilités pour la définition et l'outillage de langages de modélisation dédiés.

3.4 Conclusion

L'ingénierie dirigée par les modèles prône l'utilisation de multiples modèles dans la conception de systèmes logiciels. Chaque modèle est exprimé dans un langage de modélisation dédié à une préoccupation particulière, permettant la séparation des préoccupations. De plus, l'ingénierie dirigée par les modèles a développé une méthodologie et des outils pour faciliter la définition et l'outillage des langages de modélisation dédiés.

Cependant, un certain nombre de ces outils ont été conçus uniquement pour contourner des limites inhérentes à la spécification d'un langage de modélisation dédié et pourtant inutiles. En effet, la relation de conformité, considérée par certains comme un principe de base de l'ingénierie dirigée par les modèles [Béz06] et sur laquelle s'appuient les outils et les environnements de modélisation est trop contraignante.

En s'appuyant sur la relation d'instanciation entre un objet et sa classe, la relation de

conformité interdit à un modèle de se conformer à plus d'un métamodèle. Cependant, lors de la conception d'un langage de modélisation dédié, l'instanciation n'est pas la seule préoccupation. Les concepteurs et utilisateurs de langages de modélisation dédiés ont besoin de visualiser, d'éditer, de connecter, de vérifier et de modifier les modèles. Or, définir les transformations de modèles à partir du métamodèle du langage empêche la réutilisation des outils et pousse à la conception de langages de modélisation dédiés monolithiques.

Pour contourner les limites de la relation de conformité et faciliter la définition des langages de modélisation dédiés, de nombreuses approches ont été proposées. Cependant, ces approches restent déconnectées les unes des autres et manquent d'un cadre unifié qui permettrait de les comparer et de les combiner.

Dans le chapitre suivant (Chapitre 4), nous présentons les systèmes de types des langages de programmation orientés-objet. Ces systèmes unifient de nombreuses facilités au travers des relations de typage et de sous-typage, y compris la réutilisation par polymorphisme de sous-type. Nous présentons également les travaux existants pour le typage de modèles afin d'amener les facilités de typage au niveau des modèles et des langages de modélisation dédiés.

Chapitre 4

Typage de modèles

L'ingénierie dirigée par les modèles est intrinsèquement liée aux concepts du paradigme objet. En effet la spécification des langages de modélisation dédiés est centrée sur la définition de la syntaxe abstraite sous la forme d'un graphe de classes. Les systèmes de types fournissent un cadre formel supportant de nombreuses facilités au sein des langages de programmation orientés-objet. Ces facilités incluent notamment différents mécanismes de réutilisation, ainsi que la possibilité de manipuler les objets comme des entités de première classe (i.e., les affecter à une variable, les passer en paramètre ou les retourner à la fin d'une opération). C'est pourquoi nous étudions dans ce chapitre les mécanismes des systèmes de types orientés-objet et les facilités qu'ils supportent. Parmi ces mécanismes, certains peuvent permettre la substituabilité de modèles par des relations de sous-typage entre groupes de types. Steel et al. ont proposé une première définition d'un type de modèles et d'une relation de sous-typage afin d'amener les facilités de typage de l'objet au niveau des modèles, notamment le polymorphisme.

Nous commençons par présenter les mécanismes de base qui sous-tendent les systèmes de types orientés-objet : objets, types objets et classes (Section 4.1). Nous présentons ensuite le polymorphisme et l'héritage, qui permettent la réutilisation de la structure et du comportement des objets (Section 4.2). Nous abordons également les approches proposées pour repousser les limites du polymorphisme dans le cadre du polymorphisme de groupe de types (Section 4.3). Enfin, nous présentons les travaux de Steel et al. inspirés de ces approches et étudions leurs limites (Section 4.4).

4.1 Principes de base du typage objet

La relation de typage dans les langages de programmation orientés-objet est le support à de nombreuses facilités telles que la réutilisation sûre, l'abstraction, la détection précoce d'erreurs, le refactoring, l'auto-complétion ou les analyses d'impact. Nous présentons dans cette section les bases sur lesquelles s'appuient les systèmes de types orientés-objet.

4.1.1 Objets

Le concept fondamental sur lequel reposent les langages orientés-objet est le concept d'objet. Un objet est un artefact logiciel qui intègre à la fois des données, sous la forme de champs, et du comportement, sous la forme d'opérations [Dah68, AC96]. Chaque champ correspond à un espace mémoire qui peut contenir une valeur ou une référence vers un autre

espace mémoire. L'ensemble des valeurs et références contenues par les champs d'un objet est généralement désigné comme l'état de l'objet.

La notion d'objet fournit aux langages orientés-objet un mécanisme d'abstraction. En effet un objet est un ensemble de valeurs et de références (contenues par ses champs). De plus, un objet intègre également des opérations permettant de manipuler ces valeurs. Cet ensemble peut être référencé, manipulé et interprété comme une seule entité, i.e., comme une entité de première classe.

4.1.2 Types objets

Le type d'un objet ou d'une valeur fournit une information sur les manipulations valides qui sont possibles sur cet objet ou cette valeur. Les types permettent de distinguer les différents termes (objets, valeurs, expressions, etc.) d'un programme [Pie02]. Cette distinction permet d'effectuer des vérifications et de détecter des termes invalides. Par exemple, la division du nombre entier `3` par la chaîne de caractères `"trois"` n'est pas une opération valide. Ici, nombre entier et chaîne de caractères peuvent être vus comme les types des deux termes (`3` et `"trois"`) impliqués dans la division. Ces types permettent de spécifier quels termes sont acceptés par une variable ou par une opération, et quels termes peuvent être retournés par une opération. Ainsi la division entière peut spécifier qu'elle accepte deux nombres entiers en paramètre et retourne un nombre entier. Grâce à cette spécification, il est possible de détecter quand un terme invalide est passé en paramètre d'une division, en fonction du type de ce terme.

Une vision intuitive des types est de considérer un type comme la spécification d'un ensemble. Les éléments de cet ensemble étant toutes les valeurs possibles de ce type. Par exemple, le type nombre entier spécifie toutes les valeurs que peuvent prendre les termes qu'il type (i.e., l'ensemble des nombres entiers).

Un type objet expose une interface à travers laquelle il est possible d'accéder aux champs et aux opérations des objets qu'il type. Pour cela, un type objet fournit un ensemble de signatures de champs et d'opérations [1]. Un objet o qui satisfait l'interface exposée par le type T est dit typé par T (noté $o : T$). Ainsi, un objet peut être typé par plusieurs types, tant qu'il satisfait les interfaces que ces types exposent.

Les types objets fournissent un mécanisme de détection d'erreurs. En effet, ils permettent de détecter l'utilisation de termes invalides (e.g., l'appel d'une opération inexistante sur un objet) et de signaler à l'utilisateur la raison d'une erreur.

4.1.3 Classes

Les objets sont créés ou instanciés à partir de générateurs qui décrivent les champs et opérations des objets qu'ils génèrent. L'instanciation est le fait d'allouer un espace en mémoire pour contenir les valeurs des champs et le corps des opérations de l'objet instancié. Selon les langages l'instanciation peut être associée à une étape d'initialisation de tout ou partie des champs de l'objet instancié. Les générateurs les plus répandus dans les langages de programmation orientés-objet sont les classes [2] [Dah68].

1. Dans certains langages (*e.g.*, Java) les types objets ne peuvent pas déclarer de signatures de champs, afin de favoriser l'encapsulation de l'état des objets.

2. Bien qu'il existe d'autres mécanismes de génération d'objets, l'ingénierie dirigée par les modèles s'appuie sur une modélisation orientée-objet à base de classes. C'est pourquoi nous nous intéressons ici uniquement aux mécanismes de typage dans les langages de programmation orientés-objet à base de classes. Nous invitons le lecteur intéressé par ces autres mécanismes à se référer aux ouvrages et articles traitant de ce sujet [DMC92, Smi94, AC96].

Une classe fournit la définition des champs (y compris leurs valeurs d'initialisation le cas échéant) et des opérations de toutes ses instances (i.e., de tous les objets instanciés à partir de cette classe). Toutes les instances d'une classe partagent les mêmes opérations. Par contre, même si les instances d'une classe partagent la même structure de champs (noms, nombres, types des champs), chaque instance possède son état propre. Une instance ne partage pas le contenu de ses champs (valeurs ou références) avec les autres instances de la même classe. Un objet, s'il peut avoir plusieurs types, n'est instance que d'une classe : celle à partir de laquelle il a été créé.

Listing 4.1 – Pseudo-code décrivant la classe `Edge` permettant d'instancier des arcs

```
class Edge {
  field source : Node
  field target : Node
  operation getSource(): Node is do
    return source
  end
  operation setSource(newSource : Node) is do
    source := newSource
  end
  operation getTarget(): Node is do
    return target
  end
  operation setTarget(newTarget : Node) is do
    target := newTarget
  end
}
```

Par exemple, un objet instancié à partir de la classe `Edge` présentée en Listing 4.1 possèdera deux champs (les fields `source` et `target`) et quatre opérations permettant d'obtenir le contenu de ces champs et de le modifier.

L'instanciation d'une classe se fait généralement à l'aide de procédures spécifiques appelées constructeurs. La tâche d'un constructeur est d'allouer en mémoire l'espace nécessaire à un objet et, le cas échéant, d'initialiser les champs de cet objet, éventuellement à l'aide de paramètres passés par l'utilisateur.

La notion de classe fournit aux langages orientés-objet un mécanisme de réutilisation. En effet, le code des opérations d'une classe est partagé par toutes les instances de cette classe, de même que le ou les constructeurs de cette classe.

4.1.4 Interface et implémentation

Certains langages orientés-objet ne séparent pas clairement les notions de type objet et de classe. Un type définit une interface par laquelle un objet peut être accédé et modifié (e.g., les interfaces Java), tandis qu'une classe définit l'implémentation de ses instances (e.g., le corps des opérations). Les classes peuvent être utilisées comme types parce qu'elles définissent également de manière implicite une interface permettant d'accéder à leurs instances. En effet, il est possible d'extraire un type objet d'une classe, simplement en ignorant les détails d'implémentation (le corps des opérations, les constructeurs et les valeurs d'initialisation).

Nous appelons ce type extrait d'une classe : type exact des objets instances de cette classe. Le type exact d'un objet est son type le plus précis. En effet, le type exact d'un objet fournit son interface complète, i.e., l'intégralité des signatures des opérations et champs qui sont accessibles sur cet objet.

Par exemple, le type extrait de la classe `Edge` contiendra la signature des deux champs `source` et `target` et des quatre opérations `getSource`, `setSource`, `getTarget` et `setTarget`. Ce type est le type exact de toutes les instances de la classe `Edge`.

Cependant, séparer l'interface (le type) de l'implémentation (la classe) permet de fournir plusieurs implémentations différentes pour une même interface. En effet, pour manipuler un objet, il n'est pas nécessaire de connaître les détails de l'implémentation de la classe dont il est instance, mais uniquement les champs et opérations qu'il possède et qui sont accessibles : son interface. Ceci est permis grâce au mécanisme de liaison dynamique (dynamic dispatch en anglais).

La liaison dynamique est un mécanisme utilisé à l'exécution quand une opération est appelée depuis un objet. Indépendamment du type de l'objet connu statiquement (et qui permet de savoir que l'opération est disponible sur l'objet), la liaison dynamique permet d'appeler l'opération de la classe dont est instance cet objet. Ainsi, même si la seule information connue statiquement sur un objet est son type, à l'exécution une opération appelée sur cet objet sera bien l'opération définie par sa classe. Ainsi, il est possible d'obtenir un comportement différent en fonction de la classe dont est instance l'objet sur lequel est appelée l'opération.

La séparation des types objets des classes offre un mécanisme d'abstraction aux langages orientés-objet. Cette séparation permet de fournir une interface uniforme au-dessus de plusieurs implémentations différentes.

4.2 Réutilisation : polymorphisme et héritage

Les langages de programmation orientés-objet fournissent des mécanismes de réutilisation plus avancés que le simple partage des opérations au sein d'une classe. Le polymorphisme permet d'utiliser une même opération avec différents types. Il existe deux incarnations majeures du polymorphisme dans les langages de programmation orientés-objet : le polymorphisme de sous-type et le polymorphisme paramétrique. L'héritage permet de réutiliser le code d'une classe pour en créer une nouvelle, évitant ainsi de dupliquer la définition de champs et d'opérations équivalents.

4.2.1 Polymorphisme de sous-type : relations de sous-typage

Le polymorphisme de sous-type ou simplement sous-typage a pour but de permettre d'utiliser un objet typé par un type objet T' là où un objet typé par un type objet T est attendu. Ainsi, il est possible d'utiliser une opération attendant un paramètre de type T sur un objet de type T', c'est à dire de réutiliser cette opération sur les objets de deux types différents. Si les objets typés par T' sont substituables "de manière sûre" aux objets typés par T, on dit que T' est un sous-type de T (et que T est un super-type de T'). Ici, "de manière sûre", signifie de manière à ne pas provoquer d'erreurs à l'exécution. Liskov et Wing définissent la substitution sûre de la manière suivante :

Définition 4.1. ***(Sous-typage)*** "Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be provable for objects y of type T' where T' is a subtype of T." [LW94]

Cependant les propriétés qu'il est possible de prouver sur un objet de type T dépendent des informations disponibles dans T [LW94]. En effet si tous les types objets embarquent la signature des opérations des objets qu'ils typent, il est certaines autres informations qui dépendent de chaque langage. Différents mécanismes de sous-typage sont donc mis en œuvre dans les différents langages orientés-objet selon le sens donné à "de manière sûre", i.e., selon les informations utilisées pour définir les types.

Si un type peut être vu comme la spécification d'un ensemble, le sous-typage peut alors être considéré comme une relation de sous-ensemble entre deux types. Le sous-type spécifie un

Listing 4.2 – Pseudo-code définissant un type objet exposant une interface de manipulation d'arcs

```
type Edge {
  operation getSource(): Node
  operation setSource(newSource : Node)
  operation getTarget(): Node
  operation setTarget(newTarget : Node)
}
```

Listing 4.3 – Pseudo-code définissant un type objet exposant une interface de manipulation d'arcs pondérés

```
type WeightedEdge {
  operation getSource(): Node
  operation setSource(newSource : Node)
  operation getTarget(): Node
  operation setTarget(newTarget : Node)
  operation getWeight() : Integer
  operation setWeight(newWeight : Integer)
}
```

sous ensemble de l'ensemble défini par son super-type. Par exemple le type `Edge` présenté en Listing 4.2 est la spécification de l'ensemble $\mathbb{E}$ des objets possédant les opérations `getSource`, `setSource`, `getTarget` et `setTarget`. Le type `WeightedEdge` présenté en Listing 4.3 est la spécification de l'ensemble $\mathbb{E}_W$ des objets possédant les opérations `getSource`, `setSource`, `getTarget` et `setTarget` et possédant deux opérations `getWeight` et `setWeight`. Les objets appartenant à $\mathbb{E}_W$ appartiennent donc également à $\mathbb{E}$ (ils possèdent les quatre opérations définies par `Edge`).

Dans la suite de cette section nous détaillons d'abord les différentes informations qui peuvent être prises en compte dans les types, et les propriétés qu'elles doivent respecter pour garantir le sous-typage et donc une substitution sûre. Nous présentons ensuite les différents moyens de déclarer et de vérifier les relations de sous-typage en fonction de ces informations. En effet, selon l'implémentation du sous-typage, celui-ci peut offrir différentes facilités.

4.2.1.1 Covariance, contravariance et invariance

Les notions de covariance et de contravariance sont des notions essentielles dans le sous-typage objet. Elles se réfèrent à la manière dont les propriétés varient avec ou contre une relation de sous-typage.

Un sous-type est une spécialisation de son super-type, i.e., l'ensemble des objets qui sont typés par le sous-type est un sous-ensemble de l'ensemble des objets typés par le super-type. Ainsi, une propriété covariante est aussi ou plus restrictive (plus spécialisée) dans un sous-type que dans son super-type (i.e., la spécialisation de la propriété varie avec, dans le même sens que, la spécialisation des types). Inversement, une propriété contravariante est aussi ou moins restrictive dans un sous-type que dans son super-type (i.e., la spécialisation de la propriété varie contre, dans le sens inverse de, la spécialisation des types).

La covariance et la contravariance peuvent être utilisées pour vérifier une relation de sous-typage entre deux types. Elles sont généralement utilisées dans les langages orientés-objet pour comparer les signatures des champs et des opérations (qui peuvent inclure la déclaration des exceptions qu'une opération pourrait soulever). Dans les langages qui incluent des contrats, tels qu'Eiffel, la covariance et la contravariance peuvent également être utilisées pour comparer les contrats définis au sein de deux types.

Types des paramètres et type de retour des opérations Pour qu'un objet de type T' soit substituable à un objet de type T, une opération de T' doit accepter au moins tous les arguments acceptés par la même opération de T. On ne peut donc pas spécialiser les types des paramètres des opérations dans un sous-type, au risque de voir un appel d'opération échouer. Les types des paramètres d'une opération d'un sous-type doivent être contravariants par rapport aux types de paramètres de l'opération du super-type.

Inversement une opération de T' ne doit pas renvoyer un objet qui ne peut pas être renvoyé par la même opération de T. Il n'est donc pas possible de généraliser le type de retour d'une opération. Le type de retour d'une opération d'un sous-type doit être covariant par rapport au type de retour de l'opération du super-type.

Invariance des types de champs Un champ typé par C d'un objet de type T peut être à la fois accédé et modifié. Il est possible de modéliser cette situation par une paire d'opérations (nommées accesseurs) l'une permettant d'accéder au champ (le getter), et retournant donc un objet de type C ; et l'autre permettant de modifier le champ (le setter), et acceptant en paramètre un objet de type C.

Afin d'être un sous-type du type T, un type T' devra fournir un champ compatible (de type C') et donc deux opérations compatibles. Le getter devra donc avoir un type de retour covariant et le setter un type de paramètre contravariant. Dans les deux cas, ce type devra être le type du champ : C'. C' doit donc être à la fois covariant et contravariant par rapport au type C. Le seul type qui soit covariant et contravariant par rapport à C est C lui-même. Les types de champs d'un sous-type doivent donc être invariants par rapport aux types des champs du super-type.

Si le champ du super-type est en lecture seule, i.e., ne peut pas être modifié, seul le getter est à prendre en compte. Dans ce cas C' doit uniquement être covariant par rapport à C, il peut donc en être un sous-type.

Exceptions Une opération d'un sous-type T' ne doit pas soulever une exception qui ne peut être soulevée par la même opération du super-type T, sinon l'exception soulevée ne serait pas traitée dans les programmes attendant un objet de type T. Ainsi, une opération d'un sous-type ne peut pas déclarer soulever plus d'exceptions que l'opération du super-type et les types des exceptions doivent être covariants.

Contrats Les contrats, sous la forme d'invariants de types et de pré et post-conditions d'opérations peuvent également être pris en compte dans une relation de sous-typage. Pour les mêmes raisons qui nécessitent que les types de paramètres soient contravariants et les types de retour soient covariants dans un sous-type, les pré-conditions doivent être contravariantes (i.e., les pré-conditions du sous-type ne doivent pas restreindre les conditions possibles d'utilisation de l'opération) et les post-conditions doivent être covariantes (i.e., les post-conditions ne doivent pas être généralisées dans le sous-type). Comme les invariants sont des contraintes sur les valeurs qu'un type peut prendre, ils ne doivent pas être généralisés dans un sous-type, sinon un objet typé par le sous-type pourrait ne pas être typé par le super-type. Les invariants de type doivent donc être covariants.

4.2.1.2 Déclaration et vérification des relations de sous-typage

Les systèmes de types offrent différents moyens de déclarer les relations de sous-typage, i.e., d'indiquer qu'un type doit être un sous-type d'un autre. Ils offrent également différents moyens de vérifier ces relations de sous-typage une fois déclarées.

Une relation de sous-typage peut être déclarée de manière *explicite* ou *implicite*. Une déclaration est explicite quand elle est faite par l'utilisateur, à l'aide d'une construction syntaxique du langage. C'est par exemple le cas en Java (via les mots-clés `extends` et `implements`). Une relation de sous-typage peut également être déclarée de manière implicite, sans l'utilisation de constructions syntaxiques. Dans ce cas, le système de types infère la relation en fonction

de l'usage qui est fait des types. Par exemple, Ruby autorise un appel à une fonction avec en argument un objet dont le type n'a pas été déclaré explicitement sous-type du type du paramètre. Le système de types infère alors qu'il doit exister une relation de sous-typage entre ces deux types, qui sera ensuite vérifiée.

Les systèmes de types qui utilisent des déclarations explicites des relations de sous-typage (e.g., Java) sont dits nominatifs, parce qu'une fois la relation de sous-typage établie, il est suffisant pour le système de types de savoir que le type nommé T' est un sous-type du type nommé T. Les systèmes de types utilisant des déclarations implicites des relations de sous-typage (e.g., Ruby) sont dits structurels car ils utilisent souvent des types non-nommés qui sont uniquement portés par l'objet ou le paramètre qu'ils typent. Ils s'appuient donc uniquement sur la structure des types pour vérifier les relations de sous-typage.

De plus les relations de sous-typage déclarées de manière explicite peuvent l'être à la définition ou après la définition d'un type. Par exemple, Java n'autorise la déclaration de relations de sous-typage qu'à la définition du type (i.e., quand la classe ou l'interface est déclarée). Au contraire, Scala propose à l'utilisateur d'écrire des conversions implicites, qui permettent de rendre les objets d'un type substituables à ceux d'un autre, et ce, n'importe où dans un programme.

Chacune de ces possibilités peut influer sur les facilités offertes par le système de types. La déclaration explicite des relations de sous-typage fournit une forme de documentation, particulièrement quand elle a lieu à la définition du type, alors que la déclaration implicite peut permettre une substituabilité accidentelle, i.e., autoriser des relations de sous-typage inconnues de l'utilisateur. Par contre, la déclaration implicite permet d'utiliser des types non-nommés, qui peuvent servir par exemple à typer les paramètres d'une opération. Plutôt que de créer un type ad-hoc nommé déclarant toutes les signatures utilisées par l'opération et de déclarer explicitement les relations de sous-typage avec tous les types nécessaires, il est possible de créer ce même type ad-hoc, de ne pas le nommer, et de laisser le système de types inférer et vérifier les relations de sous-typage à l'appel de l'opération. Enfin, la déclaration d'une relation de sous-typage après la définition du type peut être utile dans le cas où l'utilisateur n'a pas la main sur les types, et ne peut donc pas les modifier.

Une fois déclarée, une relation de sous-typage doit être vérifiée. Le système de types doit vérifier que les objets du sous-type sont bien substituables à ceux du super-type. Cette vérification peut se faire statiquement, avant l'exécution du programme, ou dynamiquement à l'exécution du programme. La façon de vérifier les relations de sous-typage peut influer sur les facilités fournies par le système de types.

D'une part, la vérification à la conception, ou statique, permet une détection plus précoce des erreurs de type et des erreurs de programmation que la vérification à l'exécution, ou dynamique. La vérification statique permet donc de signaler les erreurs à l'utilisateur plus tôt. Elle permet également le développement d'outils, comme les optimisations, l'auto-complétion ou les analyses d'impact basées sur les types. En outre, par rapport à la vérification dynamique, la vérification statique demande beaucoup moins de tests pour atteindre le même niveau de sécurité à l'exécution.

D'autre part, la vérification dynamique utilise des informations de type plus précises. Lorsque le programme est lancé, le type réel d'une variable est connu et pas seulement son type déclaré. La vérification dynamique autorise donc des programmes valides qui seraient interdits par une vérification statique en raison d'un manque d'informations.

Le sous-typage, ou polymorphisme de sous-type, offre un mécanisme de réutilisation aux langages de programmation orientés-objet, en fournissant la possibilité de réutiliser une

opération écrite pour un type sur un objet d'un autre type.

4.2.2 Héritage

L'héritage permet de créer une classe en réutilisant le code d'une ou plusieurs autres classes. Les objets instances de la classe ainsi créée (la sous-classe) héritent des champs et des opérations définis par les classes étendues (les super-classes) [Dah68].

De plus il existe des mécanismes permettant de modifier les champs et opérations hérités, e.g., de redéfinir (override en anglais) dans la sous-classe certains champs ou opérations afin de modifier leur comportement ou leur signature. Certains langages fournissent également des moyens d'appeler la (ou les) super-opération (l'opération correspondante de la super-classe) d'une opération redéfinie. Ainsi, il est possible de spécialiser une opération, sans devoir réécrire le code de la super-opération.

Par exemple, la classe `WeightedEdge` présentée en Listing 4.4 étend la classe `Edge` (cf. Listing 4.1) en définissant un nouveau champ `weight` et deux nouvelles opérations permettant de manipuler ce champ. La classe `WeightedEdge` redéfinit également l'opération `setSource` afin de modifier son comportement. La nouvelle opération fait appel à sa super-opération pour assigner l'objet passé en paramètre au champ `source` et assigne une valeur au champ `weight` en fonction d'un test. Les objets instances de `WeightedEdge` posséderont donc trois champs : `source` et `target`, hérités de la super-classe `Edge`, et `weight` et six opérations (trois héritées telles quelles de `Edge`, une héritée de `Edge` et redéfinie, et deux définies par la sous-classe `WeightedEdge`).

Listing 4.4 – Pseudo-code définissant la classe `WeightedEdge` étendant la classe `Edge` et permettant d'instancier des arcs pondérés

```
class WeightedEdge extends Edge {
  field weight : Integer
  operation getWeight(): Integer is do
    return weight
  end
  operation setWeight(newWeight : Integer) is do
    weight := newWeight
  end
  override operation setSource(newSource : Node) is do
    super(newSource)
    if (newSource.isRoot()) {weight := 0}
    else {weight := 1}
  end
}
```

L'héritage offre un mécanisme de réutilisation aux langages orientés-objets. Il permet en effet de réutiliser sans recopie les champs et les opérations d'une classe à l'autre.

4.2.3 Sous-typage ⇔ Héritage

De la même manière que certains langages séparent les notions de classes et de types objets et que d'autres les combinent, certains langages séparent le sous-typage de l'héritage et d'autres non. Ainsi, l'héritage n'implique pas toujours le sous-typage, pas plus que le sous-typage n'implique automatiquement l'héritage [CHC90].

En effet, si l'héritage implique le sous-typage, i.e., si le type exact de la sous-classe est automatiquement un sous-type du type exact de la super-classe, certaines redéfinitions ne sont pas possibles. Ces redéfinitions sont illégales car elles brisent le sous-typage. Par exemple, Eiffel autorise de modifier la signature d'une opération héritée en changeant un type de paramètre par

Listing 4.5 – Pseudo-code définissant une classe générique utilisant un paramètre de type (`P`).

```
class Edge[P <: Node] {
  field source : P
  field target : P
  operation getSource(): P is do
    return source
  end
  operation setSource(newSource : P) is do
    source := newSource
    newSource.getOutGoingEdges.add(self)
  end
  operation getTarget(): P is do
    return target
  end
  operation setTarget(newTarget : P) is do
    target := newTarget
    newTarget.getInGoingEdges.add(self)
  end
}
```

Listing 4.6 – Pseudo-code définissant une classe générique utilisant une variable de type (`V`).

```
class Edge {
  typevar V <: Node
  field source : V
  field target : V
  operation getSource(): V is do
    return source
  end
  operation setSource(newSource : V) is do
    source := newSource
    newSource.getOutGoingEdges.add(self)
  end
  operation getTarget(): V is do
    return target
  end
  operation setTarget(newTarget : V) is do
    target := newTarget
    newTarget.getInGoingEdges.add(self)
  end
}
```

un type covariant. D'après la définition du sous-typage présentée dans la section précédente, une telle redéfinition brise le sous-typage. C'est pourquoi certains langages autorisent l'héritage sans sous-typage, afin d'offrir plus de flexibilité dans la réutilisation du code de la super-classe.

À l'inverse, certains langages autorisent à ce que le sous-typage n'implique pas l'héritage. Il s'agit généralement de langages utilisant des systèmes de types structurels. En effet, dans ces langages, la relation de sous-typage n'a pas besoin d'être déclarée à la création du type, contrairement à l'héritage.

4.2.4 Polymorphisme paramétrique : paramètres de types et variables de types

Le polymorphisme paramétrique permet de définir un code de manière générique, en utilisant des paramètres ou des variables à la place des types [Pie02, Chap. 23]. Les paramètres de types et variables de types sont deux mécanismes qui permettent de contourner les contraintes de contravariance des types de paramètres des opérations et de covariance des types de retour par la définition de classes génériques. Une classe générique est une classe dont certains champs ou opérations sont typés à l'aide de types inconnus à la création de la classe. Les classes génériques permettent de définir des opérations pour ces types inconnus, et de spécialiser ces opérations de manière uniquement covariante ou uniquement contravariante, tout en restant bien typées.

Par exemple les deux classes `Edge` présentées en Listings 4.5 et 4.6 sont deux classes génériques utilisant un paramètre de type pour la première (`P`) et une variable de type pour la deuxième (`V`).

Ces types seront fixés plus tard, à l'utilisation de la classe générique pour les paramètres de type (le type réel est passé en paramètre) ou dans une sous-classe pour les variables de types. Dans les Listings 4.5 et 4.6, le paramètre et la variable de type possèdent une borne supérieure : le type `Node`. Cette borne indique que le paramètre réel ou la valeur prise par la variable doivent être des sous-types du type `Node`. Il est également possible de définir des bornes inférieures, n'autorisant que les super-types de la borne.

A l'usage, le type fixé déterminera le type des paramètres des opérations `getSource` et

`setSource`. Chaque classe obtenue de cette manière sera spécialisée pour un type et correcte vis-à-vis du système de types.

On ne peut pas extraire de type ni instancier une classe générique. Il faut pour cela que les types "inconnus" soient fixés. Une classe générique définit donc un ensemble de types : tous les types qui peuvent être extraits de toutes les classes obtenues après avoir fixé les types "inconnus".

Les classes génériques fournissent un mécanisme de réutilisation aux langages de programmation orientés-objet, puisqu'elles permettent de définir des opérations valables sur une infinité de types tout en restreignant les usages de ces opérations. Ainsi, un objet de type `Edge[WeightedNode]` (le type obtenu en paramétrant la classe `Edge[P <: Node]` du Listing 4.5 par le type `WeightedNode`) n'acceptera comme `source` ou `target` qu'un objet de type `WeightedNode`, contrairement à un objet de type `Edge` (le type extrait de la classe `Edge` du Listing 4.1) qui acceptera tout objet typé par le type `Node`.

L'usage le plus connu des classes génériques est la définition de structures de données comme les collections, qui, quel que soit le type d'éléments qu'elles stockent, ont le même comportement, mais pour lesquelles il n'est pas possible de mélanger des éléments de différents types.

4.3 Polymorphisme de groupe de types et spécialisations de groupe de classes

Dans le cas général, définir un type objet revient à définir un groupe de types. En effet, un type objet est constitué de signatures de champs et d'opérations, qui sont typées. Ces signatures définissent donc des liens entre types. À part dans les cas où le type objet ne définit aucune signature, ou définit uniquement des signatures auto-référentielles, un type objet fait donc partie d'un groupe de types auxquels il est lié.

Le polymorphisme de groupe de types permet la substitution sûre d'un ensemble d'objets à un autre ensemble d'objets. Cette situation se présente dans le cadre de l'ingénierie dirigée par les modèles lorsque l'on cherche à substituer un modèle (i.e., un ensemble d'objets) à un autre. La spécialisation de groupe de classes permet d'hériter et d'étendre un ensemble de classes de manière à conserver le polymorphisme de groupe de types (il s'agit donc de l'équivalent de l'héritage avec sous-typage pour les groupes de classes). Le polymorphisme de groupe de types et la spécialisation de groupe de classes s'appuient sur des relations entre types objets différentes du sous-typage. En effet le sous-typage permet la réutilisation sûre tant qu'un seul objet est concerné. Cependant quand plusieurs types doivent être spécialisés en parallèle, ou quand un ensemble d'objets doit être substitué à un autre, les mécanismes de typage présentés dans la Section 4.2 trouvent leurs limites.

4.3.1 Limites du sous-typage objet pour les groupes de types

Prenons l'exemple de nœuds colorés. Les types représentant ces nœuds sont décrits en Listings 4.7 et 4.8 et en Figures 4.1 et 4.2. Le type `ColoredNode` déclare un champ `color` de type `Color`. Le type `Color` possède un champ `name` qui permet de l'identifier. Le type `RGBColoredNode` est un genre de nœud coloré particulier, dont le champ `color` est de type `RGBColor`. Le type `RGBColor` possède en plus d'un champ `name`, trois champs de type `Integer` permettant d'encoder la couleur.

Listing 4.7 – Pseudo-code déclarant un groupe de types objets composé de `ColoredNode` et de `Color`.

```
type ColoredNode {
  field color : Color
  [...]
}

type Color {
  field name : String
}
```

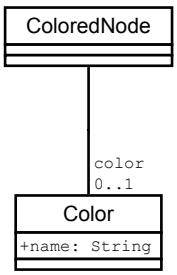

FIGURE 4.1 – Groupe de types objets composé de `ColoredNode` et de `Color`.

Listing 4.8 – Pseudo-code déclarant un groupe de types objets composé de `RGBColoredNode` et de `RGBColor`.

```
type RGBColoredNode {
  field color : RGBColor
  [...]
}

type RGBColor {
  field name : String
  field red : Integer
  field green : Integer
  field blue : Integer
}
```

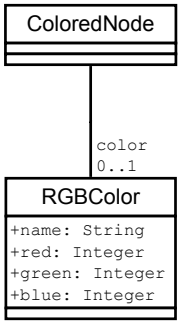

FIGURE 4.2 – Groupe de types objets composé de `RGBColoredNode` et de `RGBColor`.

Dans une telle configuration, il est impossible d'établir une relation de sous-typage entre les types `ColoredNode` et `RGBColoredNode` présentés en Figures 4.1 et 4.2. En effet, le champ `color` du type `ColoredNode` doit être de type invariant dans les sous-types de `ColoredNode`. Or le type du champ `color` du type `RGBColoredNode` n'est pas `Color` mais `RGBColor`. Il n'est donc pas invariant mais covariant. D'une manière plus générale, spécialiser deux types ou plus en parallèle brise le sous-typage tel que présenté jusqu'ici, en violant la contrainte d'invariance des types des champs.

`RGBColoredNode` n'étant pas un sous-type de `ColoredNode`, il n'est pas possible d'écrire un programme manipulant des objets d'un type ou de l'autre s'appuyant sur le sous-typage. Il existe deux solutions pour contourner ce frein à la réutilisation :

- Typer le champ `color` du type `RGBColor` par `Color` exigerait de tester et de convertir des objets de type `RGBColor` afin de s'assurer de la spécialisation du champ (i.e., afin de s'assurer qu'un objet de type `Color` n'est pas assigné au champ `color` d'un objet de type `RGBColoredNode`). De telles conversions sont dangereuses pour le typage et sources d'erreurs.
- Utiliser des classes génériques paramétrées par les types de nœuds et de couleur est une solution qui ne passe pas à l'échelle en terme d'utilisabilité. En effet chaque classe doit posséder autant de paramètres de type que de types concernés par la spécialisation, chacun dépendant des autres, rendant le code verbeux (cf. Listing 4.9). De plus, les inter-dépendances entre deux types ne peuvent pas être exprimées à l'aide des bornes supérieures et mènent à l'utilisation de types bruts (raw types en anglais), imprécis et dangereux (cf. Listing 4.10).

Listing 4.9 – Exemple de groupe de types utilisant les paramètres de type en Java.

```
class ColoredGraph<C extends Color, N extends ColoredNode<C,E>, E extends Edge<N>>
```

Listing 4.10 – Exemple de groupe de types utilisant des types *bruts* en Java.

```
class Node<E extends Edge>

class Edge<N extends Node>
```

Cette limite du sous-typage objet se retrouve dans l'ingénierie dirigée par les modèles. En effet, afin de réutiliser des transformations de modèles d'un langage à un autre, il faut être capable de substituer un modèle (un ensemble d'objets typés par un ensemble de types objets) à un autre. Or, le sous-typage objet ne permet pas de substituabilité de groupes d'objets typés par des groupes de types différents.

Plusieurs travaux ont été menés dans la communauté des langages de programmation orientés-objet afin de rendre possible l'intuition que, tant que les objets d'un groupe ne sont pas mêlés aux objets de l'autre, la substitution d'un groupe de type à l'autre est sûre et permet une réutilisation à plus grande échelle [BV99, Ern01, BSvGF03, DJAO04, ISV05]. Cette substitution sûre peut être établie soit par construction : un groupe de classes qui en spécialise un autre en est forcément le sous-type [Ern01, DJAO04, ISV05] ; soit par vérification : les propriétés des types des deux groupes de types sont comparées afin de déterminer si une relation de correspondance entre les types objets des groupes existe [BV99, BSvGF03]. Ces travaux s'appuient sur la notion de types chemin-dépendants (path-dependent types en anglais).

4.3.2 Types *chemin-dépendants*

Les types chemin-dépendants sont des types dont la "valeur" réelle est liée à l'exécution (de la même manière que les paramètres de type).

La valeur d'un type chemin-dépendant dépend du chemin utilisé pour référencer le type. Ce chemin est variable, contrairement aux noms qualifiés utilisés pour référencer les types objets classiques. Ainsi, quand la valeur du chemin change, celle du type chemin-dépendant change également.

Les types chemin-dépendants sont utilisés dans le polymorphisme de groupes de types pour faire varier les types de variables ou d'opérations en fonction du groupe de types auquel elles appartiennent. Ainsi, le chemin utilisé pour référencer un type chemin-dépendant est une variable qui désigne un groupe de types.

Selon les approches, cette variable peut être un objet [Ern01] ou un paramètre de type [BV99, BSvGF03, DJAO04, ISV05]. Le Listing 4.11 présente deux types chemin-dépendants (`group.ColoredNode` et `group.Color`) exprimés à l'aide d'une variable objet (le paramètre `group`). Ainsi, le type des paramètres `node` et `color` change selon l'objet référencé par `group`.

Le Listing 4.12 présente deux types chemin-dépendants (`G.ColoredNode` et `G.Color`) exprimés à l'aide d'un paramètre de type (`G`). Ainsi, le type des paramètres `node` et `color` change selon le type passé en paramètre à l'opération.

4.3.3 Spécialisation de groupes de classes

Les approches proposant d'utiliser la spécialisation de groupes de classes pour supporter le polymorphisme de groupes de types représentent un groupe de types à l'aide d'une classe contenant des classes internes [Ern01, DJAO04, ISV05]. Le Listing 4.13 reprend l'exemple du groupe de types du Listing 4.7 à l'aide de classes internes. Le groupe de types est représenté

Listing 4.11 – Utilisation d'un objet pour définir les types chemin-dépendants `group.ColoredNode` et `group.Color`.

```
operation addColor(group : ColorGroup, node : group.ColoredNode, color : group.Color) is do
  ...
end
```

Listing 4.12 – Utilisation d'un paramètre de type pour définir les types chemin-dépendants `G.ColoredNode` et `G.Color`.

```
operation addColor[G <: ColorGroup](node : G.ColoredNode, color : G.Color) is do
  ...
end
```

Listing 4.13 – Réécriture de l'exemple du Listing 4.7 à l'aide de la spécialisation de groupes de classes.

```
class ColorGroup {
  class ColoredNode {
    field color : MyGroup.Color
    [...]
  }
  class Color {
    field name : String
  }
}
```

Listing 4.14 – Réécriture de l'exemple du Listing 4.8 à l'aide de la spécialisation de groupes de classes.

```
class RGBGroup extends ColorGroup {
  class Color {
    field red : Integer
    field green : Integer
    field blue : Integer
  }
}
```

par la classe `ColorGroup`, qui contient deux classes représentant les types du groupe de types (`ColoredNode` et `Color`).

Pour créer un groupe de types substituables au groupe représenté par la classe `ColorGroup`, il faut hériter de cette classe. Toute la structure de la classe, y compris les classes internes, est héritée. Par exemple, le Listing 4.14 présente la classe `RGBGroup` qui hérite de la classe `ColorGroup`. Les classes internes de `ColorGroup` sont héritées par `RGBGroup`. De plus la classe interne `Color` est étendue par l'ajout de trois champs (`red`, `green` et `blue`).

Afin d'assurer la substituabilité de groupes, ces approches s'appuient sur les types chemin-dépendant. Le Listing 4.13 présente à la ligne 3 l'usage d'un type chemin-dépendant pour typer le champ `color` : `MyGroup.Color`. `MyGroup` est un mot-clé indiquant une variable spécifique (à la manière des mots-clés `self` ou `this`) qui référence une classe. La valeur de cette variable dépend du groupe dans laquelle elle est trouvée. Dans le cas du Listing 4.13, le groupe est la classe `ColorGroup`. La valeur de `MyGroup` est donc la classe `ColorGroup`. La valeur du type chemin-dépendant `MyGroup.Color` est donc résolue comme la classe `Color` interne à la classe `ColorGroup` (ou `ColorGroup.Color`).

Le champ `color` est hérité tel quel dans la classe `ColoredNode` interne à la classe `RGBGroup` du Listing 4.13. Mais dans cette classe, `MyGroup` ne désigne plus `ColorGroup` mais `RGBGroup`. Le type du champ `color`, `MyGroup.Color` est donc résolu comme la classe `RGBGroup.Color` qui contient les trois champs `red`, `green` et `blue`.

Ainsi, les objets typés par les types du groupe de types `RGBGroup` peuvent être manipulés comme des objets typés par les types objets du groupe de types `ColorGroup` et donc leur être substitués, tout en conservant leur spécificité (l'encodage RGB des couleurs) et sans risquer de mêler des objets typés par des types de groupes différents.

Listing 4.15 – Réécriture de l'exemple du Listing 4.7 à l'aide de types internes.

```
type ColorGroup {
  type ColoredNode {
    field color : ColorGroup.Color
    [...]
  }
  type Color {
    field name : String
  }
}
```

Listing 4.16 – Réécriture de l'exemple du Listing 4.8 à l'aide de types internes.

```
type RGBGroup {
  type ColoredNode {
    field color : RGBGroup.RGBColor
    [...]
  }
  type RGBColor {
    field name : String
    field red : Integer
    field green : Integer
    field blue : Integer
  }
}
```

4.3.4 Correspondance entre types objets

Dans cette deuxième approche, un opérateur de comparaison entre les types objets de deux groupes a été établi : la correspondance de types (match en anglais). On note T' <# T si T' correspond à T dans le cadre de groupes de types. Pour qu'un type T' corresponde à un autre type T, il doit exposer les mêmes champs et les mêmes signatures, dans lesquels les types appartenant au groupe de T peuvent être remplacés par les types appartenant au groupe de T'.

L'exemple donné plus haut est repris dans les Listings 4.15 et 4.16. Dans le Listing 4.15, le type externe `ColorGroup` représente le groupe de types, et ses types internes représentent les types de ce groupe.

Dans cet exemple, `RGBGroup.ColoredNode` correspond à `ColorGroup.ColoredNode` (noté `RGBGroup.ColoredNode` <# `ColorGroup.ColoredNode`). En effet, `RGBGroup.ColoredNode` expose le même champ `color` que `ColorGroup.ColoredNode`, dans lequel le type `Color` appartenant au groupe `ColorGroup` a été remplacé par le type `RGBColor` appartenant au groupe `RGBGroup`.

On peut donc substituer des objets typés par `RGBGroup.ColoredNode` à des objets de type `ColorGroup.ColoredNode` si l'on substitue également les autres objets typés par un type du groupe `ColorGroup` par des objets typés par un type du groupe `RGBGroup` (`ColorGroup.Color` par `RGBGroup.Color` en l'occurrence).

C'est cette relation de correspondance entre types objets que Steel et al. ont étendu pour la définition d'une relation de sous-typage entre types de modèles (cf. Section 4.4) et que nous reprenons afin d'étendre le champ d'application des relations de sous-typage entre types de modèles (cf. Chapitre 7).

4.4 Typage de modèles

Le typage de modèles a été proposé par Steel et al. pour amener les facilités de typage connues dans les langages orientés-objet au niveau des modèles, en particulier le polymorphisme [SJ07]. Cette approche comporte deux concepts principaux : les types de modèles et la relation de sous-typage entre deux types de modèles. Grâce à la relation de sous-typage entre types de modèles, il devient possible de réutiliser de manière sûre une transformation définie pour un type de modèles MT sur un modèle typé par un sous-type de MT.

4.4.1 Types de modèles

L'approche la plus générale de définition d'un modèle est de définir un modèle comme un simple ensemble d'objets, ces objets étant instances d'un ensemble de classes (la syntaxe abstraite du langage auquel appartient le modèle). À partir de cette définition, Steel et al. définissent donc un type de modèles comme l'ensemble des types objets de tous les objets contenus par ce modèle :

Définition 4.2. ***(Type de modèles (1) [Ste07])*** *"The type of a model is the set of the object types of all the contained objects."*

Steel et al. précisent également que la représentation d'un type de modèles se fait à travers un ensemble de classes MOF, i.e., un ensemble d'instances de la classe `Class` du métalangage MOF. En effet le MOF ne distingue pas les types objets des classes.

Définition 4.3. ***(Type de modèles (2) [SJ07])*** *"We define a model type as a set of MOF classes (and, of course the references that they contain)."*

Selon la définition adoptée, un modèle peut ou non être typé par plusieurs types de modèles. Pour obtenir la substituabilité de modèles, il est cependant nécessaire d'autoriser un modèle à être typé par plusieurs types de modèles. On peut considérer le type de modèles défini en définition 4.2 (et qui peut être extrait directement à partir d'un métamodèle) comme le type exact d'un modèle.

4.4.2 Relation de sous-typage entre types de modèles

Une des évolutions simple et fréquente des métamodèles est l'ajout d'un champ à une classe. Dans ce cas, toute classe qui réfère la classe modifiée va voir varier son propre type. Les métamodèles étant en général des ensembles connexes de classes, l'évolution d'une classe peut faire changer en cascade le type associé à une majorité des classes du métamodèle. Plus formellement, l'ajout d'un champ est susceptible de causer une redéfinition covariante d'un type de champ quelque part dans le métamodèle, empêchant toute relation de sous-typage entre deux langages.

Pour définir une relation de sous-typage entre deux types de modèles, et donc autoriser un modèle à être typé par plusieurs types de modèles, Steel et al. s'appuient sur les travaux de Bruce et al. sur les groupes de types [BV99, BSvGF03]. En effet un type de modèles est un ensemble de types objets et les contraintes liées à l'invariance des types de champs se posent également lorsque l'on cherche à spécialiser un type de modèles. Steel et al. redéfinissent donc l'opérateur de correspondance de classes de Bruce et al. pour l'adapter aux classes MOF et prendre en compte leurs spécificités.

4.4.2.1 Correspondance de classes MOF

La correspondance de classes MOF définie par Steel et al. compare les champs et opérations de deux classes MOF pour définir si leurs instances sont substituables dans le cadre d'une relation de sous-typage de types de modèles. Cette relation est formalisée dans la définition 4.4 [Ste07].

Pour qu'une classe C' corresponde à une classe C, i.e., pour que les instances de C' puissent être substituées aux instances de C dans le cadre du typage de modèles, C' doit être instantiable si C l'est également (cf. Définition 7.1.i), dans le cas contraire une transformation pourrait essayer d'instancier la classe abstraite C' provoquant une erreur à l'exécution. Pour chaque

champ et pour chaque opération de C, C' doit posséder un champ ou une opération correspondant, hérité ou non (cf. Définition 7.1.ii et Définition 7.1.iii), afin qu'une transformation de modèles puisse appeler les mêmes champs et opérations sur les éléments d'un modèle, quelque soit son type.

Un champ a' correspond au champ a si il possède le même nom (cf iii(a)) et si son type est covariant par rapport, ou correspond, au type de a (cf iii(e)). Il doit également avoir des multiplicités correspondantes. Steel et al. autorisent les multiplicités à être restreintes dans un sens ensembliste (cf iii(f) et iii(g)), i.e., la borne inférieure de a' peut être supérieure à celle de a et la borne supérieure de a' peut être inférieure à celle de a. Certains attributs de a' doivent être égaux (composition iii(c) et unicité iii(i)) à ceux de a. Si a est ordonné, ou en lecture seule, a' doit l'être également (cf iii(d) et iii(b)). Enfin si a possède un champ opposé (opposite en anglais), a' doit posséder un champ opposé du même nom.

Une opération op' correspond à une opération op si elle possède le même nom (cf ii(a)), si son type de retour est covariant par rapport, ou correspond, au type de retour de op (cf ii(b)) et si elle possède un paramètre p' correspondant pour chaque paramètre p de op (cf ii(c)).

Les contraintes pour qu'un paramètre p' corresponde à un paramètre p sont équivalentes aux contraintes pour les attributs (multiplicités, unicité, composition, ordre), à l'exception du type de p' qui ne doit pas être covariant mais contravariant par rapport au type de p.

Définition 4.4. ***(Correspondance de classe MOF proposée par Steel* et al.*)*** La classe MOF T' correspond à T (noté $T'<\#T$) ssi :

- i $T'.isAbstract \Rightarrow T.isAbstract$
- ii $\forall op \in T.ownedOperation, \exists S' \in SuperClasses(T')$ tel que $\exists op' \in S'.ownedOperation$ and :
  - ii(a) $op.name = op'.name$
  - ii(b) $op'.type<\#op.type \vee op.type<:op'.type$
  - ii(c) $p \in op.ownedParameter \Leftrightarrow p' \in op'.ownedParameter$ tel que :
    - i $\exists U' \in SubClasses(p'.type)$ tel que $U'<\#p.type \vee p.type<:p'.type$
    - ii $p.rank = p'.rank$
    - iii $p.lower \leq p'.lower$
    - iv $p.upper \geq p'.upper$
    - v $p.isUnique = p'.isUnique$
    - vi $p.isOrdered \Rightarrow p'.isOrdered$
- iii $\forall a \in T.ownedAttribute, \exists S' \in SuperClasses(T')$ tel que $\exists a' \in S'.ownedAttribute$ tel que :
  - iii(a) $a.name = a'.name$
  - iii(b) $a'.isReadOnly \Rightarrow a.isReadOnly$
  - iii(c) $a.isComposite = a'.isComposite$
  - iii(d) $a.isOrdered \Rightarrow a'.isOrdered$
  - iii(e) $a'.type<\#a.type \vee a'.type<:a.type$
  - iii(f) $a.lower \leq a'.lower$
  - iii(g) $a.upper \geq a'.upper$
  - iii(h) $a.opposite \neq void \Rightarrow a'.opposite \neq void \wedge a.opposite.name = a'.opposite.name$
  - iii(i) $a.isUnique = a'.isUnique$

```
package graph {
  class Graph {
    attribute nodes : Node[0..*]
    attribute edges : Edge[0..*]
  }
  class Node {
    reference outEdges : Edge[0..*]
    reference inEdges : Edge[0..*]
  }
  class Edge {
    reference source : Node[1..1]
    reference target : Node[1..1]
  }
}

modeltype MTGraph {
  graph::Graph,
  graph::Node,
  graph::Edge
}
```

```
package coloredgraph {
  class ColoredGraph {
    attribute nodes : ColoredNode[0..*]
    attribute edges : Edge[0..*]
  }
  class ColoredNode {
    reference outEdges : Edge[0..*]
    reference inEdges : Edge[0..*]
    attribute color : String
  }
  class Edge {
    reference source : ColoredNode[1..1]
    reference target : ColoredNode[1..1]
  }
}

modeltype MTColoredGraph {
  coloredgraph::ColoredGraph,
  coloredgraph::ColoredNode,
  coloredgraph::Edge
}
```

FIGURE 4.3 – Deux types de modèles `MTGraph` et `MTColoredGraph` en Kermeta tels que `MTColoredGraph` <:`MTGraph`.

Où *SuperClasses*(*T*) est l'ensemble de toutes les super-classes de *T*, et *SubClasses*(*T*), l'ensemble de toutes ses sous-classes, tous deux incluant *T*.

A partir de cette relation de correspondance, Steel et al. définissent qu'un type de modèles *MT'* est sous-type d'un type de modèles *MT* si pour chaque classe MOF *C* de *MT* il existe une classe MOF correspondante *C'* dans *MT'* telle que *C'*<#*C* :

Définition 4.5. ***(Sous-typage de types de modèles (Steel* et al. *[SJ07]))*** "Model Type *MT'* is a subtype of another model type *MT* (denoted *MT'*<:*MT*) iff for each class *C* in *MT*, there is one and only one corresponding class *C'* in *MT'* such that every property *p* and operation *op* in *MT.C* matches in *MT'.C'* respectively with a property *p'* and an operation *op'* with parameters of the same type as in *MT.C*."

4.4.2.2 Implémentation dans Kermeta

Les relations de correspondance entre classes MOF et de sous-typage de types de modèles ont été implémentées par Steel et al. dans le système de types de Kermeta [3]. Elles permettent de déclarer des classes paramétrées par un type de modèles réutilisables sur des modèles issus de différents métamodèles. La Figure 4.3 présente un exemple de code Kermeta comportant deux types de modèles `MTGraph` et `MTColoredGraph` tels que `MTColoredGraph` <:`MTGraph`. Les classes sont déclarées dans un package de manière habituelle. Les mots-clé `attribute` et `reference` désignent respectivement un champ composite et un champ simple. Les valeurs entre crochets indiquent les multiplicités des champs (`[0..*]` indiquant un ensemble potentiellement infini et `[1..1]` un élément obligatoire). Les types de modèles sont déclarés séparément et référencent les classes qui les composent.

La figure 4.17 présente un code Kermeta réutilisable sur tous les sous-types du type de modèles `MTGraph` sous la forme d'une classe paramétrée par un type de modèles `MT`. `MT` est borné par `MTGraph`, ce qui implique qu'on ne peut pas passer un type de modèles qui n'est pas sous-type de `MTGraph` comme paramètre de la classe `GraphTransformer`. Le paramètre de type

3. http://www.kermeta.org

Listing 4.17 – Classe paramétrée par un type de modèles et réutilisable sur tous les sous-types du type de modèles `MTGraph`.

```
class GraphTransformer[MT <: MTGraph] {
  operation connect(n1 : MT::Node, n2 : MT::Node): MT::Edge is do
    var e : MT::Edge init MT::Edge.new
    e.source := n1
    e.target := n2
    result := e
  end
}
```

`MT` est ensuite utilisé pour déclarer les types chemins-dépendants `MT::Node` et `MT::Edge`. La valeur réelle de ces types dépend du paramètre passé à la classe `GraphTransformer`.

Ainsi, si l'on passe le type de modèles `MTColoredGraph`, le type chemin-dépendant `MT::Node` sera résolu comme le type réel `coloredgraph::ColoredNode`. Le code de l'opération `connect` est donc réutilisable entre différents types de modèles, grâce à la relation de sous-typage. La relation de sous-typage est inférée statiquement, à l'usage : quand l'utilisateur écrit le code `GraphTransformer[MTColoredGraph]`, le système de types compare les deux types de modèles (le paramètre réel, `MTColoredGraph` et la borne `MTGraph`). Si il existe une relation de sous-typage entre les deux, le fragment de code est valide, dans le cas contraire une erreur est signalée.

4.4.3 Extension aux hétérogénéités structurelles

Moha et al. ont proposé une extension au typage de modèles de Steel et al. afin de prendre en compte des adaptations de type de modèles, et donc d'autoriser une relation de sous-typage entre deux types de modèles présentant des hétérogénéités structurelles [MMBJ09, SMM+12]. En s'appuyant sur un exemple de refactoring entre trois types de modèles de langages orientés-objet (celui de Java, celui d'UML et celui du MOF), Moha et al. listent plusieurs types d'hétérogénéités structurelles :

- les éléments des types de modèles (e.g., opérations et champs) peuvent avoir des noms différents (e.g., la notion de champ porte le nom de `Property` ou de `Variable` selon le type de modèles) ;
- les types d'éléments peuvent être différents (e.g., le champ visibilité est typé par une énumération : private, protected, public ou par une chaîne de caractères) ;
- il peut y avoir des éléments supplémentaires ou manquants d'un type de modèles à l'autre (e.g., la hiérarchie de classes de `ClassDefinition` inclut moins de classes dans le type de modèles de Java) ;
- les champs opposés peuvent être absents ;
- les relations entre classes peuvent être exprimées différemment (la classe `Operation` n'est pas directement accessible depuis `Class` dans le type de modèles de Java).

Ces hétérogénéités posent problème lorsque l'on tente d'établir des relations de sous-typage entre ces types de modèles en utilisant la relation de sous-typage de Steel et al. Moha et al. identifient deux limitations fortes de la relation de sous-typage. Ces deux contraintes sont la contrainte d'égalité des noms et ce que Moha et al. nomment la "conformité structurelle stricte", i.e., l'impossibilité pour une sous-classe de correspondre à une classe. Dans le cas de trois classes, C, C' et SC' telles que SC'<#C et que SC' est une sous-classe de C', La "conformité structurelle stricte" interdit la substitution de C' à C. En effet, dans le cas général, il n'est pas possible d'assurer que lors d'une substitution, on obtiendra toujours un objet instance de

SC' possédant tous les champs et opérations attendus. Moha et al. proposent de relâcher la définition de la relation de sous-typage comme suit, afin d'autoriser de telles substitutions :

Définition 4.6. ***(Sous-typage de types de modèles (Moha* et al. *[MMBJ09]))*** "Model Type *MT'* is a subtype of another model type *MT* (denoted *MT'*<:*MT*) iff for each class *C* in *MT*, there is one and only one corresponding class *or subclass C'* in *MT'* such that every property *p* and operation *op* in *MT.C* matches in *MT'.C'* respectively with a property *p'* and an operation *op'* with parameters of the same type as in *MT.C*."

Cette définition lève la contrainte liée à la "conformité structurelle stricte" par l'extension de la correspondance aux sous-classes. Ceci permet à une classe *C'* de correspondre à une classe *C* si *C'* ou l'une de ses sous-classes possède un champ correspondant pour chaque champ de *C*.

La contrainte liée aux noms est relâchée manuellement en combinant deux fonctions de Kermeta : les aspects et les champs dérivés.

Les aspects permettent d'étendre un métamodèle existant avec de nouveaux éléments structurels (classes, opérations et champs) par introduction statique [MFJ05]. Ce mécanisme est proche du mécanisme de classe ouverte (open-class en anglais) [CLCM00], c'est à dire qu'il permet de modifier une classe sans modifier sa déclaration initiale, en déclarant de nouveaux champs et opérations dans un autre fichier, l'aspect. Le résultat du tissage (i.e., de la combinaison de la classe et de ses aspects) est donc une classe qui possède tous les champs et opérations initialement déclarés plus les champs et les opérations déclarés dans son ou ses aspects.

Les champs dérivés sont des champs dont la valeur n'est pas stockée mais calculée (dérivée) par leurs accesseurs. Un champ dérivé contient donc un corps, comme le font les opérations, et peut être consulté en lecture et en écriture. Grâce aux champs dérivés, il est possible de calculer la valeur d'un champ sur la base des valeurs d'autres champs appartenant à la même classe.

En ajoutant des champs dérivés et des opérations à une classe MOF par aspect, il est possible de faire correspondre deux classes présentant des hétérogénéités structurelles avant le tissage des aspects.

4.5 Limites actuelles du typage de modèles

Le typage de modèles défini par Steel et al. et étendu par Moha et al. permet de considérer les modèles comme des entités de première classe typées, qu'il est possible de stocker dans une variable, de passer en paramètre à une opération ou d'utiliser comme valeur de retour d'une opération. La relation de sous-typage entre types de modèles introduit le polymorphisme de sous-type au niveau des modèles, permettant de réutiliser des transformations de modèles entre différents langages de modélisation dédiés.

Cependant, les relations entre les différents concepts pris en compte dans le typage de modèles (types de modèles, métamodèles, transformations de modèles) ne sont pas définies clairement. Notamment, l'interface et l'implémentation des langages de modélisation dédiés ne sont pas séparées, mais mêlées à la fois au sein des types de modèles et des métamodèles. De plus, il n'y a pas, dans l'approche proposée par Steel et al. de séparation entre la modélisation in-the-small et la modélisation in-the-large.

Par ailleurs, les définitions du sous-typage, que ce soit celle de Steel et al. (définition 4.5) ou celle de Moha et al. (définition 4.6), ne sont pas sûres : elles autorisent des correspondances "illégales" entre classes MOF. Ces correspondances illégales peuvent provoquer des erreurs à l'exécution, mais pourraient être détectées à partir des informations contenues par les classes MOF.

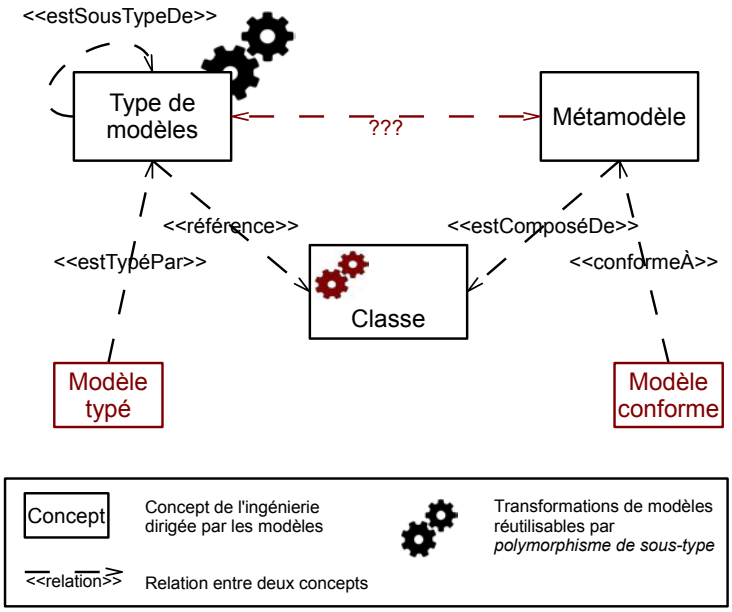

FIGURE 4.4 – Transformations de modèles réutilisables par *polymorphisme de sous-type*.

4.5.1 Définition du typage de modèles

Steel et al. donnent deux définitions pour les types de modèles (cf. Définitions 4.2 et 4.3). La première suggère qu'un modèle n'a qu'un seul type, mais que celui-ci change au cours des modifications du modèle : "the type of a model is the set of the object types of all the contained objects". Cependant, cette définition va à l'encontre de celle du sous-typage, qui autorise un modèle à être typé par plusieurs types. La seconde définition donnée par Steel et al. concerne uniquement la structure d'un type de modèles (un ensemble de classe MOF), sans préciser la relation qui unit un modèle à son type. L'implémentation Kermeta semble s'appuyer uniquement sur la seconde définition, puisqu'il est possible de créer un type de modèles à partir de n'importe quel ensemble de classes Kermeta.

Un type de modèles est donc formé de classes. Cependant la relation de typage, au lieu de remplacer la relation de conformité, est définie en parallèle de celle-ci. Chacune des classes d'un type de modèles appartient donc également à un métamodèle. De plus, dans Kermeta les types de modèles sont instanciables, le résultat de leur instanciation étant un modèle.

Un modèle peut donc être créé à partir d'un type de modèles ou à partir d'un métamodèle. Néanmoins, rien ne garantit que les différentes classes d'un type de modèles appartiennent au même métamodèle. Un modèle typé par un type de modèles n'est donc pas forcément conforme à un métamodèle. Ainsi, le typage de modèles mène à la définition de deux notions de modèles : les modèles créés depuis les types de modèles, et les modèles créés depuis les métamodèles. Les deux différents modèles en Figure 4.4 représentent cette disparité.

Dans l'approche proposée par Steel et al., les types de modèles et les modèles sont manipulés au sein de classes et d'objets. Les transformations de modèles sont représentées sous la forme d'opérations appartenant à des classes paramétrées par un type de modèles. Les variables

```
class InfiniteNode {
  field edges : Edge[0..*]
  [...]
}
```

```
class FiniteNode {
  field edges : Edge[1..5]
  field name : String[1..1]
  [...]
}
```

FIGURE 4.5 – Deux classes `InfiniteNode` et `FiniteNode` tels que `FiniteNode <# InfiniteNode` d'après la définition 4.4.

contenant des modèles (i.e., les variables typées par des types de modèles) sont déclarées dans des classes et des opérations. Il n'y a donc pas de séparation claire entre la modélisation in-the-small (les objets, les classes et les champs) et la modélisation in-the-large (les types de modèles).

La Figure 4.4 représente cette situation : les transformations de modèles sont définies en fonction des types de modèles (les engrenages à l'extérieur du type de modèles), mais elles sont définies au sein de classes (les engrenages à l'intérieur de la classe). De plus, la relation entre métamodèle et type de modèles n'est pas définie clairement et la relation de typage cohabite avec la relation de conformité.

Cette absence de séparation empêche de traiter et de raisonner de manière indépendante sur les deux préoccupations que sont la définition et l'outillage des concepts pris en compte par un langage de modélisation dédié (modélisation in-the-small) et la définition et l'outillage des relations entre les modèles de plusieurs langages, telles que les transformations de modèles (modélisation in-the-large).

4.5.2 Correspondances illégales

Dans la définition 4.4, Steel et al. autorisent les multiplicités à être restreintes du super-type de modèles au sous-type de modèles. La figure 4.5 présente deux classes `InfiniteNode` et `FiniteNode` telles que `FiniteNode <# InfiniteNode`. Cependant la substitution d'objets instances de `InfiniteNode` par des objets instances de `FiniteNode` n'est pas sûre. En effet une transformation de modèles écrite par rapport à la classe `InfiniteNode` peut modifier le champ `edges` en en enlevant tous les éléments ou en ajoutant plus de cinq éléments. Ces deux manipulations sont légales et ne causent pas d'erreurs sur une instance de `InfiniteNode`. Cependant sur une instance de `FiniteNode`, ces deux manipulations provoqueraient des erreurs à l'exécution. La première en provoquant un underflow (moins d'éléments que nécessaire), la seconde un overflow (trop d'éléments).

De plus, la relation de correspondance ne prend pas en compte les champs obligatoires (mandatory en anglais) de la classe "correspondante" (e.g., `FiniteNode`) si ils n'apparaissent pas dans la classe "correspondue" (e.g., `InfiniteNode`). Une transformation de modèles écrite par rapport à la classe `InfiniteNode` pourrait chercher à instancier un nouvel objet. Cependant si la classe réelle est `FiniteNode` et non `InfiniteNode`, le champ `name` ne sera pas initialisé et l'objet obtenu ne sera pas conforme à sa classe (la classe `FiniteNode` exige que le champ `name` ne soit pas vide).

La définition relâchée proposée par Moha et al. (définition 4.6) pose un autre problème : en autorisant la correspondance au niveau des sous-classes, les transformations de modèles ne sont pas garanties de trouver certains champs. La figure 4.6 présente deux classes `RGBColoredNode` et `ColoredNode` telless que `ColoredNode <# RGBColoredNode`. Cette correspondance est autorisée car la sous-classe `SpecificColor` de la classe `Color` correspond à la classe `RGBColor`. Cependant une transformation de modèles cherchant à accéder au champ `red` de l'objet ré-

```
class RGBColoredNode {
  field color : RGBColor
  [...]
}

class RGBColor {
  field name : String
  field red : Integer
  field green : Integer
  field blue : Integer
}
```

```
class ColoredNode {
  field color : Color
  [...]
}

class Color {
  field name : String
}

class SpecificColor extends Color {
  field red : Integer
  field green : Integer
  field blue : Integer
}
```

FIGURE 4.6 – Deux classes `RGBColoredNode` et `ColoredNode` telles que `ColoredNode` <# `RGBColoredNode` d'après la définition 4.6.

férencé par le champ `color` n'est pas garantie de trouver ce champ. Si `color` pointe sur une instance de `Color` et non une instance de `SpecificColor`, le code `.color.red` causera une erreur à l'exécution.

4.5.3 Conclusion

Le typage de modèles offre à l'ingénierie dirigée par les modèles certaines facilités fournies par les systèmes de types dans les langages de programmation orientés-objet. Cependant, certaines limites subsistent afin d'offrir le niveau de flexibilité et de sécurité que l'on peut trouver dans les langages orientés-objet tels que Java ou Scala. Notamment, la séparation de l'interface à travers laquelle un modèle est manipulable et de l'implémentation à partir de laquelle il est créé, et la séparation de la modélisation in-the-small et de la modélisation in-the-large ne sont pas prises en compte.

C'est pourquoi, dans la suite de cette thèse, nous proposons une clarification et une extension des types de modèles, ainsi que des relations de typage de modèles et de sous-typage entre types de modèles. Ces extensions prennent en compte la séparation de l'interface et de l'implémentation d'un langage de modélisation dédié au travers des types de modèles d'une part et des métamodèles de l'autre, et la séparation de la modélisation in-the-small et de la modélisation in-the-large.

Notre relation de typage offre un socle pour de nombreuses facilités d'ingénierie, supportées par plusieurs relations de sous-typage. Nous proposons de remplacer la relation de conformité, trop contraignante, par cette relation de typage afin de faciliter la définition et l'outillage des langages de modélisation dédiés.

Deuxième partie

Facilités de typage pour l'ingénierie des langages

Chapitre 5

Présentation de l'approche

Nous proposons de remplacer la relation de conformité, trop contraignante, par une relation de typage des modèles. La relation de typage de modèles, et par extension les types de modèles, fournissent une base sur laquelle il est possible de définir des systèmes de types orientés-modèle, à même de supporter les mêmes facilités que les systèmes de types orientés-objet. Pour pouvoir implémenter de tels systèmes de types au sein d'environnements de développement de langages de modélisation dédiés, il est nécessaire de fournir une base formelle définissant les types de modèles, la relation de typage et les relations entre ces types de modèles.

Ce chapitre commence par motiver le besoin de tels systèmes de types orientés-modèle en rappelant les limites de l'ingénierie dirigée par les modèles et des facilités qu'elle propose (cf. Section 5.1). Nous présentons ensuite la contribution de cette thèse : le métalangage METAL, qui regroupe les concepts et relations nécessaires pour supporter les systèmes de types orientés-modèle (cf. Section 5.2). Ce métalangage s'articule autour de deux axes : la séparation de l'interface et de l'implémentation des modèles (cf. Section 5.2.1) ; et la séparation de la modélisation in-the-small et de la modélisation in-the-large (cf. Section 5.2.2).

5.1 Limites des approches de métamodélisation actuelles

Malgré les avancées de l'ingénierie dirigée par les modèles pour faciliter la définition et l'outillage des langages dédiés (e.g., définition de la syntaxe abstraite sous forme de graphe de classes, outils de génération de syntaxes concrètes et d'analyseurs syntaxiques), les approches de métamodélisation existantes présentent encore certaines limites. La relation de conformité, en limitant un modèle à n'être lié qu'à un métamodèle, est un frein à la réutilisation de la structure et du comportement entre langages de modélisation dédiés. De plus, les différentes facilités proposées jusqu'ici (e.g., facilités de réutilisation de transformations de modèles, manipulation des modèles comme entités de première classe) manquent d'un cadre unificateur permettant de les combiner et de les comparer. Nous pensons que le typage de modèles peut fournir un tel cadre, mais ses premières incarnations souffrent de limites, notamment parce qu'elles s'appuient sur la relation de conformité.

5.1.1 Relation de conformité

La relation de conformité, qui lie un modèle à un métamodèle, permet de déterminer si un modèle est un mogram valide d'un langage de modélisation dédié. C'est donc un concept central de l'ingénierie dirigée par les modèles, qui est utilisé comme une relation de typage

pour, par exemple, autoriser ou non le passage d'un modèle à une transformation de modèles, ou le chaînage de transformations de modèles.

Cependant, la relation de conformité s'appuie sur les classes de la syntaxe abstraite d'un langage de modélisation dédié. La relation de conformité dépend donc de l'implémentation des langages de modélisation dédiés, puisqu'elle s'appuie sur la relation d'instanciation entre éléments de modèles (i.e., objets) et classes. Un objet ne peut être instance que d'une classe, qui elle-même n'appartient qu'à un métamodèle. Un modèle étant composé d'objets, la relation de conformité limite un modèle à être lié à un unique métamodèle, qui contient les classes de tous les éléments du modèle. Un modèle n'est donc valide que par rapport à un unique langage de modélisation dédié, qui a permis de le construire. En interdisant à un modèle d'être valide vis-à-vis de plusieurs langages de modélisation dédiés, la relation de conformité empêche toute forme de polymorphisme : un modèle n'a qu'une "forme", celle définie par le métamodèle auquel il se conforme (cf. Section 3.2.4).

5.1.2 Facilités d'ingénierie

L'ingénierie dirigée par les modèles a proposé plusieurs approches pour faciliter la définition et l'outillage de langages de modélisation dédiés et la manipulation de modèles. Ces facilités incluent la définition de transformations de modèles génériques, qui s'appuie sur des entités différentes des métamodèles (cf. Section 3.3.2.1). Elles incluent également les transformations de modèles composables (cf. Section 3.3.2.1). Ces deux approches permettent de réutiliser des transformations de modèles soit entre différents métamodèles, soit au sein de transformations de modèles plus complexes. Enfin, la modélisation in-the-large propose de réifier les relations entre modèles afin de faciliter leur manipulation en s'abstrayant des détails de la modélisation in-the-small (cf. Section 3.3.1).

Cependant, ces différentes facilités (transformations de modèles génériques et composables, réification des relations entre modèles) sont déconnectées les unes des autres. Elles manquent notamment d'un cadre unifié qui permette de les comparer et de les combiner (cf. Section 3.3.3).

5.1.3 Typage de modèles

Le typage de modèles, tel que défini par Steel et al. [Ste07] puis par Moha et al. [MMBJ09], supporte une relation de typage de modèles et une relation de sous-typage qui permettent d'aller au-delà des limitations du sous-typage objet. Cette relation de sous-typage autorise la substituabilité de groupes d'objets instances de classes MOF, et donc de modèles. Ceci permet la réutilisation de transformations de modèles par polymorphisme de sous-type [1].

Cependant, les types de modèles s'appuient sur les classes et sont instanciables, ce qui mène à des situations où un modèle peut être typé mais n'être conforme à aucun métamodèle (cf. Section 4.5). En ne séparant pas clairement l'interface d'un modèle de son implémentation, le typage de modèles donne deux définitions différentes de la notion de modèle : d'un côté les modèles conformes à un métamodèle, de l'autre les modèles typés par un ou plusieurs types de modèles. De plus, les transformations de modèles définies sur les types de modèles et les variables typées par des types de modèles sont contenues par des classes, mêlant ainsi modélisation in-the-large et modélisation in-the-small (cf. Section 4.5).

1. Ce polymorphisme de sous-type au niveau des modèles est permis grâce au polymorphisme de groupes de types au niveau des objets.

5.2 METAL : un métalangage supportant une famille de systèmes de types orientés-modèle

Pour unifier les différentes facilités proposées par l'ingénierie dirigée par les modèles, nous proposons de définir des systèmes de types orientés-modèles. En effet les systèmes de types ont été développés dans les langages de programmation pour supporter un ensemble de facilités d'ingénierie grâce à un socle formel (cf. Sections 4.1 et 4.2). Nous proposons donc de définir des systèmes de types capables de supporter les mêmes facilités pour les modèles et les langages de modélisation dédiés. L'ingénierie dirigée par les modèles étant intrinsèquement liée au paradigme objet, nous nous inspirons des systèmes de types orientés-objet pour amener les facilités de typage au niveau des modèles.

Pour cela, nous utilisons différents concepts et relations, présentés dans la Figure 5.1 selon deux dimensions : la séparation de l'interface et de l'implémentation et la séparation des modélisations in-the-small et in-the-large. Dans ce but, nous avons défini un métalangage pour la modélisation in-the-large nommé METAL (pour MÉTAlangage pour la modélisation in-the-Large)[2]. La syntaxe abstraite de METAL définit les concepts utilisés pour la définition de systèmes de types orientés-modèle : métamodèles et types de modèles, classes et types objets ; et les relations entre ces concepts : relations de sous-typage et d'héritage. Cette syntaxe abstraite est présentée en Figure 5.2 à la fin de ce chapitre et sera détaillée dans les deux chapitres suivants (cf. Chapitre 6 et Chapitre 7).

5.2.1 Séparation des interfaces et des implémentations

Afin de s'abstraire de la relation d'instanciation, sur laquelle s'appuie la relation de conformité et de clarifier les relations entre types de modèles et métamodèles, METAL sépare les concepts d'interface et d'implémentation. METAL fournit les moyens d'exprimer les types objets et les types de modèles séparément des classes et des métamodèles. Les types (objets ou de modèles) fournissent une interface à travers laquelle il est possible de manipuler l'entité (objet ou modèle) qu'ils typent. Les classes et les métamodèles fournissent l'implémentation des objets et des modèles, et permettent de les créer par instanciation. Ainsi, un objet n'a qu'une classe, à partir de laquelle il est créé, mais peut avoir un ensemble de types. De même, un modèle n'a qu'un métamodèle, mais possède plusieurs types. Il n'est par contre pas possible de créer un modèle à partir d'un type de modèles, évitant ainsi la situation provoquée par le typage de modèles de Steel et al.

Parmi les différents types d'un objet ou d'un modèle, nous nous intéressons particulièrement au type exact, qui est le type le plus précis de l'objet ou du modèle. Le type exact d'un modèle expose l'ensemble des champs, des opérations et des éléments accessibles du modèle. Nous nous appuyons sur le type exact d'un modèle pour calculer l'ensemble des types qui typent un modèle, i.e., pour calculer la relation de typage de modèles. Cette relation de typage autorise un modèle à être typé par un ensemble de types de modèles. Cet ensemble est composé du type exact du modèle et de tous ses super-types.

5.2.2 Séparation des modélisations *in-the-small* et *in-the-large*

Le terme modélisation in-the-large désigne les activités de modélisation aux niveaux des entités "macroscopiques" de l'ingénierie dirigée par les modèles (modèles, métamodèles, rela-

2. Notre métalangage METAL n'est pas le premier à porter ce nom. Un métalangage nommé Metal était déjà utilisé au sein de CENTAUR [KLM+83, CIK90], l'un des premiers environnements de développement de langages. Merci au professeur Olivier Ridoux pour avoir pointer cette curieuse coïncidence.

tions entre modèles, etc.) [BJT05]. Afin de faciliter ces activités, il est nécessaire de s'abstraire des détails des entités "microscopiques" (éléments de modèles et relations entre ces éléments) et de fournir une vision de plus haut niveau à l'utilisateur. La possibilité de manipuler les modèles comme des entités de première classe plutôt qu'à travers leurs éléments faciliterait notamment la définition de relations entre modèles.

Parmi d'autres facilités, les systèmes de types orientés-objet proposent de manipuler un ensemble de données et d'opérations liées à ces données au travers du concept d'objet. Nous proposons de faire de même pour le concept de modèle. Ainsi METAL permet de manipuler l'ensemble des données d'un modèle (éléments du modèle et relations avec d'autres modèles) et les transformations liées à ces données directement depuis un modèle.

Pour cela, les types de modèles prennent en compte les éléments qui peuvent composer un modèle, sous la forme de types objets et de classes, mais également les relations entre modèles, sous la forme de champs et d'opérations de modèles. Ainsi, un modèle contient des éléments de modèles (i.e., des objets), mais possède également des champs qui le lient à d'autres modèles et des opérations de modèles, i.e., des transformations de modèles attachées aux modèles, à la manière des opérations attachées aux objets. Ces champs et ces opérations de modèles sont définis par le métamodèle dont le modèle est instance, tout comme les champs et opérations attachés à un objet sont définis par sa classe. Ainsi, les relations entre modèles sont accessibles directement depuis un modèle, à travers les champs et opérations de ce modèle.

La couche de modélisation in-the-large de METAL dépend de la couche de modélisation in-the-small : les métamodèles sont composés de classes et les types de modèles de types objets, de la même manière qu'un modèle est composé d'éléments de modèles (i.e., d'objets) (cf. Figure 5.1). À l'inverse, la couche de modélisation in-the-small ne dépend pas de la couche de modélisation in-the-large. Il est possible de définir ou de manipuler un concept in-the-small (e.g., objet, classe, type objet) sans connaître ou utiliser la couche de modélisation in-the-large.

5.2.3 METAL : un MÉTAlangage pour la modélisation *in-the-Large*

La Figure 5.1 présente une vue d'ensemble des concepts et relations pris en compte dans METAL. Ces concepts et relations sont représentés selon deux axes : la séparation interface - implémentation et la séparation modélisation in-the-small - modélisation in-the-large.

Un *métamodèle* fournit l'implémentation de tous les modèles qu'il sert à instancier. Cette implémentation comprend les classes des éléments des modèles, mais également les transformations qu'il est possible d'appeler sur ces modèles sous la forme d'opérations de modèles. L'instanciation d'un métamodèle consiste à réserver en mémoire l'espace nécessaire pour les champs d'un modèle.

Les *transformations de modèles* sont définies par rapport aux types de modèles, mais contenues par les métamodèles. Tout comme les opérations, qui sont définies par rapport à des types objets mais au sein de classes, les transformations de modèles sont définies par rapport à des types de modèles (engrenages à l'extérieur du type de modèles) et contenues par un métamodèle (engrenages à l'intérieur du métamodèle). Nous les désignons alors comme des opérations de modèles. Elles agissent sur les modèles typés par des types de modèles donnés, et non sur des modèles construits à l'aide de métamodèles donnés.

Une relation d'*héritage* lie deux métamodèles : le super-métamodèle et le sous-métamodèle. Les relations d'héritage permettent de réutiliser la structure (classes) et le comportement (opérations et opérations de modèles) d'un métamodèle à un autre.

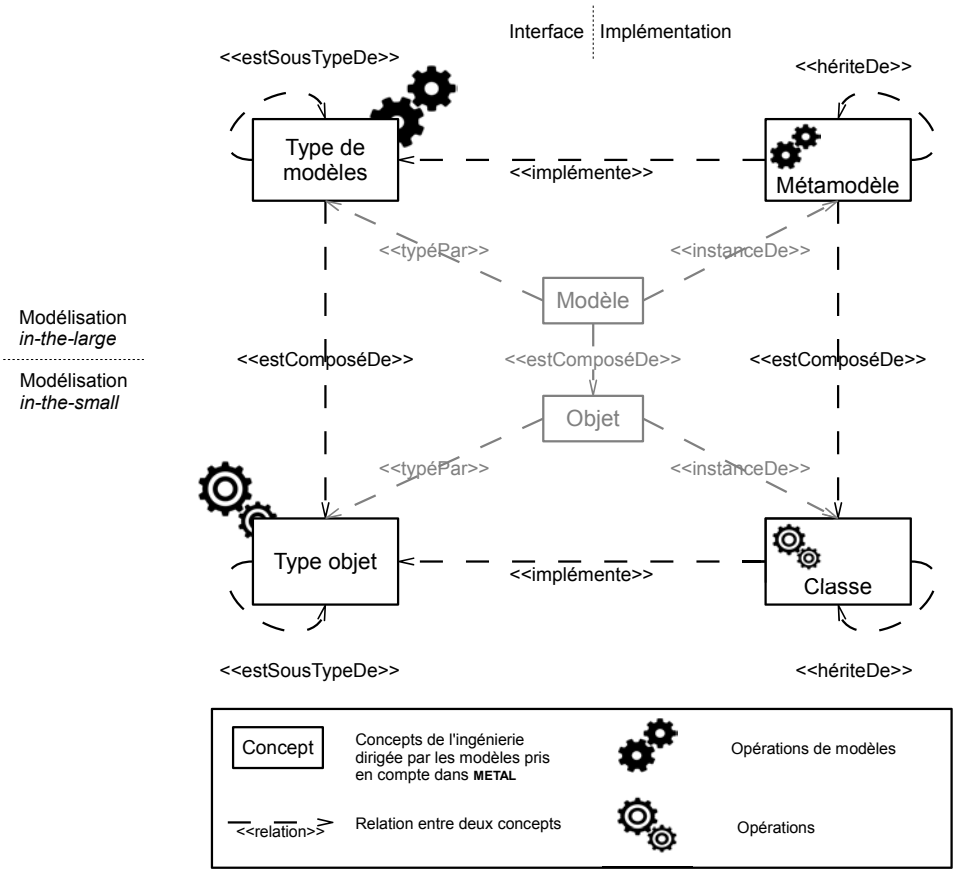

FIGURE 5.1 – Concepts et relations pris en compte dans METAL pour la définition de systèmes de types orientés-modèle.

Un *type de modèles* expose l'interface à travers laquelle il est possible de manipuler les modèles qu'il type. Un type de modèles est donc composé de types objets, qui eux-mêmes exposent l'interface des éléments de modèles. Un métamodèle implémente un ou plusieurs types de modèles, fournissant l'implémentation correspondant à l'interface que ces types de modèles exposent.

Les relations de *sous-typage* lient deux types de modèles. Une relation de sous-typage entre deux types de modèles détermine la substituabilité des modèles typés par le sous-type aux modèles typés par le super-type.

Un modèle est une instance d'un métamodèle et est typé par un ensemble de types de modèles. Un modèle m est composé d'objets, instances des classes du métamodèle de m et typés par les types objets des types de modèles qui typent m.

La couche de modélisation *in-the-small*, sur laquelle est construite la couche de modélisation in-the-large, s'appuie également sur les notions d'interfaces (i.e., types objets) et d'implémentation (i.e., classes). Un objet est instance d'une classe et est typé par un ensemble de types objets. Les classes peuvent être liées par des relations d'héritage et les types objets par des relations de sous-typage.

5.2.4 Une famille de systèmes de types orientés-modèle

En s'appuyant sur les concepts et relations définis par METAL, il est possible d'implémenter plusieurs systèmes de types, supportant différentes facilités. Ainsi le choix des relations de sous-typage et d'héritage implémentées, ainsi que de la manière dont ces relations sont déclarées et vérifiées, ouvre la voie à certaines facilités et peut en empêcher d'autres. Tout en s'appuyant sur les concepts présentés dans METAL, les choix d'implémentation peuvent donc mener à différents systèmes de types au sein de différents environnements de développement de langages de modélisation dédiés. La relation de typage de modèles et les relations entre langages de modélisation dédiés fournissent donc un socle pour définir une famille de systèmes de types orienté-modèles. Chacun de ces systèmes de types supporte différentes facilités de typage, en fonction des choix faits lors de son implémentation.

Un modèle étant typé, et instancié depuis un métamodèle, les systèmes de types qui s'appuient sur les concepts de METAL offrent de manipuler un modèle comme une entité de première classe (i.e., un modèle peut être créé, assigné à une variable, passé en paramètre ou retourné par une opération de modèles). L'implémentation d'une ou plusieurs relations de sous-typage autorise le polymorphisme de sous-type, qui permet de manipuler un modèle à travers différentes interfaces (i.e., types de modèles). L'héritage et l'héritage avec sous-typage peuvent également être autorisés afin de réutiliser le code d'un super-métamodèle pour créer un sous-métamodèle. Les informations contenues par les types de modèles peuvent servir de support à l'auto-complétion dans un éditeur, afin de proposer les champs et opérations de modèles accessibles sur un modèle. Un mécanisme de visibilité peut permettre l'encapsulation de l'état d'un modèle en limitant l'interface exposée par les types de modèles. Enfin, les types de modèles et les relations de sous-typage entre ces types permettent de détecter des erreurs de programmation ou des modifications provoquant de telles erreurs à l'aide d'un vérificateur de types (type checker en anglais) et d'analyses d'impact.

Nous avons implémenté un tel système de types au sein de l'environnement de développement de langages de modélisation dédiés Kermeta. Ce système de types supporte la relation de sous-typage totale isomorphique et une relation d'héritage. L'implémentation de ce système de types et son utilisation sur un exemple provenant de la compilation optimisante sont l'objet de la Partie III.

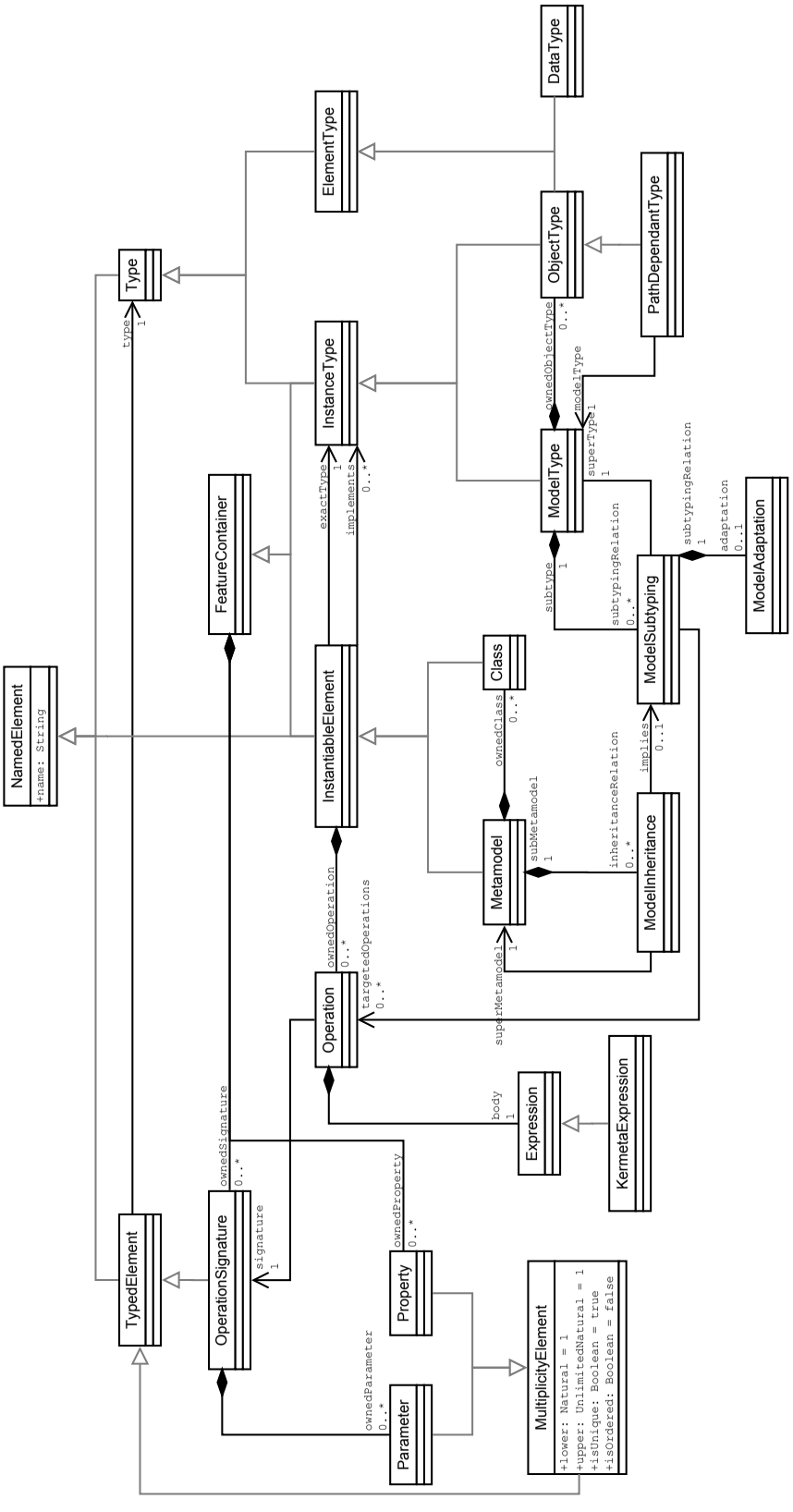

FIGURE 5.2 – Syntaxe abstraite du métalangage METAL.

Chapitre 6

Réification de concepts pour la modélisation *in-the-large*

La relation de conformité entrave la flexibilité de la conception et de l'outillage des langages de modélisation dédiés, en contraignant un modèle à n'être lié qu'à un unique métamodèle. L'ingénierie dirigée par les modèles nécessite une autre relation liant les modèles aux langages de modélisation dédiés. Cette relation doit être moins restrictive que la relation de conformité pour offrir l'accès à des facilités telles que la réutilisation et l'abstraction. C'est pourquoi nous proposons de remplacer la relation de conformité par une relation de typage entre les modèles et les types de modèles.

Les définitions de la relation de typage et des types de modèles que nous proposons peuvent être implémentées au sein d'environnements de développement de langages de modélisation dédiés. La relation de typage entre modèles et types de modèles pose ainsi le socle sur lequel peuvent être construits des systèmes de types orientés-modèle. C'est en effet sur les types que s'appuient l'ensemble des facilités offertes par les systèmes de types : le polymorphisme est fourni au travers de relations entre types, les informations contenues dans un type sont utilisées pour la détection d'erreurs et l'auto-complétion, etc.

Dans ce chapitre, nous présentons un sous-ensemble de notre métalangage METAL permettant d'exprimer des types de modèles. Ces types de modèles permettent de manipuler les modèles en s'abstrayant des concepts de plus bas niveaux (classes, objets). Nous proposons également la définition formelle d'une relation de typage permettant de s'abstraire des contraintes de la relation de conformité.

La Figure 6.1 présente les concepts et relations abordés dans ce chapitre : types de modèles et métamodèles ; types objets et classes ; relations de typage et d'instanciation. L'autre partie de METAL, consacrée aux relations entre langages de modélisation dédiés (sous-typage et héritage) est présentée dans le Chapitre 7.

La Section 6.1 présente le sous-ensemble de METAL dédié à la modélisation in-the-small. Ce sous-ensemble inclut les types objets et les classes. La Section 6.2 introduit ensuite les concepts de METAL dédiés à la modélisation in-the-large : types de modèles et métamodèles. Cette couche de modélisation in-the-large de METAL s'appuie sur la couche de modélisation in-the-small définie en Section 6.1. Enfin la Section 6.3 donne la définition de la relation de typage entre modèles et types de modèles.

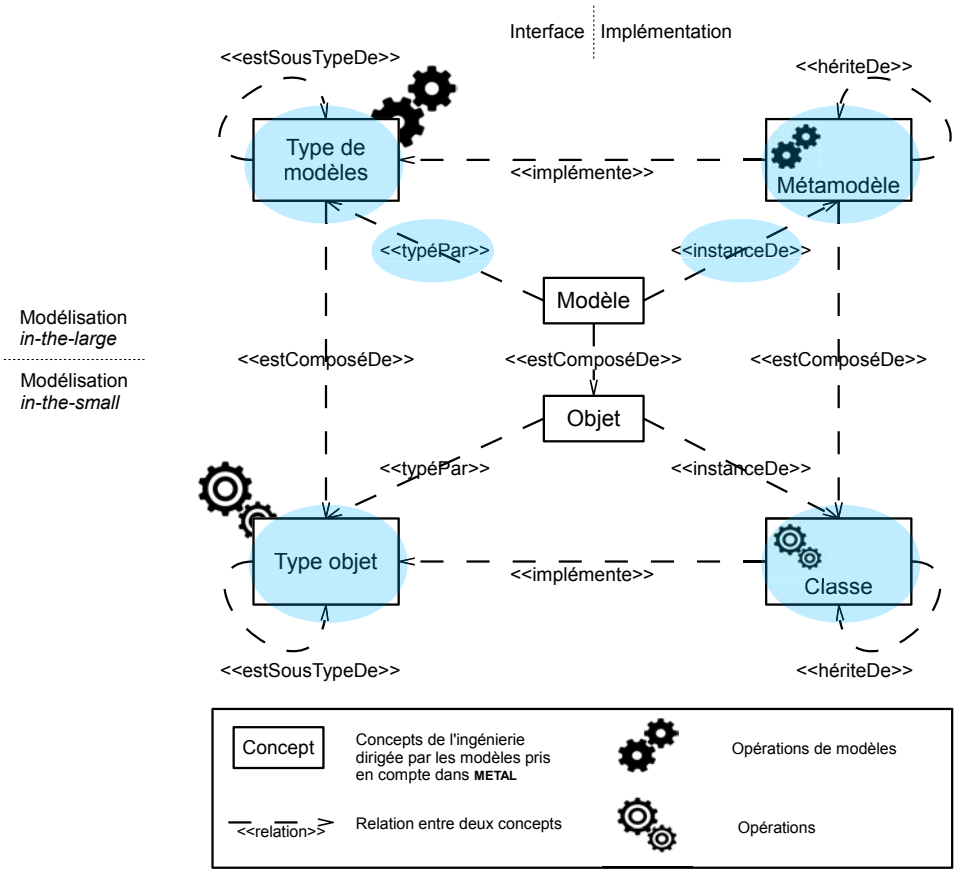

FIGURE 6.1 – Concepts et relations de METAL abordés dans ce chapitre.

6.1 Modélisation *in-the-small*

Cette section présente la couche de modélisation in-the-small de METAL, sur laquelle s'appuie la couche de modélisation in-the-large qui définit les concepts et relations nécessaires à l'implémentation de systèmes de types orientés-modèle. La syntaxe abstraite complète de METAL est présentée en Figure 5.2, à la fin du chapitre précédent.

Pour définir une relation de typage de modèles, il est nécessaire de définir ce qu'est un type de modèles, quelles informations il contient. Un modèle étant composé d'éléments de modèles (i.e., d'objets), un type de modèles fournit des informations sur ces éléments de modèles. Afin de clairement séparer l'interface de l'implémentation d'un élément de modèle, nous définissons deux points de vue sur un élément de modèle : son type (i.e., interface) et sa classe (i.e., implémentation).

Les classes fournissent l'implémentation des types objets, tandis que les types objets exposent la façon dont un élément de modèle peut être manipulé. Nous considérons également deux types objets particuliers : le type exact d'un élément de modèle, qui est le type le plus précis de l'élément de modèle et les types chemin-dépendants, qui sont des types variables. Le

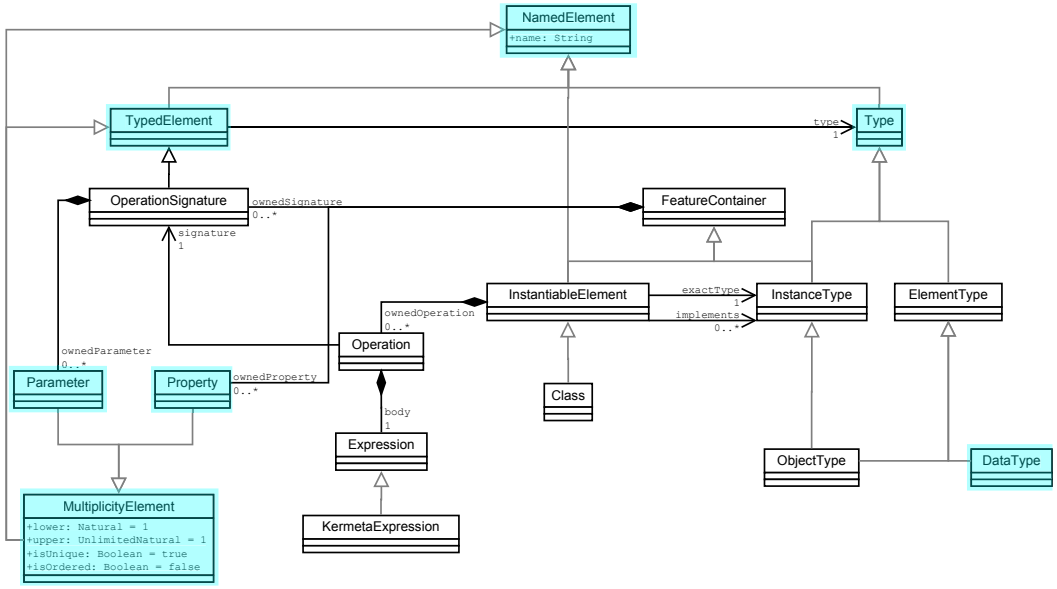

FIGURE 6.2 – Sous ensemble de la syntaxe abstraite du métalangage METAL exposant les concepts communs avec le MOF et Ecore.

type exact d'un objet est utilisé dans le calcul de la relation de typage du modèle qui contient l'objet. Les types chemins-dépendants sont utilisés pour représenter des types dépendants du type de modèles auxquels ils appartiennent.

6.1.1 METAL, MOF et Ecore

METAL ajoute des concepts nécessaires à la modélisation in-the-large aux métalangages standards tels que le Meta-Object Facility (MOF) [OMG06] ou Ecore [SBPM09]. En effet, ces métalangages ne fournissent que les concepts dédiés à la modélisation in-the-small. Il n'y a donc pas de comparaison possible pour ce qui est de l'expression de concepts tels que les métamodèles ou les types de modèles.

Pour ce qui est de la modélisation in-the-small, par contre, il est possible de comparer les concepts apportés par METAL à ceux du MOF ou d'Ecore, tels qu'ils sont présentés en Section 3.1.2.

Certains concepts sont communs à METAL, au MOF et à Ecore. Ainsi, les classes `Type`, `DataType`, `TypedElement`, `NamedElement`, `MultiplicityElement`, `Property`, et `Parameter` de METAL sont équivalentes aux classes du même nom du MOF (cf. Figure 5.2). Leurs attributs (e.g., multiplicité, unicité, ordre, etc. des `MultiplicityElement`) sont les mêmes et portent la même sémantique. Ces concepts sont encadrés sur la Figure 6.2.

Cependant, l'arbre d'héritage est légèrement différent entre METAL et le MOF. Certaines classes ont été ajoutées pour généraliser les associations partagées par les types objets et les types de modèles, ou par les classes et les métamodèles. D'autres classes permettent au contraire de différencier les concepts liés à la modélisation in-the-small de ceux liés à la modélisation in-the-large.

Les principales différences entre METAL et le MOF et Ecore portent sur les classes `Class` et `Operation`. Ces différences sont résumées en Table 6.1.

Le MOF et Ecore ne séparent pas les notions d'interface et d'implémentation. Ainsi, une

TABLE 6.1 – Classes du MOF, d'Ecore et de METAL représentant les interfaces et les implémentations des objets et des opérations.

Concept représenté	Classe du MOF et d'Ecore	Classe de METAL
Types objets (Interface des objets)	`Class`	`ObjectType`
Classes (Implémentation des objets)	`Class`	`Class`
Signatures d'opérations (Interface des opérations)	`Operation`	`OperationSignature`
Opérations (Implémentation des opérations)	-	`Operation` `Expression`

classe (`Class`) d'un de ces métalangages est à la fois utilisée comme type et comme implémentation des objets qui en sont instances. Ce n'est pas le cas dans METAL, où la classe `Class` n'hérite pas de la classe `Type`.

De plus, les classes MOF et Ecore (`Class`) représentent des entités instanciables, mais sans comportement. En effet, les opérations (`Operation`) qu'elles contiennent ne représentent que des signatures d'opérations, sans corps. METAL sépare les opérations de leurs signatures, les premières étant contenues par les classes et les secondes par les types objets. Les opérations de METAL ne sont donc pas de simples signatures, mais bien des opérations possédant un corps exécutable. Pour cela, METAL s'appuie sur les expressions du métalangage Kermeta (cf. `KermetaExpression`).

6.1.2 Types objets

Pour séparer les interfaces et les implémentations des objets d'un modèle, METAL définit un concept de type objet. La figure 6.3 présente un sous-ensemble de la syntaxe abstraite de METAL destinée à exprimer ces types objets. Un type objet (`ObjectType`) expose la structure et le comportement des éléments de modèles. Un type objet se compose d'un ensemble de signatures d'opérations (`OperationSignature`) et d'un ensemble de déclaration de champs (`Property`). Pour cela un type objet hérite de la classe `InstanceType`.

Les signatures d'opérations exposent les paramètres (`Parameter`) attendus (nom, type, multiplicités, etc.), les exceptions levées et le type de retour d'une opération. Ceci permet d'appeler une opération sans connaître son implémentation, et de fournir plusieurs implémentations pour une même signature d'opération. Les champs représentent des associations nommées avec un autre type d'élément de modèle.

Les classes `OperationSignature` et `Property` représentent aussi bien les signatures d'opérations et champs définis dans les types objets, et donc attachés aux objets, que les champs et signatures d'opérations définis dans les types de modèles, et donc attachés aux modèles (cf. Section 6.2.1). C'est pourquoi nous définissons des invariants (i.e., des contraintes) pour garantir que les opérations et champs des objets ne manipulent que des types in-the-small : types objets (`ObjectType`) ou types de "valeurs" (`DataType`). Les types in-the-small sont regroupés grâce à la classe `ElementType`. Les invariants sont présentés en Listing 6.1.

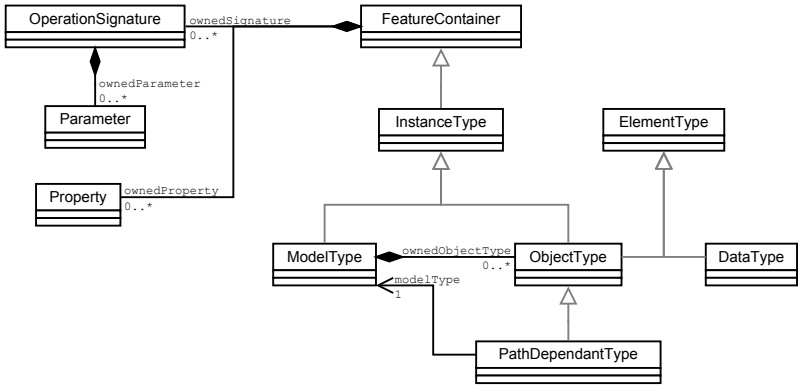

FIGURE 6.3 – Sous-ensemble de la syntaxe abstraite de METAL pour les *types objets*.

Listing 6.1 – Invariants OCL limitant le type des champs, opérations et paramètres de types objets.

```
context ObjectType inv:
  ownedSignature->forall(
    type.oclIsKindOf(ElementType) and
    ownedParameter->forall(type.oclIsKindOf(ElementType))
  )

context ObjectType inv:
  ownedProperty->forall(type.oclIsKindOf(ElementType))
```

6.1.3 Types chemins-dépendants

Les types chemins-dépendants (`PathDependantType`) sont des types objets particuliers. Il s'agit de types objets dont la valeur réelle dépend d'une variable (cf. Section 4.3.2). Dans METAL, cette variable est un type de modèles (cf. `modelType`) qui représente le groupe de types objets auquel le type chemin-dépendant appartient (cf. Section 4.3).

Les types chemins-dépendants permettent notamment de définir des champs et des opérations dont le type varie avec l'héritage de métamodèles (cf. Section 7.2).

6.1.4 Classes et *type exact*

METAL représente l'interface des objets au travers de la classe `ObjectType`. Pour représenter l'implémentation des objets d'un modèle, nous définissons la classe `Class`. La figure 6.4 présente un sous-ensemble de la syntaxe abstraite de notre métalangage destiné à exprimer les classes.

Une classe (`Class`) implémente un ensemble de types objets (cf. association `implements` de la classe `InstantiableElement`), en fournissant une opération (`ownedOperation`) et son corps (`body`) pour chaque signature d'opération des types objets implémentés.

Un élément de modèle peut être typé par plusieurs types objets, mais est instance d'une seule classe, à partir de laquelle il a été créé. L'ensemble des types implémenté par la classe d'un objet est également l'ensemble des types de cet objet.

Une classe ne peut implémenter que des types objets, et pas d'autres types (e.g., des types de modèles). L'invariant OCL présenté en Listing 6.2 garantit qu'une classe n'implémente que des types objets et pas d'autres `InstanceType`.

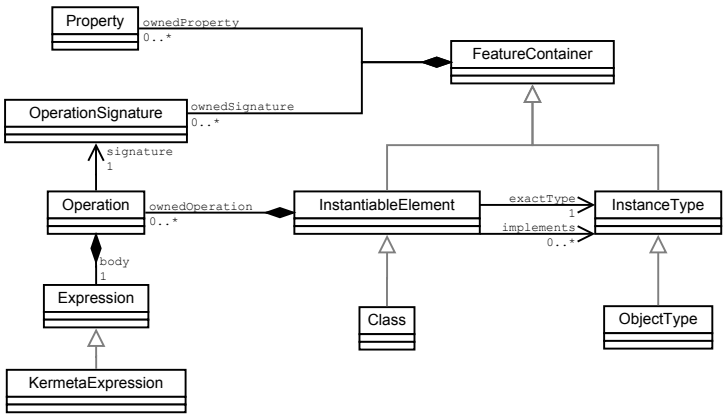

FIGURE 6.4 – Sous-ensemble de la syntaxe abstraite de METAL pour les *classes*.

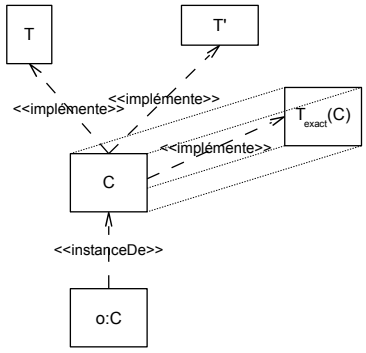

FIGURE 6.5 – Exemple de *type exact*.

Listing 6.2 – Invariant OCL forçant une classe à n'implémenter que des types objets.

```
context Class inv:
  implements->forAll(oclIsKindOf(ObjectType))
```

6.1.4.1 Type exact

Le type exact d'un élément de modèle est le type le plus précis de cet élément, exposant tous ses champs et toutes ses opérations. Il est possible d'extraire le type exact d'un élément de modèle à partir de sa classe, i.e., de la classe qui a servi à l'instancier. En effet un objet étant construit à partir de sa classe, celle-ci déclare toutes les opérations de l'objet. De plus, l'ensemble des champs de l'objet est déclaré par l'ensemble des types objets implémentés par cette classe (à travers l'association `implements`). En ignorant le corps des opérations, les constructeurs et les éventuels champs et opérations invisibles d'une classe (e.g., par le biais de mécanismes de visibilité), et en calculant l'ensemble des champs déclarés par les types objets, on obtient le type exact des instances de cette classe.

La Figure 6.5 présente un exemple de type exact. L'objet o est instance de la classe C et typé par les types implémentés par C : T, T' et $T_{exact}(C)$. Les types T et T' ne présentent qu'une interface partielle sur o, ils n'exposent pas certains champs ou certaines signatures d'opérations.

À l'inverse, le type $T_{exact}(C)$ expose l'intégralité des champs et des signatures de C, et donc accessibles sur o.

6.1.4.2 Visibilité des champs et signatures d'opérations

Les mécanismes de visibilité permettent de ne pas faire apparaître certains des champs ou des opérations d'une classe dans les types objets qui exposent son interface. Ces champs et opérations sont uniquement des artefacts d'implémentation, qui ne sont exposés par aucune interface et qui sont dits privés. Ils n'apparaissent pas dans les types objets et sont donc invisibles à l'extérieur de la classe.

Afin de représenter des champs et des opérations privés, une classe (`Class`) peut contenir des signatures d'opérations (`OperationSignature`) et des champs (`Property`). Contrairement aux autres signatures d'opérations et aux autres champs, les signatures d'opérations et les champs contenus par une classe n'appartiennent à aucun type objet, et n'apparaissent donc pas en dehors de la classe.

6.1.5 Autres types d'éléments de modèles : types de "valeurs"

Certains types d'éléments de modèles ne sont pas des types objets. Ces types ne sont pas instanciables, mais définissent directement les valeurs qu'ils typent : valeurs de types primitifs (entiers, flottants, booléens) ou énumérateurs de types énumérés. Les affectations et passages en paramètre de ces valeurs se font par copie et non par référence. Les types de "valeurs" (par opposition aux types objets qui sont des types de "références") sont représentés dans la Figure 6.3 par la classe `DataType`. Il n'y a pas ici de séparation de l'interface et de l'implémentation, car c'est le type qui définit l'ensemble des valeurs qui le composent (e.g., ensemble des nombres entiers, ensemble des énumérateurs d'un type énuméré).

6.2 Modélisation *in-the-large*

Une fois définies les informations de la modélisation in-the-small prises en compte dans la relation de typage de modèles, et donc dans les types de modèles, nous nous intéressons aux informations de la modélisation in-the-large : les relations entre modèles. À partir de ces informations, nous définissons les types de modèles. Comme pour les éléments de modèles, nous séparons l'interface d'un modèle (son type de modèles) de son implémentation, nous définissons donc également les métamodèles, utilisés pour créer les modèles. Ces concepts de la couche de modélisation in-the-large de METAL dépendent de la couche de modélisation in-the-small : les types de modèles sont composés de types objets et les métamodèles de classes.

6.2.1 Relations entre modèles : champs et opérations de modèles

De nombreuses tâches de l'ingénierie dirigée par les modèles demandent d'établir ou de raisonner sur des relations entre plusieurs modèles : synchronisation et vérification de la cohérence de différents modèles du même système, composition de modèles représentant des préoccupations différentes, etc. Pour permettre de manipuler ces relations, nous les réifions sous la forme de champs et d'opérations de modèles.

Nous réifions dans METAL les différentes relations possibles entre modèles à travers deux concepts, venant directement du paradigme objet : les champs et les opérations de modèles. Les champs de modèles permettent, de la même manière que les champs d'un objet, de stocker une valeur ou une référence. À la différence des champs d'un objet, cette référence peut être une

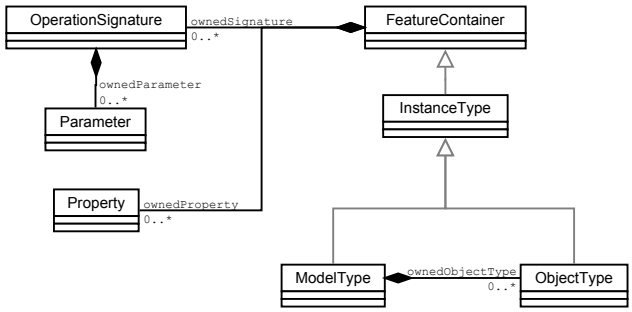

FIGURE 6.6 – Sous-ensemble de la syntaxe abstraite de METAL pour les *types de modèles*.

référence à un autre modèle. Ainsi, il est possible de conserver et de manipuler les relations entre plusieurs modèles. Les opérations de modèles sont des transformations de modèles attachées aux modèles. Elles contiennent un comportement attaché à l'ensemble des modèles issus du même métamodèle, de la même manière que les opérations attachées aux objets.

Les champs de modèles et les opérations de modèles sont représentés par les mêmes classes de la syntaxe abstraite que les champs et les opérations d'objets : `Property`, `OperationSignature` et `Operation`. Cependant, puisque le but des champs et des opérations de modèles est de représenter des relations entre modèles, les champs et opérations de modèles ne sont pas soumis aux invariants présentés en Listing 6.1 : les champs et opérations de modèles peuvent être typés par un type de modèles.

6.2.2 Types de modèles

Un type de modèles expose l'interface à travers laquelle il est possible de manipuler les modèles qu'il type. Cette interface inclut un ensemble de types objets exposant l'interface des éléments des modèles, des champs de modèles et des signatures d'opérations de modèles. La Figure 6.6 présente un sous-ensemble de la syntaxe abstraite de METAL dédié à l'expression des types de modèles.

Un type de modèles (`ModelType`) fournit une interface exposant ce qu'on peut attendre des modèles qu'il type, aux travers de types objets (`ObjectType`), de signatures d'opérations (`OperationSignature`) et de champs (`Property`). Pour cela un type de modèles possède un ensemble de types objets (`ownedObjectType`) et hérite de la classe `InstanceType` qui possède un ensemble de signatures d'opérations (`ownedSignature`) et un ensemble de champs (`ownedProperty`). Ainsi, un type de modèles expose l'interface à travers laquelle il est possible de manipuler les modèles qu'il type autant au niveau des éléments de modèles et de leurs relations (in-the-small) au travers des types objets, qu'au niveau du modèle et de ses relations (in-the-large) au travers des champs de modèles et des signatures d'opérations de modèles.

6.2.3 Métamodèles

Si les types de modèles exposent l'interface des modèles, les métamodèles définissent leur implémentation. La Figure 6.7 présente la syntaxe abstraite de METAL, qui contient l'ensemble des informations nécessaires à la définition de métamodèles.

Un métamodèle (`Metamodel`) implémente (`implements`) un ou plusieurs types de modèles (`ModelType`), en fournissant une classe (`Class`) pour chaque type objet et une opération de modèles (`Operation`) pour chaque signature d'opération. En héritant de `InstantiableElement`, un métamodèle (`Metamodel`) possède un ensemble d'opérations (`ownedOperation`), chacune

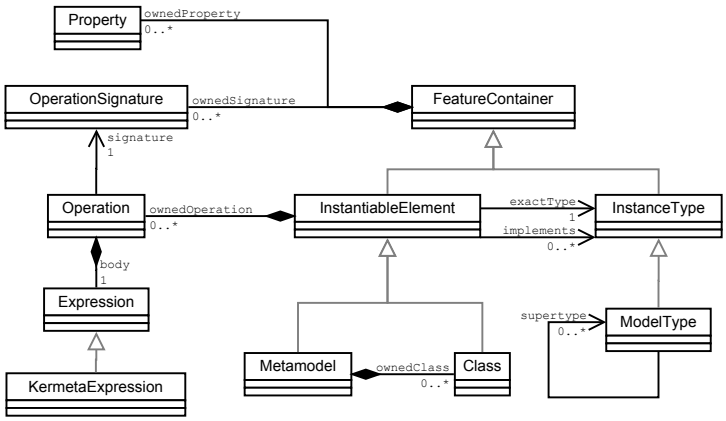

FIGURE 6.7 – Syntaxe abstraite de METAL pour les *métamodèles*.

pointant vers sa signature (`signature`) et contenant son corps (`body`). Afin de s'assurer qu'un métamodèle n'implémente que des types de modèles, nous avons défini un invariant OCL présenté en Listing 6.3.

Listing 6.3 – Invariant OCL forçant un métamodèle à n'implémenter que des types de modèles.

```
context Metamodel inv:
  implements->forAll(oclIsKindOf(ModelType))
```

6.2.3.1 Instanciation de métamodèle

Un modèle possède des champs définis par ses types de modèles. Il est donc nécessaire de stocker en mémoire les valeurs de ces champs. C'est pourquoi un métamodèle est instanciable, ses instances étant des modèles. Instancier un métamodèle revient à allouer en mémoire l'espace nécessaire à un modèle, i.e., à allouer l'espace nécessaire pour stocker les champs du modèle et une référence vers la définition des opérations du métamodèle. Les éléments du modèle peuvent ensuite être créés, accédés et manipulés au travers des champs et des opérations du modèle.

La relation d'instanciation qui lie un modèle à un métamodèle est proche de la relation de conformité (tous les objets du modèle sont instances de classes du métamodèle) mais s'étend à la modélisation in-the-large. De plus, elle est clairement distinguée de la relation de typage, présentée en Section 6.3.

6.2.3.2 *Type de modèles exact*

Tout comme un objet, un modèle possède un type exact (cf. Sction 4.1.4) qui expose toutes les opérations et champs de modèles accessibles depuis le modèle. Il est possible d'extraire le type de modèles exact d'un modèle à partir du métamodèle dont il est instance. Le type exact d'un modèle instance du métamodèle *MM* est constitué de tous les types exacts extraits des classes de *MM* ainsi que de la signature de chaque opération de modèles de *MM* et de chaque champ déclaré par les types de modèles implémentés par *MM*.

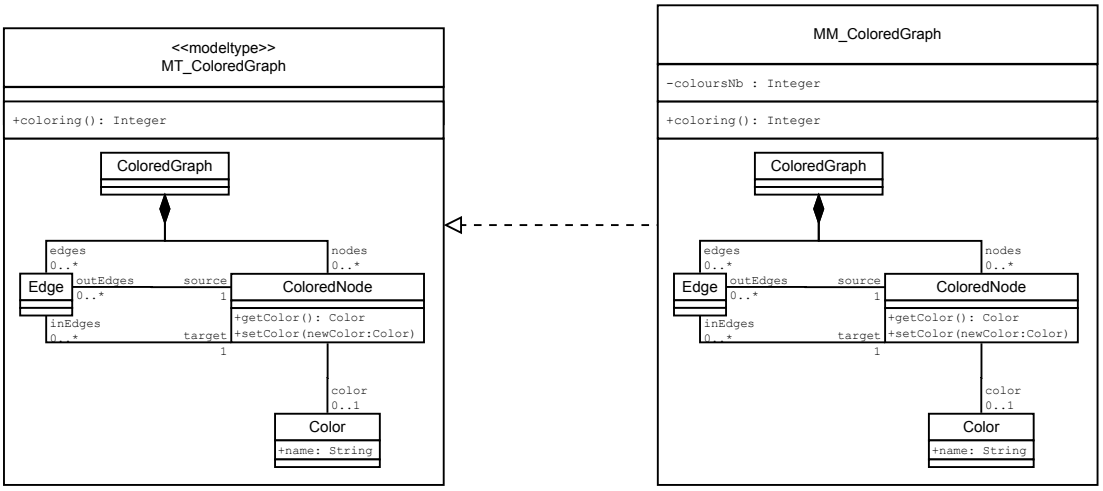

FIGURE 6.8 – Type de modèles $MT_{ColoredGraph}$ (à gauche) et métamodèle $MM_{ColoredGraph}$ implémentant $MT_{ColoredGraph}$ (à droite).

6.2.3.3 Visibilité

Tout comme une classe, un métamodèle peut contenir des signatures d'opérations (`OperationSignature`) et des champs (`Property`), qui sont alors invisibles en dehors du métamodèle. C'est pourquoi la classe `Metamodel` hérite de `FeatureContainer` au travers de `InstantiableElement`.

6.2.4 Représentation des types de modèles et métamodèles

Nous introduisons pour la suite de cette thèse une notation graphique pour les types de modèles et les métamodèles. Cette notation est inspirée de la notation des classes et interfaces des diagrammes de classes UML. Ainsi, un type de modèles est représenté en quatre "compartiments" :

- le premier contient le nom du type de modèles et le stéréotype «`modeltype`», pour différencier les types de modèles des métamodèles ;
- le deuxième contient les champs de modèles exposés par le type de modèles ;
- le troisième contient les signatures d'opérations de modèles exposées par le type de modèles ;
- le quatrième contient un diagramme de classes représentant les types objets contenus par le type de modèles. Dans un souci de simplicité, les types objets ne portent pas de stéréotype : tous les éléments du diagramme de classe contenus par un type de modèles sont des types objets.

De même, un métamodèle est représenté en quatre "compartiments" :

- le premier contient le nom du métamodèle ;
- le deuxième contient les champs de modèles exposés par les types de modèles que le métamodèle implémente ainsi que les champs de modèles privés (précédés d'un -) ;
- le troisième contient les signatures des opérations de modèles implémentées par le métamodèle, y compris les opérations de modèles privées (précédées d'un -) ;

– le quatrième contient un diagramme de classes représentant les classes contenues par le métamodèle.

La Figure 6.8 donne un exemple de type de modèles `MT_ColoredGraph` exposant une opération de modèles (`coloring`) et quatre types objets (`ColoredGraph`, `Edge`, `ColoredNode` et `Color`). La Figure 6.8 donne également un exemple de la représentation d'un métamodèle `MM_ColoredGraph`, qui possède un champ de modèles privé (`coloursNb`), qui implémente une opération de modèles (`coloring`) et possède quatre classes (`ColoredGraph`, `Edge`, `ColoredNode` et `Color`).

Le métamodèle `MM_ColoredGraph` implémente donc le type de modèles `MT_ColoredGraph`, puisqu'il fournit une opération de modèles pour la signature d'opérations `coloring` et une classe pour chaque type objet de `MT_ColoredGraph`. La relation entre un métamodèle et un type de modèles que ce métamodèle implémente est représentée par la flèche pointillée à pointe creuse, présentée en Figure 6.8.

6.3 Relation de typage de modèle

Une fois définie la structure d'un type de modèles : un ensemble de types objets (Section 6.1.2), de signatures d'opérations de modèles et de champs de modèles (Section 6.2.1), il faut définir la relation qui lie un modèle à ses types de modèles. Cette relation est la relation de typage, plus précise que la relation de conformance, puisqu'elle prend en compte les relations entre modèles, mais également plus flexible, puisqu'elle autorise un modèle à avoir plusieurs types de modèles, i.e., elle autorise le polymorphisme de sous-type.

Pour déterminer si un modèle est typé par un type de modèles donné, nous utilisons le type de modèles exact du modèle, i.e., le type extrait du métamodèle dont le modèle est instance, ainsi que les quatre relations de sous-typage entre types de modèles définies dans le chapitre suivant (Chapitre 7).

Ces quatre relations de sous-typage entre types de modèles peuvent être ramenées à la première d'entre elles : la relation de sous-typage totale isomorphique (notée <:). Par définition de la relation de sous-typage, un modèle typé par un type de modèles MT est également typé par les super-types de MT. Le type exact MT_m d'un modèle m est le type le plus précis de m, c'est à dire qu'il n'existe pas de sous-type de MT_m qui type également m. Le type exact d'un modèle est donc le sous-type de tous les types de ce modèle. Ainsi, un modèle m est typé par un type de modèles MT si MT est un super-type du type exact de m. L'ensemble de tous les types de modèles d'un modèle est donc l'ensemble de tous les super-types du type exact du modèle.

Définition 6.1. ***(Relation de typage de modèles)*** La relation de typage de modèles est une relation binaire : de **M**, l'ensemble de tous les modèles, à **MT**, l'ensemble de tous les types de modèles, tel que $(m, MT) \in :$ (également noté $m : MT$) ssi $MT_m <: MT$, où $MT \in \mathbf{MT}$, $m \in \mathbf{M}$ et MT_m est le type de modèles exact de m.

6.4 Conclusion

Dans ce chapitre, nous avons proposé une syntaxe abstraite permettant d'exprimer les types de modèles comme un ensemble de types objets, de champs de modèles et de signatures d'opérations de modèles. Un type de modèles fournit donc une interface permettant de manipuler les modèles aussi bien à travers ses éléments (i.e., in-the-small) qu'à travers les champs et les signatures d'opérations de modèles (in-the-large).

TABLE 6.2 – Équivalence entre les concepts de modélisation *in-the-small* et les concepts de modélisation *in-the-large* définis dans METAL.

	Objet (Modélisation *in-the-small*)	Modèle (Modélisation *in-the-large*)
Interface	Type objet	Type de modèles
Implémentation	Classe	Métamodèle
Relations	Champs	Champs de modèles
Comportement	Opérations	Opérations de modèles
Mécanisme d'instanciation	Réserve l'espace mémoire pour les champs définis par la classe, plus une référence vers la table d'opérations de la classe.	Réserve l'espace mémoire pour les champs définis par le métamodèle, plus une référence vers la table d'opérations de modèles du métamodèle.

Nous proposons également de représenter les métamodèles comme des entités instanciables contenant un ensemble de classes et d'opérations de modèles. L'instanciation d'un métamodèle crée un modèle sur lequel il est possible d'appeler des opérations de modèles et d'accéder à des champs de modèles. Ces champs et opérations de modèles permettent de réifier les relations entre modèles de plusieurs types différents (e.g., cohérence, synchronisation, dépendances). La Table 6.2 résume les équivalences entre les concepts de modélisation in-the-large définis au sein de METAL et les concepts orientés-objet, de la modélisation in-the-small.

Nous avons également proposé une définition de la relation de typage, qui s'appuie sur le type exact d'un modèle et les relations de sous-typage entre types de modèles.

L'ensemble des définitions proposées dans ce chapitre fournissent le socle nécessaire à l'implémentation de systèmes de types orientés-modèle au sein d'environnements de développement des langages dédiés. Nous fournissons les moyens de définir les types de modèles et de les lier aux modèles par la relation de typage. Il est ensuite possible de s'appuyer sur ces types de modèles pour implémenter différents outils supportant différentes facilités de typage : vérificateur de types détectant les erreurs de typage, relations de sous-typage autorisant le polymorphisme, etc.

Dans le chapitre suivant, nous définissons deux relations entre langages de modélisation dédiés : le sous-typage, qui s'établit entre types de modèles et permet la réutilisation de transformations de modèles par polymorphisme de sous-type ; et l'héritage, qui s'établit entre métamodèles et qui permet la réutilisation du code des métamodèles dont on hérite. Dans la Partie III, nous présentons l'implémentation d'un système de types orienté-modèles s'appuyant sur les concepts de METAL, i.e., sur les types de modèles et la relation de typage.

Chapitre 7

Relations entre langages : sous-typage et héritage

Afin d'implémenter un système de types orienté-modèle supportant des facilités telles que la réutilisation, il n'est pas suffisant de définir la structure des langages de modélisation dédiés. Il est également nécessaire de définir des relations entre ces langages. En effet, pour réutiliser la structure ou le comportement d'un langage de modélisation dédié à un autre, il est nécessaire de ne pas considérer les langages en isolation, mais liés les uns aux autres.

Dans ce chapitre, nous nous appuyons sur les définitions des types de modèles et des métamodèles données dans le chapitre précédent pour définir des relations entre langages de modélisation dédiés. Les relations de sous-typage entre types de modèles permettent la substituabilité des modèles issus de deux langages différents, et donc le polymorphisme de sous-type (cf. Section 7.1). Les relations d'héritage entre métamodèles permettent la réutilisation de la structure d'un langage existant et de son comportement (cf. Section 7.2). La Figure 7.1 présente les relations abordées dans ce chapitre : sous-typage et héritage.

Les relations de sous-typage entre modèles s'appuient sur une relation de correspondance in-the-small entre les types objets et sur deux relations de correspondance in-the-large entre les signatures d'opérations de modèles d'une part et entre les propriétés de modèles d'autre part (cf. Section 7.1.1, 7.1.2 et 7.1.3 respectivement). Nous distinguons quatre relations de sous-typage entre types de modèles en fonction de deux critères : la présence ou non d'hétérogénéités structurelles entre les deux types de modèles considérés ; et le contexte dans lequel la relation de sous-typage est utilisée (cf. Section 7.1.6).

Les relations d'héritage entre métamodèles permettent d'étendre et de redéfinir les éléments d'un super-métamodèle pour créer un sous-métamodèle (cf. Section 7.2). Séparer l'héritage du sous-typage permet d'offrir plus de flexibilité dans la redéfinition des éléments du sous-métamodèle, mais également dans la définition des relations de sous-typage, en autorisant la déclaration ou l'inférence de relations de sous-typage après la définition des types (cf. Section 7.3.1).

Les relations de sous-typage et d'héritage, ainsi que la manière de les déclarer et de les vérifier, forment une famille de systèmes de types orientés-modèle. Selon les choix de conception et d'implémentation faits pour un système de types donné, celui-ci pourra supporter un ensemble de facilités pour la définition et l'outillage de langages de modélisation dédiés (e.g., réutilisation) mais également pour la manipulation de modèles (e.g., auto-complétion) (cf. Section 7.3).

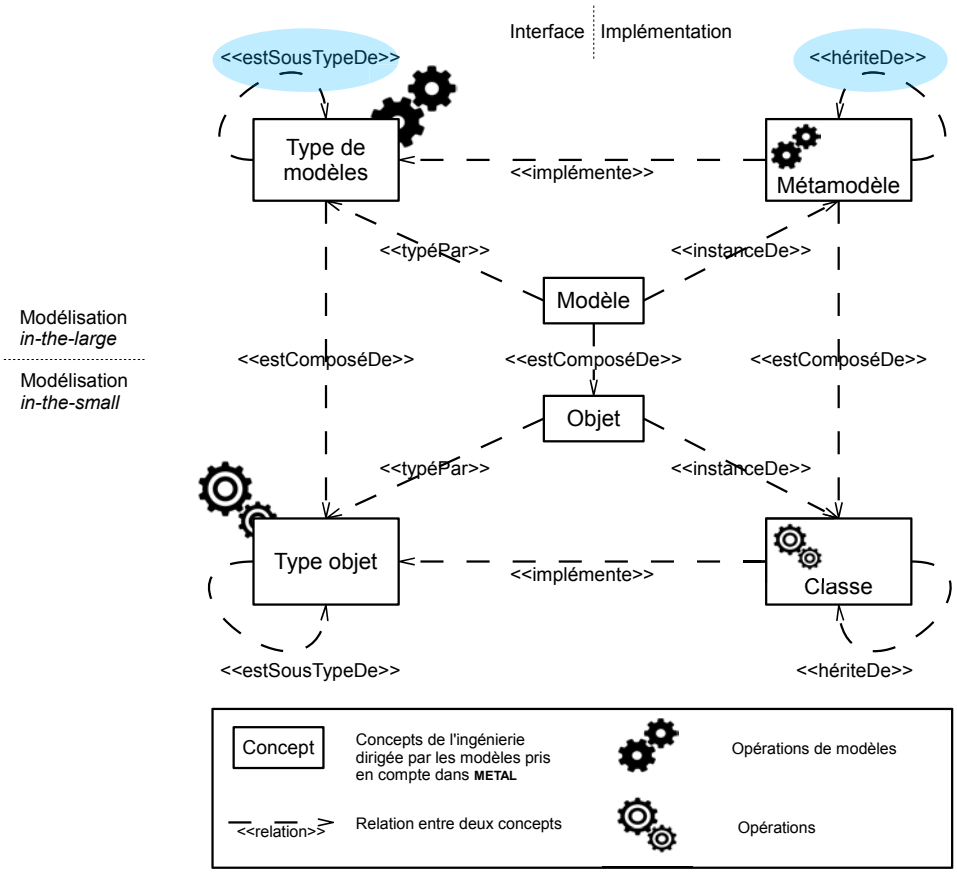

FIGURE 7.1 – Relations de METAL abordées dans ce chapitre.

7.1 Relations de sous-typage entre types de modèles

L'une des facilités principales fournies par les systèmes de types orientés-objet est la réutilisation grâce au polymorphisme de sous-type, qui permet la substituabilité des objets. Pour implémenter des systèmes de types orientés-modèle supportant le polymorphisme de sous-type entre types de modèles, il est donc nécessaire d'établir dans quelles conditions un type de modèles est sous-type d'un autre.

Afin de déterminer si un modèle m' est substituable à un autre modèle m, il faut comparer leurs types et établir s'il existe une relation de sous-typage entre eux. Pour cela, il faut non seulement comparer les champs et les signatures d'opérations de ces types, afin de s'assurer que toutes les opérations ou champs de modèles qui pourraient être appelés sur m peuvent être appelés de manière sûre sur m', mais il faut également comparer les types objets qui composent les types de modèles.

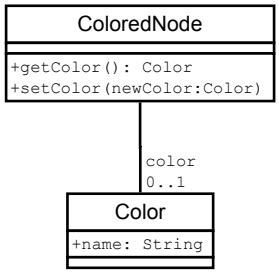

FIGURE 7.2 – Groupe de types objets *ColoredNode* et *Color*.

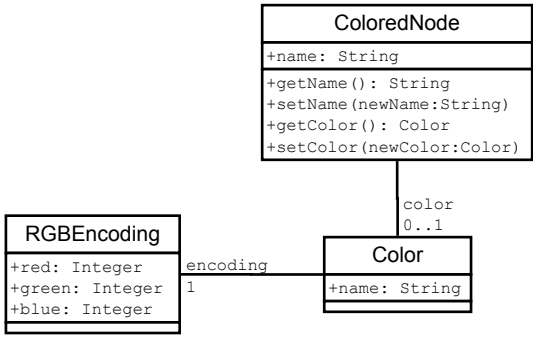

FIGURE 7.3 – Groupe de types objets *ColoredNode'*, *Color'* et *RGBEncoding'*.

7.1.1 Correspondance de types objets

Afin de déterminer si un type de modèles *MT'* est un sous-type d'un type de modèles *MT*, i.e., si les modèles typés par *MT'* sont substituables aux modèles typés par *MT*, il faut déterminer si les objets typés par les types objets de *MT'* sont substituables aux objets typés par les types objets de *MT*. Cette substituabilité des objets n'a pas besoin d'être totale, i.e., les objets de *MT'* n'ont pas à être substituables aux objets de *MT* dans tous les contextes possibles, mais uniquement dans le cadre d'une substitution de groupes d'objets (cf. Section 4.3), i.e., dans le cadre d'une substitution de modèles.

Les Figures 7.2 et 7.3 présentent deux groupes de types composés d'un type `ColoredNode` et d'un type `Color` (plus un type `RGBEncoding` pour la Figure 7.3). Nous différencions les types de chaque figure par la convention suivante : le type objet `X` de la Figure 7.2 sera noté *X* et le type objet `X` de la Figure 7.3 sera noté *X'*.

Pour cet exemple, il n'existe pas de relation de sous-typage entre les types d'un groupe et de l'autre, l'invariance des types de champs n'étant pas respectée (cf. Section 4.2.1.1). Cependant, un objet typé par le type objet *ColoredNode'* peut-être substitué à un objet typé par le type objet *ColoredNode* dans le cadre d'un substitution de groupe, i.e., si des objets typés par *ColoredNode'* et *Color'* sont substitués à tous les objets typés par *ColoredNode* et *Color* (cf. Section 4.3). Dans ce cas, il n'y a pas de "mélange" possible entre les objets d'un groupe et ceux d'un autre et la substitution est sûre.

De la même manière, lorsque l'on substitue un modèle *m'* typé par le type de modèles *MT'* à un autre modèle *m* typé par le type de modèles *MT*, tous les objets typés par des types objets appartenant à *MT* sont remplacés par des objets typés par des types objets appartenant à *MT'*. Il n'est donc pas nécessaire d'établir de relations de sous-typage entre les types objets de deux types de modèles. C'est pourquoi, pour déterminer la substituabilité des objets dans le cadre d'une substitution de modèles, nous utilisons une définition élargie de la correspondance de types objets présentée par Bruce et al. [BSvGF03] et spécialisée par Steel et al. [SJ07] pour les classes MOF (définition 4.4).

7.1.1.1 Définition

Nous définissons ici la relation de correspondance entre deux types objets, sur laquelle s'appuient les relations de sous-typage entre types de modèles (cf. Définition 7.1). La relation de correspondance entre types objets (notée <#) s'appuie elle-même sur la correspondance des signatures des opérations des types objets, abordée en Section 7.1.2 et sur la correspondance

de leurs champs, abordée en Section 7.1.3.

Notre relation de correspondance de types est plus stricte que celle définie par Steel et al. : les noms de classes doivent être inchangés d'un type objet à l'autre (cf. Définition 7.1.i). L'égalité des noms simplifie l'algorithme de sous-typage de types de modèles en évitant de tester toutes les paires possibles de types objets entre deux types de modèles et en évitant d'avoir à résoudre des ambiguïtés (i.e., deux types objets pouvant correspondre à un même type). Cette restriction peut être levée dans le cadre d'une relation de sous-typage non-isomorphique à l'aide de fonctions d'adaptation, telles que présentées en Section 7.1.6.2.

La correspondance entre deux types objets T' et T s'appuie elle-même sur la correspondance des signatures d'opérations (cf. Définition 7.1.ii) et la correspondance des champs (cf. Définition 7.1.iii) qui déterminent si une opération (respectivement un champ) de T' peut être substituée à une opération (respectivement un champ) de T dans le cadre d'une substitution de modèles (cf. Sections 7.1.2 et 7.1.3).

Définition 7.1. ***(Correspondance de types objets)*** Le type objet $T' \in MT'$ correspond au type objet $T \in MT$ (noté $T' <\# T$) ssi :

i $T.name = T'.name$

ii $\forall op \in T.ownedSignature, \exists op' \in T'.ownedSignature \wedge op' <\# op$ (cf. Section 7.1.2)

iii $\forall a \in T.ownedProperty, \exists a' \in T'.ownedProperty \wedge a' <\# a$ (cf. Section 7.1.3)

7.1.1.2 Exemple de correspondance entre types objets : nœuds colorés

Les Figures 7.2 et 7.3 présentent deux types objets *ColoredNode* et *ColoredNode'*, tous deux nommés `ColoredNode`. Ces deux types déclarent un champ (`color`) ainsi que les signatures de deux opérations permettant d'y accéder (`getColor` et `setColor`). De plus *ColoredNode'* déclare un autre champ (`name`) et deux signatures d'opérations supplémentaires (`getName` et `setName`). Les Figures 7.2 et 7.3 présentent également d'autres types objets : deux types objets nommés `Color` et un type nommé `RGBEncoding`. Notre exemple se concentre sur la correspondance entre *ColoredNode'* et *ColoredNode*, mais utilisent les types `Color` et `RGBEncoding` dans la vérification de cette correspondance.

Les deux types objets ayant le même nom, ils remplissent donc la première condition pour que l'un d'eux corresponde à l'autre (cf. Définition 7.1.i).

ColoredNode' correspond à *ColoredNode* (noté *ColoredNode'*$<\#$*ColoredNode*) s'il possède une signature d'opération correspondante à chacune des signatures d'opérations de *ColoredNode* (cf. Définition 7.1.ii et Section 7.1.2) ; et si il possède un champ correspondant à chacun des champs de *ColoredNode* (cf. Définition 7.1.iii et Section 7.1.3). Pour correspondre à *ColoredNode*, *ColoredNode'* doit donc posséder deux signatures d'opérations correspondant respectivement à *ColoredNode*.`getColor` et *ColoredNode*.`setColor` et un champ correspondant à *ColoredNode*.`color`.

7.1.1.3 Correspondance de types objets et types n'appartenant pas à un groupe

La correspondance de types objets ne s'applique qu'aux types appartenant à un des deux groupes de types impliqués dans la substitution. Par exemple, si les types *Color'* et *Color* n'appartenaient pas aux mêmes groupes de type que *ColoredNode'* et *ColoredNode*, il ne serait pas sûr de substituer un objet de type *ColoredNode'* à un objet de type *ColoredNode*. En effet, la correspondance de types objets n'est possible que si l'ensemble des types objets impliqués font partie d'un des deux groupes de types. Les autres types (i.e., ceux qui ne font pas partie que d'un des deux groupes de types, ou d'aucun) doivent être comparés à l'aide du sous-typage objet.

7.1.2 Correspondance de signatures d'opérations

Pour établir une relation de correspondances entre deux types objets, il est nécessaire de comparer les signatures d'opérations que ces types exposent. En effet, pour substituer un objet à un autre, le système de types doit assurer que toutes les opérations qui peuvent être appelées sur l'objet original peuvent également être appelées sur l'objet qu'on lui substitue. Pour cela nous définissons une relation de correspondance entre signatures d'opérations.

Afin de pouvoir substituer une opération de signature op' appartenant à un type objet T' du type de modèles MT' à une opération de signature op appartenant à un type objet T du type de modèles MT dans le cadre d'une substitution de modèles, il faut que op' corresponde à op (noté $op' <\# op$). De même, pour pouvoir substituer une opération de modèles de signature op' appartenant à un type de modèles MT' à une opération de modèles de signature op appartenant à un type de modèles MT dans le cadre d'une substitution de modèles, il faut que op' corresponde à op.

7.1.2.1 Définition

La Définition 7.2 donne la définition de la relation de correspondance entre deux signatures d'opérations op' et op. Pour que op' corresponde à op, les deux signatures d'opérations doivent avoir le même nom (cf. Définition 7.2.i), afin qu'une transformation de modèles définie par rapport à T puisse appeler l'opération de signature op' lors de la substitution d'un objet de type T' à un objet de type T (respectivement lors de la substitution d'un modèle de type MT' à un modèle de type MT).

Si les deux types de retour n'appartiennent pas respectivement à MT' et à MT, le type de retour de op' doit être un sous-type de celui de op. Dans le cas où les deux types de retour appartiennent respectivement à MT' et à MT, le type de retour de op' doit être un sous-type de celui de op, ou correspondre à celui de op (cf. Définition 7.2.ii). En effet, la correspondance entre types objets n'est sûre qu'au sein de deux groupes de types, i.e., au sein de deux types de modèles. op' doit également déclarer le même nombre de paramètres que op, dans le même ordre (cf. Définition 7.2.iii(b)). op' ne peut pas déclarer plus d'exceptions que op, celles-ci devant être covariantes par rapport aux exceptions de op (cf. Définition 7.2.iv).

Chaque paramètre p' de op' doit avoir un type : correspondant au type du paramètre p de op si $p'.type \in MT'$ et $p.type \in MT$, ou un type super-type du type de p sinon (cf. Définition 7.2.iii(a)).

p' et p doivent également avoir les mêmes multiplicités pour éviter les underflows et overflows (cf. Définition 7.2.iii(c) et Définition 7.2.iii(d)). En effet, une transformation de modèles pourrait essayer d'assigner le contenu de p' à une variable ou un champ en fonction des multiplicités déclarées par p.

p' et p doivent soit être uniques tous les deux, soit ne pas l'être tous les deux (cf. Définition 7.2.iii(e)). Dans le cas contraire, une transformation de modèles pourrait s'attendre à un comportement en fonction de l'unicité de p (e.g., augmentation de la taille d'une collection non-unique à l'ajout d'un élément déjà présent) mais obtenir un comportement différent avec p'.

Enfin, si p n'est pas ordonné, p' ne doit pas l'être non plus (cf. Définition 7.2.iii(f)). En effet un appel à op pourrait passer en paramètre une collection non-ordonnée, et op' doit être en mesure d'être appelée partout où op est appelée.

7.1.2.2 Exemple de correspondance entre signatures d'opérations : nœuds colorés (suite)

Les Figures 7.4 et 7.5 étendent l'exemple précédent (Figure 7.2 et 7.3) en présentant deux types de modèles ($MT_{ColoredGraph}$ et $MT_{RGBColoredGraph}$) contenant respectivement les types objets

Définition 7.2. ***(Correspondance de signatures d'opérations et d'opérations de modèles)*** La signature d'opération $op' \in T'$, tel que $T' \in MT'$ (respectivement la signature d'opération de modèles $op' \in MT'$), correspond à la signature d'opération $op \in T$, tel que $T \in MT$ (respectivement la signature d'opération de modèles $op \in MT$), (noté $op' <\#op$) ssi :

i $op.name = op'.name$

ii $(op'.type <\#op.type \wedge op'.type \in MT' \wedge op.type \in MT) \vee op'.type <: op.type$

iii $p \in op.ownedParameter \Leftrightarrow p' \in op'.ownedParameter$ tel que :

(a) $(p'.type <\#p.type \wedge p'.type \in MT' \wedge p.type \in MT) \vee p.type <: p'.type$

(b) $p.rank = p'.rank$

(c) $p.lower = p'.lower$

(d) $p.upper = p'.upper$

(e) $p.isUnique = p'.isUnique$

(f) $p'.isOrdered \Rightarrow p.isOrdered$

iv $\forall e' \in op'.raisedException, \exists e \in op.raisedException$ tel que $(e' <\#e \wedge e' \in MT' \wedge e \in MT) \vee e' <: e$

ColoredNode et *ColoredNode'*. Comme la majeure partie des types objets de $MT_{ColoredGraph}$ et $MT_{RGBColoredGraph}$ portent les mêmes noms, nous utiliserons dans la suite la même convention que précédemment : le type objet `X` du type de modèles $MT_{ColoredGraph}$ sera noté X et le type objet `X` du type de modèles $MT_{RGBColoredGraph}$ sera noté X'.

Pour que le type objet *ColoredNode'* corresponde au type objet *ColoredNode*, *ColoredNode'* doit posséder une signature d'opération correspondante pour chaque signature d'opération de *ColoredNode* : `getColor` et `setColor` (cf. Section 7.1.1).

ColoredNode' possède une signature d'opération nommée `getColor`. Celle-ci ne déclare ni paramètre, ni exception, ce qui est également le cas de la signature d'opération `getColor` de *ColoredNode*. Pour correspondre à *ColoredNode*.`getColor`, *ColoredNode'*.`getColor` doit également respecter les règles pour les types de retours (cf. Définition 7.2.ii). *Color'*, le type de retour de *ColoredNode'*.`getColor`, doit soit être un sous-type de *Color*, le type de retour de *ColoredNode*.`getColor` (*Color'* <: *Color*), soit lui correspondre (*Color'* <#*Color*).

ColoredNode' possède également une signature d'opération nommée `setColor`. Comme la signature d'opération `setColor` de *ColoredNode*, elle ne déclare pas de type de retour ni d'exception. Pour savoir si *ColoredNode'*.`setColor` correspond à *ColoredNode*.`setColor`, il faut donc comparer leurs paramètres. Chacune des deux signatures déclare un unique paramètre : ils ont donc le même rang (cf. Définition 7.2.iii(b)). Ils possèdent tous deux les multiplicités par défaut (respectivement 0 pour la borne inférieure et 1 pour la borne supérieure), qui sont donc égales (cf.Définition 7.2.iii(c) et Définition 7.2.iii(d)). Puisque la borne supérieure est 1, les deux paramètres ne sont pas des collections, et ont tous les deux leur valeur d'unicité et d'ordre à `faux`. Pour correspondre à *ColoredNode*.`setColor`, *ColoredNode'*.`setColor` doit également respecter la contrainte sur les types des paramètres (cf. Définition 7.2.iii(a)). *Color'*, le type du paramètre de *ColoredNode'*.`setColor` doit soit être un super-type de *Color*, le type du paramètre de *ColoredNode*.`setColor` (*Color* <: *Color'*), soit lui correspondre (*Color'* <#*Color*).

Si *Color* et *Color'* n'appartenaient pas respectivement aux types de modèles de *ColoredNode* et *ColoredNode'* (i.e., $MT_{ColoredGraph}$ et $MT_{RGBColoredGraph}$), *Color'* devrait satisfaire la contrainte *Color'* <: *Color* $\wedge$ *Color* <: *Color'*, i.e., *Color'* = *Color*, pour que *ColoredNode'* corresponde à *ColoredNode*. En effet, si *Color* et *Color'* n'appartenaient pas à $MT_{ColoredGraph}$ et $MT_{RGBColoredGraph}$, il n'y aurait aucune garantie que des objets typés par *Color'* soient substitués aux objets typés par

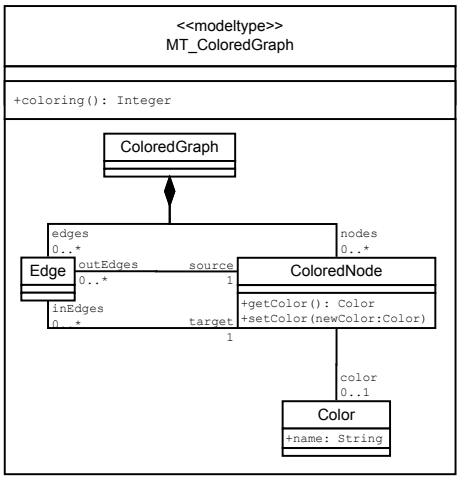

FIGURE 7.4 – Type de modèles $MT_{ColoredGraph}$.

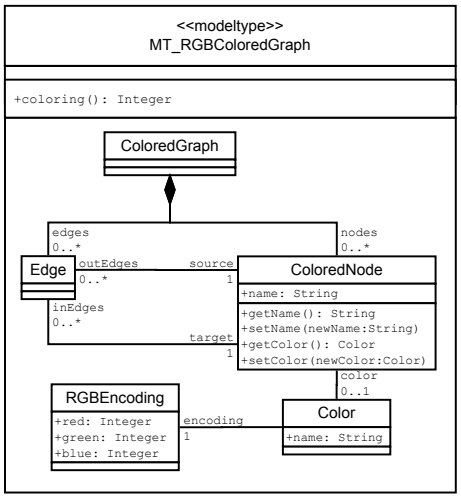

FIGURE 7.5 – Type de modèles $MT_{RGBColoredGraph}$.

Color lors de la substitution d'objets typés par *ColoredNode*′ à des objets typés par *ColoredNode*.

Dans un tel cas, la correspondance des types objets n'est pas suffisante pour assurer un comportement sûr, il faut donc utiliser le sous-typage objet (cf. Section 7.1.1.3).

Dans notre cas, puisque *Color* appartient à $MT_{ColoredGraph}$ et que *Color*′ appartient à $MT_{RGBColoredGraph}$, *Color*′ doit satisfaire la contrainte *Color*′<#*Color*. Pour que *ColoredNode*′ corresponde à *ColoredNode* il faut également que *ColoredNode*′ possède un champ correspondant pour chaque champ de *ColoredNode* (cf. Section 7.1.3).

7.1.3 Correspondance de champs

Pour établir une relation de correspondance entre deux types objets, il est nécessaire de comparer les champs que ces types exposent. En effet, pour substituer un objet à un autre, le système de types doit assurer que tous les champs qui peuvent être appelés sur l'objet original peuvent également être appelés sur l'objet qu'on lui substitue. Pour cela nous définissons une relation de correspondance entre champs.

Afin de pouvoir substituer un champ a' appartenant à un type objet T' du type de modèles MT' (respectivement un champ de modèles a' appartenant à un type de modèles MT') à un champ a appartenant à un type objet T du type de modèles MT (respectivement un champ de modèles a appartenant à un type de modèles MT) dans le cadre d'une substitution de modèles, il faut que a' corresponde à a (noté a'<#a).

7.1.3.1 Définition

La Définition 7.3 donne la définition de la relation de correspondance entre deux champs a' et a. Pour que a' corresponde à a, il faut que a' et a aient le même nom (cf. Définition 7.3.i). Si a est un champ en lecture seule (read only en anglais), le type de a' peut être un sous-type de celui de a. Sinon dans le cas où les deux types appartiennent respectivement à MT' et à MT, le type de a' doit correspondre à celui de a. Enfin, si les types de a et de a' n'appartiennent pas respectivement à MT et à MT', le type a' doit être identique à celui de a (cf. Définition 7.3.vi).

Un champ en lecture seule ne peut correspondre qu'à un champ lui-même en lecture seule (cf. Définition 7.3.ii), afin d'éviter qu'une transformation de modèles n'essaie de modifier un champ qui ne peut pas l'être, après substitution.

Tout comme pour les paramètres d'opération, l'unicité (cf. Définition 7.3.iv) et les multiplicités (cf. Définition 7.3.vii et Définition 7.3.viii) doivent rester inchangées entre a et a'.

Si a est ordonné, a' doit l'être également (cf. Définition 7.3.v) afin de conserver le même comportement après substitution. a et a' doivent également avoir la même sémantique de composition (cf. Définition 7.3.iii).

Enfin, si a possède un champ opposé a_{opp}, a' doit également posséder un champ opposé a'_{opp} tel que $a'_{opp} <\# a_{opp}$. Puisque deux champs appartenant à un même type ne peuvent pas avoir le même nom, il suffit de s'assurer que a_{opp} et a'_{opp} ont le même nom (cf. Définition 7.3.ix).

Définition 7.3. ***(Correspondance de champs)*** Le champ $a' \in T'$, tel que $T' \in MT'$ (respectivement $a' \in MT'$), correspond au champ $a \in T$, tel que $T \in MT$ (respectivement $a \in MT$), (noté $a' <\# a$) ssi :

i $a.name = a'.name$

ii $a'.isReadOnly \Rightarrow a.isReadOnly$

iii $a.isComposite = a'.isComposite$

iv $a.isUnique = a'.isUnique$

v $a.isOrdered \Rightarrow a'.isOrdered$

vi $(a'.type <\# a.type \wedge a'.type \in MT' \wedge a.type \in MT) \vee (a'.type <: a.type \wedge a.isReadOnly) \vee a'.type = a.type$

vii $a.lower = a'.lower$

viii $a.upper = a'.upper$

ix $a.opposite \neq void \Rightarrow a'.opposite \neq void \wedge a.opposite.name = a'.opposite.name$

7.1.3.2 Exemple de correspondance de champs : nœuds colorés (suite)

Pour que le type objet *ColoredNode'* de la Figure 7.5 corresponde au type objet *ColoredNode* de la Figure 7.4, *ColoredNode'* doit posséder un champ correspondant à chacun des champs de *ColoredNode* (cf. Section 7.1.1).

Le type objet *ColoredNode* du type de modèles $MT_{ColoredGraph}$ possède trois champs : `color`, `outEdges` et `inEdges`. Pour correspondre à *ColoredNode*, *ColoredNode'* doit donc posséder trois champs correspondants. *ColoredNode'* possède trois champs possédant respectivement les mêmes noms (cf. Définition 7.3.i). Ils possèdent également les mêmes multiplicités : `0..1` pour `color` et `0..*` pour `outEdges` et `inEdges` (cf. Définition 7.3.vii et Définition 7.3.viii). Que ce soit dans *ColoredNode* ou dans *ColoredNode'*, aucun d'entre eux n'est déclaré en lecture seule, composite, unique, ou ordonné (cf. Définition 7.3.ii, Définition 7.3.iii, Définition 7.3.iv et Définition 7.3.v).

Pour que *ColoredNode'* corresponde à *ColoredNode* il faut également comparer les types et les champs opposés des champs `color`, `outEdges` et `inEdges`.

Le champ *ColoredNode*.`color` ne possède pas de champ opposé. Puisqu'il ne s'agit pas d'un champ en lecture seule, le type de *ColoredNode'*.`color` doit être soit invariant (i.e., être le même type) soit correspondre au type de *ColoredNode*.`color` (cf. Définition 7.3.vi).

De la même manière, pour les champs *ColoredNode'*.`inEdges` et *ColoredNode'*.`outEdges`, leur type (qui est le même : *Edge'*) doit être le même que le type des champs *ColoredNode*.`inEdges` et *ColoredNode*.`outEdges` ou correspondre à ce dernier (*Edge*). De plus, les champs *ColoredNode*.`inEdges` et *ColoredNode*.`outEdges` possèdent un champ opposé (respectivement

`target` et `source`). Les champs *ColoredNode*′.`inEdges` et *ColoredNode*′.`outEdges` doivent également posséder des champs opposées, qui ont le même nom (cf. Définition 7.3.ix) pour leur correspondre, ce qui est le cas. Nous pouvons donc préciser les contraintes pour que *ColoredNode*′ corresponde à *ColoredNode*.

ColoredNode′<#*ColoredNode* si *Color*′<#*Color* ∧ (*Edge*′ = *Edge* ∨ *Edge*′<#*Edge*). Les champs `source` et `target` de *Edge*′ ne pointent pas vers le même type que les champs `source` et `target` de *Edge*. En effet, *ColoredNode*′ possède un champ `name`, ce qui n'est pas le cas de *ColoredNode*, ce ne sont donc pas les mêmes types (de plus, leur champ `color` ne pointe pas vers le même type). *Edge*′ n'est donc pas le même type que *Edge*. Pour que *ColoredNode*′ corresponde à *ColoredNode*, il faut donc que *Color*′ corresponde à *Color* et que *Edge*′ corresponde à *Edge*, ce qui peut être vérifié en utilisant la même méthode que celle suivie pour *ColoredNode*′ (en vérifiant les noms, puis les signatures d'opérations et les champs de chaque type objet).

Cependant, il est possible d'arriver à une dépendance cyclique de correspondances. Dans notre cas, après avoir comparé chaque paire de types objets portant le même nom, nous arrivons à la conclusion que *ColoredNode*′<#*ColoredNode* si *Edge*′<#*Edge* et que *Edge*′<#*Edge* si *ColoredNode*′<#*ColoredNode*. Mais il est également possible d'obtenir des dépendances cycliques à travers plusieurs types objets, qui peuvent donc être plus difficiles à détecter. Ces dépendances cycliques peuvent être aisément résolues. En effet, rien ne s'oppose à ce que chacune des paires de types objets correspondent. Considérer que l'une des paires correspond entraîne donc la résolution de chacune des vérifications de correspondance en chaîne, pour obtenir un ensemble de paires correspondantes. Dans notre exemple, considérer que *Edge*′<#*Edge* permettra de vérifier que *ColoredNode*′<#*ColoredNode*, l'inverse étant vrai également.

7.1.4 Correspondance de champs et de signatures d'opérations de modèles

Nous avons défini les règles permettant d'établir la correspondance des éléments in-the-small d'un type de modèles : types objets (cf. Section 7.1.1) et leurs champs (cf. Section 7.1.3) et signatures d'opérations (cf. Section 7.1.2). Ces relations de correspondance permettent d'établir si les éléments d'un modèle sont substituables en groupe aux éléments d'un autre modèle. Cependant, les types de modèles prennent également en compte des éléments in-the-large : les champs de modèles et les opérations de modèles. C'est pourquoi nous abordons dans cette section la correspondance de ces champs et opérations de modèles.

Afin de déterminer si un champ de modèles est substituable à un autre dans le cadre d'une substitution de modèles, nous utilisons les mêmes règles de correspondance que pour les champs d'objets (Définition 7.3). De même, la correspondance de signature d'opérations de modèles suit les mêmes règles de correspondance que la correspondance de signature d'opérations d'objets (Définition 7.2).

Un point à noter est qu'il n'existe pas de correspondance de types de modèles. En effet, la correspondance est une relation relâchée par rapport au sous-typage, établie entre les types de deux groupes qui ne peuvent être sous-types. Les types de modèles n'appartiennent pas eux mêmes à des groupes de types de modèles. Les règles de correspondance de champs et d'opérations de modèles typés par des types de modèles s'appuient donc sur le sous-typage et la variance des types (cf. Section 4.2.1.1).

Ainsi si le type de retour de l'opération de modèles *op* est un type de modèles *MT*, une opération de modèles *op*′ ne pourra correspondre à *op* que si son type de retour est un sous-type de *MT* (covariance). De la même manière, les types des paramètres de *op*′ doivent être des super-types des types des paramètres de *op* si ceux-ci sont typés par des types de modèles (contravariance). Enfin, un champ de modèles *a*′ typé par un type de modèles $MT_{a'}$ ne peut correspondre à un champ de modèles *a* typé par un type de modèles MT_a que si $MT_{a'} = MT_a$ (invariance) ou dans le cas où *a* est en lecture seule si $MT_{a'}$ est un sous-type de MT_a (covariance).

Dans notre exemple, il est possible de vérifier la correspondance entre les deux signatures d'opérations de modèles `coloring` des types de modèles $MT_{ColoredGraph}$ et $MT_{RGBColoredGraph}$ en appliquant les règles présentées en Définition 7.2.

7.1.5 Correspondance de classes et instanciation

La correspondance de types objets permet d'assurer la substitution sûre d'un modèle à un autre. Cependant, certaines transformations de modèles créent des éléments de modèles. Il est nécessaire de s'assurer que les éléments créés soient des éléments du bon type en fonction du type du modèle auquel est appliquée la transformation. Par exemple une transformation acceptant en paramètre un modèle typé par $MT_{ColoredGraph}$ et créant un objet typé par *ColoredNode* pour l'ajouter au modèle ne doit pas se comporter de la même manière quand elle est appelée sur un modèle typé par $MT_{RGBColoredGraph}$. Elle doit en effet créer un objet de type *ColoredNode'* et non de type *ColoredNode*.

Pour cela il est nécessaire d'assurer deux propriétés :

- il faut avoir un moyen de créer un objet du bon type, en instanciant une classe qui n'est pas abstraite ;
- il faut s'assurer que les champs obligatoires (i.e., avec une borne inférieure strictement supérieure à 0) de l'objet sont bien initialisés. Par exemple, créer un objet de type *Color'* sans initialiser son champ `encoding` mène à un modèle invalide (qui ne respecte pas les contraintes de son type de modèles et de son métamodèle).

Il existe deux moyens d'assurer ces propriétés : utiliser une Factory ou les prendre en compte dans la correspondance de types objets.

Pour s'assurer que les objets peuvent être créés avec tous leurs champs obligatoires initialisés il est possible d'utiliser une Factory. Il est possible de forcer chaque type de modèles à être accompagné d'une Factory en charge de créer les objets des types objets appartenant au type de modèles. Cette Factory est alors appelée par la transformation en fonction du type du modèle passé en paramètre. Une manière d'implémenter une telle Factory est simplement d'inclure dans le type de modèles une signature d'opération de modèles de création d'objet par type objet. Ainsi les métamodèles qui implémentent ces types de modèles doivent fournir les opérations de création d'objets, qui sont accessibles par les transformations de modèles. De plus, la vérification de la possibilité d'instancier les classes est automatiquement réalisée par la vérification de la correspondance des opérations de modèles entre deux types de modèles.

Certains langages ne séparent l'interface de l'implémentation qu'au niveau de l'outillage (e.g., du compilateur), i.e., ne fournissent à l'utilisateur que le moyen de définir et d'utiliser des classes, dont sont ensuite extrait les types.

Un autre moyen de s'assurer que les objets peuvent être créés avec tous leurs champs obligatoires initialisés dans ces langages est de prendre en compte certaines informations relatives aux classes dans la correspondance de types objets. En effet, dans de tels langages, les types (objets ou de modèles) sont uniquement les types exacts extraits des classes ou des métamodèles. Il n'existe donc qu'une classe implémentant un type objet donné. Dans cette situation, il est possible pour le type objet de savoir si la classe dont il est extrait est abstraite ou non. De plus, les champs obligatoires des objets sont déclarés par les types objets. Il est donc possible de savoir, lorsque l'on vérifie si T' correspond à T, si T possède les mêmes champs obligatoires que T'. Si c'est le cas une transformation créant un objet de type T doit prévoir d'initialiser ces champs, et pourra donc créer un objet de type T' valide.

La Définition 7.4 présente une extension de la correspondance de types objets pour prendre en compte l'instanciation dans un contexte ou chaque type n'est implémenté que par une classe. Deux règles sont ajoutées par rapport à la Définition 7.1 : l'une qui compare l'attribut *isAbstract*

des classes liées aux types objets comparés (cf. Définition 7.1.ii) ; et l'autre qui assure que, pour chaque champ obligatoire de T', T possède un champ équivalent (cf. Définition 7.1.v).

Définition 7.4. ***(Extension de la correspondance de types objets pour l'instanciation)*** Le type objet $T' \in MT'$ correspond au type objet $T \in MT$ (noté $T' <\# T$) ssi :

i $T.name = T'.name$

ii $T'.class.isAbstract \Rightarrow T.class.isAbstract$

iii $\forall op \in T.ownedSignature, \exists op' \in T'.ownedSignature \wedge op' <\# op$

iv $\forall a \in T.ownedProperty, \exists a' \in T'.ownedProperty \wedge a' <\# a$

v $\forall a' \in T'.ownedProperty$ tel que $a'.lower > 0$, $\exists a \in T.ownedProperty \wedge a' <\# a$

7.1.6 Classification des relations de sous-typage de types de modèles

Les trois relations de correspondance (de types objets, de champs et de signatures d'opérations) permettent de déterminer si la substitution des constituants d'un modèle est sûre dans le cadre d'une substitution de modèles, i.e., d'une substitution de groupes (cf. Section 4.3). Il est donc possible de construire des relations de sous-typage entre types de modèles s'appuyant sur ces relations de correspondance.

Dans cette section, nous présentons deux critères selon lesquels nous différencions ces relations de sous-typage entre types de modèles : le contexte d'utilisation de la relation de sous-typage et la présence ou non d'hétérogénéités structurelles entre le super-type et le sous-type. Le contexte d'utilisation de la relation de sous-typage permet de différencier la substituabilité totale d'un modèle m' à un modèle m, où m' peut être utilisé de manière sûre dans tous les cas possibles d'utilisation de m, de la substituabilité partielle, où m' ne peut être substitué à m que dans un nombre de cas d'utilisation restreint. Les hétérogénéités structurelles désignent les différences dans la manière d'exprimer la même information au sein de deux langages de modélisation dédiés.

7.1.6.1 Contexte d'utilisation

La notion de contexte d'utilisation (usage context en anglais) a été introduite par Kühne afin de définir les cas dans lesquels la substitution de deux modèles dont les métamodèles sont différents est sûre, alors qu'elle n'est pas sûre dans le cas général [Küh13]. Le contexte d'utilisation est un moyen de restreindre les possibilités de substitution aux cas sûrs. Nous distinguons donc deux formes de substituabilité, et donc de sous-typage : la substituabilité totale, qui autorise une substitution quelque soit le contexte ; et la substituabilité partielle, qui n'autorise une substitution que dans un contexte donné.

Dans le cadre de cette thèse nous considérons comme contexte d'utilisation l'utilisation d'un modèle comme paramètre d'une transformation de modèles. Il faut donc assurer que tout modèle passé en paramètre de cette transformation de modèles (i) possède toutes les informations nécessaires à l'exécution de la transformation, et (ii) ne peut pas être modifié par la transformation d'une manière qui le rendrait incompatible vis-à-vis de son métamodèle ou de ses types de modèles.

Sous-typage total Quand un modèle typé par MT' peut être utilisé dans tous les contextes d'utilisation dans lesquels un modèle typé par MT est attendu, nous parlons de substituabilité totale. Par conséquent, une relation de sous-typage qui garantit une substituabilité totale est une relation de sous-typage totale.

Définition 7.5. ***(Sous-type total)*** *MT'* est un sous-type total de *MT* si les modèles typés par *MT'* peuvent être utilisés de manière sûre dans tous les contextes d'utilisation où un modèle typé par *MT* est attendu.

Sous-typage partiel Inversement, une relation de sous-typage partielle permet à un modèle typé par *MT'* d'être utilisé dans un contexte d'utilisation donné (i.e., d'être passé en paramètre d'une transformation de modèles donnée) où un modèle typé par *MT* est attendu, mais pas dans tous les contextes où un modèle typé par *MT* est attendu.

Typiquement, une relation de sous-typage partielle permet à un modèle typé par *MT'* d'être substitué à un modèle *m* typé par *MT* dans le cadre de l'appel `Transfo(m)` si *MT'* contient les informations requises par `Transfo`, même si *MT'* n'est pas un sous-type total de *MT*.

Définition 7.6. ***(Sous-type partiel)*** *MT'* est un sous-type partiel de *MT* dans le contexte de l'utilisation d'une transformation de modèles f si les modèles typés par *MT'* peuvent être passés en paramètre de f de manière sûre là où un modèle typé par *MT* est attendu.

MT' est un sous-type partiel de *MT* par rapport à f si *MT'* est un sous-type total de MT_f, où MT_f est un type de modèles qui ne contient que les informations nécessaires pour appliquer f de manière sûre et tel que *MT* est un sous-type total de MT_f. Nous appelons MT_f le type de modèles effectif de f.

Définition 7.7. ***(Type de modèles effectif)*** Le type de modèles effectif MT_f d'une transformation de modèles f extrait du type de modèles *MT* est le type de modèles qui contient toutes les informations requises par f et tel que *MT* est un sous-type total de MT_f.

Le type de modèles effectif d'une transformation de modèles f peut être obtenu à l'aide d'une fonction qui analyse le type de modèles et en extrait le sous-ensemble nécessaire à l'exécution de f.

Définition 7.8. ***(Extraction de type de modèles effectif)*** La fonction d'extraction de type de modèles effectif est une fonction *extractEffectiveMT* de $\mathbf{MT} \times \mathbf{F}$ vers **MT**, avec **MT** l'ensemble de tous les types de modèles, et **F** l'ensemble des transformations de modèles, telle que MT_f = *extractEffectiveMT*(*MT*, f) est le type de modèles effectif de la transformation de modèles f extrait du type de modèles *MT*.

Par exemple, les figures 7.6 et 7.8 présentent deux types de modèles de graphes de flot de contrôle, l'un pouvant contenir des instructions entières et booléennes ($CFG_{IntBool}$), l'autre des instructions flottantes uniquement (CFG_{Float}).

Une transformation de modèles *Reach* calculant une analyse d'atteignabilité des nœuds d'un graphe de flot de contrôle définie sur $CFG_{IntBool}$ ne pourrait pas être réutilisée sur les modèles typés par CFG_{Float} dans un système de types n'autorisant que les relations de sous-typage totales. En effet CFG_{Float} ne fournit pas de garantie que les modèles qu'il type puissent être utilisés dans tous les contextes où un modèle typé par $CFG_{IntBool}$ est attendu. Plus particulièrement, une transformation de modèles tentant d'accéder à un objet de type *IntInstruction* provoquerait une erreur à l'exécution si elle était appelée sur un modèle typée par CFG_{Float}.

Cependant, l'analyse d'atteignabilité des nœuds n'a pas besoin d'accéder aux instructions contenues par les nœuds, mais uniquement aux nœuds et à leurs relations (représentées par l'association bidirectionnelle `next`/`previous`). Les informations concernant les types d'instructions contenues par les graphes de flot de contrôle peuvent donc être ignorées dans le contexte de l'utilisation de l'analyse d'atteignabilité. Le type de modèles effectif de la transformation *Reach* est donc le type de modèles CFG_{Reach} présenté en figure 7.7. CFG_{Reach} représente le sous-ensemble de $CFG_{IntBool}$ nécessaire à l'exécution de l'analyse d'atteignabilité.

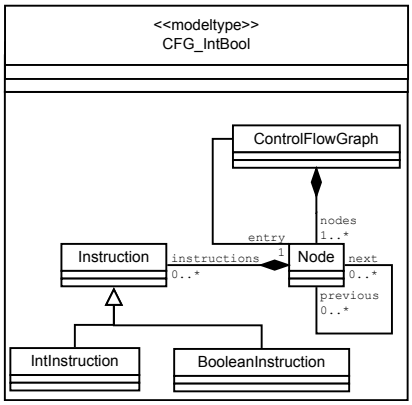

FIGURE 7.6 – Type de modèles $CFG_{IntBool}$ de graphes de flot de contrôle contenant des instructions entières et booléennes.

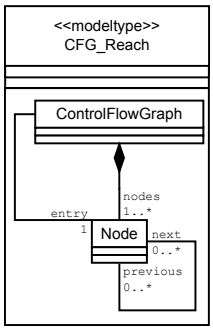

FIGURE 7.7 – Type de modèles effectif de l'analyse d'atteignabilité CFG_{Reach} extrait de $CFG_{IntBool}$.

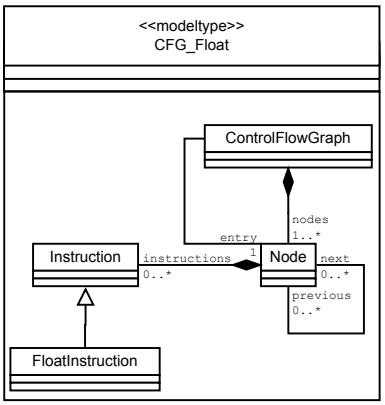

FIGURE 7.8 – Type de modèles CFG_{Float} de graphes de flot de contrôle contenant des instructions flottantes.

Si CFG_{Float} possède un type objet correspondant à chacun des types objets de CFG_{Reach}, il est sûr de substituer les modèles typés par CFG_{Float} aux modèles typés par $CFG_{IntBool}$ dans le contexte de la transformation *Reach*.

Il est possible d'obtenir le sous-ensemble nécessaire à l'exécution d'une transformation f à l'aide de l'empreinte de f. L'empreinte de f est l'ensemble des types, champs et opérations touchés ou appelés pendant l'exécution de f [JGB11]. Cette empreinte peut être estimée statiquement en analysant le code de f, ou calculée dynamiquement en utilisant une trace de l'appel de f sur un modèle donné. L'empreinte dynamique est plus précise car elle ne contient que les types et les champs qui ont été réellement touchés par l'exécution, alors que l'empreinte statique contient tous les types, champs et opérations qui pourraient être touchés. Dans les deux cas, le calcul de l'empreinte d'une transformation de modèles nécessite d'accéder à son implémentation, et n'est donc pas possible uniquement à partir d'un type de modèles.

7.1.6.2 Hétérogénéités structurelles

Les hétérogénéités structurelles entre deux types de modèles ou entre deux métamodèles sont dues à la représentation de la même information sous différentes formes, i.e., l'utilisation de différentes constructions de modélisation pour représenter la même information [KKR+08]. On a donc une équivalence sémantique entre deux types de modèles (i.e., la même information est exprimée), mais au travers d'une structure différente (i.e., les hétérogénéités structurelles).

Par exemple, les figures 7.9 et 7.10 présentent deux types de modèles de graphes de flot de contrôle présentant des hétérogénéités structurelles. Le premier type de modèles, CFG_{Type} représente les arcs des graphes de flot de contrôle à l'aide d'un type `Edge` (cf. figure 7.9), alors que le deuxième type de modèles, CFG_{Asso} représente ces mêmes arcs à l'aide d'une association bidirectionnelle sur le type `Node` (cf. figure 7.10).

Malgré les hétérogénéités structurelles, qui empêchent une relation de correspondance entre les deux types objets `Node`, les deux types de modèles représentent bien la même information. Il semble donc raisonnable de vouloir réutiliser des algorithmes écrits pour un type de modèles sur l'autre. Une manière de contourner ce problème est d'utiliser des opérateurs d'adaptation de modèles, qui permettent de convertir automatiquement un modèle d'un type en un modèle d'un autre type. Les opérateurs d'adaptation permettent de retrouver structurellement une

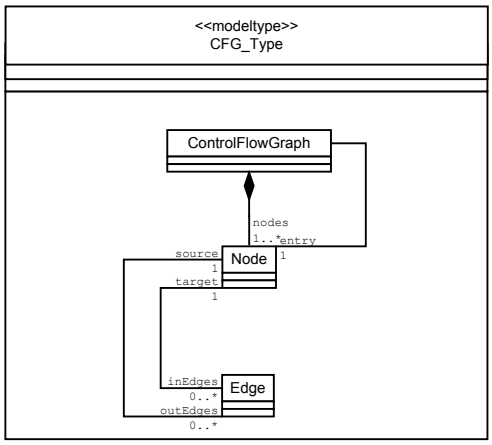

FIGURE 7.9 – Type de modèles CFG_{Type} de graphes de flot de contrôle représentant les arcs à l'aide d'un type.

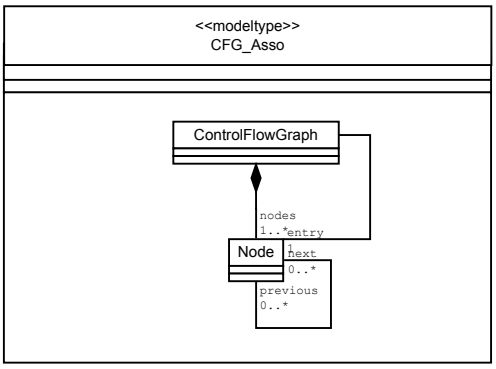

FIGURE 7.10 – Type de modèles CFG_{Asso} de graphes de flot de contrôle représentant les arcs à l'aide d'une association.

équivalence sémantique connue entre types de modèles.

Adaptation de modèles L'adaptation de modèles est le processus de récupération de l'information sous la forme attendue à partir d'un modèle où l'information est présente sous une autre forme. L'adaptation de modèles consiste à adapter un modèle m' en un modèle m à travers lequel il est possible d'accéder à l'information désirée sous la forme attendue. Ainsi, une adaptation de modèles est un moyen de créer une relation de sous-typage entre deux types de modèles.

Une adaptation de modèles est une fonction définie au niveau des types de modèles et appliquée sur des modèles. Une adaptation de modèles prend en paramètre un modèle m' typé par MT' et retourne un modèle m contenant la même information, mais sous la forme définie par MT, i.e., un modèle dont le type est sous-type de MT.

Définition 7.9. ***(Adaptation de modèles)*** Une adaptation de modèles est une fonction $adapt_{MT}$ de MT' vers MT'', où MT, MT' et MT'' sont des types de modèles et telle que MT'' est un sous-type de MT.

Une adaptation de modèles peut être implémentée sous la forme d'une transformation de modèles de MT' à MT, auquel cas $MT'' = MT$. Par exemple, une transformation de CFG_{Type} à CFG_{Asso} peut servir à implémenter une adaptation $adapt_{CFG_{Asso}}$.

Une autre façon d'implémenter une adaptation de modèles est d'ajouter les types, champs, et signatures d'opérations manquants à MT' pour créer un nouveau type MT''. De cette manière, MT'' possède un type objet (respectivement un champ de modèles ou une signature d'opération de modèles) pour chaque type objet (respectivement chaque champ de modèles ou chaque signature d'opération de modèles) de MT' et pour chaque type objet (respectivement chaque champ de modèles ou chaque signature d'opération de modèles) de MT.

Le Listing 7.1 présente un aspect Kermeta déclarant deux champs dérivés (des champs dont la valeur est calculée par rapport à d'autres champs ou opérations) `next` et `previous` à l'aide du mot-clé `property`. Ces champs dérivés permettent d'accéder aux successeurs et aux prédécesseurs d'un nœud dans le graphe de flot de contrôle. Le champ `next` est calculé en collectant les cibles des arcs sortants (`outEdges`) du nœud (l'opération permettant de calculer

Listing 7.1 – Aspect Kermeta déclarant les champs dérivés `next` et `previous` calculés à partir de l'ensemble des arcs d'un nœud.

```
1  aspect class Node {
2    property next: Node[0..*]
3    _get is do
4      result := OrderedSet[Node].new
5      outEdges.each{e|
6        result.add(e.target)
7      }
8    end
9
10   property previous: Node[0..*]
11   _get is do
12     result := OrderedSet[Node].new
13     inEdges.each{e|
14       result.add(e.source)
15     }
16   end
17 }
```

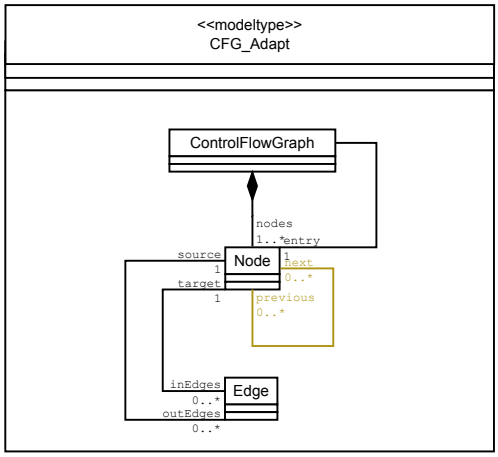

FIGURE 7.11 – Type de modèles CFG_{Adapt} résultat de l'ajout de champs dérivés sur CFG_{Type}.

la valeur du champ dérivé est déclarée à l'aide du mot-clé `_get`). De la même manière, le champ `previous` est calculé à partir des sources des arcs entrant (`inEdges`) du nœud.

L'ajout de ces champs à la classe `Node` permet au type objet qui est extrait de cette classe de correspondre au type objet `Node` du type de modèles CFG_{Asso}. Le type de modèles CFG_{Adapt} résultat de l'ajout de ces champs dérivés est présenté en Figure 7.11, où les champs ajoutés sont représentés en couleur.

Cette approche est celle suivie par Moha et al. pour réutiliser des transformations de modèles entre plusieurs langages orientés-objet [MMBJ09, SMM+12]. En utilisant des champs dérivés, Moha et al. ajoutent à un métamodèle existant les éléments nécessaires pour obtenir une équivalence structurelle, sans pour autant modifier sa sémantique.

Adaptation bidirectionnelle de modèles Une adaptation bidirectionnelle de modèles couple une adaptation "en avant" de *MT'* à *MT* et une adaptation "en arrière" de *MT* à *MT'*.

Une adaptation bidirectionnelle peut être nécessaire afin de réutiliser une transformation de modèles endogène. En effet, dans le cas où la transformation de modèles endogène modifie ou produit un modèle typé par le type de modèles adapté, le résultat doit être exprimé sous la forme du type de modèles original.

Par exemple, si une transformation de modèles définie sur CFG_{Asso} ajoutait un successeur à un nœud par le biais de l'association `next`, il serait nécessaire de refléter cet ajout dans un modèle typé par CFG_{Type}, i.e., de créer un objet `Edge` liant le nœud à son successeur.

Les adaptations "en avant" et "en arrière" forment ensemble une adaptation bidirectionnelle, qui permet l'adaptation d'un modèle typé par *MT'* dans une forme qui correspond au type de modèles attendu *MT*, mais également d'exprimer le résultat dans le modèle original, selon la structure définie par *MT'*. Une adaptation aller-retour, i.e., l'application de l'adaptation "en avant" à un modèle, suivie de l'adaptation "en arrière" au résultat devrait conduire à un modèle inchangé. Foster et al. ont défini des règles permettant d'assurer le comportement des adaptations bidirectionnelles, qu'ils nomment lenses [FGM+07]. Nous reproduisons ces règles en Définition 7.10.

Définition 7.10. ***(Adaptation bidirectionnelle de modèles)*** Une adaptation bidirectionnelle de modèles $adapt_{MT}$ entre les types de modèles *MT'* et *MT* est constituée d'une fonction $adapt_{MT} \nearrow$

de MT' vers MT'' et d'une fonction $adapt_{MT} \searrow$ de $MT' \times MT''$ vers MT', où MT'' est un type de modèles tel que MT'' est un sous-type de MT et :

- $adapt_{MT} \nearrow (adapt_{MT} \searrow (m', m'')) = m'', \forall (m', m'') \in MT' \times MT''$
- $adapt_{MT} \searrow (m', adapt_{MT} \nearrow (m')) = m', \forall m' \in MT'$ [FGM+07]

Les adaptations bidirectionnelles peuvent être implémentées au travers de transformations bidirectionnelles. Les transformations bidirectionnelles sont étudiées dans différentes disciplines de l'informatique (e.g., l'ingénierie dirigée par les modèles, les transformations de graphes ou les bases de données) pour synchroniser deux structures de données (une source et une vue) [CFH+09, HSST11]. Dans notre cas, la source est le modèle typé par MT' trouvé dans un contexte où un modèle typé par MT (la vue) est attendu.

Les adaptations bidirectionnelles peuvent également être implémentées à l'aide de champs dérivés, auquel cas ceux-ci doivent déclarer à la fois le moyen de les accéder (adaptation "en avant") et le moyen de les modifier (adaptation "en arrière"). Pour les champs dérivés dont la borne supérieure est 1, il faut donc fournir un accesseur (getter) et un manipulateur (setter). Pour les champs dérivés dont la borne supérieure est supérieure à 1 (i.e., les collections dérivées), il faut fournir un accesseur, ainsi que les moyens de modifier la collection obtenue tout en affectant les champs à partir de laquelle elle est calculée.

Par exemple, le listing 7.2 étend le listing 7.1 en déclarant le moyen d'ajouter (opération `_add`) et de retirer (opération `_remove`) un successeur à un nœud. Pour ajouter un successeur à un nœud, il faut créer un arc les reliant, tandis que pour retirer un successeur à un nœud il faut retirer l'arc les reliant de l'ensemble des arcs sortant du nœud et de l'ensemble des arcs entrants du successeur.

Les opérations de modifications des collections dérivées à fournir dépendent de l'implémentation sous-jacente. Il faut en effet fournir les mêmes opérations de modification de la collection dérivée que celles disponibles sur la collection originale. Dans le cas d'une collection non-ordonnée, les opérations `add` et `remove` peuvent être suffisantes. En effet, les opérations telles que `addAll` et `removeAll` s'appuient sur `add` et `remove`, et les opérations telles que `includes`, ou que celles qui construisent une nouvelle collection (`union` sur deux ensembles) ne nécessitent pas de modifier la collection dérivée. Pour les collections ordonnées, il faut également fournir des opérations telles que `addAt` et `removeAt`, pour que la collection dérivée soit manipulable comme une collection classique.

7.1.7 Définition de quatre relations de sous-typage entre types de modèles

Nous avons défini trois relations de correspondance entre éléments d'un type de modèles : correspondance de types objets (cf. Section 7.1.1), correspondance de signatures d'opérations (cf. Section 7.1.2) et correspondance de champs (cf. Section 7.1.3). Nous avons également proposé deux critères permettant de différencier les relations de sous-typage entre types de modèles : présence d'hétérogénéités structurelles (cf. Section 7.1.6.2) et contexte d'utilisation de la relation de sous-typage (cf. Section 7.1.6.1).

À partir de ces trois relations de correspondance et de ces deux critères, nous définissons quatre relations de sous-typage entre types de modèles. Ces quatre relations de sous-typage fournissent chacune une forme de substituabilité entre modèles et peuvent être ramenées à la première d'entre elles : la relation de sous-typage totale isomorphique.

Dans la suite de cette Section, MT et MT' sont deux types de modèles et **MT** est l'ensemble de tous les types de modèles.

Listing 7.2 – Aspect Kermeta déclarant un champ dérivé `next` permettant une adaptation bidirectionnelle de CFG_{Type} vers CFG_{Asso}.

```
aspect class Node {
  property next : Node[0..*]
  _get is do
    result := Set[Node].new
    outEdges.each{e|
      result.add(e.target)
    }
  end
  _add is do
    var edgeToAdd : Edge init Edge.new
    edgeToAdd.source := self
    edgeToAdd.target := _value
  end
  _remove is do
    var e : Edge init outEdges.detect{e|e.target == _value}
    _value.inEdges.remove(e)
    outEdges.remove(e)
  end
}
```

7.1.7.1 Sous-typage total isomorphique

La première relation de sous-typage entre types de modèles est la relation de sous-typage totale isomorphique, à laquelle les trois autres se ramènent. MT' est sous-type total isomorphique de MT s'il contient un type objet correspondant à chaque type objet de MT (cf. Section 7.1.1).

Pour établir une relation de sous-typage totale isomorphique entre deux types de modèles, nous calculons donc un isomorphisme de sous-graphe entre ces deux types de modèles [1]. Chaque type de modèles est alors considéré comme un graphe dont les nœuds sont des types objets et les arcs les associations définies par les champs de ces types. Afin d'établir la relation de sous-typage, le sous-type potentiel doit contenir un sous-graphe isomorphe au super-type : l'ensemble des types objets correspondants aux types objets du super-type.

De plus MT' doit contenir un champ de modèles correspondant à chaque champ de modèles de MT (cf. Section 7.1.3) et une signature d'opération de modèles correspondante à chaque signature d'opération de modèles de MT (cf. Section 7.1.2).

Définition 7.11. ***(Relation de sous-typage totale isomorphique)*** La relation de *sous-typage totale isomorphique* est une relation binaire $<:$ sur **MT**, tel que $(MT', MT) \in <:$ (noté également $MT' <: MT$) ssi :

i $\forall T \in MT.ownedObjectType, \exists T' \in MT'.ownedObjectType$ tel que $T' <: \#T$;

ii $\forall p \in MT.ownedProperty, \exists p' \in MT'.ownedProperty$ tel que $p' <: \#p$;

iii $\forall op \in MT.ownedSignature, \exists op' \in MT'.ownedSignature$ tel que $op' <: \#op$.

7.1.7.2 Sous-typage partiel isomorphique

Une relation de sous-typage partielle isomorphique par rapport à la transformation de modèles f existe entre MT' et MT si on peut utiliser les modèles typés par MT' là où des modèles typés par MT sont attendus, dans le contexte de l'utilisation de f. Le type de modèles effectif de f extrait de MT contient tous les types objets, tous les champs de modèles et toutes les signatures d'opérations de modèles nécessaires à l'utilisation de f.

1. Pour plus de détails sur l'isomorphisme de sous-graphe, nous reportons le lecteur à la définition initiale de Cook [Coo71].

MT' est un sous-type partiel isomorphique de MT par rapport à f si MT' est un sous-type total isomorphique de MT_f, MT_f étant le type de modèles effectif de f extrait de MT. MT' est donc un sous-type partiel isomorphique de MT par rapport à f s'il contient un type objet (respectivement un champ de modèles et une signature d'opération de modèles) correspondant à chacun des types objets de MT_f (respectivement à chacun des champs de modèles et à chacune des signatures d'opérations de modèles de MT_f).

Définition 7.12. ***(Relation de sous-typage partielle isomorphique)*** La relation de sous-typage *partielle isomorphique* par rapport à la transformation de modèles f est une relation binaire $<:_f$ sur **MT** telle que $(MT', MT) \in <:_f$ (noté également $MT' <:_f MT$) ssi $MT' <: extractEffectiveMT(MT, f)$.

7.1.7.3 Sous-typage total non-isomorphique

MT' est un sous-type total non-isomorphique de MT s'il existe une adaptation de modèles capable d'adapter tous les modèles typés par MT' en modèles typés par MT. Cette adaptation de modèles doit être bidirectionnelle, sans quoi il serait impossible de réutiliser les transformations de modèles endogènes manipulant des modèles typés par MT, et la relation de sous-typage ne serait donc pas totale.

Définition 7.13. ***(Relation de sous-typage totale non-isomorphique)*** La relation de sous-typage *totale non-isomorphique* est une relation binaire $\lesssim$: sur **MT** telle que $(MT', MT) \in \lesssim$: (noté également $MT' \lesssim : MT$) ssi $\exists adapt_{MT}$ une adaptation bidirectionnelle de MT' vers MT'' telle que $MT'' <: MT$.

7.1.7.4 Sous-typage partiel non-isomorphique

Le type de modèles MT' est un sous-type partiel non-isomorphique du type de modèles MT par rapport à la transformation de modèles f si il existe une adaptation en mesure d'adapter un modèle typé par MT' en un modèle typé par un sous-type total isomorphique de MT_f, le type de modèles effectif de f extrait de MT. Cette adaptation doit être bidirectionnelle si f est une transformation de modèles endogène.

Définition 7.14. ***(Relation de sous-typage partielle non-isomorphique)*** La relation de sous-typage *partielle non-isomorphique* par rapport à la transformation de modèles f est une relation binaire $\lesssim :_f$ sur **MT** telle $(MT', MT) \in \lesssim :_f$ (noté également $MT' \lesssim :_f MT$) ssi $\exists adapt_{MT}$ une adaptation de MT vers MT'' telle que $MT'' <: extractEffectiveMT(MT, f)$ et que $adapt_{MT}$ est une adaptation bidirectionnelle si f est une transformation de modèles *endogène*.

7.1.8 Syntaxe abstraite de METAL pour les relations de sous-typage entre types de modèles

La Figure 7.12 présente le sous-ensemble de la syntaxe abstraite de METAL dédié aux relations de sous-typage entre types de modèles. Une relation de sous-typage (`ModelSubtyping`) relie un sous-type (`subType`) à son super-type (`superType`). Une relation de sous-typage peut être non-isomorphique, auquel cas elle possède une fonction d'adaptation (`ModelAdaptation`), ou partielle, auquel cas elle référence un ensemble de transformations de modèles (`Operation`), à partir des corps desquelles il est possible d'extraire un type de modèles effectif.

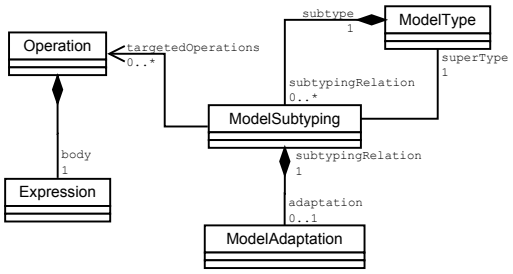

FIGURE 7.12 – Sous ensemble de la syntaxe abstraite du métalangage METAL exposant les concepts nécessaires aux relations de sous-typage entre types de modèles.

7.2 Relation d'héritage entre métamodèles

Dans les systèmes de types orientés-objet, les relations d'héritage permettent de ne pas créer les nouvelles classes de zéro, mais de réutiliser la structure et le comportement définis au sein d'une classe existante. Nous proposons d'amener cette facilité de réutilisation au niveau des métamodèles, pour permettre de créer un nouveau métamodèle, le sous-métamodèle, en réutilisant la structure et le comportement du métamodèle hérité, le super-métamodèle. Une relation d'héritage entre métamodèles peut être couplée à une relation de sous-typage entre types de modèles. Certaines relations d'héritage autorisent également la redéfinition de tout ou partie des éléments hérités.

Dans cette section, nous décrivons les différents mécanismes possibles pour l'héritage entre métamodèles et leurs répercussions sur les relations de sous-typage entre types de modèles.

7.2.1 Héritage, extension et redéfinition

L'extension du sous-métamodèle correspond à l'ajout de classes, d'opérations ou de champs de modèles par rapport au super-métamodèle. La redéfinition permet de modifier dans le sous-métamodèle certains des éléments hérités du super-métamodèle.

L'héritage avec sous-typage garantit que le type exact du sous-métamodèle sera un sous-type du type exact du super-métamodèle. Ainsi, les modèles instances du sous-métamodèle sont typés par le type exact du super-métamodèle, et donc substituables aux modèles instances du super-métamodèle. Pour assurer que le type exact du sous-métamodèle est un sous-type total isomorphique du type exact du super-métamodèle, la redéfinition des éléments du sous-métamodèle (classes, champs et opérations de modèles) ne doit pas briser les relations de correspondance (cf. Section 7.1). L'extension du sous-métamodèle par l'ajout de classes, d'opérations de modèles et de champs de modèles n'affecte pas les relations de sous-typage entre types de modèles. En effet les modèles issus du sous-métamodèle étendu répondent toujours à l'interface exposée par le type de modèles extrait par le super-métamodèle. La Table 7.1 résume les différentes redéfinitions autorisées par l'héritage avec sous-typage qui sont présentées dans la suite de cette section.

Si l'héritage n'implique pas le sous-typage, la relation d'héritage est utilisée uniquement pour réutiliser le code appartenant au super-métamodèle, sans définir de relation de substituabilité entre les modèles instances du sous-métamodèle et du super-métamodèle. Dans ce cas, toutes les redéfinitions possibles sont légales, et uniquement limitées par le métalangage (e.g., redéfinition des noms, des types, des visibilités, des multiplicités, suppression d'éléments, etc.).

TABLE 7.1 – Redéfinitions autorisées par la relation d'héritage entre métamodèles avec sous-typage.

Élément du *super-métamodèle*	Redéfinitions possibles de l'élément du *sous-métamodèle*
Champ et champ de modèles	Ajout de champ opposé Champ ordonné
Champ et champ de modèles en lecture seule	Type covariant Champ en lecture et en écriture
Opération et opération de modèles	Type de retour covariant Types des exceptions covariants Redéfinition du corps (avec appel à la super-opération pour les opérations de modèles)
Paramètre	Type contravariant Paramètre non-ordonné

Dans la suite de cette section, *MM'*, le sous-métamodèle, hérite de *MM*, le super-métamodèle. *MT* et *MT'* désignent les types de modèles exacts extraits respectivement de *MM* et de *MM'*.

7.2.2 Héritage de champs de modèles

Les champs de modèles sont définis par les types de modèles et non les métamodèles. C'est pourquoi pour hériter des champs de modèles d'un métamodèle il faut implémenter le type qui déclare ces champs de modèles. Si le sous-métamodèle hérite de l'intégralité des éléments du super-métamodèle, ce type est le type exact du super-métamodèle. Si ce n'est pas le cas (e.g., parce que l'héritage ne garantit pas le sous-typage ou parce qu'il ne garantit qu'un sous-typage partiel), il faut calculer un type de modèles qui définit uniquement les champs hérités.

Seuls quelques changements dans la définition d'un champ de modèles *a* sont légaux dans le cadre d'un héritage avec sous-typage du super-métamodèle *MM* au sous-métamodèle *MM'*.

Si *a* est en lecture seule dans *MM*, il est possible de redéfinir le type de *a* dans *MM'* de manière covariante. En effet, le type d'un champ en lecture seule n'a pas besoin d'être invariant (cf. Section 4.2.1.1). La redéfinition covariante du type d'un champ peut se faire de deux manières :

- soit le nouveau type de *a* dans *MM'* est un sous-type du type de *a* dans *MM* ;
- soit, dans le cas où le type de *a* est un type objet appartenant à *MT*, le nouveau type de *a* dans le sous-métamodèle appartient à *MT'* et correspond au type de *a* dans le super-métamodèle.

Si *a* est en lecture seule dans le super-métamodèle, il est possible de le redéfinir en accès lecture et écriture dans le sous-métamodèle. Si *a* est une collection qui n'est pas ordonnée dans le super-métamodèle, il peut être redéfini comme une collection ordonnée dans le sous-métamodèle. Enfin, si *a* n'a pas de champ opposé, il est possible de le redéfinir en lui ajoutant un champ opposé. Dans ces trois cas, la redéfinition est possible puisque toutes les manipulations effectuées sur *a* définies à partir de *MT* restent valides sur les modèles instances de *MM'* : il est toujours possible de manipuler *a* en lecture uniquement, comme une collection non-ordonnée, ou comme s'il n'avait pas de champ opposé.

Les autres attributs de *a* (nom, multiplicité, unicité et sémantique de composition) doivent rester inchangés, afin de garantir la relation de correspondance entre *a* dans le sous-métamodèle et *a* dans le super-métamodèle.

7.2.3 Héritage d'opérations de modèles

Les opérations de modèles peuvent être héritées par référence au super-métamodèle et à un type de modèles contenant les signatures des opérations à hériter. En effet, tout comme les champs de modèles, les signatures des opérations de modèles sont définies au sein des types de modèles. Hériter de l'ensemble des opérations de modèles d'un métamodèle revient donc à référencer ce métamodèle (pour avoir accès au corps des opérations héritées) et à implémenter son type exact. Dans le cas où *MM*' n'hérite pas de l'ensemble des opérations de modèles de *MM*, il faut calculer un type de modèles définissant les signatures des opérations héritées.

La redéfinition d'une opération de modèles *op* peut survenir à deux niveaux : au niveau de la signature de l'opération, y compris des paramètres, et au niveau du corps de l'opération.

Le type de retour d'une opération de modèles peut être redéfini de manière covariante (cf. Section 4.2.1.1) :

- soit par un sous-type ;
- soit par un type correspondant, si les deux types de retour sont des types objets et appartiennent respectivement à *MT*' et à *MT*.

De la même manière il est possible de redéfinir le type des exceptions levées par *op*' de manière covariante, ou de retirer certaines exceptions de l'ensemble des exceptions levées. Il n'est par contre pas possible d'ajouter de nouvelles exceptions. En effet, une telle exception pourrait être levée par un modèle instance de *MM*' et ne pas être récupérée, puisqu'une transformation de modèles définie à partir de *MT* ne pourrait pas connaître cette exception.

Les types des paramètres d'une opération de modèles peuvent être redéfinis de manière contravariante (cf. Section 4.2.1.1) :

- si le type du paramètre défini dans *MM* appartient à *MT*, il peut être redéfini par un type auquel il correspond et qui appartient à *MT*' ;
- sinon il peut être redéfini par un super-type.

Si le paramètre est défini dans *MM* comme une collection ordonnée, il peut être redéfini comme une collection non-ordonnée dans *MM*'.

Le corps d'une opération de modèles peut-être redéfini pour peu qu'il soit bien typé, i.e., qu'il ne contienne pas d'erreur de types, et qu'il retourne un objet ou un modèle typé par le type de retour de l'opération. Il est possible de définir des mécanismes afin d'appeler la (ou les) super-opération, à condition que cet appel soit lui aussi bien typé, i.e., que les paramètres passés soient du type attendu.

Aucun autre attribut de l'opération ou de ses paramètres (e.g., leur nom, multiplicité, etc.) ne peut être redéfini sans briser la correspondance d'opérations, et donc le sous-typage total isomorphique entre le type exact du sous-métamodèle et le type exact du super-métamodèle.

7.2.4 Héritage de classes

La redéfinition des champs et des opérations des classes héritées par le sous-métamodèle obéit aux mêmes contraintes que la redéfinition des champs et des opérations de modèles, à une exception près : les appels aux super-opérations.

Lors d'un héritage de métamodèle, aucun lien n'est conservé entre les classes du super et du sous-métamodèles, afin de ne pas créer de dépendance de la modélisation in-the-small (les classes) vers la modélisation in-the-large (les métamodèles et leur relation d'héritage). En effet, manipuler ces liens (e.g., au travers d'un appel à une super-opération) demande d'avoir une connaissance de la couche de modélisation in-the-large du langage de modélisation dédié à un niveau de modélisation in-the-small (la définition des classes et de leurs opérations).

Sans lien entre ces classes un appel à une super-opération au travers de l'héritage de métamodèle n'est pas possible, puisque la super-opération n'est pas connue. C'est pourquoi, si le corps d'une opération peut être redéfini, il ne peut pas faire d'appel aux super-opérations au travers de l'héritage de métamodèle. Il reste cependant possible d'appeler une super-opération obtenue au travers de l'héritage entre classes, i.e., de l'héritage in-the-small.

Il est également possible de redéfinir les classes en leur ajoutant des opérations et des champs. La seule contrainte concernant ces redéfinitions dépend de la politique choisie pour l'instanciation des objets. En effet, ajouter un champ obligatoire (i.e., dont la borne inférieure est supérieure ou égale à 1) peut briser la correspondance des types objets dans certains cas (cf. Section 7.1.5).

7.2.5 Héritage non-isomorphique, héritage partiel

Il est possible de définir une relation d'héritage non-isomorphique, i.e., une relation autorisant des redéfinitions qui brise les relations de correspondance, mais qui conserve une relation de sous-typage non-isomorphique entre *MT*' et *MT*. Pour cela les redéfinitions des éléments de *MM*' doivent être définies comme des adaptations bidirectionnelles (cf. Section 7.1.6.2). En effet, si la redéfinition (e.g., le renommage d'une opération) est accompagnée de l'adaptation inverse (le renommage inverse), cette adaptation inverse permet de conserver une relation de sous-typage non-isomorphique entre *MT*' et *MT*.

Il est également possible de définir une relation d'héritage partielle, i.e., une relation d'héritage par laquelle le sous-métamodèle n'hérite que d'une partie des éléments du super-métamodèle et qui garantit une relation de sous-typage partielle entre *MT*' et *MT*. Pour cela, il faut calculer les empreintes de chaque opération héritée. En effet hériter d'une opération qui manipule un champ ou qui appelle une autre opération peut mener à un sous-métamodèle invalide si le champ manipulé ou l'opération appelée n'est pas également hérité.

7.2.6 Héritage entre métamodèles et types chemin-dépendants

Lors de l'héritage entre métamodèles, les références à des types chemin-dépendants restent inchangées. Par contre, le type de modèles utilisé pour le type chemin-dépendant peut changer. Par exemple, si le type de modèles utilisé est le type exact du métamodèle dans lequel le type chemin-dépendant est utilisé, ce type varie avec l'héritage. Par conséquent, la valeur du type chemin-dépendant varie également avec l'héritage entre métamodèles (cf. Section 4.3.2).

7.2.7 Syntaxe abstraite de METAL pour les relations d'héritage entre métamodèles

La Figure 7.13 présente le sous-ensemble de la syntaxe abstraite de METAL dédié aux relations d'héritage entre métamodèles. Une relation d'héritage (`ModelInheritance`) relie un sous-métamodèle (`subMetamodel`) à son super-métamodèle (`superMetamodel`). Une relation d'héritage peut impliquer une relation de sous-typage (`implies`).

7.3 Conception d'un système de types orienté-modèle

Définir des relations de sous-typage et d'héritage n'est pas suffisant pour construire un système de types. En effet, un système de types implémente une ou plusieurs de ces relations mais fournit également des moyens de déclarer et de vérifier ces relations (cf. Section 4.2.1.2). Nous abordons ici les impacts spécifiques aux types de modèles des déclarations (cf. Section 7.3.1) et

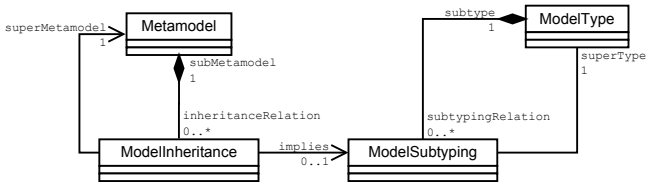

FIGURE 7.13 – Sous ensemble de la syntaxe abstraite du métalangage METAL exposant les concepts nécessaires aux relations d'héritage entre métamodèles.

vérifications (cf. Section 7.3.2) des relations de sous-typage sur la conception d'un système de types orienté-modèle.

Selon les relations entre langages de modélisation dédiés implémentées et selon les moyens de les déclarer et de les vérifier, il est possible de concevoir différents systèmes de types orientés-modèle. Ces différents systèmes de types forment une famille de systèmes de types, que nous présentons en Section 7.3.3.

7.3.1 Déclaration des relations de sous-typage entre types de modèles

La façon de déclarer les relations de sous-typage peut affecter les possibilités que ces relations offrent à travers le système de types. Par exemple, une relation de sous-typage non-isomorphique peut être déclarée implicitement. À cette fin, un outil doit être capable de déduire les adaptations nécessaires, par exemple à l'aide de motifs connus pour être bidirectionnels. Toutefois, un mécanisme d'adaptation implicite sera plus limité en termes d'adaptations possibles qu'un mécanisme explicite, qui permet à l'utilisateur de définir ses adaptations en fonction de sa connaissance des deux types de modèles concernés. D'autre part, un mécanisme d'adaptation explicite doit fournir les constructions syntaxiques pour construire une adaptation et doit supporter des analyses appropriées pour s'assurer que l'adaptation est sûre.

Enfin, la déclaration d'une relation de sous-typage explicite à la définition peut accompagner la déclaration d'une relation d'héritage entre métamodèles (cf. Section 7.2). En effet, il n'est pas possible de déclarer qu'un métamodèle hérite d'un autre une fois qu'il est défini, pas plus qu'il n'est possible d'inférer le besoin d'une relation d'héritage entre deux métamodèles. Si la relation d'héritage garantit le sous-typage, elle fait office de déclaration explicite, à la définition du type exact qui est extrait du sous-métamodèle.

7.3.2 Vérification des relations de sous-typage entre types de modèles

Tout comme les relations de sous-typage entre types objets, les relations de sous-typage entre types de modèles peuvent être vérifiées statiquement ou dynamiquement. Le fait de vérifier à la conception (statiquement) ou à l'exécution (dynamiquement) ces relations n'a pas d'impact sur l'outil de vérification des relations de sous-typage à proprement parler, mais uniquement sur le moment où il est appelé.

Comme pour la vérification des relations de sous-typage entre types objets, une vérification statique permet la détection plus précoce d'erreurs et facilite la mise en place d'outils comme l'auto-complétion ou certaines optimisations qui s'appuient sur les types. Par contre, la vérification dynamique a accès à plus d'informations sur les types d'un modèle. Ces informations peuvent être utilisées pour autoriser des relations de sous-typage interdites statiquement. De plus, la vérification dynamique peut utiliser des informations relatives à l'exécution de transformations pour calculer un type de modèles effectif plus précis dans le cadre d'une relation

TABLE 7.2 – Fonctions permettant de ramener les quatre relations de sous-typage entre types de modèles à la relation de sous-typage *totale isomorphique*

Relation de sous-typage	Fonctions ramenant à la relation de sous-typage *totale isomorphique*
Totale isomorphique (notée $MT' <: MT$)	–
Partielle isomorphique (notée $MT' <:_f MT$)	$extractEffectiveMT : \mathbf{MT} \times \mathbf{F} \rightarrow \mathbf{MT}$ avec $MT' <: extractEffectiveMT(MT, f)$ et $MT <: extractEffectiveMT(MT, f)$
Totale non-isomorphique (notée $MT' \lesssim : MT$)	$adapt_{MT} : MT' \rightarrow MT''$ avec $MT'' <: MT$
Partielle non-isomorphique (notée $MT' \lesssim :_f MT$)	$extractEffectiveMT : \mathbf{MT} \times \mathbf{F} \rightarrow \mathbf{MT}$ $adapt_{MT} : MT' \rightarrow MT''$ avec $MT'' <: extractEffectiveMT(MT, f)$ et $MT <: extractEffectiveMT(MT, f)$

de sous-typage partielle.

Enfin, le fait de pouvoir toujours se ramener à la relation de sous-typage totale isomorphique permet de ne définir qu'un seul outil de vérification des relations de sous-typage, qu'il soit utilisé à la conception ou à l'exécution. L'implémentation d'un système de types orienté-modèle peut donc s'appuyer sur ce seul outil et être construit de manière modulaire afin de fournir des relations de sous-typage partielles et non-isomorphiques. Les facilités offertes par ce système de types seront donc dépendantes, non pas du cœur de la vérification des relations de sous-typage, mais des possibilités offertes pour la définition d'adaptations et pour l'extraction de type de modèles effectif.

La Table 7.2 présente comment les quatre relations de sous-typage entre types de modèles peuvent être ramenées à la relation totale isomorphique à l'aide des fonctions d'extraction de type de modèles effectif et d'adaptation définie plus tôt (cf. Définitions 7.8 et 7.10). Dans cette table, **MT** est l'ensemble de tous les types de modèles, **F** est l'ensemble des transformations de modèles, MT, MT' et MT'' sont des types de modèles et f est une transformation de modèles.

7.3.3 Une famille de systèmes de types orientés-modèle

Les relations de sous-typage, ainsi que la façon de les déclarer et de les vérifier, et les relations d'héritage définissent une famille de systèmes de types orientés-modèle. La Figure 7.14 présente un feature model de cette famille, où chaque rectangle présente une feature, i.e., une caractéristique d'un tel système de types.

Un système de types orienté-modèle peut être composé de différentes relations de sous-typage entre types de modèles et d'héritage entre métamodèles (dénotées par la relation optionnelle vers chacune des features). Les relations de sous-typage doivent prendre en compte la structure des types des modèles qu'elles lient et le contexte dans lequel elles sont utilisées. Une relation d'héritage peut impliquer différentes formes de sous-typage. De plus les relations de sous-typage doivent être déclarées et vérifiées.

Le feature model présenté en Figure 7.14 permet de définir 954 systèmes de types orientés-modèle différents [2]. Selon les caractéristiques sélectionnées (relations de sous-typage et d'héri-

2. Le nombre de configurations possibles issues du *feature model* de la Figure 7.14 a été calculé à l'aide de l'outil FAMILIAR (http://familiar-project.github.io/) [ACLF13]

tage, déclaration et vérification), le système de types résultant supportera différentes facilités connues au niveau des systèmes de types orientés-objet.

7.4 Conclusion

Ce chapitre présente deux types de relations pour la réutilisation entre langages de modélisation dédiés : les relations de sous-typage entre types de modèles et les relations d'héritage entre métamodèles. Ces relations peuvent être implémentées au sein d'environnements de développement de langages de modélisation dédiés supportant les types de modèles et les métamodèles tels que définis au chapitre précédent (cf. Chapitre 6).

Les relations de sous-typage autorisent un modèle typé par un sous-type à être utilisé là où un modèle typé par un super-type est attendu, autorisant ainsi le polymorphisme de sous-type pour les modèles. Les relations de sous-typage s'établissent à partir de la correspondance des différents éléments des types de modèles (types objets, champs de modèles et signatures d'opérations de modèles). À l'aide de ces relations de correspondance, de la présence ou non d'hétérogénéités structurelles et du contexte d'utilisation de la relation de sous-typage, nous définissons quatre relations de sous-typage : totale isomorphique, partielle isomorphique, totale non-isomorphique et partielle non-isomorphique.

Ces quatre relations peuvent être ramenées à la relation de sous-typage totale isomorphique à l'aide d'adaptations de modèles, qui permettent d'établir un isomorphisme de sous-graphe entre deux types de modèles malgré les hétérogénéités structurelles, et de l'extraction du type de modèles effectif, qui permet d'obtenir le sous-ensemble d'un type de modèles utilisé dans un contexte d'utilisation donné.

Les relations d'héritage permettent la réutilisation du code d'un super-métamodèle à un sous-métamodèle par l'extension et la redéfinition des éléments du super-métamodèle. Les relations d'héritage entre métamodèles peuvent impliquer une relation de sous-typage entre les types exacts des métamodèles. Selon la relation de sous-typage considérée, différentes redéfinitions des éléments du super-métamodèle sont possibles. À l'inverse, si l'héritage n'implique pas le sous-typage, les modèles instances du sous-métamodèle ne sont pas substituables aux modèles instances du super-métamodèle, mais les redéfinitions valides sont uniquement limitées par le métalangage.

Les relations entre langages de modélisation dédiés, et les moyens de les définir et de les vérifier, définissent une famille de systèmes de types orientés-modèle. Chacun de ces systèmes de types supporte différentes facilités de typage pour la définition et l'outillage de langages de modélisation dédiés. Dans la partie suivante, nous validons la définition de cette famille de systèmes de types au travers de l'implémentation et de l'application de l'un d'entre eux.

Le Chapitre 8 présente la validation de nos travaux au travers de l'implémentation d'un système de types orienté-modèle au sein de l'environnement de développement de langages de modélisation dédiés Kermeta. Ce système de types supporte les features suivantes de notre famille de systèmes de types :

- une relation de sous-typage totale isomorphique entre types de modèles ;
- la déclaration des relations de sous-typage à la définition des types, de manière explicite ;
- la vérification statique des relations de sous-typage ;
- une relation d'héritage entre métamodèles n'impliquant pas le sous-typage.

Le Chapitre 9 présente lui l'application du système de types implémenté dans Kermeta à un cas d'utilisation provenant de la compilation optimisante : le passage en forme SSA. Cette transformation courante dans les compilateurs optimisants est implémentée à l'aide de types

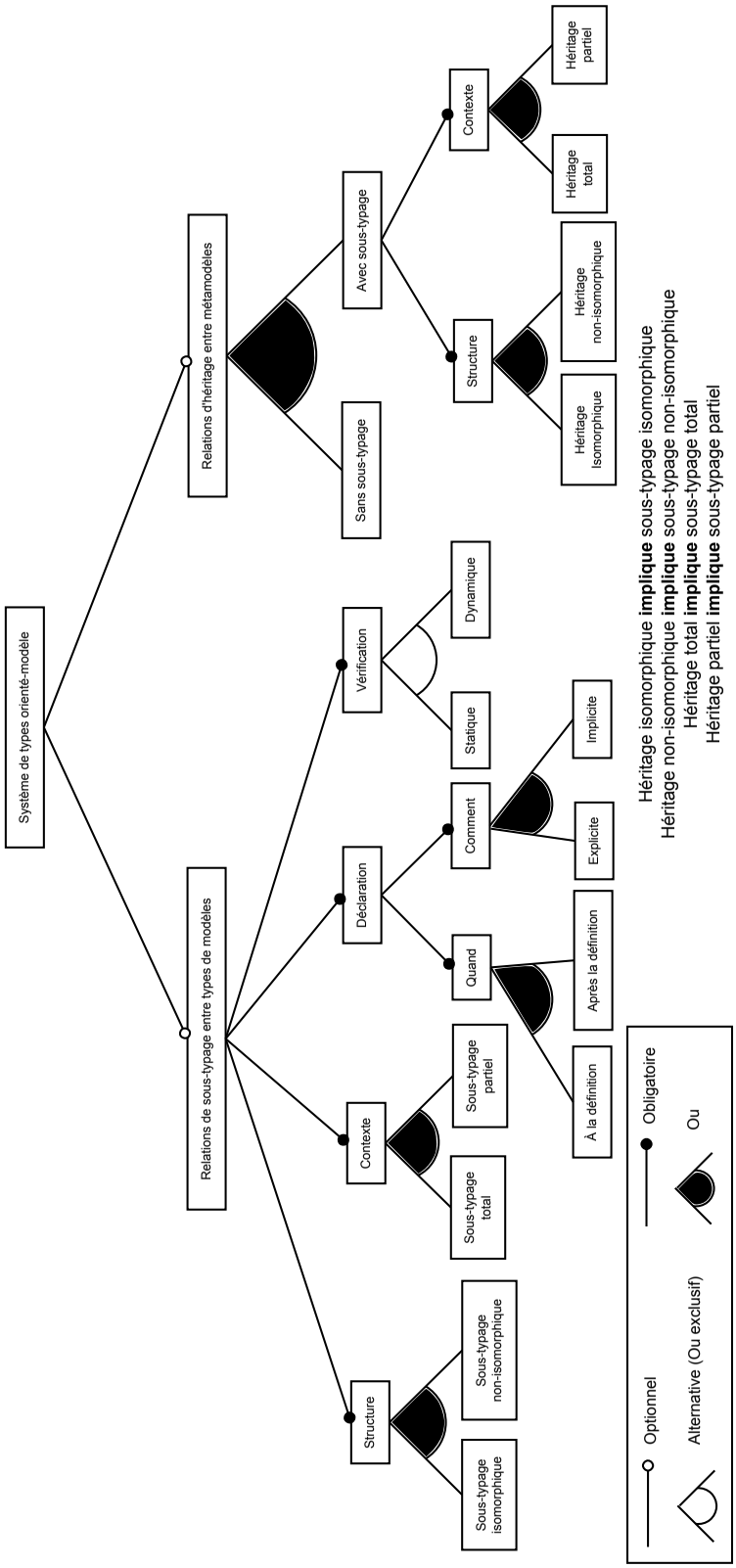

FIGURE 7.14 – *Feature model* de la famille de systèmes de types orientés-modèle.

de modèles et de transformations de modèles réutilisables grâce à la relation de sous-typage totale isomorphique.

Troisième partie

Implémentation et Validation

Chapitre 8

Implémentation d'un système de types orienté-modèle dans Kermeta

La relation de typage de modèles et les relations de sous-typage entre types de modèles définissent une famille de systèmes de types orientés-modèle. Ces systèmes de types fournissent un socle pour développer des facilités d'ingénierie pour la définition et l'outillage de langages de modélisation dédiés. Dans ce chapitre, nous présentons notre implémentation d'un tel système de types orienté-modèles au sein de Kermeta.

Kermeta [1] désigne à la fois un environnement de développement de langages de modélisation dédiés et un métalangage utilisé dans cet environnement. L'environnement Kermeta permet de séparer les différentes préoccupations au sein de la définition et de l'outillage d'un langage de modélisation dédié à l'aide de différents métalangages dédiés à ces préoccupations [JCB+13].

Le système de types que nous avons implémenté dans Kermeta supporte la relation de sous-typage totale isomorphique, à partir de laquelle il est possible de définir et d'implémenter nos trois autres relations de sous-typage entre types de modèles. Notre système de types fournit également une relation d'héritage entre métamodèles.

Ce chapitre commence par présenter Kermeta (cf. Section 8.1). Il présente ensuite notre implémentation au sein de Kermeta, qui inclut la modification de la syntaxe abstraite du métalangage Kermeta (cf. Section 8.3), l'ajout de nouvelles notations dans sa syntaxe concrète textuelle (cf. Section 8.4) et l'extension de sa sémantique par l'ajout d'un outil de vérification des relations de sous-typage entre types de modèles et la modification de la génération de code Scala (cf. Section 8.5.1 et 8.5.2).

8.1 Présentation de Kermeta

Kermeta est un environnement de développement de langages de modélisation dédiés. Kermeta fait appel à différents métalangages liés aux différentes préoccupations impliquées dans la conception de langages de modélisation dédiés : la syntaxe abstraite, la sémantique statique (i.e., les contraintes exprimées sur la structure et le comportement du langage), et la sémantique comportementale [JCB+13].

L'environnement Kermeta fournit également des opérateurs responsables de la composition de ces différentes préoccupations dans un moteur d'exécution (interpréteur ou compilateur)

1. http://www.kermeta.org/

du langage de modélisation dédié.

8.1.1 Définition d'un langage de modélisation dédié au sein de l'environnement Kermeta

Pour définir un langage de modélisation dédié au sein de l'environnement Kermeta, on commence par définir la syntaxe abstraite, qui précise les concepts du domaine et leurs relations. La syntaxe abstraite est exprimée d'une manière orientée-objet, en utilisant le métalangage Essential Meta-Object Facility (EMOF) [OMG06].

La sémantique statique d'un langage de modélisation dédié est l'union des règles de bonne formation (well-formedness en anglais) sur la syntaxe abstraite (e.g., des invariants de classes) et de la sémantique axiomatique (e.g., des pré et post-conditions sur les opérations). Kermeta utilise l'Object Constraint Language (OCL) [OMG03] pour exprimer la sémantique statique d'un langage, directement tissée dans la syntaxe abstraite à l'aide du mot-clé aspect de Kermeta.

EMOF ne comprend pas les concepts nécessaires à la définition de la sémantique du comportement et OCL est un langage sans effets de bord. Pour définir la sémantique comportementale d'un langage de modélisation dédié, l'environnement Kermeta s'appuie sur le langage d'action Kermeta [MFJ05]. La sémantique comportementale est-elle aussi tissée dans la syntaxe abstraite à l'aide du mot-clé aspect, sous la forme d'opérations.

Le langage Kermeta est impératif, typé statiquement et inclut des structures de contrôle classiques tels que les blocs, les conditionnelles, les boucles, et les exceptions. Le langage Kermeta met également en œuvre des mécanismes orientés-objet traditionnels pour traiter l'héritage multiple et les classes génériques. Le langage Kermeta fournit une sémantique d'exécution pour toutes les constructions de l'Essential Meta-Object Facility qui doivent avoir une sémantique au moment de l'exécution comme la composition et les champs opposés.

8.1.2 Composition des différentes préoccupations

Comme présentée ci-dessus, la composition des différentes préoccupations impliquées dans la définition d'un langage de modélisation dédié dans l'environnement Kermeta se fait à l'aide du mot-clé aspect. La composition s'appuie également sur un langage de construction de langages de modélisation dédiés : kp (pour Kermeta Project).

En Kermeta, tous les éléments de la sémantique statique (invariants, et pre et post-conditions) et comportementale (opérations) sont encapsulés dans les classes de la syntaxe abstraite. Le mot-clé aspect permet aux concepteurs de langages de modélisation dédiés de lier les différentes préoccupations (syntaxe abstraite, sémantique statique, et sémantique comportementale). Il permet de réouvrir une classe de la syntaxe abstraite pour y ajouter de nouveaux champs, opérations ou contraintes. Ce mécanisme est inspiré des mécanismes d'open-class [CLCM00].

Le langage kp permet la composition en elle-même. Il permet de définir un langage de modélisation dédié en déclarant les différentes spécifications issues des métalangages utilisés dans l'environnement Kermeta. kp permet également de définir des dépendances vers d'autres langages de modélisation dédiés ou vers des bibliothèques (e.g., la bibliothèque standard de Kermeta qui fournit des types primitifs et des collections). Un fichier kp définit donc les différents composants d'un langage de modélisation dédié, ainsi que la déclaration des dépendances de ce langage vers d'autres langages ou vers des bibliothèques.

8.1.3 Compilation de la composition

L'environnement Kermeta fournit également un compilateur qui produit un langage de modélisation dédié exécutable, à partir des entrées décrites au sein d'un fichier kp. Ces entrées

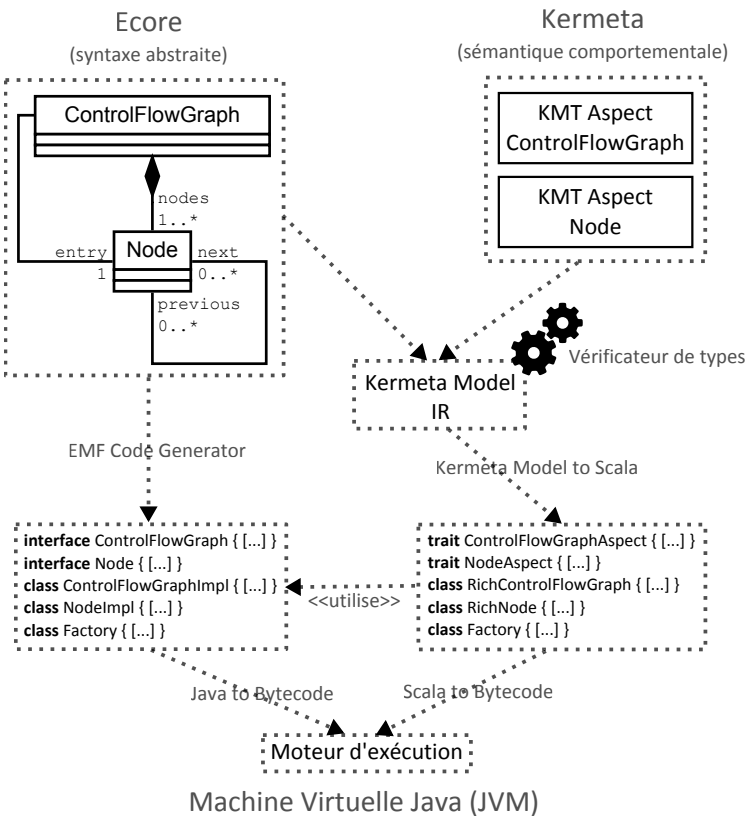

FIGURE 8.1 – Flot de compilation de l'environnement de développement de langages de modélisation dédiés Kermeta.

sont un ensemble de classes Ecore, la sémantique statique définie en OCL et la sémantique comportementale définie à l'aide du langage Kermeta (cf. Figure 8.1) [JCB+13]. Le Listing 8.3 présente un exemple d'un tel fichier `kp`. Les lignes 1 et 2 déclarent puis importent une dépendance pointant vers la bibliothèque standard de Kermeta. Les lignes 4 à 9 déclarent le nom du langage de modélisation dédié et les différents fichiers utilisés pour le définir : un fichier Ecore, `simple.ecore` présenté en Figure 8.2, et deux fichiers Kermeta `ControlFlowGraphAspect.kmt` et `NodeAspect.kmt`, présentés en Listings 8.1 et 8.2.

Le processus de compilation génère une représentation intermédiaire (cf. Kermeta Model IR dans la Figure 8.1) à partir de laquelle il produit quatre artefacts pour chaque classe A de la syntaxe abstraite du langage de modélisation dédié :

- une interface Java générée à partir du générateur Eclipse Modeling Framework (A_{Java})
- une classe Java implémentant l'interface, générée à partir du générateur Eclipse Modeling Framework ($AImpl_{Java}$)

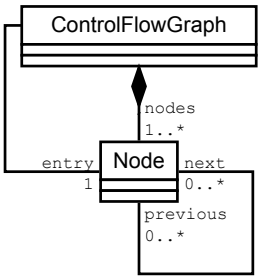

FIGURE 8.2 – Fichier `simple.ecore` déclarant la syntaxe abstraite d'un langage de graphes de flot de contrôles.

Listing 8.1 – Fichier `ControlFlowGraphAspect.kmt` déclarant un invariant OCL dans la classe `ControlFlowGraph`.

```
aspect class ControlFlowGraph {
  inv nodes_includes_entry :
    self.nodes->includes(self.entry)
}
```

Listing 8.2 – Fichier `NodeAspect.kmt` déclarant une variable et une opération dans la classe `Node`.

```
aspect class Node {
  var isReachable : Boolean
  operation computeIsReachable() : Boolean is do
    ...
  end
}
```

Listing 8.3 – Exemple de fichier *kp* pour la création d'un langage de modélisation dédié.

```
resource library_standard = "org.kermeta.language.library.standard"
importProject library_standard

metamodel CFG_Simple {
  import "${project.baseUri}/src/main/ecore/simple.ecore"
  import "${project.baseUri}/src/main/kmt/ControlFlowGraphAspect.kmt"
  import "${project.baseUri}/src/main/kmt/NodeAspect.kmt"
}
```

– un trait Scala [2] contenant la sémantique statique et le comportement de la classe ($AAspect_{Scala}$) ;
– une classe Scala composant le trait Scala et la classe Java générés ($RichA_{Scala}$).

Le compilateur Kermeta génère également une Factory sous la forme d'une classe Scala ($Factory_{Scala}$) étendant la Factory générée par le générateur Eclipse Modeling Framework sous la forme d'une classe Java ($Factory_{Java}$).

L'ensemble de ces classes et traits est ensuite compilé vers le bytecode Java à l'aide des compilateurs Java et Scala. Ce bytecode correspond au moteur d'exécution du langage de modélisation dédié, exécutable grâce à la machine virtuelle Java.

8.2 Présentation du système de types implémenté

C'est au sein de l'environnement de développement de langages de modélisation dédiés Kermeta que nous avons implémenté un système de types orienté-modèles. Ce système de types supporte un sous-ensemble des features du feature model présenté en Section 7.3.3. Il supporte une relation de sous-typage entre types de modèles et une relation d'héritage entre métamodèles (cf. Section 7.4). Il offre la possibilité de manipuler les modèles comme des entités de première classe grâce aux types de modèles et permet de définir des transformations de modèles réutilisables entre plusieurs langages de modélisation dédiés.

2. Un *trait* fournit un ensemble d'opérations définissant un comportement et requiert un ensemble d'opérations utilisées pour paramétrer ce comportement. Les traits peuvent être composés, entre eux et avec des classes. [DNS+06, OZ05]

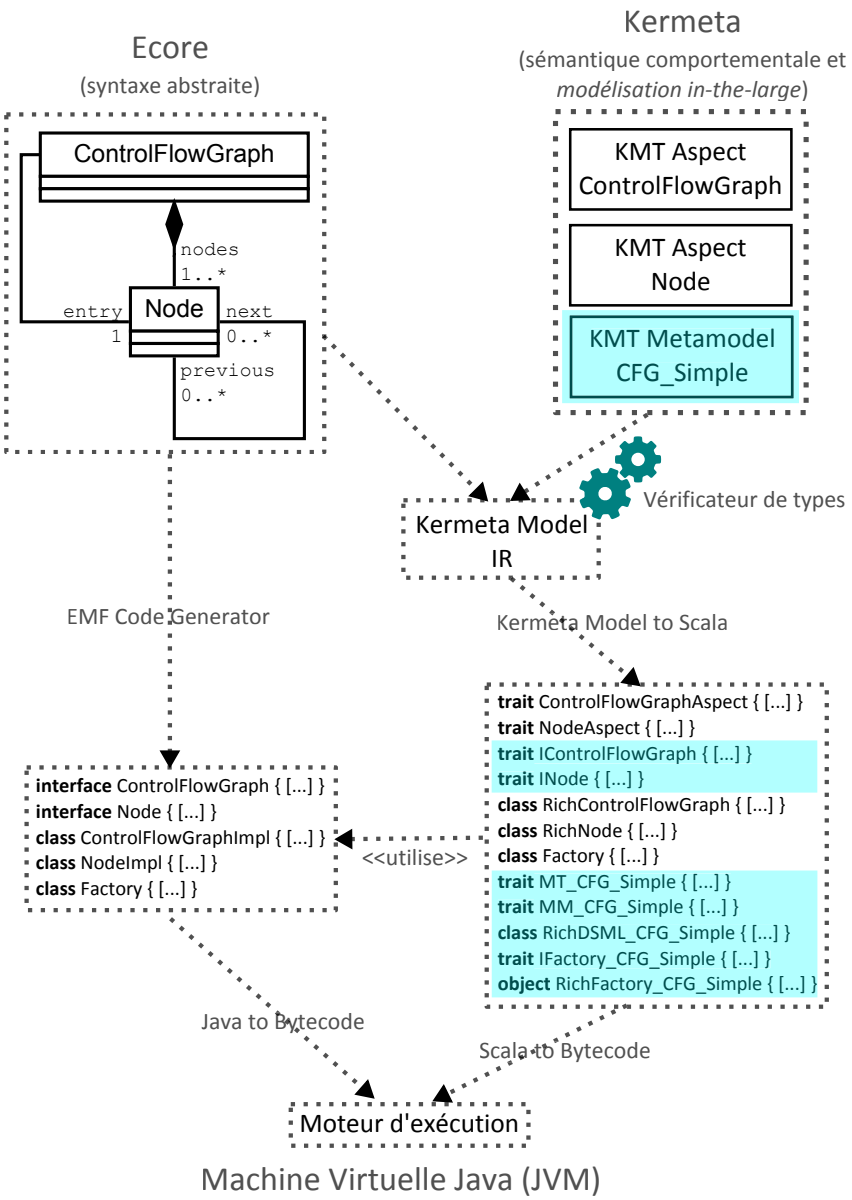

FIGURE 8.3 – Flot de compilation de Kermeta étendu pour intégrer le typage de modèles.

Le système de types orienté-modèle que nous avons implémenté au sein de l'environnement Kermeta supporte la relation de sous-typage totale isomorphique. Les trois autres relations de sous-typage entre types de modèles (partielle isomorphique, totale non-isomorphique et partielle non-isomorphique) peuvent être ramenées à la relation de sous-typage totale isomorphique à l'aide de fonction d'adaptation et d'extraction de types de modèles (cf. Section 7.3.2). Ainsi, l'outil de vérification des relations de sous-typage implémenté au sein de Kermeta peut être utilisé pour vérifier les quatre relations de sous-typage.

Cependant, seule la relation de sous-typage totale isomorphique est supportée par Kermeta. En effet, même si il est possible de réutiliser l'outil de vérification des relations de sous-typage, il n'existe pas actuellement de support pour les fonctions d'adaptation et d'extraction de types de modèles. Ces fonctions d'adaptation et d'extraction peuvent être ajoutées de manière modulaire au système de types implémenté dans Kermeta et ne nécessitent pas de connaissance de l'outil de vérification des relations de sous-typage, ni du générateur de code du compilateur Kermeta.

Les relations de sous-typage entre types de modèles sont déclarées dans notre extension de Kermeta explicitement (i.e., à l'aide du mot-clé `subtypeof`) et à la définition du sous-type, de manière équivalente aux relations de sous-typage entre classes. Nous avons choisi de vérifier statiquement les relations de sous-typage entre types de modèles, afin de fournir une détection des erreurs plus précoce et d'ouvrir la possibilité d'implémenter d'autres facilités de typage comme l'auto-complétion. Nous avons ajouté dans le compilateur Kermeta un outil de vérification travaillant sur la représentation intermédiaire dans ce but.

Notre système de types supporte également une relation d'héritage entre métamodèles. Cette relation d'héritage n'implique pas le sous-typage et est donc déclarée séparément des relations de sous-typage.

Le typage de modèles tel que présenté dans la Partie II ne prend pas en compte la sémantique statique des langages de modélisation dédiés. C'est pourquoi, dans la suite de ce chapitre, l'intégration des contraintes OCL dans le système de types implémenté au sein de Kermeta n'est pas abordé. La prise en compte de ces contraintes et de leur variance (cf. Section 4.2.1.1) est l'une des possibilités d'extension de nos travaux abordées dans les perspectives de cette thèse (cf. Section 10.2).

La Figure 8.1 présente le flot de compilation de Kermeta étendu pour le typage de modèles. Les modifications que nous avons apportées à l'environnement Kermeta incluent la modification de la syntaxe abstraite du langage pour intégrer les concepts de METAL (cf. Partie II), l'extension de la syntaxe concrète pour la manipulation de ces concepts et la modification de la composition et de la compilation des langages de modélisation dédiés pour prendre en compte la sémantique du typage de modèles.

8.3 Syntaxe abstraite

Pour intégrer les concepts de type de modèles et de métamodèle dans Kermeta, nous avons modifié la syntaxe abstraite du métalangage Kermeta. Nous y avons intégré les concepts et relations de METAL (cf. Partie II).

Cependant, afin de conserver la compatibilité avec les différents métalangages existants de l'environnement de développement, la syntaxe abstraite de Kermeta n'est pas strictement identique à celle présentée dans la Partie II. En effet Kermeta s'appuie sur Ecore pour la définition de la syntaxe abstraite des langages de modélisation dédiés et doit donc conserver certains des choix faits au sein d'Ecore (cf. Section 6.1.1).

La Figure 8.4 présente un extrait de la syntaxe abstraite de Kermeta étendue pour le typage de modèles. La figure ne représente pas l'ensemble des expressions Kermeta permettant de

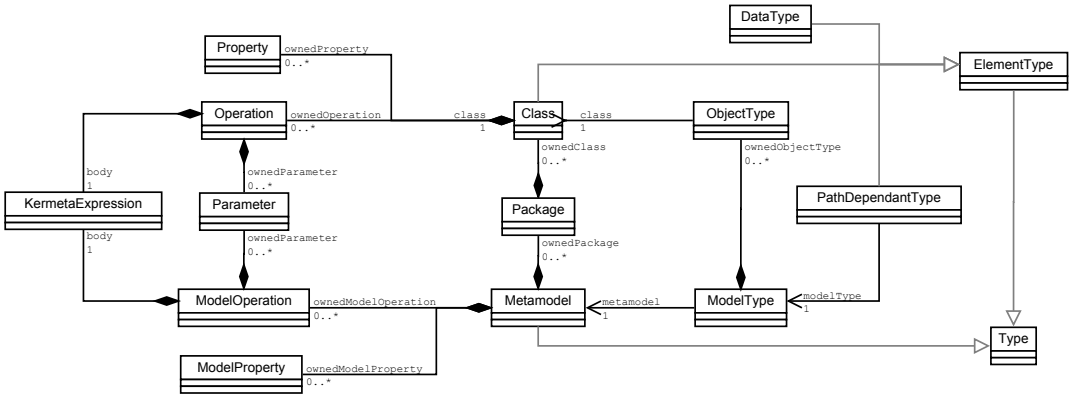

FIGURE 8.4 – Extrait de la syntaxe abstraite du langage Kermeta.

définir le corps d'une opération. De même, certaines classes ont été omises par souci de simplicité : les classes `NamedElement`, `TypedElement` et `MultiplicityElement` de Kermeta sont équivalentes à celles de METAL et n'apparaissent pas en Figure 8.4.

Ecore ne séparant pas les concepts de classe et de type, notre implémentation ne les sépare que de manière interne, en extrayant les types exacts des classes. L'utilisateur utilise les classes, et les métamodèles, indifféremment pour instancier des objets ou des modèles et pour typer des variables, champs, paramètres et opérations. C'est pourquoi `Class` et `Metamodel` héritent de `Type`. C'est aussi la raison pour laquelle il n'y a pas de classe représentant les signatures d'opérations. Les types objets et les types de modèles sont extraits d'une classe ou d'un métamodèle et accèdent à leurs éléments pour déterminer leur interface (à travers les associations `class` et `metamodel`).

De plus, Ecore possède la notion de package (`Package`). Un package contient des classes et d'autres packages. Les classes de Kermeta sont également contenues par des packages, qui apparaissent dans leur nom qualifié (qualified name en anglais). Une classe, qu'elle soit utilisée pour instancier un objet ou pour typer une variable est donc déclarée en utilisant la forme `package::classe`.

Les classes représentant les champs et les opérations Ecore (`Property` et `Operation`) possèdent un champ référençant leur conteneur (`class`). Ce conteneur est forcément une classe (`Class`) et ne peut pas être un métamodèle, puisqu'Ecore ne réifie pas les métamodèles. Afin de ne pas modifier les classes existantes représentant les champs et opérations d'objets, nous avons ajouté des classes spécifiques représentant les champs et opérations de modèles (`ModelProperty` et `ModelOperation`).

8.4 Syntaxe concrète

Les métalangages utilisés au sein de l'environnement Kermeta permettent de déclarer et manipuler les concepts de modélisation in-the-small. Afin de permettre à l'utilisateur de manipuler les concepts de modélisation in-the-large introduits dans la syntaxe abstraite de Kermeta, nous avons introduit de nouvelles notations dans sa syntaxe concrète. Nous fournissons ainsi le moyen de déclarer les propriétés et les opérations de modèles au sein d'un métamodèle. Le corps d'une transformation de modèles peut contenir des types chemins-dépendants pour lesquels nous avons également fourni une syntaxe concrète. Enfin, nous avons défini des mot-clés permettant de déclarer les relations de sous-typage et d'héritage.

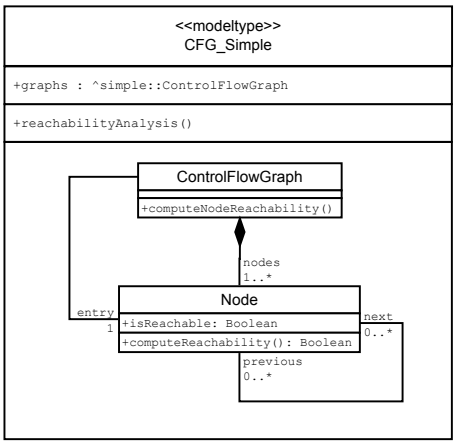

FIGURE 8.5 – Type de modèles `CFG_Simple`.

Listing 8.4 – Exemple de déclaration de métamodèle à l'aide du langage *kp*.

```
1  metamodel CFG_Simple {
2    import "${project.baseUri}/src/main/ecore/
3      simple.ecore"
4    import "${project.baseUri}/src/main/kmt/
5      NodeReachabilityAnalysis.kmt"
6    import "${project.baseUri}/src/main/kmt/
7      GraphAspect.kmt"
8    import "${project.baseUri}/src/main/kmt/
9      NodeAspect.kmt"
10 }
```

Listing 8.5 – Exemple de variable contenant un modèle en Kermeta.

```
1  var model : CFG_Simple init CFG_Simple.new
```

8.4.1 Métamodèles et types de modèles

Par défaut, Kermeta crée un métamodèle quand il compose différents fichiers déclarés à l'aide du métalangage kp. Selon les différents fichiers (et les métalangages utilisés dans ces fichiers), ce métamodèle est composé de classes, de champs et de signatures d'opérations (déclarés à l'aide d'Ecore), d'opérations (déclarées à l'aide du métalangage d'action Kermeta) et de champs et d'opérations de modèles (également déclarés avec le métalangage Kermeta).

Un métamodèle est donc déclaré au sein du fichier kp et prend la forme présentée en Listing 8.4. Le Listing 8.4 présente un métamodèle nommé `CFG_Simple` qui implémente le type de modèles présenté en Figure 8.5. Le mot-clé `metamodel` suivi de l'identifiant `CFG_Simple` déclare un métamodèle. Les `import` déclarent les chemins vers les fichiers utilisés pour construire ce métamodèle :

- un fichier Ecore (`simple.ecore`) qui déclare le graphe de classes du métamodèle ;
- un fichier Kermeta (`NodeReachabilityAnalysis.kmt`) déclarant les champs et opérations de modèles nécessaires à l'analyse d'atteignabilité des blocs d'un graphe de flot de contrôle ;
- deux fichiers Kermeta (`GraphAspect.kmt` et `NodeAspect.kmt`) qui déclarent les champs et opérations d'objets nécessaires à l'analyse d'atteignabilité dans les classes `ControlFlowGraph` et `Node`.

Puisque les types de modèles ne sont pas séparés des métamodèles d'un point de vue utilisateur, le nom d'un métamodèle est utilisé pour typer des variables, des paramètres ou des opérations. Le Listing 8.5 montre un exemple de l'utilisation du métamodèle `CFG_Simple` pour typer une variable.

Le nom du métamodèle peut également être utilisé pour instancier un modèle à l'aide de l'opération spécifique `new`. En Kermeta, `new` est une opération disponible sur les classes et les métamodèles qui permet de produire une instance de ces classes ou métamodèles. Dans le Listing 8.5, l'expression `CFG_Simple.new` utilisée pour initialiser la variable `model` produit un nouveau modèle instance de `CFG_Simple`, i.e., réserve l'espace mémoire pour le modèle, contenant le champ de modèles graphs et un pointeur vers la table des opérations de modèles définies par le métamodèle `CFG_Simple`.

Listing 8.6 – Exemple de types chemin-dépendants en Kermeta.

```
attribute rootNode : ^simple::Node

operation convertToDot[MT ^: Dot]() : MT^dot::DotGraph is do
  ...
end
```

8.4.2 Types *chemins-dépendants*

Les types chemins dépendants sont utilisés pour déclarer des variables, des opérations ou des paramètres dont le type objet varie en fonction d'un type de modèles (cf. Section 4.3). Nous différencions deux sortes de types chemin-dépendants : les types qui appartiennent au même type de modèles que le terme qu'ils typent (variable, opération ou paramètre) ; et les types qui appartiennent à un autre type de modèles.

Le Listing 8.6 présente un exemple de chacun de ces types chemin-dépendants, supposant que l'attribut `rootNode` et l'opération `convertToDot` sont contenus par le métamodèle `CFG_Simple` :

- le champ `rootNode` déclare un type `^simple::Node` ;
- l'opération `convertToDot` déclare un type de retour `MT^dot::DotGraph`.

Ici le caractère `^` est utilisé comme séparateur entre un type de modèles et un package dans le nom qualifié d'un type objet et les caractères `::` sont utilisés comme séparateur entre packages et entre un package et un type. Si aucun type de modèles n'est indiqué avant le `^`, le type objet appartient au même type de modèles que le terme typé. Ainsi, `^simple::Node` indique le type `Node`, du package `simple` appartenant au même type de modèles que le champ `rootNode`, i.e., le type de modèles extrait du métamodèle `CFG_Simple` (`^simple::Node` indique donc ici le type `CFG_Simple^simple::Node`). Si le champ `rootNode` est hérité dans un métamodèle `CFG_Dominance`, son type changera avec le type de modèles pour indiquer le type `CFG_Dominance^simple::Node`.

Il est également possible d'utiliser un paramètre de type (cf. Section 4.2.4) pour déclarer un type chemin-dépendant. L'opération `convertToDot` possède un paramètre de type nommé `MT` dont la borne supérieure est le type de modèles `Dot` (en Kermeta, la borne supérieure d'un paramètre de type est indiquée par `:` pour les types objets et par `^:` pour les types de modèles). Le paramètre de type `MT` est ensuite utilisé pour définir le type de retour de l'opération (`MT^dot::DotGraph`). Selon le type de modèles passé en paramètre à l'opération lors de son appel, `MT^dot::DotGraph` désignera donc un type objet différent.

8.4.3 Opérations et champs de modèles

L'ajout de champs et d'opérations de modèles se fait dans un métamodèle déclaré à l'aide d'un fichier kp. Les champs et opérations de modèles sont donc déclarés indépendamment dans un fichier Kermeta, comme celui présenté en Listing 8.7. Le mot-clé `metamodel`, suivi du nom du métamodèle, est utilisé pour déclarer un ensemble de champs et d'opérations de modèles.

Les champs et opérations de modèles sont déclarés d'une manière équivalente aux champs et opérations : à l'aide des mot-clés `attribute` (pour les champs composite), `reference` (pour les champs non-composite) et `operation` pour les opérations. Les champs et opérations de modèles sont nommés et typés : le champ de modèles `graphs` est un ensemble d'objets typés par `ControlFlowGraph`, l'opération de modèles `reachabilityAnalysis` ne prend aucun paramètre

Listing 8.7 – Exemple de déclaration de champ et d'opération de modèles en Kermeta.

```
metamodel CFG_Simple {
  attribute graphs : set ^simple::ControlFlowGraph[0..*]

  operation reachabilityAnalysis() : Void is do
    graphs.each{g|
      g.computeNodeReachability()
    }
  end
}
```

Listing 8.8 – Exemple d'appel de champ et d'opération de modèles en Kermeta.

```
var model : CFG_Simple init CFG_Simple.new

model.reachabilityAnalysis()

var modelGraphs : Set[CFG_Simple^simple::ControlFlowGraph] init model.graphs()
```

et ne retourne rien. Le corps d'une opération de modèles, comme le corps d'une opération d'objets est déclaré entre les délimiteurs `do` et `end` à l'aide du métalangage d'action Kermeta (i.e., le corps d'une opération de modèles est une séquence d'expressions Kermeta).

Un champ de modèles ou une opération de modèles peut ensuite être appelé à partir d'un modèle, à l'aide de la notation pointée classique des langages orientés-objet (cf Listing 8.8).

8.4.4 Relation de sous-typage et d'héritage

La relation d'héritage est implémentée dans Kermeta comme une facilité de construction des métamodèles. C'est pourquoi, la relation d'héritage est déclarée au moment de la déclaration d'un métamodèle dans un fichier kp, là où sont déclarés les différents imports nécessaires à la construction du métamodèle. La déclaration de la relation d'héritage se fait à l'aide du mot-clé `extends` suivi du nom du métamodèle à étendre (cf. Listing 8.9).

Une relation de sous-typage se déclare à l'aide du mot-clé `subtypeof` suivi du nom du super-type (cf. Listing 8.10). Puisque la syntaxe concrète ne sépare pas les métamodèles des types de modèles, les noms utilisés sont ceux des métamodèles, bien que la relation de sous-typage soit définie entre les types de modèles qui sont extraits de ces métamodèles. Cette déclaration se fait dans les fichiers Kermeta dans lesquels sont déclarés les champs et les opérations de modèles.

La déclaration des deux relations est séparée afin de laisser la possibilité de fournir des facilités de redéfinition découplées du sous-typage et d'autoriser la déclaration de sous-typage après la définition de deux métamodèles (cf. Section 4.2.3).

Listing 8.9 – Exemple de déclaration d'héritage entre CFG_Simple et CFG_Dominance en *kp*

```
metamodel CFG_Dominance extends CFG_Simple {
  ...
}
```

Listing 8.10 – Exemple de déclaration d'une relation de sous-typage entre CFG_Simple et CFG_Dominance en Kermeta.

```
metamodel CFG_Dominance {
  subtypeof CFG_Simple
}
```

8.5 Sémantique

Les modifications apportées au compilateur Kermeta pour prendre en compte la sémantique des types de modèles sont abordées en deux parties : la première (Section 8.5.1) s'intéresse aux modifications apportées au front-end (composition et vérification d'un métamodèle au sein de la représentation intermédiaire) ; la deuxième (Section 8.5.2) présente la modification du back-end (le générateur de code Scala).

8.5.1 Modifications apportées au *front-end* du compilateur Kermeta

Le compilateur Kermeta cible le bytecode Java, vers lequel sont compilés tous les artefacts impliqués dans la définition d'un langage de modélisation dédié. Le mapping entre la syntaxe abstraite de Kermeta et le bytecode Java est implémenté au travers du compilateur Kermeta. Le compilateur Kermeta est responsable du chargement et de la composition des différents fichiers spécifiant le métamodèle compilé, de la vérification de la validité de ce métamodèle (vérification de types, vérification des contraintes OCL) et de la génération du code Scala.

Nous avons modifié le chargement et la composition des fichiers afin de prendre en compte l'héritage entre métamodèles, ainsi que la vérification de la validité des métamodèles afin de vérifier les relations de sous-typage entre types de modèles. Ces deux étapes ont lieu dans le front-end du compilateur Kermeta et sont présentées dans cette section, tandis que la génération de bytecode Java, réalisée par le back-end du compilateur est présentée dans la section suivante.

8.5.1.1 Héritage entre métamodèles

L'héritage est traité par le compilateur Kermeta avant la phase de composition des différentes parties de la spécification du langage de modélisation dédié. Les éléments du super-métamodèle deviennent ainsi des entrées de l'outil de composition qui génère la représentation intermédiaire du sous-métamodèle.

Ces éléments venant du super-métamodèle sont donc mêlés aux éléments définis dans les différentes parties de la spécification du sous-métamodèle. Ainsi, c'est l'outil de composition qui prend en charge la composition des éléments déclarés deux fois (e.g., une même classe déclarée dans le super et le sous-métamodèle). Cette approche permet la redéfinition des classes du super-métamodèle par l'ajout de champs ou d'opérations, de la même manière que le langage Kermeta permet habituellement l'ajout de champs et d'opérations dans les classes Ecore.

Puisque la relation d'héritage entre métamodèles que nous proposons n'implique pas le sous-typage, et que l'ensemble du traitement est fait avant la génération du code Scala, aucune modification n'a été apportée à la génération de code de Kermeta pour supporter la relation d'héritage.

8.5.1.2 Vérification du sous-typage entre types de modèles

Nous avons implémenté un outil de vérification des relations de sous-typage totales isomorphiques qui s'appuie sur la Définition 7.11. Bien que nous n'ayons pas implémenté d'autres relations de sous-typage dans Kermeta, cet outil de vérification peut être utilisé pour vérifier des relations partielles ou non-isomorphiques. En effet, ces relations de sous-typage peuvent être ramenées à la relation de sous-typage totale isomorphique grâce aux fonctions d'adaptation (cf. Définition 7.10) et d'extraction de type de modèles effectif (cf. Définition 7.8).

Notre implémentation suit la Définition 7.11. Pour vérifier une relation de sous-typage déclarée entre le type de modèles *MT'* et le type de modèles *MT*, l'outil de vérification cherche

dans *MT'* un type objet correspondant à chaque type objet de *MT* (cf. Définition 7.1), un champ de modèles correspondant à chaque champ de modèles déclaré par *MT* (cf. Définition 7.3), et une signature d'opération de modèles pour chaque signature d'opération de modèles déclarée par *MT* (cf. Définition 7.2).

Puisque Kermeta ne sépare pas la déclaration des classes de la déclaration des types objets, chaque type objet contient un champ déterminant si la classe dont il est extrait est abstraite ou non. Nous utilisons cette information pour vérifier la substituabilité de classes lors de l'instanciation d'objets par une transformation de modèles (cf. Définition 7.4).

Pendant la vérification, il peut arriver qu'une relation de correspondance entre deux types objets (respectivement deux champs de modèles ou deux signatures d'opérations de modèles) dépende d'une autre relation de correspondance en cours de vérification. Ces dépendances sont stockées au fur et à mesure de la vérification. Dès qu'une relation de correspondance est résolue, le résultat est transmis à toutes les relations qui dépendent d'elle : si la relation de correspondance résolue est invalide, elle invalide toutes les relations qui en dépendent ; si la relation de correspondance résolue est valide, la dépendance est supprimée et la vérification des relations qui en dépendaient continue. Si une dépendance cyclique est créée, l'outil de vérification attend la fin de la vérification pour la résoudre. Si une des dépendances est invalidée avant la fin, le cycle est brisé. Sinon, les relations de correspondance inter-dépendantes sont toutes valides et sont acceptées par l'outil.

8.5.2 Modifications apportées au *back-end* du compilateur Kermeta : Sémantique translationnelle en Scala

Afin de prendre en compte notre extension du métalangage Kermeta, nous avons modifié le compilateur Kermeta afin qu'il génère un nouveau trait Scala par classe du métamodèle (IA_{Scala}) : ce trait représente le type objet des instances de la classe. Le compilateur génère également cinq autres artefacts Scala à partir des fichiers décrivant les champs et opérations de modèles d'un métamodèle :

- un trait pour le métamodèle (MM_{Scala}) ;
- un trait pour le type de modèles extrait du métamodèle (MT_{Scala}) ;
- une classe composant les deux traits précédents ($RichDSML_{Scala}$) ;
- un trait exposant l'interface d'une Factory créant des objets liés à un type de modèles ($IFactory_{Scala}$) ;
- un objet (i.e., singleton) implémentant la Factory ($Factory_{Scala}$).

Scala ne propose pas de mécanismes de correspondance entre types objets, ni de mécanismes de types de modèles ou de groupes de types comme ceux de Kermeta. Afin de générer du code Scala bien typé (i.e., valide pour le compilateur Scala), nous utilisons des variables de types pour représenter les groupes de types et des traits exposant l'interface que doivent respecter tous les types qui correspondent à un type objet. Afin d'assurer la substitution de modèles, i.e., de groupes d'objets, les types objets qui correspondent à un type *T* en Kermeta sont générés comme des sous-types de *T* en Scala. Ceci assure que les objets typés par un type *T'* correspondant à *T* sont substituables aux objets typés par *T*, puisque *T'* est un sous-type de *T* au niveau du Scala généré.

Cette section commence par présenter comment nous avons utilisé les variables de types de Scala pour représenter les relations qui unissent les types au sein de groupes de types. Elle suit ensuite une structure proche des Chapitres 6 et 7 et présente d'abord la compilation des concepts de Kermeta dédiés à la modélisation in-the-small : types objets, types chemin-dépendants et classes ; puis la compilation des concepts dédiés à la modélisation in-the-large :

types de modèles et métamodèles ; et enfin la compilation des relations de sous-typage entre types de modèles.

8.5.2.1 Groupes de types : variables de types

Pour représenter les groupes de types, i.e., les ensembles de types objets liés, en Scala, nous utilisons les variables de types (cf. Section 4.2.4). Les variables de types se comportent comme des champs, à l'exception qu'elles ne pointent pas un objet ou une valeur mais un type. Les variables de types peuvent être utilisées pour typer une opération ou un champ. Si une variable de type n'est pas initialisée dans la classe qui la déclare, elle peut être fixée par les classes qui en héritent. Ainsi, il est possible de déclarer les signatures d'opérations avec une variable de type non-initialisée et de fournir plusieurs implémentations de cette opération dans plusieurs classes, chacune retournant un type réel différent.

Chaque type de modèles possède une variable de type par type objet qu'il contient. Par exemple, le Listing 8.11 présente l'ensemble des variables de types du type de modèles `CFG_Simple` présenté en Figure 8.5, i.e., une variable de type pour le type objet `ControlFlowGraph` et une pour le type objet `Node`.

Chaque métamodèle qui implémente un type de modèles fixe ensuite les variables de types avec un type réel (ici un trait Scala). Par exemple, le Listing 8.12 présente l'ensemble des variables de types du métamodèle `CFG_Simple`. La valeur de ces variables de types est fixée à l'aide du trait $Aspect_{Scala}$ généré par le compilateur Kermeta. Ainsi chaque métamodèle fixe ses variables de types en fonction des classes qui le composent.

Listing 8.11 – Exemple de variables de types du type de modèles `CFG_Simple` en Scala.

```
type ControlFlowGraph <: IControlFlowGraph
type Node <: INode
```

Listing 8.12 – Exemple de variables de types d'un métamodèle en Scala.

```
type ControlFlowGraph = ControlFlowGraphAspect
type Node = NodeAspect
```

Chaque type objet déclare également le même ensemble de variables de types que son type de modèles et chaque classe fixe ces variables de la même manière que son métamodèle. Les variables de types générées permettent donc d'émuler les groupes de types en fournissant des ensemble de variables qui varient ensemble et assurent donc de ne pas mélanger des objets typés par des types de deux groupes (i.e., de deux types de modèles) différents.

8.5.2.2 Compilation des concepts de modélisation *in-the-small*

Chaque type objet A est compilé vers un trait Scala IA_{Scala} qui expose l'interface de tous les objets dont les types correspondent à T (cf. Section 7.1.1). Ce trait Scala déclare les mêmes variables de type que son type de modèles. Les champs et opérations du trait, qui exposent l'interface des objets typés par le type objet, sont typés à l'aide de ces variables de type.

Par exemple, le Listing 8.13 présente le trait `IControlFlowGraph` généré pour le type objet `ControlFlowGraph`. Ce trait déclare deux variables de types, correspondant aux deux types objets du type de modèles `CFG_Simple`. Chacune de ces variables de type est bornée par le trait généré pour le type objet. Ainsi, nous garantissons que tous les objets typés par ces variables respecteront l'interface définie par les traits générés. Ceci nous permet d'émuler la correspondance entre types objets : deux objets typés par une même variable de type peuvent appartenir à deux modèles typés par différents types de modèles, mais ils respectent tous les deux une interface commune, qui permet de les substituer l'un à l'autre.

Listing 8.13 – Exemple de trait Scala généré pour le type objet `ControlFlowGraph`.

```
trait IControlFlowGraph {
  //Variables de types pour l'ensemble des types objets du type de modèles auquel le type
      objet appartient
  type ControlFlowGraph <: IControlFlowGraph
  type Node <: INode
  //Opérations typées à l'aide des variables de types
  def computeNodeReachability()
  //Accesseurs aux champs typés à l'aide des variables de types
  def getNodes() : Set[Node]
  def setNodes(newNodes : Set[Node])
  def getEntry() : Node
  def setEntry(newEntry : Node)
}
```

Le trait `IControlFlowGraph` déclare également une opération (`computeNodeReachability`) et les accesseurs à deux champs (`nodes` et `entry`). Les opérations et accesseurs sont typés par les variables de types du trait.

Nous avons défini deux sortes de types *chemin-dépendants* (cf. Section 8.4.2,) compilés de manière différente, selon que ces types appartiennent au même type de modèles que l'opération ou le champ qu'ils typent ou non.

Un type chemin-dépendant utilisé au sein du type de modèles auquel il appartient (déclaré sous la forme `^pck::Type`) est compilé vers la variable de type correspondante du type de modèles. Par exemple, un type `^simple::Node` utilisé dans l'opération de modèles `reachabilityAnalysis` de `CFG_Simple` est compilé en Scala vers la variable de type `Node`. De cette manière, le type chemin-dépendant varie avec son type de modèles.

Les types chemins dépendants utilisés en dehors du type de modèles auquel ils appartiennent sont les types chemin-dépendants déclarés à l'aide des paramètres de type des opérations de modèles (sous la forme `MT^pck::Type`, où `MT` est un paramètre de type). Ces types chemin-dépendants sont compilés vers le trait généré pour le type objet correspondant de la borne du paramètre. Par exemple, le type `MT^simple::Node` où `MT` est borné par `CFG_Simple` est compilé vers le trait Scala `INode` généré pour le type `CFG_Simple^simple::Node`. En effet ce trait expose l'interface commune à tous les types objets correspondant à `CFG_Simple^simple::Node` et donc à tous les types réels vers lesquels peut être résolu le type chemin-dépendant `MT^simple::Node`.

Chaque classe A est compilée vers un trait Scala $AAspect_{Scala}$ et une classe $RichA_{Scala}$. $AAspect_{Scala}$ fournit le corps des opérations déclarées dans le Ecore et étend le trait IA_{Scala}. Par exemple, le trait `ControlFlowGraphAspect`, présenté en Listing 8.14, étend `IControlFlowGraph`. `ControlFlowGraphAspect` fournit l'implémentation des opérations dont le corps a été décrit à l'aide du langage d'action de Kermeta (e.g., `computeNodeReachability`).

De plus, `ControlFlowGraphAspect` doit fixer les variables de types et implémenter les opérations déclarées par `IControlFlowGraph`. Chaque classe d'un même métamodèle possédant les mêmes variables de type, ceci permet de garantir la spécialisation de groupes de classes. Un objet du trait `ControlFlowGraphAspect`, présenté en Listing 8.14, ne peut accepter et retourner que des objets du trait `NodeAspect` avec les accesseurs `getNodes`, `setEntry` et `getEntry`. Le groupe de types objets Kermeta composé de `ControlFlowGraph` et `Node` est donc bien conservé une fois compilé en Scala.

Enfin, pour chaque classe Kermeta, une classe Scala est générée afin de pouvoir instancier les éléments de modèles. La classe `RichControlFlowGraph`, présentée en Listing 8.15 étend la classe

Listing 8.14 – Exemple de trait Scala généré pour la classe `ControlFlowGraph`.

```
trait ControlFlowGraphAspect extends IControlFlowGraph {
  /**
   * Code original généré par Kermeta
   */
  //Opérations
  def computeNodeReachability() = {
    ...
  }
  //Les champs nodes et entry sont déclarés dans la classe Java ControlFlowGraphImpl générée par EMF

  /**
   * Code ajouté par l'extension de Kermeta pour les types de modèles
   */
  //Variables de types pour l'ensemble des types objet du type de modèles exact extrait du métamodèle auquel la classe appartient
  type ControlFlowGraph = ControlFlowGraphAspect
  type Node = NodeAspect
  //Accesseurs aux champs typés à l'aide des variables de types
  def getNodes() : Set[Node] = {
    return nodes
  }
  def setNodes(newNodes : Set[Node]) = {
    nodes = newNodes
  }
  def getEntry() : Node = {
    return entry
  }
  def setEntry(newEntry : Node) = {
    entry = newEntry
  }
}
```

Listing 8.15 – Exemple de classe Scala générée pour la classe `ControlFlowGraph`.

```
class RichControlFlowGraph extends ControlFlowGraphImpl with ControlFlowGraphAspect
```

Java `ControlFlowGraphImpl` générée par EMF et le trait `ControlFlowGraphAspect`. Ainsi, un objet instance de `RichControlFlowGraph` est un objet EMF, compatible avec l'ensemble des outils développés au-dessus d'EMF et fournit l'interface d'un élément de modèle substituable aux éléments d'un modèle d'un autre type.

8.5.2.3 Compilation des concepts de modélisation *in-the-large*

Un type de modèles est compilé vers un trait Scala MT_{Scala}**.** Ce trait expose l'interface des modèles qu'il type, sous la forme de signatures d'opérations et d'accesseurs pour les champs. Ainsi les signatures d'opérations de modèles sont représentées par des signatures d'opérations et les champs de modèles par une paire d'accesseurs. Par exemple, le type de modèles `CFG_Simple` présenté en Figure 8.5 est compilé vers le trait Scala `MT_CFG_Simple` présenté en Listing 8.16.

Les opérations et les accesseurs d'un type de modèles sont typés à l'aide des variables de types du type de modèles. Ainsi, l'interface exposée par un type de modèles reste valide pour tous les modèles typés par le type de modèles quelque soit le métamodèle dont ils sont instances.

De plus, pour chaque type de modèles, l'interface d'une Factory est générée. Cette interface définit les opérations de création nécessaires pour créer des objets typés par les types objets

Listing 8.16 – Exemple de trait Scala généré pour le type de modèles `CFG_Simple`.

```
trait MT_CFG_Simple {
  //Variables de types pour l'ensemble des types objets du type de modèles
  type ControlFlowGraph <: IControlFlowGraph
  type Node <: INode
  //Opérations de modèles typées à l'aide des variables de types
  def reachabilityAnalysis()
  //Accesseurs aux champs de modèles typés à l'aide des variables de types
  def getGraphs() : Set[ControlFlowGraph]
  def setGraphs(newGraphs : Set[ControlFlowGraph])
  //Accesseur à la Factory
  def getFactory() : IFactory_CFG_Simple
}
```

Listing 8.17 – Exemple de trait Scala généré pour la *Factory* du type de modèles `CFG_Simple`.

```
trait IFactory_CFG_Simple {
  //Variables de types pour l'ensemble des types objets du type de modèles
  type ControlFlowGraph <: IControlFlowGraph
  type Node <: INode
  //Opérations de créations d'objet typées à l'aide des variables de types
  def createControlFlowGraph() : ControlFlowGraph
  def createNode() : Node
}
```

du type de modèles ou des types objets correspondants. Le Listing 8.17 présente le trait `IFactory_CFG_Simple` généré pour le type de modèles `CFG_Simple`. Chaque type de modèles définit la signature d'un accesseur à sa Factory (`getFactory`), qui doit être fourni par les métamodèles qui l'implémentent.

Chaque métamodèle est compilé vers un trait Scala MM_{Scala} et une classe $RichDSML_{Scala}$. MM_{Scala} fournit le corps des opérations de modèles déclarées par les types de modèles qu'il implémente. Il fixe également les variables de types des types de modèles qu'il implémente. Par exemple, le trait `MM_CFG_Simple`, présenté en Listing 8.18, fournit le corps des opérations de modèles et fixe les variables de types du type de modèles qu'il implémente : `MT_CFG_Simple`.

Pour chaque métamodèle, une Factory est générée sous la forme d'un trait fournissant l'implémentation des méthodes de création et d'un objet (i.e., singleton) étendant ce trait. Par exemple, le Listing 8.19 présente le trait `Factory_CFG_Simple` et l'objet `RichFactory_CFG_Simple`. `Factory_CFG_Simple` fixe les variables de type et implémente les opérations de création d'objets de `CFG_Simple`. `RichFactory_CFG_Simple` est un singleton qui étend `Factory_CFG_Simple` et qui est appelé pour créer des objets par les opérations de modèles.

De plus, pour chaque métamodèle, une classe Scala est générée afin de pouvoir instancier les modèles. La classe `RichDSML_CFG_Simple`, présentée en Listing 8.20 est instanciable et permet donc de créer des objets Scala représentant un modèle et fournissant l'accès aux champs et aux opérations de modèles déclarés par `MM_CFG_Simple`.

8.5.2.4 Relations de sous-typage entre types de modèles

Les relations de sous-typage entre types de modèles sont assurées au niveau des types de modèles par une relation de sous-typage entre les traits Scala générés pour ces types de modèles. La Figure 8.6 présente un type de modèles `CFG_Instructions`, sous-type de `CFG_Simple`. Le Listing 8.21 présente le trait Scala généré pour ce type de modèles : `MT_CFG_Instructions`.

Listing 8.18 – Exemple de trait Scala généré pour le métamodèle `CFG_Simple`.

```
trait MM_CFG_Simple extends MT_CFG_Simple {
  //Variables de types pour l'ensemble des types objet du type de modèles exact extrait du
      métamodèle
  type ControlFlowGraph = ControlFlowGraphAspect
  type Node = NodeAspect
  //Champs de modèles typés à l'aide des variables de types
  var graphs : Set[ControlFlowGraph]
  //Accesseurs aux champs typés à l'aide des variables de types
  def getGraphs() : Set[ControlFlowGraph] = {
    return graphs
  }
  def setGraphs(newGraphs : Set[ControlFlowGraph]) = {
    graphs = newGraphs
  }
  //Accesseur à la Factory
  def getFactory() : IFactory_CFG_Simple = {
    return RichFactory_CFG_Simple
  }
}
```

Listing 8.19 – Exemple de trait et d'objet Scala générés pour la *Factory* du métamodèle `CFG_Simple`.

```
trait Factory_CFG_Simple extends IFactory_CFG_Simple {
  //Variables de types pour l'ensemble des types objets du type de modèles
  type ControlFlowGraph = ControlFlowGraphAspect
  type Node = NodeAspect
  //Opérations de créations d'objet typées à l'aide des variables de types
  def createControlFlowGraph() : ControlFlowGraph = {
    return new RichControlFlowGraph
  }
  def createNode() : Node = {
    return new RichNode
  }
}

object RichFactory_CFG_Simple extends Factory_CFG_Simple
```

Listing 8.20 – Exemple de classe Scala générée pour le métamodèle `CFG_Simple`.

```
class RichDSML_CFG_Simple extends MM_CFG_Simple
```

`MT_CFG_Instructions` étend `MT_CFG_Simple`, le trait généré pour le type de modèles `CFG_Simple`[3].

Puisque le sous-typage total isomorphique assure que chaque type objet du super-type a un type objet correspondant dans le sous-type, on retrouve dans `MT_CFG_Instructions` les mêmes variables de types que dans `MT_CFG_Simple`. De même, `MT_CFG_Instructions` déclare les mêmes signatures d'opérations et d'accesseurs pour les opérations et champs de modèles correspondants aux opérations et champs de modèles de `MT_CFG_Simple`.

Les métamodèles qui implémentent `MT_CFG_Instructions` doivent fixer les variables de types déclarées par `MT_CFG_Instructions`, et donc celles déclarées par `MT_CFG_Simple`. Ils doivent également fournir le code des opérations de modèles et des accesseurs déclarés par `MT_CFG_Instructions`, qui sont également déclarés par `MT_CFG_Simple`. Ainsi, les ob-

3. Puisque les types objets appartenant à deux types de modèles liés par une relation de sous-typage portent les mêmes noms, nous les différencions dans les exemples qui suivent par leurs noms qualifiés, incluant les packages : le package `simple` pour les types objets du type de modèles `MT_CFG_Simple` et le package `instructions` pour le type de modèles `MT_CFG_Instructions`

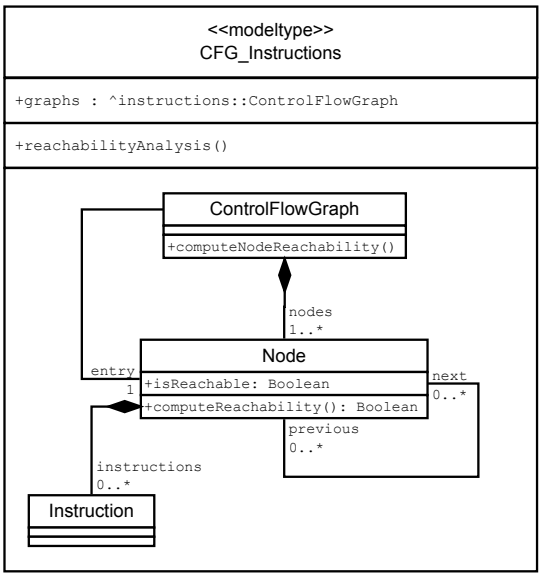

FIGURE 8.6 – Type de modèles CFG_Instructions tel que CFG_Instructions <: CFG_Simple.

Listing 8.21 – Exemple de trait Scala généré pour le type de modèles `CFG_Instructions`.

```
trait MT_CFG_Instructions extends MT_CFG_Simple {
  //Variables de types pour l'ensemble des types objets du type de modèles
  type ControlFlowGraph <: instructions.IControlFlowGraph
  type Node <: instructions.INode
  type Instruction <: instructions.IInstruction
  //Opérations de modèles typées à l'aide des variables de types
  def reachabilityAnalysis()
  //Accesseurs aux champs de modèles typés à l'aide des variables de types
  def getGraphs() : Set[ControlFlowGraph]
  def setGraphs(newGraphs : Set[ControlFlowGraph])
  //Accesseur à la Factory
  def getFactory() : IFactory_CFG_Simple
}
```

jets typés par `MT_CFG_Instructions` (qui représentent des modèles) sont également typés par `MT_CFG_Simple`.

Les relations de sous-typage entre types de modèles assurent la substituabilité de modèles, c'est à dire de groupes d'objets. Pour assurer cette substituabilité entre groupes d'objets Scala, nous nous appuyons sur les variables de types pour permettre des relations de sous-typage entre types objets de deux types de modèles différents.

Le Listing 8.22 présente le trait Scala généré pour le type objet `ControlFlowGraph` du type de modèles `MT_CFG_Instructions`. Ce trait, `instructions.IControlFlowGraph`, étend le trait généré pour le type objet `ControlFlowGraph` du type de modèles `MT_CFG_Simple`, `simple.IControlFlowGraph`. Ainsi, les objets typés par `instructions.IControlFlowGraph`, le sont aussi par `simple.IControlFlowGraph`. Ceci assure la substituabilité des objets dont les types appartiennent à deux types de modèles différents.

Les variables de types sont générées de la même manière pour toutes les classes d'un métamodèle. Une variable de type a donc la même valeur dans toutes les classes d'un même métamodèle. Cette valeur est donc partagée par tous les objets instances de ces classes. À l'inverse, la valeur d'une variable de type n'est partagée qu'au sein des classes d'un même

Listing 8.22 – Exemple de trait Scala généré pour le type objet `ControlFlowGraph` du type de modèles `CFG_Instructions`.

```
package instructions

trait IControlFlowGraph extends simple.IControlFlowGraph {
  //Variables de types pour l'ensemble des types objets du type de modèles auquel le type
      objet appartient
  type ControlFlowGraph <: instructions.IControlFlowGraph
  type Node <: instructions.INode
  type Instruction <: instructions.IInstruction
  //Opérations typées à l'aide des variables de types
  def computeNodeReachability()
  //Accesseurs aux champs typés à l'aide des variables de types
  def getNodes() : Set[Node]
  def setNodes(newNodes : Set[Node])
  def getEntry() : Node
  def setEntry(newEntry : Node)
}
```

métamodèle. En effet, les variables de type référencent des traits générés pour les classes d'un métamodèle donné. Ces traits ne sont donc pas partagés entre différents métamodèles. Ceci assure que les objets instances de classes appartenant à des métamodèles différents ne puissent pas être mélangés, malgré la substituabilité individuelle des objets.

8.6 Conclusion

Nous avons présenté dans ce chapitre l'implémentation d'un système de types orienté-modèle au sein de l'environnement de développement de langages de modélisation dédiés Kermeta. Ce système de types s'appuie sur les définitions des types de modèles, des métamodèles et des relations entre langages de modélisation dédiés développées en Partie II. Notre implémentation supporte la relation de sous-typage totale isomorphique entre types de modèles (cf. Définition 7.11) et une relation d'héritage entre métamodèles. Grâce à ces relations, il est possible de réutiliser la structure et le comportement d'un langage de modélisation dédié à un autre.

Nous avons modifié la syntaxe abstraite du métalangage Kermeta afin d'y intégrer les concepts de METAL, également présentés en Partie II. L'alignement entre METAL et Kermeta n'est pas parfait, afin de conserver la compatibilité avec le métalangage Ecore et l'Eclipse Modeling Framework. Nous avons également fourni une syntaxe concrète textuelle permettant de déclarer les concepts introduits dans Kermeta.

Enfin, nous avons exprimé la sémantique de notre système de types orienté-modèle au travers de la description du compilateur vers Scala. Afin d'émuler la correspondance de types objets (cf. Définition 7.1) et la substituabilité des groupes d'objets, le compilateur fait appel aux variables de types de Scala, qui permettent de définir des types chemin-dépendants.

Notre implémentation d'un système de types orienté-modèle au sein de Kermeta est extensible et peut supporter d'autres facilités de typage. En effet, le processus de compilation s'appuie uniquement sur les types de modèles et la relation de sous-typage totale isomorphique, depuis la vérification des relations de sous-typage jusqu'à la génération de bytecode. Ainsi, tout le back-end de Kermeta peut rester inchangé lors de l'ajout de nouvelles fonctionnalités au système de types.

Pour supporter d'autres relations de sous-typage (partielle ou non-isomorphique), l'extension de l'environnement de développement de langages de modélisation peut se faire en amont et indépendamment des étapes de vérification et de génération. L'ajout de fonctions d'adap-

tation et d'extraction de types de modèles effectifs peut donc se faire de manière modulaire, ces fonctions produisant des types de modèles pour ramener les relations de sous-typage partielles ou non-isomorphiques à la relation de sous-typage totale isomorphique (cf. Définitions 7.10 et 7.8).

De même, l'ajout de mécanismes de visibilité joue sur l'extraction des types de modèles depuis les métamodèles, mais ne modifie pas la manière de vérifier les relations de sous-typage. La plupart des autres mécanismes offrant des facilités de typage s'appuient directement sur la relation de typage (e.g., auto-complétion ou analyses d'impact). Étendre le système de types de Kermeta ne demande donc qu'un travail sur le front-end et la représentation intermédiaire de Kermeta, avant la vérification des relations de sous-typage.

Chapitre 9

Application à une passe de compilation optimisante : le passage en forme *Static Single Assignment*

Dans ce chapitre, nous présentons l'utilisation des types de modèles et des relations de sous-typage et d'héritage sur un cas d'utilisation provenant de la compilation optimisante. Le passage en forme Static Single Assignment (SSA) est une transformation courante au sein des compilateurs optimisants. La forme Static Single Assignment (SSA) est une représentation intermédiaire spécifique qui peut être extraite d'un graphe de flot de contrôle et qui facilite l'application de différentes analyses et optimisations [ALSU06].

Nous commençons par présenter la forme SSA (Section 9.1). Nous présentons ensuite les types de modèles et les transformations de modèles que nous avons définis pour implémenter le passage en forme SSA (Sections 9.2 et 9.3). Enfin, nous montrons comment ces transformations peuvent être réutilisées sur des modèles typés par d'autres types de modèles[1]. Nous établissons une relation de sous-typage totale isomorphique entre `SSA_Form`, le type de modèles qui supporte la conversion en forme SSA, et un type de modèles simple permettant d'exprimer des programmes manipulant des valeurs et variables entières (Section 9.4). Nous définissons également les adaptations nécessaires à l'établissement d'une relation de sous-typage totale non-isomorphique entre `SSA_Form` et un langage intermédiaire du compilateur GeCoS (Section 9.5).

9.1 Présentation de la forme *Static Single Assignment*

La forme Static Single Assignment (SSA) est une représentation intermédiaire particulière des graphes de flot de contrôle où chaque variable a une seule définition. Ceci permet de relier facilement les définitions et les usages des différentes variables d'un programme et d'éliminer certaines dépendances artificielles dues à l'usage d'une même variable pour deux tâches différentes. La forme Static Single Assignment (SSA) facilite ainsi un certain nombre d'analyses et d'optimisations au sein du compilateur (e.g., la propagation de constante, l'élimination de code mort, ou la ré-allocation de mémoire).

1. Ce chapitre ne présente pas l'intégralité du code écrit pour le passage en forme SSA. Plus de détails sur l'implémentation en Kermeta peuvent être trouvés sur la page https://sites.google.com/site/clementgguy/ssa.

Listing 9.1 – Programme original en pseudo-code

```
1 x := 0
2 x := 2
3 y := x+1
```

Listing 9.2 – Forme *Static Single Assignement* correspondante

```
1 x1 := 0
2 x2 := 2
3 y1 := x2+1
```

FIGURE 9.1 – Exemple simple de conversion en forme SSA

Listing 9.3 – Programme original en pseudo-code.

```
1 if (cond) then
2   x := 0
3 else
4   x := 1
5 end
6 y := x+1
```

Listing 9.4 – Forme SSA incomplète correspondant au programme du Listing 9.3.

```
1 if (cond) then
2   x1 := 0
3 else
4   x2 := 1
5 end
6 y1 := x?+1
```

Listing 9.5 – Forme SSA complète correspondant au programme du Listing 9.3.

```
1 if (cond) then
2   x1 := 0
3 else
4   x2 := 1
5 end
6 x3 := φ(x1, x2)
7 y1 := x3+1
```

FIGURE 9.2 – Exemple de conversion en forme SSA d'un programme contenant des branchements

Nous avons choisi d'appliquer notre système de types au passage en forme SSA car il s'agit non seulement d'une transformation courante au sein des compilateurs mais également non-triviale à implémenter. Le passage en forme SSA est donc susceptible d'être réutilisé, pour épargner aux concepteurs de compilateurs optimisants la tâche de l'implémenter. De plus, être capable de réutiliser le passage en forme SSA ouvre la voie à la réutilisation des autres passes d'analyses et d'optimisation qu'il facilite ou rend possible.

Pour transformer un graphe de flot de contrôle, ne possédant pas de contraintes sur le nombre de définitions d'une variable donnée, en une représentation sous la forme Static Single Assignment (SSA), il faut réécrire les définitions de variables afin de remplacer les n définitions de la même variable `x` par n définitions de différentes variables (x_1 ... x_n).

La Figure 9.1 présente un exemple simple de conversion d'un programme sous la forme SSA, où les définitions de `x` et `y` dans le Listing 9.1 sont réécrites en définitions de x_1, x_2, et y_1 dans le Listing 9.2. L'utilisation des variables doit alors être réécrite de la même manière afin de référencer la définition la plus récente de la variable utilisée (dans notre exemple , l'utilisation de `x` est réécrite en une utilisation de x_2).

Si la conversion à la forme SSA est simple au sein d'un bloc de base (i.e., en l'absence de branchement), la difficulté provient du flot de contrôle d'un programme. Lorsque deux (ou plus) branches différentes du graphe de flot de contrôle se rencontrent, il n'est pas possible de définir statiquement quelle est la définition à utiliser. En effet, le chemin pris par l'exécution du programme est déterminé dynamiquement et une variable peut être définie dans plusieurs branches.

Le Listing 9.3 présente une telle situation, où la dernière définition (`y`) fait usage de la définition de `x`. En fonction de la valeur de la condition booléenne `cond`, la définition de la branche `then` ou la définition de la branche `else` sera utilisée. Lors de la conversion du programme du Listing 9.3 en forme SSA, il n'y a pas de dernière variable définie entre x_1 et x_2 (nous indiquons donc $x_?$ dans le Listing 9.4).

Pour résoudre ce problème, la forme SSA insère une nouvelle définition dans chaque bloc

de base possédant plusieurs prédécesseurs. Cette définition fait appel à une ϕ-fonction. Une ϕ-fonction est une fonction qui prend n arguments x_1 ... x_n, n étant le nombre de chemins dans lesquels la variable x est définie et qui passe par le bloc où la ϕ-fonction est insérée, et qui renvoie la définition correspondante au chemin pris par le programme à l'exécution. Le Listing 9.5 présente la forme SSA complète pour le programme du Listing 9.3, où nous avons inséré une ϕ-fonction pour la variable `x`.

La conversion d'un graphe de flot de contrôle en forme SSA se fait en trois étapes :

- le calcul de la frontière de dominance pour chaque nœud du graphe de flot de contrôle ;
- l'insertion des ϕ-fonctions ;
- le renommage des variables.

Dans les deux sections suivantes, nous détaillons ces étapes et présentons les types de modèles et les transformations de modèles que nous avons définis pour implémenter le calcul de la frontière de dominance (cf. Section 9.2), et l'insertion de ϕ-fonctions et le renommage de variables (cf. Section 9.3).

Il existe différentes variantes du passage en forme SSA. Certaines, dites pruned et semi-pruned [BCHS98], permettent d'éviter d'insérer des ϕ-fonctions inutiles, e.g., à un endroit du programme où la variable n'est plus utilisée. D'autres permettent de prendre en compte les tableaux et les pointeurs, par exemple à l'aide d'analyse d'aliasing qui détectent les pointeurs vers la même variable.

La version que nous avons implémentée est plus simple. Elle ne pratique aucune optimisation du nombre de ϕ-fonctions insérées et ne permet pas de gérer les tableaux et les pointeurs. Pour une description complète des algorithmes mis en œuvre, nous reportons les lecteurs intéressés au Chapitre 19 du livre d'Appel et Palsberg [AP03, Chap. 19].

9.2 Calcul de la frontière de dominance

Le calcul de la frontière de dominance est un moyen efficace pour détecter les nœuds d'un graphe de flot de contrôle où il est nécessaire d'insérer des ϕ-fonctions. Les nœuds situés à la frontière de dominance d'un autre nœud peuvent nécessiter l'insertion d'une ϕ-fonction. Un nœud d d'un graphe de flot de contrôle domine un autre nœud n si tous les chemins du nœud d'entrée à n passent par d.

Parmi les dominateurs d'un nœud n, on différencie le nœud n (qui se domine lui-même) de ses dominateurs stricts : tous les nœuds qui dominent n mais qui ne sont pas n. La frontière de dominance d'un nœud d est l'ensemble de tous les nœuds n tels que d domine un prédécesseur immédiat de n, mais qu'il ne domine pas strictement n. Les nœuds à la frontière de dominance de d sont les nœuds où au moins deux branches se rejoignent : une branche venant de d et ses successeurs, et une autre. Ce sont dans ces nœuds qu'il peut être nécessaire d'insérer des ϕ-fonctions pour les variables qui sont définies dans d.

Le calcul de la frontière de dominance peut être fait sur une structure simple exposant les branches du graphe de flot de contrôle. La Figure 9.3 présente le type de modèles que nous avons défini pour le calcul de la frontière de dominance. Ce type de modèles représente des graphes de flot de contrôle, composés de nœuds liés par une relation successeurs/prédécesseurs et possédant une frontière de dominance composée d'un ensemble de nœuds.

Le Listing 9.6 déclare un métamodèle `CFG_Dominance` implémentant ce type de modèles. `CFG_Dominance` hérite du métamodèle `CFG_Simple` présenté en Chapitre 8. `CFG_Dominance` ajoute un champ `dominanceFrontier` aux nœuds (fichier `NodeAspect.kmt`).

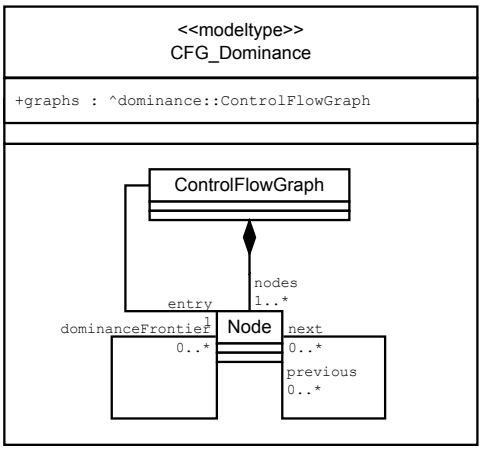

FIGURE 9.3 – Type de modèles `CFG_Dominance` pour le calcul la frontière de dominance.

Listing 9.6 – Déclaration du métamodèle `CFG_Dominance` implémentant le type de modèles de la Figure 9.3 en *kp*.

```
metamodel CFG_Dominance extends CFG_Simple {
  import "${project.baseUri}/src/main/kmt/
    NodeAspect.kmt"
}
```

Listing 9.7 – Calcul de la frontière de dominance définie sur le type de modèles `CFG_Dominance` en Kermeta.

```
metamodel DominanceFrontierAnalysis {
  //See Modern Compiler Implementation in Java, 2nd ed. p.406
  operation computeDominanceFrontier[MT ^: CFG_Dominance](m : MT) is do
    computeDominatorTree[MT](m)
    m.graphs.each{cfg|
      cfg.nodes.each{n|
        var s : Set[MT^simple::Node] init Set[MT^simple::Node].new
        n.next.each{y|
          if y.iDominator != n then
            s.add(y)
          end
        }
        n.iDominated.each{c|
          computeDominanceFrontier(c)
          c.dominanceFrontier.each{w|
            if (not w.dominators.contains(n)).orElse{x|n.equals(w)} then
              s.add(w)
            end
          }
        }
        n.dominanceFrontier.addAll(s)
      }
    }
  end
}
```

Nous avons défini une transformation de modèles calculant la frontière de dominance et acceptant un modèle typé par `CFG_Dominance` (cf. Listing 9.7). Cette transformation déclare un paramètre de type `MT` (cf. ligne 3) dont la borne est le type de modèles `CFG_Dominance`. Ce paramètre de type est utilisé pour typer le paramètre de la transformation et pour déclarer des types chemin-dépendants (e.g., le type de l'ensemble `s` à la ligne 7). Ainsi, cette transformation est réutilisable pour tous les sous-types du type de modèles `CFG_Dominance`. Elle accepte en paramètre un modèle typé par `CFG_Dominance` ou l'un de ses sous-types et l'ensemble `s` est un ensemble d'objets dont le type objet correspond au type objet `Node` de `CFG_Dominance`.

9.3 Insertion de ϕ-fonctions et renommage de variables

L'insertion de ϕ-fonctions est nécessaire lorsque plusieurs branches du graphe de flot de contrôle se rencontrent et que plusieurs définitions possibles de la même variable sont disponibles (une pour chaque branche). La structure de classes nécessaire pour l'insertion de ϕ-fonctions comprend, en plus de la structure de graphe de flot de contrôle, les instructions qui définissent et utilisent des variables et les symboles utilisés pour nommer ces variables.

Le renommage des variables remplace les définitions et les utilisations de variables, par des définitions numérotées et des utilisations directement liées à une définition. En plus des concepts nécessaires à l'insertion de ϕ-fonctions, il faut donc ajouter des concepts représentant ces définitions et utilisations spécifiques pour pouvoir renommer les variables.

Le type de modèles de la Figure 9.4 présente les types objets utilisés pour représenter ces concepts. Un graphe de flot de contrôle contient, en plus de ses nœuds des symboles (`Symbol`) qui désignent les variables. Les nœuds du graphe de flot de contrôle contiennent une séquence d'instructions (`Instruction`). Chaque instruction peut définir plusieurs variables (`Definition`) et utiliser plusieurs variables (`Use`). Définitions et utilisations pointent vers un symbole qui désigne la variable définie ou utilisée.

Les ϕ-fonctions (`PhiFunction`) peuvent être utilisées pour définir une variable en fonction des définitions précédentes présentes dans les branches du graphe. Lors du renommage des variables, les définitions sont remplacées par des définitions SSA (`SSADefinition`) numérotées. Les utilisations de variables sont elles remplacées par des utilisations SSA (`SSAUse`) qui sont directement liées à la définition la plus récente de la variable utilisée.

Comme pour le calcul de la frontière de dominance, nous avons défini un métamodèle `SSA_Form` qui étend le métamodèle `CFG_Dominance` et qui implémente le type de modèles présenté en Figure 9.4. Nous avons également défini des transformations de modèles qui acceptent en paramètre un modèle typé par le type de modèles `SSA_Form`. Ces transformations insèrent les ϕ-fonctions nécessaires et renomment les variables de chacun des graphes de flot de contrôle du modèle afin de les convertir en forme SSA (cf. Listing 9.8 et 9.9). Là encore, ces transformations sont réutilisables sur tous les modèles typés par les sous-types de `SSA_Form`. L'utilisation d'un paramètre de type permet la déclaration de types chemin-dépendants et permet également d'instancier des objets typés par ces types chemin-dépendants. Par exemple, les lignes 19 et 20 du Listing 9.8 créent la nouvelle définition utilisant une ϕ-fonction qui sera insérée dans un nœud. Pour cela, une définition, de type `MT^ssa::Definition`, et une ϕ-fonction, de type `MT^ssa::PhiFunction`, sont instanciées. Selon le paramètre de type réel passé à l'appel de la transformation, la définition et la ϕ-fonction seront instanciées à partir de la classe appartenant au métamodèle du modèle traité. Ainsi, l'ajout de cette définition au modèle ne risque pas de mener à un modèle invalide.

La transformation de renommage de variables s'appuie notamment sur les opérations `replaceUse` et `replaceDef`, qui remplacent une instance de `Use` (respectivement une instance de `Definition`) par une instance de `SSAUse` (respectivement une instance de `SSADefinition`).

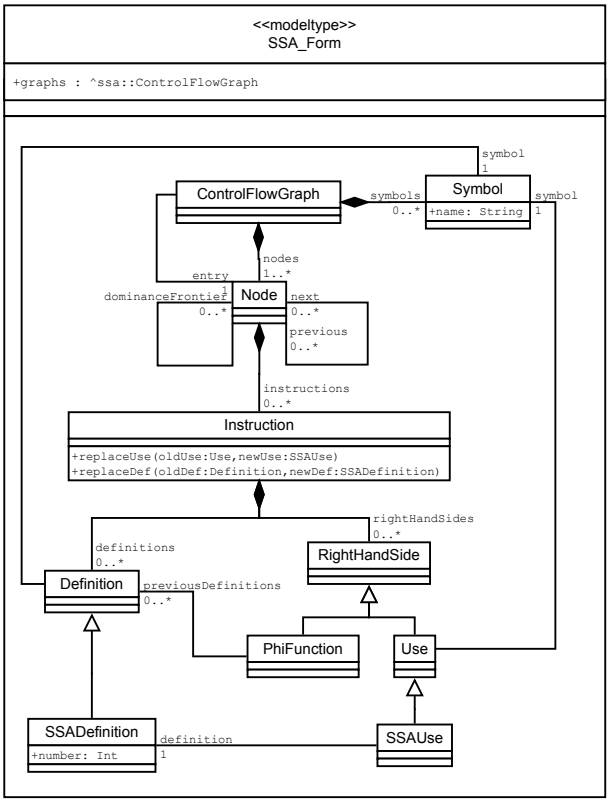

FIGURE 9.4 – Type de modèles `SSA_Form` pour le passage en forme SSA.

Le code de ces opérations est simple dans le cas du métamodèle `SSA_Form`, il s'agit d'insérer la nouvelle instance de `SSAUse` à l'index de l'ancienne instance de `Use` avant de retirer celle-ci (idem pour les instances de `SSADefinition` et les instances de `Definition`).

Ces opérations peuvent être plus complexes dans les métamodèles qui réutilisent le passage en forme SSA et qui contiennent des instructions plus complexes formant un arbre, par exemple. Dans de tels cas, l'instance de `Use` ou l'instance de `Definition` à remplacer n'est pas forcément contenue directement par l'instruction contenue par le nœud, mais peut être contenue par une sous-instruction. L'opération `replaceUse` (ou `replaceDefinition`) doit donc calculer l'endroit où insérer la nouvelle instance de `SSAUse` (ou `SSADefinition`).

9.4 Réutilisation du passage en forme *Static Single Assignment*

La Figure 9.5 présente un type de modèles `CFG_Int` de graphes de flot de contrôles permettant d'exprimer des programmes manipulant des valeurs et variables entières. Il est possible de réutiliser les transformations définies pour la conversion à la forme Static Single Assignment (SSA) sur les modèles typés par ce type de modèles. Ce type de modèles simple présente une extension de `SSA_Form` qui conserve le sous-typage total isomorphique. Il est ainsi possible de l'étendre encore pour prendre en compte des modèles plus complets représentant des programmes complexes, utilisant différents types de données et différentes opérations pour

Listing 9.8 – Insertion de ϕ-fonction définie sur le type de modèles `SSA_Form` en Kermeta.

```
metamodel SSAConversion {
  //See Modern Compiler Implementation in Java, 2nd ed. p.407
  operation insertPhiFunctions[MT ^: SSAForm](m : MT) : Void is do
    computeDefSites[MT](m)
    m.graphs.each{cfg|
      var workingSet : OrderedSet[MT^simple::Node] init OrderedSet[MT^simple::Node].new
      cfg.symbols.each{a|
        workingSet.addAll(a.defSites)
        from var finish : Boolean init workingSet.isEmpty() until finish loop
          var n : MT^simple::Node init workingSet.first
          workingSet.remove(n)
          var a_phi : MT^ssa::PhiFunction
          n.dominanceFrontier.each{y|
            if y.phiSymbols.isVoid() then
              y.phiSymbols := Set[MT^ssa::Symbol].new
            end
            if not y.phiSymbols.contains(a) then
              var phi_instruction : MT^ssa::Instruction init MT^ssa::Instruction.new
              var phi_def : MT^ssa::Definition init MT^ssa::Definition.new
              a_phi := MT^ssa::PhiFunction.new
              phi_def.symbol := a
              phi_instruction.rightHandSides.add(a_phi)
              phi_instruction.definitions.add(phi_def)
              y.instructions.addAt(0, phi_instruction)
              y.phiSymbols.add(a)
              if not y.definedSymbols.contains(a) then
                workingSet.add(y)
              end
            end
          }
          finish := workingSet.isEmpty()
        end
      }
    }
  end
}
```

manipuler ces types.

`CFG_Int` définit les concepts nécessaires à l'expression de programmes manipulant des valeurs entières, sous la forme de graphes de flot de contrôle dont les nœuds contiennent des instructions manipulant des entiers. Parmi ces instructions, les `Definition` correspondent à la déclaration d'une variable et les `Use` à la référence d'une variable. `CFG_Int` définit également les opérations binaires classiques sur les entiers (`Plus`, `Mult`, `Minus` et `Div`) et la déclaration de constante (`IntConstant`).

Le Listing 9.11 présente le fichier kp déclarant un métamodèle implémentant le type de modèles `CFG_Int`. Ce métamodèle étend `SSA_Form` en lui ajoutant les classes nécessaires aux manipulations simples de valeurs entières. Le Listing 9.12 déclare une relation de sous-typage entre le type extrait du métamodèle `CFG_Int` et celui extrait de `SSA_Form`.

Cette relation de sous-typage est vérifiée par le compilateur Kermeta, en suivant la définition de la relation de sous-typage totale isomorphique (cf. Sections 7.1.7.1 et 8.5.1.2) :

- `SSA_Form` ne possède qu'un champ de modèles `graphs`, typé par `^ssa::ControlFlowGraph`, `CFG_Int` possède également un champ de modèles `graphs`, et ce champ est typé par le type `^int::ControlFlowGraph` tel que `^ssa::ControlFlowGraph <# ^int::ControlFlowGraph` ;
- `SSA_Form` ne possède pas d'opération de modèles ;
- pour chacun des dix types objets de `SSA_Form`, il existe un type correspondant dans `CFG_Int`.

Afin d'assurer la cohérence de la forme SSA obtenue après l'application de la conversion,

Listing 9.9 – Passage en forme SSA défini sur le type de modèles `SSA_Form` en Kermeta.

```
metamodel SSAConversion {
  //See Modern Compiler Implementation in Java, 2nd ed. p.409
  operation renameVariables[MT ^: SSA_Form](m : MT) is do
    insertPhiFunctions[MT](m)
    m.graphs.each{cfg|
      cfg.symbols.each{a|
        a.counter := 1
        a.ssaDefStack := Sequence[MT^ssa::SSADefinition].new

        var initialDef : SSADefinition init MT^ssa::SSADefinition.new
        initialDef.symbol := a
        initialDef.number := 0
        a.ssaDefStack.add(initialDef)
        initialInstruction.definitions.add(initialDef)
      }
      renameVariablesInNode[MT](cfg.entry)
    }
  end
}
```

Listing 9.10 – Redéfinition de l'opération `replaceUse` dans `CFG_Int`.

```
aspect class Instruction
  operation replaceUse(oldUse : Use, newUse : SSAUse) is do
    //Replace the oldUse by the newUse in the rightHandSides
    var index : Integer init self.rightHandSides.getIndexOf(oldUse)
    self.rightHandSides.addAt(index, newUse)
    self.rightHandSides.remove(oldUse)
    //Replace references to the oldUse
    if (not oldUse.operationLeft.isVoid) then
      oldUse.operationLeft.left := newUse
    end
    if (not oldUse.operationRight.isVoid) then
      oldUse.operationRight.right := newUse
    end
  end
}

aspect class RightHandSide {
  //Reference to the binary operation for which this RightHandSide is the left operand
  reference operationLeft : BinaryOperation#left
  //Reference to the binary operation for which this RightHandSide is the right operand
  reference operationRight : BinaryOperation#right
}
```

`CFG_Int` redéfinit l'opération `replaceUse`. En effet, quand une instance de `Use` est remplacée par une instance de `SSAUse` au sein d'une instruction, il faut également que les potentielles opérations (`BinaryOperation`) qui référençaient l'instance de `Use` en tant qu'opérande gauche ou droit référencent à la place l'instance de `SSAUse`. Le Listing 9.10 présente la redéfinition de `replaceUse`, ainsi que l'ajout de deux champs à la classe `RightHandSide`, qui permettent de connaître l'opération qui référence une instance de cette classe, si il y en a une. Ces champs sont déclarés champs opposés des champs `left` et `right` de la classe `Operation` et sont donc modifiés simultanément.

Grâce à la relation de sous-typage entre `CFG_Int` et `SSA_Form`, il est possible de réutiliser les transformations définies dans les Sections 9.2 et 9.3 sur les modèles typés par `CFG_Int`.

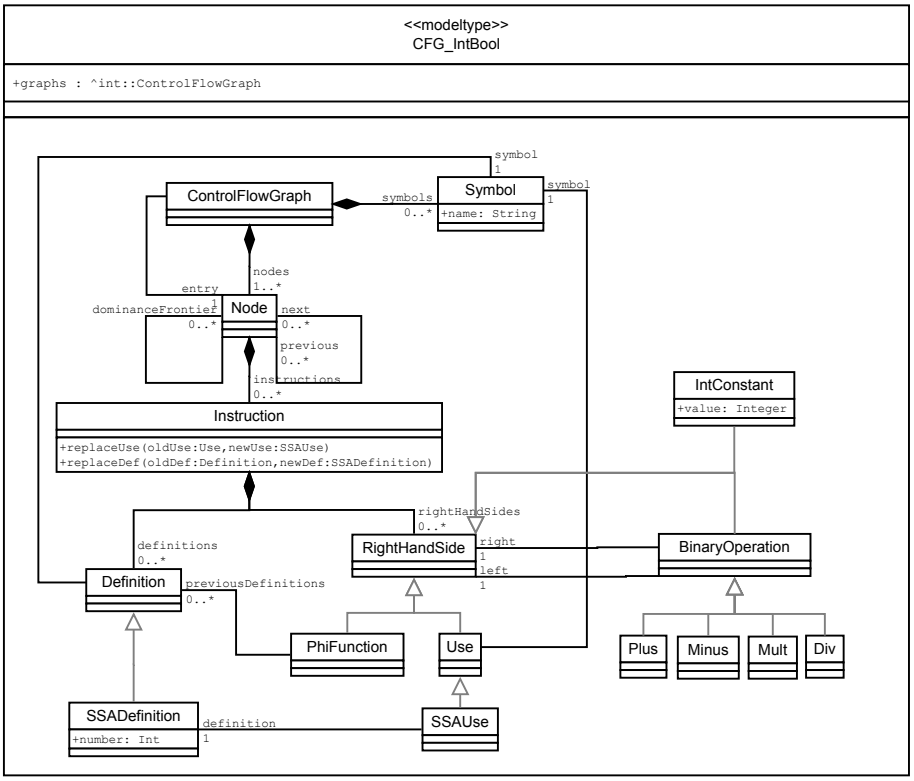

FIGURE 9.5 – Type de modèles `CFG_Int` contenant des instructions entières.

Listing 9.11 – Déclaration du métamodèle `CFG_Int` implémentant le type de modèles de la Figure 9.5 en *kp*.

```
metamodel CFG_Int extends SSA_Form {
  import "${project.baseUri}/src/main/ecore/
    BinaryOperation.ecore"
  import "${project.baseUri}/src/main/kmt/
    BinaryOperationAspect.kmt"
  import "${project.baseUri}/src/main/kmt/
    IntConstant.kmt"
}
```

Listing 9.12 – Déclaration de la relation de sous-typage entre le type de modèles `CFG_Int` et le type de modèles `SSA_Form` en Kermeta.

```
metamodel CFG_Int {
  subtypeof SSA_Form
}
```

Listing 9.13 – Réutilisation du passage en forme SSA sur un modèle typé par `CFG_Int`.

```
var model : CFG_Int
...
var ssaConverter : SSAConversion init SSAConversion.new
ssaConverter.renameVariables[CFG_Int](model)
```

Le Listing 9.13 présente l'appel de la transformation `renameVariables` sur un modèle typé par `CFG_Int`. En passant `CFG_Int` en paramètre de la transformation, les types chemin-dépendants que cette transformation déclare sont résolus vers les types objets réels. Ainsi, quand `CFG_Int` est passé en paramètre de `renameVariables`, le type `MT^ssa::SSADefinition` (cf. ligne 8 du Listing 9.9) est résolu vers le type `CFG_Int^ssa::SSADefinition`.

Les transformations définies sur les types de modèles `CFG_Dominance` et `SSA_Form` sont ainsi réutilisables sur des modèles instances de différents métamodèles, i.e., sur des modèles dont les objets sont instances de classes appartenant à différents groupes.

9.5 Adaptations nécessaires pour réutiliser le passage en forme *Static Single Assignment* dans GeCoS

Dans cette section, nous présentons les adaptations qui seraient nécessaires pour pouvoir réutiliser notre passage en forme Static Single Assignment (SSA) sur les représentations intermédiaires d'un compilateur existant, GeCoS. Bien que l'intégralité de ces adaptations ne soient pas supportées par Kermeta à l'heure actuelle, nous exposons la forme qu'elles devraient prendre et la façon dont elles devraient être traitées par le compilateur Kermeta.

GeCoS[2] (pour Generic Compiler Suite) est une infrastructure de compilation optimisante développée pour la conception de processeurs dédiés et d'accélérateurs matériels. GeCoS comprend notamment un compilateur C source-à-source qui pratique différentes passes d'optimisation avant de faire appel à des outils de génération de descriptions matérielles. De plus, GeCoS est conçu selon les principes de l'ingénierie dirigée par les modèles autour de plusieurs langages de modélisation dédiés à différentes représentations intermédiaires (graphes de flot de contrôle et forme SSA, représentation polyédrique).

GeCoS est donc une infrastructure de compilation appliquant aux représentations intermédiaires des passes d'analyses et d'optimisation complexes sous la forme de transformation de modèles. Ces transformations de modèles sont définies sur les langages de modélisation dédiés internes à GeCoS. La réutilisation de passes de transformation définies sur d'autres langages de modélisation dédiés, comme le passage en forme SSA défini plus haut permettrait de réduire les coûts de développement du compilateur en facilitant l'ajout de nouvelles passes.

9.5.1 Graphes de flot de contrôle dans GeCoS

L'un des langages de modélisation dédiés de GeCoS est consacré aux graphes de flot de contrôle des procédures C et définit la représentation intermédiaire centrale de GeCoS. Dans la suite, nous désignerons ce langage sous le nom de `GeCoS IR`.

GeCoS fournit de nombreuses passes d'analyse et d'optimisation pour les graphes de flot de contrôle définies sur la syntaxe abstraite de `GeCoS IR`, y compris le passage en forme SSA. La syntaxe abstraite de `GeCoS IR` est bien plus complexe que celle du métamodèle `SSA_Form` : elle contient au total 122 classes. Cependant, il est possible d'isoler un sous-ensemble de ces classes qui sont concernées par le passage en forme SSA.

La Figure 9.6 présente ce sous-ensemble. La racine de la syntaxe abstraite est la classe `Procedure` qui contient un ensemble de `BasicBlock`, dont l'un est le point d'entrée (`start`) et qui sont reliés entre eux par des `ControlEdge`. Un `BasicBlock` contient des `Instruction`, parmi lesquelles on trouve les `SymbolInstruction` (les références à une variable) et les `PhiInstruction` (les ϕ-fonctions). Les `SymbolInstruction` référencent un `Symbol` (qui contient le nom de la

2. http://gecos.gforge.inria.fr

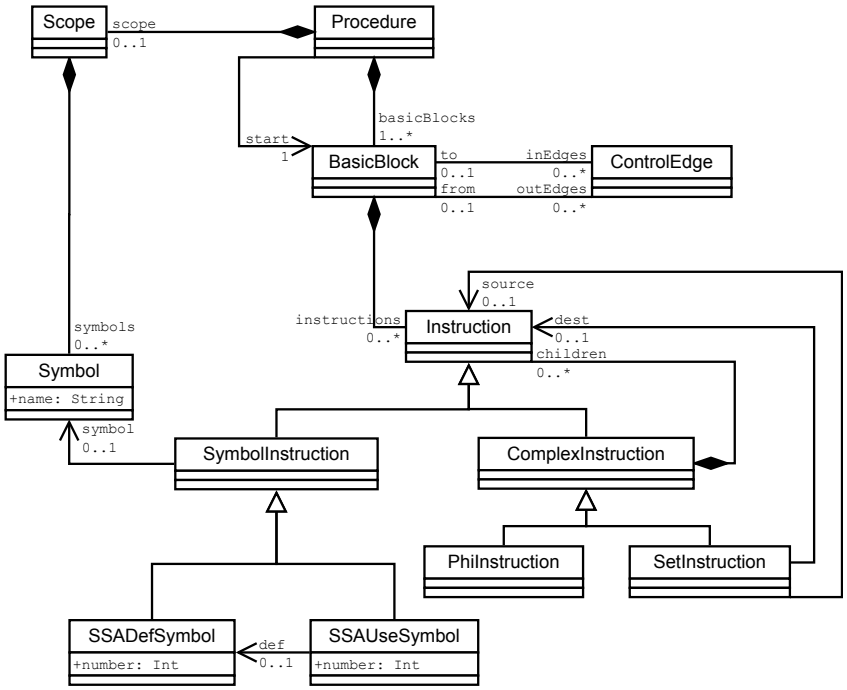

FIGURE 9.6 – Extrait de la syntaxe abstraite de `GeCoS IR`, le langage de graphes de flot de contrôle de GeCoS.

variable) contenu dans le `Scope` de la `Procedure`. Enfin, il existe deux `SymbolInstruction` particulières : les `SSADefSymbol` (les définitions de variables en forme SSA) et les `SSAUseSymbol` (les utilisations de variables en forme SSA).

9.5.2 Adaptations entre `GeCoS IR` et `SSA_Form`

Malgré les noms de classes et de champs différents, on retrouve dans la syntaxe abstraite de `GeCoS IR` des concepts équivalents de ceux de la syntaxe abstraite de `SSA_Form`. Une `Procedure` de `GeCoS IR` est équivalente à un `ControlFlowGraph` de `SSA_Form`. Elle est composée de `BasicBlock`, là où un `ControlFlowGraph` est composé de `Node`. Les `BasicBlock`, comme les `Node` contiennent des `Instruction` et sont liés par des relations de successeurs/prédécesseurs, etc. La Table 9.1 résume ces équivalences entre les classes de `SSA_Form` et les classes de `GeCoS IR`.

En dehors des différences de noms, il subsiste quelques hétérogénéités structurelles entre `SSA_Form` et `GeCoS IR` :

- les `BasicBlock` de `GeCoS IR` sont reliés au travers de la classe `ControlEdge`, là où les `Node` de `SSA_Form` sont reliés par une association de `Node` vers `Node` ;
- `BasicBlock` ne définit pas de champ pour stocker les blocs de sa frontière de dominance, contrairement à `Node` ;
- les `Symbol` de `GeCoS IR` ne sont pas contenus directement par la `Procedure`, mais par son `Scope` ;
- les définitions et les utilisations de variables ne sont pas différenciées dans `GeCoS IR`, ce sont dans tous les cas des `SymbolInstruction` ;

TABLE 9.1 – Équivalences entre les classes de `SSA_Form` et les classes de `GeCoS IR`.

Classe de `SSA_Form`	Classe de `GeCoS IR`
`ControlFlowGraph`	`Procedure`
`Symbol`	`Symbol`
`Node`	`BasicBlock`
`Instruction`	`Instruction`
`RightHandSide`	`Instruction`
`Definition`	`SymbolInstruction`
`Use`	`SymbolInstruction`
`PhiFunction`	`PhiInstruction`
`SSADefinition`	`SSADefSymbol`
`SSAUse`	`SSAUseSymbol`

– là où toutes les `Instruction` de `SSA_Form` peuvent être une affectation, si elles contiennent une où plusieurs `Definition`, `GeCoS IR` différencie les affectations comme une `Instruction` particulière : `SetInstruction` ;

– les `PhiInstruction` de `GeCoS IR` ne pointent pas directement sur les définitions précédentes, l'ensemble des définitions précédentes est calculé aux travers des instructions filles (`children`) d'une `PhiInstruction`, qui sont des `SSAUseSymbol` pointant vers les définitions précédentes.

Pour pouvoir réutiliser le passage en forme SSA écrit sur le type de modèles `SSA_Form`, il faut établir une relation de sous-typage totale non-isomorphique entre `GeCoS IR` et `SSA_Form` : totale puisque l'on veut réutiliser toutes les transformations définies sur `SSA_Form`, et non-isomorphique à cause des hétérogénéités structurelles présentées ci-dessus.

Pour établir cette relation de sous-typage, il est nécessaire de définir une adaptation entre `GeCoS IR` et `SSA_Form`. Cette adaptation doit être bidirectionnelle, puisque le passage en forme SSA est une transformation qui modifie le modèle en y ajoutant de nouveaux éléments (cf. Section 7.1.6.2).

Dans la suite de cette section nous présentons la manière dont cette adaptation bidirectionnelle pourrait être implémentée en Kermeta. Nous définissons les adaptations individuelles (au niveau des classes et des champs) qui composent l'adaptation entre `GeCoS IR` et `SSA_Form` à l'aide d'aspects, permettant d'étendre la structure de `GeCoS IR` pour créer une relation de sous-typage totale isomorphique.

Bien que ces adaptations ait été établies manuellement et qu'une partie d'entre elles ne soient actuellement pas supportées par Kermeta, elles s'appuient sur des motifs reproductibles. Ainsi, il serait possible de générer de telles adaptations à partir d'une spécification de plus haut niveau que les aspects présentés dans la suite de cette section. De plus, la structure de `GeCoS IR` est modifiée par l'ajout de classes et d'aspects dans des fichiers séparés, permettant la définition d'une adaptation sans modifier les spécifications existantes de `GeCoS IR`, et plus particulièrement le fichier Ecore décrivant sa syntaxe abstraite.

9.5.2.1 Renommage de classes

Les différences les plus évidentes entre les syntaxes abstraites de `GeCoS IR` et de `SSA_Form` sont probablement les différences de nommage des classes équivalentes. Pour que l'adaptation de `GeCoS IR` vers `SSA_Form` crée une relation de sous-typage totale isomorphique, il faut que cette adaptation renomme les classes de `GeCoS IR`.

Afin de ne pas modifier l'outil de vérification des relations de sous-typage et de rester le moins invasif possible par rapport au métamodèle de `GeCoS IR`, nous avons choisi de mettre en œuvre ce renommage par la création de nouvelles classes "de renommage".

Nous étendons donc la syntaxe abstraite originale pour qu'elle contienne des classes correspondant aux classes de `SSA_Form`. Ces classes de renommage fournissent une indirection vers les classes originales, à travers laquelle il est possible d'accéder aux informations des modèles selon l'interface attendue par les transformations. Une classe de renommage porte le nom d'une classe du super-type (dans notre cas `SSA_Form`) et fournit une indirection vers une classe originale du sous-type (`GeCoS IR`).

Listing 9.14 – Exemple de renommage de classes par ajout de la classe `Node` dans `GeCoS IR` en Kermeta.

```
1  package gecos::core {
2    //BasicBlock2Node adaptation
3    class Node {
4      //Reference to the BasicBlock object for which this object provides an adaptation
5      reference basicBlock_adaptee : BasicBlock
6      ...
7      property instructions : seq Instruction[0..*]
8      _get is do
9        result := basicBlock_adaptee.instructions
10     end
11   }
12   aspect class BasicBlock {
13     //Reference to the Node object adapting this object
14     reference node_adapter : Node
15   }
16 }
```

Par exemple, le Listing 9.14 ajoute une classe `Node` au package `gecos::core`. Cette classe déclare un champ `basicBlock_adaptee` vers la classe qu'elle renomme : `BasicBlock`. C'est à travers ce champ que les objets `Node` pourront accéder aux informations contenues dans l'objet `BasicBlock` correspondant. Le Listing 9.14 ajoute également un champ `node_adapter` à la classe `BasicBlock` qui permet de naviguer en sens inverse la relation entre une variable adapter (de la classe originale vers la classe de renommage) et un objet adaptee (de la classe de renommage vers la classe originale).

Les variables adapter et adaptee sont utilisées pour calculer des champs dérivés qui permettent d'accéder aux informations d'une classe au travers de la classe qui la renomme. Par exemple, pour accéder à l'ensemble des instructions contenues par un bloc de base, le passage en forme SSA défini sur `SSA_Form` fait appel au champ `instructions` de la classe `Node`. Pour exposer la même interface depuis un modèle de `GeCoS IR`, il faut que la classe `Node` ajoutée au package `gecos::core` de `GeCoS IR` contienne un champ `instructions` qui pointe vers l'ensemble des instructions d'un `BasicBlock`. Ces instructions sont accessibles au travers du champ `basicBlock_adaptee` (cf. ligne 7 à 10 du Listing 9.14).

La création des classes de renommage, des aspects sur les classes originales et des variables adapter et adaptee obéit à un motif simple : pour chaque paire de classes (A, B), équivalentes mais ne portant pas le même nom, A étant la classe du sous-type et B la classe du super-type, une classe de renommage nommée B est générée dans le sous-type. Cette classe possède une variable nommée A`_adaptee` pointant vers A. De plus un aspect est généré pour A lui ajoutant une variable nommée B`_adapter` et pointant vers la classe B générée.

Ces adaptations peuvent donc être générées à partir d'une spécification de plus haut niveau, plutôt qu'écrites manuellement. Une telle spécification devrait uniquement contenir les paires de classes équivalentes.

9.5.2.2 Renommage de champs

Le renommage de champs peut se faire en Kermeta à l'aide d'aspects injectant un champ dérivé portant le bon nom et redirigeant vers le champ originel. Par exemple le champ `start` de la classe `Procedure` de `GeCoS IR` équivaut au champ `entry` de la classe `ControlFlowGraph` de `SSA_Form`. Pour renommer `start` en `entry`, il est possible de déclarer un aspect sur `Procedure` définissant un champ dérivé `entry` calculé à partir de `start` (cf. Listing 9.15).

Le champ dérivé peut nécessiter de passer par les objets `adapter` et `adaptee` si la classe du champ a été renommée. Par exemple, dans le Listing 9.15, le champ dérivé `entry` est typé par `Node`, alors que le champ correspondant de `Procedure`, `start`, est typé par `BasicBlock`. C'est pourquoi les accesseurs utilisent les objets `node_adapter`, pour obtenir un `Node` à partir du `BasicBlock` `start` (ligne 5), et `basicBlock_adaptee`, pour obtenir un `BasicBlock` à partir du `Node` `_value` (ligne 8).

Listing 9.15 – Exemple de renommage de champ à l'aide d'un champ dérivé en Kermeta.

```
1  aspect class Procedure {
2    //Adaptation to ControlFlowGraph.entry
3    property entry : Node
4    _get is do
5      result := start.node_adapter
6    end
7    _set is do
8      start := _value.basicBlock_adaptee
9    end
10 }
```

Les champs dérivés dont la borne supérieure est 1, i.e., les champs dérivés qui ne retournent qu'un objet (comme le champ dérivé `entry` de la classe `Procedure` dans le Listing 9.15) sont intégralement supportés par Kermeta. Ils sont compilés en Scala vers une paire d'accesseurs (`getter` et `setter`), dont le corps est le résultat de la compilation des opérations `_get` et `_set`.

Cependant, les champs dérivés dont la borne supérieure est supérieure à 1, i.e., les champs dérivés qui retournent une collection d'objets (i.e., les collections dérivées) ne sont supportés qu'en lecture seule par Kermeta. Ceci ne permet pas de définir des adaptations bidirectionnelles, puisque les champs renommés ne peuvent alors pas être modifiés par les transformations de modèles.

Pour pouvoir définir des collections dérivées modifiables, il faut définir les opérations de manipulations de ces collections (e.g., `add`, `remove`, etc.). Il est ensuite nécessaire de compiler chaque collection dérivée vers une classe générique Scala spécifique avec ses propres implémentations des opérations `add`, `remove`, etc.

Par exemple, le Listing 9.16 présente la définition d'un champ dérivé `nodes` pour la classe `Procedure` tel qu'il pourrait apparaître en Kermeta. En plus de définir un accesseur en lecture (`_get`), le champ dérivé définit deux opérations (`_add` et `_remove`).

Une fois compilé en Scala, `Procedure` devrait donc posséder une opération `getNodes`. Cette opération devrait retourner un ensemble de `BasicBlock`, sous la forme d'une collection définie spécifiquement pour ce champ dérivé. Cette collection devrait donc redéfinir les opérations `add` et `remove` en fonction du comportement décrit par les opérations Kermeta `_add` et `_remove`.

Tout comme le renommage de classes, le renommage de champs peut être généré à partir d'une spécification des paires de champs correspondants. Les collections dérivées doivent fournir les opérations de manipulation des collections, mais celles-ci ne sont que des appels aux opérations correspondantes de la collection renommée. Cependant, un renommage de champ doit prendre en compte les éventuels renommages de classes existants. Si le champ renommé est typé par une classe A renommée en B, le champ dérivé doit pointer vers la classe

Listing 9.16 – Exemple de renommage de champ à l'aide d'une collection dérivée en Kermeta.

```
aspect class Procedure {
  //Adaptation to ControlFlowGraph.nodes
  property nodes : set Node[0..*]
  //Returns Node objects from the original collection of BasicBlock objects
  _get is do
    result := Set[Node].new
    basicBlocks.each{bb|
      result.add(bb.node_adapter)
    }
  end
  //Adds to the original collection a BasicBlock object from the Node object _element
  _add is do
    basicBlocks.add(_element.basicBlock_adaptee)
  end
  //Removes from the original collection a BasicBlock object from the Node object _element
  _remove is do
    basicBlocks.remove(_element.basicBlock_adaptee)
  end
}
```

de renommage *B* et utiliser les variables adapter et adaptee pour accéder et modifier le champ original.

Bien que notre exemple ne présente pas de renommage d'opération, les opérations peuvent être renommées d'une manière similaire aux champs. Il est possible d'ajouter par aspect une nouvelle opération portant le bon nom à une classe. Le corps de cette nouvelle opération est alors simplement un appel à l'opération originale. Le renommage d'opérations doit également prendre en compte le renommage de classes, en utilisant les variables adapter et adaptee.

9.5.2.3 Autres adaptations

Certaines hétérogénéités structurelles présentes entre `GeCoS IR` et `SSA_Form` ne concernent pas les noms des classes ou des champs, mais des différences plus complexes. Par exemple, le lien entre `Procedure` et `Symbol` dans `GeCoS IR` n'est pas direct, contrairement au lien entre `ControlFlowGraph` et `Symbol` dans `SSA_Form`. Ces hétérogénéités doivent donc être traitées de manière plus spécifique. Toutes ces adaptations s'appuient également sur les aspects et les champs dérivés Kermeta.

Le champ `symbols` peut être défini dans la classe `Procedure` à l'aide d'un champ dérivé. Ce champ doit alors retourner le contenu du champ `symbols` du `scope` de la `Procedure`, si le `scope` n'est pas vide.

Les champs `next` et `previous` peuvent également être ajouté à `BasicBlock` sous la forme de champs dérivés. Ces champs doivent collecter toutes les destinations (`to`), respectivement toutes les sources (`from`), des arcs sortant d'un `BasicBlock` (`outEdges`), respectivement des arcs entrant d'un `BasicBlock` (`inEdges`).

Les champs `definitions` et `rightHandSides` peuvent être ajoutés à `Instruction` à l'aide de champs dérivés. Les définitions d'une variable dans `GeCoS IR` sont représentées par les SymbolInstruction trouvées dans le champ `dest` d'une affectation (`SetInstruction`). Les `RightHandSide` sont représentées par les Instruction trouvées dans le champ `source` d'une affectation (`SetInstruction`). Pour collecter ces instructions, il est possible d'écrire des opérations parcourant les `Instruction` (cf. Listing 9.17).

Il faut également être capable de modifier les collections d'instructions, pour insérer une `SSAUse` à la place d'une `Use`, ou une `SSADefinition` à la place d'une `Definition`. Ce remplacement ne peut pas se faire simplement à l'aide des opérations `add` et `remove` de la collection dérivée. En effet, l'ajout de la nouvelle `SSAUse` (respectivement `SSADefinition`) doit se faire à l'endroit exact où se trouvait l'ancienne `Use` (respectivement `Definition`). Pour cela, les opérations `replaceUse` et `replaceDefinition` sont redéfinies. Le Listing 9.18 présente la redéfinition de l'opération `replaceDef`. Celle-ci est redéfinie dans les différentes sous-classes d'`Instruction`. `replaceDef` parcourt l'arbre des instructions, jusqu'à trouver l'instruction contenant la `Definition` à remplacer (`oldDef`) et insère dans cette instruction la `SSADefinition` (`newDef`). L'opération `replaceUse` suit le même processus pour insérer au bon endroit la `SSAUse`.

Le champ `previousDefinition` peut être ajouté à `PhiInstruction` au moyen d'un champ dérivé. Ce champ dérivé doit collecter les `SSADefSymbol` référencées par les `SSAUseSymbol` qui forment les `children` d'une `PhiInstruction`.

Enfin, pour obtenir une adaptation complète entre `GeCoS IR` et `SSA_Form`, il est nécessaire d'ajouter un champ `dominanceFrontier` à `BasicBlock`. L'ajout du champ `dominanceFrontier` peut se faire simplement à l'aide d'un aspect. En effet, le contenu de ce champ est calculé au cours de du passage en forme SSA. Il n'a donc pas à être dérivé à partir des autres champs de `BasicBlock`.

9.5.2.4 Exemple : adaptation de `Procedure` à `ControlFlowGraph`

Le Listing 9.19 présente l'ensemble du code nécessaire pour que `GeCoS IR` possède une classe correspondante à la classe `ControlFlowGraph` de `SSA_Form`. Cet exemple présente à la fois le code utilisé pour renommer une classe (`Procedure` devenant `ControlFlowGraph`), pour renommer des champs (`start` devenant `entry` et `basicBlocks` devenant `nodes`) et pour réaliser des adaptations plus complexes (le champ dérivé `symbols`).

Pour commencer, une classe `ControlFlowGraph` est ajoutée au package `gecos::core` (ligne 3). Cette classe possède un champ référençant un objet `Procedure` (`procedure_adaptee`) à partir duquel tous les champs de `ControlFlowGraph` sont calculés (ligne 5). Les lignes 56 à 70 donnent également la définition de la classe `Node` qui adapte `BasicBlock` et les aspects déclarés sur les classes `Procedure` et `BasicBlock` afin de déclarer les champs adapter et adaptee de chacune.

La classe `ControlFlowGraph` déclare trois champs dérivés : `entry`, ligne 7, `nodes`, ligne 16, et `symbols`, ligne 34. Ces champs dérivés sont calculés à partir des champs `start` et `basicBlocks` de `Procedure` pour les deux premiers et à partir du champ `scope` de `Procedure` et du champ `symbols` de la classe `Scope` pour le troisième.

9.6 Conclusion

Nous avons présenté dans ce chapitre une application du système de types présenté au Chapitre 8 à un cas d'utilisation provenant de la compilation optimisante. Le passage en forme Static Single Assignment (SSA) est une passe de transformation couramment utilisée dans les compilateurs optimisants afin de faciliter des passes d'analyse et d'optimisation ultérieures. Nous avons présenté plusieurs types de modèles sur lesquels nous définissons les différentes analyses et transformations nécessaires au passage en forme SSA. Ces transformations sont réutilisables sur des modèles issus de différents langages de modélisation.

Nous avons présenté un exemple d'une telle réutilisation sur les modèles issus d'un langage de modélisation dédié simple, permettant d'exprimer des programmes manipulant des valeurs

Listing 9.17 – Ajout d'un champ dérivé `definitions` à la classe `Instruction` de `GeCoS IR`.

```
aspect class Instruction {
  //Adaptation to Instruction.definitions
  property definitions : oset Definition[0..*]
  _get is do
    result := getAllDefinitionChildren()
  end
  ...
  //Operation collecting all the variable definitions in an Instruction
  operation getAllDefinitionChildren() : OrderedSet[Definition] is abstract
}

aspect class ComplexInstruction {
  method getAllDefinitionChildren() : OrderedSet[Definition] is do
    result := OrdredSet[Definition].new
    self.children.each{c|
      result.addAll(c.getAllDefinitionChildren())
    }
  end
}

aspect class SetIntruction {
  //Overriding of Instruction.getAllDefinitionChildren
  method getAllDefinitionChildren() : OrderedSet[Definition] is do
    result := OrdredSet[Definition].new
    result.addAll(self.dest.getAllDefinitionChildren())
  end
}

aspect class SymbolInstruction {
  //Reference to the Definition object adapting this object
  reference definition_adapter : Definition

  //Overriding of Instruction.getAllDefinitionChildren
  method getAllDefinitionChildren() : OrderedSet[Definition] is do
    result := OrdredSet[Definition].new
    result.add(self.definition_adapter)
  end
}
```

Listing 9.18 – Redéfinition de l'opération `replaceDef` dans `GeCoS IR`.

```
aspect class ComplexInstruction {
  method replaceDef(oldDef : Definition, newDef : SSADefinition) is do
    //If this instruction contains the definition to replace, replace it
    if (self.children.contains(oldDef)) then
      var index : Integer init self.children.getIndexOf(oldDef)
      self.children.addAt(index, newDef)
      self.children.remove(oldDef)
    //Otherwise, pass to children
    else
      self.children.each{c|
        c.replaceDef(oldDef, newDef)
      }
    end
  end
}

aspect class SetIntruction {
  method replaceDef(oldDef : Definition, newDef : SSADefinition) is do
    //If this instruction contains the definition to replace, replace it
    if (self.dest.equals(oldDef)) then
      self.dest := newDef
    //Otherwise, pass to child
    else
      self.dest.replaceDef(oldDef, newDef)
    end
  end
}
```

Listing 9.19 – Exemple d'adaptation de la classe `Procedure` de `GeCoS IR` vers la classe `ControlFlowGraph` de `SSA_Form` en Kermeta.

```
package gecos::core {
  //Procedure2ControlFlowGraph adaptation
  class ControlFlowGraph {
    //Reference to the Procedure object for which this object provides an adaptation
    reference procedure_adaptee : Procedure
    //Adaptation to ControlFlowGraph.entry
    property entry : Node
    _get is do
      result := start.node_adapter
    end
    _set is do
      start := _value.basicBlock_adaptee
    end

    //Adaptation to ControlFlowGraph.nodes
    property nodes : set Node[0..*]
    //Returns Node objects from the original collection of BasicBlock objects
    _get is do
      result := Set[Node].new
      self.procedure_adaptee.basicBlocks.each{bb|
        result.add(bb.node_adapter)
      }
    end
    //Adds to the original collection a BasicBlock object from the Node object _element
    _add is do
      self.procedure_adaptee.basicBlocks.add(_element.basicBlock_adaptee)
    end
    //Removes from the original collection a BasicBlock object from the Node object _element
    _remove is do
      self.procedure_adaptee.basicBlocks.remove(_element.basicBlock_adaptee)
    end

    //Adaptation to ControlFlowGraph.symbols
    property symbols : set Symbol[0..*]
    //Returns the set of symbols contained by the scope of the Procedure object, if any
    _get is do
      if (not procedure_adaptee.scope.isVoid()) then
        result := procedure_adaptee.scope.symbols
      else
        result := Set[Symbol].new
      end
    end
    //Adds a Symbol object to the scope of the Procedure, initializing it if necessary
    _add is do
      if (procedure_adaptee.scope.isVoid()) then
        procedure_adaptee.scope := Scope.new
      end
      procedure_adaptee.scope.add(_element)
    end
    //Removes a Symbol object to the scope of the Procedure, if any
    _remove is do
      if (not procedure_adaptee.scope.isVoid()) then
        procedure_adaptee.scope.remove(_element)
      end
    end
  }
  aspect class Procedure {
    //Reference to the ControlFlowGraph object adapting this object
    reference controlFlowGraph_adapter : ControlFlowGraph
  }

  //BasicBlock2Node adaptation
  class Node {
    //Reference to the BasicBlock object for which this object provides an adaptation
     reference basicblock_adaptee : BasicBlock
     ...
  }
  aspect class BasicBlock {
    //Reference to the Node object adapting this object
    reference node_adapter : Node
  }
}
```

et variables entières. Grâce à la relation de sous-typage totale isomorphique établie entre le type de modèles de ce langage et le type de modèles `SSA_Form` sur lequel est défini le passage en forme SSA, il est possible de réutiliser les différentes transformations nécessaires au passage en forme SSA (calcul de la frontière de dominance, insertion de ϕ-fonction et renommage de variables) sur des modèles instances de différents métamodèles.

Enfin, nous avons présenté comment il serait possible d'établir une relation de sous-typage totale non-isomorphique entre notre type de modèles `SSA_Form` et le type de modèles extrait de `GeCoS IR`, le langage de modélisation dédié du compilateur GeCoS. La relation de sous-typage totale non-isomorphique s'appuie sur une adaptation bidirectionnelle, construite à l'aide d'ajouts de classes et de champs dérivés à la syntaxe abstraite de `GeCoS IR`. Ces classes et champs dérivés permettent aux modèles de `GeCoS IR` d'exposer l'interface attendue par notre passage en forme SSA, sans altérer la structure originale de `GeCoS IR`.

Le système de types orienté-modèle implémenté au sein de Kermeta permet donc de réutiliser des transformations de modèles entre différents métamodèles, si les types de modèles extraits de ces métamodèles présentent une relation de sous-typage totale isomorphique. Il est possible d'étendre ce système de types afin de prendre en compte des relations de sous-typage plus complexes, notamment des relations non-isomorphique, comme celle entre `GeCoS IR` et `SSA_Form`. Un système de types orienté-modèle supportant de telles relations de sous-typage permettrait la réutilisation de transformations de modèles complexes (e.g., des passes de compilation), entre différents langages de modélisation dédiés, diminuant ainsi le coût de création et d'évolution de ces langages.

Quatrième partie

Conclusion et perspectives

Chapitre 10

Conclusion et perspectives

Ce chapitre présente un bilan des contributions de cette thèse (Section 10.1) et discute les perspectives ouvertes par ces contributions (Section 10.2).

10.1 Conclusion

La multiplication des préoccupations à prendre en compte lors de la conception d'un système logiciel complexe pousse les concepteurs à séparer ces préoccupations pour les traiter de manière indépendante. L'ingénierie dirigée par les modèles propose de séparer ces préoccupations au sein de modèles, chaque modèle étant exprimé dans un langage de modélisation dédié à une préoccupation donnée. La spécification d'un langage de modélisation dédié, son métamodèle, est constituée d'une structure définissant les concepts du langage sous la forme d'un graphe de classes, et de comportement, implémenté sous la forme de transformations de modèles. La méthodologie prônée par l'ingénierie dirigée par les langages dédiés pousse à la multiplication de langages de modélisations dédiés.

Cette multiplication amène un besoin de facilités d'ingénierie pour la définition et l'outillage des langages de modélisation dédiés. En effet, la création d'un langage de modélisation dédié est une tâche longue et coûteuse. Cependant, la relation de conformité, relation centrale à l'ingénierie dirigée par les modèles, est un frein à l'établissement de telles facilités, et particulièrement à la réutilisation de transformations de modèles. En effet, la relation de conformité ne lie un modèle qu'à un unique langage de modélisation dédié, à travers les relations d'instanciation liant les éléments du modèle aux classes du métamodèle du langage. La relation de conformité empêche donc toute forme de polymorphisme au niveau des modèles.

Les systèmes de types supportent, dans les langages de programmation, un large éventail de facilités. Inspirés par les systèmes de types orientés-objet et par les premiers travaux sur le typage de modèles, nous proposons dans cette thèse une notion de type de modèles et une relation de typage liant un modèle à ses différents types.

Les types de modèles et la relation de typage s'appuient sur la séparation de l'interface et de l'implémentation des langages de modélisation dédiés, afin d'éviter les contraintes imposées par la relation de conformité. De plus, les types de modèles et la relation de typage prennent également en compte la séparation de la modélisation in-the-small (objets, types objets, classes, etc.) de la modélisation in-the-large (modèles, types de modèles, métamodèles, etc.). Ainsi, les types de modèles permettent d'encapsuler la structure et le comportement d'un modèle et de manipuler les modèles comme des entités de première classe, notamment au travers des champs de modèles et des opérations de modèles.

Nous définissons également des relations pouvant être établies entre deux langages de modélisation pour faciliter la réutilisation : quatre relations de sous-typage entre types de modèles qui autorisent la réutilisation par polymorphisme de sous-type et une relation d'héritage entre métamodèles. Il est ainsi possible de réutiliser les transformations de modèles sur des modèles issus de différents langages de modélisation dédiés si ils partagent le même type. Il est également possible de réutiliser la structure et le comportement d'un langage de modélisation dédié pour en créer un nouveau.

Les types de modèles et les relations entre langages de modélisation dédiés définissent une famille de systèmes de types orientés-modèle. Ces systèmes de types sont à même de supporter au niveau des langages de modélisation dédiés les facilités offertes par les systèmes de types orientés-objet au niveau des classes et des types objets.

Nous avons implémenté l'un de ces systèmes de types orientés-modèle au sein de l'environnement Kermeta. Ce système de types fournit un vérificateur pour la relation de sous-typage totale isomorphique, à laquelle peuvent être ramenées les trois autres relations de sous-typage (partielle isomorphique, totale non-isomorphique et partielle non-isomorphique). Nous fournissons également une relation d'héritage entre métamodèles permettant de réutiliser la structure et le comportement d'un langage de modélisation dédié pour en créer un nouveau.

10.2 Perspectives

Nos travaux sur le typage de modèles offrent un cadre formel pour la définition de facilités d'ingénierie pour les langages de modélisation dédiés. Nous pensons que ce cadre peut être étendu, non seulement au sein de l'ingénierie dirigée par les modèles, mais également dans le cadre plus large de l'ingénierie des langages logiciels. De plus, une évaluation expérimentale des facilités offertes par le typage de modèles permettrait d'évaluer les contextes dans lesquels l'utilisateur gagnerait à utiliser ces facilités, plutôt que des approches existantes.

10.2.1 "Fonctions de modèles"

Nous proposons dans cette thèse de réifier les transformations de modèles sous la forme d'opérations de modèles, i.e., de transformations attachées aux modèles. Cependant, toutes les transformations de modèles ne trouvent pas forcément leur place au sein d'un unique langage de modélisation dédié. C'est, entre autres, le cas des transformations de modèles dites symétriques, i.e., les transformations qui attribuent une importance égale aux différents modèles qu'elles traitent. Par exemple, une transformation vérifiant la cohérence de deux modèles $m1$ et $m2$ l'un par rapport à l'autre n'a pas vocation à être appelée depuis un modèle plus que l'autre. Pourquoi $m1.isConsistentWith(m2)$ aurait plus de sens que $m2.isConsistentWith(m2)$? Il est probable que dans certains cas, l'utilisateur ne veuille pas attacher une transformation à un langage de modélisation dédié donné, mais par exemple écrire $checkConsistency(m1, m2)$.

Pour de tels cas, il peut être intéressant de considérer des transformations de modèles externes aux langages de modélisation dédiés : des "fonctions de modèles", i.e., des transformations de modèles qui soient des entités de première classe. Il existe déjà, au niveau des langages de programmation objet, des langages tels que Scala, qui mêlent les paradigmes objets et fonctionnels afin d'apporter aux programmeurs les facilités de chaque paradigme. En s'appuyant sur les travaux de Vignaga et al. [VJBB13], qui typent les transformations de modèles comme des fonctions en utilisant les métamodèles pour typer leurs paramètres, il serait possible de fournir des "fonctions de modèles" bénéficiant du polymorphisme de sous-type que nous proposons.

10.2.2 Opérateurs d'adaptation

Les relations de sous-typage non-isomorphiques peuvent être ramenées à une relation de sous-typage isomorphique à l'aide d'une adaptation de modèles, qui doit être bidirectionnelle afin d'assurer la substituabilité des modèles dans le cas général. Il existe différents moyens de réaliser une telle adaptation.

Il est possible de définir une transformation de modèles entre le sous-type et le super-type, et inversement. Cependant, définir une adaptation bidirectionnelle à l'aide d'une paire de transformations de modèles peut se révéler une tâche complexe. En effet, il est alors nécessaire de garantir que deux transformations définies indépendamment assurent la bidirection.

Il est également possible d'ajouter à un type de modèles des classes et des champs dérivés pour créer un isomorphisme entre le sous-type et le super-type. La définition de ces adaptations à la main, par exemple à l'aide d'aspects, peut être longue et fastidieuse et donc source d'erreurs.

C'est pourquoi nous présentons ici deux pistes de recherche pour faciliter la définition d'adaptations bidirectionnelles entre types de modèles.

10.2.2.1 Langages d'adaptation bidirectionnelle de modèles

Plutôt que de laisser à l'utilisateur la charge de la vérification de la bidirectionnalité, des langages dédiés aux adaptations bidirectionnelles pourraient fournir des opérateurs assurant la bidirectionnalité.

Par exemple, Pierce et al. ont défini des langages d'adaptation bien typés, i.e., des langages dédiés qui définissent un ensemble d'opérateurs bidirectionnels permettant de passer d'une structure de données à une autre [BPV06, FGM+07, BFP+08, FPP08, BCF+10]. Un programme dans un tel langage décrit une conversion bidirectionnelle et est appelé lense. Pierce et al. ont proposé plusieurs langages de lenses entre différentes structures de données à base d'arbres (listes, chaînes de caractère, bases de données relationnelles) et prouvé que les opérateurs de ces langages étaient bien typés et combinables.

Hidaka et al. se sont inspiré des travaux sur les lenses pour proposer un langage de transformation bidirectionnel pour les graphes [HHI+10]. Étendre cette approche aux types de modèles permettrait d'offrir un langage d'adaptation bidirectionnelle pour les relations de sous-typage non-isomorphiques. Les lenses écrites dans ce langage seraient par construction bien typées et bidirectionnelles, évitant à l'utilisateur la vérification de transformations de modèles complexes. En utilisant un mécanisme proche des conversions implicites de Scala pour appeler ces lenses (i.e., les appeler de manière implicite quand un modèle typé par le sous-type est substitué à un modèle typé par le super-type) il serait possible de garantir la substituabilité sûre des modèles du sous-type aux modèles du super-type.

10.2.2.2 Inférence d'adaptation

Quelque soit la manière employée pour définir les adaptations bidirectionnelles de modèles (transformations de modèles ou ajout de champs dérivés), il est possible de fournir aux utilisateurs des mécanismes inférant l'adaptation nécessaire entre deux types de modèles. Un tel mécanisme d'inférence peut être utilisé de deux manières : comme un outil de suggestion pour aider l'utilisateur dans la définition d'une adaptation ; ou dans le cadre de relations de sous-typage déclarées implicitement.

Pour inférer une adaptation de modèles, il est possible d'utiliser des algorithmes de comparaison de graphes s'appuyant sur la structure du graphe de types objets des types de modèles. Ces algorithmes permettent de calculer des alignements pondérés entre deux types de modèles [FHLN08]. Les poids de ces alignements peuvent être calculés en fonction de motifs

d'adaptation connus, ou d'opérateurs d'un langage d'adaptation, afin de sélectionner l'alignement nécessitant l'adaptation la moins complexe ou de suggérer à l'utilisateur différentes adaptations.

10.2.3 Application du typage de modèles à d'autres espaces technologiques

Les facilités d'ingénierie pour la définition et l'outillage des langages de modélisation dédiés offertes par le typage de modèles se placent dans le contexte plus large de l'ingénierie des langages logiciels. Il serait donc intéressant de développer des approches équivalentes dans d'autres espaces technologiques afin de faciliter la définition et l'outillage de langages logiciels.

Ainsi, le polymorphisme de mograms, que ceux-ci soient issus de langages de modélisation dédiés ou non, permettrait de réutiliser les outils définis sur un langage pour les mograms d'un autre. Pour cela, il est possible d'établir des relations de sous-typage entre schémas XSD (qui définissent la structure des documents XML) ou entre structures d'arbre de syntaxe abstraite, en s'appuyant sur les types de données que ces structures définissent. Afin d'établir des manipulations réutilisables entre deux structures (schémas XSD ou structures d'arbres de syntaxes abstraites), il faudrait également développer des langages de manipulation, ou étendre les langages existants, afin d'autoriser l'expression de types chemins-dépendants.

10.2.4 Extension des types de modèles

Les métamodèles et les types de modèles tels que nous les définissons ne prennent en compte la structure des langages de modélisation dédiés que sous la forme d'un graphe de types objets et leur comportement sous la forme d'un ensemble de signatures d'opérations et de champs de modèles. Cependant, le nombre d'erreurs que peut détecter un système de types dépend de la précision des informations contenues par les types (cf. Section 4.2). Ainsi, il est possible de considérer d'autres informations que le graphe de classes, les signatures des opérations de modèles et les champs de modèles dans les types de modèles et dans les relations de typage de modèles.

10.2.4.1 Invariants, pre et post-conditions

L'ingénierie dirigée par les modèles s'appuie sur des contrats pour définir des contraintes sur la structure d'un langage de modélisation dédié (i.e., des invariants) ou sur le comportement des opérations et des transformations de modèles (i.e., des pre et post-conditions). Ces contraintes peuvent par exemple être exprimées à l'aide du langage OCL [OMG03]. Sun et al. ont présenté une approche permettant de prendre en compte de telles contraintes définies au sein des types de modèles et de la relation de sous-typage totale isomorphique, en prenant en compte la variance des contraintes (cf. Section 4.2.1.1) [SCDF13]. Porter la comparaison de contraintes proposée par Sun et al. dans nos autres relations de sous-typage permettrait de fournir une substituabilité plus sûre. Pour cela, il faudrait prendre en compte les effets des fonctions d'extraction de type de modèles effectif et d'adaptation sur les contraintes (suppression, réécriture, etc.).

10.2.4.2 Concurrence

Récemment, Combemale et al. ont proposé de réifier au sein de la spécification des langages de modélisation dédiés la gestion de la concurrence [CDVL+]. Pour cela la concurrence est exprimée au sein d'un modèle de calcul explicite. Le modèle de calcul définit l'ordonnanceur en

terme d'évènements qui provoquent l'évolution des modèles. La prise en compte des modèles de calcul dans les types de modèles et les relations de sous-typage permettrait d'assurer la substituabilité comportementale de modèles.

10.2.5 Évaluation expérimentale

Dans les perspectives à court terme, il serait également intéressant d'évaluer les facilités apportées par le typage de modèles de manière expérimentale. Les facilités telles que la réutilisation, l'encapsulation et l'abstraction sont généralement admises dans les langages de programmation comme des bonnes pratiques dans la conception logicielle. Cependant, évaluer les bénéfices apportés par ces facilités et les difficultés qu'elles peuvent engendrer (prise en main, définition manuelle d'adaptations entre types de modèles) permettrait de déterminer dans quels contextes (tailles et types de projets, possibilité de réutilisation, etc.) il est intéressant d'utiliser le typage de modèles.

Une telle évaluation pourrait se faire en comparant l'utilisation du typage de modèles à celle des outils de l'état de l'art (e.g., les concepts de de Lara et Guerra [DLG10]) et à celle de langages de programmation proposant des systèmes de types avancés (e.g., Scala).

Bibliographie

[AC96] Martin Abadi and Luca Cardelli. A Theory of Objects. Springer, 1st edition, 1996.

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. FAMILIAR : A Domain-Specific Language for Large Scale Management of Feature Models. Science of Computer Programming (SCP) Special issue on programming languages, 78(6) :657–681, 2013.

[AEM12] Vincent Aranega, Anne Etien, and Sebastien Mosser. Using feature model to build model transformation chains. In Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems (MoDELS '12), pages 562–578. Springer, 2012.

[AK02] Colin Atkinson and Thomas Kühne. Profiles in a strict metamodeling framework. Science of Computer Programming, 44(1) :5–22, 2002.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers : Principles, Techniques, and Tools (2nd Edition). Addison Wesley, 2006.

[AP03] Andrew W. Appel and Jens Palsberg. Modern Compiler Implementation in Java. Cambridge University Press, 2nd edition, 2003.

[BBK+07] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom : Piggybacking rewriting on java. In Proceedings of the Term Rewriting and Applications, 18th International Conference (RTA '07), volume 4533 of Lecture Notes in Computer Science, pages 36–47, 2007.

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

[BCF+10] Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses : alignment and view update. In Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10), pages 193–204. ACM, 2010.

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. Software : Practice and Experience, 28(8) :859–881, 1998.

[BCMP12] Jean-Christophe Bach, Xavier Crégut, Pierre-Etienne Moreau, and Marc Pantel. Model transformations with tom. In Proceedings of the International Workshop on Language Descriptions, Tools, and Applications, (LDTA '12), page 4. ACM, 2012.

[BDFB07] Mikaël Barbero, Marcos Didonet, Del Fabro, and Jean Bézivin. Traceability and provenance issues in global model management. In Proceedings of the 3rd ECMDA-FA Traceability Workshop, 2007.

[Béz06] Jean Bézivin. Model driven engineering : an emerging technical space. In Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering (GTTSE'05), pages 36–64. Springer, 2006.

[BFP+08] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang : resourceful lenses for string data. In Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08), pages 407–419. ACM, 2008.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In Proceedings of the 16th IEEE international conference on Automated software engineering (ASE '01), pages 273–280. IEEE Computer Society, 2001.

[BGMR03] J. Bézivin, S. Gérard, P. A. Muller, and L. Rioux. MDA components : Challenges and Opportunities. In Proceedings of the 1st International Workshop on Metamodelling for MDA, pages 23–41, 2003.

[BJRV05] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the large and modeling in the small. In Proceedings of the 2003 European Conference on Model Driven Architecture : Foundations and Applications (ECMDA-FA'03), pages 33–46. Springer, 2005.

[BJT05] J. Bezivin, F. Jouault, and D. Touzet. Principles, standards and tools for model engineering. In Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on, pages 28–29, 2005.

[BPV06] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses : a language for updatable views. In Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '06), pages 338–347. ACM, 2006.

[BSvGF03] Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. Polytoil : A type-safe polymorphic object-oriented language. ACM Transactions on Programming Languages Systems (TOPLAS), 25(2) :225–290, 2003.

[BV99] Kim B. Bruce and Joseph Vanderwaart. Semantics-driven language design : Statically type-safe virtual types in object-oriented languages. Electronic Notes in Theoretical Computer Science, 20 :50–75, 1999.

[CDVL+] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. Will be presented at the 6th International Conference on Software Language Engineering (SLE 2013), and published in Springer's Lecture Notes in Computer Science series.

[CFH+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations : A Cross-Discipline Perspective. In Proceedings of the 2nd International Conference on Model Transformations (ICMT '09), 2009.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. IBM Systems Journal, 45(3) :621–645, 2006.

[CHC90] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance Is Not Subtyping. In Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles Of Programming Languages (POPL '90), pages 125–135. ACM, 1990.

[Cho56] Noam Chomsky. Three models for the description of language. IRE Transactions on Information Theory, 2 :113–124, 1956.

[CIK90] Dominique Clement, Janet Incerpi, and Gilles Kahn. CENTAUR : towards a software tool box for programming environments. In Fred Long, editor, Proceedings of the International Workshop on Software Engineering Environments (SEE '90), number 467 in Lecture Notes in Computer Science, pages 287–304. Springer, 1990.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava : modular open classes and symmetric multiple dispatch for java. In Proceedings of the 15th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00), pages 130–145. ACM, 2000.

[CMR09] Horatiu Cirstea, Pierre-Etienne Moreau, and Antoine Reilles. TomML : A Rule Language for Structured Data. In Proceedings of the International Symposium on Rule Interchange and Applications, (RuleML '09), volume 5858 of Lecture Notes in Computer Science, pages 262–271. Springer, 2009.

[CMTG07] Arnaud Cuccuru, Chokri Mraidha, François Terrier, and Sébastien Gérard. Templatable metamodels for semantic variation points. In Proceedings of the 3rd European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA '07), pages 68–82, 2007.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In Proceedings of the 3rd annual ACM Symposium on Theory Of Computing (STOC '71), pages 151–158. ACM, 1971.

[Dah68] Ole-Johan Dahl. SIMULA 67 common base language, (Norwegian Computing Center. Publication). 1968.

[DDMvL97] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. Grail/kaos : an environment for goal-driven requirements engineering. In Proceedings of the 19th International Conference on Software engineering (ICSE '97), pages 612–613. ACM, 1997.

[DJAO04] Sophia Drossopoulou, Paul Jolly, Christopher Anderson, and Klaus Ostermann. Simple Dependent Types : Concord. In ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP 2004), 2004.

[DK75] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. SIGPLAN Notices, 10(6) :114–121, 1975.

[DLG10] Juan De Lara and Esther Guerra. Generic meta-modelling with concepts, templates and mixin layers. In Proceedings of the 13th international conference on Model Driven Engineering Languages and Systems (MoDELS '10), pages 16–30, 2010.

[DMC92] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages : from a new taxonomy to constructive proposals and their validation. In Proceedings of the 7th annual conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92), pages 201–217. ACM, 1992.

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits : A mechanism for fine-grained reuse. ACM Transactions on Programming Languages Systems (TOPLAS), 28(2) :331–388, 2006.

[EABP12] Anne Etien, Vincent Aranega, Xavier Blanc, and Richard F. Paige. Chaining model transformations. In Proceedings of the 1st Workshop on the Analysis of Model Transformations (AMT '12), pages 9–14. ACM, 2012.

[ER10] Marina Egea and Vlad Rusu. Formal executable semantics for conformance in the MDE framework. Innovations in Software and Systems Engineering (ISSE), 6(1-2) :73–81, 2010.

[Ern01] Erik Ernst. Family polymorphism. In Proceedins of the 15th European Conference on Object-Oriented Programming (ECOOP '01), volume 2072 of Lecture Notes in Computer Science, pages 303–326. Springer, 2001.

[Fav04a] Jean-Marie Favre. Foundations of Meta-Pyramids : Languages vs. Metamodels - Episode II : Story of Thotus the Baboon. Dagstuhl Reports, 2004.

[Fav04b] Jean-Marie Favre. Foundations of Model (Driven) (Reverse) Engineering : Models - Episode I : Stories of The Fidus Papyrus and of The Solarus. Dagstuhl Reports, 2004.

[FGLW09] Jean-Marie Favre, Dragan Gasević, Ralf Lammel, and Andreas Winter. Guest Editors' Introduction to the Special Section on Software Language Engineering. IEEE Transactions on Software Engineering, 35(6) :737–741, 2009.

[FGM+07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations : A linguistic approach to the view-update problem. ACM Transactions on Programming Languages Systems (TOPLAS), 29(3), 2007.

[FHLN08] Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel matching for automatic model transformation generation. In Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS '08), Lecture Notes in Computer Science, pages 326–340. Springer, 2008.

[FPP08] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In Proceeding of the 13th ACM SIGPLAN international conference on Functional programming (ICFP '08), pages 383–396. ACM, 2008.

[FYG+11] Antoine Floch, Tomofumi Yuki, Clément Guy, Steven Derrien, Benoit Combemale, Sanjay Rajopadhye, and Robert France. Model-Driven Engineering and Optimizing Compilers : A bridge too far ? In Jon Whittle, Tony Clark, and Thomas Kühne, editors, Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS '11), number 6981 in Lecture Notes in Computer Science, pages 608–622. Springer, 2011.

[GdLKP10] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, and Richard F. Paige. Intermodelling : from theory to practice. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems : Part I (MoDELS '10), number 6394 in Lecture Notes in Computer Science, pages 376–391. Springer, 2010.

[GKH07] Dragan Gasević, Nima Kaviani, and Marek Hatala. On Metamodeling in Megamodels. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS '07), volume 4735 of Lecture Notes in Computer Science, pages 91–105. Springer, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Monticore : a framework for the development of textual domain specific languages. In Companion to the proceedings of the 30th International Conference on Software Engineering (ICSE '08), pages 925–926. ACM, 2008.

[HHI+10] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In Proceeding of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10), pages 205–216. ACM, 2010.

[HJK+09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA '09), pages 114–129. Springer, 2009.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10) :576–580, 1969.

[HSG10] Regina Hebig, Andreas Seibel, and Holger Giese. On the Unification of Megamodels. In Proceedings of the 4th International Workshop on Multi Paradigm Modeling at the MODELS 2010 Conference (MPM '10), volume 42 of Electronic Communications of the EASST, pages 1–13. EASST, 2010.

[HSST11] Zhenjiang Hu, Andy Schürr, Perdita Stevens, and James F. Terwilliger. Bidirectional transformations "bx". Dagstuhl Reports, 2011.

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11), pages 471–480. ACM, 2011.

[ISO11a] ISO/IEC. ISO/IEC 9899 : 2011 Information technology-Programming languages-C. International Organization for Standardization, Geneva, Switzerland, 27 :59, 2011.

[ISO11b] ISO/IEC JTC 1. ISO/IEC FDIS 9075-1 Information technology - Database languages - SQL - Part 1 : Framework (SQL/Framework), 2011.

[ISV05] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. In Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS '05), volume 3780 of Lecture Notes in Computer Science, pages 161–177. Springer, 2005.

[JCB+13] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and Francois Fouquet. Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. Journal of Software and Systems Modeling (SoSyM), 2013. to appear.

[JCG04] Adam Jaworski, Nikolas Coupland, and Dariusz Galasiński. Metalanguage. Walter de Gruyter, 2004.

[JCV12] Jean-Marc Jézéquel, Benoit Combemale, and Didier Vojtisek. Ingénierie Dirigée par les Modèles : des concepts à la pratique. Références sciences. Ellipses, 2012.

[JGB11] Cédric Jeanneret, Martin Glinz, and Benoit Baudry. Estimating footprints of model operations. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11), pages 601–610. ACM, 2011.

[KBA02] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological Spaces : An Initial Appraisal. In International Symposium on Distributed Objects and Applications (DOA '02), 2002.

[KKR+08] Gerti Kappel, Horst Kargl, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, Michael Strommer, and Manuel Wimmer. A framework for building mapping operators resolving structural heterogeneities. In Information Systems and e-Business Technologies, volume 5 of Lecture Notes in Business Information Processing, pages 158–174. Springer, 2008.

[Kle06] Anneke Kleppe. MCC : a model transformation environment. In Proceedings of the 2nd European Conference on Model Driven Architecture : Foundations and Applications (ECMDA-FA'06), pages 173–187. Springer, 2006.

[Kle08] Anneke Kleppe. Software Language Engineering : Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Professional, 2008.

[KLM+83] G. Kahn, B. Lang, B. Melese, E. Morcos, and Communicated G. Berry. Metal : A formalism to specify formalisms. Science of Computer Programming, 3(2) :151–188, 1983.

[KSW+13] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. Reuse in Model-to-Model Transformation Languages : Are we there yet ? Journal of Software and Systems Modeling (SoSyM), 2013. available online.

[KV10] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '10), pages 444–463. ACM, 2010.

[Küh13] Thomas Kühne. On model compatibility with referees and contexts. Journal of Software and Systems Modeling (SoSyM), 12(3) :475–488, 2013.

[LDA13] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-oriented language families : a case study. In Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13), pages 1–8. ACM, 2013.

[Len95] Douglas B. Lenat. CYC : a large-scale investment in knowledge infrastructure. Communications of the ACM, 38(11) :33–38, 1995.

[LG13] Juan Lara and Esther Guerra. From types to type requirements : genericity for model-driven engineering. Journal of Software and Systems Modeling, 12(3) :453–474, 2013.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(6) :1811–1841, 1994.

[Mar05] Fowler Martin. Language workbenches : The killer-app for domain specific languages ? http://martinfowler.com/articles/languageWorkbench.html, 2005. [Online ; accessed 20-Septembre-2013].

[MC07] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse : a review of industrial studies. Empirical Software Engineering, 12(5) :471–516, 2007.

[MFBC12] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoit Combemale. Modeling Modeling Modeling. Journal of Software and Systems Modeling (SoSyM), 11(3) :347–359, 2012.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS '05), volume 3713 of Lecture Notes in Computer Science, pages 264–278, 2005.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. ACM Computing Surveys, 37(4) :316–344, 2005.

[Mit90] Richard Mitchell. Managing complexity in software engineering. Peregrinus on behalf of the Institution of Electrical Engineers, 1990.

[MMBJ09] Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. Generic model refactorings. In Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS '09), pages 628–643. Springer, 2009.

[MVG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science, 152 :125–142, 2006.

[Old05] Jon Oldevik. Transformation composition modelling framework. In Proceedings of the 5th IFIP WG 6.1 international conference on Distributed Applications and Interoperable Systems (DAIS '05), pages 108–114. Springer, 2005.

[OMG03] OMG. UML Object Constraint Language (OCL) 2.0 Specification, 2003.

[OMG06] OMG. Meta Object Facility (MOF) 2.0 Core Specification, 2006.

[OMG08] OMG. Software & Systems Process Engineering Meta-Model (OMG SPEM) Specification, 2008.

[OMG11] OMG. Unified Modeling Language (OMG UML), Infrastructure, 2011.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. In Proceedings of the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05), pages 41–57. ACM, 2005.

[Pie02] Benjamin C. Pierce. Types and programming languages. MIT Press, 2002.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical report, University of Aarhus, 1981.

[PQ95] T. J. Parr and R. W. Quong. Antlr : a predicated-ll(k) parser generator. Software : Practice and Experience, 25(7) :789–810, 1995.

[PVSGB08] Jens Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach, and Yolande Berbers. Constructing and visualizing transformation chains. In Proceedings of the 4th European conference on Model Driven Architecture : Foundations and Applications (ECMDA-FA '08), pages 17–32. Springer, 2008.

[RKPP10] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model migration with epsilon flock. In Proceedings of the 3rd International Conference on Model Transformations (ICMT '10), pages 184–198. Springer, 2010.

[RRGLR+09] José E. Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. Orchestrating ATL model transformations. In Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL '09), pages 34–46, 2009.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. EMF : Eclipse Modeling Framework. Addison-Wesley, 2 edition, 2009.

[SCDF13] Wuliang Sun, Benoît Combemale, Steven Derrien, and Robert B. France. Using Model Types to Support Contract-Aware Model Substitutability. In Proceedings of the 9th European Conference on Modelling Foundations and Applications, ECMFA '13, volume 7949 of Lecture Notes in Computer Science, pages 118–133. Springer, 2013.

[SCGdL11] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Generic model transformations : Write once, reuse everywhere. In Proceedings of the 4th International Conference on Model Transformations (ICMT '11), pages 62–77, 2011.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams : A survey and a formal semantics. In Proceedings of the 14th IEEE International Requirements Engineering Conference (RE '06), pages 136–145. IEEE Computer Society, 2006.

[SJ07] Jim Steel and Jean M. Jézéquel. On Model Typing. Software and Systems Modeling, 6(4) :401–413, 2007.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model Transformation : The Heart and Soul of Model-Driven Software Development. IEEE Software, 20(5) :42–45, 2003.

[SME09] Rick Salay, John Mylopoulos, and Steve Easterbrook. Using macromodels to manage collections of related models. In Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE '09), pages 141–155. Springer, 2009.

[Smi94] Randall B. Smith. Prototype-based languages (panel) : object lessons from class-free programming. In Proceedings of the 9th annual conference on Object-Oriented Programming Systems, Language, and Applications (OOPSLA '94), pages 102–112. ACM, 1994.

[SMM+12] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry, and Jean-Marc Jézéquel. Reusable model transformations. Journal of Software and Systems Modeling (SoSyM), 11(1) :111–125, 2012.

[SNG10] Andreas Seibel, Stefan Neumann, and Holger Giese. Dynamic hierarchical mega models : comprehensive traceability and its efficient maintenance. Journal of Software and Systems Modeling (SoSyM), 9(4) :493–528, 2010.

[Ste07] Jim Steel. Typage de Modèles. PhD thesis, Université Rennes 1, 2007.

[TR03] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+ : defining and using domain-specific modeling languages and code generators. In Companion of the 18th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03), pages 92–93. ACM, 2003.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages : an annotated bibliography. SIGPLAN Notices, 35(6) :26–36, 2000.

[VJBB13] Andrés Vignaga, Frédéric Jouault, MaríaCecilia Bastarrica, and Hugo Brunelière. Typing artifacts in megamodeling. Journal of Software and Systems Modeling (SoSyM), 12(1) :105–119, 2013.

[Voe11] Markus Voelter. Language and IDE Modularization and Composition with MPS. In Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering IV (GTTSE '11), pages 383–430, 2011.

[VP04] Dániel Varró and András Pataricza. Generic and meta-transformations for model transformation engineering. In Proceedings of the 7th International Conference on the Unified Modeling Language (UML '04), pages 290–304, 2004.

[W3Ca] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/xml/. [Online ; accessed 17-October-2013].

[W3Cb] W3C. XML Schema Part 0 : Primer Second Edition. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/. [Online ; accessed 17-October-2013].

[Wik13] Wikipedia. Extended Backus-Naur Form — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form, 2013. [Online ; accessed 14-July-2013].

[WKR+11] Manuel Wimmer, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Jesús Cuadrado, Esther Guerra, and Juan de Lara. Reusing model transformations across heterogeneous metamodels. In Proceedings of the 5th International Workshop on Multi-Paradigm Modeling (MPM '11), 2011.

[Yu97] Eric S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE '97), pages 226–235. IEEE Computer Society, 1997.

Liste des figures

2.1 Graphe de flot de contrôle du programme présenté en Listing 2.1. 15
2.2 Spécification d'une syntaxe de graphe utilisant l'*Extended Backus-Naur Form* (EBNF) et exemple d'utilisation de cette syntaxe. 19
2.3 Méta-outils générant des outils pour les spécifications écrites dans le métalangage sur lequel ils sont définis et exemple du métalangage EBNF et du méta-outil Antlr. 21

3.1 Concepts au cœur d'EMOF sous la forme d'un diagramme de classes 30
3.2 Syntaxe abstraite d'un langage de graphe de flot de contrôle définie à l'aide d'EMOF, sous la forme d'un diagramme de classes et d'un diagramme d'objets. 31
3.3 Facilités pour la définition et l'outillage de langages de modélisation dédiés. . . 43

4.1 Groupe de types objets composé de `ColoredNode` et de `Color`. 55
4.2 Groupe de types objets composé de `RGBColoredNode` et de `RGBColor`. 55
4.3 Deux types de modèles `MTGraph` et `MTColoredGraph` en Kermeta tels que `MTColoredGraph` < :`MTGraph`. 61
4.4 Transformations de modèles réutilisables par *polymorphisme de sous-type*. 64
4.5 Deux classes `InfiniteNode` et `FiniteNode` tels que `FiniteNode` <# `InfiniteNode` d'après la définition 4.4. 65
4.6 Deux classes `RGBColoredNode` et `ColoredNode` telles que `ColoredNode` <# `RGBColoredNode` d'après la définition 4.6. 66

5.1 Concepts et relations pris en compte dans METAL pour la définition de systèmes de types orientés-modèle. 73
5.2 Syntaxe abstraite du métalangage METAL. 75

6.1 Concepts et relations de METAL abordés dans ce chapitre. 78
6.2 Sous ensemble de la syntaxe abstraite du métalangage METAL exposant les concepts communs avec le MOF et Ecore. 79
6.3 Sous-ensemble de la syntaxe abstraite de METAL pour les *types objets*. 81
6.4 Sous-ensemble de la syntaxe abstraite de METAL pour les *classes*. 82
6.5 Exemple de *type exact*. 82
6.6 Sous-ensemble de la syntaxe abstraite de METAL pour les *types de modèles*. . . . 84
6.7 Syntaxe abstraite de METAL pour les *métamodèles*. 85
6.8 Type de modèles $MT_{ColoredGraph}$ (à gauche) et métamodèle $MM_{ColoredGraph}$ implémentant $MT_{ColoredGraph}$ (à droite). 86

7.1 Relations de METAL abordées dans ce chapitre. 90
7.2 Groupe de types objets *ColoredNode* et *Color*. 91
7.3 Groupe de types objets *ColoredNode'*, *Color'* et *RGBEncoding'*. 91

7.4 Type de modèles $MT_{ColoredGraph}$. 95
7.5 Type de modèles $MT_{RGBColoredGraph}$. 95
7.6 Type de modèles $CFG_{IntBool}$ de graphes de flot de contrôle contenant des instructions entières et booléennes. 101
7.7 Type de modèles effectif de l'analyse d'atteignabilité CFG_{Reach} extrait de $CFG_{IntBool}$. 101
7.8 Type de modèles CFG_{Float} de graphes de flot de contrôle contenant des instructions flottantes. 101
7.9 Type de modèles CFG_{Type} de graphes de flot de contrôle représentant les arcs à l'aide d'un type. 102
7.10 Type de modèles CFG_{Asso} de graphes de flot de contrôle représentant les arcs à l'aide d'une association. 102
7.11 Type de modèles CFG_{Adapt} résultat de l'ajout de champs dérivés sur CFG_{Type}. . 103
7.12 Sous ensemble de la syntaxe abstraite du métalangage METAL exposant les concepts nécessaires aux relations de sous-typage entre types de modèles. . . . 107
7.13 Sous ensemble de la syntaxe abstraite du métalangage METAL exposant les concepts nécessaires aux relations d'héritage entre métamodèles. 111
7.14 *Feature model* de la famille de systèmes de types orientés-modèle. 114

8.1 Flot de compilation de l'environnement de développement de langages de modélisation dédiés Kermeta. 121
8.2 Fichier `simple.ecore` déclarant la syntaxe abstraite d'un langage de graphes de flot de contrôles. 122
8.3 Flot de compilation de Kermeta étendu pour intégrer le typage de modèles. . . 123
8.4 Extrait de la syntaxe abstraite du langage Kermeta. 125
8.5 Type de modèles `CFG_Simple`. 126
8.6 Type de modèles CFG_Instructions tel que CFG_Instructions < : CFG_Simple. . 136

9.1 Exemple simple de conversion en forme SSA 140
9.2 Exemple de conversion en forme SSA d'un programme contenant des branchements 140
9.3 Type de modèles `CFG_Dominance` pour le calcul la frontière de dominance. . . . 142
9.4 Type de modèles `SSA_Form` pour le passage en forme SSA. 144
9.5 Type de modèles `CFG_Int` contenant des instructions entières. 147
9.6 Extrait de la syntaxe abstraite de `GeCoS IR`, le langage de graphes de flot de contrôle de GeCoS. 149

Liste des listings

2.1 Programme calculant et affichant la puissance de x par y. 15
2.2 Spécification métacirculaire de l'Extended Backus-Naur Form (EBNF) [Wik13] . 20
4.1 Pseudo-code décrivant la classe `Edge` permettant d'instancier des arcs 47
4.2 Pseudo-code définissant un type objet exposant une interface de manipulation d'arcs 49
4.3 Pseudo-code définissant un type objet exposant une interface de manipulation d'arcs pondérés 49
4.4 Pseudo-code définissant la classe `WeightedEdge` étendant la classe `Edge` et permettant d'instancier des arcs pondérés 52
4.5 Pseudo-code définissant une classe générique utilisant un paramètre de type (`P`). 53
4.6 Pseudo-code définissant une classe générique utilisant une variable de type (`V`). 53
4.7 Pseudo-code déclarant un groupe de types objets composé de `ColoredNode` et de `Color`. 55
4.8 Pseudo-code déclarant un groupe de types objets composé de `RGBColoredNode` et de `RGBColor`. 55
4.9 Exemple de groupe de types utilisant les paramètres de type en Java. 56
4.10 Exemple de groupe de types utilisant des types bruts en Java. 56
4.11 Utilisation d'un objet pour définir les types chemin-dépendants `group.ColoredNode` et `group.Color`. 57
4.12 Utilisation d'un paramètre de type pour définir les types chemin-dépendants `G.ColoredNode` et `G.Color`. 57
4.13 Réécriture de l'exemple du Listing 4.7 à l'aide de la spécialisation de groupes de classes. 57
4.14 Réécriture de l'exemple du Listing 4.8 à l'aide de la spécialisation de groupes de classes. 57
4.15 Réécriture de l'exemple du Listing 4.7 à l'aide de types internes. 58
4.16 Réécriture de l'exemple du Listing 4.8 à l'aide de types internes. 58
4.17 Classe paramétrée par un type de modèles et réutilisable sur tous les sous-types du type de modèles `MTGraph`. 62
6.1 Invariants OCL limitant le type des champs, opérations et paramètres de types objets. 81
6.2 Invariant OCL forçant une classe à n'implémenter que des types objets. 82
6.3 Invariant OCL forçant un métamodèle à n'implémenter que des types de modèles. 85
7.1 Aspect Kermeta déclarant les champs dérivés `next` et `previous` calculés à partir de l'ensemble des arcs d'un nœud. 103
7.2 Aspect Kermeta déclarant un champ dérivé `next` permettant une adaptation bidirectionnelle de CFG_{Type} vers CFG_{Asso}. 105
8.1 Fichier `ControlFlowGraphAspect.kmt` déclarant un invariant OCL dans la classe `ControlFlowGraph`. 122

8.2 Fichier `NodeAspect.kmt` déclarant une variable et une opération dans la classe `Node`. . . . 122
8.3 Exemple de fichier kp pour la création d'un langage de modélisation dédié. . . 122
8.4 Exemple de déclaration de métamodèle à l'aide du langage kp. . . . 126
8.5 Exemple de variable contenant un modèle en Kermeta. . . . 126
8.6 Exemple de types chemin-dépendants en Kermeta. . . . 127
8.7 Exemple de déclaration de champ et d'opération de modèles en Kermeta. . . . 128
8.8 Exemple d'appel de champ et d'opération de modèles en Kermeta. . . . 128
8.9 Exemple de déclaration d'héritage entre CFG_Simple et CFG_Dominance en kp 128
8.10 Exemple de déclaration d'une relation de sous-typage entre CFG_Simple et CFG_Dominance en Kermeta. . . . 128
8.11 Exemple de variables de types du type de modèles `CFG_Simple` en Scala. . . . 131
8.12 Exemple de variables de types d'un métamodèle en Scala. . . . 131
8.13 Exemple de trait Scala généré pour le type objet `ControlFlowGraph`. . . . 132
8.14 Exemple de trait Scala généré pour la classe `ControlFlowGraph`. . . . 133
8.15 Exemple de classe Scala générée pour la classe `ControlFlowGraph`. . . . 133
8.16 Exemple de trait Scala généré pour le type de modèles `CFG_Simple`. . . . 134
8.17 Exemple de trait Scala généré pour la Factory du type de modèles `CFG_Simple`. 134
8.18 Exemple de trait Scala généré pour le métamodèle `CFG_Simple`. . . . 135
8.19 Exemple de trait et d'objet Scala générés pour la Factory du métamodèle `CFG_Simple`. 135
8.20 Exemple de classe Scala générée pour le métamodèle `CFG_Simple`. . . . 135
8.21 Exemple de trait Scala généré pour le type de modèles `CFG_Instructions`. . . . 136
8.22 Exemple de trait Scala généré pour le type objet `ControlFlowGraph` du type de modèles `CFG_Instructions`. . . . 137
9.1 Programme original en pseudo-code 140
9.2 Forme Static Single Assignement correspondante 140
9.3 Programme original en pseudo-code. . . . 140
9.4 Forme SSA incomplète correspondant au programme du Listing 9.3. . . . 140
9.5 Forme SSA complète correspondant au programme du Listing 9.3. . . . 140
9.6 Déclaration du métamodèle `CFG_Dominance` implémentant le type de modèles de la Figure 9.3 en kp. . . . 142
9.7 Calcul de la frontière de dominance définie sur le type de modèles `CFG_Dominance` en Kermeta. . . . 142
9.8 Insertion de ϕ-fonction définie sur le type de modèles `SSA_Form` en Kermeta. . 145
9.9 Passage en forme SSA défini sur le type de modèles `SSA_Form` en Kermeta. . . . 146
9.10 Redéfinition de l'opération `replaceUse` dans `CFG_Int`. . . . 146
9.11 Déclaration du métamodèle `CFG_Int` implémentant le type de modèles de la Figure 9.5 en kp. . . . 147
9.12 Déclaration de la relation de sous-typage entre le type de modèles `CFG_Int` et le type de modèles `SSA_Form` en Kermeta. . . . 147
9.13 Réutilisation du passage en forme SSA sur un modèle typé par `CFG_Int`. . . . 147
9.14 Exemple de renommage de classes par ajout de la classe `Node` dans `GeCoS IR` en Kermeta. . . . 151
9.15 Exemple de renommage de champ à l'aide d'un champ dérivé en Kermeta. . . . 152
9.16 Exemple de renommage de champ à l'aide d'une collection dérivée en Kermeta. 153
9.17 Ajout d'un champ dérivé `definitions` à la classe `Instruction` de `GeCoS IR`. . 155
9.18 Redéfinition de l'opération `replaceDef` dans `GeCoS IR`. . . . 155
9.19 Exemple d'adaptation de la classe `Procedure` de `GeCoS IR` vers la classe `ControlFlowGraph` de `SSA_Form` en Kermeta. . . . 156

Liste des tables

3.1 Définitions de *métamodèle* dans la littérature 28
3.2 Définitions de la *relation de conformité* dans la littérature 34

6.1 Classes du MOF, d'Ecore et de METAL représentant les interfaces et les implémentations des objets et des opérations. 80
6.2 Équivalence entre les concepts de modélisation *in-the-small* et les concepts de modélisation *in-the-large* définis dans METAL. 88

7.1 Redéfinitions autorisées par la relation d'héritage entre métamodèles avec sous-typage. 108
7.2 Fonctions permettant de ramener les quatre relations de sous-typage entre types de modèles à la relation de sous-typage *totale isomorphique* 112

9.1 Équivalences entre les classes de `SSA_Form` et les classes de `GeCoS IR`. 150

Liste des publications

1. Jean-Marc Jézéquel, Benoît Combemale, Steven Derrien, Clément Guy and Sanjay V. Rajopadhye. Bridging the chasm between MDE and the world of compilation. Journal of Software and System Modeling (SoSyM), 11(4) :581-597, 2012.
2. Clément Guy, Benoît Combemale, Steven Derrien, Jim Steel and Jean-Marc Jézéquel. On Model Subtyping. In Proceedings of the 8th European Conference on Modelling Foundations and Applications, ECMFA '12, pages 400-415, 2012.
3. Antoine Floch, Tomofumi Yuki, Clément Guy, Steven Derrien, Benoît Combemale, Sanjay V. Rajopadhye and Robert B. France. Model-Driven Engineering and Optimizing Compilers : A Bridge Too Far ?. In Proceedings of the 14th International Conference Model Driven Engineering Languages and Systems, MoDELS '11, pages 608-622, 2011.
4. Clément Guy, Steven Derrien, Benoît Combemale and Jean-Marc Jézéquel. Vers un rapprochement de l'IDM et de la compilation. In Proceedings of the Journées sur l'Ingénierie Dirigée par les Modèles, IDM '11, pages 91-96, 2011.

Résumé

Le nombre et la complexité toujours croissants des préoccupations prises en compte dans les systèmes logiciels complexes (e.g., sécurité, IHM, scalabilité, préoccupations du domaine d'application) poussent les concepteurs de tels systèmes à séparer ces préoccupations afin de les traiter de manière indépendante.

L'ingénierie dirigée par les modèles (IDM) prône la séparation des préoccupations au sein de langages de modélisation dédiés. Les langages de modélisation dédiés permettent de capitaliser le savoir et le savoir-faire associés à une préoccupation au travers des constructions du langage et des outils associés. Cependant la définition et l'outillage d'un langage dédié demandent un effort de développement important pour un public par définition réduit.

Nous proposons dans cette thèse une relation liant les modèles et une interface de modèle permettant de faciliter la mise en place de facilités de typage pour la définition et l'outillage d'un langage dédié. Cette interface expose les éléments de modèle et les transformations de modèles associés à un langage de modélisation dédié. Nous représentons une telle interface par un type de modèles supportant des relations de sous-typage et d'héritage. Dans ce but nous définissons :

- une relation de typage entre les modèles et les langages de modélisation dédiés permettant de considérer les modèles comme des entités de première classe ;
- des relations de sous-typage entre langages de modélisation dédiés permettant la réutilisation de la syntaxe abstraite et des transformations de modèles.

Abstract

The ever growing number and complexity of concerns in software intensive systems (e.g., safety, HMI, scalability, business domain concerns, etc.) leads designers of such systems to separate these concerns to deal with them independently.

Model-Driven Engineering (MDE) advocates the separation of concerns in Domain-Specific Modeling Languages (DSMLs). DSMLs are used to capitalize the knowledge and know-how associated with a concern through the language constructs and its associated tools. However, both definition and tooling of a DSML require a significant development effort for a limited audience.

In this thesis, we propose a relationship between models and model interfaces in order to ease the design of typing facilities for the definition and tooling of a DSML. This interface exposes the model elements and model transformations associated with a DSML. We represent such an interface by a model type supporting subtyping and inheritance relationships. For this purpose we define :

- a typing relationship between models and DSMLs allowing to consider models as first-class entities ;
- subtyping relationships between DSMLs enabling the reuse of abstract syntax and model transformations.