Ingénierie Dirigée par les Modèles : des concepts à la pratique...

Jean-Marc Jézéquel, Benoit Combemale, Didier Vojtisek

Ingénierie Dirigée par les Modèles
des concepts à la pratique...

Jean-Marc Jézéquel, Benoît Combemale et Didier Vojtisek

Université de Rennes 1 (ISTIC – ESIR)
{jean-marc.jezequel, benoit.combemale}@irisa.fr

INRIA Rennes Bretagne Atlantique
didier.vojtisek@inria.fr

Table des matières

Introduction

De la programmation à la modélisation

Dans de nombreux domaines, l'industrie du logiciel fait face à une évolution exponentielle de la taille et de la complexité des systèmes logiciels. Le logiciel représente donc aujourd'hui un enjeu majeur pour la valeur ajoutée qu'il amène dans ces domaines. Pour en maîtriser les coûts et la qualité, cela a conduit à adopter des approches fondées sur l'utilisation de composants développés indépendamment les uns des autres. Cependant les composants logiciels interagissent entre eux et avec le monde réel de manière de plus en plus complexe, impliquant de nombreuses dimensions extra-fonctionnelles (fiabilité, sécurité, ponctualité, consommation d'énergie, etc.). Comme dans les autres sciences, on a donc de plus en plus recours à la modélisation pour essayer de maîtriser cette complexité. La modélisation, dans le sens le plus large, est en effet l'utilisation efficace d'une représentation simplifiée d'un aspect du monde réel pour un objectif donné. Par exemple, le modèle des différents composants d'un système informatique, de leurs interrelations et leurs interactions est appelé architecture logicielle. Cette notion d'architecture logicielle est née à la fin des années 1960 de l'invention de la programmation structurée. Un programme informatique était alors conceptualisé comme une suite d'étapes (flot de contrôle) représentée par les premiers diagrammes d'architecture, les organigrammes.

Aujourd'hui, un système complexe est modélisé selon de nombreux points de vue (structurel, dynamique, fonctionnel, etc.) qui peuvent eux-mêmes varier en termes d'abstraction (idées ou concepts, interfaces, composants abstraits, composants logiciels physiques) et de précision (ébauche, solution à améliorer ou solution finale).

La modularité des développements peut alors être abordée par la séparation explicite des préoccupations et par l'utilisation pour chacune d'entre elles d'un langage dédié (*Domain-Specific Modeling Language*) offrant des concepts spécifiques ayant le meilleur niveau d'abstraction pour le développeur.

De la modélisation à l'ingénierie dirigée par les modèles

Un des enjeux de l'ingénierie dirigée par les modèles (IDM) est de capitaliser les savoir-faire de conception et de validation de tels systèmes en capturant et en réutilisant les connexions entre ces points de vue, que ce soit de manière horizontale, par tissage de liens entre modèles, ou verticale, par transformations de modèle. Même s'il existe une longue expérience de l'utilisation de l'ingénierie dirigée par les modèles dans certains domaines comme les télécommunications, sa généralisation à l'ensemble de l'industrie n'en est qu'à ses débuts. Visant à automatiser une partie du processus de développement, l'IDM requiert un effort d'abstraction plus important de la part des développeurs. En contrepartie, l'IDM promet de conserver le savoir faire de conception proche des centres de décision grâce aux économies d'échelle dues à l'automatisation. Une étude

récente de l'utilisation effective de l'IDM dans l'industrie la qualifie même d'arme suprême contre la pratique controversée de l'offshore pour les tâches répétitives de codage d'application [HWRK11].

À qui est destiné ce livre ?

Ce livre est destiné à toute personne souhaitant acquérir les connaissances théoriques et pratiques pour la mise en œuvre d'un développement dirigé par les modèles. Étudiants en génie logiciel et ingénieurs logiciels pourront y trouver les informations nécessaires à la compréhension des outils et à leurs mises en œuvre dans des projets complexes de développement de systèmes informatiques. Ce livre permettra également aux responsables de projets informatiques d'acquérir les connaissances nécessaires pour mettre en œuvre un projet de développement dirigé par les modèles. Tout en réutilisant les compétences des équipes actuelles, il sera capable de mettre en œuvre les changements nécessaires à l'introduction des modèles dans le cycle de développement, et d'avoir un regard critique sur les gains apportés par de tels changements.

Des connaissances préalables en programmation (principalement orientée-objet), ainsi qu'en analyse et conception orientée-objet (UML, patron de conception, conception par contrat) simplifieront la lecture de ce livre mais ne sont pas indispensable.

Structure du livre et chemins de lecture

Ce livre est volontairement structuré en deux parties afin de permettre au lecteur de se concentrer sur l'une et/ou l'autre des parties en fonction de ses attentes. La première partie introduit l'ensemble des concepts qu'il est utile de maitriser pour comprendre les motivations, les enjeux, et la mise en oeuvre d'un développement dirigé par les modèles. La deuxième partie présente pas à pas et de manière très pragmatique la mise en œuvre de chaque étape du processus d'un développement dirigé par les modèles.

Si nous conseillons à l'ensemble des lecteurs de parcourir le chapitre 1, le praticien pourra ensuite s'attacher à lire la deuxième partie afin de mettre en œuvre un développement de logiciel dirigée par les modèles, et revenir ponctuellement à la première partie pour approfondir les concepts utilisés. Les lecteurs souhaitant comprendre l'évolution des techniques de développement du logiciel et acquérir les fondements des pratiques les plus modernes pourront s'attacher à lire la première partie. Enfin, les étudiants en génie logiciel devront être capables de lire et de comprendre l'ensemble du livre.

Première partie

Les concepts

Ceux qui sont férus de pratique sans posséder la science sont comme le pilote qui s'embarquerait sans timon ni boussole, et ne saurait jamais avec certitude où il va.

Léonard de Vinci.

Chapitre 1

Les modèles au cœur du développement de logiciel

> « Pour un observateur A, M est un modèle de l'objet O, si M aide A à répondre aux questions qu'il se pose sur O » [Min68].

1.1 Principes généraux

Après l'approche objet des années 80, l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM). Comme décrit dans [Béz04b, Béz05], cette nouvelle approche peut être considérée à la fois en *continuité* et en *rupture* avec les précédents travaux. Tout d'abord en continuité car c'est la technologie objet qui a déclenché l'évolution vers les modèles. En effet, une fois acquise la conception des systèmes informatiques sous la forme d'objets communicant entre eux, il s'est posé la question de les classifier en fonction de leurs différentes origines (objets métiers, techniques, etc.). L'IDM vise donc, de manière plus radicale que pouvaient l'être les approches des *patterns* [GHJ95] et des *aspects* [KLM+97], à fournir un grand nombre de modèles pour exprimer séparément chacune des préoccupations des utilisateurs, des concepteurs, des architectes, etc. C'est par ce principe de base fondamentalement différent que l'IDM peut être considérée en rupture par rapport aux travaux de l'approche objet.

1.1.1 Modèle

Alors que l'approche objet est fondée sur les relations essentielles d'*instanciation* et d'*héritage*, l'IDM est basée sur un autre jeu de concepts et de relations. Le concept central de l'IDM est la notion de *modèle*, pour laquelle il n'existe pas à ce jour de définition universelle. Néanmoins, de nombreux travaux s'accordent à un relatif consensus d'une certaine compréhension. A partir des travaux de l'OMG [1], de Muller *et al.* [MFBC10], de Bézivin *et al.* [BG01] et de Seidewitz [Sei03], nous considérerons la définition suivante d'un modèle.

Définition (*Modèle*) 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.

1. *The Object Management Group (OMG)*, cf. http://www.omg.org/.

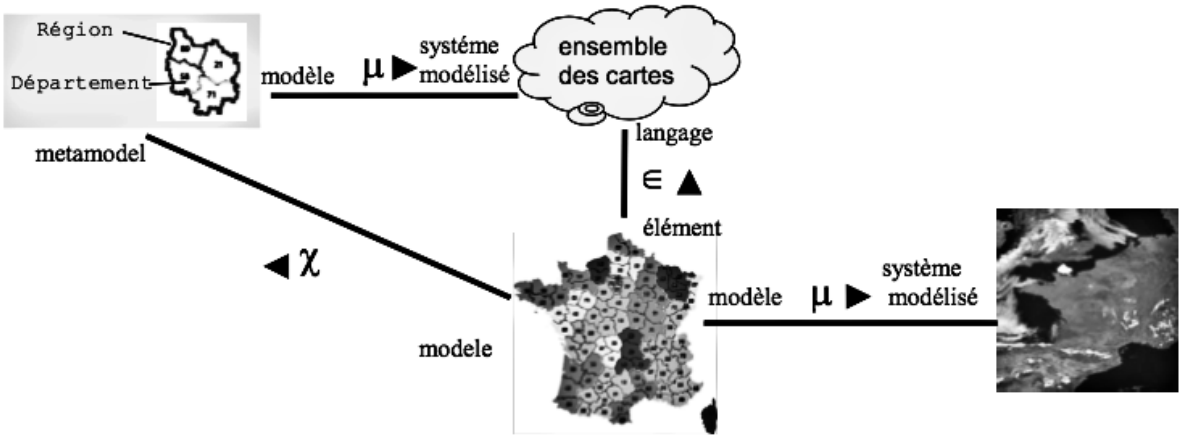


FIGURE 1.1 – Relations entre système, modèle, métamodèle et langage (extrait de [FEB06])

« En cet empire, l´Art de la Cartographie fut poussé à une telle Perfection que la Carte d'une seule Province occupait toute une ville et la Carte de l'Empire toute une Province. Avec le temps, ces Cartes Démesurées cessèrent de donner satisfaction et les Collèges de Cartographes levèrent une Carte de l'Empire, qui avait le Format de l'Empire et qui coïncidait avec lui, point par point. Moins passionnées pour l'Etude de la Cartographie, les Générations Suivantes réfléchirent que cette Carte Dilatée était inutile et, non sans impiété, elle l´abandonnèrent à l´Inclémence du Soleil et des Hivers. Dans les Déserts de l'Ouest, subsistent des Ruines très abimées de la Carte. Des Animaux et des Mendiants les habitent. Dans tout le Pays, il n'y a plus d´autre trace des Disciplines Géographiques. »

FIGURE 1.2 – Citation attribuée par Jorge Luis Borges à un auteur de son invention et publiée en français dans le recueil « L'auteur et autres textes »

On déduit de cette définition la première relation majeure de l'IDM, entre le modèle et le système qu'il représente, appelée *représentationDe* dans [AK03, Béz04a, Sei03], et nommée μ sur la figure 1.1, où nous reprenons l'exemple utilisé dans [FEB06], qui s'appuie sur la cartographie pour illustrer l'IDM.

Dans cet exemple, une carte est un modèle (une représentation) de la réalité, avec une intention particulière (carte routière, administrative, des reliefs, etc.).

Notons que même si cette relation a fait l'objet de nombreuses réflexions, il reste toutefois difficile de répondre à la question « qu'est ce qu'un bon modèle ? » et donc de formaliser précisément la relation μ.

Sans tomber dans les excès de la carte blanche de Lewis Carroll (figure 1.3) ou de la carte à l'échelle 1 :1 de Borges (figure 1.2), toute la difficulté de l'activité de modélisation est de trouver le bon niveau d'abstraction pour représenter un système selon un ensemble de modèles facilitant sa conception et sa validation.

Un modèle doit donc par définition être une abstraction pertinente du système qu'il modélise pour un point de vue particulier, c'est-à-dire qu'il doit être suffisant et nécessaire pour répondre aux questions sous-jacentes à ce point de vue particulier, en lieu et place du système qu'il représente et exactement de la même façon que le système aurait répondu lui-même. Ce principe, dit de *substituabilité*, assure que le modèle peut se substituer au système pour permettre d'analyser de manière plus abstraite certaines de ses propriétés [Min68].

« He had bought a large map representing the sea,
without the least vestige of land :
and the crew were much pleased when they found it to be
a map they could all understand.

"What's the good of Mercator's North Poles and Equators,
Tropics, Zones, and Meridian Lines ?"
So the Bellman would cry : and the crew would reply
"They are merely conventional signs !"

Other maps are such shapes, with their islands and capes !
But we've got our brave Captain to thank :
(so the crew would protest) "that he's bought us the best–
a perfect and absolute blank !" »

FIGURE 1.3 – Citation de Carroll Lewis dans « The Hunting of the Snark »

Définition (*Principe de substituabilité*) Un modèle doit capturer les informations nécessaires et suffisantes pour permettre de répondre aux questions que l'on se pose sur un aspect du système qu'il représente, exactement de la même façon que le système lui-même aurait répondu.

1.1.2 Modèles, aspects et lignes de produits

La modélisation est parfois vue comme le moyen d'exprimer une solution à un plus haut niveau d'abstraction que le code. Cette vision étroite de la modélisation a eu son utilité dans le passé (langages d'assemblages abstrayant du code machine, langages de troisième génération abstrayant les langages d'assemblages, etc.) et est encore utile de nos jours, par exemple pour obtenir une vue holistique d'un gros programme C++. Mais la modélisation ne s'arrête pas à ça.

La modélisation est en effet la pierre angulaire de toute activité scientifique (en lien avec la validation des modèles obtenus vis-à-vis d'expérimentations effectuées dans le monde réel). À cet égard, la spécificité du domaine de l'ingénierie est que les ingénieurs construisent des modèles d'artefacts qui en général n'existent pas encore (ne serait-ce que parce que le but ultime est de construire ces artefacts).

En ingénierie on souhaite en général décomposer un système complexe en autant de modèles que nécessaire pour aborder efficacement toutes les préoccupations pertinentes [BC04]. Ces modèles peuvent être exprimés avec un langage de modélisation généraliste comme UML, ou avec des langages de modélisation spécifiques à des domaines (DSML en anglais pour *Domain Specific Modeling Language*) lorsque cela semble plus approprié (voir figure 1.4). Chacun de ces modèles peut être vu comme l'abstraction d'un aspect de la réalité pour gérer une préoccupation particulière. L'expression explicite de solutions efficaces pour gérer ces préoccupations permet d'établir des compromis adéquats relativement tôt dans le cycle de vie du logiciel mais aussi de gérer efficacement des variantes du système (notion de ligne de produits) [PBvdL05].

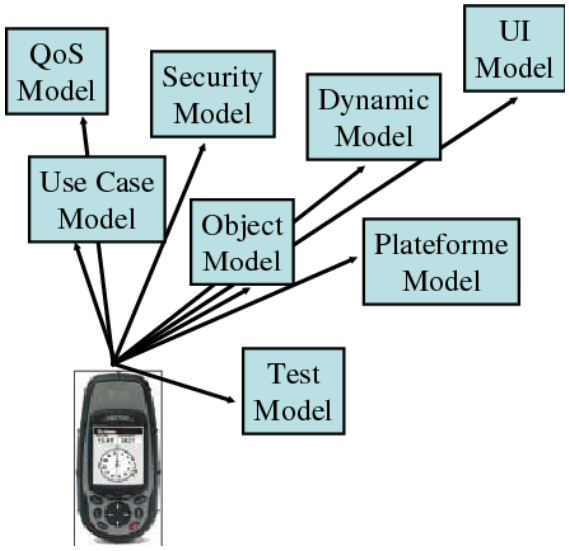


FIGURE 1.4 – Modéliser plusieurs aspects

Lignes de produits : il est de plus en plus rare de nos jour de construire des systèmes informatiques qui ne soient pas déclinés en de multiples variantes, et ceci au delà des habituelles différences de fonctionnalités dues à des choix marketing. En effet il faut aussi tenir compte de différences dans le matériel (cartes graphiques, dispositifs d'affichage, variété de périphériques d'interaction etc.), dans les systèmes d'exploitations et autres intergiciels, des différences liées aux aspects législatifs, normatifs ou culturels, et bien sûr des préférences des utilisateurs. Ces points de variations sont souvent assez orthogonaux, ce qui engendre une explosion combinatoire du nombre de variantes possibles d'un système. De plus, comme chacune de ces variantes peut avoir des versions successives, une véritable vision bi-dimentionnelle (axe des variantes et axe temporel) est nécessaire pour appréhender dans sa globalité ce concept de ligne de produits

Notons que dans la communauté de la programmation orientée aspect, la notion d'aspects est définie de manière sensiblement plus restrictive comme la modularisation de préoccupations transverses [FF00]. Si nous avons en effet déjà un paradigme de décomposition (comme l'orientation objet), il existe toujours de nombreuses classes de préoccupations pour lesquels une encapsulation dans des modules n'est pas possible (d'où leur nom de préoccupations transverses) : par exemple certains types de fonctionnalités (comme l'authentification) sont par construction difficiles à encapsuler, ce qui est aussi le cas pour un certain nombre d'exigences non-fonctionnelles qui sont de manière inhérente transverses, comme la sécurité, la disponibilité, la répartition, la gestion des ressources, ou les contraintes temps-réels.

Maintenant que la notion d'aspect devient de plus en plus utilisée au-delà du monde de la programmation, il y a une acceptation grandissante pour une définition plus large où un aspect est simplement une préoccupation qui peut être modularisée. L'identification systématique de ces aspects, leur modularisation et leur composition ont ainsi pour objectif général d'améliorer notre capacité à raisonner sur le domaine du problème et sur la solution correspondante. Cela doit permettre de réduire la taille des modèles de logiciels et du code des applications, et donc les coûts de développement et le temps de maintenance.

En substance, la modélisation est une activité visant à séparer les préoccupations dans le domaine du problème, activité que l'on appelle souvent *analyse*. Si les solutions à ces préoccupations peuvent être décrites comme des aspects, le processus de conception peut alors être caractérisé comme le tissage (*weaving* en anglais) de ces aspects dans un modèle de conception détaillée (aussi appelé l'espace de la solution, voir figure 1.5) [JPW+02]. Ceci n'est bien sûr pas nouveau : c'est en réalité ce que les concepteurs de logiciels ont toujours fait. Le plus souvent cependant, la plupart des aspects ne sont pas *explicites*, ou quand ils le sont, c'est le plus souvent sous la forme d'une définition informelle. Ceci conduit le concepteur à effectuer ce tissage entre préoccupations en quelque sorte *de tête*. Et ensuite à produire la conception détaillée résultante comme un gros programme où les différentes préoccupations s'entremêlent (et ceci même si un paradigme de décomposition, comme l'orienté objet, est utilisé). Si cela fonctionne sur des projets de taille ou de complexité réduite, il est bien connu que ça devient extrêmement difficile à gérer lorsque la complexité des projets augmente.

Insistons sur le point que le défi abordé ici n'est pas *comment* concevoir un système pour prendre en compte un aspect particulier : il existe en effet dans l'industrie d'importants savoir-faire, souvent d'ailleurs capturés sous la forme de patrons de conception. Bien sûr, prendre en compte plusieurs aspects à la fois est un peu plus complexe, mais de nombreux projets complexes menés à bien dans l'industrie sont là pour nous montrer que les ingénieurs (la plupart du temps) y arrivent. Le vrai défi est plutôt lié, dans un contexte où l'agilité des développements prime, au besoin fréquent de s'adapter au changement des exigences. Ce changement se traduit par un nouveau choix de variante ou de version d'une variante par aspect du système, et nécessite alors d'être recomposé pour obtenir une nouvelle déclinaison (ou configuration) du système, c'est à dire un nouveau produit dans une ligne de produits. Bien sûr, cette nouvelle déclinaison du système doit être obtenue rapidement, de manière fiable, et à bon marché. De ce point de vue, faire la composition manuelle de chaque aspect n'est plus une panacée.

L'IDM ne propose pas de résoudre ce problème directement, mais simplement de mécaniser et de reproduire le processus que les concepteurs expérimentés suivent à la main [HJPP02]. L'idée est alors que lorsqu'une nouvelle variante a besoin d'être dérivée dans une ligne de produits, on peut automatiquement rejouer la plus grosse partie de ce processus de conception, en apportant juste quelques petites modifications [LMV+07].

Usuellement en sciences, un modèle a une nature différente de la réalité qu'il modélise (par exemple le dessin d'un pont vs. le pont lui-même). C'est seulement en logiciel et en linguistique qu'un modèle a la même nature que la réalité modélisée. En logiciel en particulier, ceci ouvre la possibilité de dériver automatiquement du logiciel depuis un modèle, c'est-à-dire de rendre les modèles "productifs" en automatisant ce processus de tissage. Cette automatisation n'est bien sûr possible que si les modèles ne sont plus informels. Cela implique que le processus de tissage soit lui-même décrit comme un programme manipulant ces modèles pour produire une conception détaillée [MFJ05a]. Cette dernière pourra être finalement transformée en code, en configuration de logiciel, en suite de tests, etc. [PJJ+07].

Cette notion de tissage de modèles est donc au cœur de la conception dirigée par les modèles.

1.1.3 Métamodèle

La notion de modèle dans l'IDM fait explicitement référence à la définition des langages utilisés pour les construire. En effet, pour qu'un modèle soit productif, il doit pouvoir être manipulé par une machine. Le langage dans lequel ce modèle est exprimé doit donc être explicitement défini. De manière naturelle, la définition d'un langage de modélisation a pris la forme d'un modèle, appelé *métamodèle*.

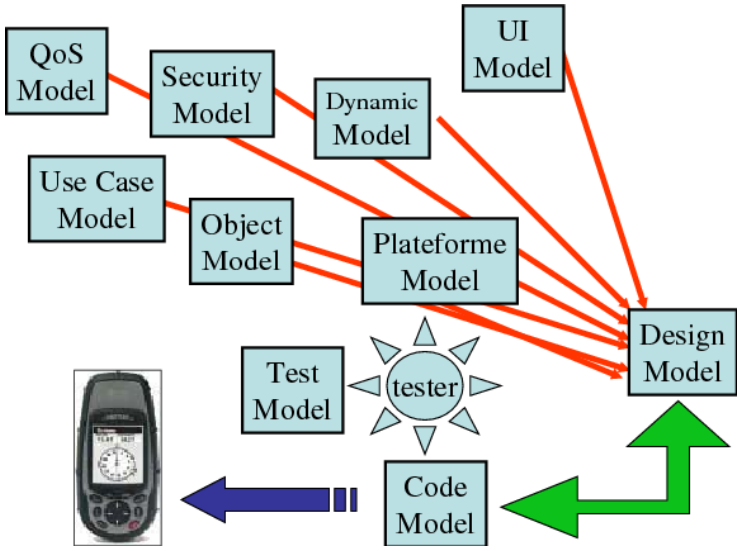


FIGURE 1.5 – Concevoir revient à tisser des modèles

Définition (*Métamodèle*) Un métamodèle est un modèle qui définit le langage d'expression d'un modèle [OMG06a], c'est-à-dire le langage de modélisation.

En pratique, un métamodèle permet de capitaliser un domaine de connaissances. Il devient naturellement le cœur d'un outillage visant à systématiser certaines étapes de développement.

La notion de métamodèle conduit à l'identification d'une seconde relation, liant le modèle et le langage utilisé pour le construire, appelée *conformeA* et nommée χ sur la figure 1.1.

Définition (*conformeA* (χ)) 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.

En cartographie (cf. figure 1.1), il est effectivement indispensable d'associer à chaque carte la description du « langage » utilisé pour la réaliser. Ceci se fait notamment sous la forme d'une légende. La carte doit, pour être utilisable, être conforme à cette légende. Plusieurs cartes peuvent être conformes à une même légende. La légende est alors considérée comme un modèle représentant cet ensemble de cartes (μ) et à laquelle chacune d'entre elles doit se conformer (χ).

Ces deux relations permettent ainsi de distinguer le langage qui joue le rôle de système, du (ou des) métamodèle(s) qui jouent le rôle de modèle(s) de ce langage.

C'est sur ces principes de base que s'appuie l'OMG pour définir l'ensemble de ses standards, en particulier UML (*Unified Modeling Language*) [OMG07a, OMG07b] dont le succès industriel est largement reconnu.

1.2 L'approche MDA

Le consensus sur UML fut décisif dans la transition vers des techniques de production basées sur les modèles. Après l'acceptation du concept clé de métamodèle comme langage de description de modèle, de nombreux métamodèles ont émergés afin d'apporter chacun leurs spécificités dans un domaine particulier (développement logiciel, entrepôt de données, procédé de développement, etc.). Devant le danger de voir émerger indépendamment et de manière incompatible cette grande variété de métamodèles, il y avait un besoin urgent de donner un cadre général pour leur description. La réponse logique fut donc d'offrir un langage de définition de métamodèles

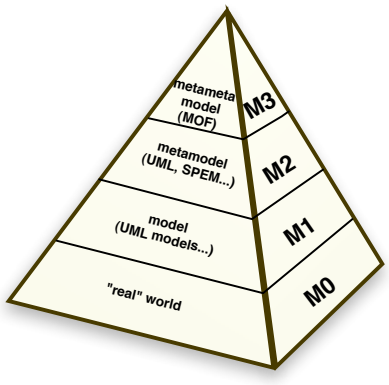


FIGURE 1.6 – Pyramide de modélisation de l'OMG

qui prit lui-même la forme d'un modèle : ce fut le *métamétamodèle* MOF (*Meta-Object Facility*) [OMG06a]. En tant que modèle, le métamétamodèle doit être défini à partir d'un langage de modélisation. Pour limiter le nombre de niveaux d'abstraction, le métamétamodèle doit alors avoir la propriété de *métacircularité*, c'est-à-dire la capacité de se décrire lui-même.

Définition (*Métamétamodèle*) Un métamétamodèle est un modèle qui décrit un langage de métamodélisation, c'est-à-dire les éléments de modélisation nécessaires à la définition des langages de modélisation. Il a de plus la capacité de se décrire lui-même.

C'est sur ces principes que se base l'organisation de la modélisation de l'OMG généralement décrite sous une forme pyramidale (cf. figure 1.6). Le monde réel est représenté à la base de la pyramide (niveau $M0$). Les modèles représentant cette réalité constituent le niveau $M1$. Les métamodèles permettant la définition de ces modèles (p. ex. UML) constituent le niveau $M2$. Enfin, le métamétamodèle, unique et métacirculaire, est représenté au sommet de la pyramide (niveau $M3$). Chaque niveau correspond à une utilisation particulière des modèles, revenant à capitaliser un domaine de connaissances utilisé au niveau inférieur.

L'approche consistant à considérer une hiérarchie de métamodèles n'est pas propre à l'OMG, ni même à l'IDM, puisqu'elle est utilisée depuis longtemps dans de nombreux domaines de l'informatique. Chaque hiérarchie définit un *espace technique* [KBA02, BJRV05, BK05]. Nous distinguons par exemple le *modelware* (espace technique des modèles), le *grammarware* (espace technique des grammaires définies par les langages tels que BNF [2] ou EBNF [3]), le *BDware* (espace technique des bases de données), etc.

Définition (*Espace technique*) Un espace technique est l'ensemble des outils et techniques issus d'une pyramide de métamodèles dont le sommet est occupé par une famille de (méta)métamodèles similaires [FEB06].

L'OMG a défini le MDA (*Model Driven Architecture*) en 2000 [Sol00] pour promulguer de bonnes pratiques de modélisation et exploiter pleinement les avantages des modèles. En 2003, les membres ont adopté la dernière version de la spécification [MM03] donnant une définition détaillée de l'architecture. Cette approche vise à mettre en valeur les qualités intrinsèques des

2. *Backus-Naur form*
3. *Extended BNF*

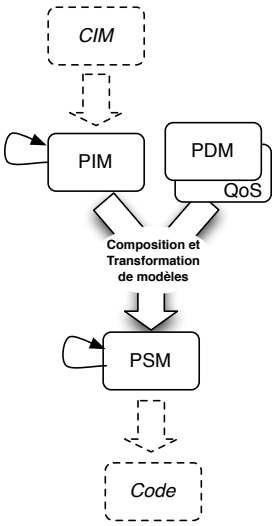


FIGURE 1.7 – MDA : Un processus en Y dirigé par les modèles

modèles, telles que la pérennité, la productivité et la prise en compte des plateformes d'exécution. Le MDA inclut pour cela la définition de plusieurs standards, notamment UML, MOF et XMI[4].

Le principe clé et initial du MDA consiste à s'appuyer sur le standard UML pour décrire séparément des modèles pour les différentes phases du cycle de développement d'une application. Plus précisément, le MDA préconise l'élaboration de modèles (cf. figure 1.7) :

— d'exigence (*Computation Independent Model* – CIM) dans lesquels aucune considération informatique n'apparaît,
— d'analyse et de conception (*Platform Independent Model* – PIM),
— de code (*Platform Specific Model* – PSM).

L'objectif majeur du MDA est l'élaboration de modèles pérennes (PIM), indépendants des détails techniques des plate-formes d'exécution (J2EE, .Net, PHP, etc.), afin de permettre la génération automatique de la totalité des modèles de code (PSM) et d'obtenir un gain significatif de productivité.

Le passage de PIM à PSM fait intervenir des mécanismes de composition et de transformation de modèle (cf. section 3.1) avec un modèle de description de la plateforme (*Platform Description Model* – PDM). Cette démarche s'organise donc selon un cycle de développement « en Y » propre au MDD (*Model Driven Development*) (cf. figure 1.7).

Le MDA a fait l'objet d'un grand intérêt dans la littérature spécialisée. Nous citons entre autre les ouvrages de X. Blanc [Bla05] et de A. Kleppe [KWB03] qui ont inspiré ce paragraphe.

1.3 Langages dédiés de modélisation

De la même façon que l'arrivée de la programmation par objets n'a pas invalidé les apports de la programmation structurée, le développement dirigé par les modèles ne contredit pas les apports de la technologie objet. Il est donc important de ne pas considérer ces solutions comme antagonistes mais comme complémentaires.

Toutefois un point de divergence entre ces deux approches concerne l'intégration de paradigmes. Initialement, la technologie objet se voulait aussi une technologie d'intégration

4. XMI, *XML Metadata Interchange*, est un format d'échange basé sur XML pour les modèles exprimés à partir d'un métamodèle MOF.

car il était théoriquement possible de représenter de façon uniforme les processus, les règles, les fonctions, etc. par des objets. Aujourd'hui, on revient à une vision moins hégémonique où les différents paradigmes de programmation coexistent sans donner plus d'importance à l'un ou à l'autre [Béz04b].

Un point important est alors de séparer clairement les approches IDM du formalisme UML, et de l'utilisation qui en est faite dans le MDA. En effet, non seulement la portée de l'IDM est plus large que celle d'UML mais la vision de l'IDM est aussi très différente de celle d'UML, parfois même en contradiction. UML est un standard assez monolithique obtenu par consensus *a maxima*, dont on doit réduire ou étendre la portée à l'aide de mécanismes comme les profils [OMG07b, §18]. Ces mécanismes n'ont pas tous la précision souhaitable et mènent parfois à des contorsions dangereuses pour « rester » dans le monde UML.

Au contraire, l'IDM favorise la définition de langages de modélisation dédiés à un domaine particulier (*Domain Specific Modeling Languages* – DSML) offrant ainsi aux utilisateurs des concepts propres à leur métier et dont ils ont la maîtrise. Ces langages sont généralement de petite taille et doivent être facilement manipulables, transformables, combinables, etc.

L'intérêt des DSML est de bénéficier des avantages bien connus des langages spécifiques d'un domaine :

- Les langages dédiés permettent d'exprimer des solutions avec les tournures idiomatiques au niveau d'abstraction du domaine traité. En conséquence, les experts du domaine eux-mêmes peuvent comprendre, valider, modifier, et souvent même développer des programmes en langage dédié.
- Les langages dédiés facilitent la documentation du code.
- Les langages dédiés améliorent la qualité, la productivité, la fiabilité, la maintenabilité, la portabilité et les possibilités de réutilisation.
- Les langages dédiés permettent la validation au niveau du domaine. Aussi longtemps que les éléments du langage sont sûrs, toute phrase écrite avec ces éléments peut être considérée comme sûre.

De tels langages sont déjà utilisés avec succès dans de nombreux domaines tels que les télécommunications, l'avionique, l'aérospatial et l'industrie automobile. Certains *zoos* proposent un recensement, une documentation et une classification de ces DSML et offrent certaines manipulations, comme de pouvoir les transformer vers différents espaces techniques. Nous citons par exemple le zoo ReMoDD [5]

Le MDE promeut donc une séparation des préoccupations selon des domaines de compétences, dont les connaissances sont capitalisés au niveau du langage et de son environnement. La définition d'un système complexe fait généralement appel à plusieurs DSML. Leurs définitions, ainsi que la description des « ponts » entre eux sont respectivement abordées dans les chapitres 2 et 3.

5. ReMoDD, *Repository for Model Driven Development*, cf. http://www.cs.colostate.edu/remodd/

Chapitre 2

La métamodélisation

Comme nous l'avons vu dans le chapitre précédent, l'ingénierie dirigée par les modèles (IDM) préconise l'utilisation de plusieurs langages de modélisation (ou *Domain Specific Modeling Language* – DSML), dédiés chacun à un domaine particulier et permettant ainsi la séparation des préoccupations. Par conséquent, la première problématique clé de l'IDM est la maîtrise de la définition de ces DSML, dont le nombre ne cesse de croître avec la multiplication des domaines d'application de l'informatique. La métamodélisation, activité correspondant à définir un DSML, doit donc être étudiée et maîtrisée. Pour cela, les premiers travaux ont consisté à définir précisément les différentes composantes d'un langage de modélisation et à offrir les outils permettant de les décrire.

Définition (*Métamodélisation*) La métamodélisation est l'activité consistant à définir le métamodèle d'un langage de modélisation. Elle vise donc à modéliser un langage, qui joue alors le rôle de système à modéliser.

Dans la suite de ce chapitre, nous présentons tout d'abord les différents constituants d'un langage (section 2.1) et détaillons ensuite pour chacun d'entre eux les moyens de description (sections 2.2, 2.3, et 2.4).

2.1 Qu'est-ce qu'un langage ?

Que ce soit en linguistique (langage naturel) ou en informatique (langage de programmation ou de modélisation), il est depuis longtemps établi qu'un langage est caractérisé par sa *syntaxe* et sa *sémantique*. La syntaxe décrit les différentes constructions du langage et les règles d'agencement de ces constructions (également appelées *context condition* [HR04]). La sémantique désigne le lien entre un signifiant (un programme, un modèle, etc.), et un signifié (p. ex. un objet mathématique) afin de donner un sens à chacune des constructions du langage. Il y a donc entre la sémantique et la syntaxe le même rapport qu'entre le fond et la forme.

Définition (*Langage*) Un langage (L) est défini selon le couple $\{S, Sem\}$ où S est sa syntaxe et Sem sa sémantique.

Cette définition est très générale et assez abstraite pour caractériser l'ensemble des langages, quel que soit le domaine. De manière plus précise, dans le contexte de l'informatique, on distingue généralement la syntaxe concrète (CS sur la figure 2.1), manipulée par l'utilisateur du langage, de la syntaxe abstraite (AS sur la figure 2.1) qui est la représentation interne (d'un programme ou d'un modèle) manipulée par l'ordinateur [ASU86]. Dans les langages de programmation, la

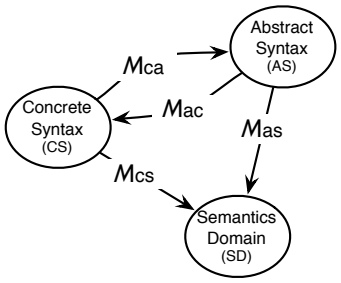


FIGURE 2.1 – Composantes d'un langage

représentation interne (l'arbre abstrait) est dérivée de la syntaxe concrète. Ainsi, la syntaxe d'un langage de programmation est définie par les syntaxes concrète et abstraite et par un lien, dit de *dérivation* ou d'*abstraction*, entre la syntaxe concrète et la syntaxe abstraite (M_{ca} sur la figure 2.1). Ce lien d'abstraction permet d'enlever tout le « sucre » syntaxique inutile à l'analyse du programme.

Par ailleurs, on distingue également le *domaine sémantique* (SD sur la figure 2.1) qui représente le sens d'un programme. La sémantique d'un langage de programmation est alors donnée en liant les constructions de la syntaxe concrète avec le sens auquel elles correspondent dans le domaine sémantique (M_{cs} sur la figure 2.1).

Définition (*Langage de programmation*) Un langage de programmation (L_p) est défini selon le tuple $\{AS, CS, M_{ca}, SD, M_{cs}\}$ où AS est la syntaxe abstraite, CS est la syntaxe concrète, M_{ca} est le *mapping* de la syntaxe concrète vers sa représentation abstraite, SD est le domaine sémantique et M_{cs} est le *mapping* de la syntaxe concrète vers le domaine sémantique.

Dans le contexte de l'IDM, la syntaxe abstraite est placée au cœur de la description d'un langage de modélisation. Elle est donc généralement décrite en premier et sert de base pour définir la syntaxe concrète. La définition de la syntaxe concrète consiste alors à définir des décorations (textuelles ou graphiques) et à définir un lien entre les constructions de la syntaxe abstraite et les décorations de la syntaxe concrète (M_{ac} sur la figure 2.1). Ce changement de sens du lien par rapport aux langages de programmation permet d'envisager de définir plusieurs syntaxes concrètes (M^*_{ac}) pour une même syntaxe abstraite et donc d'avoir plusieurs représentations d'un même modèle. Le langage peut alors être manipulé avec différents formalismes mais avec les mêmes constructions et la même représentation abstraite. De plus, dans le cadre des langages de modélisation, la sémantique est exprimée à partir des constructions de la syntaxe abstraite par un lien vers un domaine sémantique (M_{as} sur la figure 2.1). Différents mapping peuvent être définis et donnent lieu à différentes sémantiques, pour des objectifs différents (*e.g.,* simulation, compilation...), à des niveaux d'abstraction différents ou pour capturer une famille de DSML. D'autre part, un même modèle peut être simultanément ou successivement conforme à plusieurs DSML.

Définition (*Langage de modélisation*) Un langage de modélisation (L_m) est défini selon le tuple $\{AS, CS^*, M^*_{ac}, SD^*, M^*_{as}\}$ où AS est la syntaxe abstraite, CS^* est la (les) syntaxe(s) concrète(s), M^*_{ac} est l'ensemble des *mappings* de la syntaxe abstraite vers la (les) syntaxe(s) concrète(s), SD^* est le(s) domaine(s) sémantique(s) et M^*_{as} est l'ensemble des *mappings* de la syntaxe abstraite vers le(s) domaine(s) sémantique(s).

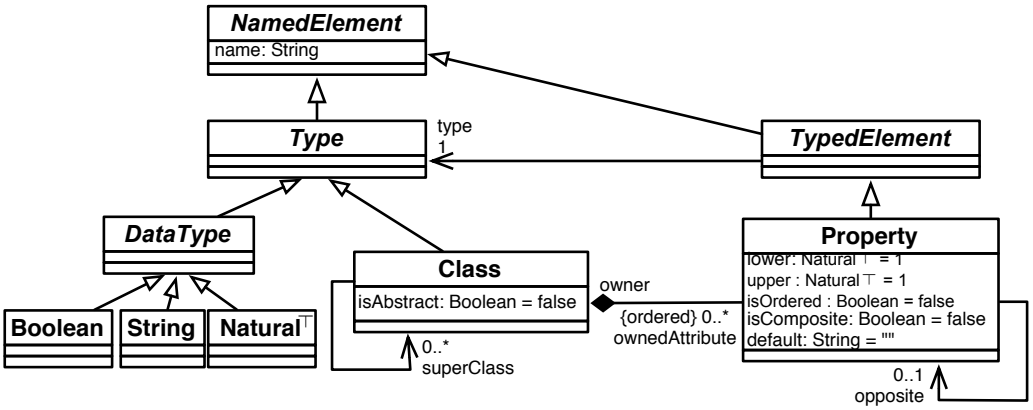


FIGURE 2.2 – Principaux concepts de métamodélisation (extrait de MOF 2.0 [OMG06a])

2.2 Syntaxe abstraite

2.2.1 Le langage de métamodélisation MOF de l'OMG

La syntaxe abstraite (*AS*) d'un langage de modélisation exprime, de manière structurelle, l'ensemble de ses concepts et leurs relations. Les langages de métamodélisation tels que le standard MOF de l'OMG [OMG06a], offrent les constructions élémentaires qui permettent de décrire une telle syntaxe abstraite au travers d'un modèle appelé *métamodèle*. Il existe à ce jour de nombreux environnements et langages de métamodélisation (cf. chapitre 5) : Eclipse-EMF/Ecore [BSE03], GME/MetaGME [LMB+01], AMMA/KM3 [JB06, ATL05], XMF-Mosaic/ Xcore [CESW04] ou Kermeta [MFJ05b]. Tous ces langages reposent toutefois sur les mêmes constructions élémentaires (cf. figure 2.2). S'inspirant de l'approche orientée objet, les langages de métamodélisation objet offrent le concept de classe (*Class*) pour définir les concepts d'un DSML. Une classe est composée de propriétés (*Property*) qui la caractérisent. Une propriété est appelée *référence* lorsqu'elle est typée (*TypedElement*) par une autre classe, et *attribut* lorsqu'elle est typée par un type de donnée (*DataType*, p. ex. booléen, chaîne de caractère et entier).

De cette manière, il est possible de définir un métamodèle par instanciation des éléments du langage de métamodélisation (*i.e.*, décrit dans le métamétamodèle). Par convention, la représentation graphique du diagramme de classe est utilisée à cet effet. Notons qu'il serait tout à fait possible d'utiliser la représentation graphique d'un diagramme d'objet, en particulier pour faire apparaître explicitement les liens d'instanciation entre les éléments du métamodèle et ceux du métamétamodèle. La figure 2.3 présente deux exemples où certains liens d'instanciation sont explicitement représentés par des flèches en pointillées.

2.2.2 Le langage de contrainte OCL de l'OMG

La représentation graphique offerte par les langages de métamodélisation ne permet pas de capturer formellement l'ensemble des propriétés du langage (c'est-à-dire les *context conditions*). Dans le domaine des langages de programmation, la sémantique axiomatique est basée sur des logiques mathématiques et exprime une méthode de preuve pour certaines propriétés des constructions d'un langage [Cou90]. Celle-ci peut être très générale (e.g. triplet de Hoare) ou restreinte à la garantie de la cohérence de ces constructions (p. ex. le typage). Dans le cadre d'un langage de modélisation, cette seconde utilisation est exprimée par le biais de règles de bonne formation (*Well-Formed Rules* – WFR), au niveau du métamodèle. Ces règles devront

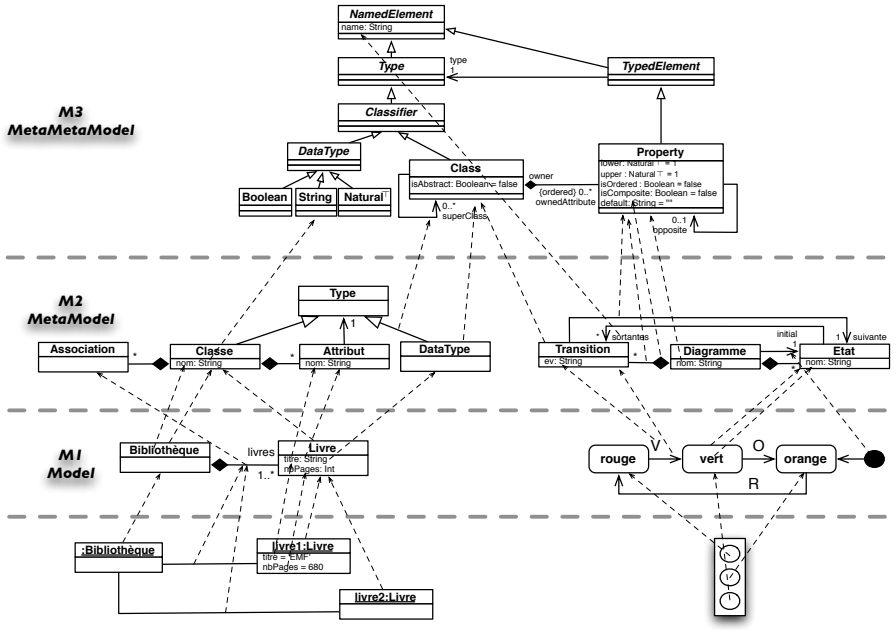


FIGURE 2.3 – Exemples de piles de modélisation

être respectées par les modèles conformes à ce métamodèle.

Pour exprimer ces règles, l'OMG préconise d'utiliser OCL (*Object Constraint Language*) [OMG10, WK03]. Appliqué au niveau du métamodèle, il permet d'ajouter des propriétés, principalement structurelles, qui n'ont pas pu être capturées par les concepts fournis par le métamétamodèle. Il s'agit donc d'un moyen de préciser la sémantique du métamodèle en limitant les modèles conformes.

La figure 2.4 illustre l'utilisation d'OCL selon ses différentes syntaxes et aux différents niveaux d'abstraction. L'expression d'une contrainte à un niveau d'abstraction permet de restreindre les instanciations possibles du modèle au niveau inférieur. Par exemple, la contrainte sur la classe *Livre* interdit à tout diagramme d'objet d'avoir un livre avec un nombre de pages négatif ou nul. La contrainte sur le métamodèle interdit à tout modèle d'avoir deux attributs de même nom dans une classe. Enfin, la contrainte sur le métamétamodèle interdit tout métamodèle avec deux classes de même nom.

2.3 Syntaxe concrète

Les syntaxes concrètes (CS) d'un langage fournissent à l'utilisateur un ou plusieurs formalismes, graphiques et/ou textuels, pour manipuler les concepts de la syntaxe abstraite et ainsi en créer des « instances », *i.e.*, des modèles. Les modèles ainsi obtenu seront conforme à la structure définie par la syntaxe abstraite. La définition d'une syntaxe concrète consiste à définir un des *mappings* de M^*_{ac} (cf. figure 2.1), $M_{ac} : AS \leftrightarrow CS$, et permet ainsi « d'annoter » chaque construction du langage de modélisation définie dans la syntaxe abstraite par une (ou plusieurs) décoration(s) de la syntaxe concrète et pouvant être manipulée(s) par l'utilisateur du langage.

La définition du modèle d'une syntaxe concrète est à ce jour bien maîtrisée et outillée (cf. chapitre 7). Il existe en effet de nombreux projets qui s'y consacrent, principalement

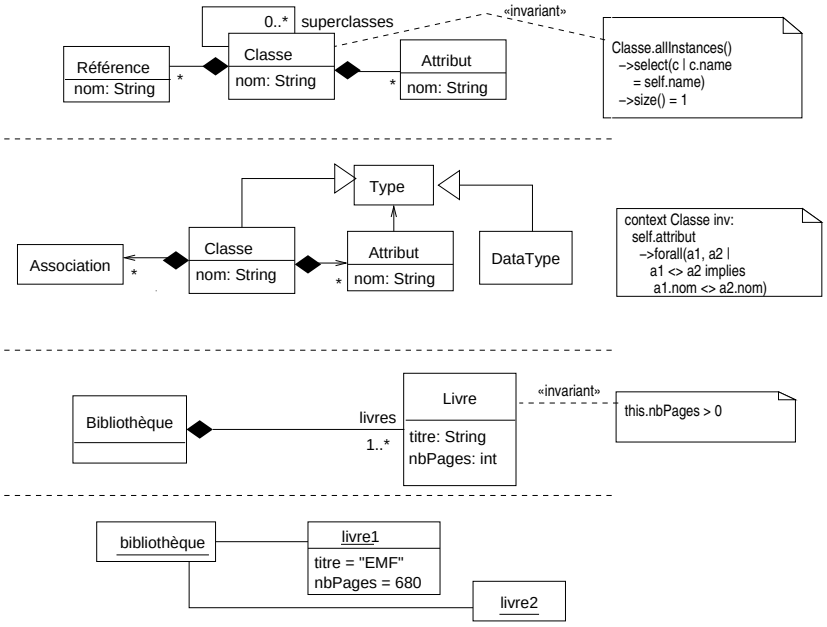


FIGURE 2.4 – Utilisations (et syntaxes) d'OCL

basés sur EMF (*Eclipse Modeling Framework*)[1] : GMF (*Generic Modeling Framework*)[2], TOPCASED[3] [FGC+06], OBEO Designer[4], etc. Si ces derniers sont principalement graphiques, d'autres projets permettent de définir des modèles de syntaxe concrète textuelle. Nous citons par exemple EMFText[5] et Xtext[6]. Ces approches génériques ont l'avantage de normaliser la construction des syntaxes concrètes.

La seule volonté de définir des modèles contemplatifs utilisés uniquement pour communiquer avec des constructions plus abstraites sur le système, ne demande pas une définition plus complète du langage de modélisation. En effet, les syntaxes abstraite et concrète sont suffisantes pour manipuler textuellement ou graphiquement un langage et réfléchir de manière plus abstraite à la construction du système. Dans ce contexte, ce sont les noms choisis pour nommer les concepts qui sont le support de la sémantique, et donc de la réflexion.

Toutefois, la manipulation automatique des modèles par des outils (à des fins de vérification, de simulation, de refactoring, ou de génération de documentation, de test ou de code) nécessite une formalisation de la sémantique jusque là implicite. Les outils peuvent ainsi interpréter automatiquement la sémantique et la prendre en compte dans leurs traitements. Dans la suite de ce chapitre, nous présentons les moyens de formaliser la sémantique d'un langage de modélisation en vue d'être supporté par des outils.

2.4 Sémantique

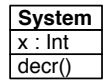
2.4.1 Rappels

La sémantique d'un langage définit de manière précise et non ambiguë la signification des constructions de ce langage. Elle permet ainsi de donner un sens précis aux modèles construits

1. cf. http://www.eclipse.org/emf
2. cf. http://www.eclipse.org/gmf
3. cf. http://www.topcased.org/
4. cf. http://obeo.fr/pages/obeo-designer/fr
5. cf. http://www.emftext.org
6. cf. http://www.eclipse.org/Xtext/

```
1  int x;
2  void decr () {
3    if ( x>0 )
4      x = x − 1;
5  }
```

(a) Programme



(b) Modèle

FIGURE 2.5 – Fonction *decr*

à partir de celui-ci. On dit qu'une sémantique est *formelle* lorsqu'elle est exprimée dans un formalisme mathématique et permet de vérifier la cohérence et la complétude de cette définition.

Il existe différentes techniques pour définir une sémantique et les importants travaux sur les langages de programmation on permis d'établir une taxonomie de ces techniques en fonction des besoins [Win93].

On distingue la *sémantique statique* qui correspond à des propriétés indépendantes de l'exécution ou valables pour toutes les exécutions et la *sémantique dynamique* (ou *comportementale*) qui permet de décrire le comportement des programmes à l'exécution. La sémantique statique est en général vérifiée statiquement lors de la compilation des programmes (et exprime des règles, comme le typage, qui ne peuvent pas être exprimées par la syntaxe). La sémantique dynamique peut être décrite selon différentes techniques. Pour aller de la plus concrète à la plus abstraite, une sémantique dynamique peut être décrite de manière opérationnelle, dénotationnelle ou axiomatique [Win93]. Une *sémantique opérationnelle* donne une vision impérative en décrivant un programme par un ensemble de transitions (ou transformations) entre les états du contexte d'exécution (p. ex. de la mémoire). Une *sémantique dénotationnelle* décrit, sous forme de fonctions, l'effet d'un programme et non la manière dont celui-ci est exécuté. Une *sémantique axiomatique* propose une vision déclarative en décrivant l'évolution des caractéristiques d'un élément lors du déroulement d'un programme.

Définir une sémantique d'un langage de modélisation revient à définir ou à choisir le domaine sémantique SD et à définir le *mapping* M_{as} entre la syntaxe abstraite et le domaine sémantique ($AS \leftrightarrow SD$). Dans le contexte de l'IDM, au même titre que les autres éléments d'un langage de modélisation, la définition du domaine sémantique et du *mapping* prend la forme de modèles [Hau05, HbR00, HR04].

On peut classer les différentes sémantiques selon les catégories définies pour les langages de programmation (sémantique axiomatique, dénotationnelle et opérationnelle), et appliquer celles-ci selon les besoins (vérification, simulation, animation, compilation, etc.) et les outils disponibles [CCGT09]. Nous les présentons dans la suite de cette partie en les illustrant à travers l'exemple simple de la fonction *decr* dont le programme [7] et un de ses modèles sont fournis sur la figure 2.5. Ce programme décrit une fonction qui décrémente de un la valeur de x à condition qu'elle soit strictement positive. Nous exprimons par la suite la sémantique du modèle correspondant selon les trois niveaux d'abstraction cités.

2.4.2 Sémantique axiomatique

La *sémantique axiomatique* ainsi que la sémantique statique (également appelée *context condition* par l'OMG et correspondant à une sémantique axiomatique très simple pouvant être vérifiée statiquement [8]) peuvent être exprimées sur la syntaxe abstraite, soit à l'aide du langage

7. Le programme est proposé selon la syntaxe et la sémantique du langage C

8. Notons qu'en UML il est possible d'exprimer des propriétés OCL portant sur des séquences d'envois de messages. Cependant, ces propriétés ne sont pas vérifiées par les vérificateurs OCL car elles nécessitent la formalisation de la sémantique d'exécution du langage, par exemple UML, et la mémorisation de la trace d'exécution

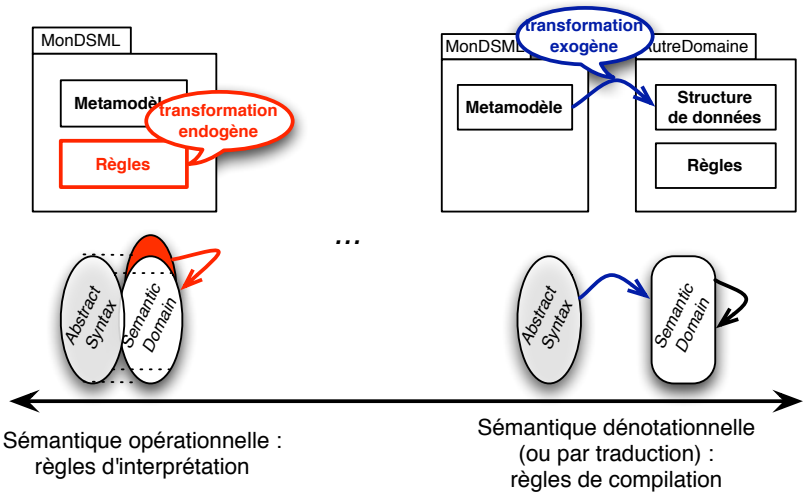


FIGURE 2.6 – Sémantiques opérationnelle et dénotationnelle (ou *par traduction*)

de métamodélisation lui-même (p. ex. les multiplicités) soit en la complétant de contraintes exprimées à l'aide d'un langage comme OCL (invariant, pré- ou post-condition). Les pré- et post-conditions supposent d'avoir identifié des opérations sur le modèle. Une spécification axiomatique de la sémantique de la fonction *decr* peut, par exemple, être définie par la propriété OCL donnée sur le *listing* 2.1.

Listing 2.1 – Sémantique axiomatique de la fonction *decr*

```
1  context System::decr()
2    post : self.x = if ( self.x@pre > 0 ) then
3           self.x@pre - 1
4         else
5           self.x@pre
6         endif
```

2.4.3 Sémantique opérationnelle

La *sémantique opérationnelle* s'exprime sur une représentation abstraite similaire à celle de la syntaxe abstraite du langage considéré, à laquelle est rajouté les informations que l'on souhaite capturer sur l'exécution (*i.e.,* les informations permettant de distinguer les différents états du modèle au cours de son exécution, en fonction du niveau d'abstraction choisi). Elle considère donc comme domaine sémantique une extension du domaine syntaxique (cf. partie gauche sur la figure 2.6). Cela permet alors d'exprimer la sémantique comportementale (i.e., comment évolue l'état du modèle) directement sur les concepts du langage dédié au domaine, instanciés à partir des paradigmes utilisés pour la définition de la syntaxe abstraite (dans notre cas, les concepts de métamodélisation). Ces concepts sont dénués de sémantique mais un *langage d'action* permet de l'exprimer et ainsi de définir les outils support à l'exécution. Un tel langage permet de décrire l'évolution du modèle et de produire, à partir d'un état donné, un ou plusieurs autres états (dit successeurs). Dans le contexte des langages naturels, une sémantique opérationnelle revient à manipuler une phrase (le plus souvent mentalement), dans la même langue que celle utilisée pour l'écrire initialement. On lui donne dans ce cas un sens selon l'expérience acquise (propre à chacun). De manière plus formelle, notons qu'une sémantique opérationnelle est, par définition, fortement bisimilaire [9] au système de transition de référence du système. En effet, la

9. Selon la définition de la bisimulation donnée dans [Mil95].

représentation abstraite sur laquelle elle est exprimée étant identique à la syntaxe abstraite du langage, chaque état exprimé par la sémantique opérationnelle est observable par le système de transition de référence.

Pour l'exemple de la fonction *decr*, la sémantique opérationnelle peut être spécifiée de la manière suivante : $\mathbb{Z} \mapsto \mathbb{Z}$ tel que $x \mapsto if(x > 0)?x - 1 : x$. Le domaine sémantique choisi dans ce cas correspond à $\mathbb{Z}$.

Dans la pratique, différentes solutions ont été explorées pour implanter une sémantique opérationnelle directement dans la syntaxe abstraite (*i.e.,* le métamodèle).

Une première solution est l'utilisation d'un langage de métamodélisation exécutable, c'est-à-dire un langage de métamodélisation offrant également un langage d'action permettant de décrire la sémantique opérationnelle (*e.g.,* Kermeta [MFJ05b], xOCL [CSW08b], langage d'action MOF [PKP06] ou même Java avec l'API d'EMF). Dans ce cas, la sémantique est directement exprimée comme un ensemble d'opérations pour chaque concept. Notons que ces langages d'action peuvent proposer des paradigmes de haut-niveau permettant de séparer malgré tout la syntaxe abstraite et la sémantique opérationnelle (*e.g.* la modélisation orientée aspect dans Kermeta [Jéz08]).

La seconde solution est d'exprimer une transformation endogène sur la syntaxe abstraite à l'aide d'un langage de transformation déclaratif. Par exemple, dans [MB08] les auteurs utilisent QVT [OMG11a] pour exprimer des règles de réécriture qui calculent graduellement les valeurs d'une expression OCL.

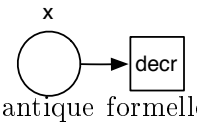
Les transformations de modèle ont également été largement utilisées au travers de la transformation de graphe [Roz97], fournissant une technique déclarative et basée sur la notion de règle pour définir une sémantique opérationnelle (*i.e.,* de manière similaire à un système de transition). Par exemple, AGG [Tae03], ATOM3 [BdLG07], et GROOVE [KKR06] peuvent être utilisé dans cette optique. Kuske *et al.* [KGKK02] ont utilisé une telle approche pour définir la sémantique opérationnelle de certains diagrammes UML ainsi que les relations entre eux. Hausmann introduit dans [Hau05] la notion de *métamodélisation dynamique* comme une technique pour décrire la sémantique opérationnelle des langages de modélisation visuels (*i.e.,* graphiques). Dans [EHS00], les règles de transformation de graphe sont graphiquement définies à l'aide de diagrammes de collaboration. Dans [FNTZ98], des transformations élémentaires sont représentées comme des diagrammes de collaboration UML indiquant les éléments à ajouter ou à supprimer. Ces transformations sont ensuite embarquées dans l'état d'un diagramme d'activité UML qui contrôle l'ordre d'application pour la transformation. Cette approche ressemble donc à un langage de méta-programmation graphique (appelé *Story Diagrams*). Les actions sont des transformations basées sur la réécriture de graphe et les structures de contrôles sont fournit par un diagramme d'activité UML.

2.4.4 Sémantique dénotationnelle (ou *par traduction*)

La *sémantique dénotationnelle* s'appuie sur une représentation abstraite différente de celle définie dans la syntaxe abstraite du langage considéré (cf. partie droite sur la figure 2.6). Donc, à la différence de la sémantique opérationnelle, une sémantique dénotationnelle traduit chaque état du modèle dans un autre espace technique (formellement défini), s'appuyant ainsi sur la sémantique de celui-ci. En pratique, définir la sémantique dénotationnelle revient à définir la traduction de chaque construction du langage de modélisation en construction(s) du langage de l'espace technique cible choisi. Celui-ci est choisi de manière à offrir des paradigmes sémantiquement bien définis et adaptés à la construction d'outils efficaces pour le support de l'exécution (p. ex. des outils de *model-checking*). Dans le contexte des langages naturels, cela

revient à interpréter une phrase dans une langue différente que celle dans laquelle elle a été initialement écrite. L'expression de la sémantique correspond à trouver alors une traduction dans la nouvelle langue pour que la phrase garde le même sens. De manière plus formelle, le domaine choisi est généralement faiblement bisimilaire au système de transition de référence du système. Il est alors nécessaire de retrouver les états que l'on souhaite observer dans le formalisme initial. Ce type de sémantique correspond dans l'IDM à une *sémantique par traduction* (*translational semantics* en anglais) qui définit un langage par sa transformation vers un autre langage formellement défini [CESW04].

Une sémantique par traduction de la fonction *decr* peut correspondre à la fonction qui traduit ce programme dans un domaine formel tel que les réseaux de Petri [Rei85]. Une telle fonction pourrait alors fournir le réseau suivant pour la fonction *decr*, où la valeur de x est représentée par le marquage de la place de même nom :



Les réseaux de Petri ayant une sémantique formelle, le programme prend alors le sens du réseau de Petri généré.

La sémantique par traduction est utilisé par le groupe *pUML*[10], appelé *Denotational Meta Modeling*, afin de formaliser certains diagrammes UML [CEK01].

D'autre part, le laboratoire ISIS propose, dans le cadre de son approche MIC (*Model-Integrated Computing*), la notion d'ancrage sémantique (*i.e., semantics anchoring* [CSAJ05]) qui est un type de sémantique dénotationnelle. L'objectif est de traduire chaque élément d'un DSML vers des unités sémantiques, elles-mêmes définies à l'aide d'une sémantique opérationnelle (par exemple en utilisant les *Abstract State Machines* – ASM).

Les ASM sont également utilisés dans [GRS09] comme un langage d'action pour définir opérationnellement la sémantique comportementale d'un DSML. Pour cela, les auteurs proposent un environnement qui permet une traduction transparente entre le langage d'action et les ASM. De manière similaire, [RV07, RRDV07] vise à formellement définir la sémantique des DSML en offrant une traduction vers l'environnement formel Maude. Maude est alors utilisé pour exprimer la sémantique opérationnelle grâce à des règles de réécriture. Les mêmes auteurs ont récemment utilisé dans [RaJdLV08] la transformation de graphe pour directement exprimer la sémantique opérationnelle sur la syntaxe abstraite du DSML. Celle-ci est ensuite automatiquement traduite dans l'environnement Maude.

2.4.5 Bilan

Une *sémantique d'exécution* permet de donner un caractère opératoire à un DSML. Le métamodèle est alors dit *exécutable* et permet de faire évoluer les modèles qui lui sont conformes au cours d'une exécution conformément à la sémantique du langage. Celle-ci peut être définie de manière opérationnelle ou par traduction vers un domaine sémantique opérationnel. Notons que ces deux solutions ne se distinguent que par le choix du domaine sémantique (cf. figure 2.6). Aux extrêmes, le choix peut se porter vers une copie du domaine syntaxique (sémantique opérationnelle), ou vers un domaine pré-existant totalement différent du domaine syntaxique (sémantique par traduction). Toute solution intermédiaire peut également être envisagée.

De nombreux travaux utilisent la sémantique par traduction, principalement pour réutiliser les outils disponibles dans l'espace technique cible (générateurs de code, *model-checkers*, simulateurs, outils de visualisation, etc.). Néanmoins, un inconvénient majeur de cette solution

10. *The precise UML group*, cf. http://www.cs.york.ac.uk/puml/

est qu'elle nécessite une connaissance approfondie du domaine cible par le concepteur du DSML, et de vérifier formellement que la traduction est correcte.

A l'inverse, une sémantique opérationnelle s'exprime plus facilement, directement sur les constructions du langage de modélisation (et donc au niveau d'abstraction choisi au moment de la définition des constructions du langage dans la syntaxe abstraite). Néanmoins, il est alors nécessaire pour chaque nouveau DSML de définir l'ensemble des outils supports à cette sémantique.

Approche outillée de métamodélisation exécutable pour la simulation et la vérification de modèles Afin de faciliter la mise en œuvre de chacune des techniques pour décrire et outiller une sémantique d'exécution, des approches méthodologiques et technologiques ont récemment été développées. Le livre « Simulation et vérification de modèle par métamodélisation exécutable » [Com10] (publié d'après le manuscrit de la thèse de Benoît Combemale [Com08]) propose par exemple une approche facilitant la description et l'outillage d'un DSML "exécutable". L'auteur propose pour cela un patron d'architecture pour la description de la syntaxe abstraite de manière à faciliter :

1. la définition et la validation de sémantique par translation (pour la réutilisation d'outils, en particulier de vérification formelle tel que des *model-checker*),
2. la définition d'outils support à une sémantique opérationnelle (tel que des animateurs ou des simulateurs de modèles).

Ces travaux ont été élaborés et utilisés dans le cadre du projet Topcased [a] dont l'objectif fut d'établir pour l'industrie des systèmes critiques, embarqués et temps réel (tel que l'avionique, le spatial ou l'automobile) un environnement – éponyme – de développement dirigée par les modèles prenant en compte l'ensemble des étapes du cycle de développement.

a. cf. http://www.topcased.org

D'autres taxonomies des techniques pour décrire la sémantique des langages de modélisation ont été proposées. Clark *et al.* [CESW04, CSW08a] partagent le fait de distinguer la sémantique opérationnelle et la sémantique par traduction. Leurs travaux se concentrent néanmoins sur la sémantique d'exécution et n'évoquent pas la sémantique axiomatique. Ils proposent également la notion de sémantique par extension, consistant à étendre les concepts et la sémantique d'un langage existant. La sémantique est néanmoins donnée de manière opérationnelle ou par traduction. Enfin, nous ne partageons pas la définition d'une sémantique dénotationnelle, décrite comme un mapping vers un domaine sémantique. Il s'agit pour nous de la définition générale qui généralise toutes les approches consistant à définir une sémantique.

Hausmann [Hau05] propose également une taxonomie des techniques pour exprimer une sémantique comportementale. Il liste pour cela les techniques disponibles pour atteindre des objectifs spécifiques (*e.g.,* vérification de propriétés, analyse de la consistance, génération de code, etc.). Il identifie également les techniques générales pour exprimer une sémantique, incluant les sémantiques opérationnelle et par traduction.

Chapitre 3

Des modèles contemplatifs aux modèles productifs

Les chapitres précédents ont permis de mettre l'accent sur l'importance d'une définition explicite des langages de modélisation afin de permettre une capitalisation d'un domaine de connaissance particulier. Nous avons vu également que cette définition du langage est indispensable à l'élaboration d'outils manipulant automatiquement les modèles qui s'y conforme. Nous citerons l'exemple d'un compilateur qui s'appuie sur la définition d'un langage afin de pouvoir s'appliquer sur l'ensemble des programmes qui s'y conforme. Ainsi sur la base des pratiques et standards de métamodélisation présentés au chapitre précédent, différents langages de transformation ont été définis pour permettre d'exprimer des traitements au niveau d'un DSML et s'appliquant ainsi à l'ensemble des modèles qui s'y conforme.

Dans ce chapitre, nous commençons par présenter les fondements de la *transformation de modèle* et les standards sous-jacents (section 3.1). Sur la base de transformations élémentaires, nous montrons ensuite quelques utilisations pour la définition d'opérateurs plus complexes et que nous regroupons sous le terme de *composition de modèle* (section 3.2).

3.1 La transformation de modèle

Une problématique clé de l'IDM consiste à pouvoir rendre opérationnels les modèles à l'aide de transformations. Cette notion est au centre de l'approche MDA et plus généralement de celle des DSML. En effet, l'intérêt de transformer un modèle Ma en un modèle Mb, que les métamodèles respectifs MMa et MMb soient identiques (transformation *endogène*) ou différents (transformation *exogène*), apparaît comme primordial (génération de code, *refactoring*, migration technologique, etc.) [Béz04b].

D'autre part, l'approche MDA repose sur le principe de la création d'un modèle indépendant de toute plateforme (PIM) pouvant être raffiné en un ou plusieurs modèle(s) spécifique(s) à une plateforme (PSM). Les méthodes de transformation sont là aussi indispensables pour changer de niveau d'abstraction (transformation *verticale*), dans le cas du passage de PIM à PSM et inversement, ou pour rester au même niveau d'abstraction (transformation *horizontale*) dans le cas de transformation PIM à PIM ou PSM à PSM [GLR+02]. Ces différentes classes de transformation sont reprises sur la figure 3.1 en indiquant leurs cas d'utilisation.

Enfin, la transformation de modèle est également utilisée dans la définition des langages de modélisation pour établir les *mappings* et des traductions entre différents langages. Ces différentes classes de transformation sont résumées sur la figure 3.2.

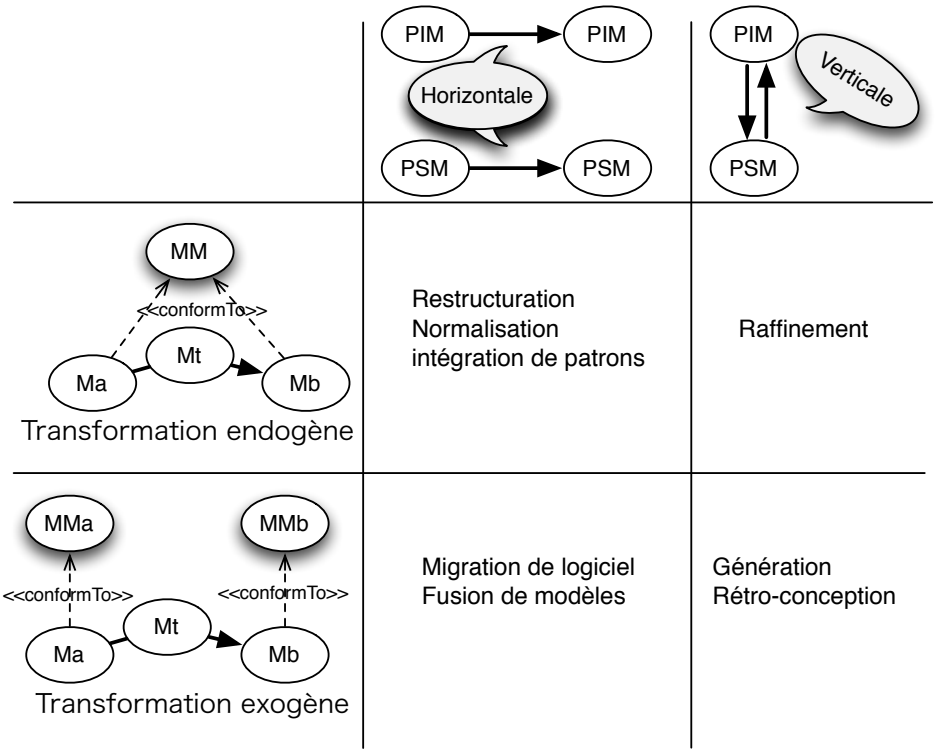


FIGURE 3.1 – Types de transformation et leurs principales utilisations

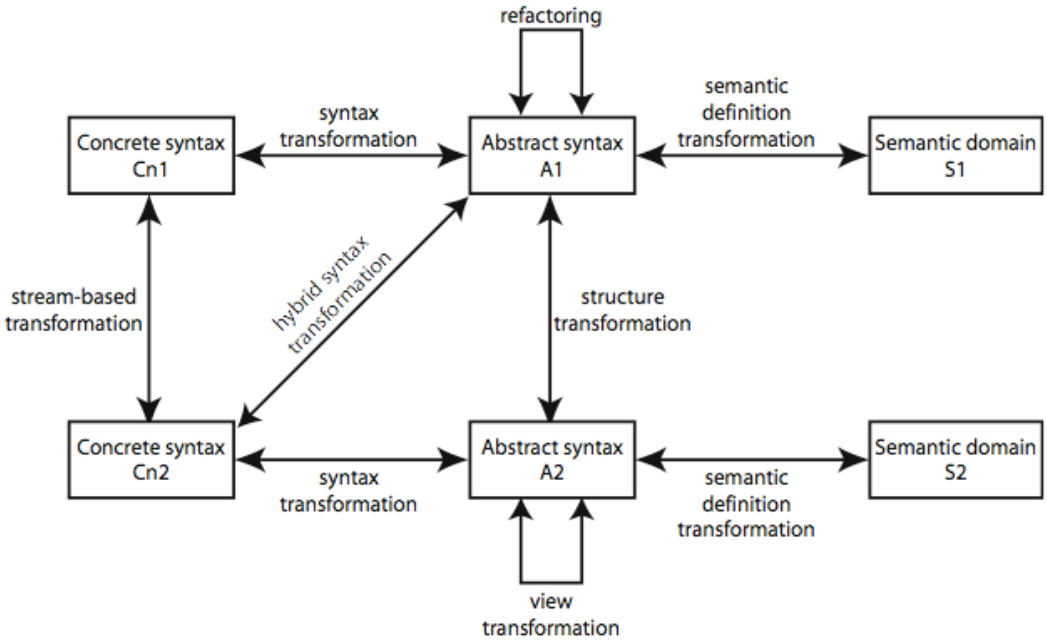


FIGURE 3.2 – Classes de transformation de modèle (extrait de [Kle06])

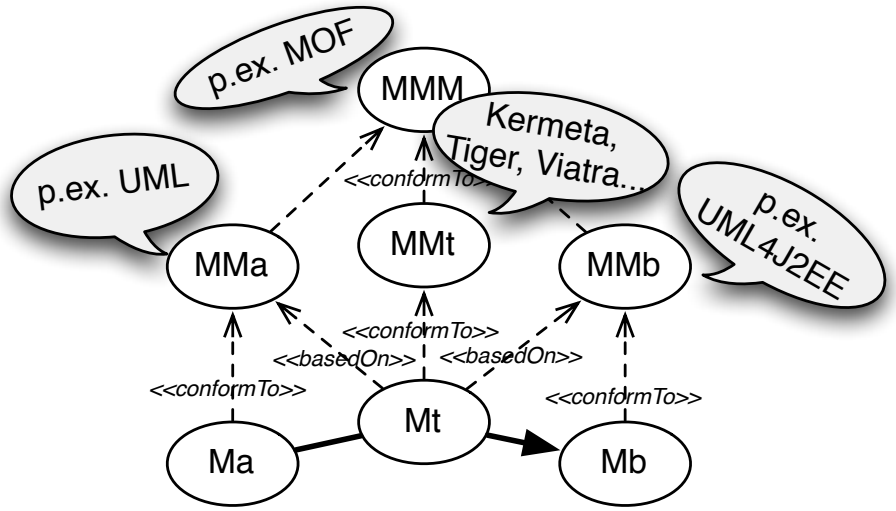


FIGURE 3.3 – Principes de la transformation de modèle

On comprend donc pourquoi le succès de l'IDM repose en grande partie sur la résolution du problème de la transformation de modèle. Cette problématique a donné lieu ces dernières années à de nombreux travaux académiques, industriels et de normalisation que nous présentons ci-après.

3.1.1 Historique

Les travaux réalisés dans le domaine de la transformation de modèle ne sont pas récents et peuvent être chronologiquement classés selon plusieurs générations en fonction de la structure de données utilisée pour représenter le modèle [Béz03] :

- *Génération 1 : Transformation de structures séquentielles d'enregistrement.* Dans ce cas un script spécifie comment un fichier d'entrée est réécrit en un fichier de sortie (p. ex. des scripts Unix, AWK ou Perl). Bien que ces systèmes soient plus lisibles et maintenables que d'autres systèmes de transformation, ils nécessitent une analyse grammaticale du texte d'entrée et une adaptation du texte de sortie [GLR+02, Béz03].
- *Génération 2 : Transformation d'arbres.* Ces méthodes permettent le parcours d'un arbre d'entrée au cours duquel sont générés les fragments de l'arbre de sortie. Ces méthodes se basent généralement sur des documents au format XML et l'utilisation de XSLT [1] ou XQuery [2].
- *Génération 3 : Transformation de graphes.* Avec ces méthodes, un modèle en entrée (graphe orienté étiqueté) est transformé en un modèle en sortie.
  Ces approches visent à considérer l' « opération » de transformation comme un autre modèle (cf. figure 3.3) conforme à son propre métamodèle (lui-même défini à l'aide d'un langage de métamodélisation, tel que le MOF). La transformation d'un modèle Ma (conforme à son métamodèle MMa) en un modèle Mb (conforme à son métamodèle MMb) par le modèle Mt peut donc être formalisée de la manière suivante :

  $$Mb^* \leftarrow f(MMa^*, MMb^*, Mt, Ma^*)$$

  Notons qu'il est possible d'avoir plusieurs modèles d'entrées (Ma^*) et de sorties (Mb^*).

1. *XSL (eXtensible StyleSheet Language) Transformation*, cf. http://www.w3.org/TR/xslt.
2. *An XML Query Language*, cf. http://www.w3.org/TR/xquery/.

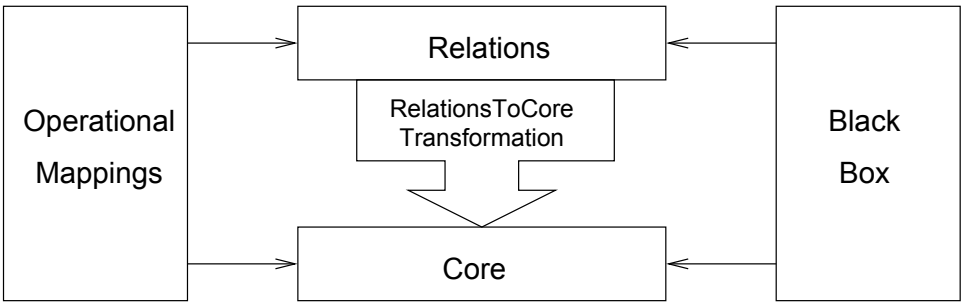


FIGURE 3.4 – Architecture du standard QVT (extrait de [OMG11a])

Cette dernière génération a donné lieu à d'importants travaux de recherche et à la proposition de diverses approches [CH03]. Celles-ci peuvent être caractérisées à partir de critères comme le paradigme pour la définition des transformations, les scénarios de transformation, la directivité des transformations établies, le nombre de modèles sources et cibles, la traçabilité, le langage de navigation utilisé, l'organisation et l'ordonnancement des règles, etc. [Jou06].

3.1.2 Standards et langages pour la transformation de modèle

A ce jour, de nombreux langages sont disponibles pour écrire des transformations de modèle de troisième génération. On retrouve d'abord les langages généralistes qui s'appuient directement sur la représentation abstraite du modèle. On citera par exemple l'API[3] d'EMF [BSE03] qui, couplée au langage Java, permet de manipuler un modèle sous la forme d'un graphe. Dans ce cas, c'est à la charge du programmeur de faire la recherche d'informations dans le modèle, d'expliciter l'ordre d'application des règles[4], de gérer les éléments cibles construits, etc.

Afin d'abstraire la définition des transformations de modèle et rendre transparent les détails de mise en œuvre, l'idée a été de définir des DSML dédiés à la transformation de modèle. Cette approche repose alors sur la définition d'un métamodèle dédié à la transformation de modèle et des outils permettant d'exécuter les modèles de transformation. De manière impérative nous citerons par exemple Kermeta [MFJ05b], qui est défini comme un langage de métamodélisation exécutable ou de métaprogrammation objet : il permet de décrire des métamodèles dont les modèles sont exécutables. Kermeta a été conçu comme un tissage entre EMOF [OMG06a] et un métamodèle comportemental (*i.e.,* un langage d'action). Celui-ci correspond à une hiérarchie de métaclasses représentant des expressions impératives et pouvant être utilisées pour définir le corps des opérations définies sur chaque concept d'une syntaxe abstraite. Ces opérations peuvent décrire n'importe quel traitement sur les modèles, et sont alors assimilées à une transformation.

Nous citerons également ATL (*ATL Transformation Language*) [JK05]. Il s'agit d'un langage hybride (déclaratif et impératif) qui permet de définir une transformation de modèle à modèle (appelée *Module*) sous la forme d'un ensemble de règle. Il permet également de définir des transformations de type modèle vers texte (appelée *Query*). Une transformation prend en entrée un ensemble de modèles (décrits à partir de métamodèles en Ecore ou en KM3).

Afin de donner un cadre normatif pour l'implantation des différents langages dédiés à la transformation de modèle, l'OMG a défini le standard QVT (*Query/View/Transformation*) [OMG11a]. Le métamodèle de QVT est conforme à MOF et utilise OCL pour la navigation dans les modèles. Le métamodèle fait apparaître trois sous-langages pour la transformation de modèles (cf. figure 3.4), caractérisés par le paradigme mis en œuvre pour la définition des transformations (déclaratif, impératif et hybride). Les langages *Relations* et *Core* sont tous deux

3. *Application Programming Interface* ou Interface de programmation.
4. Une règle est l'unité de structuration dans les langages de transformation de modèle.

déclaratifs mais placés à différents niveaux d'abstraction. L'un des buts de *Core* est de fournir une base pour la spécification de la sémantique de *Relations*. La sémantique de *Relations* est donnée comme une transformation de *Relations* vers *Core*. Il est parfois difficile de définir une solution complètement déclarative à un problème de transformation donné. Pour adresser cette question, QVT propose deux mécanismes pour étendre *Relations* et *Core* : un troisième langage appelé *Operational Mappings* et un mécanisme d'invocation de fonctionnalités de transformation implémentées dans un langage arbitraire (boîte noire ou *black box*). *Operational Mappings* étend *Relations* avec des constructions impératives et des constructions OCL avec effets de bord.

3.2 La composition de modèle

L'application systématique des principes de l'IDM entraine une multiplication des DSML dans les processus de développement. En plus de définir et d'outiller individuellement chacun de ces DSML, de nombreuses situations requièrent une formalisation de relations complexes entre les différents DSML. Ces relations sont souvent plus complexes que des transformations élémentaires et donnent lieu à des opérateurs dit de composition de modèles. Ces opérateurs facilitent la manipulation de différents modèles, hétérogènes (*i.e.,* issus de différents métamodèles) ou non, en raisonnant à un niveau d'abstraction plus élévé que la description de transformations élémentaires. Nous présentons dans la suite de cette section différentes situations récurrentes donnant lieu à la définition de relations complexes entre langages.

3.2.1 Chaînes de compilation

Plus un langage est défini à un haut niveau d'abstraction (*i;e.,* proche du domaine du problème et non de la solution), plus la définition de la chaine de compilation vers du code interprétable par une machine est complexe. Celle-ci nécessite souvent d'introduire de nombreux langages intermédiaires pour réduire la complexité d'une étape de compilation par l'introduction de plusieurs étapes de compilation, chacune dédiée à un aspect particulier de la compilation. Les plate-formes d'exécution se multipliant, il est également fréquent d'avoir recourt à la définition de langage pivot (*i.e.,* intermédiaire) comme une abstraction d'une famille de plateformes cibles. Un langage pivot permet à la fois de capitaliser des étapes de compilations vers différentes plate-formes, et de réduire la distance sémantique entre les langages sources et les langages cibles.

La multiplication des étapes de compilation donne lieu à des relations complexes entre les différents langages qu'il est nécessaire de maitriser, par exemple pour assurer une optimisation optimale de la compilation du modèle.

3.2.2 Co-évolution ou synchronisation de modèles

L'utilisation de différents DSML permet de séparer les préoccupations d'un système complexe en les capturant au sein de modèles dédiés. Il est alors indispensable de garantir la cohérence et la co-évolution (*i.e.,* synchronisation) entre ces différents modèles. Pour cela, les liens entre les différents DSML auxquels se conforment les modèles doivent être définis (cf. Fig 3.5) afin d'assurer la cohérence entre les différentes vues su un même système.

3.2.3 Interopérabilité de systèmes

La conception de système complexe (ou *Ultra-Large Scale System*) se ramène parfois à les considérer comme une fédération de (sous-)systèmes intéropérables. On parle alors de *système de systèmes*, *i.e.,* un système défini comme une orchestration de sous-systèmes. Cette orchestration

nécessite de mettre en œuvre des relations complexes d'interopérabilité entre les différents sous-systèmes (cf. Fig 3.6) afin d'assurer leurs communications et leurs cohérences.

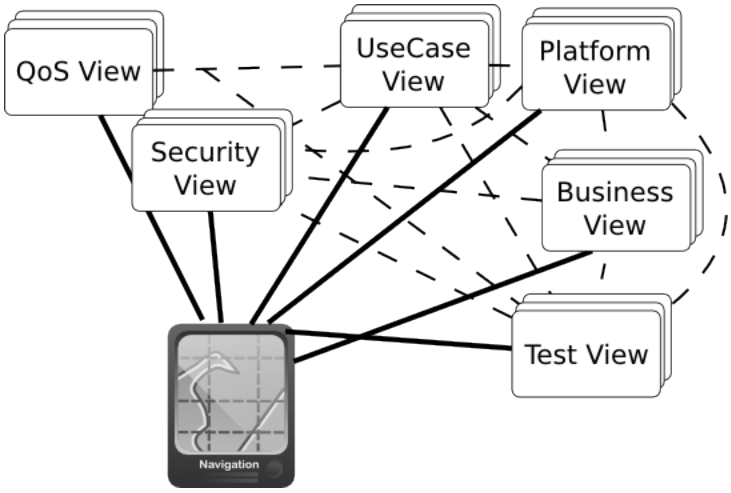


FIGURE 3.5 – Co-évolution ou synchronisation de modèles

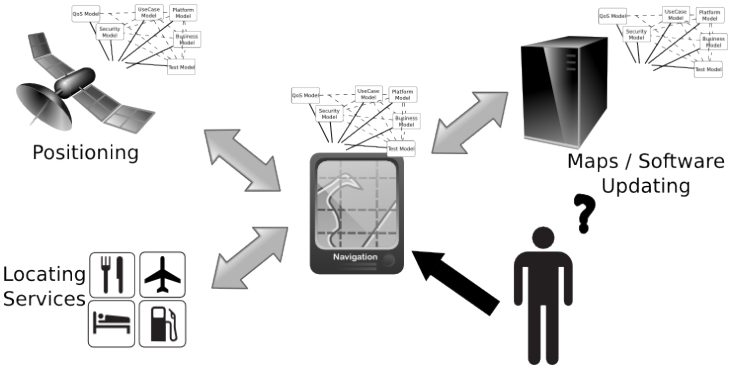


FIGURE 3.6 – Interopérabilité entre systèmes

3.2.4 Évolution de système

Les évolutions technologiques ne cessent de se multiplier. Malgré cela, la durée de vie des systèmes est de plus en plus longue (*e.g.,* jusqu'à 75 ans dans l'avionique). Par conséquent, les systèmes doivent de plus en plus être migrés d'une technologie à l'autre en les faisant évoluer tout au long de leur cycle de vie. L'automatisation de telles évolutions (*i.e.*, la migration automatique de modèles) nécessite de pouvoir définir des relations complexes entre les différentes technologies (cf. Fig 3.7).

3.2.5 Bilan

Comme on peut le constater, l'IDM est encore en pleine évolution. De nombreuses recherches sont en cours pour conceptualiser et développer des opérateurs de plus haut niveau pour manipuler les modèles à un niveau d'abstraction plus important. Ce constat est le résultat d'une forte adoption de l'IDM dans de nombreux processus industriel, entrainant une multiplication des DSML. La maîtrise de leur multiplication (en particulier des coûts et de la facilité pour les outiller) est un des principaux futurs challenges. Malgré tout, le déploiement de l'IDM dans

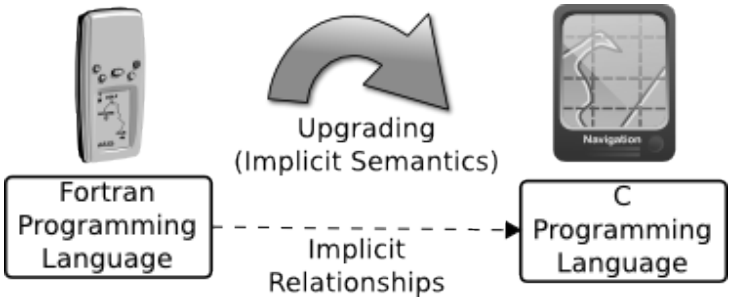


FIGURE 3.7 – Évolution de systèmes

les processus de développement est déjà pleinement fonctionnel. Nous vous proposons dans la deuxième partie de ce livre une mise en pratique de chacune des étapes d'un processus de métamodélisation (de la définition à l'outillage d'un DSML) au travers d'un exemple pédagogique, et d'un ensemble de bonnes pratiques pour aider la mise en œuvre dans des processus particuliers.

Deuxième partie

La pratique

In theory, there is no difference between theory and practice. But, in practice, there is.

Jan L.A. van de Snepscheut.

Chapitre 4

Processus de métamodélisation

Nous avons vu dans la première partie qu'un métamodèle définit un langage pour un domaine métier donné. Nous allons maintenant aborder les étapes permettant de définir et d'outiller concrètement un métamodèle afin de construire un environnement qui en tire parti.

4.1 Différents profils d'utilisateurs

L'IDM considère explicitement deux niveaux de processus pour le développement de systèmes :

- *un processus de métamodélisation* qui consiste à capitaliser un domaine de connaissance au niveau d'un DSML et de son environnement ;
- *un processus de modélisation* qui consiste à utiliser les résultats du premier processus pour définir des systèmes complexes tout en bénéficiant de l'expérience capitalisée au sein du DSML (p.-ex., des concepts dédiés) et des outils associés (p.-ex., pour l'automatisation de certaines tâches récurrentes des développements dans ce domaine particulier).

Ces deux processus se retrouvent dans la figure 4.1 où l'on observe pour chacun d'eux des rôles similaires (Experts, Développeurs, Application) assurés par des acteurs différents du développement :

- *L'équipe de métamodélisation* développe des outils de modélisation (par exemple, créer un modeleur ou un générateur de code).
- *L'équipe de modélisation* utilise les outils de modélisation pour développer un système final (par exemple une application web). Cette équipe exprime ses expériences et ses connaissances auprès de l'équipe de métamodélisation pour qu'elle assure une capitalisation au sein des outils de modélisation (par exemple, demander à améliorer le générateur de code pour supporter un nouveau type de base de données).
- *L'utilisateur final* est le client qui va utiliser le système. Il exprimera ses besoins auprès de l'équipe de modélisation (par exemple, décrire les fonctionnalités de l'application web souhaitée).

On notera que l'équipe de modélisation doit parfois changer de rôle : d'un coté elle utilise des outils (en tant que développeur), et de l'autre elle contribue à améliorer les outils (en tant qu'expert d'un domaine particulier).

Ce changement de rôle est d'autant plus important pour les petites équipes où l'équipe de modélisation crée ses propres outils de modélisation et donc alterne les rôles.

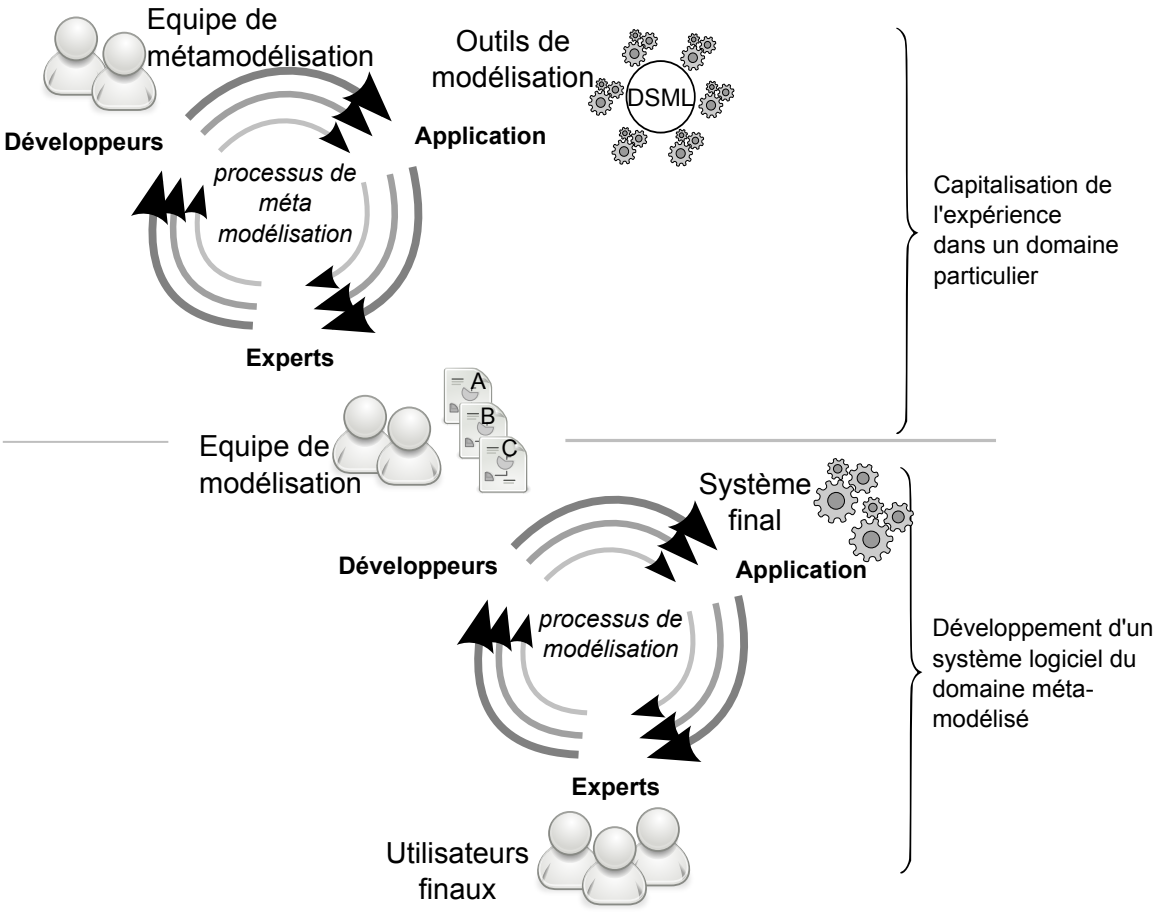


FIGURE 4.1 – Organisation des activités de modélisation et de métamodélisation

4.2 Éléments à construire pour un environnement de modélisation dédié

Pour construire un environnement dédié autour d'un domaine métier, nous retrouverons un ensemble d'outils récurrents. Suivant le domaine et les applications visées, certains de ces outils pourront être des éditeurs, des vérificateurs de modèle, des générateurs de documentations, des simulateurs, des compilateurs, des traducteurs (ou convertisseurs) vers d'autres domaines, des outils de refactoring, des générateurs de suite de tests, etc (cf. figure 4.2). Dans la majorité des cas, il ne sera pas nécessaire de créer tous les outils. Parfois même, certaines versions générées entièrement automatiquement ou faiblement personnalisées seront suffisantes.

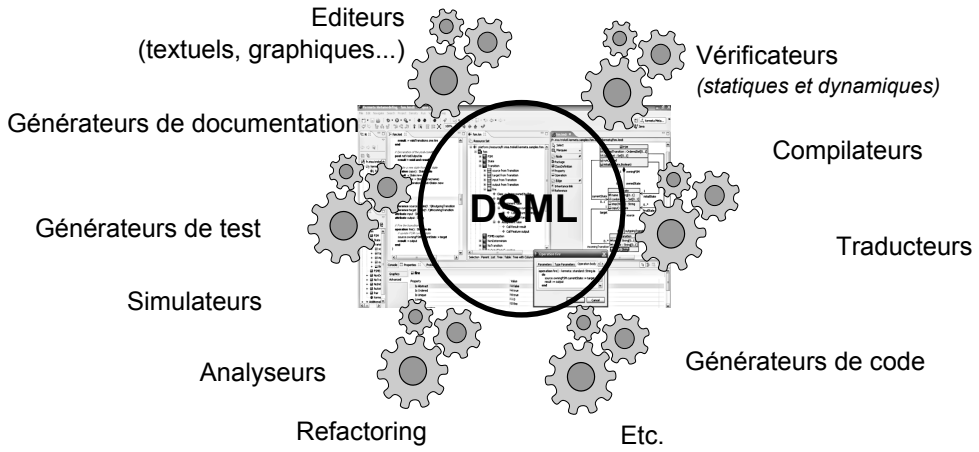


FIGURE 4.2 – Exemples d'outil pour un environnement de modélisation dédié

4.3 Étapes d'un processus de métamodélisation

La conception, la mise au point et l'outillage d'un métamodèle (par l'équipe de métamodélisation) suivent généralement un processus itératif. La figure 4.3 présente ce processus. Elle évoque aussi les relations entre les activités théoriques de la première partie de ce livre et les tâches de développement concrétisées par la réalisation d'outils qui seront détaillés dans cette seconde partie.

L'intérêt de l'aspect itératif ce processus est de pouvoir rapidement s'assurer que le métamodèle correspond au besoin. En outre, le processus présenté peut être adapté librement en fonction des besoins du domaine que l'on cherche à outiller. Exceptée la phase initiale de création du métamodèle, toutes les autres étapes peuvent être omises ou retardées (par exemple, certains DSML utilisés comme représentations intermédiaires dans une chaine de compilation peuvent ne pas avoir immédiatement besoin d'une syntaxe concrète). Grâce à une vision centrée autour du métamodèle, les différents outils qui se rapportent à ce dernier peuvent être conçus séparément en suivant leur propre cycle de vie. Leurs conceptions peuvent donc être confiées à différentes équipes pour être développées en parallèle et peuvent chacune être développées de manière incrémentale. Par exemple, on pourra construire un éditeur et un simulateur dans les premières itérations pour rapidement prototyper le comportement du domaine métier. Dans les itérations suivantes, on pourra développer un compilateur et des éditeurs plus sophistiqués afin d'obtenir de meilleures performances et une facilité d'utilisation accrue.

Le processus de métamodélisation suivra typiquement les étapes suivantes :

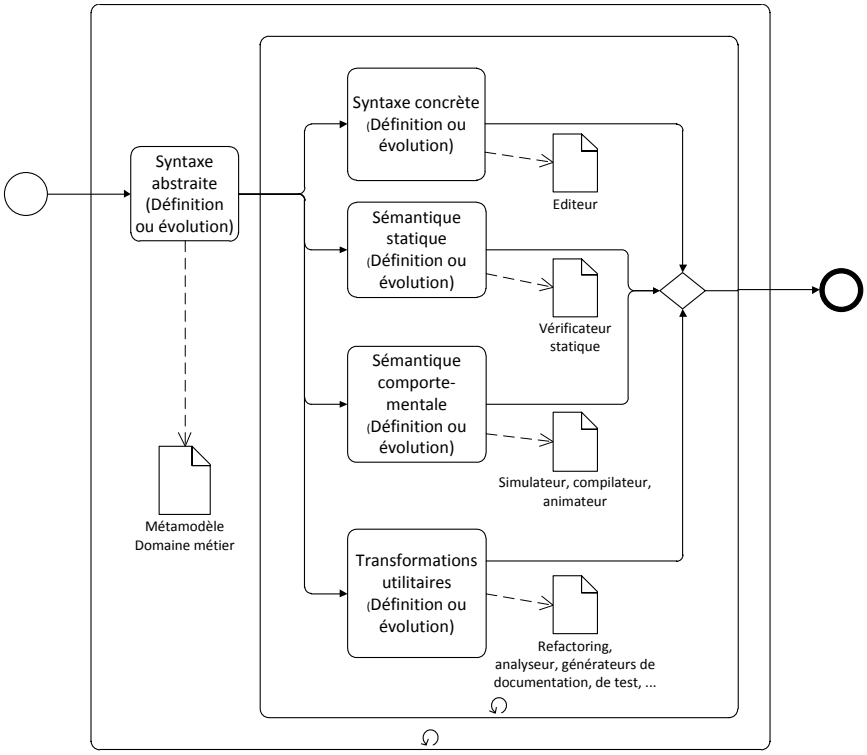


FIGURE 4.3 – Processus de construction d'un environnement de modélisation

— *Définition d'un domaine métier.* Pour construire des outils autour d'un domaine métier, il faut tout d'abord en définir la syntaxe abstraite qui permet d'en exprimer les concepts. Cela se fait typiquement via la création d'un métamodèle. Cette étape est illustrée dans le chapitre 5.

— *Définition de la sémantique statique.* Ensuite, on précisera le métamodèle à l'aide de contraintes dites de "bonne formation". En effet, en exploitant uniquement les syntaxes abstraites et concrètes, il est souvent possible de créer des modèles qui seraient incorrects vis-à-vis de l'intention du domaine. Il s'agit ici de restreindre les modèles autorisés grâce à ces contraintes. De manière pragmatique, on complètera le plus souvent les contraintes au fur et à mesure où l'on rencontrera des modèles invalides plutôt que de chercher l'exhaustivité dès le départ. Cette étape est illustrée dans le chapitre 6.

— *Définition de syntaxes concrètes.* Cette étape consiste à s'assurer de l'expressivité du métamodèle au regard de son objectif. Pour créer aisément les premiers modèles, on pourra générer automatiquement des éditeurs. On privilégiera dans un premier temps les éditeurs entièrement génériques comme l'éditeur arborescent d'EMF ou un éditeur textuel basé sur une syntaxe générique comme le format HUTN [OMG04]. Par la suite, au fur et à mesure que le métamodèle du domaine métier se stabilisera, on construira d'autres éditeurs fournissant une représentation plus adaptée comme un éditeur graphique ou bien une syntaxe textuelle dédiée. Cette étape est illustrée par la pratique dans le chapitre 7.

— *Définition de la sémantique comportementale.* Cette étape consiste à préciser la sémantique comportementale. Cette tâche a pour but d'exprimer le comportement attendu des modèles. Comme vu dans la section 2.4, plusieurs techniques sont possibles, soit par interprétation, soit par compilation vers un autre domaine. Le choix de l'une ou l'autre de ces techniques dépend des objectifs mais également des besoins en termes de performance et de temps de développement. Il faudra en particulier prendre en considération l'existence

d'un domaine cible déjà outillé ayant une sémantique suffisamment proche. Cette étape est illustrée dans le chapitre 8 pour la définition d'un simulateur et dans la section 9.2 pour la définition d'un compilateur.

Dans le cadre d'un processus itératif et en l'absence d'un domaine cible déjà outillé ayant un comportement suffisamment similaire à celui attendu, une approche pragmatique consiste à créer d'abord un simulateur par interprétation directe des modèles (*i.e.,* s'appuyant sur une sémantique opérationnelle). N'étant lié qu'au métamodèle du DSML, celui-ci sera relativement peu coûteux à réaliser et facilement adaptable aux évolutions de sémantique pour les mises au point. Lorsque la sémantique sera stabilisée et si de meilleures performances sont nécessaires, on pourra créer un compilateur pour viser une plateforme d'exécution donnée. En bonus, cela permettra de valider l'implémentation du compilateur en comparant les sorties des deux outils.

— *Définition de transformations utilitaires.* Avec la définition d'une syntaxe abstraite et d'une sémantique, il est maintenant possible d'exploiter ces informations de manière automatique ou semi-automatique. Il existe de nombreux usages qui dans la pratique feront l'objet de transformation de modèles. Parmi les usages courants, on citera par exemple :

- le refactoring (ou réingénierie) [1], par exemple pour faire de l'optimisation, de l'injection de patron de conception ou raffiner des modèles.
- les analyses de modèle serviront à raisonner sur le modèle par exemple pour en extraire des métriques. Ces métriques sont particulièrement intéressantes pour automatiser les phases de test et de validation.
- le tissage et/ou composition de modèle, qui permet d'assembler des modèles ayant des natures homogènes ou hétérogènes.
- la conversion vers d'autres domaines, qui permet de faire des analyses ou des traitements en exploitant les particularités du domaine cible. Ceci est particulièrement utile lorsque l'on souhaite profiter des outillages issus de plusieurs domaines connexes.

- la génération de documentation. Par exemple, de manière à générer automatiquement la documentation ("à la" javadoc) ou des documentations de conception.
- la génération de test (ou *Model-Based Testing* [UL07]), permettant de valider la conception et le système en cours de construction.

Le chapitre 9 montre les techniques permettant de d'implanter de telles transformations. Ainsi, la section 9.2 décrit comment définir compilateur et la section 9.3 illustre la définition d'un mini outil de refactoring.

4.4 Quelques outils supports à l'IDM

Bien qu'il existe de nombreux environnements support à l'IDM, nous illustrerons ici chaque étape du processus en utilisant les outils fonctionnant avec Eclipse [2] et son framework EMF [SBPM09]. Eclipse est un atelier de génie logiciel (AGL) ouvert accueillant de nombreux outils de la communauté IDM. Le framework EMF est la bibliothèque offrant les services liés à création du métamodèle et à sa manipulation en Java. Les autres outils pourront donc s'appuyer sur ces services de base pour traiter les autres aspects du processus.

1. Dans la suite de ce livre nous préfèrerons utiliser le terme anglo-saxon *refactoring* qui est communément utilisé par les développeurs.
2. cf. http://www.eclipse.org

Nous avons sélectionné un ensemble d'outils et de techniques les plus pertinents suivant les critères de qualité didactique, de facilité de mise en œuvre ou de représentativité pour l'activité. Les outils utilisés et présentés dans la suite de ce livre peuvent évidement avoir des alternatives dans Eclipse ou bien leur équivalent dans d'autres environnements.

L'objectif de cette partie est de montrer comment sont conçus les langages de modélisation. Il existe une multitude de ces langages avec éventuellement des variantes qui ne sont souvent pertinentes que pour des spécialistes des domaines visés par ces DSML. Afin de couvrir les différentes étapes du processus sans entrer dans de multiples descriptions de DSML spécifiques, nous utiliserons l'exemple didactique du langage Logo. Néanmoins, tous les DSML existants, qu'ils soient très spécifiques ou bien généralistes comme UML ont suivi un processus de développement similaire à celui présenté. Ainsi, toutes les techniques abordées ici s'appliquent aussi à ces DSML et permet de les compléter en construisant de nouveaux outils.

4.4.1 Eclipse Modeling Project

La fondation Eclipse héberge un ensemble de projets utiles pour la communauté de l'IDM. En particulier, le projet Eclipse Modeling Project [3] est organisé en un ensemble de sous projets fournissant une base pour le processus. La figure 4.4 montre d'ailleurs que plusieurs de ces sous-projets sont regroupés en thèmes correspondants aux activités du processus (*Abstract syntax development*, *Concrete syntax development* et *Model transformation*). On notera que le projet Model Development Tools est pour sa part plutôt orienté vers le support et la fourniture d'outils pour certains langages spécifiques de différentes natures dont OCL et UML. D'autres projets plus exploratoires sont aussi directement hébergés par la fondation dans le sous projet Technology and Research.

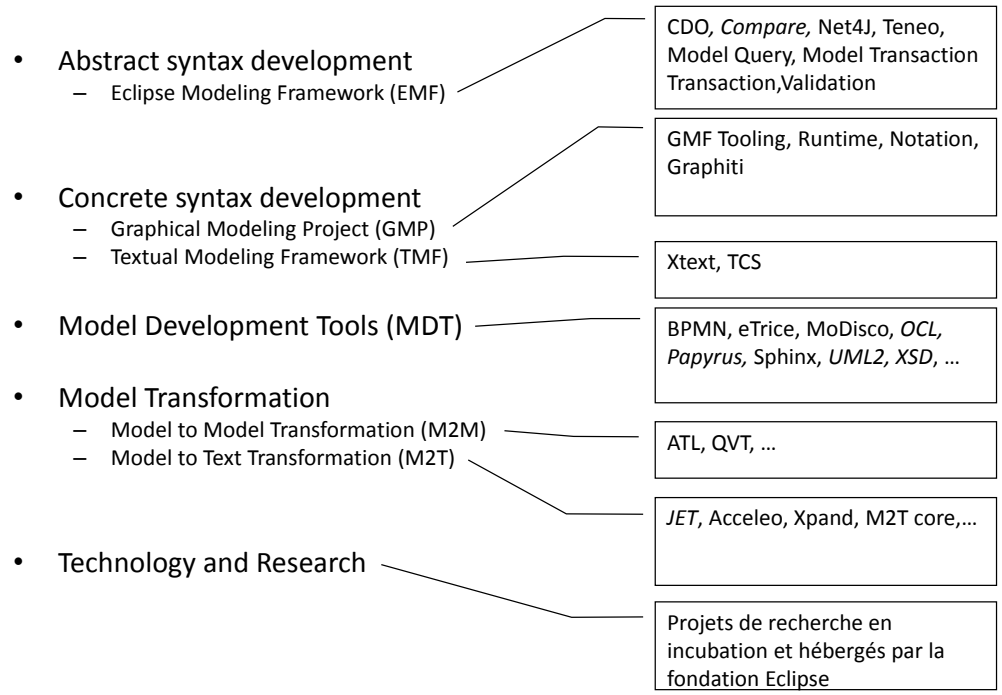


FIGURE 4.4 – Organisation du projet Eclipse Modeling Project

3. cf. http://www.eclipse.org/modeling/

4.4.2 Kermeta

Kermeta workbench [4] est une plate-forme d'ingénierie dirigée par les modèles dont l'objectif est de compléter les outils de modélisation disponibles dans l'environnement Eclipse. Kermeta s'appuie sur deux concepts principaux : l'utilisation d'aspects et l'ajout de l'exécutabilité aux métamodèles.

L'association de ces deux paradigmes permet d'introduire facilement dans des métamodèles existant différentes préoccupations, comme la définition de sémantique statique, la définition de sémantique dynamique, la définition de transformations de modèles, etc. Ces préoccupations pourront éventuellement être exprimées dans des langages hétérogènes.

Grâce aux modes interprété et compilé de Kermeta, il permet non seulement de définir complètement des DSMLs, mais aussi de développer les outils basés sur ces DSMLs, incluant interpréteurs, compilateurs et autres outils de traitement de modèles.

Fonctionnalités clés de Kermeta :

- *Orientation modèle.* Les éléments de modèle sont les concepts clé du langage. Ils peuvent être manipulés aussi aisément que les objet d'un langage orienté objet..
- *Orientation aspect.* Permet de tisser des éléments issus de diverses sources (Ecore, Kermeta, OCL, ...) pour construire des outils basés sur des métamodèles pré-existants. Ajout d'opération et de propriétés directement dans les métaclasses pour des conceptions claires et maintenables.
- *Typage fort.* Le langage dispose de la généricité, de type fonction (par exemple, vous permettant de construire votre propre fonction *foreach*) et un type Modèle. Permet de remonter les erreurs communes dans les phases initiales de processus de développement.
- *Orienté contrat (Design by contract).* Permet d'augmenter la fiabilité des outils.
- *Cœur orienté objet.* Si les fonctionnalités précédentes ne sont pas suffisantes pour une bonne conception, il est toujours possible de les compléter par des patrons de conception issus de la programmation objet. Cela assure de toujours pouvoir trouver une solution qui passe à l'échelle.

4.5 Le format XMI pour la sérialisation des modèles

Afin de sérialiser (i.e. stocker physiquement les modèles sous forme de fichiers) et d'échanger les modèles, il est nécessaire d'avoir un format qui soit compris par l'ensemble des outils qui vont les manipuler. L'OMG a standardisé pour cela le format XMI, pour *XML Metadata Interchange* [OMG11b]. Ce format est en fait un format XML dont les balises et les attributs sont déduits automatiquement à partir des concepts du métamodèle. Par exemple, la figure 4.5 présente le fichier XMI correspondant au modèle représenté graphiquement dans la bulle de droite. Les balises XML *eClassifiers* ou *eStructuralFeatures* sont ici spécifiques au métamodèle utilisé. Dans ce cas, c'est le métamodèle Ecore (qui sera abordé dans la section 5.1.1).

Le fichier XML produit est certes du texte, mais il n'est guère lisible pour un humain et on lui préfèrera des syntaxes plus dédiées comme celles que nous aborderons dans les chapitres suivants.

4. cf. http://www.kermeta.org/

FIGURE 4.5 – Exemple de fichier au format XMI

Notons qu'il est important de distinguer la notion de modèle avec celle de diagramme. Le diagramme est une vue graphique sur le modèle. A ce titre, elle peut ne pas représenter la totalité du modèle mais présente les éléments du modèle de manière lisible. Plusieurs vues graphiques peuvent aussi être utilisées pour présenter un même modèle. Le format XMI est fait pour échanger des modèles. On l'assimile souvent à tord à de l'échange de diagrammes. Dans le premier cas, on parle bien d'échanger des concepts, alors que dans le second cas, on échange des informations sur la présentation de ces concepts.

Pour l'échange des diagrammes, l'OMG a standardisé le format DI ou Diagram Interchange [OMG06b]. Ce format dispose d'un métamodèle particulier pour capturer les concepts de positionnement graphique des éléments sur les diagrammes [5]. Malheureusement, si l'on arrive généralement à échanger les modèles via le format XMI, très peu de vendeurs d'outils implémentent correctement la partie DI et l'on perd souvent la présentation des modèles (i.e. les diagrammes) lorsque l'on transfère des modèles d'un outil à l'autre.

5. Initialement prévu pour les diagrammes d'UML, le format DI est applicable avec d'autres diagrammes que ceux d'UML.

Chapitre 5

Définition d'un domaine métier

L'IDM propose de capitaliser les connaissances d'un domaine métier en définissant un langage dédié (ou DSML) et un ensemble d'outils s'appuyant sur la définition du langage pour automatiser certaines tâches du développement. Le langage est généralement décrit à l'aide d'un modèle, appelé métamodèle, qui devient alors l'élément central de toutes les activités de métamodélisation et des outils développés pour systématiser certaines étapes du développement de systèmes. C'est pourquoi il est primordial d'apporter un soin particulier à la définition du métamodèle, au même titre que l'activité de modélisation dans la conception des systèmes eux-mêmes. Ce chapitre présente les langages et outils, dit de métamodélisation, disponibles pour la définition d'un métamodèle.

5.1 Langages de métamodélisation

5.1.1 Le langage de métamodélisation Ecore

Le standard de fait pour la création de métamodèle dans l'environnement Eclipse est l'Eclipse Modeling Framework (EMF) qui propose en particulier le langage de métamodélisation Ecore. Même si Ecore n'est pas une stricte implémentation du MOF tel que spécifié par l'OMG, il s'en rapproche suffisamment pour que dans la suite de ce livre nous ne parlions plus que de Ecore. En effet, tout comme le standard MOF, Ecore supporte une définition orientée objet des métamodèles en termes de packages, de classes, de propriétés et de signatures d'opérations, enrichie de constructions plus spécifiques aux modèles comme la notion d'associations entre classes ou de composition. Ce langage est ainsi suffisant pour spécifier la structure des métamodèles.

En pratique, ce format Ecore joue un rôle clé pour les développements de DSML car c'est à partir des informations issues des métamodèles écrits en Ecore que seront construits les outils qui manipulent les modèles. Par exemple, le framework EMF exploite des métamodèles Ecore pour charger et sauver les modèles au format XMI.

Dans Eclipse, plusieurs éditeurs permettent de créer des modèles en Ecore. Tout d'abord, on pourra utiliser l'éditeur arborescent de base fourni avec EMF (éditeur à gauche dans la figure 5.1). Celui-ci permet de représenter, créer et supprimer les éléments d'un modèle sous la forme d'un arbre. Les informations de chaque élément peuvent être éditées à l'aide de la vue *Properties* d'Eclipse après avoir sélectionné l'élément dans l'arbre. Il est fonctionnel et conviendra surtout pour naviguer à l'intérieur d'un métamodèle en suivant l'arbre de contenance, mais il reste plutôt rudimentaire.

Arbre de contenance : Un modèle est un ensemble d'éléments reliés entre eux grâce aux associations sous la forme d'un graphe. Parmi ces associations, les associations de composition (représentées avec un diamant noir dans les diagrammes de classe du métamodèle) indiquent comment les éléments se contiennent entre eux. Cela définit un sous-graphe dans les modèles qui a la propriété de ne pas contenir de cycle que l'on appellera arbre de contenance. Les propriétés particulières de cet arbre seront par la suite exploitées par de nombreux outils.

Pour une édition plus avancée des métamodèles, on utilisera de préférence l'éditeur graphique fourni par le projet Eclipse *Ecore Tools*[1]. Cet éditeur offre la possibilité d'éditer graphiquement un métamodèle sous la forme d'un diagramme de classe (au centre dans la figure 5.1). Pour cela cet éditeur complète le métamodèle sérialisé au format Ecore (extension *.ecore*) avec un second fichier ayant l'extension *.ecorediag* et stockant les informations du diagramme, c-a-d., de la vue graphique.

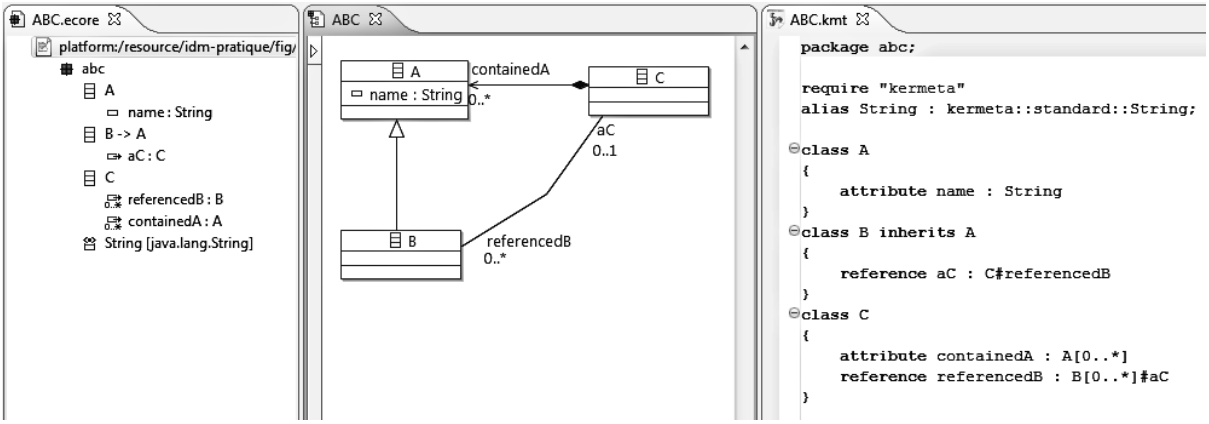


FIGURE 5.1 – Syntaxes arborescente, graphique et textuelle d'Ecore

Notons qu'il est possible d'obtenir un métamodèle au format Ecore à partir de sources externes. Par exemple, on pourra convertir un schéma XML (fichiers **.xsd*), convertir des profils UML, etc. Néanmoins, suivant les cas, le fichier généré au format Ecore peut contenir des annotations spécifiques qui en alourdissent l'exploitation. Les concepts exposés dans le métamodèle Ecore obtenus sont organisés suivant une structure dérivée du format d'entrée. Cette structure sera moins naturelle que si on avait conçu directement ce métamodèle en utilisant les concepts du domaine métier visé. C'est par exemple le cas des schémas XML qui autorisent des données mixtes[2]. Quand ils sont traduits en Ecore, ces métamodèles utilisent une structure de contenance très plate (proche de la syntaxe XML d'origine) qui est complétée avec propriétés dérivées (ou calculées) pour capturer les données mixtes. Ils sont donc assez peu pratiques à utiliser dans le cadre de manipulation de modèle. En contrepartie, ils autorisent une sérialisation des modèles vers un format XML conforme au schéma XML d'origine plutôt que du vers format XMI.

5.1.2 Le langage de métamodélisation exécutable Kermeta

Syntaxe textuelle. Un métamodèle peut également être décrit textuellement, par exemple en utilisant la syntaxe textuelle et l'éditeur de Kermeta (partie droite de la figure 5.1). Kermeta

1. Eclipse Ecore Tools project : http://www.eclipse.org/modeling/emft/?project=ecoretools
2. mélange de balises et de texte comme <mabalise> un peu de texte <sousbalise></sousbalise> encore un peu de texte </mabalise>

propose de sauvegarder la description textuelle d'un métamodèle dans un simple fichier texte avec l'extension *.kmt*. L'environnement fournit ensuite les transformations permettant de traduire un fichier du format Kermeta (*.kmt*) au format Ecore, et inversement (*.ecore*).

Modularisation et séparation des préoccupations. Parmi les fonctionnalités de Kermeta, les opérateurs *require* et *aspect* jouent un rôle important car ils permettent de mettre en relation les préoccupations de conception de DSML (syntaxe abstraite, sémantique statique, sémantique comportementale, ...) en complétant un métamodèle existant avec de nouveaux éléments.

Le mot clé *aspect* offre un mécanisme proche de celui connu sous le nom de *classe ouverte* (ou *open-class* en anglais) [CL00] ou d'introduction statique. Il permet au concepteur de réouvrir une classe créée précédemment et lui ajouter de nouvelles informations telles que de nouvelles propriétés, contraintes et opérations.

L'opérateur de composition statique *require* assure l'assemblage de différentes unités (*i.e.,* fichiers) et permet de les composer automatiquement dans un unique métamodèle. La composition est effectuée statiquement et le modèle composé subit une vérification de type pour garantir la bonne intégration de toutes ces unités.

Ces deux mécanismes sont utilisés pour organiser des préoccupations transverses au métamodèle qu'elles concernent, ce qui est un trait caractéristique de la programmation orientée aspect. Dans ce cas, on exploite la possibilité d'étendre et compléter une métaclasse sans modifier physiquement le métamodèle pour lui ajouter une nouvelle préoccupation ainsi que la possibilité de définir de nouveaux paquetages et de nouvelles classes. Cela permet ainsi de réutiliser facilement des métamodèles existants, ou de découper des métamodèles en composants réutilisables et modulaires.

Par exemple, le listing 5.1 complète le métamodèle Ecore présenté sur la figure 5.1 en ajoutant une nouvelle métaclasse D et en introduisant un lien bidirectionnel entre A et D.

Listing 5.1 – Ajout par aspect de nouvelles préoccupations

```
1  package abc;
2  require "ABC.ecore" // importe le métamodèle
3  aspect class A {
4     attribute aD : D[0..*]#aA
5  }
6  class D {
7    reference aA : A[0..1]#aD
8  }
```

Nous verrons par la suite que le mécanisme de composition offert par l'opérateur *require* peut aussi être utilisé pour offrir un certain degré de flexibilité dans la construction des outils d'un environnement de modélisation.

Cependant, Kermeta n'est pas juste une représentation textuelle flexible de Ecore. En effet, si Ecore est suffisant pour spécifier la structure des métamodèles, il n'inclut pas de concepts permettant la définition de contraintes, et ne permet pas de définir de sémantique opérationnelle : les opérations en Ecore ne contiennent pas de corps. Nous verrons dans la section 8.1 que Kermeta ajoute la notion d'exécutabilité. Il peut ainsi être vu comme une extension de Ecore avec un langage d'action impératif (orienté objet) pour spécifier des contraintes et des corps d'opérations dans les métamodèles.

5.2 Exemple de domaine métier : Logo

Pour illustrer l'approche proposée ici, nous allons utiliser l'exemple du langage Logo. La raison du choix de cet exemple est que Logo est un langage de programmation à la fois très puissant (*i.e.,* Turing-complet) et très simple, car il a été créé à l'origine pour des objectifs d'apprentissage de l'informatique par de jeunes enfants. L'exécution d'un programme Logo permet de diriger une tortue sur un écran et de lui faire tracer des figures géométriques quand son crayon est baissé [3].

Le listing 5.2 présente un exemple de programme Logo qui dessine un carré. Pour cela, le programme définit tout d'abord une procédure (square) dessinant un carré dont la longueur des côtés correspond au paramètre de la procédure. L'instruction clear permet ensuite d'effacer l'écran. Les instructions suivante permettent dans l'ordre de baisser le stylo de la tortue (ligne 13), d'appeler la procédure dessinant un carré (ligne 14), et de lever le stylo de la tortue (ligne 15).

Listing 5.2 – Algorithme traçant un *carré* en logo

```
1  # definition of the square procedure
2  to square :size
3    repeat 4 [
4      forward :size
5      right 90
6    ]
7  end
8
9  # clear screen
10 clear
11 # draw a square
12 pendown
13 square(50)
14 penup
```

Dans la suite de cette partie, nous appliquerons les techniques de l'ingénierie dirigée par les modèles à cet exemple. Le déroulement de ce processus permettra de construire un environnement complet pour le langage Logo.

Vous pouvez expérimenter la plupart des exemples et outils autours du langage Logo qui sont présentés dans ce livre en installant l'exemple Logo fourni avec l'environnement Eclipse de Kermeta. Ils sont présentés plus en détail (en anglais) dans un tutoriel dédié http://www.kermeta.org/documents/tutorials/building_dsl_tutorials/.

5.3 Metamodélisation de Logo

La première étape d'un processus de métamodélisation consiste à définir la syntaxe abstraite du langage que l'on cherche à créer sous la forme d'un métamodèle au format Ecore. La syntaxe abstraite capture les concepts du langage (dans le cas de Logo les instructions primitives, les expressions, les structures de contrôle, les définitions de procédures, etc.) ainsi que les relations entre ceux-ci (p.-ex., une expression est soit une constante, soit une expression binaire qui elle-même contient deux expressions).

A l'aide de ce métamodèle il sera ensuite possible d'éditer, charger et sauvegarder automatiquement des modèles qui s'y conformes (*i.e.,* des programmes Logo).

3. une histoire complète du langage Logo ainsi que de nombreux exemples de code peuvent être trouvés sur wikipedia (http://en.wikipedia.org/wiki/Logo_(programming_language))

La figure 5.2 présente un métamodèle pour la syntaxe abstraite du langage Logo. Celui-ci inclut comme concepts :

— des instructions primitives (*Forward*, *Back*, *Left*, *Right*, *PenUp* et *PenDown*). Les instructions permettent de déplacer et de faire pivoter la tortue Logo, ainsi que de contrôler son crayon.

— des expressions arithmétiques (*Constant*, *BinaryExp* et leurs sous-classes). Dans notre version simple de Logo, les constantes sont des entiers et les opérateurs traitent uniquement des entiers.

— Procédures (*ProcDeclaration*, *ProcCall*, *Parameter* et *ParameterCall*) qui permettent de définir et d'appeler des fonctions avec des paramètres (notez que la récursivité est supportée en Logo)

— des structures de contrôle comme classiquement dans les langages de programmation impérative : la séquence (*Block*), la conditionnelle (*If*), la répétition d'une action un nombre donné de fois (*Repeat*), et la boucle non bornée (*While*).

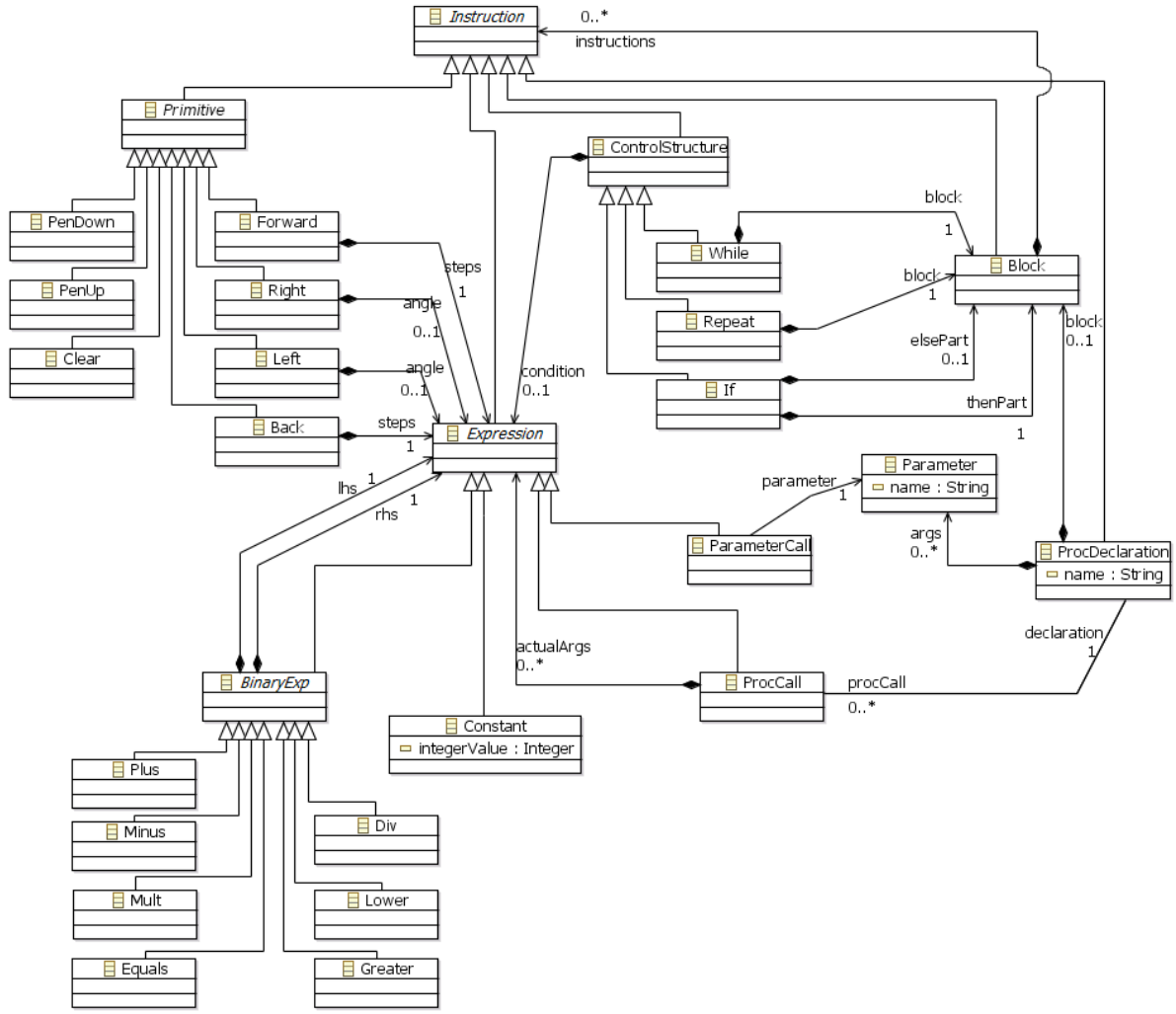


FIGURE 5.2 – Métamodèle (syntaxe abstraite) du langage Logo

En pratique, ces concepts correspondent aux instructions que l'on souhaite faire exécuter à une tortue. Par exemple, lever le crayon, avancer d'un nombre (entier) donné de pas ou tourner

d'un certain angle (exprimé, pour simplifier, comme un nombre entier de degrés).

Les instances de ces classes représenteront les instructions du programme Logo de l'utilisateur, par exemple *FORWARD 10* ou *RIGHT 3*30*. Les classes *Forward*, et *Right* étant paramétrées par une expression arithmétique, elles sont reliées à une classe *Expression*, qui peut être, par exemple, une expression binaire (*i.e.* qui compose deux expressions par un opérateur de type +, *, etc.) ou une constante entière, ce qui est représenté par une relation d'héritage.

Dès que le métamodèle de la figure 5.2 est défini, il est possible de construire des modèles qui s'y conforme (*i.e.,* des programmes Logo) avec des éditeurs génériques comme ceux que nous aborderons dans la section 7.1. Ces éditeurs simples permettront d'expérimenter l'expressivité du métamodèle pour le mettre au point.

5.4 Bonnes pratiques

Cette section présente quelques éléments à prendre en compte lors de la définition d'un domaine métier.

5.4.1 Arbre de contenance

Comme nous l'avons vu, les liens de composition dans un métamodèle définissent un arbre de contenance. Bien que non obligatoire, il est important de définir clairement cet arbre lors de la définition d'un métamodèle pour profiter au mieux de l'approche IDM.

Tout d'abord, cette notion de contenance est utilisée pour la sérialisation des modèles. Cela peux avoir un impact sur la facilité de découper un modèle en plusieurs unités (ou fichiers). Par exemple, avec un élément de modèle ayant pour rôle de conteneur racine, la sauvegarde de l'ensemble des éléments d'un modèle sera facilitée puisqu'il suffit d'ajouter l'élément racine à un fichier plutôt que d'ajouter tous les éléments du modèle un à un dans le fichier. De même, il sera plus simple de déplacer des sous-ensembles d'un modèle s'il existe une hiérarchie de conteneurs pour matérialiser ces sous-ensembles.

Ensuite, les propriétés particulières de l'arbre sont exploitées par de nombreux outils et en particulier les outils automatiques. Les exemples les plus courants sont les outils permettant de donner automatiquement une représentation visuelle des modèles. Plusieurs exemples d'éditeurs génériques (arborescents ou textuels) seront présentés dans le chapitre 7 et exploiteront cette information pour produire des syntaxes génériques directement exploitable. Les outils de composition ou de tissage de modèle génériques exploiteront aussi l'arbre de contenance pour déterminer les éléments de modèle à fusionner. De même, suivant les besoins, les fonctions de vérifications de modèle (comme la fonction *checkAllInvariants* vue au chapitre 6) pourront être appliquées sélectivement à un sous-ensemble du modèle plutôt qu'à tout celui-ci. Définir une structure hiérarchique pour l'arbre de contenance permettra d'éviter un travail supplémentaire d'adaptation de ces outils automatiques au métamodèle en cours de définition.

Plus généralement les différentes transformations de modèle exploiteront le fait qu'un patron visiteur appliqué à un arbre va garantir que l'on visitera tous les éléments du modèle une seule et unique fois. Ceci permettra ensuite de générer pour chacun de ces éléments le ou les éléments dans les modèles cibles.

5.4.2 Composition/extension de domaine métier

Dans un souci de réutilisabilité, il est préférable de composer ou d'étendre des métamodèles existants. De manière assez classique, on exploitera les caractéristiques orientées object des métamodèles. Par exemple, ou pourra réaliser certaines extensions par ajout de métaclasses qui

héritent du domaine de base. En général, ces extensions ne seront possibles que si le métamodèle de base a été conçu en prévision de celles-ci. Les patrons de conception tels que *Décorateur*, *Fabrique*,... permettront d'aménager élégamment des points d'extension.

Il est parfois impossible d'étendre un métamodèle si on n'en est pas maître. Dans ce cas, des techniques de compositions de modèle telles que vues dans la section 3.2 pourront aider à résoudre ce problème.

Il arrive parfois que l'on crée plusieurs métamodèles de manière séparée en vue de les composer et les utiliser conjointement. Ce sera le cas par exemple pour construire un DSML supportant de nombreux concepts. Une erreur courante est de créer des dépendances fortes entre les différents métamodèles. Par exemple, la création d'une association bidirectionnelle entre deux métaclasses de deux métamodèles va les lier fortement. Il sera alors impossible de créer un modèle conforme à seulement l'un des deux métamodèles.

La navigation inverse d'un lien unidirectionnel peut être nécessaire pour une application donnée. L'approche classique serait d'utiliser une table de hachage, néanmoins, il est préférable d'exploiter la possibilité d'ajouter ce lien inverse par aspect. C'est par exemple possible en utilisant des outils tels que Kermeta ou le récent projet EMF Facet [4]. Ainsi, ce lien sera non intrusif pour la conception des métamodèles et disponible uniquement dans le cadre de cette application.

5.4.3 Autres bonnes pratiques issues de la modélisation objet

De manière générale, les patrons de conception comme ceux issus du GOF [GHJV95], voire les antipatrons [5], seront une bonne inspiration pour des conceptions efficaces des métamodèles. Pour le nommage des méta-éléments, et plus généralement pour d'autres considérations de style, il est possible de s'inspirer des bonnes pratiques qui s'appliquent aux diagrammes de classe d'UML. Par exemple http://www.agilemodeling.com/style/classDiagram.htm.

4. cf. http://www.eclipse.org/modeling/emft/facet/
5. voir par exemple : http://www.irit.fr/GOPROD

Chapitre 6

Vérification statique de modèle par tissage de la sémantique statique

Un métamodèle peut être vu comme un moyen de définir l'ensemble des configurations admissibles d'un graphe d'objets. Toutes les structures sont représentées comme des classes, des relations, ainsi que des propriétés structurelles. Dans l'IDM, un métamodèle définit donc un ensemble de modèles valides et donc dans notre exemple des programmes Logo valides. Cependant, certaines contraintes ne peuvent pas directement être exprimées en utilisant un langage de métamodélisation tel que MOF. Par exemple, il n'y a pas de moyen simple d'exprimer que les noms des paramètres formels doivent être différents les uns des autres lorsqu'on déclare des procédures, ou encore qu'un programme Logo valide demande à ce que le nombre d'arguments effectifs passés lors de l'appel d'une procédure soit le même que le nombre d'arguments formels dans la déclaration. Ces contraintes font partie de ce qu'on appelle généralement la sémantique statique du langage (cf. sec. 2.4), et sont majoritairement exprimées à l'aide du langage OCL (cf. sec. 2.2.2).

Ce chapitre présente les moyens de compléter un métamodèle Ecore par un ensemble de contraintes OCL, et de les vérifier sur des modèles. Pour cela, les différents outils (éditeurs, vérificateurs) sont illustrés par l'expression des contraintes OCL du métamodèle Logo et la vérification de ces contraintes sur des programmes Logo.

6.1 Utilisation d'OCL dans la métamodélisation

OCL (*Object Constraint Language*) [RG02] est défini par l'OMG pour exprimer des contraintes sous la forme de formules du premier ordre sur des métamodèles objet décrit à l'aide du MOF. OCL permet ainsi d'exprimer précisément la sémantique statique du domaine que l'on souhaite capturer dans la syntaxe du DSML. OCL a été initialement développé au sein d'IBM pour exprimer des contraintes sur des modèles UML. C'est aujourd'hui un langage textuel relativement simple pour exprimer des contraintes et des requêtes sur n'importe quel modèle ou métamodèle utilisant l'infrastructure MOF lorsque celles-ci ne peuvent facilement être exprimées dans un diagramme.

Les implémentations permettant de vérifier des contraintes OCL sont parfois liées à un langage. C'est à dire, que le moteur de vérification est capable de vérifier les contraintes seulement si elles sont appliquées à un métamodèle donné. Le plus souvent, ce sera UML.

Une expression OCL est décrite par les éléments suivants :

un contexte qui définit la situation dans laquelle la contrainte doit être valide. Si le contexte est une des métaclasses du DSML, la contrainte sera appliquée à l'ensemble des instances de cette métaclasse. Si le contexte est une propriété d'une métaclasse (un attribut ou une opération), la contrainte sera évaluée sur toutes les instances de la métaclasse portant cet attribut mais ne s'appliquera qu'à cette propriété.

un contrat qui défini l'expression qui devra être respectée. Cette expression pourra utiliser des opérations de navigations pour accéder aux informations du modèle. L'expression pourra aussi utiliser des opérateurs par exemple pour spécifier des expressions conditionnelles.

OCL propose plusieurs types de contrats : *invariant*, *précondition*, *postcondition*, *dérivation d'attribut*, *initialisation d'attribut*, et *résultat d'opération*, exprimés respectivement avec les mots-clés *inv*, *pre*, *post*, *derive*, *init* et *body*.

Le type de contrat le plus fréquent et le plus utile dans le cadre de la métamodélisation est l'*invariant*. Comme expliqué dans la section 2.2.2, ce type de contrat exploite une expression booléenne pour exprimer des règles de bonne formation de nos modèles. Ces règles seront alors exploitées pour construire des vérificateurs de modèle.

Les *pré* et *postconditions* sont aussi des expressions booléennes qui doivent s'appliquer respectivement avant et après l'exécution d'une opération pour vérifier cette dernière. Dans notre processus, nous les utiliserons pour améliorer les définitions des opérations définies dans le cadre des simulateurs (vus dans le chapitre 8) et des transformations de modèle (vues dans le chapitre 9) en les complétant avec de la programmation par contrat[Mey92].

La *dérivation d'attribut* permet de définir un attribut calculé à partir d'autres informations dans le modèle. Cet attribut peut ainsi être utilisé normalement sans être stocké dans le modèle mais en étant calculé lors de son utilisation.

L'*initialisation d'attribut*, comme son nom l'indique, est une expression définissant la valeur initiale d'un attribut.

Le *résultat d'opération* permet de spécifier le retour d'une opération. Cette expression peut se ramener à une postcondition portant sur le résultat de l'opération.

6.2 Expression de la sémantique statique de Logo en OCL

Le métamodèle Logo de la figure 5.2 ne définit en fait que la structure générale des programmes Logo. Parmi les modèles qu'il permet de créer, seuls certains d'entre eux peuvent être considérés comme des programmes Logo valides vis-à-vis de la sémantique du langage. Nous allons donc attacher un ensemble de contraintes OCL à la syntaxe abstraite afin de réduire l'ensemble admissible des modèles qui s'y conforme.

Le listing 6.1 en OCL décrit deux contraintes attachées au métamodèle de Logo. La première est un invariant pour la classe *ProcCall* qui vérifie que le nombre d'arguments effectifs passés lors d'un appel est égal au nombre d'arguments formels attendus par la déclaration de la procédure. Pour cela, en considérant comme contexte la classe *ProcCall* on accède à son attribut *actualArgs* pour en connaitre la taille. On navigue aussi son attribut *declaration* pour obtenir un *ProcDecl* et lui demander la taille de son attribut *args*.

Le second invariant est attaché à la classe *ProcDeclaration* et assure que les noms des paramètres formels dans une déclaration de procédure sont unique et donc tous différents les uns des autres. Cet invariant est exprimé dans le contexte de la classe *ProcDeclaration* et compare

Listing 6.2 – Tissage de la sémantique statique dans le métamodèle Logo

```
1  package kmLogo;
2  require "ASMLogo.ecore"       // import Logo metamodel (ie. abstract syntax)
3  require "StaticSemantics.ocl" // import OCL constraints definitions
4  [. . .]
5  class Main {
6    operation Main(): Void is do
7      // Load a Logo model and get the main block instruction
8      var rootInstruction : Instruction init load("../models/carre.logo")
9      // check the model
10     rootInstruction.checkAllInvariants()
11   end
12   operation load(uri : String) : Instruction is do
13     [. .]  // load the root model element from an XMI file
14   end
15 end
```

tous les paramètres de la procédure (*args*) deux à deux afin de vérifier que si leurs noms sont égaux alors il s'agit des même paramètres.

Listing 6.1 – Contraintes OCL du métamodèle Logo

```
1  package kmLogo::ASM
2
3  context ProcCall
4  inv same_number_of_formals_and_actuals :
5    actualArgs->size() = declaration.args->size()
6
7  context ProcDeclaration
8  inv unique_names_for_formal_arguments :
9    args->forAll ( a1, a2 | a1.name = a2.name implies a1 = a2 )
10
11 endpackage
```

6.3 Tissage et vérification de la sémantique statique de Logo dans son métamodèle

Dans l'environnement Kermeta, les contraintes OCL peuvent être tissées directement dans le métamodèle, ce qui permet de les vérifier ensuite sur chaque modèle conforme à ce métamodèle. En pratique, une fois qu'un concepteur a créé un métamodèle en Ecore (et qui sera nommé par exemple ASMLogo.ecore), il peut l'importer dans Kermeta (voir la ligne 2 dans le listing 6.2) en utilisant l'instruction *require* comme décrit au paragraphe 5.1.2. En supposant que la sémantique statique de Logo (décrite dans le listing 6.1) soit placée dans un fichier appelé StaticSemantics.ocl, alors la même instruction *require* peut être utilisée en Kermeta pour tisser la sémantique statique de Logo dans le métamodèle décrivant la syntaxe abstraite du langage (voir ligne 3 dans le listing 6.2).

L'intégration d'OCL dans Kermeta s'appuie sur deux fonctionnalités :

— support natif de la conception par contrat [Mey92] qui, appliqué au niveau du métamodèle, permet d'exprimer des invariants dans les classes du métamodèle, et des pré- et post-conditions dans les méthodes des classes du métamodèle. L'évaluation de ces contraintes peut se faire soit de manière programmatique, soit depuis l'interface graphique.

— support de la syntaxe concrète d'OCL implantée comme une transformation de modèles (écrite en Kermeta) depuis le métamodèle d'OCL vers le métamodèle de Kermeta. Le

résultat de cette transformation est automatiquement fusionné avec le métamodèle cible, en utilisant à nouveau le mécanisme d'introduction statique offert par Kermeta.

Kermeta permet à l'utilisateur de choisir quand les contraintes doivent être vérifiées. On pourra vérifier une sélection de contraintes ou bien toutes celles d'un modèle complet avec respectivement les primitives *checkInvariant* (qui vérifie un invariant particulier sur une classe) ou *checkAllInvariants* (qui vérifie l'ensemble des invariants applicables à un élément du modèle ainsi qu'aux éléments qu'il contient via son arbre de contenance) [1]. Les contraintes portant sur les opérations (préconditions, et postconditions) sont vérifiées de manière optionnelle en fonction du type d'exécution qui est choisi depuis le menu Eclipse : exécution normale, ou exécution avec vérification de contraintes.

Un éditeur de modèle s'appuyant uniquement sur le métamodèle Logo permet de créer des programmes invalides. Néanmoins, ces programmes peuvent être validés vis-à-vis des contraintes de la sémantique statique du langage Logo au sein de l'environnement Kermeta. Par exemple si nous modifions le programme Logo de la figure 5.2 en appelant square(50,10) à la place de square(50), et nous chargeons ce programme dans Kermeta (lignes 8 du listing 6.1), alors en appelant *checkAllInvariants* (ligne 10 du listing 6.1) nous obtenons le message d'erreur suivant :

```
\index{invariant@invariant}%
Invariant same_number_of_formals_and_actuals
  has been violated for: square(50,10)
```

6.4 Intégration du vérificateur statique de modèle dans un AGL

La remontée des contraintes qui ne sont pas respectées vers l'IHM de l'utilisateur final est similaire à de la remontée d'erreur. Celle-ci est très spécifique de l'AGL utilisé et des habitudes de l'utilisateur.

Dans le cas de Kermeta les invariants non respectés sont remontés soit individuellement sous forme d'une exception *ConstraintViolatedException*, soit de manière groupée sous forme d'un modèle de diagnostique. Dans les deux cas, il est ensuite possible d'obtenir l'élément de modèle mis en cause et l'invariant. Avec ces informations, on pourra soit composer un message texte pour l'utilisateur en mode ligne de commande, soit intégrer l'appel de *checkAllInvariants* dans l'IHM pour valider un modèle directement depuis l'éditeur. Les erreurs pouvant alors être présentées par mise en évidence des éléments concernés dans l'éditeur de modèle ou bien dans une vue séparée.

Dans le cadre d'Eclipse, cette intégration pourra par exemple être réalisée en s'appuyant sur l'EMF Validation Framework [2]. Ce framework offre différentes stratégies de déclenchement des vérifications, ce qui permet de bien l'insérer dans l'environnement graphique. On pourra suivant les cas, préférer avoir des vérifications à la volée lors de chaque changement dans le modèle, ou bien préférer lancer les vérifications uniquement à la demande de l'utilisateur.

L'un des autres intérêts de ce framework est qu'il permet aussi d'harmoniser les retours issus de différents moteurs de vérification. En effet, ces moteurs ont des spécificités les rendant utiles dans différentes situations. Certains pourront être généralistes ou bien spécifiques à un ou plusieurs métamodèles. Le langage utilisé pourra dans certains cas être différent d'OCL, par exemple en utilisant la programmation logique "à la Prolog" ou bien directement dans un langage de programmation impératif généraliste comme Java. Enfin, certaines implémentations

1. Il suffit donc d'appliquer la primitive *checkAllInvariants* sur l'élément racine de l'arbre de contenance d'un modèle pour vérifier le modèle complet.
2. cf. http://www.eclipse.org/modeling/emf/?project=validation

pourront interpréter les contraintes ou bien dans d'autre cas, pourront les compiler en code natif (Java dans le cas d'Eclipse) pour plus de performance.

Chapitre 7

Édition de modèle (arborescente, graphique et textuelle)

Comme vu précédemment, afin de pouvoir éditer des modèles conformes à un métamodèle, il est nécessaire de compléter la description de la syntaxe abstraite décrire par le métamodèle par la description de la syntaxe concrète. En s'appuyant sur celle-ci il est ainsi possible d'obtenir (semi-)automatiquement un ou plusieurs éditeurs de modèle.

En fonction des besoins (expressivité du langage, public visé, etc.) un modèle pourra être présenté de différentes façons. Classiquement, on pourra utiliser des syntaxes concrètes graphiques pour représenter le modèle sous forme de diagrammes ou d'arbres. Il est également possible de décrire une syntaxe concrète textuelle permettant de manipuler un modèle sous la forme d'un fichier texte. Enfin, il est possible d'imaginer n'importe qu'elle autre représentation du moment qu'elle a un sens pour l'utilisateur du langage (p.ex., sous forme de tableau, de formulaire, etc.). L'idéal est généralement d'utiliser la syntaxe qui soit la plus expressive et simple d'utilisation pour l'utilisateur final. Il est également possible d'outiller ces syntaxes en fournissant différents éditeurs plus ou moins sophistiqués en fonction des besoins.

On peut non seulement utiliser plusieurs syntaxes pour un même métamodèle, mais aussi avec certains outils, les combiner. Par exemple, ou pourra utiliser une représentation graphique pour certaines métaclasses, et une représentation textuelle pour d'autres.

Si l'on utilise des techniques *traditionnelles*, la définition et l'outillage pour ces syntaxes est souvent un processus assez coûteux du fait qu'il est nécessaire de se familiariser avec chacune des techniques dédiées et connaitre les API spécifiques pour l'intégration à un AGL. Par exemple, Lex/Yacc, SableCC, Antlr pour *parser* du texte ; patron visiteur pour traduire le modèle vers du texte ; Swing ou GEF (Eclipse Graphical Editing Framework) [1] pour définir l'interface graphique, ...

Heureusement, certains spécialistes du domaine se sont penchés sur la question et ont construit des langages dédiés pour nous assister dans cette tâche. Nous allons voir ici comment ce processus est simplifié en exploitant les capacités génératives de l'IDM. Dans les cas idéaux, certaines de ces syntaxes peuvent même être entièrement obtenues à partir de la structure du métamodèle traité.

1. cf. http://www.eclipse.org/gef/

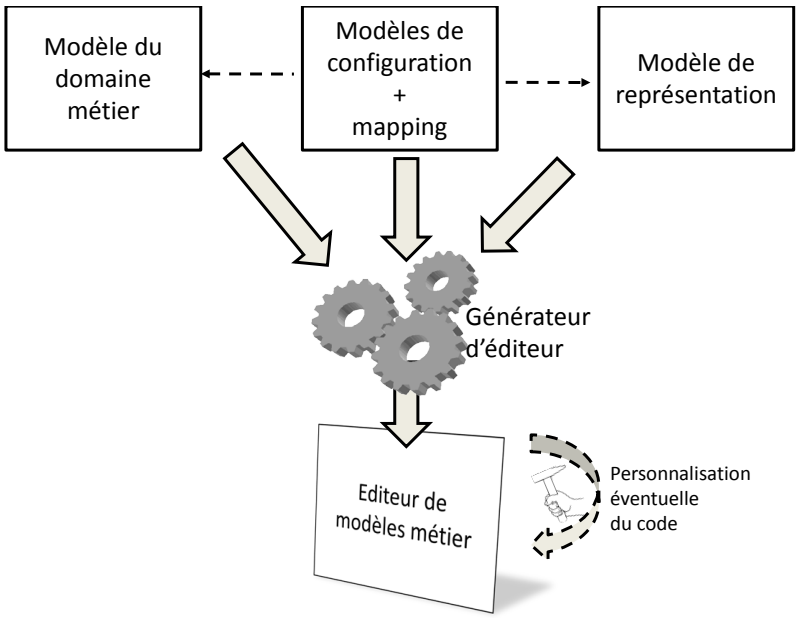


FIGURE 7.1 – Elements d'un générateur d'éditeur

Le processus général pour la génération automatique d'un éditeur est présenté sur la figure 7.1. L'éditeur sera construit à partir du modèle du domaine métier (*i.e.,* le métamodèle représentant la syntaxe abstraite), d'un modèle de la syntaxe concrète (décrivant par exemple des représentations textuelles ou graphiques) et d'un modèle de mapping entre les deux modèles précédents. Un modèle de configuration permet également d'ajuster le code généré à l'environnement dans lequel sera intégré l'éditeur produit (par exemple gérer l'extension de fichier associée).

Certains générateurs d'éditeurs utilisent une représentation générique et ne requirent pas dans ce cas le modèle de la syntaxe concrète et le mapping avec la syntaxe abstraite. De même, certains outils proposent de générer une version initiale de certains modèles nécessaires pour générer un éditeur, soit à partir du modèle du domaine métier, soit à partir du modèle de représentation. Cette pratique permet d'avoir un point de départ pour ensuite adapter l'éditeur en retouchant les différents modèles.

Le format XMI normalisé par l'OMG et utilisé pour sauvegarder les modèles n'est finalement qu'une forme de syntaxe textuelle générique mais qui a été adapté pour un traitement par la machine plutôt que d'être lu par un humain.

La suite de ce chapitre présente comment définir plusieurs types d'éditeurs : du plus simple utilisant une vue arborescente aux plus sophistiqués utilisant des syntaxes textuelles ou graphiques.

7.1 Éditeur arborescent

C'est le plus simple des éditeurs graphiques. Il est fourni par défaut avec EMF [2]. S'appuyant sur la structure de contenance des éléments du métamodèle pour créer un arbre, il peut être complètement dérivé du métamodèle. Il est même disponible en mode dynamique (ou réflexif) sans aucune génération de code. Pour l'utiliser en mode réflexif, il suffit d'ouvrir la classe racine

2. cf. http://www.eclipse.org/emf

FIGURE 7.2 – Programme *carré* dans l'éditeur arborescent et sérialisé en XMI.

dans le métamodèle et de créer une instance dynamique. Le fichier créé au format XMI pourra ensuite être ouvert directement comme dans la partie gauche de la figure 7.2. La partie droite de la figure montre comment ce modèle a été sérialisé en XMI.

Bien que ce mode dynamique (partie gauche de la figure 7.2) soit pratique lors des phases de mise au point du métamodèle, on préfèrera générer son code. En effet, cela permet d'améliorer à moindre coût l'intégration à l'AGL. En plus du support pour l'association d'une extension de fichier, le code généré de l'éditeur pourra être très facilement customisé en utilisant des libellés et des icônes plus parlants et qui rendra l'éditeur plus attrayant.

Cet éditeur sert généralement de base pour construire une vue *outline* pour d'autres éditeurs plus sophistiqués comme ceux que nous abordons dans les sections suivantes (par exemple dans la partie gauche de la figure 7.5).

7.2 Éditeur textuel

Une syntaxe textuelle pour un langage de modélisation permet généralement d'offrir une syntaxe expressive et permettant une manipulation efficace des modèles. Grâce à la définition d'un modèle de correspondance entre la syntaxe et les concepts du métamodèle, les outils comme EMFText [3], Xtext [4] ou Sintaks [5] permettent de générer automatiquement le code des différents éléments d'un éditeur textuel. En particulier, ces outils permettent d'obtenir automatiquement un *parser* et un *pretty-printer* qui permettent respectivement de traduire le modèle du format texte au format XMI et inversement. L'éditeur généré propose en outre la coloration syntaxique et la complétion automatique au moment de l'édition. Enfin, la glue et l'intégration à Eclipse est obtenue automatiquement pour ajouter dans l'AGL les différents menus et boutons associés à cette syntaxe, et profiter également des fonctionnalités offertes par la plateforme (impression...).

Le listing 7.1 décrit la syntaxe concrète textuelle de certains concepts du langage Logo en utilisant le langage de l'outil EMFText. Ce code est utilisé pour générer un éditeur de modèle

3. cf. http://www.emftext.org
4. cf. http://www.eclipse.org/Xtext/
5. cf. http://www.kermeta.org/sintaks

Listing 7.1 – Extrait de la syntaxe textuelle de Logo en EMFtext

```
1  SYNTAXDEF Logo
2  FOR <http://www.kermeta.org/kmLogo>
3      <platform:/resource/org.kermeta.kmLogo.Logoasm.model/model/ASMLogo.genmodel>
4  START ASM.LogoProgram
5
6  IMPORTS{
7    ASM: <http://www.kermeta.org/kmLogo/ASM>
8         <platform:/plugin/org.kermeta.kmLogo.Logoasm.model/model/ASMLogo.genmodel>
9  }
10 TOKENS{
11   DEFINE INTEGER$('-')?('1'..'9')('0'..'9')*|'0'$;
12 }
13 TOKENSTYLES{
14   "INTEGER" COLOR #007F55, BOLD;
15   "FORWARD" COLOR #CC8000, ITALIC;
16   "LEFT" COLOR #CC8000, ITALIC;
17   "RIGHT" COLOR #CC8000, ITALIC;
18 }
19 RULES{
20   ASM.LogoProgram::= (instructions !0)* ;
21   ASM.Forward::= "FORWARD" steps ;
22   ASM.Left::=    "LEFT" angle ;
23   ASM.Right::=   "RIGHT" angle ;
24   ASM.Constant::= integerValue[INTEGER] ;
25   ASM.Repeat::= "REPEAT" #0 condition !0 block ;
26   ASM.Block ::= "["(instructions:ASM.Right | instructions:ASM.Left |
27                  instructions:ASM.Forward | instructions:ASM.Repeat )* !0 "]";
28   ASM.Plus::=    (("(" lhs  "+" rhs ")" )|("+""(" lhs "," rhs ")")) ;
29   ASM.Equals::= (("(" lhs  "=" rhs ")" )|("=""(" lhs "," rhs ")")) ;
30 }
```

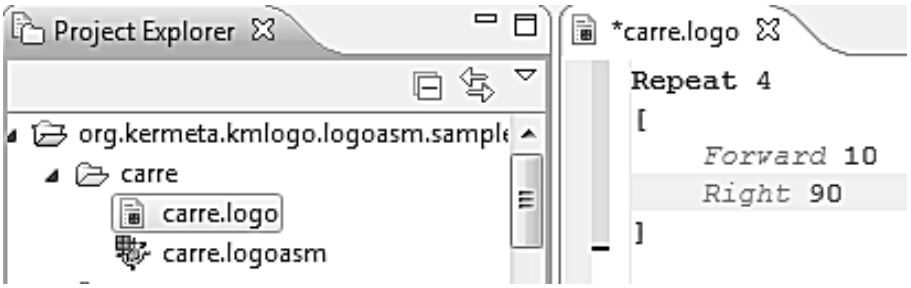


FIGURE 7.3 – Editeur textuel généré avec EMFtext pour le métamodèle Logo

s'intégrant à l'environnement Eclipse tel que présenté sur la figure 7.3.

Pour chaque classe du domaine métier nous retrouvons une règle qui traite l'élément correspondant. A l'intérieur de ces règles, les références entre éléments sont gérées grâce à la génération d'une classe *Resolver* en Java qui donne la manière de retrouver l'élément de modèle à partir du texte de la syntaxe concrète. Pour les syntaxes un peu plus complexes, le texte qui permet de pointer vers un élément utilise souvent une syntaxe dédiée, ou qui dépend du contexte. Il sera alors nécessaire d'indiquer à EMFtext comment retrouver les éléments grâce à la surcharge du *Resolver* associé à cette référence.

Pour démarrer rapidement une nouvelle syntaxe pour un métamodèle, une bonne pratique consiste à générer une préversion utilisant une syntaxe générique comme HUTN (Human Usable Textual Notation)[OMG04] de l'OMG, puis de la faire évoluer vers la syntaxe envisagée.

7.3 Éditeur graphique

La représentation graphique est souvent celle qui est préférée des utilisateurs pour permettre la réflexion en phase amont du processus de développement. Toutefois les compétences requises pour créer ce genre d'éditeur freinent souvent le développement. En effet, si l'on utilise les techniques traditionnelles, il est nécessaire de se familiariser avec les bibliothèques graphiques, éventuellement spécifiques à un environnement de développement. De même, il faut se familiariser avec les patrons d'architecture dédiés tel que le MVC (Modèle-Vue-Controleur).

Des outils comme les générateurs d'éditeurs graphiques des projets Topcased[6] ou GMF[7] permettent d'en limiter la complexité et la maintenance. Ces outils exploitent des modèles comme présenté dans la figure 7.1 pour relier les concepts du domaine métier avec leurs représentations graphiques souhaitées.

Prenons l'exemple Logo avec le modèle de configuration Topcased présenté sur la figure 7.4. On peut associer les différentes instructions primitives avec des noeuds (*Node* dans la figure) que l'on affichera sous forme de boîtes avec une image. Les instructions plus complexes telles que les *Block* ou les *Repeat* seront affichées dans des boites ayant un comportement de conteneur (c'est-à-dire qu'elles s'afficheront par exemple sous forme de rectangles à l'intérieur desquels on pourra placer d'autres formes). Pour les liens entre instructions, nous associerons la propriété *next* de *Instruction* du métamodèle[8] à la notion de trait (*Edge* dans la figure).

Les autres éléments de l'éditeur seront aussi configurables dans le modèle de configuration. Par exemple, on pourra préciser l'agencement des éléments dans la palette. On pourra aussi améliorer la vue *propriété* en y ajoutant des onglets personnalisés pour la rendre plus fonctionnelle.

Ce modèle de configuration permet de générer le code Java d'un éditeur de diagrammes Logo qui s'intègre parfaitement dans l'environnement Eclipse (cf. figure 7.5). D'autre part, le générateur d'éditeur proposé par Topcased gére le multi-diagramme. Cela veut dire que l'utilisateur de l'éditeur peut créer plusieurs diagrammes montrant différentes parties de son modèle. Si par la suite, on ajoute une deuxième configuration de diagramme pour fournir une vue différente, l'utilisateur pourra créer différents types de diagramme pour un même modèle. Par exemple, le modeleur UML fourni par Topcased propose sur les modèles UML tous les types de diagrammes associés tels que diagramme de classe, séquence, activité, etc.

Notons enfin que dans le cas d'une syntaxe concrète complexe ou pour des besoins particuliers de l'interface graphique, il sera parfois nécessaire de personnaliser le code généré. Dans ce cas, il est vivement conseillé de limiter les modifications aux zones de code protégées (*i.e.* qui seront conservées si on génère à nouveau l'éditeur à partir de sa configuration).

Zones protégées. Les générateurs de codes utilisent fréquemment des zones particulières dans le code générés pour permettre à l'utilisateur d'y insérer sont propre code. Les mots clés employés différent d'un générateur à l'autre. Ce pourront être une paire de balises ouvrante et fermantes, ou bien dans le cas de Topcased et de la plupart des outils basés sur EMF, se sera une annotation *@generated NOT* associée à une opération Java qui indiquera qu'il faut réutiliser cette opération plutôt que de la régénérer.

6. cf. http://www.topcased.org/
7. cf. http://www.eclipse.org/gmf
8. Pour des raisons de limitations de l'expressivité du langage de mapping de la version de Topcased utilisée, le métamodèle utilisé pour la syntaxe graphique est légèrement différent de la version standard. Ceci est présenté plus en détails dans la section 7.4.

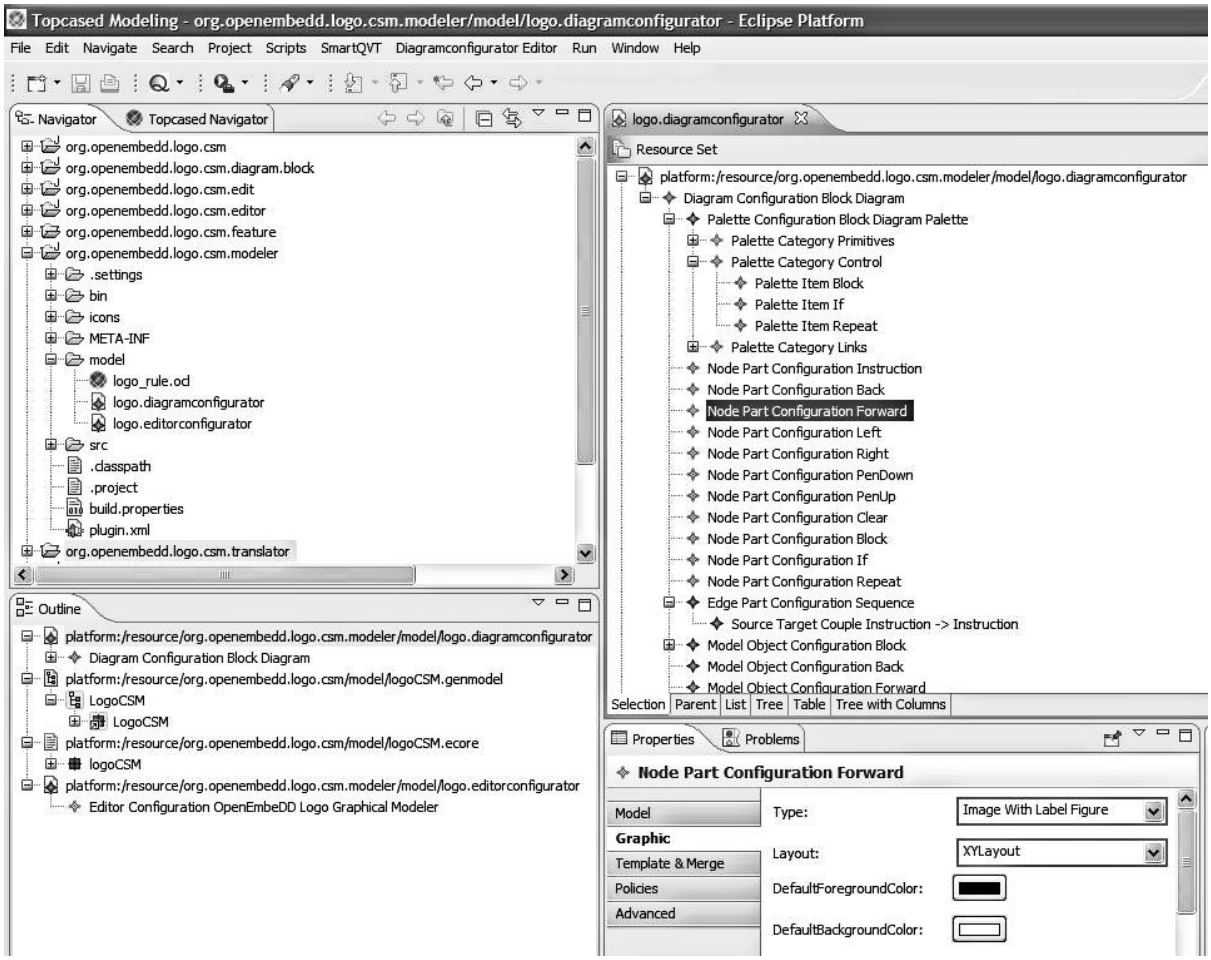


FIGURE 7.4 – Configuration permettant de générer un modeleur Logo

7.4 Bonnes pratiques

Les générateurs d'éditeurs exploitent pour la plupart un mapping entre les concepts du domaine métier et les concepts de représentation (graphique ou textuelle). Parfois l'expressivité de ces mappings n'est pas suffisante pour passer directement de la syntaxe abstraite à la syntaxe concrète visée. Il arrive que la syntaxe concrète ait une structure assez différente de la syntaxe abstraite du domaine métier. En effet, on peut généralement définir plusieurs représentations équivalentes qui ont leurs propres avantages ou inconvénients en fonction de l'usage. Ces usages peuvent varier, par exemple une syntaxe textuelle et une syntaxe graphique pourront avoir des représentations privilégiées différentes. C'est pourquoi le domaine métier essaye de s'en abstraire et de ne pas introduire de considération de syntaxe concrète dans son modèle (*i.e.*, le métamodèle).

Par exemple, on peut imaginer que la notion de liste ordonnée soit capturée dans la syntaxe abstraite via une propriété ordonnée, mais que l'on souhaite la représenter sous forme d'une liste chainée comme sur la figure 7.6. Cette seconde forme est plus facilement manipulable avec certains outils car plus proche de leur modèle de représentation interne. Par exemple, avec l'outil Topcased, on pourra associer le concept graphique *trait* à la propriété de métamodèle permettant le chainage.

De même, on peut avoir besoin de présenter le domaine sous une vue qui emprunte des

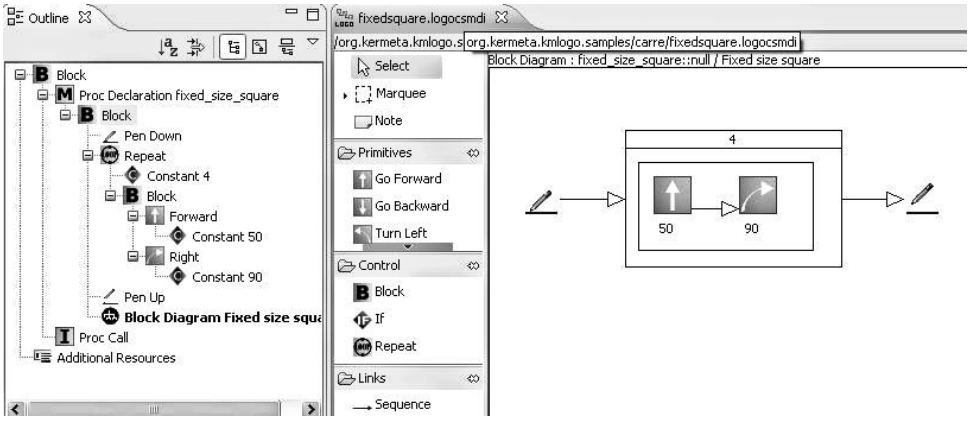
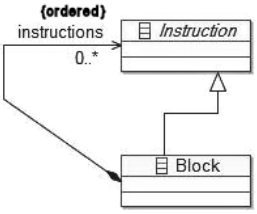
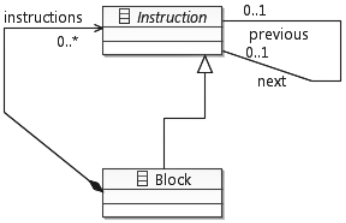


FIGURE 7.5 – Exemple de modeleur généré avec Topcased



(a) Propriété *instructions* est ordonnée



(b) Propriété *instructions* non ordonnée + liste chainée

FIGURE 7.6 – Exemple de structure proches : Propriété ordonnée vs Liste chainée

concepts à un autre domaine.

Dans ces situations, une solution consiste à découper le mapping en deux, en passant par un métamodèle intermédiaire. On basera le générateur d'éditeur sur ce métamodèle intermédiaire et on créera une transformation entre la syntaxe abstraite et le métamodèle intermédiaire. De cette façon, il devient possible de profiter des capacités de génération d'éditeur pour notre domaine métier.

Pour que cette technique fonctionne de manière optimale, il faut néanmoins faire attention à écrire la transformation en prenant en considération la traçabilité entre les éléments. En effet, pour que les retours d'erreurs (issus d'un autre outil par exemple) puissent se faire correctement dans l'éditeur, il est nécessaire de pouvoir lier les éléments entre eux. Cela se réalise soit en utilisant une transformation bidirectionnelle (par mapping ou par création de deux transformations), soit via la génération d'un modèle de trace lors de la transformation (en faisant en sorte que la transformation crée non seulement son modèle de sortie mais aussi un modèle contenant des liens entre le modèle d'entrée et de sortie).

De manière générale, en IDM, lorsqu'un problème introduit une trop grande complexité dans une transformation, on la découpe en plusieurs transformations plus simples en passant par un ou plusieurs métamodèle(s) intermédiaire(s).

Chapitre 8

Simulation de modèle par tissage de la sémantique dynamique

L'étape suivante dans la construction d'un environnement pour le langage Logo consiste à définir sa sémantique opérationnelle et son environnement d'exécution correspondant. Un tel environnement pourra être utilisé pour interpréter, animer ou simuler des modèles, dans notre cas des programmes Logo. La définition d'un tel environnement est généralement nécessaire pour valider et vérifier des modèles conformes au domaine métier tôt dans le processus de développement, avant leurs raffinements vers des implémentations plus concrètes intégrant de nouvelles préoccupations.

Nous proposons dans ce chapitre de définir un environnement d'exécution pour les programmes logo. Pour cela, nous définirons tout d'abord la sémantique dynamique du langage Logo sous la forme d'une machine virtuelle (également appelée *interpréteur*), caractérisant le domaine sémantique et le mapping entre la syntaxe abstraite et le domaine sémantique. Nous élaborerons ensuite un outil support à cette machine virtuelle consistant à simuler graphiquement l'exécution d'un programme. De cette manière, autant d'outils que nécessaire (animateur, simulateur interactif et/ou *batch*, outils de *debug*, etc.) peuvent être définis en s'appuyant sur la même machine virtuelle du langage Logo.

8.1 Utilisation de Kermeta pour décrire le comportement des modèles

Intégré au niveau des métamodèles, le langage d'action de Kermeta a été spécialement conçu pour manipuler des modèles. Il permet ainsi d'importer ou de définir des métamodèles au format Ecore et de les compléter en définissant le corps des méthodes dans les classes du métamodèle. Les comportements ainsi spécifiés au niveau du langage pourront être appliqués à l'ensemble des modèles qui s'y conforment.

Le langage d'action de Kermeta est un langage impératif et inclut donc les structures de contrôle classiques, comme les blocs, les conditionnelles et les boucles. Comme le MOF spécifie des structures orientées objet (classes, propriétés et opérations), Kermeta propose les mécanismes qu'on trouve traditionnellement dans les approches orientées objet pour supporter l'héritage multiple et la redéfinition de comportement avec une sémantique de liaison dynamique. Les conflits d'héritage multiple (permis par Ecore) sont traités explicitement à l'aide d'un mécanisme simple de sélection de comportement (similaire à celui proposé par le langage Eiffel [Mey97]). À

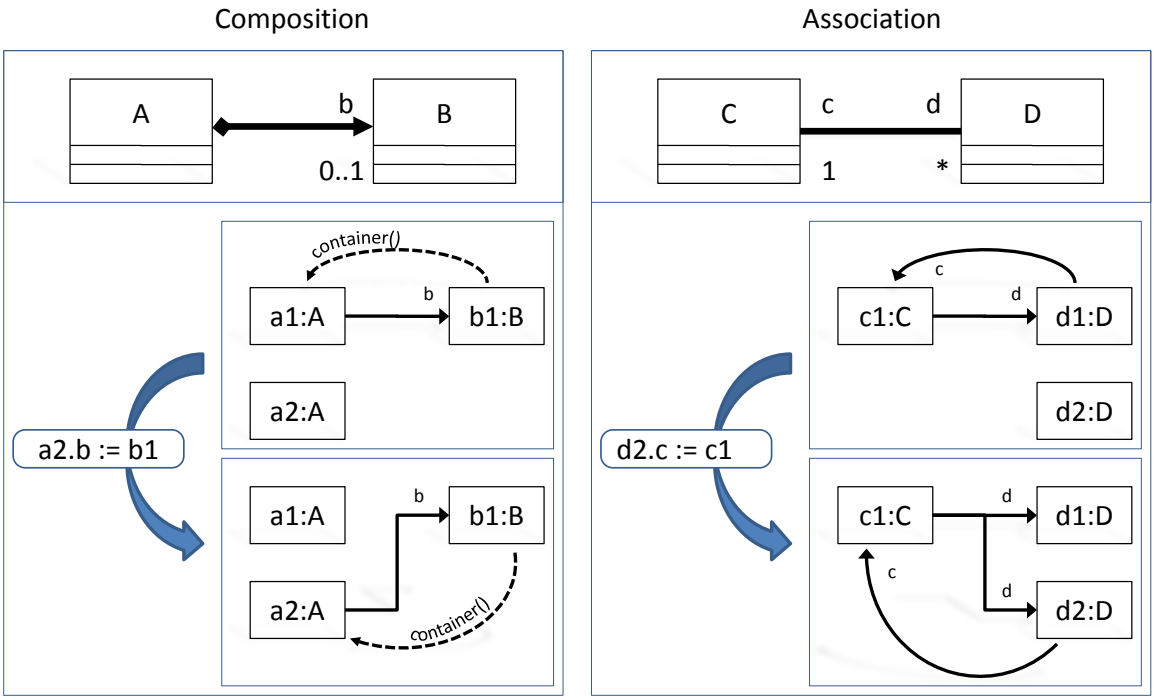


FIGURE 8.1 – Sémantique de l'affectation dans la métamodélisation

l'instar de la plupart des langages orientés objets modernes, Kermeta supporte la généricité et propose aussi des mécanismes de réflexion et de gestion d'exceptions.

Au-delà de ces structures orientées objets classiques, le MOF propose des concepts orientés modèles comme les associations et la composition. Ces éléments demandent donc une sémantique spécifique du langage d'action pour maintenir l'intégrité des associations et des relations de composition. Par exemple, comme illustré sur la figure 8.1, en Kermeta, l'affectation d'une nouvelle valeur à une propriété gère automatiquement l'autre extrémité de l'association si celle-ci fait partie d'une association bidirectionnelle, et la relation de contenance d'objet si la propriété est une composition.

Les expressions Kermeta sont très similaires aux expressions OCL excepté qu'elles autorisent les modifications (*i.e.*, effets de bord) sur le modèle. Kermeta dispose de fermetures lexicales similaires aux itérateurs de collections OCL tels que *each, collect, select* ou *detect*. Le *Framework standard* de Kermeta inclut aussi toutes les opérations définies dans la librairie standard d'OCL. Comme nous avons pu le voir au chapitre 6, cet alignement entre Kermeta et OCL permet aux contraintes OCL d'être directement importées et évaluées dans Kermeta. Des préconditions et des postconditions peuvent être définies pour les opérations, et des invariants peuvent être définis pour des classes. L'exécution d'un programme Kermeta dispose d'un mode spécifique qui permet de surveiller ces contrats et de rapporter toute violation (cf. section 6.3).

Comme Kermeta est une extension du MOF, un métamodèle MOF peut très bien être vu comme un programme Kermeta valide qui se contente de déclarer des packages, des classes etc. mais ne fait rien. Dans cette optique, Kermeta peut être utilisé pour insuffler la vie dans un tel métamodèle en introduisant incrémentalement différents aspects pour gérer les différentes préoccupations que sont la sémantique statique, la sémantique dynamique, ou les transformations de modèle [MFV+05].

Rendue possible grâce à la notion d'aspect, la modularisation exposée dans la section 5.1.2

autorise aussi une certaine flexibilité dans la manière d'exprimer les sémantiques. Par exemple, différentes sémantiques opérationnelles pourraient être définies pour un même métamodèle et encapsulées dans différentes unités de manière à être alternativement composées en fonction de besoins particuliers. Ceci permet par exemple de supporter très facilement les différentes variations sémantiques du métamodèle UML.

8.2 Définition du domaine sémantique de Logo

Comme mentionné précédemment, l'environnement d'exécution des modèles Logo (*i.e.* le domaine sémantique également appelé le modèle du *runtime*) le plus populaire fait appel à une tortue qui peut tracer des segments de droite avec un crayon.

Nous proposons ainsi de distinguer le modèle de la syntaxe abstraite (*i.e.,* le métamodèle définit dans le chapitre 5) et le modèle de l'environnement d'exécution en le modélisant lui-même par un métamodèle différent, également décrit à l'aide du langage Ecore. Notons que ce n'est pas obligatoire et que ce n'est pas toujours le cas. Par exemple, le tutoriel [1]expliquant l'implémentation en Kermeta d'un langage de machine à états étend directement la syntaxe abstraite du langage pour définir le modèle de l'environnement d'exécution.

Comme pour la syntaxe abstraite du langage, la structure du modèle de runtime peut aussi être définie par un métamodèle. L'avantage de cette approche est que l'état du programme Logo en cours d'exécution est aussi un modèle, et donc comme pour n'importe quel autre modèle, tous les outils disponibles dans un framework comme EMF sont immédiatement utilisables pour observer, sérialiser, charger ou encore éditer l'état du modèle (p.-ex., pour un animateur de modèle ou un simulateur interactif).

La possibilité de sérialiser ou d'éditer l'état du modèle permet de créer assez simplement des simulateurs sophistiqués proposant par exemple de capturer des traces sous forme de scénario, suspendre et reprendre l'exécution dans différents contextes, rejouer des scénarios, debugger, corriger à la volée, introduire plus facilement des erreurs pour tester le comportement aux limites. Toutes ces fonctionnalités sont généralement très appréciées des utilisateurs finaux pour tester et valider leurs modèles.

La figure 8.2 présente le diagramme du modèle de la machine virtuelle de Logo. Ce modèle défini seulement 3 classes : *Turtle*, *Point* et *Segment*. L'état d'un programme Logo en cours d'exécution est modélisé comme une instance de la classe *Turtle* qui contient une position (qui est un *Point*), un cap (exprimé en degrés) et un booléen pour indiquer si le crayon de la tortue est levé ou baissé. La tortue contient aussi tous les segments qui ont été tracés précédemment dans l'exécution du programme Logo.

En pratique, ce modèle est défini comme un métamodèle Ecore. Celui-ci permet de définir les informations dynamiques permettant de capturer la séquence d'états d'un programme au cours de son exécution. Ecore permet également de définir dans chaque classe du métamodèle la signature des méthodes de la sémantique opérationnelle (*i.e.,* la signature des méthodes permettant de capitaliser le comportement à l'exécution des concepts métiers de la syntaxe abstraite).

Néanmoins, Ecore ne permet pas de décrire le corps des opérations, c'est-à-dire qu'un métamodèle Ecore ne peut contenir l'implantation des instructions de la machine virtuelle de

1. cf. http://kermeta.org/documents/tutorials/building_dsl_tutorials/tut-fsm_sample/

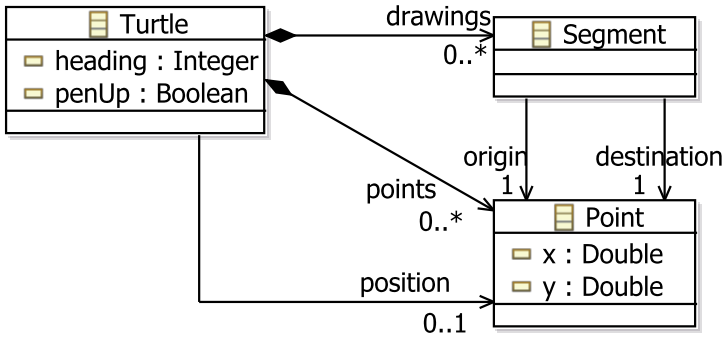


FIGURE 8.2 – Modèle de l'environnement d'exécution des programmes Logo

Listing 8.1 – Opérations en Kermeta du domaine sémantique de Logo

```
1  package kmLogo;
2  require "VMLogo.ecore" // import the structure of the Logo virtual machine
3  [...]
4  package VM {
5    aspect class Turtle {
6      operation setPenUp(b : Boolean) is do
7        penUp := b
8      end
9      operation rotate(angle : Integer)  is do
10       heading := (heading + angle).mod(360)
11     end
12     operation forward(steps : Integer)  is do
13       var radian : Real init math.toRadians(heading.toReal)
14       move(scale(steps,math.sin(radian)), scale(steps,math.cos(radian)))
15     end
16     [...]
17     /* scale the "steps" by factor */
18     operation scale(steps : Integer, factor : Real) : Real is do
19       result := steps.toReal * factor
20     end
21   }
22 }
```

Logo, comme par exemple calculer une nouvelle position d'après la position courante, un cap et une distance. Celles ci-peuvent cependant être facilement décrites en Kermeta, et ensuite tissées dans le métamodèle Ecore pour ainsi fournir une définition orientée objet de la machine virtuelle Logo. Le listing 8.1 présente l'ajout en Kermeta de quatre opérations à la classe *Turtle* (cf. figure 8.2). Ces opérations offrent des fonctions utilitaires pour faciliter les changements d'état du système par exemple pour effectuer une rotation (lignes 10 à 12) ou pour faire un déplacement (lignes 13 à 16)[2].

8.3 Définition de la sémantique dynamique de Logo

Nous allons maintenant définir la sémantique opérationnelle de chaque construction de la syntaxe abstraite. L'idée est à nouveau de tisser chaque fragment de sémantique opérationnelle,

2. On notera que même si le métamodèle du domaine sémantique Logo ne nécessite qu'une précision sur des entiers, le calcul de déplacement doit utiliser une précision avec des réels afin que les simulations soient suffisamment réalistes.

décrite en Kermeta, directement dans le métamodèle de manière à ce que chaque type d'instruction contienne une opération *eval* effectuant les actions appropriées sur le modèle sous-jacent de l'environnement d'exécution. Pour cela, les opérations *eval* prennent en paramètre un contexte. En pratique, ce contexte contient une instance de la classe tortue (*Turtle*) conforme au métamodèle décrivant l'environnement d'exécution (cf. figure 8.2), ainsi qu'une pile pour gérer les appels de procédures, en particulier dans les cas de récursivité.

Le listing 8.2 présente la manière dont l'opération *eval* est tissée dans la syntaxe abstraite de Logo. Tout d'abord, une opération abstraite *eval* est définie dans la classe *Statement*, puis réalisée de manière concrète dans chaque sous-classe qui le nécessite pour réaliser la mise en œuvre opérationnelle de ce concept.

L'approche présentée ici est très similaire au design pattern *interpréteur* [GHJV95] en programmation orientée objet. Celui-ci aussi exploite des méthodes *interprete* (ou *eval*) associées à chacune des classes représentant les concepts du langage. Néanmoins, le code de la structure du langage est parfois fourni par une tierce partie et n'est donc pas toujours directement modifiable pour lui ajouter les méthodes appropriées. Les aspects permettent de contourner ce point.

Listing 8.2 – Sémantique opérationnelle de Logo

```
1  package kmLogo;
2  require "ASMLogo.ecore"      // import Logo metamodel (ie. abstract syntax)
3  require "LogoVMSemantics.kmt" // import Logo virtual machine and its operations
4  [...]
5  package ASM {
6    aspect class Statement {
7      operation eval(context : Context) : Integer is abstract
8    }
9    aspect class Plus {
10     method eval(context : Context) : Integer is do
11       result := lhs.eval(context) + rhs.eval(context)
12     end
13   }
14   aspect class Greater {
15     method eval(context : Context) : Integer is do
16       result := if lhs.eval(context) > rhs.eval(context) then 1 else 0 end
17     end
18   }
19   aspect class If {
20     method eval(context : Context) : Integer is do
21       if condition.eval(context) != 0 then
22         result := thenPart.eval(context)
23       else
24         result := elsePart.eval(context)
25       end
26     end
27   }
28   aspect class Forward {
29     method eval(context : Context) : Integer is do
30       context.turtle.forward(steps.eval(context))
31       result := void
32     end
33   }
34   aspect class PenDown {
35     method eval(context : Context) : Integer is do
```

```
36        context.turtle.setPenUp(false)
37        result := void
38      end
39    }
40    [...]
41  }
```

Dans les cas simples comme celui de l'instruction *PenDown*, le mapping vers la machine virtuelle est immédiat : il suffit d'appeler l'instruction correspondante sur la tortue fournie par la machine virtuelle (cf. ligne 36 du listing 8.2 : context.turtle.setPenUp(false)).

Dans les cas plus complexes, comme celui de l'instruction *Plus*, il y a deux choix possibles. Le premier illustré aux lignes 9–13 du listing 8.2, fait la supposition que la sémantique du *Plus* du langage Logo peut être directement traduite sur la sémantique du "+" du langage Kermeta. L'intérêt de cette première solution est qu'elle fournit une manière simple et évidente de définir la sémantique de ce genre d'opérateur. En revanche, si pour une raison ou une autre (par exemple nous ne voulons gérer que des entiers de 8 bits en Logo), la sémantique que nous voulons pour le *Plus* de Logo n'est pas celle fournie directement par Kermeta, nous pouvons alors définir la sémantique souhaitée pour l'opération *Plus* dans la machine virtuelle de Logo (en utilisant Kermeta bien sûr). Il suffira de changer la méthode aux lignes 9–13 du listing 8.2 de manière à ce qu'elle appelle d'abord l'opération *eval* sur sa partie gauche (lhs), puis qu'elle empile le résultat sur une pile dans la machine virtuelle, ensuite qu'elle appelle l'opération *eval* sur sa partie droite (rhs), empile à nouveau le résultat sur la pile de la machine virtuelle, et finalement appelle l'opération *Plus* définie dans la machine virtuelle.

8.4 Définition d'un interpréteur de programme Logo

Une fois que la sémantique opérationnelle pour Logo a été définie tel que décrit ci-dessus, obtenir un interpréteur est extrêmement simple. La première chose à faire est d'importer chaque aspect de l'environnement dans le métamodèle de Logo (en utilisant les instructions *require*, comme illustré aux lignes 2–5 du listing 8.3). Nous pouvons alors charger le programme Logo dans l'environnement Kermeta (cf. lignes 9–12 du listing 8.3) pour obtenir l'élément racine du programme Logo (ici un *Block*[3]), instancier un *Context* (qui contient la machine virtuelle Logo) et finalement appeler *eval(Context)* sur cet élément (cf. ligne 16 du listing 8.3).

Listing 8.3 – Définition d'un interpréteur de modèle en Kermeta

```
1  package kmLogo;
2  require "ASMLogo.ecore"       // import Logo metamodel (ie. abstract syntax)
3  require "StaticSemantics.ocl" // import OCL constraints definitions
4  require "LogoVMSemantics.kmt" // import Logo virtual machine and its operations
5  require "OperationalSemantics.kmt" // import Logo operational semantic (i.e. eval operations)
6  [...]
7  class Main {
8    operation Main(): Void is do
9      var rep : EMFRepository init EMFRepository.new
10     var LogoProgram : ASMLogo::Block
11     // load LogoProgram from its XMI file
12     LogoProgram ?= rep.getResource("Square.xmi").one
13     // Create a new Context containing the Logo VM
14     var context : LogoVMSemantics::Context init LogoVMSemantics::Context.new
15     // now execute the LogoProgram
```

3. On voit apparaître en ligne 12 l'opérateur *?=* qui est une assignation conditionnelle ou *downcast*. Il permet d'assigner à la variable un object seulement si l'objet fourni est du bon type, sinon la variable sera mise à *void*.

```
16      LogoProgram.eval(context)
17    end
18  end
```

En chargeant le programme *Square* de la figure 5.2 et en l'exécutant de cette manière, on obtiendra une série de modifications successives de l'état du modèle de la machine virtuelle Logo : pendant l'exécution 4 nouvelles instances de *Segments* seront ainsi ajoutées à la tortue (instance de *Turtle*), et la position ainsi que le cap de cette dernière évolueront.

8.5 Définition d'un simulateur graphique de programme Logo

Bien sûr, nous aimerions voir graphiquement des traces de cette exécution. La solution pour cela est assez simple : nous avons juste besoin de placer un observateur (cf. le patron de conception *observateur* [GHJV95]) sur le modèle de la machine virtuelle Logo pour afficher graphiquement le résultat de cette exécution dans un widget Java. Cet observateur est implémenté en Kermeta (lignes 34 à 62 du listing 8.4). L'opération *move* de la classe *Turtle* se charge de notifier tous ses observateurs [4] que la tortue a bougé. L'observateur exploite alors les informations des tracés (lignes 56 à 59) et appelle les méthodes pertinentes d'une interface homme machine écrite en Java (via les appels vers des méthodes *extern* lignes 25 et 30) pour dessiner à l'écran le tracé parcouru par le stylo de notre tortue.

Listing 8.4 – Code Kermeta de l'observateur pour une simulation graphique

```
1  package kmLogo;
2  require kermeta
3  [. .]
4  package VM {
5    aspect class Turtle {
6      reference moveListener : kmLogo::listener::TurtleMoveListener[0..*]
7      operation move(dx : Real, dy : Real) is do
8        var newPos : Point init position.getDelta(dx,dy)
9        var trace : Segment
10       trace := Segment.new
11       trace.make(position,newPos)
12       if not penUp then
13         drawings.add(trace)
14       end
15       position := newPos
16       moveListener.each{l | l.turtleHasMoved(trace) }
17     end
18   }
19 }
20 package gui{
21   class TurtleGUI
22   {
23   operation initialize(name : String) : TurtleGUI is do
24       //  call the java wrapper
25       extern org::kermeta::kmLogo::wrapper::TurtleGUIWrapper.initialize(self, name)
26       result := self
27     end
28     operation drawLine(x1: Integer, y1: Integer, x2: Integer, y2 : Integer) : Void is do
29       // call the java wrapper
```

4. On pourrait imaginer plusieurs observateurs, pour par exemple avoir des traces textuelles de l'exécution du programme Logo.

```
30        extern org::kermeta::kmLogo::wrapper::TurtleGUIWrapper.drawLine(self, x1,y1, x2, y2)
31      end
32    }
33  }
34  package listener{
35    class TurtleMoveListener
36    {
37      operation turtleHasMoved(newSegment : kmLogo::VM::Segment) : Void is abstract
38    }
39    class TurtleControler inherits kmLogo::listener::TurtleMoveListener
40    {
41      attribute registeredGUI : TurtleGUI
42      reference turtleModel : Turtle
43
44      operation initialize(model : Turtle) : TurtleControler is do
45        turtleModel := model
46        turtleModel.moveListener.add(self)
47        // create a default GUI
48        var gui : TurtleGUI init TurtleGUI.new.initialize("My Turtle")
49        registeredGUI := gui
50        result := self
51      end
52
53      /** implement TurtleMoveListener operation */
54      method turtleHasMoved(newSegment : kmLogo::VM::Segment) : Void is do
55        // use the data in the model to draw the board
56        turtleModel.drawings.each{s |
57          registeredGUI.drawLine(s.origin.x.toInteger, s.origin.y.toInteger,
58              s.destination.x.toInteger, s.destination.y.toInteger)
59        }
60      end
61    }
62  }
```

8.6 Autres techniques de simulation

Nous venons de voir comment réaliser un simulateur en utilisant une sémantique opérationnelle (cf. section 2.4.3). Cette approche est aussi réalisable en utilisant un langage généraliste et son framework de manipulation de modèle comme le couple Java/EMF. Néanmoins, sans la notion d'aspect, cela suppose que l'on ait accès au code des classes du métamodèle pour leur associer les méthodes *eval*. Alternativement, on pourra utiliser la structure du design pattern *visiteur* pour implémenter un interpréteur. L'utilisation d'une telle API impose également d'exprimer la sémantique opérationnelle dans le code généré à partir du métamodèle et non sur les concepts du métamodèle directement. En plus d'induire une certaine complexité, cela requiert également de connaitre la structure du code généré à partir d'un métamodèle (*i.e.*, le fonctionnement du compilateur, par exemple celui fournit par EMF)

Il est aussi possible de réaliser un simulateur en utilisant une sémantique dénotationnelle (cf. section 2.4.4) à condition que le domaine cible dispose déjà de sa propre plate-forme de simulation. Dans ce cas, cela revient à transformer (*i.e.,* compiler) nos modèles vers le domaine choisi. On utilisera pour cela des techniques de transformation comme celles abordées dans le chapitre 9 suivant.

Cette technique est généralement utile lorsqu'il existe déjà un simulateur pour un domaine

proche de celui que l'on cherche à simuler. L'objectif étant de faire en sorte que le coût de développement de la traduction soit moins élevé que de développer directement un simulateur.

De plus, pour que l'utilisateur puisse comprendre les résultats, il faudra tenir compte que la simulation se fera dans le domaine cible et non dans le domaine de l'utilisateur. On utilisera généralement des mécanismes de traçabilité pour mettre en relation les résultats de la simulation avec les éléments du modèle d'entrée ce qui permettra d'afficher les résultats dans le formalisme d'entrée. Dans les cas simples, la traçabilité pourra s'appuyer sur une convention de nommage entre les domaines. Dans les cas de mapping non trivial, la traçabilité utilisera un modèle dédié qui sera, soit stocké séparément, soit lorsque c'est possible stocké sous forme d'annotation dans le modèle cible. Ces derniers points tendent à rendre la compilation plus complexe si les domaines diffèrent beaucoup. Il faut donc limiter cette pratique pour les cas où le domaine cible utilise une sémantique proche du domaine d'entrée que l'on souhaite simuler ou que le simulateur du domaine cible dispose d'une boite à outils conséquente que l'on souhaite réutiliser.

Chapitre 9

Compilation, *refactoring* et autres transformations de modèle

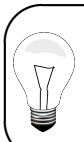
Les chapitres précédents ont permis d'illustrer la définition d'un DSML, des éditeurs associés et des techniques pour décrire la manière dont un modèle doit évoluer au cours de son exécution. D'autres outils peuvent être envisagé dans un environnement de modélisation dédié pour automatiser certaines tâches du développement à partir de modèles (refactoring, génération de code, de test et de documentation, etc.). Pour cela, ce chapitre présente les techniques de transformations les plus courantes. Ces techniques sont illustrées sur deux exemples de transformation : une compilation vers une plateforme cible et un refactoring.

9.1 Transformations Modèle-vers-Texte et Modèle-vers-Modèle

On distingue généralement deux grands types de transformation de modèle en fonction de la nature des données produites : les transformations de modèle vers du texte sont souvent appelées M2T (Model To Text), et les transformations de modèle vers modèle qui sont appelées M2M (Model To Model).

Transformations de Modèle-vers-Texte : ces transformations sont extrêmement utiles pour générer du code, du XML, du HTML, ou toute autre sorte de texte de manière relativement simple. Cela s'applique surtout si le besoin est en fait un transcodage de niveau syntaxique (*e.g., Pretty-Printing*). Dans ce cas, on peut s'appuyer sur deux approches qui peuvent éventuellement être combinées :

- une approche basée sur le design pattern *Visiteur*, qui est utilisé pour traverser la représentation interne d'un modèle et écrire au fur et à mesure le texte sur un flot de sortie textuelle.
- une approche basée sur la notion de *Template* (ou gabarit), c'est-à-dire la forme concrète du texte cible qui contient des éléments de "méta-code" pour accéder à l'information pertinente dans la source et d'effectuer la sélection de textes et des expansions itératives. L'avantage de cette méthode est que la structure d'un template ressemble très étroitement au texte qui doit être généré.
  Les templates simplifient la génération de toutes sortes d'artefacts textuels. Ils ne sont toutefois pas adapté à des traitements algorithmiques complexes pour lesquels l'approche avec un *Visiteur* est plus adapté.



Lorsque l'on souhaite faire du modèle-vers-texte, on peut aussi considérer l'approche par mapping telle que celle utilisée par les éditeurs textuels vus dans la section 7.2. Néanmoins, cela limite l'utilisation aux cas où l'on souhaite que le texte obtenu soit une représentation fidèle du modèle.

Transformations de Modèle-vers-modèle : ces transformations doivent être utilisées pour des cas plus complexes en particulier lorsque les transformations ne sont pas purement syntaxiques et font appel à des calculs ou à des éléments de sémantique qui rendront l'approche précédente difficile à développer, mais encore plus à comprendre, à maintenir et à faire évoluer. Elles sont aussi à privilégier dans le cas d'une succession de transformations.

Les deux sections suivantes vont illustrer ces deux techniques.

9.2 Compilateur de modèle par tissage d'une transformation de modèles vers texte

Dans cette section, nous allons décrire comment traduire les programmes décrits avec notre DSML Logo vers le langage d'une plateforme d'exécution cible. Cela revient à construire un compilateur (*i.e.,* un générateur de code) pour notre langage Logo.

9.2.1 Choix d'une plateforme cible pour l'exécution d'un programme Logo

L'idée centrale consiste à mettre en correspondance chaque élément du métamodèle Logo avec l'interface de programmation (API) offerte par les Lego Mindstorms de manière à ce que les programmes Logo puissent en pratique être utilisés pour piloter un petit robot imitant la tortue de Logo. Ce *Robot-tortue* est construit à l'aide de pièces du jeu Lego Mindstroms, et en particulier avec deux moteurs pour contrôler les roues et un troisième pour contrôler la montée et la descente du crayon (voir figure 9.1).

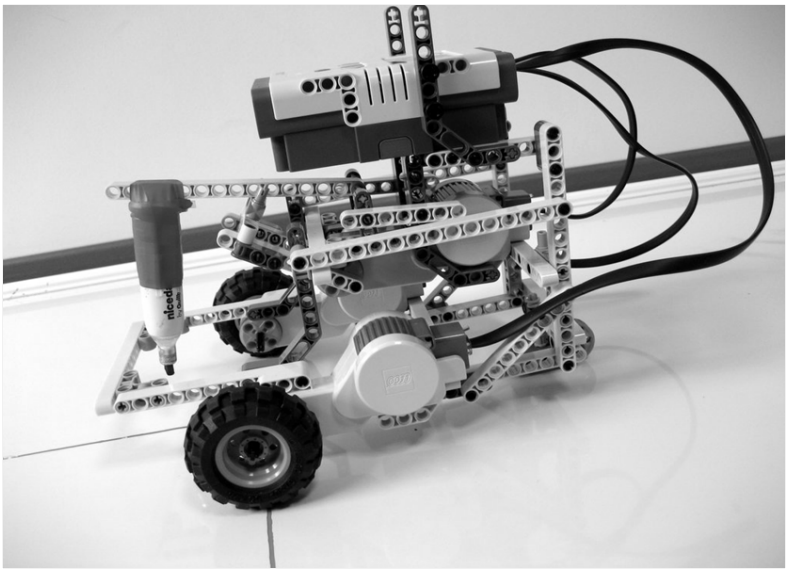


FIGURE 9.1 – Une tortue Lego Mindstorms

La cible que nous visons est un langage de programmation simplifié pour les Lego Mindstroms appelé NXC (ce qui signifie *Not eXactly C*). La construction d'un compilateur Logo pour cette cible revient alors à écrire un traducteur depuis Logo vers NXC. De ce point de vue, ce problème est extrêmement proche de la problématique du MDA (*Model Driven Architecture*) promue par l'OMG, avec dans le rôle d'un modèle indépendant de la plate-forme (*Platform Independent Model* ou PIM) le programme Logo, et dans celui du modèle dépendant de la plate-forme (*Platform Specific Model* ou PSM) le programme NXC. Si on suit cette interprétation, la compilation dont nous avons besoin est simplement une transformation de modèle.

Plus particulièrement dans notre cas, ce sera une transformation de modèle vers du texte. Comme évoqué dans la section 3.2.1, les chaines de compilations sont parfois plus sophistiquées et nécessitent parfois l'usage d'une succession de transformations M2M et M2T. Par exemple si des transformations complexes ou multi-passes avaient été nécessaires pour traduire Logo en NXC, il aurait pu être intéressant d'avoir un métamodèle explicite de NXC, puis d'implémenter explicitement la transformation M2M entre le métamodèle Logo et celui de NXC, et finalement d'appeler un *pretty-printer* (*i.e.,* transformation M2T) pour générer le code en NXC.

9.2.2 Définition d'un compilateur pour Logo par l'utilisation d'un visiteur

La traduction du langage Logo vers NXC étant plutôt simple, nous pouvons implanter directement une transformation M2T. Nous montrons dans cette partie comment la définir à l'aide d'une approche basée sur un visiteur. En pratique, nous allons une fois de plus utiliser le mécanisme du tissage d'aspects de Kermeta pour introduire de manière simplifiée le design pattern visiteur dans le métamodèle de Logo. En effet, plutôt que d'utiliser, comme prescrit dans le design pattern visiteur (partie gauche sur la figure 9.2), une paire de méthodes *accept* et *visit*, où chaque méthode *accept* introduite dans les classes du métamodèle Logo devrait rappeler (*call back*) la méthode *visit* pertinente du visiteur, nous pouvons simplement tisser une méthode *compile()* dans chacune des classes du métamodèle Logo (voir partie droite sur la figure 9.2 et listing 9.1). Dans le cas d'un *pretty-printer* simple, le résultat de la transformation étant du texte, les appels pourront se chainer pour assembler le texte. Chaque opération *compile* aura pour rôle de construire le texte approprié. Ce sera par exemple le cas de l'assemblage des instructions à l'intérieur d'un bloc comme présenté à la ligne 11 du listing 9.1. La classe *Context* pourra être complétée avec des données permettant de gérer l'indentation ou bien de multiples fichiers de sortie si nécessaire.

L'intégration de cet aspect de compilation dans notre environnement de développement Logo se fait en utilisant la directive *require* dans le programme principal Kermeta (voir listing 9.2). Ce dernier est ainsi très similaire à celui du simulateur vu dans le chapitre précédent (cf. listing 8.3) mais au lieu d'appeler la méthode *eval*, on appelle la méthode *compile*.

9.2.3 Définition d'un compilateur pour Logo par l'utilisation de templates

Il est également possible de simplifier certaines parties de la génération de code en combinant la définition de visiteurs avec des templates définit avec des langages de génération de texte comme KET (*Kermeta Emitter Template*) [1].

KET est basé sur le moteur de JET (*Java Emitter Template*) [2] en proposant en plus la puissance d'expression du langage Kermeta pour la navigation (issue d'OCL) et la manipulation impérative des modèles. Le principe de template est similaire à celui proposé par exemple par

1. cf. http://www.kermeta.org/mdk/ket/
2. cf. http://www.eclipse.org/modeling/m2t/?project=jet

Listing 9.1 – Définition en Kermeta d'un compilateur pour Logo

```
1  package kmLogo;
2  require "ASMLogo.ecore"       // import Logo metamodel
3  [...]
4  package ASMLogo {
5    aspect class Block
6    {
7      method compile(context : NXCCompiler) : String is do
8        var content : StringBuffer init StringBuffer.new
9        content.append("{\n")
10       instructions.each{ instruction  |
11         content.append( instruction.compileToNXC(context) + "\n")
12       }
13       content.append("}")
14       result := content.toString
15     end
16   }
17   aspect class PenUp {
18     method compile (ctx: Context) : String {
19       result := "penUp();\n"
20     }
21   }
22 [...]
23 }
```

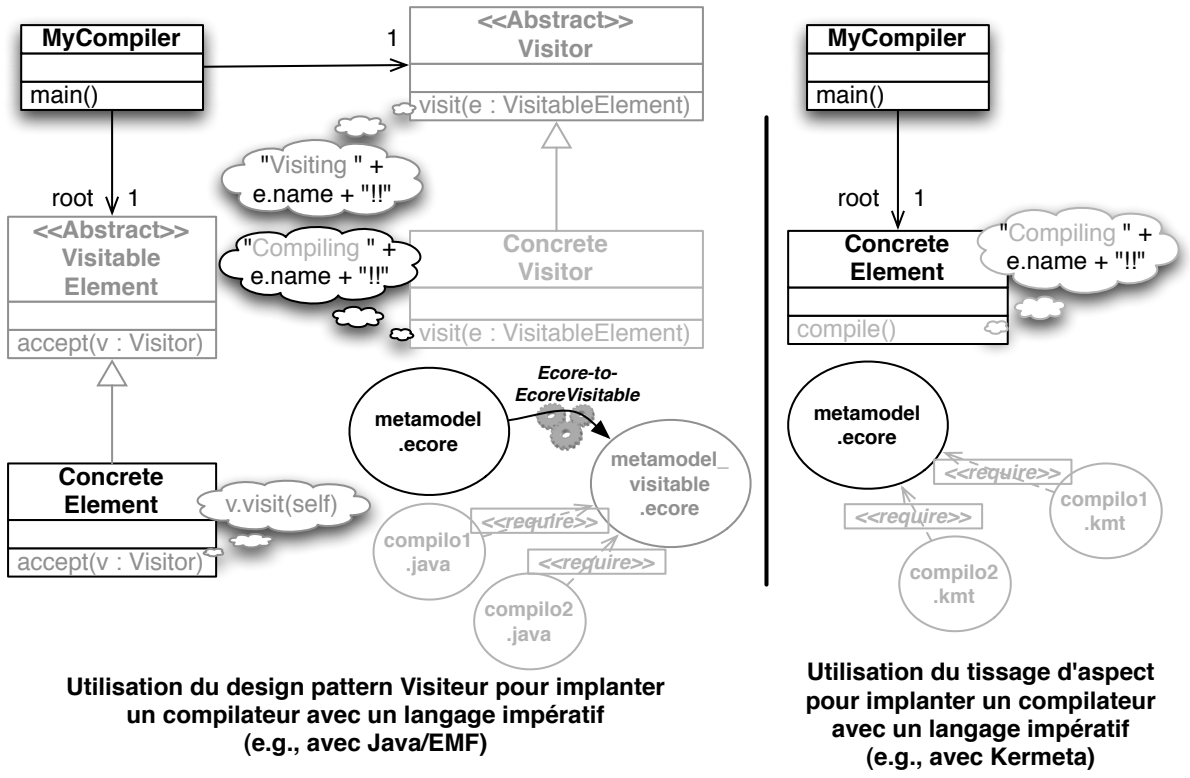


FIGURE 9.2 – Définition d'un compilateur : patron du visiteur vs. aspects

Listing 9.2 – Utilisation d'un compilateur en Kermeta

```
1  package kmLogo;
2  require "ASMLogo.ecore"      // import Logo metamodel (ie. abstract syntax)
3  require "StaticSemantics.ocl" // import OCL constraints definitions
4  require "Compiler.kmt"       // import compiler code (i.e. compile operations)
5  [...]
6  class Main {
7    operation Main(): Void is do
8      var rep : EMFRepository init EMFRepository.new
9      var LogoProgram : ASMLogo::Block
10     // load LogoProgram from its XMI file
11     LogoProgram ?= rep.getResource("Square.xmi").one
12     // Create a new Context for storing global data during the compilation
13     var context : Context init Context.new
14     // now compile the LogoProgram to NXC
15     LogoProgram.compile(context)
16   end
17 end
```

Velocity[3] et JSP (Java Server Pages)[4]. Il permet de générer du texte en utilisant un template KET (une sorte de patron) et un modèle. Les templates sont composés de deux éléments principaux : le texte qui doit être écrit sur la sortie et les tags (définis entre les balises <% et %>) qui doivent être interprétés pour générer des chaînes de caractères issues de calculs sur le modèle. Les templates KET sont ensuite vus par Kermeta comme des opérations qui prennent éventuellement des paramètres et qui renvoient le texte correspondant.

Dans notre cas, nous pouvons utiliser des templates pour générer les parties du code qui sont essentiellement écrites dans le langage cible avec un peu de navigation dans le modèle. Par exemple, nous pouvons l'utiliser pour la partie déclarant l'API de connexion au robot matériel, ou bien encore dans la partie correspondant aux entêtes des différents fichiers générés comme illustré dans le listing 9.3. Dans cet exemple, les lignes 1 à 7 définissent comment le template peux être appelé pour être intégré dans une transformation. On notera que ce template prend en paramètre le *Block* racine du programme Logo. Les lignes 23 à 27 évaluent des instructions Kermeta permettant de trier les instructions simples des déclarations de procédure. Grâce à la balise <%=, les lignes 30 à 32 permettent d'injecter dans le texte généré toutes les déclarations de procédure trouvées précédemment. De la même manière, les instructions primaires du programme Logo sont ajoutées à la tâche *main* du code NXC grâce aux lignes 36 à 38. Cette version s'appuie ainsi sur un sous ensemble des fonctions du compilateur vu précédemment tout en laissant une grande part du texte NXC à générer directement lisible dans le template.

Listing 9.3 – Utilisation de KET pour la transformation *Logo-vers-NXC*

```
1  <%@ket
2  package="kmLogo"
3  require="http://www.kermeta.org/kmLogo"
4  require="Compiler.kmt"
5  class="NXCCodeGenerator"
6  parameters="rootBlock : Block"
7  %>
8  /**
9   * This is the implementation of a LOGO Turtle for NXT
10  * Author : John Doe
11  * License : GPL
12  */
```

3. cf. http://velocity.apache.org/
4. cf. http://www.oracle.com/technetwork/java/javaee/jsp/

```
13
14 #include "NXCDefs.h"
15 // Correponds to D/d
16 #define ANGLE_FACTOR 18
17 // The Motors connections
18 #define PEN_MOTOR OUT_A
19 #define RIGHT_MOTOR OUT_B
20 #define LEFT_MOTOR OUT_C
21 #define MOVE_MOTORS OUT_BC
22 [...]
23 <% var subProcedures : Sequence<Instruction> init Sequence<Instruction>.new
24         var mainInstructions : Sequence<Instruction> init Sequence<Instruction>.new
25         rootBlock.instructions.each{ instruct | if instruct.instanceOf(ProcDeclaration)
26             then subProcedures.add(instruct)
27             else mainInstructions.add(instruct) end }%>
28
29 // Logo operations
30 <% subProcedures.each{ instruct | %>
31 <%=     instruct.compile(Context.new.initialize)%>
32 <% }%>
33
34 task main()
35 {
36 <% mainInstructions.each{ instruct | %>
37 <%=     instruct.compile(Context.new.initialize)%>
38 <% }%>
39 }
```

Les transformations à base de template peuvent être vues comme des transformations qui sont dirigées par la structure de l'artefact en sortie plutôt que la structure du modèle en entrée. C'est particulièrement utile lorsque l'on connait le modèle de sortie et que l'on veut le compléter ou le raffiner avec des informations que l'on cherche dans un autre modèle.

Contrairement aux approches de mapping utilisées dans la conception d'éditeur dans la section 7.2, l'approche par template ne permet pas de régénérer nativement le modèle à partir du texte. Elle est plutôt adaptée aux situations où l'on ne conserve pas nécessairement toute l'information du modèle dans le texte comme par exemple pour la génération de documentation. Elle aura en outre l'avantage d'être relativement intuitive et simple à mettre en œuvre.

9.3 Outils de refactoring par tissage de transformations de modèle à modèle

Le lecteur attentif aura pu remarquer que pour fonctionner correctement, le compilateur ci-dessus faisait l'hypothèse implicite que toutes les déclarations de fonctions Logo devaient être effectuées dans le bloc le plus externe, car le langage NXC ne supporte pas les déclarations de fonctions imbriquées. Comme rien dans le métamodèle Logo n'empêche un utilisateur de déclarer des fonctions imbriquées, nous devons soit l'empêcher de faire ça (par exemple en ajoutant des contraintes OCL additionnelles aux contraintes de bonne formation du langage, *i.e.* sa sémantique statique), soit effectuer un pré-traitement avant l'étape de compilation. Dans le but d'illustrer les capacités de Kermeta vis-à-vis des transformations de modèle vers modèle, nous allons décrire comment mettre en œuvre la seconde solution.

Nous avons donc besoin d'un nouvel aspect dans notre environnement de développement, que nous appellerons l'aspect *local-to-global* (voir le Listing 9.4) par référence à un exemple pris dans le tutorial du langage TXL [Cor06]. Nous allons utiliser une conception orientée objet très simple qui va déclarer une méthode vide *local2global* (prenant comme paramètre le bloc racine d'un programme Logo donné) dans la classe la plus haute dans la hiérarchie d'héritage du métamodèle Logo, à savoir la classe *Statement*. Il ne restera plus qu'à la redéfinir dans les classes pertinentes du métamodèle, comme la classe *ProcDeclaration* où nous avons à déplacer la déclaration courante vers le bloc racine, puis rappeler de manière récursive la méthode *local2global* sur son bloc (qui contient le corps de la fonction). Ensuite dans la classe *Block*, la méthode *local2global* doit itérer un appel récursif à *local2global* sur chaque instruction de ce bloc.

Listing 9.4 – Transformation *local-to-global* de Logo en Kermeta

```
1  package kmLogo;
2
3  require "ASMLogo.ecore"   // import Logo metamodel (ie. abstract syntax)
4  [...]
5  package ASMLogo {
6    aspect class Statement
7      method local2global(rootBlock: Block) is do
8      end
9    end
10   aspect class ProcDeclaration
11     method local2global(rootBlock: Block) is do
12       rootBlock.add(self)
13       block.local2global(rootBlock)
14     end
15   end
16   aspect class Block
17     method local2global(rootBlock: Block) is do
18       statements.each(i| i.local2global(rootBlock))
19     end
20   end
21 }
```

Notons que si l'on permet aussi des déclarations de fonctions (*ProcDeclaration*) à l'intérieur des structures de contrôle comme *Repeat* ou *If*, alors nous devrions aussi ajouter par aspect une méthode *local2global* pour visiter leur blocs (*thenPart* et *elsePart* dans le cas d'une instruction *If*).

On notera que la structure de cette transformation est similaire à celle utilisée dans la section 9.2, mais cette fois le contexte passé aux opérations est directement une référence (ici un *Block*) vers le modèle en cours de construction.

Une fois de plus cette préoccupation *local2global* est implémentée d'une manière modulaire en Kermeta, et peut être aisément ajoutée ou retirée de l'environnement de programmation Logo sans aucun impact sur le reste, et en particulier sur le métamodèle central de Logo. De plus, de nouvelles instructions peuvent être ajoutées en Logo (c'est-à-dire en étendant son métamodèle avec des nouvelles classes) sans impact significatif sur la préoccupation *local2global*, du moins dans la mesure où ces instructions ne comprennent pas de structure de bloc. Ce faible couplage est une bonne illustration des fonctionnalités de modularité avancée de Kermeta, permettant ainsi un développement parallèle plus facile et une maintenance simplifiée d'un environnement de développement pour un langage de modélisation spécifique à un domaine.

Conclusion

Nous avons voulu montrer dans ce livre à la fois les principes et les pratiques de l'ingénierie dirigée par les modèles (IDM). Loin d'une vision réductrice dans laquelle l'IDM est parfois cantonnée (par exemple comme une sorte de compilateur monolithique d'UML), nous avons présenté l'IDM comme une approche fondée sur la capture d'abstractions de différents aspects pertinents d'un domaine dans des métamodèles. Ces métamodèles peuvent ensuite être outillés pour répondre de manière *ad hoc* à des besoins spécifiques du développement logiciel, qui peuvent aller de problématiques de génération de code, de documentation, de configurations, jusqu'à des problématiques de validation et de vérification.

Devant les gains déjà obtenus et ceux encore attendus, la communauté de l'IDM est encore en pleine ébullition pour répondre à ces enjeux. La panoplie d'outils exploitables est déjà conséquente mais n'en est encore qu'à ses débuts. Certains points théoriques restent à éclaircir et les résultats scientifiques et techniques contribuent régulièrement à les compléter et les améliorer. Ce livre avait pour objectif de donner les bases pour bien comprendre comment sont conçus les outils existants et appréhender les différentes offres disponibles sur le marché. L'illustration par la pratique d'un panorama des outils les plus caractéristiques devrait permettre de les déployer et de les utiliser rapidement, et ainsi d'intégrer l'IDM dans les processus de développement existants. Enfin et surtout, nous espérons que la compréhension de l'approche permettra de concevoir de nouveaux outils adaptés à tout un ensemble de domaines spécifiques et ainsi de capitaliser le savoir faire des experts de ces domaines.

Dans un article récent intitulé *"Empirical Assessment of MDE in Industry"* [HWRK11], des chercheurs britanniques ont étudié en détail différentes expériences de déploiement de l'IDM dans différents secteurs industriels. Outre que leurs conclusions vont tout à fait dans le sens de ce qui a été exposé dans ce livre, en particulier sur le fait que réduire l'IDM à une chaîne de compilation pour un UML vu comme une sorte de super C++ est plutôt contre-productif, ces chercheurs tirent de leur étude quelques conclusions sur l'adoption de l'IDM dans l'industrie qu'il nous semble utile de résumer ici.

Tout d'abord un déploiement de l'IDM demande une approche progressive et itérative. Dans les cas étudiés, le déploiement de l'IDM a pu poser problème lorsque celui-ci a été fait de manière autocratique, sans compréhension des changements de processus nécessaires. Ayant potentiellement un impact important sur de nombreux processus de l'entreprise, l'adoption d'une démarche IDM ne peut s'affranchir des habituelles contraintes managériales liées à tous les changements de technologie, et en particulier des aspects formation.

Outre les gains de productivité souvent mentionnés comme motivation première pour l'adoption d'une approche IDM, les principaux bénéfices constatés par ces chercheurs britanniques se situent aussi en termes de contrôle et de communication, c'est-à-dire la possibilité de répondre rapidement à des changements d'exigences, et de mieux maîtriser les processus complexes d'entreprise. A cet égard, l'IDM peut donc être vue comme une alternative à la pratique controversée de l'offshore pour les tâches répétitives de codage d'application.

Livres de références

Livres en français

— Jean-Marie Favre, Jacky Estublier et Mireille Blay : *L'Ingénierie Dirigée par les Modèles : au-delà du MDA.* Informatique et Systèmes d'Information. Hermes Science, lavoisier édition, 2006
— Xavier Blanc : *MDA en action.* EYROLLES, mars 2005

Livres en anglais

— JosWarmer et Anneke Kleppe : *The Object Constraint Language : Getting Your Models Ready for MDA.* Addison-Wesley Longman Publishing Co., Inc., 2003
— Klaus Pohl, Günter Böckle et Frank J van der Linden : *Software Product Line Engineering : Foundations, Principles and Techniques.* Springer, 2005
— Anneke Kleppe, Jos Warmer et Wim Bast : *MDA Explained. The Model Driven Architecture : Practice and Promise.* Addison-Wesley, 2003
— Krzysztof Czarnecki et Ulrich W. Eisenecker : *Generative programming : methods, tools, and applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000
— David Steinberg, Franck Budinsky, Marcelo Paternostro et Ed Merks : *EMF : Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd édition, 2009

Bibliographie

[AK03] Colin Atkinson and Thomas Kuhne. Model-Driven Development : A Metamodeling Foundation. *IEEE Software*, 20(5) :36–41, 2003. (Cité page 4.)

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. (Cité page 13.)

[ATL05] INRIA ATLAS. KM3 : Kernel MetaMetaModel. Technical report, LINA & INRIA, Nantes, August 2005. (Cité page 15.)

[BC04] Elisa Baniassad and Siobhan Clarke. Theme : An approach for aspect-oriented analysis and design. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society. (Cité page 5.)

[BdLG07] Paolo Bottoni, Juan de Lara, and Esther Guerra. Action patterns for the incremental specification of the execution semantics of visual languages. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, pages 163–170. IEEE Computer Society, 2007. (Cité page 20.)

[Béz03] Jean Bézivin. *La transformation de modèles*. INRIA-ATLAS & Université de Nantes, 2003. Ecole d'Eté d'Informatique CEA EDF INRIA 2003, cours #6. (Cité page 25.)

[Béz04a] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, V(2) :21–24, 2004. (Cité page 4.)

[Béz04b] Jean Bézivin. Sur les principes de base de l'ingénierie des modèles. *RSTI-L'Objet*, 10(4) :145–157, 2004. (Cité pages 3, 11, et 23.)

[Béz05] Jean Bézivin. On the unification power of models. *Software and System Modeling (SoSym)*, 4(2) :171–188, 2005. (Cité page 3.)

[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)*, page 273, San Diego, USA, 2001. IEEE Computer Society Press. (Cité page 3.)

[BJRV05] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the Large and Modeling in the Small. In U. Aßmann, M. Aksit, and A. Rensink, editors, *Model Driven Architecture, European MDA Workshops : Foundations and Applications, MDAFA 2003 and MDAFA 2004, , June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46, Twente, The Netherlands, 2005. Springer. (Cité page 9.)

[BK05] Jean Bezivin and Ivan Kurtev. Model-based Technology Integration with the Technical Space Concept. In *Metainformatics Symposium*, Esbjerg, Denmark, 2005. Springer-Verlag. (Cité page 9.)

[Bla05] Xavier Blanc. *MDA en action*. EYROLLES, March 2005. (Cité page 10.)

[BSE03] Franck Budinsky, David Steinberg, and Raymond Ellersick. *Eclipse Modeling Framework : A Developer's Guide*. Addison-Wesley Professional, 2003. (Cité pages 15 et 26.)

[CCGT09] Benoit Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. *Journal of Software*, 4(9) :943–958, November 2009. (Cité page 18.)

[CEK01] Tony Clark, Andy Evans, and Stuart Kent. The Metamodelling Language Calculus : Foundation Semantics for UML. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *Lecture Notes In Computer Science*, pages 17–31, London, UK, 2001. Springer. (Cité page 21.)

[CESW04] Tony Clark, Andy Evans, Paul Sammut, and James Willans. Applied Metamodelling – A Foundation for Language Driven Development. version 0.1, 2004. (Cité pages 15, 21, et 22.)

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003. (Cité page 26.)

[CL00] Curtis Clifton and Gary T. Leavens. Multijava : Modular open classes and symmetric multiple dispatch for java. In *In OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000. (Cité page 43.)

[Com08] Benoit Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle – Application à l'ingénierie des procédés.* PhD thesis, Institut National Polytechnique, Université de Toulouse, July 2008. (Cité page 22.)

[Com10] Benoit Combemale. *Simulation et vérification de modèle par métamodélisation exécutable.* Editions Universitaires Européennes, June 2010. (Cité page 22.)

[Cor06] James R. Cordy. The txl source transformation language. *Sci. Comput. Program.*, 61(3) :190–210, 2006. (Cité page 79.)

[Cou90] Patrick Cousot. Methods and Logics for Proving Programs. In *Handbook of theoretical computer science (vol. B) : formal models and semantics*, pages 841–994. MIT Press, Cambridge, MA, USA, 1990. (Cité page 15.)

[CSAJ05] Kai Chen, Janos Sztipanovits, Sherif Abdelwalhed, and Ethan Jackson. Semantic Anchoring with Model Transformations. In *Proceedings of the First European Conference Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 3748 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2005. (Cité page 21.)

[CSW08a] Tony Clark, Paul Sammut, and James Willans. Applied Metamodelling – A Foundation for Language Driven Development. Second Edition, 2008. (Cité page 22.)

[CSW08b] Tony Clark, Paul Sammut, and James Willans. SUPERLANGUAGES – Developing Languages and Applications with XMF. First Edition, 2008. (Cité page 20.)

[EHS00] Gregor Engels, Reiko Heckel, and Stefan Sauer. Dynamic meta modeling : A graphical approach to the operational semantics of behavioral diagrams in UML. In *UML'00*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000. (Cité page 20.)

[FEB06] Jean-Marie Favre, Jacky Estublier, and Mireille Blay. *L'Ingénierie Dirigée par les Modèles : au-delà du MDA.* Informatique et Systèmes d'Information. Hermes Science, lavoisier edition, 2006. (Cité pages 4 et 9.)

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. pages 21–35. Addison-Wesley, 2000. (Cité page 6.)

[FGC+06] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The TOPCASED project : a Toolkit in OPen source for Critical Aeronautic SystEms Design. In *3rd European Congress* EMBEDDED REAL TIME SOFTWARE *(ERTS)*, Toulouse, France, January 2006. (Cité page 17.)

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and Albert Zündorf. Story diagrams : A new graph transformation language based on UML and Java. In *TAGT'98*, volume 1764 of *LNCS*. Springer, 1998. (Cité page 20.)

[GHJ95] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. (Cité page 3.)

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns : elements of reusable object-oriented software.* Addison-Wesley, 1995. (Cité pages 47, 67, et 69.)

[GLR+02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation : The Missing Link of MDA. In A. Corradini, H. Ehrig, H. Kreowski, and G. Rozenberg, editors, *Proceedings of the First International Conference on Graph Transformation (ICGT)*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105, Barcelona, Spain, October 2002. Springer. (Cité pages 23 et 25.)

[GRS09] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A semantic framework for metamodel-based languages. *Autom. Softw. Eng.*, 16(3-4) :415–454, 2009. (Cité page 21.)

[Hau05] Jan Hendrik Hausmann. *Dynamic Meta Modeling – A Semantics Description Technique for Visual Modeling Languages.* PhD thesis, University of Paderborn, 2005. (Cité pages 18, 20, et 22.)

[HbR00] David Harel and bernhard Rumpe. Modeling Languages : Syntax, Semantics and All That Stuff, Part I : The Basic Stuff. Technical report, Mathematics & Computer Science, Weizmann Institute Of Science, Weizmann Rehovot, Israel, August 2000. (Cité page 18.)

[HJPP02] Wai Ming Ho, Jean-Marc Jézéquel, François Pennaneac'h, and Noël Plouzeau. A toolkit for weaving aspect oriented UML designs. In *Proceedings of 1st ACM International Conference on Aspect Oriented Software Development, AOSD 2002*, Enschede, The Netherlands, April 2002. (Cité page 7.)

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling : What's the Semantics of "Semantics" ? *Computer*, 37(10) :64–72, 2004. (Cité pages 13 et 18.)

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 471–480. ACM, 2011. (Cité pages vi et 81.)

[JB06] Frédéric Jouault and Jean Bézivin. KM3 : a DSL for Metamodel Specification. In *Proceedings of the IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2006. (Cité page 15.)

[Jéz08] Jean-Marc Jézéquel. Model Driven Design and Aspect Weaving. *Journal of Software and Systems Modeling (SoSyM)*, 7(2), March 2008. (Cité page 20.)

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference, Proceedings of the Model Transformations in Practice Workshop*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Montego Bay, Jamaica, 2005. Springer. (Cité page 26.)

[Jou06] Frédéric Jouault. *Contribution à l'étude des langages de transformation de modèles*. PhD thesis, Université de Nantes, September 2006. (Cité page 26.)

[JPW+02] Jean-Marc Jézéquel, Noël Plouzeau, Torben Weis, , and Kurt Geihs. From contracts to aspects in uml designs. In *Proc. of the Workshop on Aspect-Oriented Modeling with UML at AOSD'02*, 2002. (Cité page 7.)

[KBA02] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological Spaces : An Initial Appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, Irvine, 2002. (Cité page 9.)

[KGKK02] Sabine Kuske, Martin Gogolla, Ralf Kollmann, and Hans-Jörg Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM)*, volume 2335 of *Lecture Notes In Computer Science*, pages 11–28, London, UK, 2002. Springer. (Cité page 20.)

[KKR06] Harmen Kastenberg, Anneke Kleppe, and Arend Rensink. Defining Object-Oriented Execution Semantics Using Graph Transformations. In R. Gorrieri and H. Wehrheim, editors, *Proceedings of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Lecture Notes in Computer Science*, pages 186–201, Bologna, Italy, June 2006. Springer-Verlag. (Cité page 20.)

[Kle06] Anneke Kleppe. MCC : A Model Transformation Environment. In *Proceedings of the First European Conference Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, pages 173–187, Bilbao, Spain, July 2006. (Cité page 24.)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, Jyväskylä, Finland, June 1997. (Cité page 3.)

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The Model Driven Architecture : Practice and Promise*. Addison-Wesley, 2003. (Cité page 10.)

[LMB+01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason IV, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Proceedings of the IEEE Workshop on Intelligent Signal Processing (WISP)*, Budapest, Hungary, May 2001. (Cité page 15.)

[LMV+07] Philippe Lahire, Brice Morin, Gilles Vanwormhoudt, Alban Gaignard, Olivier Barais, and Jean-Marc Jézéquel. Introducing variability into aspect-oriented modeling approaches. In *In Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*, Nashville, TN, USA, October 2007. (Cité page 7.)

[MB08] Slavisa Markovic and Thomas Baar. Semantics of OCL specified with QVT. *Software and System Modeling*, 7(4) :399–422, 2008. (Cité page 20.)

[Mey92] B. Meyer. Applying "design by contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10) :40–52, October 1992. (Cité pages 50 et 51.)

[Mey97] Bertrand Meyer. *Eiffel. Le Langage*. Intereditions, 1997. (Cité page 63.)

[MFBC10] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoit Combemale. Modeling modeling modeling. *Software and Systems Modeling (SOSYM)*, 2010. (Cité page 3.)

[MFJ05a] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer. (Cité page 7.)

[MFJ05b] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In L. Briand and C. Williams, editors, *Proceedings of the 8th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer. (Cité pages 15, 20, et 26.)

[MFV+05] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, October 2005. (Cité page 64.)

[Mil95] Robin Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, c. a. r. hoare edition, 1995. (Cité page 19.)

[Min68] Marvin Minsky. Matter, mind, and models. *Semantic Information Processing*, pages 425–432, 1968. (Cité pages 3 et 4.)

[MM03] Joaquin Miller and Jishnu Mukerji. *Model Driven Architecture (MDA) 1.0.1 Guide*. Object Management Group, Inc., June 2003. (Cité page 9.)

[OMG04] Human-usable textual notation 1.0 specification, 2004. http://www.omg.org/spec/HUTN/. (Cité pages 36 et 58.)

[OMG06a] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Specification*, January 2006. Final Adopted Specification http://www.omg.org/spec/MOF/. (Cité pages 8, 9, 15, et 26.)

[OMG06b] Uml diagram interchange 1.0 specification, 2006. http://www.omg.org/spec/UMLDI/. (Cité page 40.)

[OMG07a] Object Management Group, Inc. *Unified Modeling Language (UML) 2.1.2 Infrastructure*, November 2007. Final Adopted Specification. (Cité page 8.)

[OMG07b] Object Management Group, Inc. *Unified Modeling Language (UML) 2.1.2 Superstructure*, November 2007. Final Adopted Specification. (Cité pages 8 et 11.)

[OMG10] Object Management Group, Inc. *Object Constraint Language (OCL) 2.2 Specification*, February 2010. http://www.omg.org/spec/OCL/. (Cité page 16.)

[OMG11a] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.1*, January 2011. http://http://www.omg.org/spec/QVT/. (Cité pages 20 et 26.)

[OMG11b] Xml metadata interchange 2.4 - beta 2 specification, 2011. http://www.omg.org/spec/XMI/. (Cité page 39.)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, 2005. (Cité page 5.)

[PJJ+07] Simon Pickin, Claude Jard, Thierry Jéron, Jean-Marc Jézéquel, and Yves Le Traon. Test synthesis from UML models of distributed software. *IEEE Transactions on Software Engineering*, 33(4) :252–268, April 2007. (Cité page 7.)

[PKP06] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. An action semantics for MOF 2.0. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC)*, pages 1304–1305, New York, NY, USA, 2006. ACM. (Cité page 20.)

[RaJdLV08] José E. Rivera, Esther Guerra annd Juan de Lara, and Antonio Vallecillo. Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude. In *SLE 2008*, 2008. (Cité page 21.)

[Rei85] Wolfgang Reisig. *Petri nets : an introduction*. Springer-Verlag, New York, NY, USA, 1985. (Cité page 21.)

[RG02] Mark Richters and Martin Gogolla. OCL : Syntax, semantics, and tools. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL : The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002. (Cité page 49.)

[Roz97] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation : volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. (Cité page 20.)

[RRDV07] J. Raul Romero, Jose E. Rivera, Francisco Duran, and Antonio Vallecillo. Formal and Tool Support for Model Driven Engineering with Maude. *Journal of Object Technology, TOOLS EUROPE*, 6(9) :187–207, 2007. (Cité page 21.)

[RV07] José E. Rivera and Antonio Vallecillo. Adding behavioral semantics to models. In *EDOC 2007*, pages 169–180. IEEE Computer Society, October 2007. (Cité page 21.)

[SBPM09] David Steinberg, Franck Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009. (Cité page 37.)

[Sei03] Ed Seidewitz. What models mean. *IEEE Software*, 20(5) :26–32, 2003. (Cité pages 3 et 4.)

[Sol00] Richard Soley. *Model Driven Architecture (MDA), Draft 3.2.* Object Management Group, Inc., November 2000. (Cité page 9.)

[Tae03] Gabriele Taentzer. AGG : A Graph Transformation Environment for Modeling and Validation of Software. In *Proceedings of the Second International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453, Charlottesville, VA, USA, October 2003. Springer. (Cité page 20.)

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing : A Tools Approach.* Morgan Kaufmann, 2007. (Cité page 37.)

[Win93] Glynn Winskel. *The formal semantics of programming languages : an introduction.* MIT Press, Cambridge, MA, USA, 1993. 0-262-23169-7. (Cité page 18.)

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language : Getting Your Models Ready for MDA.* Addison-Wesley Longman Publishing Co., Inc., 2003. (Cité page 16.)

Table des figures

Listings

Index