Institut National de Recherche en Informatique et Automatique — Université de Lorraine
École doctorale IAEM (Informatique, Automatique, Électronique et Mathématiques)

Un îlot formel pour les transformations de modèles qualifiables

THÈSE

présentée et soutenue publiquement le 12 septembre 2014

pour l'obtention du

Doctorat de l'Université de Lorraine

(spécialité informatique)

par

Jean-Christophe Bach

Composition du jury

Rapporteurs :	Antoine Beugnard	Professeur, Telecom Bretagne
	Laurence Duchien	Professeur, Université de Lille 1, LIFL
Examinateurs :	Mark van den Brand	Professeur, Eindhoven University of Technology
	Benoît Combemale	Maître de conférences, Université de Rennes 1, IRISA, Inria
	Marc Pantel	Maître de conférences, Université de Toulouse, INPT, IRIT
Encadrant de thèse :	Pierre-Etienne Moreau	Professeur, Université de Lorraine, École des Mines de Nancy

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Remerciements

Je tiens à remercier mes encadrants de thèse, Marc Pantel et Pierre-Etienne Moreau, qui ont accepté de m'encadrer en tant que doctorant, malgré la distance pour l'un et les responsabilités administratives pour l'autre.

Je voudrais aussi remercier les personnes qui m'ont fait l'honneur d'accepter de faire partie de mon jury de soutenance : Laurence Duchien et Antoine Beugnard qui ont accepté d'être rapporteurs de ce travail de thèse, Benoît Combemale et Mark van den Brand d'en être examinateurs.

Merci aux membres et ex-membres de l'équipe Pareo, à Loïc ainsi qu'au groupe du projet quarteFt, sans qui ce travail n'aurait pas été possible.

Et bien sûr, un merci à toutes celles et ceux qui m'ont accompagnés durant cette thèse. En particulier Karën qui a été présente tout au long de ma thèse malgré sa vie bien chargée, son mari, son fils et ses concours.

Merci à Pierre, Gilles, Martin et ses trolls (et pas forcément dans cet ordre) pour les discussions scientifiques, techniques et politiques, ainsi que les rafraîchissantes pauses de « bidouille & médiation scientifique ».

Évidemment, je n'oublie pas mes impitoyables relectrices-correctrices — Chloé, Karën et Marion — et leur rigueur qui m'ont aidé à améliorer la qualité grammaticale, orthographique et typographique de mon tapuscrit.

Une mention spéciale à mes soutiens distants : à Cláudia qui m'a soutenu par la mise en place d'une filière brésilienne d'importation de *Tang* et de café ; à Valérie depuis Rennes qui rédigeait en parallèle.

Et parce que dans la thèse il n'y a pas que la thèse, merci à toutes les personnes avec qui j'ai discuté et passé du temps : Clément, Sergueï et le triumvirat Céline-Fabrice-Guillaume pour les discussions politico-syndicales ; Paul, Pini, Jonathan, Christophe et tous les connectés du *canal de discussion relayée par Internet* `#linux` pour leurs trolls et débats libro-informatiques ; Ju, Séb, les membres de LDN et autres défenseurs du logiciel libre et d'un Internet neutre ; Nico lors de mes retours pour mise au vert à Lille.

Une pensée pour les adorables petits bouts — Léonard, Margaux et Thomas — qui m'ont aidé à me changer les idées.

Enfin, merci à mes parents sans qui je n'aurais pu commencer ni finir cette épreuve.

Table des matières

Table des figures

Introduction

Le sujet de cette thèse est l'élaboration de méthodes et outils pour le développement logiciel fiable s'appuyant sur des transformations de modèles. Plus précisément, la question est de savoir comment améliorer la confiance dans un logiciel et dans son processus de développement alors que les chaînes de développement se complexifient et que les outils se multiplient.

L'industrie adopte progressivement les techniques de l'Ingénierie Dirigée par les Modèles. L'un des intérêts de ce domaine est d'accélérer le développement logiciel à moindre coût par l'usage de langages dédiés et d'outils de génération de code. Dans ce contexte, les logiciels sont donc en partie issus de chaînes de transformations opérées sur des modèles jusqu'à la génération du code et sa compilation.

Dans le cadre du développement de systèmes critiques, les logiciels doivent être vérifiés afin d'être certifiés. Se pose alors la question de la qualification des logiciels utilisés dans les chaînes de développement des systèmes critiques. Il faut s'assurer que les outils n'introduisent pas de bogue à chaque étape du processus de développement et qu'une trace puisse être conservée, de la spécification au code.

Le langage Tom *repose sur le calcul de réécriture et fournit des fonctionnalités telles que le filtrage à des langages généralistes comme* Java *ou* Ada*. Des constructions de haut niveau permettent de décrire formellement des algorithmes. Il offre la possibilité d'établir des règles de transformation pour écrire des outils de transformation sûrs.*

La finalité de cette thèse est donc de proposer des méthodes et outils pour exprimer des transformations de modèles qualifiables. On s'intéresse à fournir des constructions dédiées pour décrire une transformation de modèles et pour assurer la traçabilité entre le modèle source et le modèle cible. Le but étant alors de donner des éléments de confiance supplémentaires à l'utilisateur responsable de la vérification de la chaîne de développement.

Contexte et motivations

Les systèmes se complexifiant, l'ingénierie dirigée par les modèles (IDM) a apporté des solutions pour faciliter et accélérer le développement logiciel. Ce domaine a véritablement pris son essor à la fin du XXème siècle avec la publication de l'initiative MDA (*Model Driven Architecture*) par l'OMG (*Object Management Group*). L'industrie a adopté les méthodes et technologies issues du monde des modèles, y compris dans le cadre du développement de systèmes critiques. Cependant, si la manière de développer un logiciel ainsi que les technologies utilisées ont changé depuis les débuts de l'informatique, les contraintes liées à la maintenance (évolution du logiciel, déboguage) ainsi qu'à la fiabilité (qualification, certification) persistent. Les technologies des domaines critiques tels que l'aéronautique, l'automobile et la médecine reposant de plus en plus sur l'informatique, il est particulièrement important de s'assurer du bon fonctionnement des logiciels avant leur mise en production. Pour cela, il est nécessaire de les vérifier. Plusieurs approches complémentaires sont disponibles pour augmenter la confiance en un logiciel : le test, la preuve, la simulation et le *model-checking*. Chacune d'entre elles comporte ses spécificités, ses avantages et ses inconvénients. Dans le cadre de la qualification et de la certification, ces approches sont rarement suffisantes une à une et sont donc généralement combinées ou pratiquées en parallèle. Lors de la certification, il est nécessaire d'avoir une confiance forte dans les outils utilisés dans le processus de développement. Cette confiance est accordée par leur qualification qui exige une traçabilité entre la spécification et l'implémentation d'un logiciel. L'intégration de l'IDM dans les chaînes de développement de systèmes critiques tendant à se généraliser, il est fondamental de développer des méthodes et des outils pour aider au processus de qualification des nouveaux outils.

Dans ce contexte, nous nous proposons de travailler à l'élaboration de méthodes et d'outils permettant d'apporter des éléments de confiance pour la qualification du logiciel. Nous nous plaçons à la frontière de deux domaines : celui de l'Ingénierie Dirigée par les Modèles ainsi que celui de la réécriture de termes. L'industrie a vu l'intérêt pratique de l'IDM et a adopté les outils qui en sont issus. Les méthodes formelles, en particulier le domaine de la réécriture qui nous intéresse, sont quant à elles moins connues, mais indispensables pour un développement logiciel de qualité. Se placer à la frontière des deux domaines permet de tirer le meilleur des deux mondes et de pousser l'utilisation des méthodes formelles dans l'ingénierie du logiciel, que ce soit dans un cadre académique ou industriel.

L'un des piliers de l'IDM réside dans les transformations de modèles. Nous souhaitons leur apporter plus de fiabilité en fournissant des outils s'appuyant sur le langage Tom. Celui-ci repose sur la réécriture et le filtrage pour manipuler et transformer des structures complexes. Il s'intègre au sein de langages généralistes tels que Java et offre des constructions dédiées permettant à l'utilisateur d'appréhender et d'exprimer plus aisément des algorithmes complexes à mettre en œuvre dans un langage généraliste. Fournir des constructions de haut niveau est un moyen de donner plus de confiance à l'utilisateur dans le code résultant : les algorithmes sont exprimés plus formellement et il est donc plus facile de raisonner dessus à des fins de vérification du logiciel. Un autre avantage d'une approche reposant sur des constructions dédiées réside dans la réduction des coûts de développement : l'usage de générateurs de code permet de réduire la quantité de travail et le temps par rapport à un développement manuel.

Contributions

Dans ce contexte, la contribution de cette thèse comprend trois volets principaux :

Transformation de modèles par réécriture : Nous avons développé une méthode de transformation de modèles par réécriture de termes. Pour cela, nous opérons une transformation permettant de manipuler un modèle sous la forme d'un terme algébrique. Nous nous appuyons ensuite sur les stratégies de réécriture du langage Tom pour mettre en œuvre la méthode de transformation. Cette méthode se déroule en deux temps : nous effectuons d'abord une transformation par parties qui fournit des résultats partiels, puis

nous résolvons ces résultats intermédiaires pour former le modèle cible résultant. Nous avons donc étendu le langage Tom en proposant un îlot formel dédié pour transformer les modèles. Ce nouvel îlot implémente notre méthode de transformation par réécriture.

Représentation algébrique de modèles : Pour transformer un modèle dans notre contexte de réécriture de termes, il est nécessaire d'opérer au préalable un changement d'espace technologique. Pour ce faire, nous proposons un outil permettant de donner une vue algébrique d'un modèle. Il traduit un métamodèle donné en une signature algébrique intégrable dans notre environnement Tom.

Traçabilité d'une transformation de modèles : Afin de répondre aux exigences du processus de qualification, nous avons travaillé sur la traçabilité des transformations et avons proposé une traçabilité de spécification. Nous l'avons implémentée par un îlot formel dédié permettant d'ajouter une forme de traçabilité au sein d'un langage généraliste tel que Java.

Durant ce travail de thèse, nous nous sommes attelés à développer des outils opérationnels implémentant notre travail. Nous avons toujours gardé à l'esprit qu'ils doivent pouvoir être aussi utilisés par des utilisateurs extérieurs en vue d'un usage dans un cadre industriel. Nous avons donc expérimenté et amélioré nos outils afin de leur permettre un passage à l'échelle. De plus, tout ce que nous avons développé est disponible librement et gratuitement sur le site du projet Tom. L'ensemble de nos expériences reposant sur du code ouvert documenté ainsi que sur des modèles diffusés librement, cela assure leur reproductibilité par un utilisateur averti. Cet aspect d'ouverture et de diffusion libre des résultats et des données s'inscrit dans le mouvement de l'*open science*, qui est indispensable à une recherche vérifiable et reproductible. Nous posons ainsi les conditions idéales pour tout utilisateur souhaitant tester, vérifier, utiliser, modifier et étendre nos travaux.

Plan de la thèse

La suite de ce document se décompose en huit chapitres répartis dans deux grandes parties. La première consiste en un état de l'art de trois chapitres, traitant trois thématiques liées au contexte de cette thèse. La seconde constitue les contributions de ce travail de thèse. Nous résumons brièvement chaque chapitre.

Conseils et guide de lecture : Chaque chapitre de la première partie peut être lu indépendamment des autres. Les chapitres 4 et 5 expliquent le fond du travail de thèse, tandis que les chapitres 6, 7 et 8 sont plus techniques. Le chapitre 6 décrit les travaux d'implémentation des deux chapitres précédents. Le chapitre 7 illustre l'utilisation des outils appliqués sur deux cas d'étude. Le chapitre 8 comprend des résultats expérimentaux et donne des perspectives d'évolution technique de nos outils.

Partie I : État de l'art

Les trois chapitres composant l'état de l'art traitent de la réécriture et du langage Tom 1, des transformations de modèles 2 ainsi que de la vérification du logiciel 3.

Chapitre 1 – Un langage basé sur la réécriture : Tom

Dans ce chapitre, nous donnons le cadre formel de la réécriture et définissons toutes les notions importantes sous-jacentes au langage Tom. Nous le décrivons ensuite et détaillons les différentes constructions le composant. Nous nous appuyons sur des exemples simples pour les illustrer. Ce chapitre peut constituer un manuel court du langage Tom.

Chapitre 2 – Transformations de modèles

Ce chapitre donne le cadre de la modélisation et des transformations de modèles. Nous donnons une taxonomie des transformations qui se répartissent selon les critères liés aux

métamodèles et aux changements de niveau d'abstraction. Nous expliquons aussi quelles sont les approches principales pour transformer des modèles et quels sont les outils existants.

Chapitre 3 – Vérification du logiciel

Ce chapitre décrit le contexte de la vérification du logiciel et plus particulièrement celui de la qualification et de la certification. C'est dans ce contexte que s'inscrit notre travail de thèse.

Partie II : Contributions

La seconde partie regroupe les contributions de ce travail de thèse, découpées en cinq chapitres résumés ci-après.

Chapitre 4 – Transformations de modèles par réécriture

Dans ce chapitre, nous exposons notre approche de transformation de modèles dans notre environnement utilisant la réécriture. Elle consiste à représenter des modèles sous la forme de termes en opérant un changement d'espace technologique, puis à effectuer une transformation par parties du modèle source. Chaque partie est transformée sans notion d'ordre grâce à une stratégie de réécriture. L'ensemble des transformations partielles est suivi d'une phase de résolution du résultat partiel et produit le modèle cible.

Chapitre 5 – Spécification et traçabilité des transformations

Dans ce chapitre, nous exprimons la traçabilité d'une transformation de modèle exigée par le processus de qualification. Elle consiste à lier les sources aux cibles de la transformation en fonction de la spécification.

Chapitre 6 – Outils pour exprimer une transformation de modèles en Tom

Dans ce chapitre, nous détaillons les outils développés durant cette thèse pour la mise en œuvre des mécanismes expliqués dans les chapitres 4 et 5. Nous expliquons donc comment nous opérons le changement d'espace technologique, en quoi consiste le nouvel îlot formel du langage Tom pour exprimer une transformation de modèles. Nous décrivons l'implémentation au sein du projet Tom.

Chapitre 7 – Études de cas : illustration et utilisation du langage

Dans ce chapitre, nous illustrons l'utilisation de nos outils sur deux études de cas : *SimplePDLToPetriNet* et l'aplatissement d'une hiérarchie de classe. La première étude permet de détailler le processus complet d'écriture. L'étude de la seconde transformation a pour but de montrer les points d'amélioration de nos outils et d'ouvrir des perspectives de recherche et de développement suite à ce travail.

Chapitre 8 – Résultats expérimentaux

Ce chapitre est composé de résultats expérimentaux pour donner une première évaluation de nos outils. Nous les avons testés sur des modèles de tailles importantes afin de les pousser à leurs limites en termes de performances et de valider leur potentiel usage dans un cadre industriel. Nous décrivons aussi l'évolution de nos outils suite aux premières expériences et nous tirons certaines conclusions.

Première partie

État de l'art

Chapitre 1

Un langage basé sur la réécriture : Tom

Dans ce chapitre, nous abordons les notions élémentaires utiles à la lecture de ce document. Nous présentons d'abord la réécriture de termes, puis nous décrivons le langage Tom [MRV03, BBK+07] et nous terminons par les ancrages formels.

1.1 Réécriture

1.1.1 Signature algébrique et termes

Cette section traite des notions de base concernant les algèbres de termes du premier ordre.

Définition 1 (Signature)**.** *Une signature $\mathcal{F}$ est un ensemble fini d'opérateurs (ou symboles de fonction), dont chacun est associé à un entier naturel par la fonction d'arité, $ar : \mathcal{F} \to \mathbb{N}$. $\mathcal{F}_n$ désigne le sous-ensemble de symboles d'arité n, c'est-à-dire $\mathcal{F}_n = \{f \in \mathcal{F} \mid ar(f) = n\}$. L'ensemble des constantes est désigné par $\mathcal{F}_0$.*

On classifie parfois les termes selon leur type dans des *sortes*. On parle alors de signature multi-sortée.

Définition 2 (Signature algébrique multi-sortée)**.** *Une signature multi-sortée est un couple $(\mathcal{S}, \mathcal{F})$ où $\mathcal{S}$ est un ensemble de sortes et $\mathcal{F}$ est un ensemble d'opérateurs sortés défini par $\mathcal{F} = \bigcup_{S_1,\ldots,S_n,S \in \mathcal{S}} \mathcal{F}_{S_1,\ldots,S_n,S}$. Le rang d'un symbole de fonction $f \in \mathcal{F}_{S_1,\ldots,S_n,S}$ noté $rank(f)$ est défini par le tuple $(S_1,\ldots,S_n,S)$ que l'on note souvent $f : S_1 \times \ldots \times S_n \to S$*

Définition 3 (Terme)**.** *Étant donné un ensemble infini dénombrable de variables $\mathcal{X}$ et une signature $\mathcal{F}$, on définit l'ensemble des termes $\mathcal{T}(\mathcal{F}, \mathcal{X})$ comme le plus petit ensemble tel que :*

- *$\mathcal{X} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$: toute variable de $\mathcal{X}$ est un terme de $\mathcal{T}(\mathcal{F}, \mathcal{X})$;*
- *pour tous $t_1, \ldots, t_n$ éléments de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ et pour tout opérateur $f \in \mathcal{F}$ d'arité n, le terme $f(t_1, \ldots, t_n)$ est un élément de $\mathcal{T}(\mathcal{F}, \mathcal{X})$.*

Pour tout terme t de la forme $f(t_1, \ldots, t_n)$, le symbole de tête de t, noté $symb(t)$ est, par définition, l'opérateur f.
Un symbole de fonction dont l'arité est variable est appelé *opérateur variadique*, c'est-à-dire qu'il prend un nombre arbitraire d'arguments.

Définition 4 (Variables d'un terme)**.** *L'ensemble $\mathcal{V}ar(t)$ des variables d'un terme t est défini inductivement comme suit :*

— $\mathcal{V}ar(t) = \emptyset$, *pour* $t \in \mathcal{F}_0$;
— $\mathcal{V}ar(t) = \{t\}$, *pour* $t \in \mathcal{X}$;
— $\mathcal{V}ar(t) = \bigcup_{i=1}^{n} \mathcal{V}ar(t_i)$, *pour* $t = f(t_1, \ldots, t_n)$.

On note souvent $a, b, c, \ldots$ les constantes et $x, y, z, \ldots$ les variables.
On représente les termes sous forme arborescente. Par exemple, on peut représenter le terme $f(x, g(a))$ comme dans la figure 1.1 :

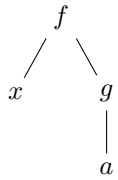


FIGURE 1.1 – Exemple de représentation arborescente d'un terme.

Définition 5 (Terme clos)**.** *Un terme t est dit clos s'il ne contient aucune variable, c'est-à-dire si $\mathcal{V}ar(t) = \emptyset$. On note $\mathcal{T}(\mathcal{F})$ l'ensemble des termes clos.*

Chaque nœud d'un arbre peut être identifié de manière unique par sa position.

Définition 6 (Position)**.** *Une position dans un terme t est représentée par une séquence ω d'entiers naturels, décrivant le chemin de la racine du terme jusqu'à un nœud $t_{|\omega}$ du terme. Un terme u a une occurrence dans t si $u = t_{|\omega}$ pour une position ω dans t.*

On notera $\mathcal{P}os(t)$ l'ensemble des positions d'un terme t et $t[t']_\omega$ le remplacement du sous-terme de t à la position ω par t'.

Par exemple, la figure 1.2 illustre la notation des positions pour le terme $t = f(x, g(a))$. On obtient l'ensemble des positions $\mathcal{P}os(t) = \{\epsilon, 1, 2, 21\}$ ce qui correspond respectivement aux sous-termes $t_{|\epsilon} = f(x, g(a))$, $t_{|1} = x$, $t_{|2} = g(a)$ et $t_{|21} = a$.

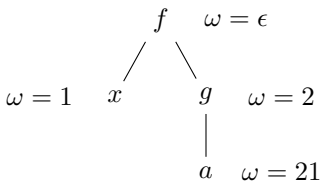


FIGURE 1.2 – Notation des positions dans un terme.

1.1.2 Filtrage

Une substitution est une opération de remplacement, uniquement définie par une fonction des variables vers les termes clos.

Définition 7 (Substitution)**.** *Une substitution σ est une fonction de $\mathcal{X}$ vers $\mathcal{T}(\mathcal{F})$, notée lorsque son domaine $\mathcal{D}om(\sigma)$ est fini, $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. Cette fonction s'étend de manière unique en un endomorphisme $\sigma' : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ sur l'algèbre des termes, qui est défini inductivement par :*

— $\sigma'(x) = \begin{cases} \sigma(x) \text{ si } x \in \mathcal{D}om(\sigma) \\ x \text{ sinon} \end{cases}$
— $\sigma'(f(t_1, \ldots, t_n)) = f(\sigma'(t_1), \ldots, \sigma'(t_n))$ *pour tout symbole de fonction* $f \in \mathcal{F}_n$.

Définition 8 (Filtrage)**.** *Étant donnés un motif $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et un terme clos $t \in \mathcal{T}(\mathcal{F})$, p filtre t, noté $p \ll t$, si et seulement s'il existe une substitution σ telle que $\sigma(p) = t$:*

$$p \ll t \Leftrightarrow \exists \sigma, \sigma(p) = t$$

On parle de *filtrage unitaire* lorsqu'il existe une unique solution à l'équation de filtrage. Si plusieurs solutions existent, le filtrage est *non unitaire*.

Le filtrage peut aussi être *modulo une théorie équationnelle*. Cela signifie que l'on a associé une théorie équationnelle au problème de filtrage.

Définition 9 (Filtrage modulo une théorie équationnelle)**.** *Étant donnés une théorie équationnelle $\mathcal{E}$, un motif $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et un terme clos $t \in \mathcal{T}(\mathcal{F})$, p filtre t modulo $\mathcal{E}$, noté $p \ll_{\mathcal{E}} t$, si et seulement s'il existe une substitution σ telle que $\sigma(p) =_{\mathcal{E}} t$, avec $=_{\mathcal{E}}$ l'égalité modulo $\mathcal{E}$:*

$$p \ll_{\mathcal{E}} t \Leftrightarrow \exists \sigma, \sigma(p) =_{\mathcal{E}} t$$

Dans la suite de cette section, nous allons expliquer ce concept, donner des exemples de théories équationnelles et illustrer notre propos.

Une paire de termes (l, r) est appelée *égalité*, *axiome équationnel* ou *équation* selon le contexte, et est notée $(l = r)$. Une théorie équationnelle peut être définie par un ensemble d'égalités. Elle définit une classe d'équivalence entre les termes. Dans la pratique, les théories équationnelles les plus communes sont l'associativité, la commutativité et l'élément neutre (ainsi que leurs combinaisons).

Définition 10 (Opérateur associatif)**.** *Un opérateur binaire f est associatif si $\forall x, y, z \in \mathcal{T}(\mathcal{F}, \mathcal{X}), f(f(x, y), z) = f(x, f(y, z))$.*

Par exemple, l'addition et la multiplication sont associatives sur l'ensemble des réels $\mathbb{R}$: $((x + y) + z) = (x + (y + z))$ et $((x \times y) \times z)) = (x \times (y \times z))$. En revanche, la soustraction sur $\mathbb{R}$ n'est pas associative : $((x - y) - z) \neq (x - (y - z))$.

Définition 11 (Opérateur commutatif)**.** *Un opérateur binaire f est commutatif si $\forall x, y \in \mathcal{T}(\mathcal{F}, \mathcal{X}) f(x, y) = f(y, x)$.*

Par exemple, l'addition et la multiplication sont commutatives sur $\mathbb{R}$, ainsi $x + y = y + x$ et $x \times y = y \times x$. Ce qui n'est pas le cas de la soustraction sur $\mathbb{R}$, en effet $x - y \neq y - x$.

Définition 12 (Élément neutre)**.** *Soit un opérateur binaire f et $x \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, la constante $e \in \mathcal{T}(\mathcal{F})$ est :*

- *neutre à gauche pour f si $f(e, x) = x$;*
- *neutre à droite pour f si $f(x, e) = x$;*
- *neutre si elle est neutre à gauche et neutre à droite pour f.*

Pour illustrer cette notion d'élément neutre, examinons la constante 0 avec l'addition sur l'ensemble des réels $\mathbb{R}$:

- $0 + x = x$, 0 est donc neutre à gauche pour l'addition ;
- $x + 0 = x$, 0 est donc neutre à droite pour l'addition ;
- on en déduit que 0 est neutre pour l'addition sur $\mathbb{R}$.

On note généralement A, U et C les théories équationnelles engendrées respectivement par l'équation d'associativité, l'équation de neutralité et celle de commutativité. On note AU la théorie engendrée par les équations d'associativité et de neutralité.

La théorie associative est associée aux opérateurs binaires. Pour des raisons techniques, elle est souvent associée à une syntaxe variadiques dans les langages de programmation fondés sur la réécriture. Par exemple, l'opérateur variadique *list* est simulé par l'opérateur *nil* d'arité nulle, ainsi que par l'opérateur binaire *cons*. Cela permet d'écrire que le terme

$list(a, b, c)$ est équivalent au terme $list(list(a, b), c)$, et qu'ils peuvent être représentés par $cons(a, cons(b, cons(c, nil)))$.

On peut alors définir des opérations modulo une théorie équationnelle : on parlera de *filtrage équationnel* lorsque le filtrage sera associé à une telle théorie. Pour illustrer cette notion, prenons deux exemples simples de filtrage.

Exemple 1 : Le filtrage associatif avec élément neutre (AU) — aussi appelé filtrage de liste — est un problème bien connu en réécriture. Il permet d'exprimer facilement des algorithmes manipulant des listes. En considérent $X1*$ et $X2*$ comme ides variables représentant 0 ou plusieurs éléments d'une liste, le problème de filtrage $list(X1*, x, X2*) \ll list(a, b, c)$ admet trois solutions : $\sigma_1 = \{x \to a\}$, $\sigma_2 = \{x \to b\}$ et $\sigma_3 = \{x \to c\}$.

Exemple 2 : Illustrons le filtrage associatif-commutatif (AC) et l'addition, dont les axiomes d'associativité et commutativité sont les suivants :

— $\forall x, y,\ plus(x, y) = plus(y, x)$;

— $\forall x, y,\ plus(x, plus(y, z)) = plus(plus(x, y), z)$.

Le problème de filtrage $plus(x, y) \ll plus(a, b)$ présente deux solutions distinctes modulo AC : $\sigma_1 = \{x \to a, y \to b\}$ et $\sigma_2 = \{x \to b, y \to a\}$.

1.1.3 Réécriture

En réécriture, on oriente des égalités que l'on appelle des *règles de réécriture* et qui définissent un calcul.

Définition 13 (Règle de réécriture)**.** *Une règle de réécriture est un couple (l,r) de termes dans $\mathcal{T}(\mathcal{F}, \mathcal{X})$, notée $l \to r$. l est appelé membre gauche de la règle, et r membre droit.*

Un exemple de règle de réécriture est l'addition de n'importe quel entier x avec 0 :

$$x + 0 \ \to \ x$$

Un autre exemple un peu plus complexe de règle de réécriture est la distributivité de la multiplication par rapport à l'addition dans un anneau, comme illustré par la figure 1.3.

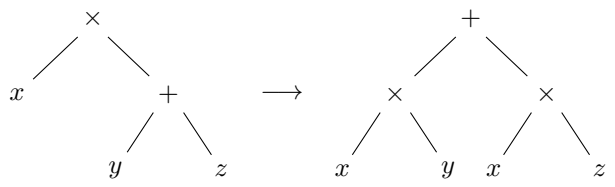


FIGURE 1.3 – Exemple de règle de réécriture : distributivité de la multiplication par rapport à l'addition dans un anneau, à savoir $x \times (y + z) \to (x \times y) + (x \times z)$.

Définition 14 (Système de réécriture)**.** *Un système de réécriture sur les termes est un ensemble de règles de réécriture (l,r) tel que :*

— *les variables du membre droit de la règle font partie des variables du membre gauche ($\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$) ;*

— *le membre gauche d'une règle n'est pas une variable ($l \notin \mathcal{X}$).*

Définition 15 (Réécriture)**.** *Un terme $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ se réécrit en t' dans un système de réécriture $\mathcal{R}$, ce que l'on note $t \to_{\mathcal{R}} t'$, s'il existe :*

— *une règle $l \to r \in \mathcal{R}$;*

— *une position* ω *dans* t *;*
— *une substitution* σ *telle que* $t_{|\omega} = \sigma(l)$ *et* $t' = t[\sigma(r)]_\omega$.

On appelle *radical* le sous-terme $t_{|\omega}$.

Pour une relation binaire $\rightarrow$, on note $\xrightarrow{*}$ sa fermeture transitive et réflexive. La fermeture transitive, réflexive et symétrique de $\rightarrow$ — qui est alors une relation d'équivalence — est notée $\overset{*}{\leftrightarrow}$.

Définition 16 (Forme normale)**.** *Soit* $\rightarrow$ *une relation binaire sur un ensemble* T*. Un élément* $t \in T$ *est réductible par* $\rightarrow$ *s'il existe* $t' \in T$ *tel que* $t \rightarrow t'$*. Dans le cas contraire, on dit qu'il est irréductible. On appelle forme normale de* t *tout élément* t' *irréductible de* T *tel que* $t \xrightarrow{*} t'$*. Cette forme est unique.*

Deux propriétés importantes d'un système de réécriture sont la *confluence* et la *terminaison*. Lorsque l'on souhaite savoir si deux termes sont équivalents, on cherche à calculer leurs formes normales et à vérifier si elles sont égales. Cela n'est possible que si la forme normale existe et qu'elle est unique. Une forme normale existe si $\rightarrow$ *termine*, et son unicité est alors assurée si $\rightarrow$ est *confluente*, ou si elle vérifie la *propriété de Church-Rosser*, qui est équivalente.

Définition 17 (Terminaison)**.** *Une relation binaire* $\rightarrow$ *sur un ensemble* T *est dite terminante s'il n'existe pas de suite infinie* $(t_i)_{i\geq 1}$ *d'éléments de* T *telle que* $t_1 \rightarrow t_2 \rightarrow \cdots$.

Définition 18 (Confluence)**.** *Soit une relation binaire* $\rightarrow$ *sur un ensemble* T*.*

(a) $\rightarrow$ *est* confluente *si et seulement si :*

$$\forall t, u, v\ (t \xrightarrow{*} u \text{ et } t \xrightarrow{*} v) \Rightarrow \exists w, (u \xrightarrow{*} w \text{ et } v \xrightarrow{*} w)$$

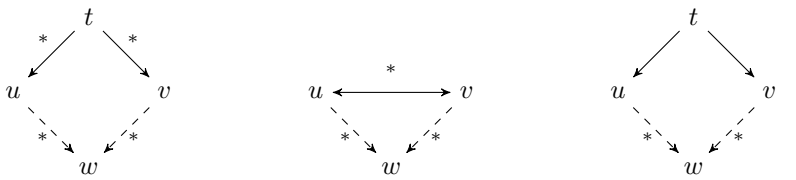
(b) $\rightarrow$ *vérifie la* propriété de Church-Rosser *si et seulement si :*

$$\forall u, v,\ u \overset{*}{\leftrightarrow} v \Rightarrow \exists w, (u \xrightarrow{*} w \text{ et } v \xrightarrow{*} w)$$

(c) $\rightarrow$ *est* localement confluente *si et seulement si :*

$$\forall t, u, v\ (t \rightarrow u \text{ et } t \rightarrow v) \Rightarrow \exists w, (u \xrightarrow{*} w \text{ et } v \xrightarrow{*} w)$$

La figure 1.4 illustre ces propriétés.



(a) Confluence (b) Church-Rosser (c) Confluence locale

FIGURE 1.4 – Propriétés sur les relations binaires.

1.1.4 Stratégies de réécriture

En réécriture, on applique habituellement de manière exhaustive toutes les règles sur le terme pour calculer sa forme normale, c'est-à-dire que l'on applique toutes les règles jusqu'à ce qu'aucune règle ne puisse plus être appliquée. Il est cependant courant d'écrire des systèmes de réécriture ne terminant pas ou n'étant pas confluents. La méthode d'application des règles adoptée prend donc de l'importance, car elle a, dans ce cas, un effet sur le résultat. Il est par

conséquent important d'avoir un contrôle sur l'application des règles du système de réécriture. C'est l'objet du concept de *stratégie* que nous allons décrire dans cette section.

Illustrons ce problème par un exemple où l'on considère le système de réécriture suivant avec la signature $\{a, f\}$, a étant d'arité 0 et f d'arité 1 :

$$\begin{cases} f(x) & \rightarrow & f(f(x)) & (r1) \\ f(a) & \rightarrow & a & (r2) \end{cases}$$

Sans aucune précision sur la manière d'appliquer les règles, nous remarquons qu'il existe une suite infinie de pas de réécriture partant de $f(a)$ si l'on applique toujours la règle $r1$:

$f(a) \xrightarrow{r1} f(f(a)) \xrightarrow{r1} f(f(f(a))) \xrightarrow{r1} \cdots$

Le calcul ne termine pas. Si l'on applique les règles de réécriture $r1$ et $r2$ différemment, nous constatons que $f(a)$ se réduit en a :

$$\begin{array}{ccccccc} f(a) & \xrightarrow{r1} & f(f(a)) & \xrightarrow{r2} & f(a) & \xrightarrow{r2} & a \\ \downarrow r2 & & & & & & \\ a & & & & & & \end{array}$$

Cet exemple illustre clairement le fait que les résultats du calcul ne sont pas les mêmes selon la méthode d'application des règles. Pour avoir des calculs qui terminent, on pourrait alors adopter une *stratégie* d'application des règles donnant la priorité à la règle $r2 : f(a) \rightarrow a$ par rapport à la règle $r1 : f(x) \rightarrow f(f(x))$.

Après cette intuition de ce qu'est une stratégie, nous pouvons donner des définitions plus formelles des concepts liés. La notion de système abstrait de réduction (*Abstract Reduction System* – ARS) [BKdV03] est une manière abstraite de modéliser les calculs par transformations pas à pas d'objets, indépendamment de la nature des objets qui sont réécrits. Un ARS peut se définir comme un couple $(\mathcal{T}, \rightarrow)$, où $\rightarrow$ est une relation binaire sur l'ensemble $\mathcal{T}$. De là, [KKK+08] donne une représentation des ARS sous forme de graphe et introduit le concept de stratégie abstraite.

Définition 19 (Système abstrait de réduction)**.** *Un système abstrait de réduction (ARS) est un graphe orienté étiqueté $(\mathcal{O}, \mathcal{S})$. Les nœuds $\mathcal{O}$ sont appelés objets, les arêtes orientées $\mathcal{S}$ sont appelées pas.*

Pour un ensemble de termes $\mathcal{T}(\mathcal{F}, \mathcal{X})$, le graphe $(\mathcal{T}(\mathcal{F}, \mathcal{X}), \mathcal{S})$ est l'ARS correspondant à un système de réécriture $\mathcal{R}$. Les arêtes correspondent à des pas de réécriture de $\mathcal{R}$ et sont étiquetées par le nom des règles de $\mathcal{R}$.

Définition 20 (Dérivation)**.** *Soit un ARS $\mathcal{A}$:*

1. *un* pas de réduction *est une arête étiquetée ϕ complétée de sa source a et de sa destination b. On note un pas de réduction $a \rightarrow^{\phi}_{\mathcal{A}} b$, ou simplement $a \rightarrow^{\phi} b$ lorsqu'il n'y a pas d'ambiguïté ;*
2. *une $\mathcal{A}$-*dérivation *(ou* séquence de $\mathcal{T}$-réductions*) est un chemin π dans le graphe $\mathcal{A}$;*
3. *lorsque cette dérivation est finie, π peut s'écrire $a_0 \rightarrow^{\phi_0} a_1 \rightarrow^{\phi_1} a_2 \cdots \rightarrow^{\phi_{n-1}} a_n$ et on dit que a_0 se réduit en a_n par la dérivation $\pi = \phi_0\phi_1 \ldots \phi_{n-1}$; notée aussi $a_0 \rightarrow^{\phi_0\phi_1\ldots\phi_{n-1}} a_n$ ou simplement $a_0 \rightarrow^{\pi} a_n$. n est la* longueur *de π ;*
   (a) *la source de π est le singleton $\{a_0\}$, notée $dom(\pi)$;*
   (b) *la destination de π est le singleton $\{a_n\}$, notée $\pi[a_0]$.*
4. *une dérivation est vide si elle n'est formée d'aucun pas de réduction. La dérivation vide de source a est notée id_a.*

Définition 21 (Stratégie abstraite)**.** *Soit un ARS $\mathcal{A}$:*

1. *une stratégie abstraite est un sous-ensemble de toutes les dérivations de $\mathcal{A}$;*
2. *appliquer la stratégie ζ sur un objet a, noté par $\zeta[a]$, est l'ensemble de tous les objets atteignables depuis a en utilisant une dérivation dans ζ : $\zeta[a] = \{\pi[a] \mid \pi \in \zeta\}$. Lorsqu'aucune dérivation dans ζ n'a pour source a, on dit que l'application sur a de la stratégie a échoué ;*

3. *appliquer la stratégie ζ sur un ensemble d'objets consiste à appliquer ζ à chaque élément a de l'ensemble. Le résultat est l'union de $\zeta[a]$ pour tous les a de l'ensemble d'objets ;*
4. *le* domaine *d'une stratégie est l'ensemble des objets qui sont la source d'une dérivation dans ζ : $dom(\zeta) = \bigcup_{\delta \in \zeta} dom(\delta)$;*
5. *la stratégie qui contient toutes les dérivations vides est $Id = \{id_a \mid a \in \mathcal{O}\}$.*

Concrètement, on peut exprimer ces stratégies de manière déclarative grâce aux langages de stratégies que proposent la plupart des langages à base de règles tels que Elan [Vit94, BKK+98, BKK+96], Stratego [VBT98, Vis01a], Maude [CELM96, CDE+02, CDE+11] et Tom, que nous allons présenter dans la section suivante.

Elan. Elan propose deux types de règles : les règles anonymes systématiquement appliquées (servant à la normalisation de termes) et les règles étiquetées pouvant être déclenchées à la demande sous contrôle d'une stratégie. Le résultat de l'application d'une telle règle sur un terme est un multi-ensemble de termes, ce qui permet de gérer le non-déterminisme. Elan a introduit la notion de stratégie en proposant un langage de combinateurs permettant de composer les stratégies et de contrôler leur application. Parmi ces combinateurs, on retiendra notamment l'opérateur de séquence, des opérateurs de choix non-déterministes et des opérateurs de répétition.

Stratego. S'inspirant d'Elan, Stratego se concentre sur un nombre restreint de combinateurs élémentaires, ainsi que sur leur combinaison. À ces combinateurs (séquence, identité, échec, test, négation, choix déterministe et non-déterministe) sont ajoutés un opérateur de récursion (μ) et des opérateurs permettant d'appliquer la stratégie : sur le $i^{ème}$ fils du sujet (*i(s)*), sur tous les sous-termes du sujet (*All(s)*), sur le premier fils du sujet sans échec (*One(s)*), sur tous les sous-termes du sujet sans échec (*Some(s)*). Grâce à ces opérateurs, il est possible d'élaborer des stratégies de haut niveau telles que *TopDown* et *BottomUp*.

Maude. L'approche de Maude est un peu différente : le contrôle sur l'application des règles s'opère grâce à la réflexivité du système [CM96, CM02]. Les objets du langage Maude ayant une représentation *meta*, il est possible d'utiliser un opérateur (`meta-apply`) pour appliquer les règles. Cet opérateur évalue les équations, normalise le terme, et retourne la *meta-représentation* du terme résultant de l'évaluation. On peut contrôler l'application des règles de réécriture par un autre programme défini par réécriture. Pour des raisons pratiques, des travaux ont été menés plus récemment [MOMV05, EMOMV07] pour offrir un langage de stratégies plus proche de ce que Elan, Stratego et Tom proposent.

Tom. À partir du concept de stratégie et en s'inspirant de ces langages, Tom implémente lui aussi un langage de stratégies [BMR08, BMR12]. Il est fondé sur des stratégies élémentaires (*Identity* et *Fail*), sur des combinateurs de composition (*Sequence*, *Recursion* —μ—, *Choice*, *Not*, *IfThenElse*) et de traversée (*All*, *One*, *Up* et *Omega*). De ces combinateurs, à l'image de Stratego, des stratégies composées peuvent être élaborées (*Try*, *Repeat*, *Innermost*, etc.). Nous reviendrons sur les stratégies de Tom dans la section suivante, en particulier sur leur utilisation concrète au sein du langage.

1.2 Le langage Tom

Tom [MRV03, BBK+07] est un langage conçu pour enrichir des langages généralistes de fonctionnalités issues de la réécriture et de la programmation fonctionnelle. Il ne s'agit pas d'un langage *stand-alone* : il est l'implémentation du concept des « îlots formels » (*formal islands*) [BKM06] qui sont ajoutés au sein de programmes écrits dans un langage hôte. Les

constructions Tom sont transformées et compilées vers le langage hôte. Parmi les fonctionnalités apportées par Tom, on compte le filtrage de motif (*pattern-matching*), les règles de réécriture, les stratégies, ainsi que les ancrages algébriques (*mappings*).

Dans la suite de cette section, nous décrirons le langage Tom en illustrant ses fonctionnalités et constructions par des exemples simples. Sauf indication contraire, les extraits de code hôte seront en Java. Pour une documentation complète et détaillée, le lecteur intéressé pourra se référer au manuel disponible en téléchargement [BBB+09] ou directement en ligne [BB+13]. Les outils sont tous accessibles *via* le site officiel du projet Tom [1].

1.2.1 Signature algébrique

Tom permet à l'utilisateur de spécifier des signatures algébriques multi-sortées *via* l'outil Gom [Rei07]. Le langage Gom permet de décrire une structure de données et d'en générer l'implémentation typée en Java. Le listing 1.1 illustre une définition de signature Gom :

```
1 module Peano
2 abstract syntax

4 Nat = zero()
5     | suc(n:Nat)
```

Listing 1.1 – Signature algébrique Gom pour les entiers de Peano.

Un module Gom est composé d'un préambule comportant un nom — `Peano` dans cet exemple — précédé du mot-clef `module` (ligne 1). Le début de la signature est annoncé par le mot-clef `abstract syntax` (ligne 2). Cet exemple étant extrêmement simple, le préambule est composé uniquement de ces deux lignes. Pour un module plus complexe qui utiliserait des types définis dans une autre signature, la clause `imports` suivie des signatures à importer peut être intercalée entre les deux clauses précédentes. Le projet Tom fournit notamment une bibliothèque de signatures pour les types primitifs (types *builtin*) tels que les *entiers* (`int`), les *caractères* (`char`), les *flottants* (`float`) et les *chaînes de caractères* (`String`). La signature en elle-même est composée d'un ensemble de sortes — `Nat` dans notre exemple — ayant des constructeurs — `zero` et `suc` ici.

Ce générateur propose un typage fort au niveau de la structure de données Java générée, ce qui garantit que les objets créés sont conformes à la signature multi-sortée. Un second aspect intéressant de cet outil est le fait qu'il offre le partage maximal [AC93], rendant les structures générées très efficaces en temps (tests d'égalité en temps constant) et en espace. Les classes générées peuvent être modifiées par l'intermédiaire de la fonctionnalité de *hooks*. Ce mécanisme permet d'ajouter des blocs de code Java aux classes générées (par exemple, intégration d'attributs invisibles au niveau algébrique, mais manipulables par la partie Java de l'application) ou d'associer une théorie équationnelle telle que A, AC, ACU à certains opérateurs. Il est même possible de spécifier cette théorie équationnelle par des règles de normalisation. Les termes construits sont ainsi toujours en forme normale.

La signature est compilée en une implémentation Java ainsi qu'un ancrage permettant l'utilisation de cette structure dans les programmes Tom. Dans un premier temps, pour présenter les constructions du langage Tom, nous ne nous préoccuperons pas de ces ancrages ni de l'implémentation concrète Java.

1.2.2 Construction *backquote* « ‘ »

La construction ‘ (*backquote*) permet de créer la structure de données représentant un terme algébrique en allouant et initialisant les objets en mémoire. Elle permet à la fois de construire un terme et de récupérer la valeur d'une variable instanciée par le filtrage de motif. Ainsi, on peut construire des termes de type `Nat` de l'exemple précédent avec des instructions

1. Voir `http://tom.loria.fr/`.

backquote. L'instruction Tom+Java « `Nat un = ‘suc(zero());` » déclare une variable `un` dont le type est `Nat`, ayant pour valeur le représentant algébrique *suc(zero())*.

Un terme *backquote* peut aussi contenir des variables du langage hôte, ainsi que des appels de fonctions. Ainsi, l'instruction `Nat deux = ‘suc(un);` permet de créer le terme `deux` à partir de `un` créé précédemment. Le compilateur Tom n'analysant pas du tout le code hôte, notons que nous supposons que la partie hôte — `un` dans l'exemple — est conforme à la signature algébrique et que le terme est donc bien formé. L'utilisation du *backquote* permet à l'utilisateur de créer un terme et de manipuler sa vue algébrique sans se soucier de son implémentation concrète dans le langage hôte.

1.2.3 Filtrage de motif

Dans la plupart des langages généralistes tels que C ou Java, on ne trouve pas les notions de type algébrique et de terme, mais uniquement celle de types de données composées (structures C et objets Java). De même, la notion de filtrage de motif (*pattern-matching*) que l'on retrouve dans les langages fonctionnels tels que Caml ou Haskell n'existe généralement pas dans la plupart des langages impératifs classiques. Le filtrage permet de tester la présence de motifs (*pattern*) dans une structure de données et d'instancier des variables en fonction du résultat de l'opération de filtrage.

Ces constructions sont apportées par Tom dans des langages généralistes. Il devient donc possible de filtrer des motifs dans Java. En outre, les constructions de filtrage de Tom étant plus expressives que dans Caml (notamment le filtrage équationnel et le filtrage non linéaire), il est possible de les employer aussi en son sein pour utiliser le filtrage équationnel.

Le lexème **%match** introduit la construction de filtrage de Tom. Celle-ci peut être vue comme une généralisation de la construction habituelle *switch-case* que l'on retrouve dans beaucoup de langages généralistes. Cependant, plutôt que de filtrer uniquement sur des entiers, des caractères, voire des chaînes de caractères, Tom filtre sur des *termes*. Les motifs permettent de discriminer et de récupérer l'information contenue dans la structure de données algébrique sur laquelle on filtre.

Dans le listing 1.2 suivant, nous reprenons l'exemple des entiers de Peano pour lesquels nous encodons l'addition avec Tom et Java. Pour ce premier exemple où les deux langages apparaissent en même temps, nous adoptons un marquage visuel pour désigner Java (noir) et Tom (gris). Cela permet de visualiser le tissage étroit entre le langage Tom et le langage hôte dans lequel il est intégré. Par la suite, le principe étant compris, nous n'adopterons plus ce code visuel.

```
1 Nat peanoPlus(Nat t1, Nat t2) {
2   %match(t1, t2) {
3     x, zero() -> { return ‘x; }
4     x, suc(y) -> { return ‘suc(peanoPlus(x,y)); }
5   }
6 }
```

Listing 1.2 – Exemple d'utilisation du filtrage avec l'addition des entiers de Peano.

Dans cet exemple, la fonction `peanoPlus` prend en arguments deux termes `t1` et `t2` de type `Nat` représentant deux entiers de Peano, et retourne la somme des deux. Le calcul est opéré par filtrage en utilisant la construction **%match** :

- elle prend un ou plusieurs arguments — deux dans notre exemple — entre parenthèses ;
- elle est composée d'un ensemble de *règles* de la forme `membre gauche -> {membre droit}` ;
- le membre gauche est composé du ou des motifs séparés les uns des autres par une virgule ;
- le membre droit est un bloc de code mixte (hôte + Tom).

Dans notre exemple, le calcul de filtrage est le suivant :

— si `zero()` filtre `t2`, alors le résultat de l'évaluation de la fonction `peanoPlus` est `x`, instancié par `t1` par filtrage ;

— si `suc(y)` filtre `t2`, alors le symbole de tête du terme `t2` est *suc*. Le sous-terme `y` est ajouté à `x` et le résultat de l'évaluation de `peanoPlus` est donc `suc(peanoPlus(x,y))`.

L'expression de la fonction `peanoPlus` est donnée par filtrage et définit une fonction Java qui peut être utilisée comme toute autre fonction Java dans le programme. Notons cette particularité qu'a le langage Tom de complètement s'intégrer au langage hôte sans pour autant être intrusif. Ainsi, le compilateur Tom n'analyse que le code Tom (parties grisées dans le listing 1.2), les instructions hôtes n'étant pas examinées et ne fournissant aucune information au compilateur. Les instructions Tom sont traduites vers le langage hôte et remplacées en lieu et place, sans modification du code hôte existant.

Une deuxième particularité des constructions de filtrage du langage Tom est liée à la composition des membres droits des règles : plutôt qu'être de simples termes, il s'agit en fait d'instructions du langage hôte qui sont exécutées lorsqu'un filtre est trouvé. Si aucune instruction du langage hôte ne rompt le flot de contrôle, toutes les règles peuvent potentiellement être exécutées. Pour interrompre ce flot lorsqu'un filtre est trouvé, il faut utiliser les instructions *ad-hoc* du langage hôte, telles que `break` ou `return`, comme dans le listing 1.2.

Filtrage associatif. Dans le cas d'un filtrage non unitaire (filtrage pour lequel il existe plusieurs solutions, voir 1.1.2), l'action est exécutée pour chaque filtre solution. Le filtrage syntaxique étant unitaire, il n'est pas possible d'exprimer ce comportement. Le langage Tom dispose de la possibilité d'opérer du *filtrage associatif avec élément neutre* (ou *filtrage de liste*, noté *AU* dans la section 1.1.2) qui n'est pas unitaire et qui permet d'exprimer aisément des algorithmes opérant du filtrage sur des listes. Il est ainsi possible de définir des opérateurs variadiques (opérateurs d'arité variable, par exemple les opérateurs de listes), comme nous l'avons vu dans la section précédente (1.1.2). Dans le listing 1.3 suivant, nous reprenons notre exemple des entiers de Peano que nous augmentons d'un nouveau constructeur, `concNat` de sorte `NatList` et dont les sous-termes sont de type `Nat`.

```
1 module Peano
2 abstract syntax

4 Nat = zero()
5     | suc(n:Nat)

7 NatList = concNat(Nat*)
```

Listing 1.3 – Signature algébrique Gom avec opérateur variadique pour les entiers de Peano.

L'opérateur `concNat` peut être vu comme un opérateur de concaténation de listes de `Nat` : `concNat()` représente la liste vide d'entiers naturels, `concNat(zero())` la liste ne contenant que `zero()` et `concNat(zero(),suc(zero()),suc(suc(zero())))` la liste contenant trois entiers naturels : 0, 1 et 2. La liste vide est l'élément neutre. Tom distingue syntaxiquement les variables de filtrage représentant un élément — par exemple `x` dans le listing 1.2 — et les variables représentant une sous-liste d'une liste existante en ajoutant le caractère `*`.

Le filtrage associatif permet d'opérer une itération sur les éléments d'une liste comme l'illustre l'exemple du listing 1.4 :

```
1 public void afficher(NatList liste) {
2   int i = 0;
3   %match(liste) {
4     concNat(X1*,x,X2*) -> {
5       i = `X1.length();
```

```
6       System.out.println("liste("+i+") = " + ‘x);
7     }
8   }
9 }
```

Listing 1.4 – Filtrage associatif.

Dans cet exemple, `liste` est une liste d'entiers naturels, de type `NatList` implémenté par le type Java `NatList`. Nous souhaitons afficher ces entiers avec leur position dans la liste. L'action est exécutée pour chaque filtre trouvé, et les variables de listes `X1*` et `X2*` sont instanciées pour chaque sous-liste préfixe et suffixe de la liste `liste`. `X1` correspond à un objet Java de type `NatList`, la méthode `length()` retournant la longueur de la liste peut donc être utilisée pour obtenir l'indice de l'élément `x`. L'énumération s'opère tant que le flot d'exécution n'est pas interrompu. Dans notre exemple, nous nous contentons d'afficher l'élément et sa position. L'exécution de la procédure `afficher` sur l'entrée `liste = ‘conc(zero(), zero(), suc(suc(zero())), zero())` donne :

```
liste(0) = zero()
liste(1) = zero()
liste(2) = suc(suc(zero()))
liste(3) = zero()
```

Le langage Tom offre aussi la possibilité de procéder à du filtrage *non-linéaire* (motifs pour lesquels une même variable apparaît plusieurs fois). Le listing 1.5 ci-après illustre ce mécanisme : dans la fonction `supprimerDoublon`, la variable `x` apparaît à plusieurs reprises dans le motif.

```
1 public NatList supprimerDoublon(NatList liste) {
2   %match(liste) {
3     concNat(X1*,x,X2*,x,X3*) -> {
4       return ‘supprimerDoublon(concNat(X1*,x,X2*,X3*));
5     }
6   }
7   return liste;
8 }
```

Listing 1.5 – Filtrage associatif non linéaire.

Parmi les notations disponibles dans le langage, certaines sont très couramment utilisées et apparaîtront dans les extraits de code de ce document. Nous les illustrons dans le listing 1.6 dont le but est d'afficher un sous-terme de la liste `liste` ayant la forme `suc(suc(y))`, ainsi que sa position dans la liste, s'il existe :

```
1 public void chercher(NatList liste) {
2   %match(liste) {
3     concNat(X1*,x@suc(suc(_)),_*) -> {
4       System.out.println(‘x + " trouvé, en position " + ‘X1.length());
5     }
6   }
7 }
```

Listing 1.6 – Notations : alias et variable anonyme.

Les notations `_` et `_*` pour les sous-listes désignent des variables dites anonymes : c'est-à-dire que leur valeur ne peut être utilisée dans le bloc action de la règle. La notation `@` permet de créer des alias : ainsi, on peut nommer des sous-termes obtenus par filtrage. L'alias permet de vérifier qu'un motif filtre — `suc(suc(_))` dans notre exemple — tout en instanciant une variable — `x` — avec la valeur de ce sous-terme. Si on applique la procédure `chercher`

à l'entrée précédente définie comme `liste = ‘conc(zero(), zero(), suc(suc(zero())), zero())`, on obtient le résultat suivant :

```
suc(suc(zero())) trouvé, en position 2
```

1.2.4 Notations implicite et explicite

Lorsque l'on écrit un *pattern* dans le membre gauche d'une règle, il est possible d'écrire l'opérateur et ses champs de deux manières. Ces derniers étant nommés, plutôt qu'écrire l'opérateur avec tous ses arguments (*notation explicite*), on peut utiliser leurs noms dans le motif pour expliciter des sous-termes particuliers et en omettre d'autres. Ces derniers sont alors ignorés lors du filtrage. Cette notation est dite *implicite* et s'utilise *via* les lexèmes [et]. Il existe également une notation à base de contraintes qui peut rendre l'écriture implicite plus naturelle dans certains cas.

Considérons un exemple simple pour illustrer ces deux notations : si on définit les personnes comme des termes construits en utilisant l'opérateur *Personne*, d'arité *3* et de type *Personne*, ayant les arguments nommés *nom* et *prenom* de type « chaîne de caractères », et *age* de type entier, les trois fonctions suivantes du listing 1.7 sont équivalentes :

```
1  %gom() {
2    module Contacts
3    abstract syntax
4    Personne = Personne(nom:String, prenom:String, age:int)
5  }
6
7  public boolean peutConduireExplicite(Personne p) {
8    %match(p) {
9      Personne(_,_,a) -> { return (‘a>18); }
10   }
11 }
12
13 public boolean peutConduireImplicite(Personne p) {
14   %match(p) {
15     Personne[age=a] -> { return (‘a>18); }
16   }
17 }
18
19 public boolean peutConduireImplicite2(Personne p) {
20   %match(p) {
21     Personne[age=a] && a>18 -> { return true; }
22   }
23   return false;
24 }
```

Listing 1.7 – Illustration des notations explicite et implicite.

Outre l'indéniable gain en lisibilité du code qu'elle apporte de par l'utilisation des noms, la notation implicite améliore la maintenabilité du logiciel développé. En effet, dans le cas d'une extension de la signature (ajout d'un champ *numTelephone*, par exemple), l'usage de la notation explicite impose d'ajouter un `_` dans toutes les règles où l'opérateur *Personne* est utilisé. Dans notre exemple, la première fonction — `peutConduireExplicite()` — devrait donc être modifiée, contrairement aux deux autres — `peutConduireImplicite()` et `peutConduireImplicite2()`. Avec cette notation, seuls les motifs manipulant explicitement les champs ayant été changés doivent être modifiés, d'où une plus grande robustesse au changement.

Dans la suite de ce manuscrit, nous utiliserons indifféremment l'une ou l'autre des notations dans les extraits de code proposés.

1.2.5 Stratégies : maîtriser l'application des règles de réécriture

On peut analyser la structure de termes et encoder des règles de transformation grâce à la construction **%match**. Le contrôle de l'application de ces règles peut alors se faire en Java en utilisant par exemple la récursivité. Cependant, traitement et parcours étant entrelacés, cette solution n'est pas robuste au changement et le code peu réutilisable. En outre, il est difficile de raisonner sur une telle transformation. Une autre solution est d'utiliser des *stratégies* [BMR08, BMR12], qui sont un moyen d'améliorer le contrôle qu'a l'utilisateur sur l'application des règles de réécriture. Le principe de programmation par stratégies permet de séparer le traitement (règles de réécriture, partie métier de l'application) du parcours (traversée de la structure de données arborescente), et de spécifier la manière d'appliquer les règles métier.

Tom fournit un puissant langage de stratégies, inspiré de Elan [BKK+98], Stratego [VBT98], et JJTraveler [Vis01b]. Ce langage permet d'élaborer des stratégies complexes en composant des combinateurs élémentaires tels que `Sequence`, `Repeat`, `TopDown` et la récursion. Il sera donc en mesure de contrôler finement l'application des règles de réécriture en fonction du parcours défini. Le listing 1.8 illustre la construction **%strategy**, ainsi que son utilisation :

```
1  %strategy transformationPeano() extends Identity() {
2    visit Nat {
3      plus(x, zero()) -> { return ‘x; }
4      plus(zero(), x) -> { return ‘x; }
5      one() -> { return ‘suc(zero()); }
6    }
7  }
8  ...
9  public static void main(String[] args) {
10   ...
11   Nat number = ‘plus(plus(suc(suc(one())), one()),
12                      plus(suc(one()), zero()) );
13   Strategy transformStrat = ‘BottomUp(transformationPeano());
14   Nat transformedNumber = transformStrat.visit(number);
15   System.out.println(number + " a été transformé en " + transformedNumber);
16   ...
17 }
```

Listing 1.8 – Exemple d'utilisation de stratégie.

Dans cet exemple, nous supposons que notre signature contient d'autres opérateurs : `plus` et `one` de type `Nat`. La stratégie `transformationPeano` réécrit les constructeurs `one` en `suc(zero())` et les constructeurs `plus(zero(),x)` en `x`. Ligne 1, `additionPeano` étend `Identity`, ce qui donne le comportement par défaut de la stratégie : si aucune règle ne s'applique, aucune transformation n'a lieu, par opposition à `Fail` qui spécifie que la transformation échoue si la règle ne peut être appliquée. La construction **visit** (ligne 2) opère un premier filtre sur les sortes de type `Nat` sur lesquelles les règles doivent s'appliquer. Les règles sont quant à elles construites sur le même modèle que les règles de la construction **%match**, décrite dans la section 1.2.3. Le membre gauche est un *pattern*, tandis que le membre droit est un bloc action composé de code Tom+Java. Ensuite, la stratégie est composée avec une stratégie de parcours fournie par une bibliothèque appelée `sl`. Le terme la représentant est créé, et sa méthode `visit` est appliquée sur le nombre que nous souhaitons transformer. Pour l'entrée donnée, le résultat affiché est le suivant (les couleurs ont été ajoutées pour faire apparaître clairement les sous-termes transformés) :

```
plus(plus(suc(suc(one())), one()), plus(suc(one()),zero()))
  a été transformé en
plus(plus(suc(suc(suc(zero()))), suc(zero())), suc(suc(zero())))
```

Pour des informations plus détaillées sur les stratégies Tom ainsi que sur leur implémentation dans le langage, nous conseillons la lecture de [Bal09, BMR08, BMR12] ainsi que la page dédiée du site officiel du projet Tom [2].

1.3 Ancrages formels

Précédemment, nous avons vu que nous pouvons ajouter des constructions de filtrage au sein de langages hôtes et que nous sommes en mesure de filtrer des termes.

Les constructions de filtrage sont compilées indépendamment des implémentations concrètes des termes. Ainsi, dans les exemples, nous filtrons sur des termes de type `Nat` et `NatList`, définis dans la signature Gom. Cependant, les fonctions Java prennent des paramètres d'un type donné que le compilateur Java est censé comprendre, et nous n'avons pas détaillé la manière dont le lien entre les types Tom et les types Java est assuré. À la compilation, les instructions sur les termes algébriques sont traduites en instructions sur le type de données concret représentant ces termes algébriques. Le mécanisme permettant d'établir cette relation entre les structures de données algébriques et concrètes s'appelle le mécanisme d'*ancrage*, ou *mapping*. Il permet de donner une vue algébrique d'une structure de données quelconque afin de pouvoir la manipuler à l'aide des constructions Tom, sans modifier l'implémentation concrète.

On spécifie les *mappings* à l'aide des constructions Tom **%typeterm**, **%op** et **%oplist** pour décrire respectivement les sortes, les opérateurs algébriques et les opérateurs variadiques. Dans la section 1.2.1, nous avons défini une signature algébrique, et l'outil Gom génère une implémentation concrète ainsi que les ancrages. Cela nous permet de rendre ce mécanisme complètement transparent pour l'utilisateur. Pour expliquer les constructions constituant les *mappings*, nous allons les expliciter sans passer par la génération de Gom. Pour la suite de cette section, nous prenons l'exemple du type `Individu` et de l'opérateur `personne`, qui prend trois arguments : un nom et un prénom, de type `String`, et un âge de type `int`. Nous adoptons aussi une notation pour lever toute ambiguïté sur les types : nous préfixons les types Java de la lettre `J`, et nous utiliserons l'implémentation qui suit. Nous considérons les classes Java `JIndividu` et `JIndividuList` suivantes qui implémentent respectivement les types algébriques `Individu` et `IndividuList`.

```
1  static class JIndividu { }
2  static class JIndividuList { }
```

Nous considérons ensuite les classes Java `Jpersonne` et `JconcJIndividu` suivantes qui implémentent les opérateurs algébriques :

```
1  static class Jpersonne extends JIndividu {
2    public String nom;
3    public String prenom;
4    public int age;
5    public Jpersonne(String n, String p, int a) {
6      this.nom = n;
7      this.prenom = p;
8      this.age = a;
9    }
10 static class JconcJIndividu extends JIndividuList {
```

2. Voir `http://tom.loria.fr/wiki/index.php5/Documentation:Strategies/`.

```
11    private LinkedList<JIndividu> list;
12    public JconcJIndividu() { list = new LinkedList<JIndividu>(); }
13    public JconcJIndividu(JIndividu i, LinkedList<JIndividu> l) {
14      if (l!=null) {
15        this.list = l;
16      } else {
17        JconcJIndividu();
18      }
19      list.addFirst(i);
20    }
21    public boolean isEmpty() {
22      return list.isEmpty();
23    }
24    public List<JIndividu> getTail() {
25      LinkedList<JIndividu> tail = null;
26      if (list.size()>1) {
27        tail = list.subList(1,list.size()-1);
28      }
29      return tail;
30    }
31  }
```

Les sortes `Individu` et `IndividuList` sont exprimées *via* le lexème **%typeterm** comme le montre le listing 1.9 suivant :

```
1 %typeterm Individu {
2   implement { JIndividu }
3   is_sort(s) { (s instanceof JIndividu) }
4 }
```

Listing 1.9 – Ancrage du type Individu avec l'implémentation Java JIndividu.

La construction **%typeterm** permet d'établir la relation entre le type algébrique `Individu` et le type concret Java `JIndividu`. Deux sous-constructions la composent :

— **implement** donne l'implémentation effective du langage hôte à lier au type algébrique ;

— **is_sort** est utilisée en interne par le compilateur pour tester le type d'un élément (optionnel) ;

La sorte `Individu` étant définie, il est nécessaire de spécifier à Tom comment *construire* et comment *détruire* (décomposer) les termes de ce type. Pour cela, nous utilisons la construction **%op**, comme illustré par le listing 1.10 où l'opérateur `personne` est défini :

```
1   %op Individu personne(nom:String, prenom:String, age:int) {
2     is_fsym(s) { (s instanceof Jpersonne) }
3     get_slot(nom,s) { ((Jpersonne)s).nom }
4     get_slot(prenom,s) { ((Jpersonne)s).prenom }
5     get_slot(age,s) { ((Jpersonne)s).age }
6     get_default(nom) { ''Simpson'' }
7     get_default(prenom) { ''Pierre-Gilles'' }
8     get_default(age) { 42 }
9     make(t0,t1,t2) { new Jpersonne(t0,t1,t2) }
10  }
```

Listing 1.10 – Constructeur `personne`.

La définition d'opérateurs passe par une construction composée de quatre sous-constructions :

— **is_fsym** spécifie la manière de tester si un objet donné représente bien un terme dont le symbole de tête est l'opérateur ;

— **make** spécifie la manière de créer un terme ;

— **get_slot** (optionnel) spécifie la manière de récupérer la valeur d'un champ du terme ;

— **get_default** (optionnel) donne la valeur par défaut de l'attribut.

Notons que Tom ne permet pas la surcharge d'opérateurs, c'est-à-dire qu'il n'est pas possible de définir plusieurs opérateurs ayant le même nom, mais des champs différents.

La construction **%oplist** est le pendant de **%op** pour les opérateurs variadiques. Ainsi, dans le listing 1.11, nous pouvons définir un opérateur de liste d'individus `concIndividu`, de type `IndividuList`, et acceptant un nombre variable de paramètres de type `Individu`.

```
%oplist IndividuList concIndividu(Individu*) {
  is_fsym(s)       { (s instanceof JconcJIndividu) }
  get_head(l)      { ((JconcJIndividu)l).list.getFirst() }
  get_tail(l)      { ((JconcJIndividu)l).getTail() }
  is_empty(l)      { ((JconcJIndividu)l).isEmpty() }
  make_empty()     { new JconcJIndividu() }
  make_insert(t,l) { new JconcJIndividu(t,l) }
}
```

Listing 1.11 – Opérateur de liste d'Individus.

Les sous-constructions suivantes la composent :

— **is_fsym** spécifie la manière de tester si un objet donné représente bien un terme dont le symbole de tête est l'opérateur ;

— **make_empty** spécifie la manière de créer un terme vide ;

— **make_insert** spécifie la manière d'ajouter un fils en première position à la liste ;

— **get_head** spécifie la manière de récupérer le premier élément de la liste ;

— **get_tail** spécifie la manière de récupérer la queue de la liste (sous-liste constituée de la liste privée du premier élément) ;

— **is_empty** spécifie la manière de tester si une liste est vide.

Notons que le langage Tom supporte aussi le sous-typage [KMT09, Tav12]. Il est donc possible de spécifier un type comme sous-type d'un autre type déjà exprimé. Cette relation de sous-typage exprimée dans les types Tom doit évidemment être cohérente avec celle exprimée dans l'implémentation concrète en Java. Reprenons l'exemple précédent des *individus* et considérons différents sens possibles : *être vivant* (sens couramment utilisé, d'où le constructeur `personne`), *spécimen vivant d'origine animale ou végétale* (en biologie) et *élément constituant un ensemble* (en statistiques). Le type *Individu* pouvant être ambigu, nous décidons de le préciser en intégrant des sous-types. Pour cela, il existe une construction Tom permettant de spécifier un sous-type dans un ancrage algébrique : **extends**, qui complète la construction **%typeterm**.

```
1 %typeterm IndividuBiologie extends Individu {
2   implement { JIndividuBiologie }
3   is_sort(s) { (s instanceof JIndividuBiologie) }
4   equals(t1,t2) { (t1.equals(t2)) }
5 }
6 public class JIndividuBiologie extends Individu {
7   ...
8 }
```

Listing 1.12 – Exemple de déclaration de type avec sous-typage.

Dans le listing 1.12, nous déclarons donc un nouveau type, `IndividuBiologie`, sous-type de `Individu` (lignes 1 à 5). Il est implémenté par le type Java `JIndividuBiologie` dont la relation de sous-typage avec `JIndividu` est exprimée par la construction Java consacrée à l'héritage : **extends** (ligne 7). Il est ensuite possible de définir de manière classique des opérateurs de type `IndividuBiologie` comme l'illustre le listing 1.13.

```
%op IndividuBiologie animal(espece:String) {
  ...
}
%op IndividuBiologie vegetal(espece:String) {
  ...
}
```

Listing 1.13 – Constructeurs de type `IndividuBiologie`.

1.4 Domaines d'applications

Tom est particulièrement utile pour parcourir et transformer les structures arborescentes comme XML par exemple. Il est donc tout à fait indiqué pour écrire des compilateurs ou interpréteurs, et plus généralement des outils de transformation de programmes ou de requêtes. Outre son utilisation pour écrire le compilateur Tom lui-même, nous noterons son usage dans les cadres académiques et industriels suivants :

— implémentation d'un mécanisme défini par Guillaume Burel dans sa thèse [Bur09] pour une méthode des tableaux particulière appelée TaMeD [Bon04] ;

— prototypage de langages tels que MiniML dont la compilation a été décrite par Paul Brauner dans sa thèse [Bra10] ;

— implémentation d'un assistant à la preuve tel que Lemuridae développé par Paul Brauner pour un calcul des séquents nommé LKMS [Bra10] ;

— implémentation de compilateurs ou interpréteurs, comme le compilateur +CAL 2.0 [3] qui traduit des algorithmes exprimés en +CAL en TLA+ pour faire du *model-checking* ;

— raisonnement formel sur la plateforme Rodin [4]. Il s'agit d'un outil industriel *open source* développé par Systerel [5] qui permet de formaliser, d'analyser et de prouver les spécifications de systèmes complexes ;

— outil de transformation de requêtes OLAP (*OnLine Analytical Processing*) en requêtes SQL (*Structured Query Language*) pour assurer la rétro-compatibilité entre les versions du logiciel Crystal Reports [6] développé par Business Object [7] ;

— transformations de modèles qualifiables pour les systèmes embarqués temps-réel dans le projet quarteFt [8] porté par le LAAS-CNRS [9], l'IRIT [10], l'ONERA-DTIM [11] et Inria Nancy ainsi que nos partenaires industriels Airbus [12] et Ellidiss Software [13]. C'est dans ce cadre que s'inscrit cette thèse.

1.5 Apport de la thèse au projet Tom

Durant cette thèse, j'ai contribué techniquement au projet Tom. La figure 1.5 illustre le fonctionnement global du système Tom au début de ma thèse, et sert de point de comparaison avec la figure 1.6 pour illustrer cet apport de développement.

La contribution technique de cette thèse au projet Tom a consisté en l'extension du langage Tom par l'ajout de constructions haut-niveau dédiées à la transformation de modèles. Ces constructions haut niveau (**%transformation**, **%resolve** et **%tracelink**) sont détaillées dans le chapitre 6. La construction **%transformation** implémente une méthode proposée dans [BCMP12] pour transformer des modèles EMF Ecore, tout en simplifiant l'écriture de la transformation pour l'utilisateur. Cette méthode se déroule en deux phases :

3. Voir `https://gforge.inria.fr/projects/pcal2-0/`.
4. Voir `http://www.event-b.org/`.
5. Voir `http://www.systerel.fr/`.
6. Voir `http://crystalreports.com/`.
7. Voir `http://www.businessobjects.com/`.
8. Site du projet : `http://quarteft.loria.fr/`.
9. Laboratoire d'Analyse et d'Architecture des Systèmes : `http://www.laas.fr/`.
10. Institut de Recherche en Informatique de Toulouse : `http://www.irit.fr/`.
11. Office national d'études et de recherches aérospatiales : `http://www.onera.fr/fr/dtim/`.
12. Voir `http://www.airbus.com/`.
13. Voir `http://www.ellidiss.com/`.

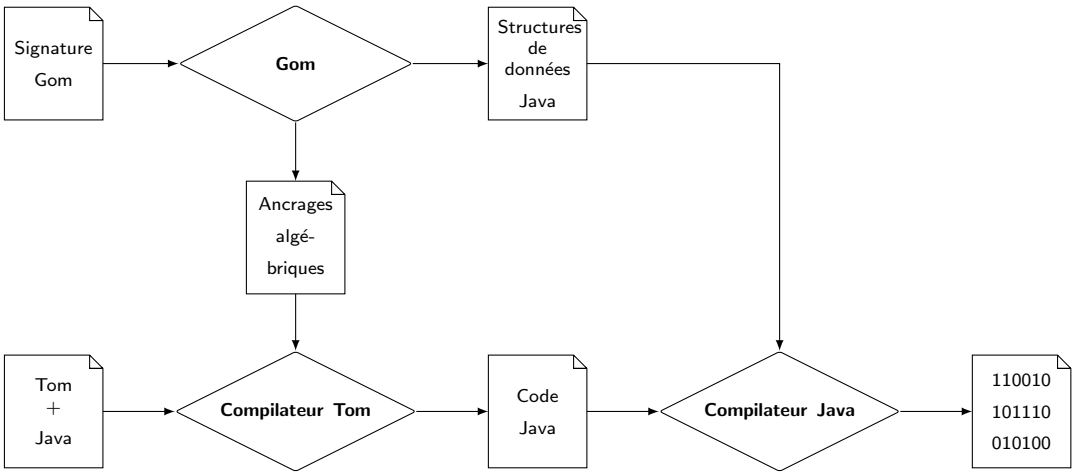


FIGURE 1.5 – Fonctionnement global du projet Tom en début de thèse.

— une phase de *décomposition* de la transformation en transformations élémentaires encodées par des stratégies Tom ;

— une phase de *réconciliation* durant laquelle le résultat intermédiaire obtenu lors de la première phase est rendu cohérent et conforme au métamodèle cible.

Les constructions **%resolve** et **%tracelink** servent à la phase de *réconciliation* d'une transformation de modèle suivant la méthode proposée. En suivant cette approche, l'utilisateur peut écrire les transformations élémentaires de manière indépendante et sans aucune notion d'ordre d'application des pas d'exécution de la transformation.

La construction **%tracelink** présente aussi un intérêt en termes de traçabilité de la transformation. Nous souhaitons générer des traces d'exécution de la transformation à la demande de l'utilisateur, ce pour quoi **%tracelink** a été conçue.

Cela a eu pour conséquence l'ajout d'un nouveau greffon (voir la figure 6.5) dans la chaîne de compilation (*transformer*), ainsi que l'adaptation de cette chaîne, du *parser* au *backend*, afin de prendre en compte cette extension du langage.
Outre l'évolution du langage Tom lui-même, le générateur d'ancrages algébriques Tom-EMF a aussi évolué et est maintenant disponible en version stable.

Nos contributions au projet Tom apparaissent en pointillés sur la figure 1.6.

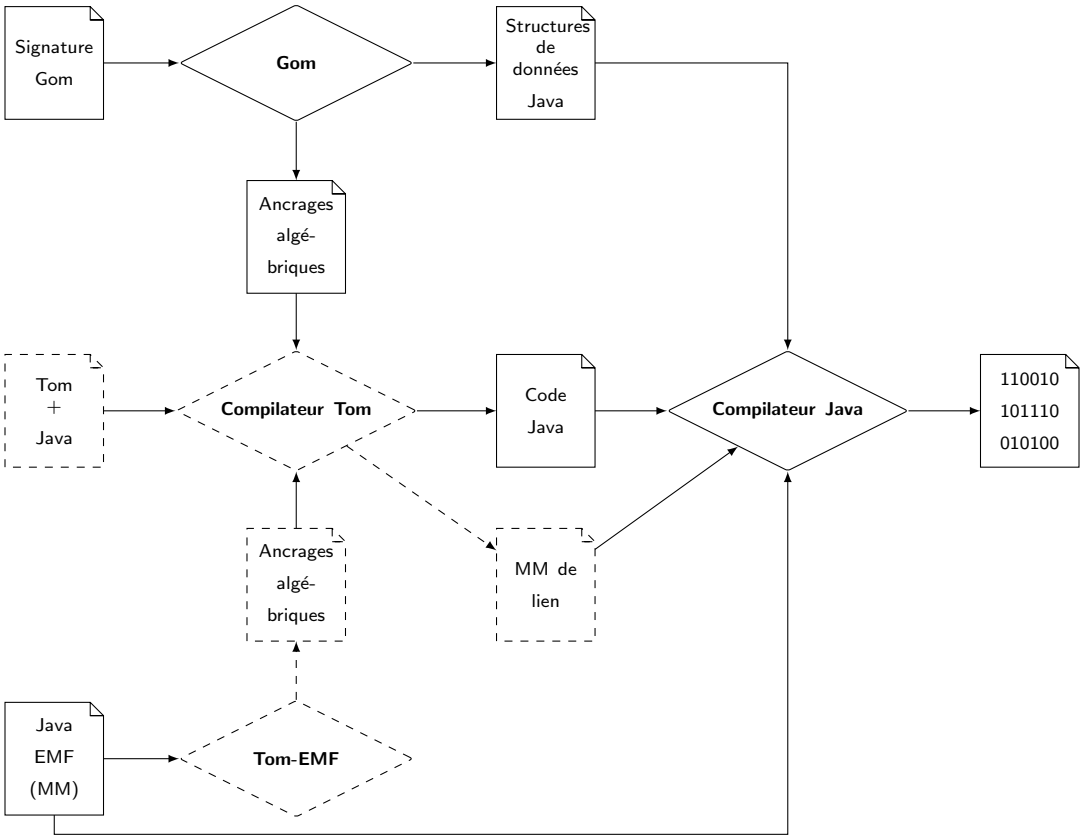


FIGURE 1.6 – Fonctionnement global de Tom et contributions de ce travail de thèse au projet.

Chapitre 2

Transformations de modèles

Dans ce chapitre, nous présentons un domaine du développement logiciel en plein essor : l'Ingénierie Dirigée par les Modèles (IDM). Nous présentons les approches actuelles et les notions importantes de ce domaine indispensables à la bonne compréhension de ce document. Dans un premier temps, nous expliquons ce qu'est l'IDM et notamment l'approche MDA (*Model Driven Architecture*). Ensuite, nous abordons la problématique des transformations de modèles, et enfin nous présentons brièvement certains outils très utilisés.

2.1 Modélisation

À partir des années 80, l'ingénierie du logiciel a été marquée par l'approche objet dans laquelle le principe « Tout est objet » prévaut. Ce principe permet d'amener les outils et les technologies à la simplicité, la généralité et la bonne intégration dans les systèmes. Cette approche repose sur la notion d'*objet*, qui *est une instance* d'une *classe*, qui peut *hériter* d'une autre classe (*surclasse*).

L'arrivée de l'IDM a renouvelé les notions phares guidant le développement logiciel. Le principe « Tout est objet » est remplacé par celui du « Tout est modèle » [BB04] [Bé05]. Les notions de *modèle représentant* un système, et *conforme à* un *métamodèle* sont apparues. L'approche IDM vise à élever le niveau d'abstraction dans la spécification logicielle, et à augmenter l'automatisation dans le développement. L'idée promue par l'IDM est d'utiliser les modèles à différents niveaux d'abstraction pour développer les systèmes, et ainsi élever le niveau d'abstraction dans la spécification du programme. L'augmentation de l'automatisation dans le développement logiciel est obtenue en utilisant des transformations de modèles exécutables. Les modèles de plus haut niveau sont transformés en modèles de plus bas niveau jusqu'à ce que le modèle puisse être exécuté en utilisant soit la génération de code, soit l'interprétation du modèle (exécution du modèle).

La définition même de modèle a longtemps été discutée, et de nombreuses propositions — dont notamment celle de [BG01], ainsi que celle de [KWB03] — nous permettent de donner la définition suivante :

Définition 22 (Modèle)**.** *Un modèle est la représentation ou l'abstraction d'un (ou d'une partie d'un) système, écrit dans un langage bien défini. Un modèle doit pouvoir être utilisé à la place du système modélisé pour répondre à une question sur le système représenté.*

Il en découle la définition de la relation *représente* entre le modèle et le système modélisé [AK03, Sei03, Bé05]. Du fait de l'utilisation du modèle à la place du système, cette représentation doit être pertinente. La définition précédente amène aussi à préciser la notion de langage bien défini :

Définition 23 (Langage bien défini)**.** *Un langage bien défini est un langage avec une forme*

bien définie (syntaxe), et une signification (sémantique), qui est appropriée pour une interprétation automatique par un ordinateur[14].

Pour pouvoir utiliser un modèle, son langage de modélisation doit être précisé. Il est représenté par un métamodèle.

Définition 24 (Métamodèle)**.** *Un métamodèle est un modèle qui définit le langage d'expression d'un modèle [OMG06a], c'est-à-dire le langage de modélisation.*
Dans la suite de ce document, nous noterons les métamodèles MM, indicés en fonction de leur nature : ainsi un métamodèle source — respectivement cible — sera noté MM_s — respectivement MM_t.

La relation qui lie un modèle à son métamodèle est la relation de *conformité*, c'est-à-dire qu'un métamodèle est la spécification d'un modèle : on dit qu'un modèle *est conforme à* son métamodèle.

La publication en 2000 de l'initiative MDA [StOS00] par l'OMG [15] a considérablement amplifié l'intérêt de l'IDM. Le MDA est la vision qu'a l'OMG de l'IDM : il s'agissait de définir un cadre pour utiliser concrètement les modèles dans le développement logiciel [KWB03, MKUW04]. Pour cela, elle propose différents standards pour résoudre des problèmes soulevés par l'ingénierie dirigée par les modèles :

— établissement de nouveaux langages de modélisation [OMG06a] ;
— modélisation des systèmes [OMG09, OMG11] ;
— expression de la sémantique statique des modèles et des métamodèles [OMG06b] ;
— représentation des modèles sous forme de documents XML (pour l'interopérabilité)[OMG13] ;
— représentation des parties graphiques d'un modèle au format XML [OMG03].

L'OMG a donc proposé le standard MOF [OMG06a] [16] comme un sous-ensemble du diagramme de classes UML. Intuitivement, un métamodèle est composé d'un ensemble de métaclasses qui contiennent des *attributs* et des *opérations* comme les classes en programmation orientée objet. Les métaclasses peuvent être liées par héritage, et par une métarelation (une association ou une composition). Chaque modèle doit se conformer à un tel métamodèle, c'est-à-dire être un ensemble d'éléments, d'attributs et de relations entre les éléments se conformant à leurs métadéfinitions.

Ce standard MOF — composé de EMOF [17] et CMOF [18] depuis la version 2.0 — permet d'établir de nouveaux langages de modélisation et se situe au sommet d'une architecture en quatre couches proposée par l'OMG (figure 2.1).

La couche M0 est celle du monde réel qui est représenté par des modèles de la couche M1. Ces modèles sont conformes à leurs métamodèles — couche M2 — qui sont eux-mêmes décrits par le métamétamodèle MOF — couche M3. Ce dernier est unique et métacirculaire, c'est-à-dire qu'il se décrit lui-même.

Définition 25 (Métamétamodèle)**.** *Un métamétamodèle est un métamodèle qui décrit un langage de métamodélisation. Il permet donc de définir des langages de modélisation et est métacirculaire, c'est-à-dire qu'il se décrit lui-même.*

Cette approche hiérarchique est classique et on la retrouve dans plusieurs autres domaines de l'informatique. Une telle hiérarchie définit ce que l'on appelle un *espace technique* (ou *espace technologique*) [KBA02, Bé06]. Ainsi, nous parlerons de *modelware*, de *grammarware* [KLV05, KKK+] pour l'espace des grammaires définies par l'EBNF [19], de *dataware* (ou parfois *DBware*)

14. Traduction de [KWB03], page 16 : *A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer.*
15. Object Management Group : `http://www.omg.org/`.
16. Meta Object Facility : `http://www.omg.org/mof/`.
17. Essential MOF
18. Complete MOF
19. Extended Backus-Naur Form

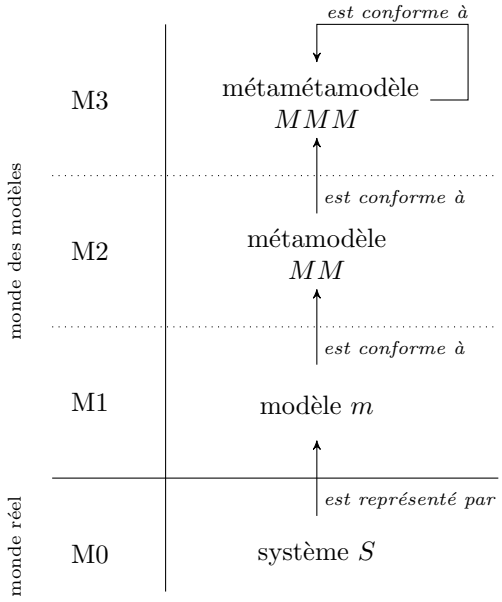


FIGURE 2.1 – Organisation du MDA en 4 niveaux d'abstraction (3+1).

avec SQL et de *documentware* avec XML. Ce que nous appelons métamodèle dans l'espace des modèles correspond à la grammaire dans l'espace *grammarware*, au type dans les langages de programmation, au schéma dans les espaces *documentware* et *dataware*. Pour illustrer notre propos, dans la figure 2.2, nous interprétons les niveaux d'abstraction de cette organisation en couches du MDA et faisons une analogie avec l'espace technique des grammaires. Ainsi, MOF correspond à l'EBNF, tandis que la couche M2 contient les langages définis par la grammaire, la couche M1 les programmes écrits dans ces langages, et M0 consiste en les exécutions de ces programmes.

Outre MOF, l'OMG propose d'utiliser d'autres standards dans son approche MDA, notamment UML (*Unified Modeling Language*) et (OCL *Object Constraint Language*) :

— UML est le standard de modélisation des systèmes logiciels. Il est composé depuis 2009 [OMG09] du paquetage *UML 2.2 Superstructure* qui est le standard pour la modélisation orientée objet, ainsi que du paquetage *UML 2.2 Infrastructure* décrivant le noyau commun entre MOF et UML.

— OCL a été proposé pour exprimer la sémantique axiomatique des modèles et des métamodèles. Il permet d'exprimer des contraintes sur les métamodèles (invariant, pre- et post-conditions sur les opérations), afin de mieux les définir pour ensuite pouvoir les outiller.

L'initiative MDA se concentre sur la variabilité technique d'un logiciel, c'est-à-dire sur la manière de spécifier un logiciel indépendamment de la plateforme, afin de mettre en œuvre des systèmes durables et maintenables. Elle propose d'élaborer différents modèles pour définir une architecture de spécification à plusieurs niveaux. Pour cela, les notions de CIM (*Computation Independent Model*), PIM (*Platform Independent Model*) et PSM (*Platform Specific Model*) ont été introduites :

— **CIM :** modèle d'exigence, défini hors de toute considération informatique (par exemple, le diagramme de cas d'utilisation UML) ;

— **PIM :** ce modèle défini indépendamment de la plateforme d'exécution est aussi appelé modèle d'analyse et conception ;

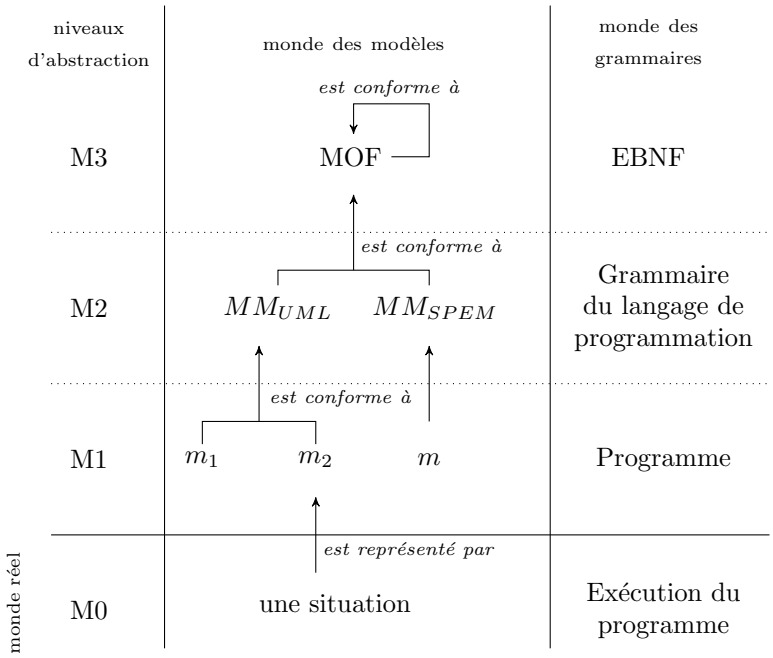


FIGURE 2.2 – Interprétation des niveaux d'abstraction du MDA.

— **PSM :** modèle de code, pour lequel les caractéristiques de la plateforme d'exécution ont été prises en compte.

Le passage de PIM à PSM peut se faire par des transformations de modèles, intégrant parfois des transformations intermédiaires de PIM à PIM (pour des raisons d'interopérabilité), d'où leur place de choix dans les approches modèles pour le développement logiciel.

Le MDA fournit un socle pour aider l'IDM à se développer. Cependant, il s'agit d'une vision parmi d'autres [FEBF06]. On pense notamment aux *usines logicielles* (*Software Factories*) [GS03] dont l'idée est de se calquer sur le processus de production du monde matériel pour mettre en œuvre celui du monde logiciel. Il s'agit de composer des applications à partir d'un grand nombre d'outils, de *patterns*, de modèles et de *frameworks*. Les idées principales de cette approche sont :

— la spécialisation des fournisseurs et développeurs de logiciels ;

— l'utilisation d'outils dédiés (transformations de modèles, langages et outils de modélisation) ;

— l'intégration systématique de code réutilisable.

Microsoft a intégré ces concepts au sein de son *framework* .NET et de son environnement de développement.

De son côté, IBM a activement contribué à l'approche MDA en étant le principal développeur de UML et en participant à l'élaboration des standards MOF, XMI et CWM. IBM donne sa vision du MDA dans [BBI+04] où ces trois principes sont développés :

— **représentation directe :** il faut se détacher du domaine technologique et se concentrer sur les idées et concepts du domaine métier. Réduire la distance sémantique entre le domaine métier et la représentation permet de répondre plus directement et précisément aux problèmes. Pour prendre en compte ces situations métier, on utilise des langages dédiés (*Domain Specific Language*, DSL) ;

— **automatisation :** il faut automatiser tout ce qui ne nécessite pas d'intelligence et de réflexion ;

— **standards ouverts :** les standards encouragent la création de communautés autour d'outils, et un développement ouvert permet d'assurer que ces standards soient implémentés de manière cohérente tout en poussant à leur adoption. Les communautés autour des plateformes de logiciel libres ont un rôle structurant.

IBM a donc proposé son langage de métamodélisation — Ecore — pour l'*Eclipse Modeling Framework* (EMF) [SBPM09]. Il s'aligne quasiment sur EMOF, et cette proximité des deux métamodèles permet à EMF de supporter directement EMOF comme une sérialisation XMI alternative de Ecore. Le passage de l'un à l'autre consiste essentiellement en des tâches de renommage de classes et de fonctionnalités. Ecore est intégré dans l'environnement Eclipse et est l'un des *frameworks* les plus utilisés pour la manipulation de modèles, ce qui en fait un standard *de facto*. Il fait le lien entre la programmation Java et la modélisation. C'est pourquoi nous avons choisi EMF comme première technologie pour les outils développés durant cette thèse.

Pour une description approfondie du MDA, le lecteur intéressé pourra se référer à [KWB03], [Bro04] et [BS05].

2.2 Transformations de modèles

Les transformations de modèles sont au cœur de l'IDM [SK03]. Elles permettent d'automatiser les tâches répétitives — application d'un traitement à un grand nombre de modèles ou multiples applications d'un traitement sur un modèle — ainsi que des tâches complexes et sujettes à erreurs si elles étaient effectuées manuellement par un utilisateur. Les transformations de modèles sont aussi utiles dans le cas où le développement d'un système est segmenté et que les rôles dans le développement sont séparés : si le spécialiste de la plateforme n'est pas le spécialiste du système développé, le traitement par l'un d'un modèle produit par l'autre est particulièrement complexe. Les transformations de modèles permettent un gain accru en termes de productivité, de maîtrise de la complexité, de diminution des risques et donc, finalement, de coût. Elles sont utilisées pour réaliser des tâches variées telles que :

— la réingénierie (*refactoring*) : tâche qui consiste à changer un logiciel de telle sorte que son comportement extérieur n'est pas affecté tout en améliorant sa structure interne [20] ;

— la génération de code : processus qui produit du code à partir d'instructions abstraites en prenant en compte les caractéristiques de la plateforme cible d'exécution ;

— le raffinement : transformation d'une spécification abstraite en une spécification (plus) concrète ;

— la normalisation : mise en conformité d'un logiciel, adaptation du style de code, *etc.* ;

— la migration : passage d'une technologie (ou d'une version à une autre d'une technologie) à l'autre ;

— la sérialisation : processus de mise en série des données sous la forme d'un fichier ;

— la rétro-conception ou rétro-ingénierie (*reverse engineering*) : étude d'un programme pour en comprendre le fonctionnement interne et sa méthode de conception.

Définition 26 (Transformation de modèle)**.** *Une transformation de modèle T est une relation d'un (ou plusieurs) métamodèle(s) source(s) MM_s vers un (ou plusieurs) métamodèle(s) cible(s) MM_t. On la notera : $T : MM_s \rightarrow MM_t$ (ou $T : MM_{s_0} \times \ldots \times MM_{s_n} \rightarrow MM_{t_0} \times \ldots \times MM_{t_m}$ dans le cas d'une transformation de n métamodèles sources vers m métamodèles cibles).*

Dans la suite du document, nous considérerons essentiellement des transformations de modèles d'un seul métamodèle source vers un seul métamodèle cible ($T : MM_s \rightarrow MM_t$).

20. Traduction de la définition de [Fow99]

Compte tenu de la diversité des transformations, tant par leurs approches de développement, que par les structures de données dont elles se servent pour représenter les modèles qu'elles manipulent, que par les caractéristiques des métamodèles sources et cibles, de nombreux travaux ont été menés pour tenter de les classifier [CH03], [SK03], [CH06], [MG06].

2.2.1 Taxonomie des transformations

On distingue différents types de transformations de modèles que l'on peut définir et classifier selon que les métamodèles source et cible sont identiques ou non, et selon qu'ils appartiennent ou non au même niveau d'abstraction. C'est à partir de ces critères que [MG06] a proposé une taxonomie des transformations de modèles sur laquelle nous nous appuyons dans cette section.

Définition 27 (Transformation endogène)**.** *Une transformation $T : MM_s \rightarrow MM_t$ est dite endogène si $MM_s = MM_t$, c'est-à-dire si les métamodèles source et cible sont identiques.*

Par exemple, une activité de réingénierie d'un logiciel est une transformation endogène. Par analogie avec les langues naturelles, la reformulation d'une phrase en est aussi une. Une transformation endogène s'opère dans un même espace technologique, si les modifications sont directement appliquées sur le modèle source, on parle de transformation *in-place*. Dans le cas contraire, il s'agit d'une transformation *out-place*.

Définition 28 (Transformation exogène)**.** *Une transformation $T : MM_s \rightarrow MM_t$ est dite exogène si $MM_s \neq MM_t$, c'est-à-dire si les métamodèles cible et source sont différents..*

Par exemple, la migration d'un système ou la réécriture d'un logiciel dans un autre langage est une transformation exogène. En utilisant l'analogie précédente, la traduction en est aussi une. Une transformation exogène est toujours *out-place* et peut permettre un changement d'espace technique.

Définition 29 (Transformation horizontale)**.** *Une transformation est dite horizontale si les modèles source et cible appartiennent au même niveau d'abstraction.*

Par exemple, l'activité de réingénierie est une transformation horizontale (endogène), tout comme celle de migration (exogène).

Définition 30 (Transformation verticale)**.** *Une transformation est dite verticale si les modèles source et cible n'appartiennent pas au même niveau d'abstraction.*

Par exemple, la rétro-conception et le raffinement sont des transformations verticales, respectivement exogène et endogène.

La figure 2.3 résume sous forme matricielle cette classification en fonction des deux critères :

— les colonnes classifient les transformations selon le changement de niveau d'abstraction (transformation verticale *vs* horizontale) ;

— les lignes classifient les transformations selon que le métamodèle cible est le même (ou non) que le métamodèle source (transformation endogène *vs* exogène).

	Transformation horizontale	Transformation verticale
MM_s conforme à m_s → m_t conforme à MM_t — Transformation exogène	Migration, Sérialisation	Génération de code, Retro-conception
MM conforme à m_s → m_t — Transformation endogène	Optimisation, Réingénierie, Normalisation, Simplification	Raffinement

FIGURE 2.3 – Classification des catégories de transformations de modèles.

2.2.2 Approches de développement des transformations de modèles

Les transformations de modèles peuvent être classifiées selon leurs approches de développement et selon les structures de données représentant les modèles.

Trois approches de transformation de modèles sont définies dans [BS05] :

— par programmation ;

— par *template* ;

— par modélisation.

Dans l'approche par programmation, une transformation est développée en utilisant des langages de programmation « classiques » (généralistes) généralement orientés objet, tels que Java. La transformation se présente sous la forme d'un programme et le développeur a accès à tout l'écosystème existant (environnement de développement, bibliothèques, documentation, communauté d'utilisateurs...). Cela en fait une approche très répandue.

Dans l'approche par *template*, un *modèle cible paramétré* (ou *modèle template*) est défini. La transformation consiste alors à procéder au remplacement des paramètres en utilisant les valeurs fournies par le modèle source. C'est ce qu'implémente le logiciel *Softeam MDA Modeler*.

La troisième approche identifiée — l'approche par modélisation — consiste à considérer la transformation elle-même comme un modèle conforme à son métamodèle de transformation. L'OMG a défini le standard *Query/View/Transformation* (QVT) pour modéliser les transformations de modèles [OMG08]. Il est composé de trois langages conformes à MOF — QVT-Operational, QVT-Relations et QVT-Core — regroupant deux paradigmes de description des transformations de modèles :

Approche opérationnelle : une transformation de modèle est exprimée comme une séquence de pas d'exécution élémentaires qui construisent le modèle cible (instanciation des nouveaux éléments, initialisation des attributs, création des liens, *etc.*) en utilisant les informations du modèle source. Cette approche, appelée *opérationnelle*, est

impérative. Elle peut être implémentée en utilisant des outils dédiés tels que Kermeta, QVT-Operational, ou en utilisant des bibliothèques au sein d'un langage généraliste.

Approche relationnelle : une transformation de modèle peut aussi être définie comme des relations devant exister entre la source et la cible à la fin de la transformation. Cette approche, appelée *relationnelle* (ou déclarative), n'est pas directement exécutable, mais peut parfois être traduite en une transformation opérationnelle.

La principale différence entre les deux approches est le fait que l'approche opérationnelle nécessite de décrire le contrôle comme une partie de la transformation, tandis que cet aspect est géré par le moteur d'exécution dans le cas de l'approche relationnelle.

Un autre aspect différenciant des transformations de modèles est la question de la représentation des modèles. Si le type de structure de données permettant de représenter un modèle peut être indépendant de l'approche, la représentation choisie a toutefois des conséquences concrètes concernant les outils utilisés pour les transformations. On peut distinguer ainsi rois grandes familles d'outils de transformation :

Transformation de texte (fichiers). L'approche la plus simple et intuitive à comprendre, mais aussi, historiquement, la plus ancienne, consiste à transformer séquentiellement des fichiers donnés en entrée à des scripts, qui produisent eux-mêmes des fichiers en sortie. C'est l'approche adoptée par les outils s'appuyant sur AWK, sed ou Perl. Ils conviennent particulièrement bien aux transformations les plus simples, qui sont généralement assez faciles à comprendre et à maintenir. Cependant, cette approche nécessite de *parser* l'entrée et de sérialiser la sortie, et ne permet pas de manipuler une structure plus abstraite [GLR+02].

Transformation d'arbres. La structure d'arbre est une seconde représentation courante des modèles. L'arbre en entrée est parcouru et des nœuds sont créés au fur et à mesure de l'exécution des pas de transformation pour générer un arbre de sortie (ou modifier directement l'arbre source). C'est l'approche des outils de transformation reposant sur XML tels que XSLT [W3C99] ou XQuery [21] [BCF+10, RCD+11]. C'est aussi traditionnellement l'approche des compilateurs et, plus généralement, des outils de l'espace technique *grammarware* (les langages respectent des grammaires — des métamodèles — conformes à EBNF).

Transformation de graphes. Un troisième type de représentation de modèle possible est le graphe (orienté et étiqueté). C'est souvent la représentation adoptée par les outils modernes dédiés à la modélisation et aux transformations de modèles (par exemple AGG [Tae04]). Cette représentation est fortement liée à l'approche de développement de transformation par modélisation que nous avons décrite plus haut. C'est aussi la principale représentation de modèles de la plupart des outils que nous allons présenter dans la section suivante.

2.2.3 Outils existants

Dans cette section, nous présentons différents outils utilisés dans le cadre de l'IDM pour modéliser ou développer des transformations de modèles. Ces outils adoptent l'une ou l'autre des approches décrites dans la section précédente, voire une combinaison de plusieurs méthodes. Du fait de leur faible niveau d'abstraction et de leur usage peu approprié pour développer des transformations complexes dans un contexte industriel, nous ne nous étendrons pas sur les outils de transformation de texte (type AWK, sed ou Perl).

Outils généralistes

Pour écrire une transformation de modèles, il est nécessaire de sélectionner non seulement une approche, mais également un langage et un type d'outil. Si le MDA plaide en faveur

21. Plus exactement XQuery Update Facility [RCD+11] étant donné que XQuery 1.0 n'a pas de fonctionnalité pour modifier les données.

d'outils et de langages dédiés (DSL), il est tout à fait possible d'écrire une transformation de modèles comme n'importe quel autre programme informatique, en choisissant un langage généraliste. On l'utilise généralement avec un *framework* fournissant des fonctionnalités pour manipuler les modèles.

EMF. EMF [SBPM09] est un *framework* de modélisation ouvert initié par IBM. Intégré à l'environnement Eclipse, il comprend une infrastructure de génération de code pour développer des applications à partir de modèles pouvant être spécifiés en utilisant UML, XML ou Java. Ces modèles peuvent ensuite être importés et manipulés avec les services fournis par EMF. Le métamodèle central du *framework* est Ecore. Il est aligné sur EMOF (le passage de l'un à l'autre consiste essentiellement en quelques renommages), ce qui en fait son implémentation de référence. Compte tenu de l'environnement (Eclipse) dans lequel il est intégré, ainsi que du langage de programmation généraliste utilisé (Java) et des standards liés (XML et UML), EMF est une technologie intéressante dans le cas d'un développement de transformation par des utilisateurs « non spécialistes » [22]. Il a l'avantage d'être bien outillé et maintenu, de bénéficier d'une large communauté d'utilisateurs et de pouvoir facilement être couplé avec les outils des écosystèmes Java, UML et XML. Nous reparlerons par la suite de EMF qui nous a servi de technologie de support pour le développement de nos outils.

XML. Une famille d'outils utilisables pour les transformations est celle de l'espace technique XML. Les modèles sont des arbres conformes à des schémas, et des outils de transformation tels que XSLT [W3C99] et XQuery Update Facility [RCD+11] permettent de parcourir et transformer les documents XML. Bien que verbeuse, la syntaxe XML a l'avantage d'être simple et très répandue. Beaucoup d'outils et d'applications utilisant nativement XML, il peut être intéressant d'avoir une vision *arbre XML* des modèles. Ces outils peuvent aussi être utilisés en conjonction de *frameworks* tels que EMF sans adaptation majeure, ceux-ci ayant généralement une fonctionnalité de sérialisation XML (standard XMI de l'initiative MDA).

Outils de transformation d'arbres. Les outils généralistes de transformation d'arbres — plus couramment associés aux mondes de la compilation, de l'analyse de code et de la réécriture qu'à l'IDM — ont aussi leur rôle à jouer en IDM et peuvent fournir une base technique solide pour la construction d'autres outils. Outre le langage Tom [MRV03, BBK+07, BB+13] sur lequel nous nous sommes appuyés pour ce travail, d'autres outils de la communauté présentent un intérêt certain. Notons par exemple Rascal [23] [KHVDB+11, KvdSV11], le langage de métaprogrammation conçu pour l'analyse de programmes qui succède à ASF+SDF [BDH+01]. Il existe sous forme d'un *plugin* Eclipse et permet d'analyser, d'effectuer des requêtes et de transformer des programmes sous la forme de modèles conformes à M3 [24].

Pour le même type d'utilisation, l'outil Spoofax [25] [KV10] — lui aussi disponible sous forme d'un *plugin* Eclipse — intègre Stratego [VBT98],[Vis01a] et SDF. Il permet d'écrire des transformations de programmes en utilisant des règles et des stratégies de réécriture.

Le système Maude [26][CM96, CELM96, CDE+02] est un autre outil de réécriture. Il comprend une infrastructure orientée objet permettant d'implémenter des modèles et métamodèles. Des travaux ont été menés dans ce sens par plusieurs équipes [RRDV07, RV08, Rus11],

22. Nous désignons par *utilisateur non spécialiste* tout utilisateur ayant bénéficié d'une formation satisfaisante en informatique avec un socle théorique minimal, un apprentissage de la programmation orientée objet, ainsi que l'utilisation d'outils tels que Eclipse. Par exemple, un jeune ingénieur en début de carrière est un développeur non spécialiste. Son expertise est donc bien supérieure à celle d'un étudiant n'ayant pas encore reçu de formation, mais inférieure à celle d'un ingénieur expert de son domaine depuis de longues années.

23. Voir `http://www.rascal-mpl.org/`.

24. Métamodèle générique pour représenter du code parsé dans Rascal, `http://tutor.rascal-mpl.org/Rascal/Libraries/lang/java/m3/m3.html`.

25. Voir `http://strategoxt.org/Spoofax/`.

26. Voir `http://maude.cs.uiuc.edu/`.

ce qui a donné naissance à divers outils dont le projet MOMENT [27] qui permet d'effectuer des transformations de modèles EMF avec Maude [BM09, BÖ10].

Outils de réécriture de graphes. L'une des représentations de modèles pouvant être le graphe, l'utilisation d'outils de réécriture de graphes est une approche qui a été expérimentée [Tae10, SK08] en utilisant des outils catégoriques tels que le *single-pushout* et le *double-pushout*, ainsi que le formalisme de spécification *Triple Graph Grammar* (TGG) [Kön05]. Dans cette catégorie d'outils, on notera Moflon [28] qui repose sur le mécanisme TGG pour implémenter des transformations de modèles [AKRS06], Henshin [29][ABJ+10] reposant sur le système AGG [30][Tae04], ainsi que VIATRA2 [31] [VB07]. Du fait de la complexité des algorithmes liés à la transformation de graphes — et donc des ressources nécessaires pour transformer efficacement des graphes de grande taille —, le principal problème de l'approche par réécriture de graphes réside dans le coût élevé des transformations et la difficulté de passer à l'échelle.

Outils dédiés

La solution largement mise en avant en IDM pour le développement de transformations de modèles est l'utilisation de langages dédiés. On pense notamment au standard QVT [OMG08] proposé par l'initiative MDA.

QVT. Il est composé de trois langages de transformation différents, comme illustré par la figure 2.4 : d'une part QVT-Relations et QVT-Core pour la partie déclarative, d'autre part QVT-Operational pour la partie impérative.

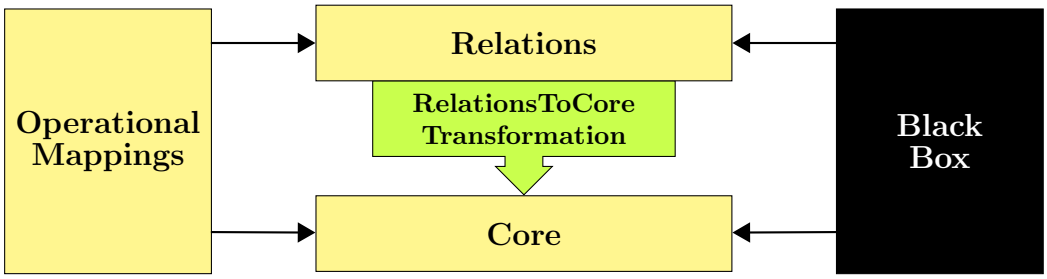


FIGURE 2.4 – Architecture du standard QVT [OMG08].

QVT-Relations et QVT-Core constituent la partie déclarative de QVT. QVT-Relations est un langage orienté utilisateur avec une syntaxe textuelle et graphique permettant de définir des transformations à un niveau d'abstraction élevé (représentation graphique possible). QVT-Core est quant à lui plus bas niveau avec uniquement une syntaxe textuelle. Il sert à spécifier la sémantique de QVT-Relations sous la forme d'une transformation *Relations2Core*. L'expression d'une transformation avec QVT déclaratif est une association de motifs (*patterns*) entre la source et la cible. Bien que relativement complexe à prendre en main, il permet de définir des transformations bidirectionnelles. OptimalJ était une implémentation commerciale de QVT-Core, mais son développement a cessé. Les outils Medini-QVT [32], Eclipse MMT (anciennement Eclipse M2M [33]) ainsi que MOMENT [BM09] implémentent quant à eux QVT-Relations.

Le langage QVT-Operational supporte la partie impérative de QVT. De par la nature même du style impératif, la navigation et la création d'éléments du modèle cible sont explicites. QVT-Operational étend QVT-Relations et QVT-Core par ajout de constructions impératives telles

27. Voir `http://moment.dsic.upv.es/`.
28. Voir `http://www.moflon.org/`.
29. Voir `http://www.eclipse.org/modeling/emft/henshin/`.
30. *Attributed Graph Grammar* : `http://user.cs.tu-berlin.de/~gragra/agg/`.
31. Sous-projet de GMT : `http://www.eclipse.org/gmt/`.
32. `http://projects.ikv.de/qvt/wikia`
33. `http://www.eclipse.org/m2m/`

que la séquence, la répétition, la sélection, *etc.* ainsi que par des constructions OCL. SmartQVT et Eclipse MMT Operational QVT implémentent QVT-Operational.

QVT-*like*. Les outils dont nous allons parler ici n'implémentent pas, à proprement parler, le standard QVT, mais sont cependant suffisamment proches et similaires pour être souvent classés dans la catégorie des *QVT-like*. Le premier d'entre eux est très certainement ATL [34] [BDJ+03, BJRV05, JK06, JABK08] : il s'agit d'un des langages les plus utilisés dans la communauté des modèles et est intégré dans l'environnement Eclipse [AI04]. Il permet de spécifier une transformation d'un ensemble de modèles sources en un ensemble de modèles cibles. Écrire un programme ATL consiste à écrire des règles (*rules*) en donnant une source et une cible. Une règle peut faire appel à des fonctions d'aide (*helper*) permettant d'effectuer un traitement (collecte de valeurs dans une liste, *etc.*). Outre ATL, d'autres outils *QVT-like* existent, notamment Tefkat [35] [LS06] (qui propose une syntaxe à la SQL et offre une implémentation de QVT-Relations), GReAT [AKS03, BNvBK07] et VIATRA2 [VB07], qui sont des outils de transformation de graphes. Ces deux derniers sont distribués sous forme d'un *plugin* Eclipse ce qui leur permet d'interagir avec l'environnement EMF. Enfin, dans cette catégorie d'outils, nous pouvons ajouter JQVT [36] qui est un moteur QVT pour Java. Il est intégré dans Eclipse, est compatible avec EMF et n'utilise pas OCL, mais Xbase [37].

Kermeta. Il s'agit d'un environnement de métamodélisation dans Eclipse. Il permet de définir et spécifier des langages de modélisation dédiés. Le langage Kermeta [38] [DFF+10] mêle plusieurs paradigmes de programmation (orienté objet, orienté aspects [MFJ05, MFV+05], par contrats, fonctionnel) et fournit un support complet de Ecore et de EMOF. Ainsi, tout modèle Kermeta conforme à EMOF est aussi un programme Kermeta valide. Une définition Kermeta peut être compilée pour produire un modèle exécutable. Le code généré est exprimé en Java et Scala, qui peut être ensuite compilé pour être exécuté sur la machine virtuelle Java.

2.3 Limitations actuelles et points d'amélioration

Le lancement de l'initiative MDA a donné un véritable élan à l'IDM et de nombreux outils ont vu le jour. Cependant, malgré l'intérêt indéniable de cette approche pour le développement logiciel, l'IDM n'est pas encore fortement établie dans les chaînes de développement industrielles et gagne lentement du terrain.

La complexité des systèmes s'accroissant, l'industrie voit dans l'IDM une solution pour améliorer leur développement tout en diminuant leurs coûts. En effet, pour appréhender cette complexité, il est nécessaire pour les développeurs d'abstraire les problèmes. Pour ce faire, il faut des outils accessibles et utilisables dans un cadre industriel.

La grande diversité des outils peut aussi être un frein à l'adoption des techniques de l'IDM : un trop grand nombre d'outils et de technologies segmente le domaine, ce qui peut empêcher que les communautés atteignent une masse critique. De plus, beaucoup d'outils adoptant les concepts de l'IDM sont issus de la recherche et sont utilisés par un public académique. N'ayant pas forcément vocation à industrialiser les prototypes développés, le monde de la recherche produit souvent des outils peu accessibles tant sur leur forme (ergonomie, non conformité aux pratiques industrielles habituelles) que sur le fond (concepts peu connus et maîtrisés par peu de monde, expertise nécessaire). Ce frein lié à la multiplicité des outils peu accessibles peut être levé par l'établissement de standards tels que ceux proposés par l'OMG, ainsi que par le déploiement d'environnements comme Eclipse pour pousser à la cristallisation et au développement de communautés autour des technologies liées.

34. `http://www.eclipse.org/atl/`
35. `http://tefkat.sourceforge.net/`
36. `http://sourceforge.net/projects/jqvt/`
37. `https://wiki.eclipse.org/Xbase`
38. `http://www.kermeta.org`

Un autre aspect limitant vient de l'adaptation des chaînes de développement logicielles existantes à ces nouvelles méthodes de développement et à ces nouveaux outils. Compte tenu de leur coût d'intégration et des changements induits par une telle modification d'un processus de développement, il n'est pas raisonnable d'adopter un nouvel outil encore à l'état de prototype ou nécessitant des compétences spécifiques maîtrisées par un petit nombre d'experts ou encore n'ayant aucun lien avec des technologies déjà présentes dans la chaîne de développement existante. Il faut donc mettre en œuvre des ponts entre les technologies pour les rendre plus attrayantes et plus facilement intégrables, afin de faciliter ce processus fortement consommateur de ressources.

Outre l'adaptation normale des chaînes de développement aux nouveaux outils, certains domaines ont des contraintes strictes de qualité concernant les outils de production du logiciel. C'est le cas des chaînes de développement des systèmes critiques que l'on retrouve notamment en aéronautique, dans le secteur automobile, ainsi que dans celui de la santé. Les logiciels produits et servant à produire sont soumis à des exigences légales nécessitant des processus lourds de validation du logiciel. Le domaine de l'IDM étant relativement jeune, il n'est pas encore pleinement adopté dans ce type d'industrie qui a besoin d'outils adaptés à ses contraintes permettant de faciliter la validation du logiciel à moindre coût (et présenter ainsi un intérêt supérieur par rapport à l'existant).

L'un des aspects les plus intéressants est l'usage de l'IDM dans le cadre du développement fiable : pour les problématiques légales évoquées précédemment, la demande industrielle est très forte, et peu de réponses concrètes existent ou alors pour des coûts déraisonnables. C'est dans ce contexte que nous nous proposons d'apporter un élément de solution avec ce travail, afin d'accompagner l'adoption de nouveaux outils tout en améliorant la qualité du logiciel. Nous abordons la problématique de la validation du logiciel dans le chapitre 3.

Chapitre 3

Vérification du logiciel

Il est extrêmement difficile de garantir l'absence totale d'erreurs dans un logiciel. Et cette tâche est d'autant plus ardue que le logiciel est complexe. Les bogues pouvant avoir des conséquences désastreuses et coûteuses, il est nécessaire de valider, vérifier et tester le logiciel avant leur mise en production. Dans ce chapitre, nous présentons le contexte de la vérification du logiciel dans lequel ce travail s'inscrit. Nous expliquons différentes approches pour vérifier le logiciel 3.1. Nous présentons aussi la qualification et la certification 3.2 qui exigent une traçabilité.

3.1 Approches pour la vérification du logiciel

Dans cette section, nous abordons différentes manières de vérifier le logiciel afin d'en améliorer sa qualité, et donc la confiance que l'utilisateur peut avoir dans les logiciels qu'il utilise. Nous présentons d'abord des approches logicielles pratiques comme la relecture, les tests et la simulation, puis la preuve et le *model-checking* qui sont des approches formelles. Ces méthodes pour vérifier le logiciel et améliorer sa qualité sont généralement utilisées de manière conjointe, selon le niveau d'exigence visé (les contraintes pour une application sur ordiphone ne sont pas les mêmes que pour celles d'un calculateur de vol).

3.1.1 Relecture

Une première approche évidente pour tout développeur est la relecture de code. Informelle, elle consiste à parcourir du code développé précédemment et à s'assurer qu'il est conforme aux exigences. Pour améliorer la fiabilité d'une relecture, la tendance est de faire relire du code d'un développeur à un autre développeur n'ayant pas participé au développement, et de croiser les relectures. Cependant, si cette méthode est largement répandue et encouragée, elle est loin d'être fiable. En effet, la relecture dépend totalement d'un relecteur humain dont l'état au moment de la relecture (compétences, fatigue, concentration, etc.) conditionne le résultat. Si la relecture permet de repérer les bogues les plus évidents, elle permet difficilement de repérer les plus subtils. De plus, dépendant des compétences et de l'expérience du relecteur, cette méthode est fortement soumise à son intuition. Une vérification d'un logiciel par relecture pourra ainsi être très différente selon le développeur. Si la relecture est indispensable dans tout développement, elle n'est pas suffisante pour du logiciel exigeant une forte fiabilité. Elle présente toutefois l'avantage d'être naturelle à tout développeur ayant suivi une formation adéquate [39] et d'être peu coûteuse : en effet, elle ne nécessite pas de compétences et d'outils supplémentaires qu'il a été nécessaire pour écrire le logiciel à relire. Cette approche de

39. La relecture fait partie intégrante de la méthodologie du développement généralement enseignée dans les formations en informatique.

vérification peut éventuellement être suffisante pour des applications peu complexes, non critiques et peu diffusées (par exemple : un script pour renommer des fichiers dans un répertoire personnel). Elle reste efficace pour trouver des erreurs et constitue un élément fondamental de certaines approches agiles.

3.1.2 Tests

Une seconde approche est de tester tout programme informatique avant sa mise en production, l'intensité des tests permettant d'obtenir plus d'assurance. Il existe diverses méthodologies et techniques pour tester du code. On citera par exemple la mise en place de tests unitaires censés tester des *unités* de code (fonctions, classes). Des *frameworks* dédiés aux tests unitaires ont ainsi été développés, comme par exemple JUnit [40]. Cela a été popularisé par les méthodes agiles, notamment *eXtreme Programming* [BA04] qui repose en partie sur l'approche dirigée par les tests (*Tests Driven Development* [Bec03]). Remarquons que les scénarios de tests peuvent être considérés comme une forme de spécification. Bien que l'utilisation de tests permette d'améliorer grandement la qualité du logiciel durant sa phase de développement en réduisant les défauts [WMV03], la solution des tests unitaires commence à montrer ses limites lors de développements de logiciels complexes où les bogues peuvent avoir des origines plus nombreuses. Du fait des coûts et des délais de développement, il n'est pas rare de ne pas écrire de tests pour se concentrer sur le logiciel lui-même. Par conséquent, certaines parties de code ne sont pas testées et sont donc susceptibles de contenir des bogues. Des techniques de génération automatique de tests ont vu le jour pour augmenter la couverture de tests et tester au maximum les cas limites. Nous noterons par exemple QuickCheck [CH00, Hug10] pour tester les programmes Haskell, ainsi que ses implémentations et variantes pour d'autres langages tels que Scala [41], Dart [42], Java [43], Ruby [44], Python [45], *etc.*

Ces méthodes ont été transposées dans le cadre de l'IDM. L'approche dirigée par les tests [GP09] s'est développée, et des travaux de vérification par le test ont été menés pour les transformations de modèles [FSB04]. Des *frameworks* de test [LZG05], ainsi que des outils de génération de tests [BFS+06, Lam07, SBM09] et de génération de code tests [RW03] ont donc aussi vu le jour, accompagnés d'outils et de techniques de qualification des données de tests pour les transformations de modèles [FBMT09].

Les tests permettent de détecter des comportements anormaux, mais d'autres comportements anormaux peuvent aussi naître d'une combinaison de comportements normaux de modules fonctionnels indépendamment. Le problème pouvant alors provenir d'une incompréhension entre les équipes de développeurs ou entre le client et le fournisseur (spécification ambiguë). Si l'intégration de tests est absolument nécessaire dans le processus de développement d'un logiciel, elle se révèle insuffisante dans le cas du logiciel critique. En effet, tester intensément un logiciel permet de tester son comportement dans la plupart des cas, mais rien ne garantit que toutes les situations ont été testées.

3.1.3 Simulation

Il peut aussi être extrêmement difficile de tester correctement un système du fait de sa complexité, des fortes ressources demandées ou de la particularité de sa plateforme d'exécution. Les tests écrits et menés durant le développement peuvent se révéler peu réalistes ou fort éloignés des conditions réelles d'utilisation. Pour répondre à ce problème de réalisme du comportement d'un système sous conditions d'utilisation réelles, une approche liée aux tests revient à simuler le système pour étudier son comportement et détecter les anomalies. L'intérêt de ce type d'approche est que l'on peut travailler sur un système qui serait coûteux à

40. http://www.junit.org
41. ScalaCheck : http://code.google.com/p/scalacheck/
42. PropCheck : https://github.com/polux/propcheck
43. QuickCheck pour Java [EF13] : https://bitbucket.org/blob79/quickcheck
44. RushCheck : http://rushcheck.rubyforge.org/
45. qc : http://github.com/dbravender/qc

déployer pour tester en conditions réelles. Notons par exemple les domaines du nucléaire ou de l'avionique qui ne permettent pas aisément (et à faible coût) des tests logiciels en conditions réelles, en particulier lors des premières étapes de développement. L'inconvénient de ce type de méthode de vérification est qu'il faut reproduire fidèlement un environnement et que les tests sont fortement conditionnés par la plateforme d'exécution sur lesquels ils sont exécutés et par les technologies employées. Dans le cas d'une informatique simple, la simulation est une option intéressante, mais sa difficulté de mise en œuvre croît avec la complexité du système (nombreuses dépendances, systèmes exotiques, matériel non disponible hors chaîne de production dédiée, modèles physiques complexes, *etc.*). Ainsi, en aéronautique, il est extrêmement complexe et coûteux de modéliser intégralement l'environnement afin de simuler tous les systèmes de l'avion. L'approche adoptée revient alors à procéder à une succession de phases simulant des modèles de l'environnement mécanique, physique, des calculateurs, du réseau, du logiciel, *etc.* La dernière étape avant le vol réel est l'*Aircraft Zero — Iron Bird* qui intègre toute l'informatique réelle sur un banc d'essai.

3.1.4 Preuve

À l'opposé de ces approches vues comme très pragmatiques et largement utilisées dans l'industrie, une autre approche pour améliorer la confiance dans un logiciel consiste à prouver formellement les algorithmes. Le problème est alors formalisé sous la forme d'axiomes et de buts qu'il faut atteindre en démontrant des théorèmes. Une preuve mathématique peut être écrite à la main, mais pour faciliter le processus et pour diminuer les risques d'introduction d'erreurs liées au facteur humain, des outils assistants à la preuve tels que Coq [The04, BC04] et Isabelle/HOL [NPW02] ont été développés. Une preuve Coq est censée donner une forte confiance dans le logiciel développé et prouvé. Cependant, si la preuve de la correction d'un algorithme donne une garantie irréfutable du bon comportement de cet algorithme, elle ne donne pas obligatoirement la garantie du bon comportement du logiciel. En effet, ce sont les algorithmes et les spécifications qui sont généralement formellement prouvés et non les implémentations elles-mêmes [46]. Or, le facteur humain n'est pas à négliger, l'implémentation concrète du logiciel dépendant fortement du développeur et des outils utilisés durant le processus. En outre, lors d'une preuve, le contexte réel n'est pas forcément pris en compte et certaines conditions d'utilisation réelles peuvent fortement influencer le comportement et la fiabilité de l'application.

3.1.5 Model-checking

Le *model-checking* est une autre approche formelle que l'on peut considérer comme étant entre preuve et test. Elle consiste à abstraire (modéliser) un problème ou un système selon un formalisme (langage) donné, puis à explorer l'espace d'états possibles de ce système pour vérifier qu'aucun état ne viole une propriété donnée (un invariant par exemple). On peut considérer que cela revient à du test exhaustif, ce qui a donc valeur de preuve. L'intérêt du *model-checking* est que — contrairement aux tests — il est généralement indépendant de la plateforme d'exécution et qu'il apporte une vérification formelle du système. Par exemple, les outils CADP [47] [GLMS11] et TINA [48] [BRV04] sont deux *model-checkers* dont les formalismes d'expression des modèles sont respectivement LOTOS et les réseaux de Petri. On peut toutefois reprocher au *model-checking* de ne pas toujours être suffisamment proche de la réalité et d'avoir un coût en ressources élevé dans le cas de systèmes complexes. Cela a conduit certains à compléter le *model-checking* par l'analyse à l'exécution sur des applications réelles simulées [BM05]. L'IDM reposant sur les modèles qui sont des abstractions de problèmes,

46. « Beware of bugs in the above code ; I have only proved it correct, not tried it » – D.E. Knuth, 1977 ; citation et explication accessibles sur sa page personnelle : `http://www-cs-faculty.stanford.edu/~uno/faq.html`

47. `http://cadp.inria.fr`

48. `http://projects.laas.fr/tina`

il est naturel d'adopter le *model-checking* pour vérifier le logiciel produit par ces méthodes. On peut adjoindre des contraintes aux modèles avec des langages tels qu'OCL dans le cas de UML, mais un modèle n'est pas nécessairement immédiatement vérifiable. Il nécessite souvent une ou plusieurs transformations afin de pouvoir être vérifié au moyen d'un *model-checker*. C'est une approche courante que d'opérer une suite de transformations permettant de passer du modèle servant de spécification au modèle vérifiable, ainsi que de la spécification vers l'application réelle. Cependant, comme précédemment, il s'agit à nouveau de vérifier un modèle et non pas le code généré réel. Chaque transformation se doit donc d'être simple afin que l'on puisse garantir la préservation du comportement d'un pas à l'autre. Ce type d'approche est très utilisé en IDM. Nous notons immédiatement qu'une telle approche pour s'assurer du bon fonctionnement du logiciel nécessite non seulement une vérification du modèle, mais aussi une garantie que la transformation elle-même est correcte.

3.2 Certification et qualification

Le processus de certification n'est pas nécessaire pour tous les logiciels et tous les secteurs d'activité. Il fait en revanche partie intégrante du processus de développement logiciel dans le cadre de développement de systèmes critiques, par exemple dans les domaines de l'aéronautique, de l'automobile (du transport en général) et de la santé. La loi impose le respect d'exigences en fonction de crédits de certification demandés.

Définition 31 (Crédit de certification)**.** *Acceptation par l'autorité de certification qu'un processus, un produit ou une démonstration satisfait les exigences de certification.*

Définition 32 (Certification)**.** *Processus d'approbation par une autorité de certification d'un produit.*

Ces exigences sont fixées par des standards de certification : en aéronautique, le standard actuel est la norme DO-178C/ED-12C [49] [Spe11a]. Elle donne des objectifs (et non des moyens) dont le nombre et le niveau augmentent avec le niveau de criticité. Ces niveaux de criticité (ou niveaux DAL, pour *Development Assurance Level*) sont notés de A à E, A étant le plus critique, E le moins critique :

Niveau A : (Erreur catastrophique) un défaut du système ou sous-système étudié peut provoquer un problème catastrophique (sécurité du vol ou atterrissage compromis, crash de l'avion) ;

Niveau B : (Erreur dangereuse) un défaut du système ou sous-système étudié peut provoquer un problème majeur entraînant des dégâts sérieux voire la mort de quelques occupants ;

Niveau C : (Erreur majeure) un défaut du système ou sous-système étudié peut provoquer un problème sérieux entraînant un dysfonctionnement des équipements vitaux de l'appareil (pas de mort) ;

Niveau D : (Erreur mineure) un défaut du système ou sous-système étudié peut provoquer un problème pouvant perturber la sûreté du vol (pas de mort) ;

Niveau E : (Erreur sans effet) un défaut du système ou sous-système étudié peut provoquer un problème sans effet sur la sûreté du vol.

Dans le cadre du processus de certification de systèmes critiques, les outils utilisés pour le développement doivent être vérifiés afin de s'assurer de leur bon fonctionnement et qu'ils n'introduisent pas d'erreurs dans le logiciel.

Les pratiques de développement logiciel évoluant, notamment par l'adoption croissante de l'approche dirigée par les modèles qui plaide pour l'automatisation maximale des tâches,

49. La notation DO- correspond au nom donné à ce standard aux États-Unis d'Amérique, tandis que la notation ED- est celle en vigueur en Europe.

de nouveaux outils sont intégrés aux processus de développement. L'introduction d'outils automatiques dans une chaîne de développement de système soumis à certification impose l'une des deux approches suivantes :

— vérifier les données produites par l'outil *a posteriori* ;

— qualifier l'outil.

La qualification d'un logiciel participe à atteindre un objectif exigé par le processus de certification et permet d'obtenir de la confiance dans les fonctionnalités d'un outil. Ce processus de qualification peut être appliqué à une ou plusieurs fonctions dans un outil, à un outil unique ou à un ensemble d'outils.

Définition 33 (Qualification)**.** *La qualification d'outils est le processus nécessaire pour obtenir des crédits de certification d'un outil. Ces crédits peuvent uniquement être garantis pour un usage donné dans un contexte donné.*

L'intérêt de la qualification d'un outil est d'obtenir suffisamment de confiance en cet outil pour pouvoir l'intégrer dans une chaîne de développement sans avoir à vérifier ses données de sortie ensuite. Le choix entre l'approche de qualification et de vérification *a posteriori* revêt un aspect stratégique : si la qualification d'un outil peut avoir un coût élevé fixe, elle n'en génère pas plus par la suite, tandis que la vérification *a posteriori* aura un coût récurrent.

La DO-178/ED-12 définit des catégories d'outils (*sans impact*, *vérification* et *compilateur, générateur de code*) ainsi que des critères de qualification. Elle précise les exigences de qualification adaptées aux différents types d'outils et à leur utilisation. La norme DO-330/ED-251 [Spe11b] donne des instructions pour la qualification d'outils. Elle définit aussi cinq niveaux de qualification d'outils (TQL — *Tool Qualification Levels*) qui sont fonction de ces trois critères ainsi que de la criticité du logiciel développé :

TQL-1 à 3 : ces niveaux concernent les générateurs de code et outils dont la sortie fait partie du logiciel embarqué et qui peuvent donc introduire des erreurs ;

TQL-4 à 5 : ces niveaux concernent les outils de vérification, qui ne peuvent pas introduire d'erreurs, mais qui peuvent échouer à en détecter (outils de tests, analyseurs de code statique)

Selon son utilisation et selon le type de production de l'outil, le niveau de qualification sera différent. Un outil ne peut donc être qualifié une fois pour toutes. En revanche, il peut être *qualifiable* pour chaque projet, c'est-à-dire *pre qualifié* pour chaque projet.

L'IDM plaidant pour l'automatisation des tâches et l'usage d'outils automatisant les tâches, il faut vérifier ces outils. Dans ce contexte, l'un des problèmes majeurs de l'IDM est le grand nombre d'outils non qualifiés et non certifiés impliqués dans les chaînes de développement, notamment des générateurs de code, des outils de traduction, et de manière plus générale des outils automatisant au maximum les tâches de transformation.

3.3 Traçabilité

Une problématique fondamentale des compilateurs dans le cadre de la certification d'un logiciel est d'assurer la traçabilité entre le code source et le code binaire, ce qu'exige la norme DO-178/ED-12. Cette exigence pour les compilateurs et générateurs de code est généralisable à toutes les transformations, donc aux transformations de modèles à partir desquelles les logiciels sont produits. Assurer cette traçabilité permet aussi de respecter la contrainte de séparation de le spécification, de l'implémentation et de la vérification tout en étant capable de relier les unes aux autres.

Définition 34 (Traçabilité)**.** *La norme DO-330 [Spe11b] définit la traçabilité comme une association entre objets telle qu'entre des sorties de processus, entre une sortie et son processus d'origine, ou entre une spécification et son implémentation.*

L'IDM plaidant pour l'automatisation maximale des tâches répétitives et l'utilisation de générateurs de code pour produire le logiciel à partir de modèles, il est nécessaire de vérifier les transformations qui en sont le cœur. L'un des angles d'attaque du problème de la qualification est de fournir des outils de transformations qualifiables, qui assurent la traçabilité des transformations.

Dans le cas d'une transformation de modèle, le métamodèle source fait partie de la spécification et le modèle source la donnée d'entrée. Il convient donc de conserver une trace de la transformation en maintenant un lien entre la source et la cible. On retrouve souvent deux situations :

— la trace se fait au niveau macroscopique (modèle d'entrée et modèle de sortie) et la granularité est extrêmement faible, ce qui rend la trace peu informative ;

— la trace s'opère de manière très détaillée (comme un *debug*) ce qui génère une trace importante de tout ce qui s'est produit durant la transformation. Si toutes les informations sont présentes, se pose le problème de la pertinence des données recueillies : la quantité d'informations peut rendre la trace inexploitable, les éléments jugés intéressants risquant d'être noyés dans la trace.

Outre le respect strict des exigences de qualification, les informations de trace peuvent être très utiles pour des tâches telles que le *debugging* ou l'analyse d'impact (conséquences d'une modification).

La traçabilité des transformations de modèle est un aspect important que la plupart des outils traitent en apportant un support dédié, comme pour les outils QVT, ATL, Tefkat ou Kermeta [FHN+06]. D'autres outils tels que AGG, VIATRA2 et GReAT n'ont pas de support dédié à cette traçabilité, cependant les informations de trace peuvent être créées comme des éléments cibles.

Généralement, la traçabilité est classée en deux catégories : *explicite* et *implicite*. La première signifie que les liens de trace sont directement capturés en utilisant explicitement une syntaxe concrète adéquate. La traçabilité *implicite* est quant à elle le résultat d'une opération de gestion des modèles (une requête sur des artefacts générés par exemple).

Un problème récurrent de l'implémentation de la traçabilité est qu'elle est souvent fortement liée à l'implémentation de la transformation, ce qui est problématique dans le cadre de la qualification qui impose de préserver une séparation entre l'implémentation et la spécification.

Pour la qualification, il faut donc fournir des informations pertinentes sans qu'elles soient perdues dans la masse de données, tout en ayant une traçabilité qui soit suffisamment découplée à l'implémentation, mais donnant une granularité intermédiaire. C'est dans ce contexte que nous nous proposons de travailler, et nous proposons de fournir des outils permettant d'aider à la qualification de transformations de modèles.

Deuxième partie

Contributions

Chapitre 4

Transformations de modèles par réécriture

Dans ce chapitre, nous expliquons l'approche que nous avons adoptée pour mettre en œuvre les transformations de modèles. Dans la section 4.1, nous présentons l'aspect hybride de notre approche, les choix liés ainsi que son intérêt. Nous abordons ensuite la problématique de la représentation des modèles dans notre approche dans la section 4.2. Nous expliquons dans la section 4.3 les mécanismes en jeu pour notre approche de transformation de modèles par réécriture.

4.1 Choix et intérêt d'une approche hybride

Si le principe de la transformation elle-même est un principe qui peut sembler classique, car adopté par d'autres outils tels que QVT ou ATL, notre environnement de travail est très différent de celui des autres outils, ce qui fait l'originalité de notre approche.

Comme nous l'avons vu dans le chapitre 2, il existe de multiples approches pour implémenter des transformations de modèles. L'un des choix possibles est d'opter pour une approche par programmation en utilisant un langage généraliste tel que Java. On lui adjoint un *framework* et un ensemble de bibliothèques pour faciliter la représentation et la manipulation de modèles, notamment EMF. Ensuite, la transformation revient à l'écriture d'un programme qui charge un modèle en entrée et produit un modèle en sortie. Cette approche présente certains avantages. Les langages généralistes sont habituellement plus accessibles à la plupart des développeurs tant par le fait qu'une formation d'informaticien comprend généralement l'apprentissage de langages généralistes que par le fait que ces langages suivent des paradigmes de programmation bien identifiés. En outre, cette plus grande accessibilité a aussi pour effet de faciliter le développement de communautés, ainsi que d'outils d'aide au développement (IDE, bibliothèques, etc.) et de documentation. Ces effets entretiennent alors le cercle vertueux de la facilité d'accès. À l'opposé, il peut être difficile ou long d'implémenter une transformation complexe alors qu'un langage dédié proposerait les constructions adéquates pour ce type de tâche.

Une deuxième possibilité est de suivre les recommandations MDA qui encouragent l'utilisation de DSL. Cela a l'avantage d'être exactement adapté à la tâche visée. L'inconvénient est que les DSL ont généralement une visibilité plus limitée, avec des communautés plus petites et un outillage moins étoffé. De plus, la compétence est plus difficile à trouver ou induit des coûts de formation plus élevés.

Notre approche est une approche hybride, pour combler le fossé entre les langages généralistes et dédiés, afin de bénéficier du meilleur des deux mondes. Nous proposons d'utiliser le langage Tom pour exprimer des transformations de modèles, ce qui nous permet de nous intégrer dans des langages généralistes tout en apportant des fonctionnalités supplémentaires

par ses constructions. Comme présenté dans le chapitre 1, Tom repose sur le calcul de réécriture et implémente la fonctionnalité de *stratégie de réécriture*, ce qui permet de découpler les règles métier du parcours des structures de données. Nous proposons donc de transformer des modèles par réécriture, en nous appuyant sur les stratégies. Manipulant des termes, il nous faut représenter les modèles sous cette forme pour pouvoir les transformer. Nous avons aussi fait le choix d'une intégration dans un environnement de production pouvant être considéré comme typique, à savoir l'écosystème Java. De plus, nous avons choisi de travailler dans un premier temps avec une technologie courante et très utilisée — EMF — qui devient un standard *de fait*.

Partant de ces choix et de nos contraintes, notre approche consiste à exprimer un modèle EMF sous la forme d'un terme puis à le transformer par réécriture. Nous opérons donc des transformations de modèles en changeant d'espace technologique pour pouvoir nous appuyer sur nos méthodes et outils déjà éprouvés. Nous décrivons le premier aspect de notre approche — la représentation des modèles — dans la section 4.2, puis nous détaillons les principes mis en œuvre dans une transformation dans la section 4.3.

4.2 Représentation de modèles par une signature algébrique

Réécrivant des termes, la première étape de notre approche est d'élaborer une représentation adéquate des modèles. Compte tenu de ce besoin et de notre approche, nous procédons en premier lieu par à un changement d'espace technologique. Un métamodèle dans le monde des modèles correspond à une signature dans celui des termes, tandis qu'un modèle — conforme à son métamodèle — correspond à un terme — conforme à sa signature algébrique —. Lors de ce changement, un aspect *a priori* problématique de notre représentation sous forme de terme d'un modèle vient du fait qu'un terme est une structure arborescente tandis qu'un modèle est généralement vu comme un graphe (un ensemble de nœuds et d'arcs). Une transformation de modèle est donc une fonction de transformation de graphe qu'il faut encoder dans notre contexte. L'un des aspects de notre approche est d'opérer une transformation préalable permettant de considérer une transformation de modèles comme une transformation de termes. Cela est possible étant donné que l'on obtient un arbre de recouvrement par la relation de composition, ce qui nous permet d'établir une vue adéquate des modèles sous la forme d'un terme.

Nous avons donc développé un outil — appelé Tom-EMF — que nous décrivons techniquement dans le chapitre 6, et qui, pour un métamodèle EMF Ecore donné, produit sa signature algébrique Tom correspondante, utilisable dans un programme Tom+Java. Bien que cet outil repose pour le moment sur une technologie particulière, il est tout à fait possible de l'étendre ou d'écrire un outil similaire adapté à d'autres technologies.

Dans notre représentation, pour chaque métaclasse, une sorte lui correspond, ainsi qu'un opérateur. Une métaclasse pouvant avoir des attributs et des opérations, l'opérateur a les champs correspondants. Les relations sont représentées par des attributs simples dans le cas d'associations. Dans le cas de relations de composition, des sortes et opérateurs variadiques additionnels sont créés, et un attribut de ce type est ajouté à l'opérateur ayant la relation de composition.

Pour illustrer notre propos, considérons un métamodèle simple permettant de décrire des graphes (figure 4.1). Un graphe (*Graph*) est composé de *Nodes* et d'*Arcs*. Chaque arc a un poids, une source et une cible. Ces extrémités sont de type *Node*, un *Node* pouvant avoir plusieurs arcs entrants et sortants. La représentation sous forme de signature algébrique de ce métamodèle est alors donnée par le listing 4.1. Les métaclasses *Graph*, *Node* et *Arc* ont chacune une sorte qui lui correspond (nous avons conservé les même noms). Un opérateur — préfixé par `op` dans l'exemple — est associé à chacune d'entre elles. Les attributs présents dans les métaclasses sont bien reportés au niveau des opérateurs (`name` et `weight`). Les relations

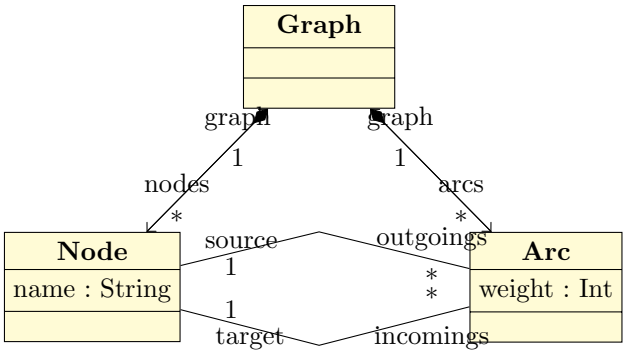


FIGURE 4.1 – Métamodèle des graphes.

d'association sont traduites par des paramètres additionnels : l'opérateur `opArc` possède ainsi deux paramètres supplémentaires `source` et `target`. Le cas des relations de composition (relations `nodes` et `arcs`) est traité par la création d'opérateurs variadiques (`opNodeList` et `opArcList` dans notre exemple) ainsi que de leurs sortes associées (`NodeList` et `ArcList`).

```
Graph = opGraph(nodes:NodeList, arcs:ArcList)

Node  = opNode(name:String, graph:Graph, incomings:ArcList, outgoings:ArcList)

Arc   = opArc(weight:Int, graph:Graph, source:Node, target:Node)

NodeList = opNodeList(Node*)

ArcList  = opArcList(Arc*)
```

Listing 4.1 – Signature algébrique correspondant au métamodèle de la figure 4.1.

Grâce à ce changement d'espace technologique, nous encodons la fonction qui transforme un modèle non plus comme une fonction de réécriture de graphe, mais comme une fonction de réécriture de termes.

4.3 Transformation de modèles par réécriture

Étant en mesure de représenter des modèles sous forme de termes, nous pouvons décrire notre approche pour les transformer. Son principe est similaire à celui de l'approche de QVT et ATL et peut sembler classique dans le domaine des modèles. Toutefois, ce n'est pas le cas dans l'environnement dans lequel nous évoluons, et plus généralement dans le domaine des outils de transformation généralistes dédiés aux arbres. Habituellement, à l'instar d'un compilateur, les outils de transformation d'arbres procèdent à des parcours et à des modifications successives sur un arbre qui est passé de phase en phase de l'outil.

Dans notre approche, nous décomposons les transformations en deux phases distinctes. La première est une transformation par parties qui consiste à créer les éléments cibles du modèle résultant ainsi que des éléments additionnels que nous appelons *éléments resolve*, en référence au *resolve* de QVT et au *resolveTemp* d'ATL. Ces éléments permettent de faire référence à des éléments qui n'ont pas encore été créés par la transformation. La seconde phase, quant à elle, a pour objectif de rendre le modèle cible résultat cohérent, c'est-à-dire conforme au métamodèle cible en éliminant les éléments *resolve* et en les remplaçant par des références vers

les éléments effectivement créés par la transformation. Cette seconde phase n'ajoute aucun nouvel élément cible au résultat.

Pour illustrer notre propos dans ce chapitre, nous nous appuierons sur un exemple visuel permettant de bien comprendre le mécanisme de *décomposition-résolution*. Supposons que nous souhaitons transformer une séquence `A;B` textuelle en sa forme graphique correspondante comme décrit par la Figure 4.2. Dans cet exemple, le choix des couleurs des connecteurs est arbitraire : nous aurions très bien pu choisir de colorer le cercle en vert et le carré en bleu. Supposons donc que ce découpage est spécifié et imposé.

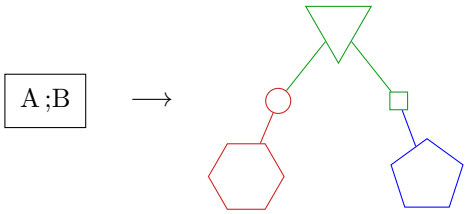


FIGURE 4.2 – Transformation du modèle source `A;B` en un modèle cible graphique.

4.3.1 Approche compositionnelle

Une fois le problème de la représentation des modèles résolu, il est possible d'instancier un modèle (création d'un terme conforme à la signature algébrique) et d'opérer une transformation sur le terme le représentant.

L'écriture d'une transformation de modèles peut se faire par une approche procédurale *monolithique*. L'utilisateur construit la transformation par étapes (*transformation steps*), dont l'ordre s'impose naturellement en fonction des besoins des différents éléments : par exemple, la transformation décrite dans la Figure 4.2 peut se décomposer en trois étapes que nous illustrons dans la Figure 4.3.

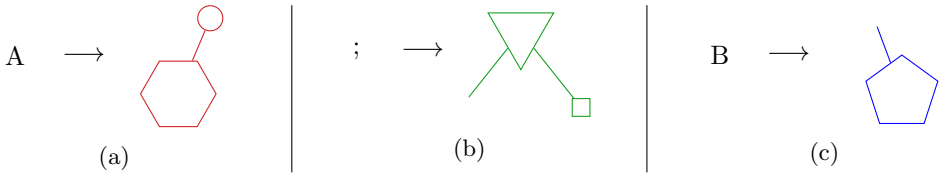


FIGURE 4.3 – Règles de transformation de `A`, `;` et `B`.

La transformation de `A` (étape (a)) donne un hexagone et un cercle rouges connectés, celle de `B` (étape (c)) un pentagone et un connecteur bleus, celle de `;` (étape (b)) produit un triangle et un carré connectés verts, ainsi qu'un connecteur supplémentaire. Pour qu'un connecteur ou arc puisse être créé, il est nécessaire de connaître chaque extrémité. Ainsi, la transformation de `A` ne pose aucun souci particulier : un seul arc est créé entre deux éléments créés (hexagone et cercle) dans cette même étape de transformation. En revanche, la transformation de `B` en un pentagone est censée aussi construire un arc dont l'une des extrémités (petit carré) n'est pas créée dans cette étape de transformation. Il est donc nécessaire que l'étape de transformation construisant cette deuxième extrémité se déroule avant celle produisant l'arc (c). Le carré servant de seconde extrémité à l'arc est construit dans l'étape de transformation de `;` qui devra donc être exécutée avant (c). Nous notons que cette étape génère un autre arc dont l'une des extrémités n'est pas connue dans (b). L'étape de transformation produisant cette extrémité d'arc qui nous intéresse (petit cercle) est (a). Il est donc naturel d'exécuter (a) avant (b). Finalement, pour que cette transformation puisse être exécutée sans qu'il n'y ait de problème d'éléments manquant, l'utilisateur doit adopter les étapes de transformation

dans l'ordre suivant : (a), puis (b), puis (c). S'il ne respecte pas cet ordre, il sera confronté au problème de création d'éléments avec des informations manquantes.

Cependant, cette approche n'est pas toujours possible, notamment lorsque l'on manipule des structures cycliques. Lorsqu'elle est possible, elle nécessite une parfaite expertise ainsi qu'une connaissance globale de la transformation pour être capable d'organiser les différentes étapes. De plus, avec une telle méthode une transformation sera généralement monolithique. Elle sera donc peu générique et le code peu réutilisable, l'encodage du parcours du modèle ainsi que les transformations étant *ad-hoc*. Généralement, le parcours du modèle sera encodé par des boucles et de la récursivité, et un traitement particulier sera déclenché lorsqu'un élément donné sera détecté. Parcours et traitement seront donc étroitement liés. La moindre modification du métamodèle source ou cible implique donc de repenser la transformation.

Nous souhaitons au contraire faciliter le développement et la maintenance du code que l'utilisateur écrit pour une transformation, tout en le rendant réutilisable pour une autre transformation. Il est donc important d'adopter une méthode permettant une grande modularité du code.

Notre approche est d'opérer une transformation par parties : il faut d'abord décomposer une transformation complexe en transformations les plus simples (ce que nous nommons *transformations élémentaires* ou *définitions*). Chacune d'entre elles est décrite par une règle ou un ensemble de règles. La transformation globale est ensuite construite en utilisant ces transformations élémentaires. Mais dans ce cas se pose le problème de la dépendance des définitions entre elles, ainsi que de l'utilisation d'éléments issus d'une transformation élémentaire dans une autre transformation élémentaire. Ce qui a des conséquences sur l'ordre d'application des définitions : il peut être absolument nécessaire qu'une partie de la transformation soit effectuée pour que les autres étapes puissent être appliquées. De plus, par souci d'utilisabilité, nous ne souhaitons pas que l'utilisateur ait besoin de se soucier de l'ordre d'application des transformations élémentaires. Nous souhaitons qu'il se concentre uniquement sur la partie métier de la transformation. Il faut donc mettre en œuvre un mécanisme permettant de résoudre les dépendances.

Dans notre contexte, nous effectuons des transformations dites *out-place*, ce qui signifie que le modèle source n'est pas modifié. Le modèle cible résultant est construit au fur et à mesure de la transformation, et n'est pas obtenu par modifications successives du modèle source — transformation *in-place*, comme le font les outils VIATRA [VVP02] / VIATRA2 [VB07] et GrGen.NET [JBK10] par exemple. Partant de ce constat, les transformations élémentaires composant notre transformation n'entretiennent aucune dépendance dans le sens où la sortie d'une transformation élémentaire n'est pas l'entrée d'une autre transformation élémentaire. Dans notre approche compositionnelle, chaque sortie d'une transformation élémentaire est une partie du résultat final.

Dans l'exemple, nous conservons la décomposition proposée dans la Figure 4.3 en trois règles simples. Nous avons décomposé une transformation complexe en *définitions* indépendantes et nous pouvons les appliquer. Subsiste cependant le problème de l'usage dans une *définition* d'éléments créés dans une autre *définition*. Comme l'ordre d'écriture et d'application des transformations élémentaires ne doit pas être une contrainte pour l'utilisateur, nous avons choisi de résoudre ce problème par l'introduction d'éléments temporaires — éléments dits *resolve* — qui font office d'éléments cibles durant la transformation, et qui sont substitués en fin de transformation lors d'une seconde phase. Cette dénomination fait évidemment référence au *resolve* de QVT et au *resolveTemp* de ATL.

Partant du principe que toutes les transformations élémentaires peuvent être déclenchées indépendamment dans n'importe quel ordre (voire en parallèle), il faut être en mesure de fournir un élément cible lorsque le traitement d'une *définition* le nécessite. Nous proposons donc de construire un terme temporaire représentant l'élément final qui a été ou qui sera construit lors de l'application d'une autre *définition*. Ce terme doit pouvoir être intégré dans le modèle cible temporaire pour être manipulé en lieu et place du terme ciblé censé être construit dans une autre *définition*. Il doit donc respecter les contraintes de types du métamodèle cible tout en portant des informations supplémentaires telles qu'un identifiant, une référence à l'élément

source d'origine et une référence à l'élément cible. Nous étendons donc le métamodèle cible MM_t afin que ces contraintes soient respectées et que le modèle intermédiaire résultant soit conforme à un métamodèle cible étendu, noté $MM_{t_{resolve}}$. Ainsi, tout élément *resolve* $e^i_{t_{resolve}}$ du modèle intermédiaire enrichi $m_{t_{resolve}}$ sera de type un sous-type de l'élément ciblé e^i_t du modèle cible m_t. Les éléments $e^i_{t_{resolve}}$ sont les éléments e^i_t décorés d'une information sur le nom de l'élément cible représenté ainsi que d'une information sur l'élément source dont ils sont issus. En termes de métamodèle (Figure 4.4), pour tout élément cible e^i_t — instance d'un élément E^i_t du métamodèle cible MM_t — issu d'un élément source e^j_s — instance du métamodèle source MM_s — et nécessitant un élément *resolve* $e^i_{t_{resolve}}$ durant la transformation, un élément $E^i_{t_{resolve}}$ est créé dans le métamodèle étendu $MM_{t_{resolve}}$. Cet élément hérite de l'élément cible E^i_t.

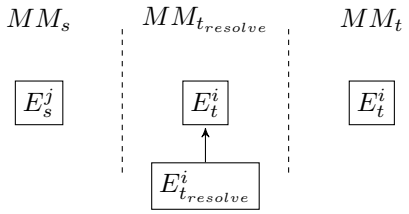


FIGURE 4.4 – Schéma d'extension du métamodèle cible par l'ajout d'éléments intermédiaires *resolve*.

Cette première phase produit donc un modèle cible non conforme au métamodèle cible MM_t de la transformation globale, mais conforme au métamodèle cible étendu $MM_{t_{resolve}}$. Elle peut s'écrire sous la forme d'une fonction $c : MM_s \rightarrow MM_{t_{resolve}}$ qui crée des éléments cible à partir des éléments du modèle source.

La Figure 4.5 permet d'illustrer le mécanisme des éléments *resolve*. Nous reprenons la Figure 4.3 décrivant les trois *définitions* composant notre transformation exemple, et nous intégrons les *éléments resolve* (représentés par des formes en pointillés). Précédemment, nous avons vu que nous pouvions appliquer la *définition* (a) complètement indépendamment étant donné qu'elle ne nécessite aucun résultat ou partie de résultat issu d'une autre *définition*. Cette étape ne change donc pas et ne crée aucun *élément résolve*. La *définition* (b) nécessite en revanche un cercle rouge – normalement créé dans la *définition* (a) – pour pouvoir créer un arc. Un élément dont le type (*cercle pointillé*) est sous-type de l'élément cible (*cercle*) est donc créé. Nous serons donc par la suite en mesure de manipuler le *cercle pointillé* comme un *cercle* classique et de filtrer sur tous les cercles, en pointillés ou non. La couleur donnée à un *élément resolve* dans la Figure 4.5 permet de représenter l'information de la *définition* d'origine de l'élément ciblé par l'*élément resolve* et donc d'encoder le lien entre les deux éléments. Le même principe est appliqué à la *définition* (c) qui nécessite un élément normalement créé dans la définition (b), d'où la génération d'un élément de type *carré pointillé* vert.

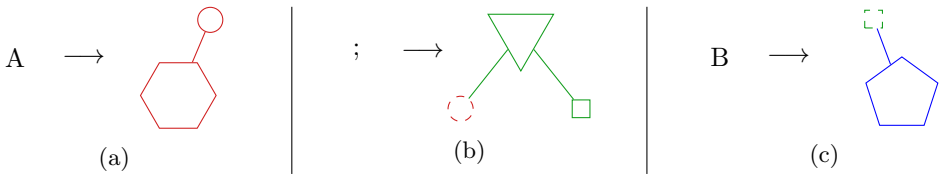


FIGURE 4.5 – Règles de transformation de `A`, `;` et `B` effectives avec la construction d'*éléments resolve* (en pointillés colorés).

Durant cette première phase de transformation par parties, chaque *définition* a produit un ensemble d'éléments tous disjoints, les éléments censés provenir d'autres *définitions* ayant été représentés par des éléments *resolve*.

Nous encodons les *définitions* par des stratégies de réécriture que nous composons avec d'autres stratégies. Il en résulte une stratégie plus complexe qui encode cette première phase de transformation. Dans notre exemple de transformation *Text2Picture*, nous avons identifié trois *définitions* — (a), (b) et (c) — qui seront encodées respectivement par les stratégies S_a, S_b et S_c. Nous les combinons avec la séquence et une stratégie de parcours — *TopDown* ici — pour former la stratégie $S_{phase1} = TopDown(Seq(S_a, S_b, S_c))$.

4.3.2 Résolution - réconciliation

L'application des transformations élémentaires sur le modèle source a produit un résultat intermédiaire non conforme au métamodèle cible MM_t, car composé de plusieurs résultats partiels incluant des *éléments resolve*, comme illustré par la Figure 4.6.

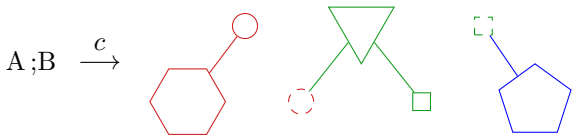


FIGURE 4.6 – Résultat intermédiaire de la transformation, avant phase de *résolution*.

Pour obtenir un résultat cohérent conforme au métamodèle cible, il est nécessaire d'effectuer un traitement sur ce résultat intermédiaire. C'est ce que nous appelons la phase de *résolution* ou *réconciliation*, dont le but est de fusionner des éléments disjoints pour les rendre identiques. Elle consiste à parcourir le terme résultant, à trouver les éléments temporaires *resolve*, puis à reconstruire un terme résultat en remplaçant ces termes temporaires par les termes qu'ils étaient censés remplacer. Étant donné que toutes les *définitions* ont été appliquées, nous sommes certains que l'élément final existe, et qu'il peut se substituer à l'élément temporaire examiné.

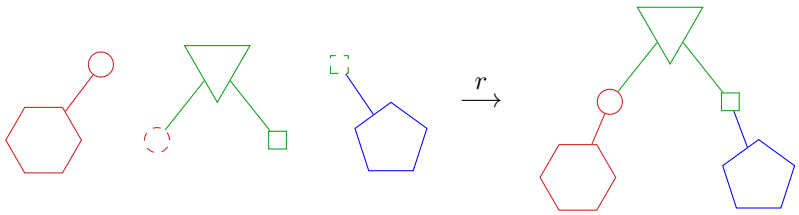


FIGURE 4.7 – Phase de *résolution*.

Cette phase de résolution est elle-même encodée par une stratégie S_{phase2} qui réécrit un terme-*resolve* (c'est-à-dire la représentation d'un modèle contenant des éléments intermédiaires *resolve* sous la forme d'un terme) en terme cible, conforme à la signature cible.

Ce remplacement est possible grâce aux informations supplémentaires qui enrichissent le type cible ainsi qu'aux informations que nous sauvegardons durant la transformation. Ces informations additionnelles sont des informations sur les relations existant entre les éléments cibles et les éléments sources dont ils sont issus. Elles sont obtenues par le biais d'actions explicites de la part de l'utilisateur : tout terme créé dans une transformation peut être tracé sur demande. C'est ce qu'illustre la Figure 4.8 : les jetons colorés correspondent à une action explicite de trace d'un terme qui a été créé dans la *définition*. Ainsi, dans cet exemple, l'utilisateur a marqué un élément dans la *définition* (a) qui correspond à l'élément ciblé par l'*élément resolve* de la définition (b). Il en est de même avec l'élément de type *carré* des *définitions* (b) et (c).

Une autre approche possible eût été de tracer systématiquement tous les termes créés, mais nous avons fait ce choix dans le but d'améliorer la lisibilité de la trace. Toutes ces

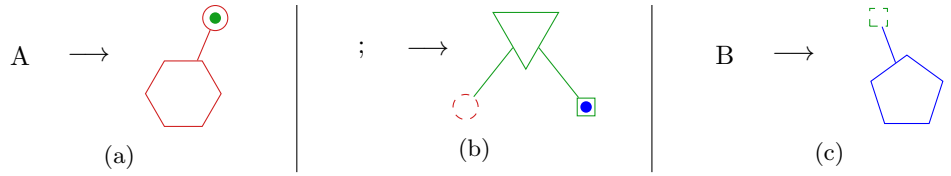


FIGURE 4.8 – Règles de transformation de `A`, `;` et `B` effectives, avec traçage des éléments correspondant à un *élément resolve* d'une autre *définition* (token coloré de la couleur du résultat d'une *définition)*.

informations supplémentaires constituent ce que nous appelons le modèle de lien. Il maintient tout au long de la transformation des relations entre les éléments sources et les éléments cibles qui en sont issus. Dans ce cas précis, le traçage des liens a eu un usage purement mécanique pour permettre la résolution de liens. Cependant, outre cet usage pour la phase de résolution, le construction dédiée au marquage de termes et le modèle de lien nous permettent d'assurer la traçabilité de la transformation à des fins de vérification *a posteriori*. Nous traitons de ce sujet dans le chapitre 5.

Ainsi, une transformation T est la composée de deux fonctions $c : MM_s \rightarrow MM_{t_{resolve}}$ et $r : MM_{t_{resolve}} \rightarrow MM_t$. La transformation complète $T : MM_s \rightarrow MM_t$ est donc définie par $T = r \circ c$. L'encodage d'une telle fonction dans notre approche est une stratégie S mettant en séquence les deux stratégies représentant chaque phase, qui peut se résumer par $S = Seq(S_{phase1}, S_{phase2})$.

Outre les avantages évoqués en début de chapitre, un des intérêts de cette approche compositionnelle reposant sur les stratégies de réécriture apparait immédiatement : reposant sur les stratégies de réécriture, nous bénéficions naturellement de la modularité intrinsèque à ce concept que le langage de stratégies de Tom implémente. Il est ainsi possible de réutiliser les *définitions* dans une autre transformation, sans adaptation lourde du code. On pourrait imaginer que la phase de résolution devienne bloquante dans ce cas, cependant, comme nous le verrons dans la description de l'implémentation de notre approche (chapitre 6), cette phase est générée et peut aussi être générée sous la forme de plusieurs stratégies distinctes — plus simples — réutilisables dans le cadre d'une autre transformation.

4.4 Validation par un cas d'étude

Pour valider la proposition, nous nous sommes appuyés sur une étude de cas : la transformation *SimplePDLToPetriNet*. Nous ne rentrons pour le moment pas dans les détails et n'expliquons pas précisément les métamodèles des formalismes considérés. Nous préciserons la cas dans le chapitre 5 qui suit, puis nous le développerons dans son intégralité dans le chapitre 7 avec son implémentation. Pour résumer cette étude de cas, l'objectif est de transformer des processus génériques décrits dans le formalisme SimplePDL en leur représentation sous la forme de réseaux de Petri. Par exemple, la figure 4.9 décrit un processus simple nommé *root* composé de deux activités : A et B. Ces deux activités sont liées par une contrainte de précédence *startToFinish* (*s2f*). Cela signifie que l'activité A doit avoir commencé pour pouvoir terminer l'activité B.

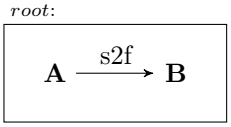


FIGURE 4.9 – Exemple de processus SimplePDL.

Ce processus générique peut s'exprimer sous la forme d'un réseau de Petri tel qu'illustré par la figure 4.10 suivante. Les places sont représentées par des cercles rouges, les transitions par des carrés bleus et les arcs par des flèches. Les flèches en pointillés sont des arcs de synchronisation, celles en trait plein noir sont des arcs normaux créés dans des différentes *définitions*, la flèche verte est un arc obtenu par la *définition* transformant la séquence (qui impose la contrainte de précédence). Dans ce réseau de Petri, lorsque la première transition de P_{root} est franchie, le jeton de la première place est ajouté à la seconde place de P_{root}. Un jeton est aussi ajouté aux premières places des réseaux A et B, ce qui a pour effet de démarrer les tâches. La transition t_{finish} de B ne peut être franchie que si A a démarré.

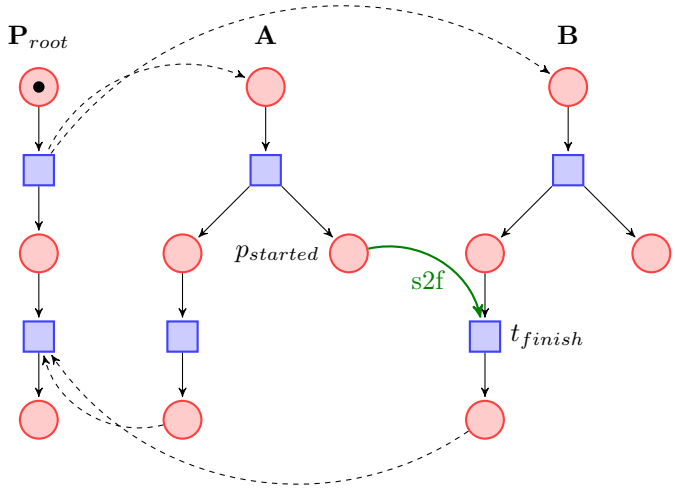


FIGURE 4.10 – Réseau de Petri correspondant au processus décrit par la figure 4.9.

Cette transformation peut être décomposée en trois *définitions*. Chacune d'entre elles produit un réseau de Petri. Nous les représentons toutes les trois dans la figure 4.11 qui suit (les nœuds en pointillés sont des éléments intermédiaires *resolve*).

Dans cette version du cas d'étude, le mécanisme de création d'éléments temporaires *resolve* est utilisé dans le cadre de deux *définitions*, celle qui transforme les *WorkDefinitions* et celle qui transforme les *WorkSequences*. Le mécanisme de marquage est quant à lui utilisé dans la *définition* qui transforme les *Process* afin d'effectuer la correspondance avec les éléments

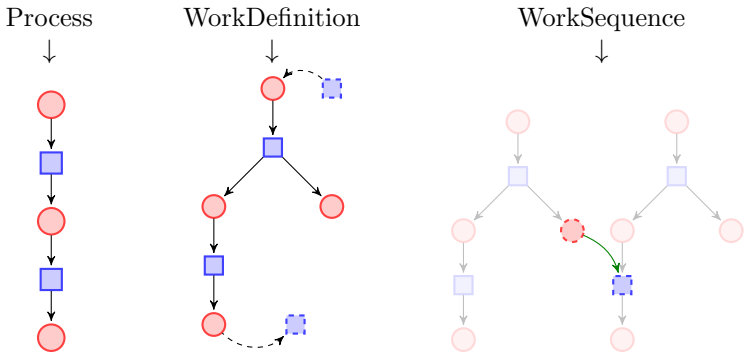


FIGURE 4.11 – Transformations élémentaires composant `SimplePDLToPetriNet`.

resolve créés dans la transformation élémentaire qui prend en entrée une *WorkDefinition*. Des éléments sont aussi tracés dans cette dernière pour assurer la résolution avec les éléments intermédiaires produits dans la *définition* qui transforme les *Worksequences*. Nous avons mis en œuvre ce mécanisme global de résolution que nous détaillons techniquement dans le chapitre 6. Le mécanisme de marquage (ou traçage) est une forme de traçabilité, la traçabilité *interne* (ou *technique*) qui est étroitement liée à l'implémentation de la transformation.

4.5 Synthèse

Dans ce chapitre, nous avons présenté notre approche pour transformer les modèles. Elle est hybride étant donné qu'il ne s'agit pas d'utiliser uniquement un langage généraliste ou un langage dédié, mais d'avoir une approche intermédiaire où nous intégrons des constructions dédiées au sein d'un langage généraliste. Ce procédé est rendu possible par l'utilisation du langage Tom présenté dans le chapitre 1 et qui repose sur le calcul de réécriture. Une telle approche a l'avantage de pouvoir faire bénéficier l'utilisateur du meilleur des deux mondes des langages généralistes et dédiés à la fois. Ainsi, l'utilisateur développant une transformation en Tom+Java aura les constructions spécifiques aux transformations de modèles tout en conservant l'outillage existant de Java.

Nous avons aussi expliqué que notre approche se base sur la réécriture de termes. Compte tenu du fait que nous transformons des modèles, notre approche commence par un changement d'espace technologique rendu possible grâce à un outil de génération d'ancrages formels que nous avons développé. Il nous permet de représenter des modèles Ecore sous la forme de termes, que nous pouvons ensuite parcourir et transformer avec des stratégies de réécriture. Ces stratégies de réécriture encodent les transformations élémentaires composant la transformation globale, elle-même encodée par une stratégie de réécriture. Dans un but d'accessibilité de l'approche et des outils pour les utilisateurs, nous avons choisi de proposer une méthode permettant à l'utilisateur de ne pas avoir à gérer l'ordonnancement des pas d'exécution. Notre solution à ce problème est l'introduction d'éléments intermédiaires dits *resolve* qui jouent le rôle d'éléments cibles tant que ces derniers ne sont pas créés. Une transformation selon notre approche est donc composée de deux phases distinctes : la première où les éléments cibles et cibles intermédiaires sont créés, et la seconde qui consiste à *résoudre* les liens, c'est-à-dire à supprimer les éléments intermédiaires et à remplacer les liens pointant vers eux par des liens vers les éléments cibles qu'ils représentaient.

Chapitre 5

Spécification et traçabilité des transformations

Dans ce chapitre, nous abordons la notion de spécification et de traçabilité d'une transformation de modèles.

Les systèmes se complexifiant, l'IDM a apporté des solutions pour faciliter et accélérer le développement logiciel. Cependant, si la manière de développer un logiciel ainsi que les technologies utilisées ont changé depuis les débuts de l'informatique, les contraintes liées à la maintenance (évolution du logiciel, déboguage) et à la fiabilité (qualification, certification) restent identiques. Ainsi, dans le cadre des transformations qualifiables, nous souhaitons avoir confiance dans les transformations développées, du point de vue formel (pour la vérification) ainsi que de celui de l'autorité de certification. La norme DO-178/ED-12 exige la traçabilité entre le code source et le code objet, c'est donc une problématique essentielle des compilateurs dans le contexte de la qualification. Cette problématique est généralisable à toute transformation, notamment aux transformations de modèles. En effet, l'ingénierie dirigée par les modèles étant de plus en plus présente dans les chaînes de développement de systèmes critiques, du code généré à partir d'un modèle peut faire partie du logiciel final.

5.1 Spécification

Dans le chapitre 3, nous avons vu qu'il fallait qualifier les transformations dans le cadre du développement de systèmes critiques. Il est important de spécifier les transformations pour vérifier leur conformité, afin de disposer de la traçabilité code source–code objet. Pour cela, nous nous appuyons sur les transformations de modèles.

Reprenons le cas d'utilisation *SimplePDLToPetriNet* brièvement illustré dans le chapitre précédent et spécifions les modèles source et destination ci-après.

Le langage SimplePDL dont le métamodèle est donné figure 7.1 permet d'exprimer simplement des processus génériques. Un processus (*Process*) est composé d'éléments (*ProcessElement*). Chaque *ProcessElement* référence son processus *parent* et peut être soit une *WorkDefinition*, soit une *WorkSequence*. Une *WorkDefinition* définit une activité qui doit être effectuée durant le processus (un calcul, une action, *etc.*). Une *WorkSequence* définit quant à elle une relation de dépendance entre deux activités. La deuxième (*successor*) peut être démarrée — ou terminée — uniquement lorsque la première (*predecessor*) est déjà démarrée — ou terminée — selon la valeur de l'attribut *linkType* qui peut donc prendre quatre valeurs : *startToStart*, *finishToStart*, *startToFinish* ou *finishToFinish*. Afin de pouvoir représenter des processus hiérarchiques, une *WorkDefinition* peut elle-même être définie par un processus imbriqué (référence *process*), qui conserve un lien vers l'activité qu'il décrit (référence *from*).

Le métamodèle donné par la figure 5.2 permet d'exprimer les réseaux de Petri. Un tel réseau se définit par un ensemble de nœuds (*Node*) qui sont soit des places de type *Place*,

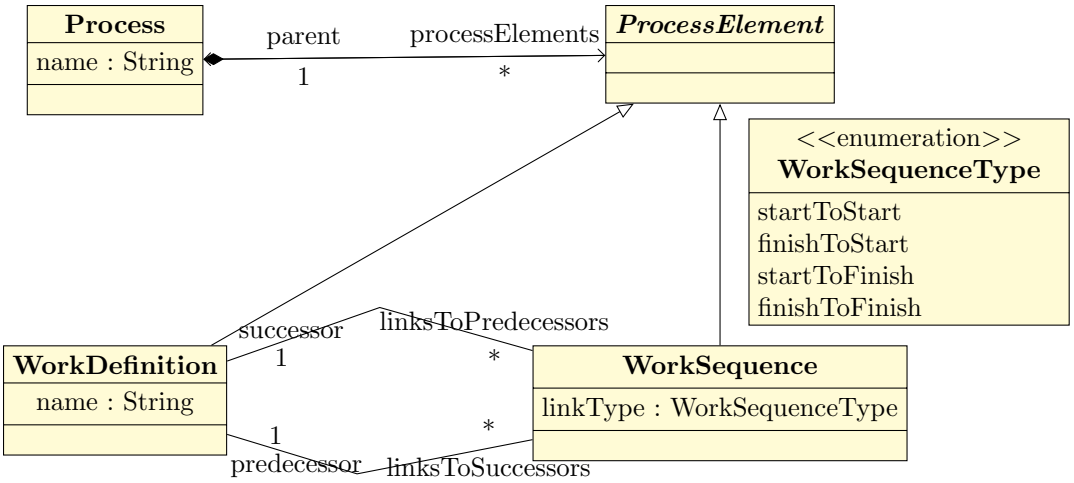


FIGURE 5.1 – Métamodèle SimplePDL possible.

soit des transitions de type *Transition*, ainsi que par des arcs (*Arc*). Un arc (orienté) relie deux nœuds de types différents (le réseau de Petri est un graphe biparti) et peut être de type *normal* ou *read-arc*. Il spécifie le nombre de jetons (*weight* — poids —) consommés dans la place source ou produits dans la place cible lorsqu'une transition est tirée. Un *read-arc* vérifie uniquement la disponibilité des jetons sans pour autant les consommer (test de franchissement). Le marquage d'un réseau de Petri est défini par le nombre de jetons dans chaque place (*marking*).

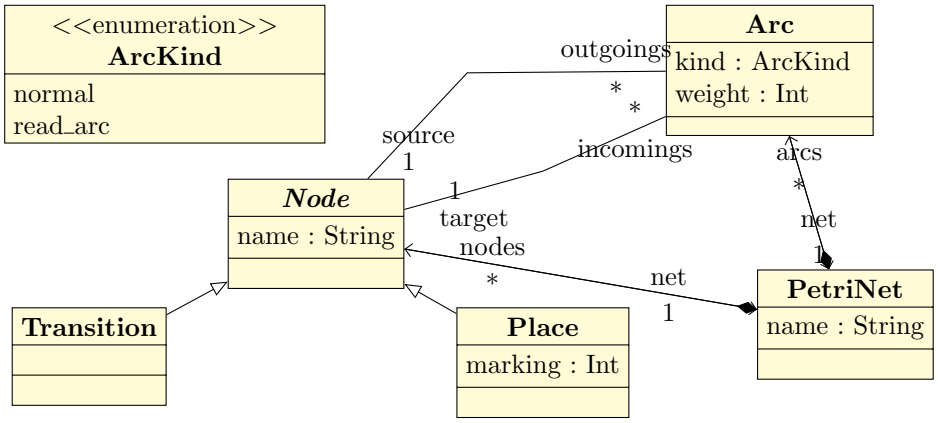


FIGURE 5.2 – Métamodèle des réseaux de Petri.

5.2 Traçabilité

La traçabilité est très importante dans le cadre de la maintenance pour pouvoir suivre l'évolution d'un logiciel et détecter des *bugs* introduits durant le cycle de vie de l'application. C'est un précieux outil pour l'analyse et la vérification de transformations, ce qui explique que ce soit une exigence de qualification. La traçabilité peut prendre plusieurs formes. Nous distinguons notamment la traçabilité *interne* (autrement appelée *technique*) de la traçabilité

de spécification.

5.2.1 Traçabilité interne

La traçabilité *interne* ou *technique* [Jou05] a un usage purement technique au sein de la transformation. C'est-à-dire qu'elle est utilisée pour opérer la transformation, mais n'est pas nécessairement utile dans le cadre d'un processus de qualification. Dans les approches compositionnelles des transformations comme la nôtre, il faut mettre en œuvre un mécanisme permettant d'assembler les résultats partiels. Dans notre cas, si des éléments intermédiaires *resolve* sont créés, il est nécessaire de pouvoir les faire correspondre à des éléments cibles réels. Il faut donc pouvoir marquer les éléments susceptibles de correspondre à un *resolve* donné afin d'assurer la réalisation de la phase de résolution. Nous avons mis en œuvre cette traçabilité dans le cadre de notre approche décrite dans le chapitre 4 et nous la décrivons plus précisément dans le chapitre 6. Nous ne nous attardons donc pas sur cette notion dans ce chapitre.

Du fait de son usage, la traçabilité technique est étroitement liée à l'implémentation de la transformation. Elle est difficilement générique, ce qui peut nécessiter de la réingénierie lors d'une évolution de la transformation.

5.2.2 Traçabilité de spécification

La traçabilité de spécification est quant à elle une exigence de qualification. C'est-à-dire qu'il faut pouvoir lier une source à une cible selon une spécification donnée. Ce type de traçabilité n'a pas d'usage technique pour le bon déroulement de la transformation elle-même et n'est pas nécessairement liée à l'implémentation.

La traçabilité des transformations de modèles étant un élément essentiel dans un cadre industriel, beaucoup de langages l'ont implémenté. Cependant, les langages tels que QVT-Relations et ATL proposent une traçabilité mécanique, mais qui n'est pas substituable à la traçabilité de spécification.

En IDM, l'usage de UML est courant et comme nous avons pu le voir dans le chapitre 2, un métamodèle permet de spécifier un langage. Pour exprimer la traçabilité dans ce contexte, il est donc naturel d'utiliser des métaclasses pour avoir un formalisme cohérent. On peut alors exprimer simplement la traçabilité par le métamodèle 5.3.

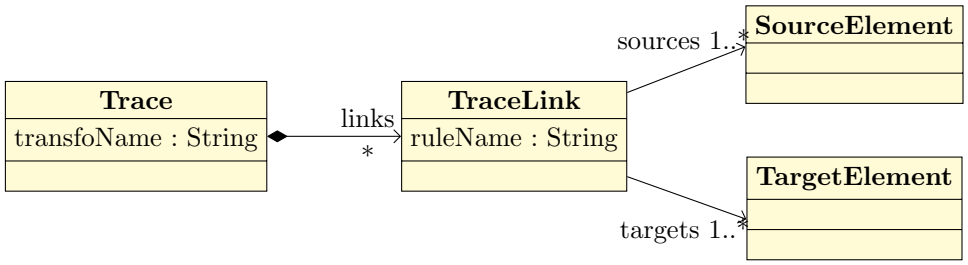


FIGURE 5.3 – Métamodèle générique de trace.

Une *Trace* concerne une transformation donnée nommée. Elle est composée d'un ensemble de relations (ou de liens de trace *TraceLink*) entre des éléments sources et cibles. Un lien de trace établit une relation entre un ou plusieurs éléments sources (*SourceElement*) et un ou plusieurs éléments cibles (*TargetElement*). Une relation de trace est nommée et peut être considérée comme une règle ayant un membre gauche (source) et en membre droit (cible). Ce métamodèle est générique afin de donner une intuition. Il doit cependant être spécifique pour chaque transformation donnée. Dans le cas de *SimplePDLToPetrinet*, *SourceElement* de la figure 5.3 est remplacé par les métaclasses *Process*, *WorkDefinition* et *WorkSequence* ou alors est leur surtype. Dans le cas d'un *framework* comme EMF, on peut utiliser *EObject*.

Lors de la certification et de la qualification, les spécifications sont cependant très souvent écrites en langue naturelle (français ou anglais). Le tâche de l'expert qualifieur consiste alors à comparer la spécification textuelle au code du logiciel et aux sorties produites pour des entrées données. Cette tâche est complexe et fortement sujette à erreurs du fait du rôle central que joue l'humain dans ce processus. Tout doit donc être mis en œuvre pour assister l'expert qualifieur en lui fournissant des éléments de confiance supplémentaires. C'est le rôle des outils et de leurs implémentations de la traçabilité.

Cette trace peut revêtir différents aspects et n'est pas forcément utilisable avec des outils automatiques. Par exemple, l'implémentation de la traçabilité peut consister en la génération de commentaires lisibles par un humain. C'est d'ailleurs l'approche adoptée par l'outil b2llvm — hors du contexte de l'IDM — qui opère une transformation du langage B vers le langage intermédiaire de LLVM [50]. Ces commentaires facilitent la lecture et la compréhension du code par l'expert, mais ne sont cependant pas réutilisables en tant que tels comme entrées d'un outil automatique de vérification. Un point particulièrement intéressant est d'assurer la traçabilité d'une transformation en générant une trace formelle exploitable *a posteriori* par d'autres outils. L'intérêt est alors de permettre de se passer de l'écriture d'une preuve formelle — coûteuse à obtenir — tout en conservant une confiance forte dans la transformation. Une telle trace générée lors d'une transformation peut-être utilisée de deux manières :

— elle peut être comparée à une trace de référence donnée en spécification ;

— des propriétés données en spécification peuvent être vérifiées.

Dans le premier cas d'utilisation, le format de la trace doit être parfaitement spécifié afin que, pour une entrée et une transformation données, la trace puisse être comparée à une trace de référence. Cette dernière fait alors office de spécification de la transformation.

Dans le second cas d'utilisation de la trace, il faut pouvoir exprimer des propriétés à vérifier. Dans le contexte de la modélisation, OCL est utilisé pour décrire des règles s'appliquant sur des modèles UML. Lors d'une transformation de modèle, des pre-conditions peuvent donc être écrites pour les éléments sources, des post-conditions pour les éléments cibles et des invariants pour les liens de traçabilité.

Dans notre approche, nous avons proposé une traçabilité de spécification. La transformation par réécriture présentée étant une transformation par parties, nous agrégeons un ensemble de règles (*définitions*) pour obtenir la transformation finale. Chacune de ces *définitions* est nommée et nous utilisons son nom pour construire le lien de trace lors de la génération du métamodèle de lien (au temps de compilation). Le membre gauche de nos règles constitue la source de chaque lien de trace, tandis que les éléments du membre droit sont les cibles de la relation de trace. Sur action explicite de l'utilisateur, nous établissons une relation un élément source et élément cible. Dans sa forme actuelle, notre implémentation permet de lier plusieurs cibles à une seule source (relations `1..N`), l'objectif étant à terme d'établir des liens entre plusieurs sources et plusieurs cibles (relations `N..N`).

Reprenons le cas de la transformation *SimplePDLToPetriNet* et choisissons une règle comme exemple : *Process2PetriNet*. Cette règle transforme les éléments *Process* du modèle source en un réseau de Petri composé de trois places (p_ready, $p\_running$ et $p\_finished$), deux transitions (t_start et t_finish) et de quatre arcs ($ready2start$, $start2running$, $running2finish$ et $finish2finished$). De cette description textuelle, nous pouvons spécifier la relation liant la source *src* aux éléments cibles comme suit :

```
Process2PetriNet = {
  src : Process ;
  p_ready, p_running, p_finished : Place ;
  t_start, t_finish : Transition ;
  ready2start, start2running, running2finish, finish2finished : Arc
}
```

50. Le projet LLVM est une infrastructure de compilateur modulaire et réutilisable, voir `http://www.llvm.org`.

On peut faire de même avec les relations *WorkDefinition2PetriNet* et *WorkSequence2PetriNet* pour compléter la spécification.

Il est ensuite possible d'exprimer des contraintes sur cette relation, notamment des contraintes de nommage des éléments cibles. Dans notre cas, supposons que l'on impose que le nom d'un élément cible soit composé de deux parties, sur le principe suivant :

— le nom de l'élément source lui ayant donné naissance en préfixe ;

— un suffixe pertinent pour décrire l'élément (par exemple *ready* pour la place *p_ready*).

Avec ces conventions, nous pouvons par exemple écrire la contrainte de nommage pour la place *p_ready* peut être écrite comme suit (`concat()` est l'opération de concaténation des chaînes de caractères dans OCL) :

```
p_ready.Name = src.Name.concat('_').concat(p_ready.Name);
```

La trace est donc exploitable à des fins de qualification.

5.3 Synthèse

Nous avons vu qu'on pouvait distinguer deux types de traçabilité : *interne* et *de spécification*. Nous avons mis en œuvre les deux, la première étant absolument nécessaire à la bonne réalisation de notre approche, la seconde étant utile pour le développement de transformations qualifiables.

Nous avons vu dans le chapitre 3.3 que la traçabilité est généralement classée en deux catégories : *implicite* et *explicite*. Un aspect important de notre approche réside dans notre choix d'une traçabilité *explicite*, ce qui implique l'introduction d'une syntaxe concrète dans le langage pour capturer explicitement les liens de trace. Dans le cas d'un choix implicite, tout doit être tracé et il faut prévoir un mécanisme pour trier et sélectionner les informations pertinentes *a posteriori* (un système de requête sur la trace générée par exemple). Notre choix présente l'intérêt de produire des traces ciblées d'une taille plus raisonnable. Elles sont donc exploitables plus facilement *a posteriori*.

Enfin, un autre point intéressant qui ne peut transparaître sans aborder la question technique tient au fait de notre environnement technologique hybride. En effet, nous apportons une traçabilité au sein d'un langage généraliste par l'ajout d'une construction dédiée. Dans un environnement homogène et parfaitement maîtrisé, la difficulté est levée rapidement, la traçabilité pouvant être assurée relativement simplement. En revanche, dans notre contexte, nous sommes à la frontière de deux mondes que l'utilisateur peut franchir selon ses convenances. Si l'utilisation du langage Tom et la manipulation de termes ne lui conviennent pas, l'utilisateur peut revenir à un mode de programmation en pur Java, auquel cas nous perdons le contrôle de cette partie du code. Ajouter la traçabilité dans cet environnement devient plus complexe. Nous détaillons la mise en œuvre de notre approche dans le chapitre 6.

Finalement, notre approche permet un usage *a posteriori* de la trace générée à des fins de vérification dans le cadre de la qualification.

Chapitre 6

Outils pour exprimer une transformation de modèles en Tom

Dans ce chapitre, nous traitons de notre contribution au langage Tom et de la manière dont notre approche de transformation de modèles par réécriture est implémentée. Dans un premier temps, nous donnons le mode opératoire concret pour utiliser nos outils en nous appuyant sur l'exemple de transformation vu précédemment. Nous présentons ensuite l'extension du langage Tom dédiée aux transformations de modèles, puis nous expliquons son implémentation technique au sein du compilateur, ainsi que l'implémentation du générateur d'ancrages formels.

6.1 Exemple d'utilisation des outils

Cette section présente l'utilisation de nos outils d'un point de vue concret. Nous expliquons le processus d'écriture d'une transformation de modèles avec eux. Pour cela, nous proposons de nous appuyer sur la transformation de texte en formes géométriques colorées présentée dans un chapitre précédent et que nous rappelons dans la section 6.1.1 ci-après. L'objectif de cette transformation est purement pédagogique et sert uniquement à illustrer l'utilisation de nos outils pour ensuite introduire notre extension du langage. Nous ne nous intéressons donc pas à sa pertinence ni à son utilité dans le cadre d'un développement en contexte industriel.

6.1.1 Exemple support

Le principe général de cette transformation de modèle est de transformer un modèle sous forme de texte en une représentation graphique, comme l'illustre la figure 6.1.

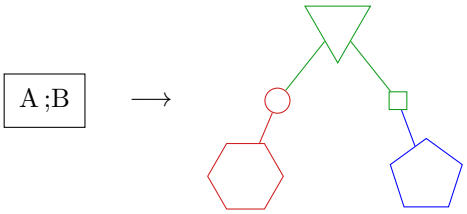


FIGURE 6.1 – Exemple de transformation de texte en formes géométriques colorées.

Les figures 6.2 et 6.3 sont deux métamodèles que nous proposons et auxquels les modèles source et cible de la figure 6.1 se conforment respectivement.

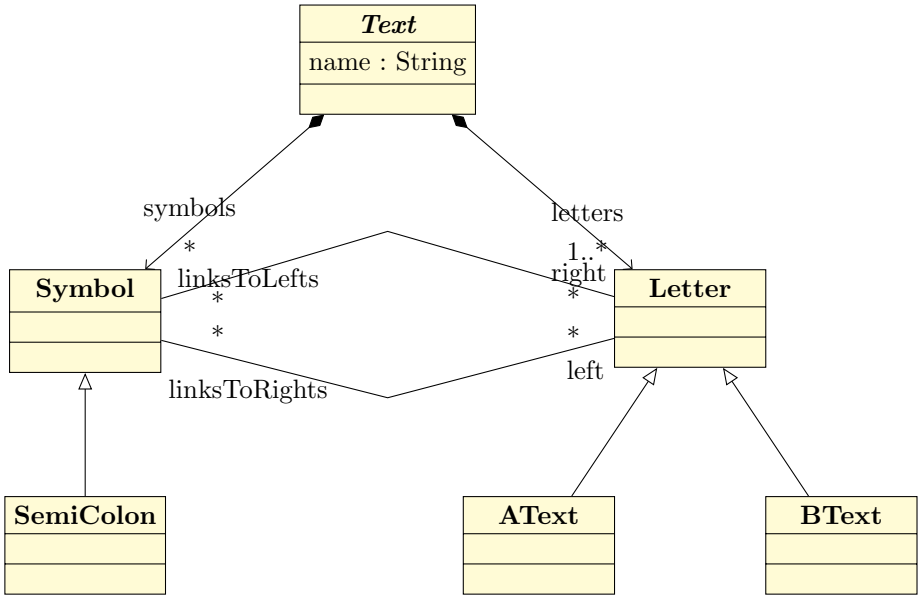


FIGURE 6.2 – Un métamodèle pouvant décrire le formalisme textuel (source) utilisé dans l'exemple support.

Un texte (*Text*) est composé de lettres (*Letter*) et de symboles (*Symbol*). Dans notre exemple, les lettres peuvent être soit des A (*AText*), soit des B (*BText*) ; tandis que les symboles sont des points-virgules (*SemiColon*).

Une image *GeoPicture* est composée de formes (*Shape*) caractérisées par une couleur (*Color*, pouvant prendre les valeurs *red*, *green* et *blue*). *Shape* est abstraite, et plusieurs formes concrètes en héritent (*Triangle*, *Pentagon*, *Hexagon*, *Square*, *Circle*). Les formes géométriques sont reliées entre elles par des segments (*Segmen*).

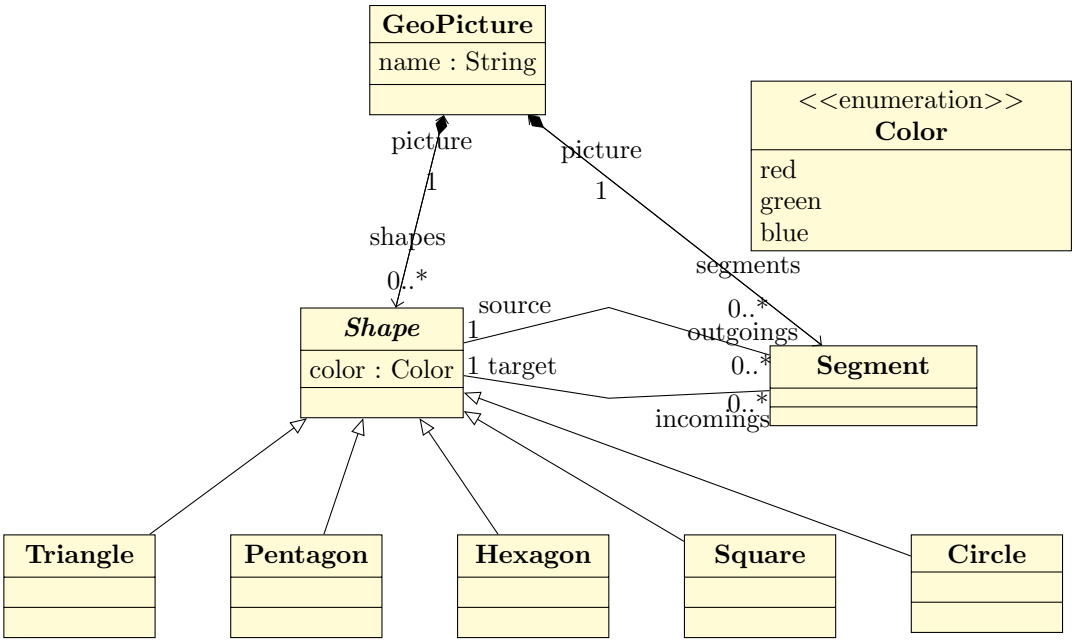


FIGURE 6.3 – Un métamodèle pouvant décrire le formalisme graphique (cible) utilisé dans l'exemple support.

6.1.2 Mode opératoire

Concrètement, un utilisateur souhaitant implémenter une transformation de modèle devra opérer par étapes, obtenues en fonction de l'outil principal à utiliser :

1. Eclipse : modélisation et génération des structures de données Java-EMF ;
2. Tom-EMF : génération des ancrages formels pour représenter les modèles comme des termes (passerelle opérant le changement d'espace technologique) ;
3. Tom+Java : écriture effective de la transformation.

Étape 1 : La première étape consiste à modéliser le problème, c'est-à-dire à définir les métamodèles source et cible, en utilisant Eclipse ou tout autre outil capable de générer un métamodèle au format `.ecore`. Supposons que ces métamodèles source et cible s'appellent respectivement `text.ecore` et `picture.ecore`.

Depuis ces métamodèles, le générateur de code de EMF (*GenModel*) permet de générer les structures de données Java correspondantes qui permettent d'instancier et de manipuler les modèles. L'utilisateur souhaitant implémenter la transformation n'a pas forcément besoin de se plonger dans ce code, étant donné que par la suite il ne manipulera pas directement les modèles en Java +EMF, mais plutôt des termes algébriques. Une fois le code des métamodèles source et cible généré, il suffit de l'exporter sous la forme d'archives `.jar` (par exemple `text.jar` et `picture.jar` dans notre cas). Une fois cette opération accomplie, Eclipse n'est plus nécessaire.

Étape 2 : La seconde étape de la mise en œuvre d'une transformation consiste à générer les ancrages algébriques à partir de la représentation EMF des métamodèles. Cette étape est réalisée en utilisant un outil nommé Tom-EMF. L'utilisateur applique Tom-EMF sur l'archive (ou les archives) en spécifiant le nom complet de l'*EPackage* (ou des *EPackages*) concerné(s).

Concrètement, la commande suivante permet d'effectuer l'opération (nous supposons que les archives `.jar` sont dans le répertoire courant noté `.` et nous ne les nommons donc pas explicitement :

```
$> emf-generate-mappings -cp . text.TextEPackage picture.PictureEPackage
```

Un mécanisme de préfixe a aussi été prévu pour résoudre les conflits de noms. Nous supposons dans notre exemple qu'il n'y en a pas et que l'utilisateur n'a pas à spécifier des préfixes. La commande précédente entraîne la génération de deux fichiers d'ancrages — un par *EPackage* spécifié — nommés respectivement `text.TextEPackage.tom` et `picture.PictureEPackage.tom`. Dans notre exemple, les types et opérateurs ne sont pas préfixés. Si l'utilisateur avait choisi de spécifier un préfixe, chaque *mapping* aurait eu un préfixe (les types primitifs et Ecore sont exclus).

La passerelle permettant d'opérer le changement d'espace technologique étant générée, l'outil Tom-EMF n'est plus utile pour le reste du développement.

Étape 3 : Il s'agit de l'écriture en Tom+Java de la transformation à proprement parler. Cette étape consiste en un développement Java classique intégrant des constructions Tom en son sein, notamment **%transformation** pour spécifier la transformation. Pour pouvoir manipuler les modèles comme des termes algébriques, il est nécessaire d'inclure les fichiers d'ancrages que nous venons de générer avec Tom-EMF. Dans notre exemple, les inclusions (îlot formel `%include`) et la transformation (îlot formel `%transformation`) apparaissent comme illustré dans le listing 6.1.

```
1  ...
2  public class TestText2Picture {
3    ...
4    %include{ text.TextEPackage.tom }
5    %include{ picture.PictureEPackage.tom }
6    ...
7    %transformation Text2Picture(gp:GeoPicture):text.ecore -> picture.ecore {
8      ...
9      definition Letter2Shape traversal ‘TopDown(Letter2Shape(gp)) {
10       AText() -> { ... }
11       BText() -> { ... }
12     }
13     ...
14     definition Symbol2Shape traversal ‘BottomUp(Symbol2Shape(gp)) {
15       SemiColon() -> { ... }
16     }
17     ...
18   }
19   ...
20   public static void main(String[] args) {
21     ...
22     ‘Text2Picture(gp).visit(input_model);
23     ‘TopDown(Resolve(gp)).visit(gp);
24   }
25   ...
26 }
```

Listing 6.1 – Forme générale de la transformation *Text2Picture* écrite en Tom+Java.

Dans ce programme, un modèle peut être soit chargé en utilisant les services EMF (chargement d'un modèle sérialisé dans un fichier `.xmi`), soit créé avec la construction *backquote* de Tom (‘), comme tout autre terme. Quelle que soit la méthode choisie, le modèle peut ensuite être manipulé en tant que terme. À ce titre, il est possible de filtrer des motifs et d'appliquer des stratégies de réécriture (donc des transformations Tom).

Une fois le développement terminé, le programme doit être compilé en deux temps. Il faut procéder à une compilation Tom pour *dissoudre* les îlots formels Tom dans le langage hôte, puis compiler normalement le code Java généré en intégrant les bibliothèques générées par EMF. L'exécution du programme est identique à toute exécution de programme Java.

```
$> tom TestText2Picture.t
$> javac -cp ./text.jar:./picture.jar:${CLASSPATH} TestText2Picture.java
$> java -cp ./text.jar:./picture.jar:${CLASSPATH} TestText2Picture
```

6.2 Extension du langage

Nous avons vu le mode opératoire pour écrire une transformation de modèles avec nos outils dans la section précédente, sans détailler le langage lui-même. Nous décrivons nos constructions dans cette section. Nous avons vu dans le chapitre 4 que nous décomposons les transformations de modèles en transformations élémentaires (*Letter2Shape* par exemple), encodées par des stratégies de réécriture. Nous formons ensuite une stratégie en les composant avec d'autres combinateurs élémentaires tels que `Sequence` ou `TopDown`. Nous avons aussi vu que cela constitue la première phase d'une transformation suivant notre approche et qu'il faut procéder à une seconde phase — *résolution* —, elle aussi encodée par une stratégie de réécriture.

Pour mettre en œuvre la méthode générale de transformation que nous proposons [BCMP12], nous avons ajouté trois constructions principales au langage Tom :

- **%transformation** pour exprimer une transformation (listing 6.2) ;
- **%resolve** pour intégrer et créer les éléments *resolve* (listing 6.3) ;
- **%tracelink** pour assurer la résolution des liens ainsi que la traçabilité de la transformation (listing 6.4).

```
TransformationConstruct ::= '%transformation' TransformationName '('[TransformationArguments]')'
                            ':' FileName '->' FileName '{' (Definition)+ '}'
TransformationArguments ::= SubjectName ':' AlgebraicType ( ',' SubjectName ':' AlgebraicType )*
Definition              ::= 'definition' DefinitionName 'traversal' Strategy '{' (DefinitionRule)+ '}'
DefinitionRule          ::= Pattern '->' '{' BlockList '}'
```

Listing 6.2 – Syntaxe concrète de la construction **%transformation**.

```
ResolveConstruct ::= '%resolve' '(' VarName ':' TypeName ',' VarName ':' TypeName ')'
VarName          ::= Identifier
TypeName         ::= Identifier
```

Listing 6.3 – Syntaxe concrète de la construction **%resolve**.

```
TracelinkConstruct ::= '%tracelink' '(' VarName ':' TypeName ',' BackQuoteTerm ')'
VarName            ::= Identifier
TypeName           ::= Identifier
```

Listing 6.4 – Syntaxe concrète de la construction **%tracelink**.

Ces constructions nous permettent d'implémenter des transformations avec les caractéristiques suivantes, selon des critères proposés par Czarnecki [CH03, CH06] :

— **règles de transformation :** une règle contient des *patterns* et des variables, et bénéficie de typage. L'application des règles peut être contrôlée par le paramétrage des transformations élémentaires que l'utilisateur averti peut faire varier ;

— **ordonnancement des règles :** nos règles ne sont pas ordonnées ni interactives. Par défaut, et si le flot d'exécution n'est pas interrompu (par des instructions du type `break` ou `return`), les règles sont appliquées sur tous les motifs filtrés du modèle source. Ce comportement peut toutefois être modifié par le paramétrage de la *définition* en cours *via* une stratégie *ad-hoc*. L'ordre d'écriture des *définitions* et des règles n'a pas d'impact sur le résultat final ;

— **découpage en phases :** nos transformations sont constituées de deux phases, l'une pour transformer, l'autre pour *résoudre* les liens ;

— **sens des règles et des transformations :** nos règles sont unidirectionnelles, nos transformations ne sont pas naturellement bidirectionnelles ;

— **modularité et mécanismes de réutilisation :** nos transformations reposent sur le langage de stratégies de réécriture de Tom dont l'une des caractéristiques est la modularité. Si une règle ne peut être directement réutilisée ailleurs, une *définition* (un ensemble de règles) peut en revanche l'être ;

— **relation entre l'entrée et la sortie :** le modèle cible est différent du modèle source, même dans le cas d'une transformation endogène (transformation *out-place*) ;

— **portée de la transformation :** tout ou partie du modèle peut être transformé, en fonction des *patterns* dans les règles définies par l'utilisateur ;

— **traçabilité :** durant une transformation, nous maintenons des liens entre les éléments sources et cibles.

La syntaxe des nouvelles constructions du langage ainsi que les caractéristiques des transformations de notre approche étant établies, détaillons précisément ces constructions. Pour illustrer notre propos, nous nous appuierons sur la transformation proposée en exemple précédemment. Cette transformation consiste à transformer le modèle texte `A;B` en une figure constituée de formes géométriques (figure 6.1).

Afin d'illustrer concrètement l'extension du langage, nous nous appuierons sur l'extrait de code donné par le listing 6.5. Les points de suspension dénotent le fait qu'il est parcellaire dans le but de mettre en avant les nouvelles constructions du langage et de rendre plus lisible l'extrait de code ainsi que son explication. Nous adoptons en outre le code visuel suivant : les nouvelles constructions du langage apparaissent en `bleu` tandis que le texte souligné et coloré marque les correspondances dans la transformation entre les éléments des constructions. Ce code visuel permet de comprendre clairement les relations entre les termes créés par les nouvelles constructions du langage.

```
%transformation Text2Picture(link:LinkClass, gp:GeoPicture):
                              "text.ecore" -> "picture.ecore" {

  definition Letter2Shape traversal `TopDown(Letter2Shape(link,gp)) {
    source@BText() -> {
      //use target_right
      Shape greenSquare = %resolve(source:BText,target_right:Square);
      Shape bluePentagon = `Pentagon(blue());
      Segment segment = `Segment(bluePentagon, greenSquare);
      ...
    }
    AText() -> {
      %tracelink(target_left:Circle,`Circle(red())); //define target_left
      Shape redHexagon = `Hexagon(red());
      Segment segment = `Segment(redHexagon, target_left);
      ...
    }
    ...
  }

  definition Symbol2Shape traversal `BottomUp(Symbol2Shape(link,gp)) {
    SemiColon[left=l] -> {
      Shape greenTriangle = `Triangle(green());
      %tracelink(target_right:Square,`Square(green())); //define target_right
      Segment right_segment = `Segment(greenTriangle, target_right);
      Shape redCircle = %resolve(l:AText,target_left:Circle);//use target_left
      Segment left_segment = `Segment(redCircle, greenTriangle);
      ...
    }
    ...
  }
  ...
}
```

Listing 6.5 – Extrait de code de la transformation *Text2Picture* illustrant les nouvelles constructions Tom dédiées aux transformations de modèles et correspondant aux transformations présentées dans la figure 4.8

6.2.1 Expression d'une transformation

Une transformation prend un modèle source en paramètre et renvoie un modèle cible. L'encodant sous la forme d'une stratégie de réécriture composée, nous proposons une construction haut niveau gérant automatiquement sa combinaison. Elle est composée de transformations élémentaires — *définitions* —, représentées par des blocs `definition`. Une *définition* opère une transformation sur des éléments d'un type unique : deux éléments ayant des types différents (et sans surtype commun) ne peuvent être transformés dans une même *définition*. Une transformation comporte donc au moins autant de *définitions* qu'il y a de types d'éléments que l'utilisateur souhaite transformer. Chaque *définition* est constituée d'un ensemble de règles de réécriture composées d'un *pattern* (membre gauche) ainsi que d'une action (membre droit). Une action est un bloc pouvant contenir du code hôte et du code Tom. De plus, chaque *définition* — encodée par une stratégie — est nommée et paramétrée par une stratégie de parcours que l'utilisateur averti peut faire varier.

Chacune de ces *définitions* est écrite sans notion d'ordre par rapport aux autres *définitions* et elles n'entretiennent aucune dépendance dans le sens où l'entrée de l'une n'est pas la sortie d'une autre. La transformation est de type *out-place* (la source n'est jamais modifiée) et la source de chaque *définition* est le modèle source. Le résultat de la transformation ne dépend donc pas de l'ordre d'écriture des *définitions* adopté par l'utilisateur. La transformation finale est encodée par une stratégie qui est une séquence de ces *définitions*.

Dans l'exemple, la transformation nommée `Text2Picture` est introduite par le lexème **%transformation** (ligne 1). Elle sert à transformer un modèle conforme au métamodèle `text.ecore` en un modèle conforme au métamodèle `picture.ecore`, ce qui est symbolisé par les noms de fichiers des deux métamodèles de part et d'autre du lexème `->` (ligne 2). Cette transformation prend deux arguments : `link` qui est le modèle de lien qui sera peuplé durant la transformation, et `gp` qui est le modèle cible qui sera construit au fur et à mesure de l'avancée des pas d'exécution.

Cette transformation est constituée d'au moins deux *définitions* nommées `Letter2Shape` (ligne 4) et `Symbol2Shape` (ligne 20). Elles sont toutes deux paramétrées par une stratégie introduite par le mot-clef `traversal` pouvant être différente pour chaque définition (bien que cela n'ait pas d'impact dans cet exemple, nous avons utilisé deux stratégies différentes — `TopDown` et `BottomUp` — pour illustrer cette possibilité). C'est cette stratégie qui est utilisée pour appliquer les transformations élémentaires. Il est à noter que les paramètres de la transformation sont aussi les paramètres de ces stratégies. Les *définitions* sont constituées de règles de réécriture à l'image des stratégies Tom. Le fait que la *définition* qui traduit les lettres en formes géométriques soit écrite avant ou après celle qui transforme les symboles n'a aucune importance. La *définition* `Letter2Shape` comprend deux règles : l'une pour transformer les éléments de type *BText* (lignes 5 à 10) et l'autre pour transformer les éléments de type *AText* (lignes 11 à 16). La *définition* `Symbol2Shape` est quant à elle constituée d'une seule règle qui transforme les éléments de type *SemiColon* (lignes 21 à 28). Une fois définie, une transformation peut être utilisée (appelée) comme toute autre stratégie Tom *via* la fonction `visit()`.

Seule, la construction **%transformation** permet d'exprimer la première phase de la transformation en générant une stratégie composée.

6.2.2 Résolution

La seconde phase de la transformation (résolution) est exprimée grâce à deux autres constructions.

Pour intégrer et créer des éléments intermédiaires *resolve*, nous avons introduit une nouvelle construction : **%resolve** (syntaxe donnée dans le listing 6.3). Elle permet de créer les termes intermédiaires pour représenter les éléments censés être créés dans une autre *définition*. Cette construction correspond à une spécialisation de la construction *backquote* (‘), en ce sens qu'elle crée un terme tout en déclenchant un autre traitement, à savoir la génération

d'un *mapping* dédié.

La construction **%resolve** prend deux paramètres : l'élément source en cours de transformation et le nom de l'élément de l'image d'un élément transformé dans une autre définition que ce terme est censé représenter.

Cet élément transformé dans une autre *définition* est quant à lui marqué comme pouvant être résolu *via* la construction **%tracelink** (syntaxe donnée dans le listing 6.4). Elle est le pendant de **%resolve** et apparaît obligatoirement dans le code si l'utilisation d'éléments intermédiaires est nécessaire. Elle correspond elle aussi à une spécialisation de la construction *backquote* : elle permet de créer un terme tout en mettant à jour la structure de données qui maintient les liens de traçabilité *interne* (pour la résolution).

La construction **%tracelink** prend deux paramètres : le nom du terme marqué (accompagné de son type) ainsi que le terme lui-même (*terme backquote*).

Dans notre exemple support, la transformation d'un élément B en la forme constituée d'un pentagone bleu et d'un connecteur nécessite l'introduction d'un élément *resolve*. En effet, le connecteur carré est créé dans une autre *définition* — `Symbol2Shape` — et nous devons utiliser le mécanisme de création d'éléments intermédiaires *resolve*. À la ligne 6, nous instancions donc un terme *resolve* qui remplacera temporairement l'élément `target_right` de l'image de la transformation du symbole *SemiColon* (`;`). Outre le fait de jouer le rôle de *placeholder* et de pouvoir être manipulé comme s'il s'agissait d'un élément classique du modèle cible, l'élément *resolve* maintient aussi un lien entre l'élément source (`b`, de type `BText`) qui a provoqué sa création et la cible représentée.

Dans notre exemple, ce mécanisme est réitéré dans la *définition* suivante (ligne 25) et peut être utilisé autant de fois que nécessaire dans la transformation.

6.2.3 Traçabilité

Nous avons également étendu le langage afin d'ajouter la traçabilité aux transformations. Une possibilité eût été de tracer systématiquement les termes créés, puis de mettre en œuvre un mécanisme de requête pour retrouver les informations intéressantes. Cependant, cette approche aurait donnée des traces extrêmement verbeuses et donc peu exploitables. Pour éviter cet écueil, nous avons fait le choix d'une *traçabilité à la demande*, implémentée par la construction **%tracelink** dont la syntaxe a été donnée en début de section dans le listing 6.4.

Bien que techniquement implémentée par un unique lexème, la notion de traçabilité est double. Ce lexème a donc actuellement deux usages — pour deux types de traçabilité — à savoir :

- assurer la traçabilité *interne* (ou *technique*) : pour spécifier les éléments correspondant aux éléments *resolve* créés lors de la transformation, afin que la phase de résolution puisse effectuer le traitement (nous pouvons parler de *resolveLink*) ;
- assurer la traçabilité au sens de la qualification logicielle : construire d'une part le métamodèle de lien à la compilation de la transformation, et d'autre part peupler la trace à l'exécution (nous parlons dans ce cas de *trace*).

Quelle que soit la traçabilité voulue, nous souhaitons spécifier à la demande quels sont les éléments que nous créons que nous souhaitons tracer. L'utilisateur doit donc explicitement désigner les termes Tom à tracer. Chaque terme traçable appartenant à une *définition* de la transformation, l'élément du modèle source dont il est issu est implicite. Du point de vue de l'utilisateur, tracer un terme revient simplement à utiliser la construction dédiée en spécifiant son nom, son type et en écrivant le terme *via* la construction *backquote*.

Dans le cadre d'une traçabilité *interne*, la construction correspond au pendant des éléments *resolve* (*resolveLink*). La création d'éléments *resolve* dans la transformation implique alors l'utilisation d'au moins un marquage de ce type. Cette utilisation est celle présentée dans la section précédente.

Dans le cadre d'une traçabilité de transformation au sens de la qualification logicielle, la construction de trace peut être utilisée de manière indépendante de la construction pour la

résolution. Une transformation peut donc comporter des constructions de trace sans qu'aucune construction de résolution ne soit utilisée. Dans notre approche, l'utilisateur ne fournit pas de métamodèle de lien *a priori*. La construction de trace permet de générer le métamodèle de lien de la transformation lors de la compilation, c'est-à-dire de générer les structures liant sources et cibles en fonction des termes tracés. À l'exécution, ces structures sont peuplées pour maintenir les relations entre les sources et les cibles de la transformation. La construction de trace lie un élément cible créé à la source en cours de transformation, ce qui permet d'établir un ensemble de relations `1..N` (une source associée à plusieurs cibles).

Dans notre exemple support, l'utilisateur a décidé d'opérer une trace minimale, c'est-à-dire que les éléments tracés sont ceux qui doivent impérativement être tracés pour que la résolution s'opère correctement. C'est donc la traçabilité technique qui est essentiellement utilisée ici. Les traces sont activées par les lexèmes **%tracelink** (lignes 12 et 23). Celui de la ligne 23 correspond à l'élément *resolve* de la ligne 6 (souligné en magenta), tandis que celui de la ligne 12 à celui de la ligne 25 (souligné en noir). L'utilisateur aurait bien évidemment pu tracer les autres éléments créés (par exemple `bluePentagon` à ligne 7). Même si l'usage de **%tracelink** dans cet exemple fait transparaître la volonté de l'utilisateur d'avoir uniquement une traçabilité *technique*, un modèle de lien minimal de la transformation est néanmoins créé. La transformation spécifie un modèle de lien (`link` dans notre cas) dans lequel nous stockons les associations établies entre les éléments sources et les éléments cibles tracés tout au long de la transformation. En fin de transformation, cette trace peut être sérialisée pour un usage ultérieur.

En expérimentant nos outils, nous nous sommes aperçus que l'utilisation d'une construction unique pour deux usages distincts pouvait perturber l'utilisateur. Nous projetons donc à terme de séparer cette construction en deux constructions distinctes, chacune adaptée à une des traçabilités.

6.3 Travaux d'implémentation

Les travaux d'implémentation pour étendre le langage ont concerné deux parties du projet Tom : d'une part le compilateur Tom, d'autre part l'outil Tom-EMF.

Avant de détailler la manière dont nous avons mis en œuvre notre extension du projet, nous décrivons l'architecture du projet Tom ainsi que le processus de compilation dans le contexte de Java. Cela nous permettra ensuite d'expliquer comment nous nous sommes intégrés dans l'existant.

6.3.1 Architecture du projet Tom et chaîne de compilation

Le projet Tom s'articule autour de trois outils distincts utilisables indépendamment : le compilateur lui-même, Gom et l'outil de génération d'ancrages formels Tom-EMF.

Le projet Tom comprend aussi un ensemble de bibliothèques (notamment la bibliothèque de stratégies `sl`, des ancrages algébriques, un outil de conversion DOM XML vers Gom et inverse, une bibliothèque de *bytecode* Java, etc.) ainsi que de très nombreux exemples et tests unitaires. La gestion des phases de compilation est assurée par la plateforme sur laquelle s'appuient Tom et Gom.

L'ensemble du projet compte environ 60 000 lignes de code, réparties dans les sous-projets `engine` (compilateur Tom lui-même), `gom`, `library`, `emf`, ainsi que `platform`. L'essentiel du code source est écrit en Tom (+Java) ainsi qu'en Java pur. La Table 6.1 résume cette répartition par projet et par langage.

Sous-projet	Nombre de lignes par langage					Total
	tom	gom	java	ada	python	
engine	20 467	471	2690	-	-	23 628
library	15 062	297	2547	2074	1746	21 726
gom	9572	188	1734	-	-	11 494
emf	1479	-	-	-	-	1479
platform	416	16	602	-	-	1034
Total	46 996	972	7573	2074	1746	59 351
%	79,17%	1,64%	12,76%	3,49%	2,94%	100%

TABLE 6.1 – Nombre de lignes de code dans le projet Tom, par sous-projet et par langage

Le langage Tom a été conçu pour étendre des langages hôtes. La chaîne de compilation (figure 6.5) a été pensée de manière à minimiser le code spécifique à chaque langage hôte. Le compilateur ne traitant que la partie Tom et n'analysant pas le code hôte, seules les constructions Tom sont *parsées* puis traitées par chaque phase du compilateur, jusqu'à la compilation à proprement parler. Durant cette phase, plutôt que de compiler directement vers le langage cible, Tom compile vers un langage intermédiaire (IL) qui sert de langage pivot. Les constructions de ce langage sont ensuite traduites dans le langage cible lors de la dernière phase de compilation. Ainsi, l'extension à un nouveau langage passe par l'écriture d'un nouveau *backend* sans forcément avoir à réécrire ou adapter la chaîne complète du compilateur.

Nos exemples sont centrés sur Java étant donné que son *backend* est le plus avancé. Cependant Tom supporte d'autres langages : Ada, C, Caml, C#, Python. Le tableau 6.2 donne l'implémentation des fonctionnalités en fonction des *backends*.

Langage	Filtrage	Ancrages	Stratégies	Gom	Tom-EMF	%transformation
Ada	✓	✓	✓	×	*en cours*	*en cours*
Java	✓	✓	✓	✓	✓	✓
C	✓	✓	×	×	×	×
Caml	✓	✓	×	×	×	×
C#	✓	✓	×	×	×	×
Python	✓	✓	✓	×	×	×

TABLE 6.2 – Implémentation de fonctionnalités de Tom par langage cible.

Un programme est écrit en Tom+langage hôte, accompagné d'ancrages pour faire le lien avec les structures de données hôtes. Le compilateur analyse les constructions Tom et génère le code hôte correspondant qui, couplé aux structures de données hôtes, constitue un programme écrit dans le langage hôte que l'on compile avec le compilateur adéquat. La figure 6.4 explique ce fonctionnement global des outils du projet Tom dans l'environnement Java. La partie haute décrit le processus de compilation d'un programme Tom couplé à Gom, tandis que la partie basse se concentre sur le processus dans le cadre des transformations de modèles, c'est-à-dire en utilisant l'outil Tom-EMF pour générer les ancrages algébriques. Ces outils peuvent bien évidemment être utilisés indépendamment.

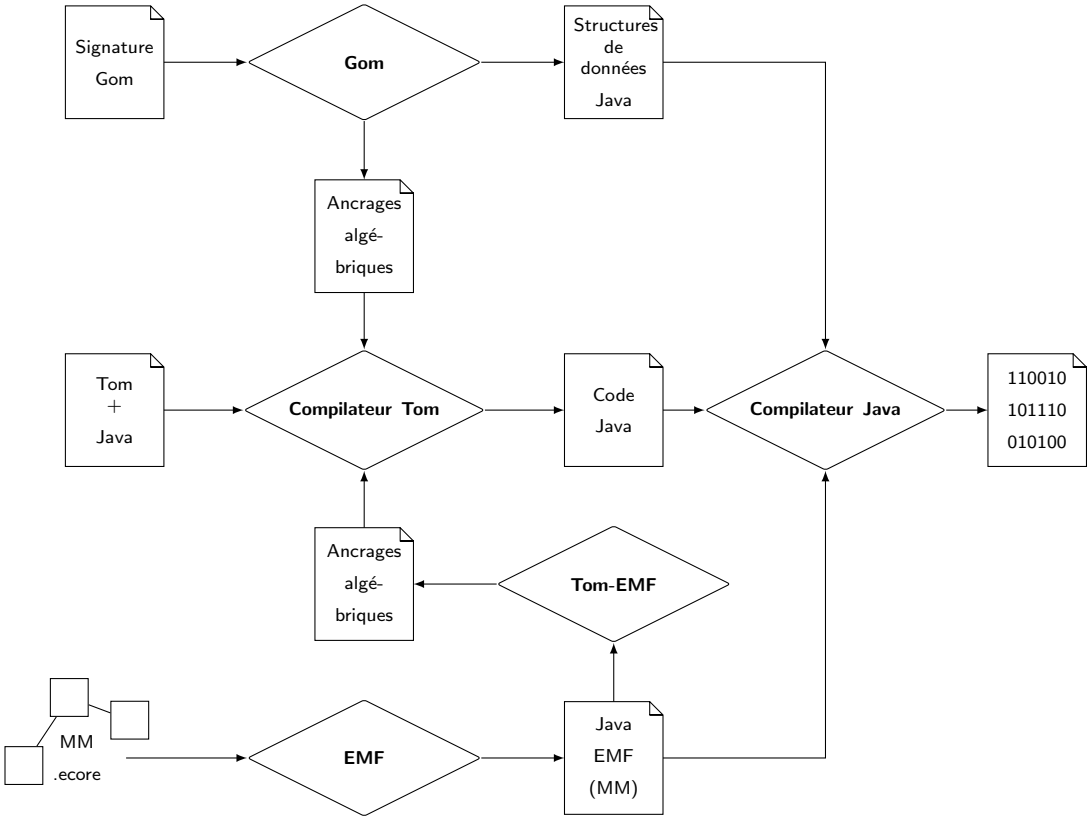


FIGURE 6.4 – Diagramme d'activité décrivant le processus de compilation d'un programme Tom.

Le compilateur Tom se décompose en phases – écrites en Tom+Java – qui sont chaînées les unes après les autres. Chacune procède à une transformation bien définie sur l'entrée, et fournit une sortie à la phase suivante. Nous décrivons brièvement chacune de ces phases ci-après, et nous représentons l'architecture du compilateur Tom avec la figure 6.5. Les blocs marqués en traits pointillés indiquent les phases où nos travaux d'implémentation se sont essentiellement concentrés tandis que la phase colorée était inexistante avant ce travail de thèse. C'est le cœur de l'implémentation permettant de traiter l'extension du langage dédiée aux transformations de modèles.

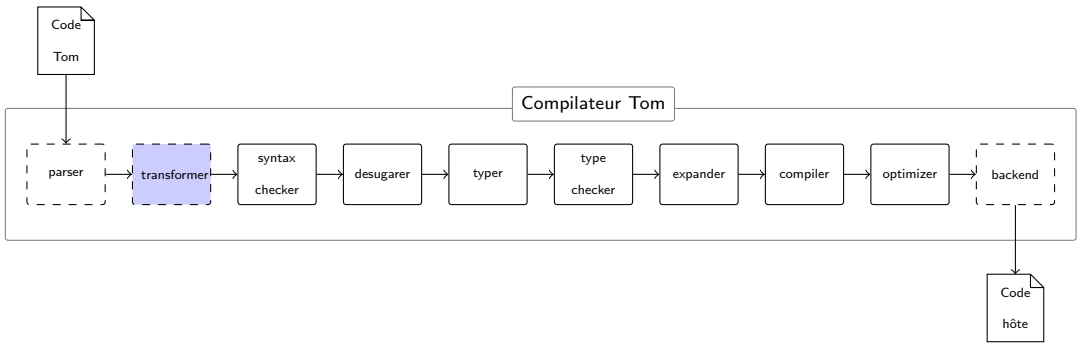


FIGURE 6.5 – Phases du compilateur Tom.

Parser : Le *parser* produit un arbre de syntaxe abstraite (AST, *Abstract-Syntax Tree*) à

partir du code d'entrée. Les constructions Tom sont représentées par des nœuds particuliers tandis que les blocs de code hôte sont stockés dans l'arbre comme des chaînes de caractères. Dans le cas d'une construction **%gom** (ou de l'utilisation d'un fichier `.gom`), l'outil Gom est appelé et son *parser* prend le relais pour l'analyse du bloc.

Transformer : Le *transformer* est une phase qui a été ajoutée durant cette thèse afin de traiter les constructions du langage dédiées aux transformations de modèles. Il traite en particulier les constructions **%transformation**, **%resolve** et **%tracelink**, les AST issus de ces deux dernières étant des sous-arbres de l'AST issu de **%transformation**. La mise en œuvre de cette phase est détaillée dans la section 6.3.3.

Syntax checker : Le *syntax checker* effectue des vérifications syntaxiques sur l'AST. Par exemple, il vérifie que chaque symbole de fonction a été déclaré et qu'il n'y a pas de dépendance circulaire entre les conditions de filtrage.

Desugarer : Le *desugarer* simplifie l'AST en transformant les différents types de nœuds représentant des constructeurs de langage hôte en nœuds génériques. De plus, les variables anonymes y sont nommées avec des noms *frais* (noms qui n'ont jamais été utilisés précédemment).

Typer : Le *typer* effectue l'inférence de type et propage les types inférés dans l'AST.

Typer checker : Le *type checker* vérifie les types et les rangs des symboles. Il vérifie par exemple que les occurrences d'une même variable sont du même type.

Expander : L'*expander* transforme une dernière fois l'AST avant la compilation : il traite les nœuds issus des constructions **%strategy** et génère les introspecteurs (structures pour parcourir un terme).

Compiler : Le *compiler* transforme les nœuds Tom en instructions du langage intermédiaire (IL, pour *Intermediate Language*) qui sert de langage pivot.

Optimizer : L'*optimizer* est responsable de l'optimisation du code écrit dans le langage intermédiaire. Il limite par exemple le nombre d'assignations de variables.

Backend : Le *backend* est le générateur de code cible. Durant cette phase, le langage intermédiaire est traduit en instructions du langage cible.

6.3.2 Générateur d'ancrages algébriques

Dans cette partie, nous décrivons techniquement le générateur d'ancrages formels Tom-EMF. Il a été écrit pour répondre au besoin de représentation des modèles EMF sous la forme de termes Tom. Il est lui-même écrit en Tom+Java et fonctionne de manière complètement indépendante de l'extension du langage dédiée aux transformations de modèles. Il est utilisé de manière *stand-alone*, c'est-à-dire comme un logiciel à part entière, exécuté sans être appelé par un programme Tom externe. La figure 6.6 décrit le principe général de fonctionnement que nous expliquons ci-après :

1. Eclipse : métamodélisation et génération des structures java
   (a) L'utilisateur écrit un métamodèle Ecore manuellement, ou en utilisant les *modeling tools* de Eclipse ;
   (b) Il génère ensuite le générateur de code Java (*GenModel*) à partir de ce métamodèle via Eclipse ;
   (c) Il génère ensuite les structures Java correspondantes et les exporte sous la forme d'une archive (`.jar`) ;
2. Tom-EMF : génération des ancrages algébriques
   (a) L'utilisateur charge les structures Java générées et spécifie à Tom-EMF le(s) *EPackage(s)* pour le(s)quel(s) il souhaite générer des ancrages ;
   (b) Un fichier de *mappings* (`<nom.complet.du.paquet>.tom`) est généré par paquet ;

3. Tom et Java : transformation de modèles
   (a) L'utilisateur inclut les ancrages dans sa transformation via la construction **%include** et importe les structures de données Java (`import`) ;
   (b) L'utilisateur peut alors utiliser les types apparaissant dans le métamodèle et écrire la transformation en Tom+Java, en utilisant ou non l'extension du langage, dédiée aux transformations de modèles ;
   (c) La suite du processus est identique à une utilisation classique de Tom (compilation du code Tom, puis du code Java).

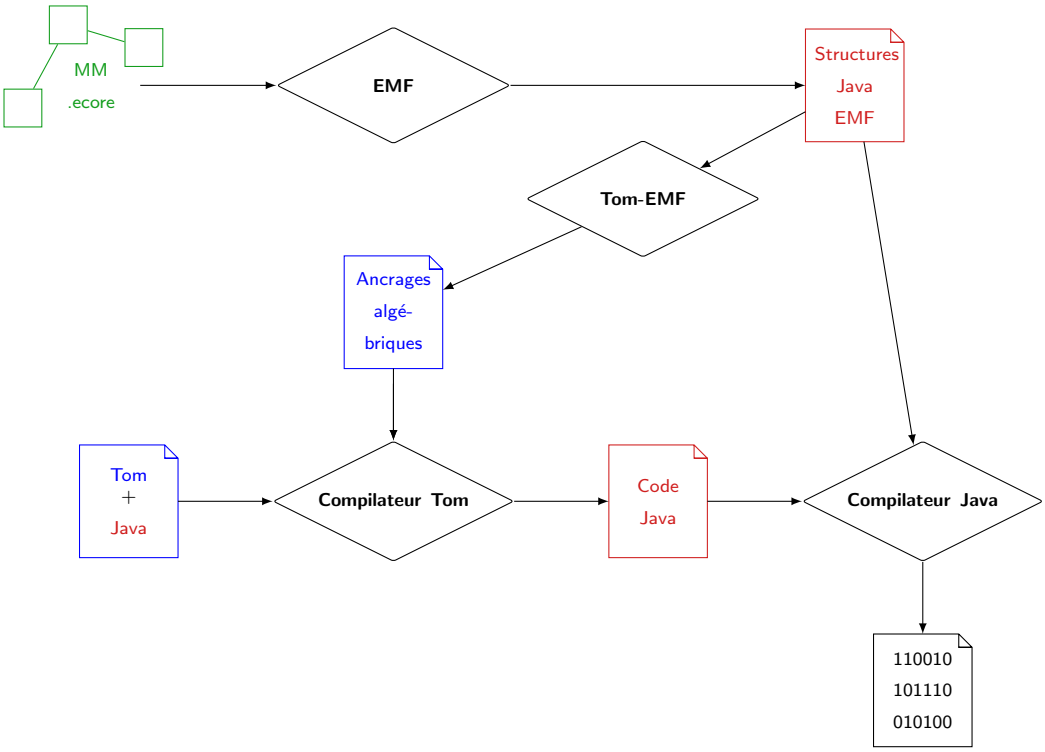


FIGURE 6.6 – Processus de compilation d'une transformation de modèles Tom-EMF.

Ce générateur extrait une signature algébrique utilisable par Tom à partir du code Java-EMF représentant le métamodèle.

Au départ en tant que *proof of concept*, nous avons adapté et étendu le générateur d'ancrages Tom-EMF afin de construire un générateur pleinement opérationnel dédié aux modèles. Compte tenu de la forte utilisation de Java dans le développement logiciel ainsi que de celle de Eclipse et de EMF dans la communauté de l'ingénierie dirigée par les modèles, nous avons fait le choix de nous concentrer dans un premier temps sur ces technologies pour obtenir un outil fonctionnel. Cependant, nous ne nous limitons pas à ces choix et n'excluons pas une ouverture de Tom-EMF à d'autres technologies et langages. Notamment, certaines expériences ont été menées dans le but d'étendre l'outil au langage Ada et au *framework* GMS. Une autre piste d'extension envisagée est l'utilisation de KMF [51]. Le développement d'une transformation en Java-EMF nécessitant de toute manière de générer la structure de données du métamodèle pour l'utiliser au sein du programme, nous avons choisi de conserver dans un premier temps ce fonctionnement de génération de signature algébrique à partir du code. Cependant, si l'outil devait être étendu à d'autres langages et technologies, nous pourrions revoir nos choix afin de générer la signature directement à partir du métamodèle au format `.ecore`.

51. *Kevoree Modeling Framework* : `http://www.kevoree.org/kmf`

Dans sa forme actuelle, Tom-EMF est en mesure de traiter plusieurs paquetages Ecore (*EPackages*) en entrée, et de générer un fichier d'ancrages pour chacun d'entre eux. Si des éléments du métamodèle sont utilisés alors qu'ils appartiennent à un autre *EPackage* du métamodèle (ou d'un autre métamodèle chargé en mémoire), alors notre générateur génère les *mappings* en cascade, tout en évitant la génération multiple d'un même paquetage. De la même manière, cet outil ne génère les ancrages formels que pour les types qui n'en ont pas déjà. Durant l'opération, nous maintenons donc un ensemble d'ancrages générés et à générer pour les types rencontrés.

Pour chaque *EPackage* spécifié par l'utilisateur, Tom-EMF récupère les *EClassifiers* (surclasse commune de *EClass* et *EDataType* permettant de spécifier les types d'opérations, de *structural features* et de paramètres) et génère les ancrages algébriques associés. Si le métamodèle spécifie qu'un élément *EClass* possède la propriété *abstract*, l'opérateur (**%op**) n'est naturellement pas généré. Lorsque la multiplicité d'un attribut est strictement supérieure à 1 (*structural feature* `many`), le champ de l'opérateur est un type liste. Un ancrage supplémentaire est alors généré, accompagné de l'opérateur variadique correspondant. Les associations et compositions sont donc représentées de cette manière. Il est à noter que Tom disposant d'un moteur d'inférence de type équipé du sous-typage [KMT09], nous avons intégré cette fonctionnalité dans Tom-EMF. Lorsque l'option adéquate (`-nt`) est activée, les ancrages générés prennent en compte le sous-typage proposé par Ecore. Si un type n'a pas de surtype explicitement donné, le surtype généré dans l'ancrage est *EObject* (le type *Object* de Ecore) afin de correspondre aux conventions de EMF. Cette fonctionnalité de génération des sous-types est toutefois limitée au sous-typage simple : en effet Ecore supporte l'héritage multiple, mais pas Java (au niveau des classes). Nous verrons dans la section suivante que la fonctionnalité de sous-typage de Tom est utilisée en interne par le mécanisme de résolution (génération et remplacement des éléments *resolve*).

S'agissant des types issus des métamodèles Ecore ou UML (ainsi que les types dits *builtin*), ceux-ci sont traités à part (et non en cascade lorsqu'ils apparaissent) étant donné qu'il s'agit de métamodèles très utilisés et que leurs types sont souvent inclus dans les transformations. De ce fait, nous diffusons les ancrages pour ces métamodèles en tant que bibliothèques dans le projet Tom. L'outil Tom-EMF lui-même est écrit en Tom+Java et utilise ces ancrages Ecore. La première utilisation de notre générateur a donc été pour terminer son *bootstrap*, ce qui nous a permis de remplacer les ancrages écrits manuellement par des ancrages générés, comme l'illustre la figure 6.7. Dans cette figure, les flèches en trait plein sont à comprendre comme des flux d'entrée et sortie, tandis que les flèches en pointillés signifient *intégration à l'outil*.

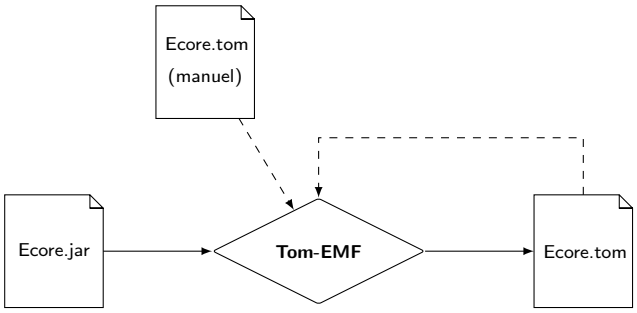


FIGURE 6.7 – *Bootstrap* de Tom-EMF : remplacement des ancrages algébriques `Ecore.tom` écrits manuellement par les ancrages générés.

Tom-EMF gère aussi le préfixage des noms de types et d'opérateurs générés. En effet, il n'est pas rare de nommer de la même manière des éléments de métamodèles différents, mais qui n'appartiennent pas aux mêmes paquetages. Cependant, Tom n'ayant pas de système d'espaces de noms (*namespaces*) pour ses types et opérateurs, nous offrons à l'utilisateur la possibilité de préfixer les ancrages générés.

Enfin, pour rendre complètement opérationnel le pont entre les deux espaces technologiques, Tom-EMF comprend un outil additionnel : `EcoreContainmentIntrospector`. Il s'agit d'une bibliothèque permettant le parcours de modèles par des stratégies. Elle repose sur le fait que le modèle possède une structure arborescente par la relation de composition et définit le parcours de modèles suivant ces relations. Cette bibliothèque s'utilise en conjonction des stratégies (l'utilisateur passe cet *introspecteur* dédié à Ecore en paramètre de la stratégie). Techniquement, cette bibliothèque spécialisée fournit des services nécessaires à Tom pour compter, récupérer et modifier les enfants d'un nœud d'un terme représentant un modèle.

6.3.3 Mise en œuvre de l'extension

Dans cette partie, nous décrivons la mise en œuvre de l'extension du langage au sein du compilateur Tom.

Lors de l'implémentation de constructions d'un langage, plusieurs modifications de la chaîne de compilation sont nécessaires, de l'analyseur syntaxique au générateur de code. Dans notre cas, s'agissant de nouvelles constructions avec de nouveaux lexèmes, il fallait étendre le *parser* (début de chaîne) pour pouvoir les reconnaître. Ensuite, ces constructions entraînant la génération de code qui n'était pas encore traité par le compilateur, le *backend* (fin de chaîne) a aussi été étendu. Cependant, ces deux tâches ne sont pas le cœur de la mise en œuvre de l'extension du langage dédiée aux transformations de modèles. En effet, l'essentiel de l'implémentation réside dans l'ajout d'une nouvelle phase — le *transformer* — dont la charge est de transformer les nœuds de l'arbre spécialisé dans la transformation de modèles en des stratégies. À cela s'ajoute la génération automatique d'ancrages ainsi que celle de la trace dont les instructions sont ajoutées à l'arbre.

Parser. L'aspect inhabituel de l'analyseur syntaxique de Tom réside dans le fait qu'il n'analyse pas tout le code, mais uniquement les îlots formels. Le code hôte est quant à lui vu comme une chaîne de caractères qui est parcourue jusqu'à l'îlot suivant. Ce type d'analyse où la grammaire du langage n'est pas totalement définie est appelé *fuzzy parsing* : seule la grammaire de Tom l'est, totalement (on parle de grammaire îlot ou *island grammar*). L'analyseur syntaxique de Tom repose sur le très populaire générateur de *parser* ANTLR[52], ainsi que sur GomAntlrAdapter, un outil du projet permettant de lier les arbres générés par ANTLR aux structures Gom. Lorsque le *parser* principal détecte une construction donnée, il donne la main à un *parser* dédié (Tom ou Gom). Il s'agissait donc d'étendre ces analyseurs pour mettre en œuvre les nouvelles constructions, en ajoutant de nouvelles règles adéquates dans la grammaire. En parallèle, la signature de Tom a été étendue afin de prendre en compte ces nouvelles constructions.

Transformer. Le cœur de la mise en œuvre de l'extension de Tom dédiée aux transformations de modèles se situe au niveau du *plugin transformer*, juste après l'analyseur syntaxique, et avant la phase de typage. Ce choix de nous insérer tôt dans la chaîne de compilation est tout à fait logique : la seule contrainte forte que nous avions était d'apparaître avant l'*expander* — *plugin* dédié aux stratégies — et nous souhaitions aussi bénéficier au maximum des éléments de contrôle déjà implémentés dans le compilateur Tom (*syntax checker*, *typer*, *type checker*).

Cette phase prend en entrée l'arbre issu de l'analyseur syntaxique et produit un arbre syntaxique ne contenant plus aucun sous-arbre de type *Transformation* obtenu après *parsing* de la construction **%transformation**. Elle est implémentée de manière classique — du point de vue d'un développeur Tom —, à savoir qu'une stratégie Tom est appliquée sur l'arbre en entrée.

Les nœuds de type *Resolve* et *Tracelink* obtenus à partir du *parsing* des constructions **%resolve** et **%tracelink** sont forcément des sous-arbres des arbres dont la racine est un nœud *Transformation* étant donné que ces constructions sont uniquement utilisables dans le

52. *ANother Tool for Language Recognition* : `http://www.antlr.org`

cadre des transformations de modèles. En ne manipulant que ces derniers, nous sommes donc en mesure de collecter et traiter tous les nœuds *Resolve* et *Tracelink*.

Dans le sous-arbre représentant une transformation, nous filtrons les nœuds de type *Resolve* correspondant aux utilisations de la construction **%resolve**. Si un tel motif est filtré, le mécanisme d'enrichissement du métamodèle cible présenté dans le chapitre 4 est déclenché. Il est alors nécessaire de créer d'une part les structures de données concrètes représentant les éléments *resolve*, et d'autre part des ancrages algébriques correspondants. Pour illustrer concrètement ce mécanisme, reprenons l'instruction `%resolve(l:AText,target_left:Circle);` de l'exemple précédent transformant du texte en figure géométrique. Elle entraîne la génération de la structure Java par le *backend*, donnée dans le listing 6.6 (par souci de lisibilité, les noms ont été simplifiés et les attributs sont publics).

```
1  private static class ResolveATextCircle extends Circle {
2      public String name;
3      public AText o;
4
5      public ResolveATextCircle(Circle o, String name) {
6        this.name = name;
7        this.o = o;
8      }
9    }
```

Listing 6.6 – Exemple de classe Java générée implémentant un élément *resolve*.

La classe générée — ici `ResolveATextCircle` — étend naturellement la classe de l'élément cible — `Circle` dans notre cas — que l'élément *resolve* est censé représenter. Cette classe est associée à un ancrage et la relation d'héritage de Java est exprimée par un sous-type dans le *mapping* décrit dans le listing 6.7. Étant donné que ce mécanisme a lieu durant le processus de compilation Tom->Java, cet ancrage n'apparaît pas en tant que tel, mais est directement généré en mémoire pour être traduit en Java par la suite. Le listing 6.7 est néanmoins une traduction fidèle du *mapping* qui serait écrit en Tom.

```
1   %typeterm ResolveATextCircle extends Circle {
2     implement { ResolveWorkATextCircle }
3     is_sort(t) { t instanceof ResolveATextCircle }
4   }
5
6   %op Circle ResolveATextCircle(o:AText,name:String) {
7     is_fsym(t) { t instanceof ResolveATextCircle }
8     get_slot(name, t) { t.name }
9     get_slot(o, t) { t.o }
10    make(o,name) { new ResolveATextCircle(o,name) }
11  }
```

Listing 6.7 – Exemple d'ancrage algébrique généré pour un élément *resolve*.

C'est aussi dans le *transformer* que nous filtrons tous les symboles de type *Tracelink*. Pour chaque instruction de traçage, le *transformer* crée (ou récupère si elle existe déjà) ce que nous appelons une `ReferenceClass`. Il s'agit d'une structure de données permettant de référencer les éléments cibles tracés.

En termes d'implémentation Java, nous fournissons une interface `ReferenceClass` extrêmement simple (listing 6.8) que les structures de données générées implémentent.

```
package tom.library.utils;

public interface ReferenceClass {
  public Object get(String name);
}
```

Listing 6.8 – Interface devant être implémentée par les structures de type `ReferenceClass`.

Ces classes sont essentiellement constituées d'attributs (les références vers les éléments cibles), de leurs méthodes d'accès (`get` et `set`) ainsi que d'une méthode permettant d'associer un nom à un élément. Le listing 6.9 montre une classe qui serait générée dans le cas de notre exemple de transformation de texte en formes géométriques.

```
public static class tom__reference_class_Letter2Shape
                                implements tom.library.utils.ReferenceClass {
  private Circle target_left;
  public Circle gettarget_left() { return target_left; }
  public void settarget_left(Circle value) { this.target_left = value; }

  public Object get(String name) {
    if(name.equals("target_left")) {
      return gettarget_left();
    } else {
      throw new RuntimeException("This field does not exist:" + name);
    }
  }
}
```

Listing 6.9 – Exemple de classe générée implémentant l'interface `ReferenceClass` dans le cas de la transformation *Text2Picture*.

Une structure de type `ReferenceClass` est elle-même liée à l'élément source filtré par la règle qui lui a donné naissance. Techniquement, cela consiste à maintenir une table de hachage dont les clefs sont des sources de type `EObject` et dont les valeurs sont ces structures de données. Cette table de hachage fait partie intégrante du modèle de lien de la transformation, implémenté par la classe `LinkClass` (fournie en tant que bibliothèque avec son ancrage associé).

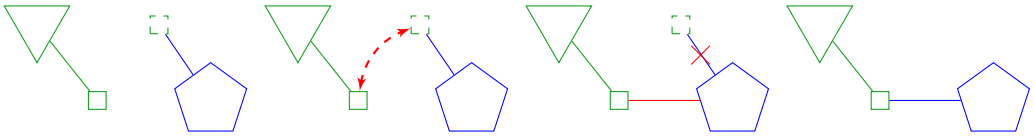
Toute *définition* d'un sous-arbre *Transformation* est quant à elle transformée en une stratégie comprenant les règles de la *définition*, accompagnée de son symbole. Une stratégie plus complexe est ensuite composée à partir de ces stratégies. Finalement, tout nœud de type *Transformation* est remplacé par sa stratégie nouvellement créée ainsi que par une éventuelle stratégie de résolution.

Backend. En fin de chaîne de compilation, au niveau du générateur de code (*backend*), le travail a consisté à ajouter des fonctions de génération de la phase de résolution ainsi que celles pour le métamodèle de lien qui n'existaient pas auparavant. Ces dernières produisent du code similaire à celui que nous avons montré précédemment dans les listings 6.8 et 6.9, dans la partie décrivant la mécanique du greffon *transformer*.

La mise en œuvre de la génération de la phase de résolution a fait l'objet de nombreuses évolutions au cours de ce travail. Nous décrirons son état stable actuel (inclus dans la dernière version stable publiée : Tom-2.10) ainsi que celui de la prochaine version stable.

La phase de résolution étant encodée sous la forme d'une stratégie de réécriture, sa génération est en partie identique à la génération de stratégies classiques. Cependant, elle se distingue d'elles par le fait qu'il faille aussi écrire une procédure supplémentaire permettant d'effectuer la *résolution de liens inverses*. Cette procédure consiste à notifier tous les objets du modèle ayant un pointeur vers l'objet qui vient d'être modifié pour que ce pointeur pointe vers le nouvel objet et non plus vers l'élément *resolve*.

La figure 6.8 illustre ce mécanisme. La figure 6.8a est l'état du modèle en fin de première phase. Il est constitué d'éléments cibles ainsi que d'éléments *resolve* (en pointillés). Les éléments à résoudre sont détectés et l'élément cible effectif est identifié (figure 6.8b). Tous les éléments faisant référence à cet élément *resolve* doivent aussi être mis à jour. Dans l'exemple, le connecteur entre le pentagone bleu et le carré vert référence le carré vert temporaire. Son extrémité (un pointeur vers l'élément *resolve*) est mise à jour pour référencer l'élément cible final (figure 6.8c). Une fois la *résolution de liens inverses* effectuée, le modèle ne comprend plus d'élément *resolve* (6.8d).



(a) (b) (c) (d)

FIGURE 6.8 – Processus de résolution de liens inverses.

Dans le contexte de EMF, ce traitement peut être fait en utilisant les services fournis. Dans la stratégie de résolution, nous effectuons donc une résolution de liens inverses pour chaque élément *resolve* trouvé et remplacé. La procédure de résolution de liens inverses peut être automatisée ; mais dépendant des types des éléments du modèle, elle ne peut être générique et doit donc être générée pour chaque transformation. Pour rendre possible la génération de ce code additionnel dans les stratégies, il a été nécessaire de modifier la signature des stratégies afin de pouvoir embarquer du code supplémentaire, ainsi que l'*expander*. Le *backend* des stratégies a ensuite été adapté afin de prendre en compte cette nouveauté.

Signalons que la phase de résolution n'apparaît pas dans l'extrait de code présenté plus tôt dans le chapitre. Il ne s'agit pas d'une simplification du code pour les besoins de notre propos, mais bien du fonctionnement normal de nos outils. En effet, la phase de résolution est entièrement générée, en fonction de l'utilisation des constructions **%resolve** au sein de la transformation. Ainsi, une transformation ne nécessitant pas d'élément *resolve* (donc sans utilisation du lexème **%resolve**) n'a pas de phase de résolution. À l'inverse, l'utilisation de la construction provoque la génération du bloc de résolution dédié au type traité. Dans la version actuellement publiée de l'outil (Tom-2.10), cette phase est encodée par une unique stratégie que l'utilisateur n'écrit qu'indirectement par l'usage judicieux des constructions de résolution. L'application de la stratégie de résolution est néanmoins à la charge de l'utilisateur au sein de son programme. Il doit appeler la stratégie en l'appliquant sur le modèle intermédiaire créé lors de la première phase.

Par la suite, nous avons expérimenté notre langage et avons constaté que l'utilisation intensive des services EMF avait des conséquences importantes sur les performances de la transformation, tant du point de vue de la mémoire consommée que du temps. Nous avons donc fait évoluer la phase de résolution en nous passant des services EMF pour la résolution de liens. Dans un premier temps, plutôt que d'utiliser les méthodes EMF, nous avons écrit notre propre méthode de résolution en évitant de parcourir tous les objets du modèle comme le fait EMF. Dans un second temps, nous avons écrit une phase de résolution sans stratégie afin de ne plus du tout parcourir le modèle intermédiaire. Pour réaliser cette optimisation, nous conservons tout au long de la transformation un ensemble de références inverses pour chaque élément *resolve* créé (les seuls éléments du modèle intermédiaire concernés par la résolution

de liens inverses). Nous détaillerons l'aspect performances de nos outils dans le chapitre 8.

Nous avons aussi expérimenté une version modulaire de la phase de résolution. Plutôt que de générer une stratégie monolithique effectuant toute la résolution, nous avons écrit une solution alternative générant plusieurs stratégies de résolution partielle, chacune de ces stratégies n'effectuant la réconciliation que sur un type de terme donné. Dans notre exemple, cette alternative générerait deux stratégies distinctes, ainsi qu'une stratégie combinant les deux. L'inconvénient de cette solution est qu'elle est plus complexe à prendre en main pour l'utilisateur. Cependant, elle a l'avantage d'offrir une plus grande modularité en décomposant la seconde phase. Ainsi, le code généré pour une *définition* peut être réutilisé dans une autre transformation, et l'utilisateur averti prendra soin d'accompagner la définition de la (ou des) stratégie(s) de résolution correspondante(s). Pour des raisons d'utilisabilité de nos outils, nous avons fait le choix de diffuser la version du générateur de phase de résolution monolithique dans la version stable de Tom.

6.4 Synthèse

Dans ce chapitre, nous avons d'abord montré une utilisation concrète de nos outils appliquée sur un exemple simple. Nous avons présenté l'extension du langage Tom permettant de mettre en œuvre notre approche de transformation. Les trois aspects principaux de cette extension sont : l'expression de la transformation elle-même, le mécanisme de résolution et d'extension du métamodèle cible par des éléments temporaires, ainsi que la traçabilité. Ces éléments sont respectivement mis en œuvre par les constructions **%transformation**, **%resolve** et **%tracelink**.

Nous avons par ailleurs présenté nos travaux d'implémentation ainsi que leur intégration au sein du projet Tom. Nous avons notamment décrit le générateur d'ancrages algébriques Tom-EMF permettant d'opérer le changement d'espace technologique *modèles* $\rightarrow$ *termes*. Nous avons de plus expliqué comment nous analysions et transformions les sous-arbres issus des nouvelles constructions du langage Tom dans la phase *transformer*, et quel était le code généré par le *backend*.

Nous avons expérimenté nos outils et les premiers retours nous donnent des pistes de travail intéressantes sur l'usage des technologies externes sur lesquelles le prototype repose, sur le concept de modularité d'une transformation (et donc aussi de la résolution), mais aussi sur la conception du langage lui-même (séparation explicite des traçabilités). Nous reparlerons de ces expériences et des perspectives offertes dans la suite de ce document.

Tout au long de ce chapitre, nous nous sommes appuyés sur l'exemple simple d'une transformation de texte en formes géométriques colorées pour expliquer les mécanismes en jeu. Nous nous proposons d'étudier une transformation complète dans le chapitre suivant.

Chapitre 7

Études de cas : illustration et utilisation du langage

Dans ce chapitre, nous présentons un cas d'étude et expliquons le processus complet ainsi que son implémentation, depuis les métamodèles jusqu'au code de la transformation.

Dans la section 7.1, nous présentons le cas d'étude *SimplePDLToPetriNet* tandis que dans la section 7.2 nous présentons le cas de la transformation de l'aplatissement d'une hiérarchie de classes.

7.1 Cas *SimplePDLToPetriNet*

Dans cette section, nous présentons la transformation *SimplePDLToPetriNet* introduite dans les chapitres précédents. Elle a été présentée et traitée par Benoît Combemale [Com08]. Son principe est de transformer la représentation d'un processus exprimé avec le langage SimplePDL en sa représentation exprimée dans le formalisme des réseaux de Petri. L'intérêt de cette transformation dans la communauté est de travailler sur la vérification : dans le formalisme SimplePDL, il n'est pas possible de vérifier directement le processus décrit tandis que sous forme de réseau de Petri — qui est un modèle mathématique —, il est tout à fait possible de l'utiliser avec un *model-checker* pour en vérifier des propriétés.

Nous rappelons d'abord les métamodèles permettant d'exprimer un processus que nous avons déjà présentés et expliqués précédemment, puis nous donnons un exemple de processus décrit en SimplePDL ainsi que sa version sous forme de réseau de Petri.

7.1.1 Métamodèles

Métamodèle source : formalisme SimplePDL

Nous reprenons le métamodèle 5.1 que nous complétons afin de pouvoir représenter des processus hiérarchiques. Nous ajoutons deux relations *opposite* entre les métaclasses *WorkDefinition* et *Process*, les autres éléments restant identiques. Une *WorkDefinition* peut ainsi être elle-même définie par un processus imbriqué (référence *process*), qui conserve un lien vers l'activité qu'il décrit (référence *from*). Nous obtenons le métamodèle illustré par la figure 7.1.

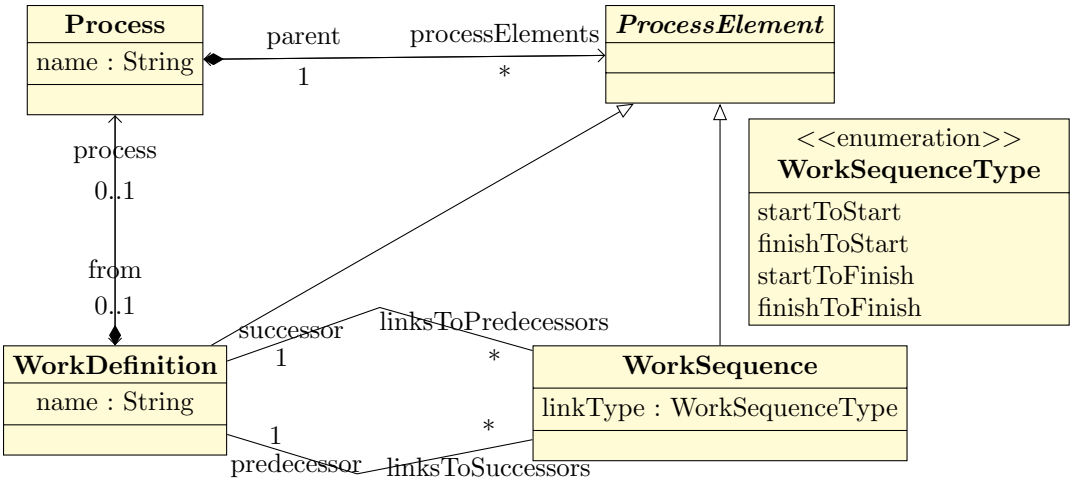


FIGURE 7.1 – Métamodèle SimplePDL.

Métamodèle cible : formalisme des réseaux de Petri

Nous reprenons le métamodèle des réseaux de Petri proposé dans le chapitre 5, sans aucune modification additionnelle. La figure 7.2 est un rappel du métamodèle des réseaux de Petri que nous utilisons.

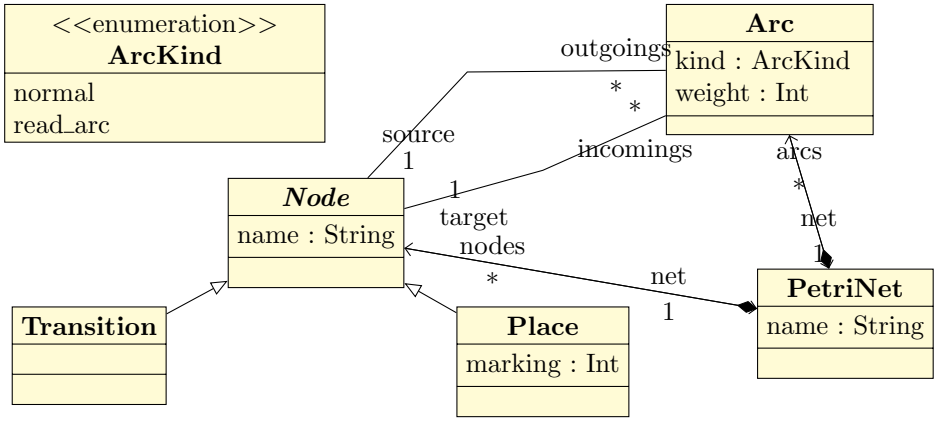


FIGURE 7.2 – Métamodèle des réseaux de Petri.

7.1.2 Exemple de processus et de réseau de Petri résultant

Nous décidons de traiter l'instance de SimplePDL suivante : le processus hiérarchique composé d'activités et de contraintes de précédence illustré par la figure 7.3.

Dans cet exemple, le processus `root` est composé de deux activités, `A` et `B`, reliées par une séquence *start2start* notée *s2s*, ce qui signifie que `B` peut démarrer uniquement si `A` a déjà démarré. `B` est elle-même décrite par un processus (`child`) composé de deux activités, `C` et `D` reliées par une séquence *finish2start* notée *f2s*. Ainsi, `C` doit être terminée pour que `D` puisse démarrer. Ce processus est conforme au métamodèle SimplePDL donné par la figure 7.1.

Notre but est de transformer sa représentation actuelle en sa représentation sous forme d'un réseau de Petri. La figure 7.4 est le résultat attendu pour cette transformation.

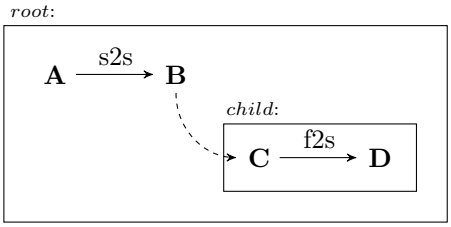


FIGURE 7.3 – Exemple de processus décrit dans le formalisme SimplePDL.

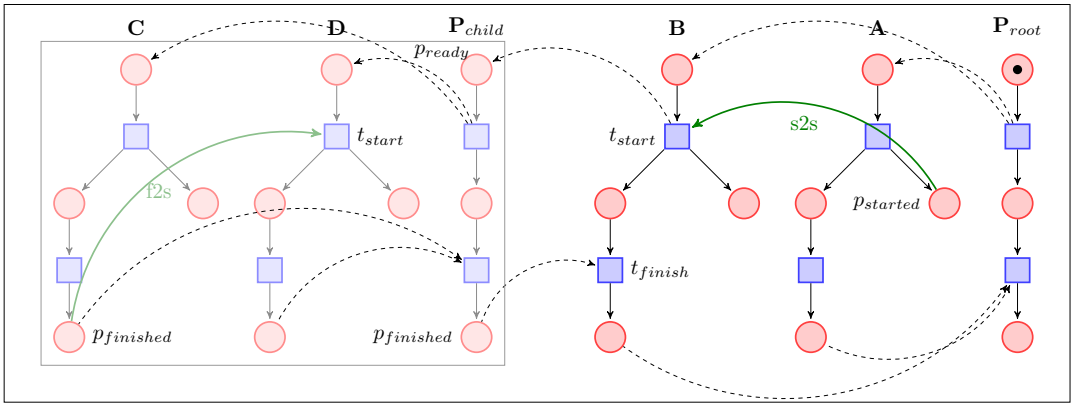


FIGURE 7.4 – Réseau de Petri équivalent au processus décrit par la figure 7.3.

Dans cette figure ainsi que dans la suite du document, nous représentons les places par des cercles rouges, et les transitions par des carrés bleus. Les arcs de type *normal* sont matérialisés par des flèches en trait plein noires, ou vertes dans le cas du résultat de la transformation d'une *WorkSequence*. Ceux en pointillés correspondent aux synchronisations entre éléments, c'est-à-dire aux arcs de type *read_arc*.

7.1.3 Implémentation en utilisant les outils développés

Les métamodèles ainsi qu'un exemple de modèle d'entrée et son résultat attendu ayant été présentés, détaillons la transformation, ainsi que sa mise en œuvre avec nos outils. Pour améliorer la lisibilité — et donc la compréhension —, les extraits de code apparaissant dans cette section sont légèrement simplifiés par rapport à l'implémentation réelle qui est donnée en annexe A.1. Nous avons notamment supprimé certains paramètres et modifié des noms de variables (ajouts de préfixes P et WD par exemple) afin d'extraire l'essentiel du code en tâchant d'éviter toute confusion au lecteur. Nous avons aussi conservé une cohérence entre les schémas et les extraits de code.

Pour transformer le processus décrit par la figure 7.3, on peut aisément isoler trois transformations élémentaires qui composent la transformation globale. Chacune d'entre elles transforme un type d'élément du modèle source : respectivement *Process2PetriNet*, *WorkDefinition2PetriNet* et *WorkSequence2PetriNet* pour les éléments *Process*, *WorkDefinition* et *WorkSequence*. Ces transformations élémentaires sont implémentées par des sous-constructions `definition`.

ProcessToPetriNet. Un *Process* SimplePDL est traduit par un réseau de Petri de la forme de celui donné par la figure 7.5. Cette transformation élémentaire est implémentée par la

définition `P2PN` donnée par le listing 7.1.

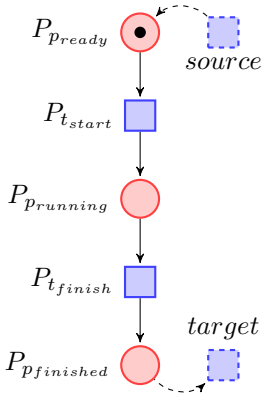


FIGURE 7.5 – Réseau de Petri résultant de la transformation d'un *Process*.

L'image d'un processus est donc constituée de trois places ($P_{p_{ready}}$, $P_{p_{running}}$ et $P_{p_{finished}}$), deux transitions ($P_{t_{start}}$ et $P_{t_{finish}}$) et quatre arcs. Dans le cas où il s'agit d'un processus hiérarchique, il peut y avoir un arc de synchronisation pointant vers la première place ($P_{p_{ready}}$) et un autre partant de la dernière place ($P_{p_{finished}}$).

Techniquement, nous implémentons cette transformation élémentaire par une *définition* comprenant une seule règle filtrant tous les éléments *Process* du modèle source (ligne 2). Dans cette règle, seul le nom du processus nous importe, nous n'instancions donc que la variable `name`. Dans le membre droit de la règle, les places et arcs de l'image d'un `Process` sont créés comme tout terme Tom, en utilisant la construction **backquote** (lignes 3 à 5, et 9 à 12). En revanche, pour créer les deux transitions `Pt_start` et `Pt_finish`, nous utilisons la construction **%tracelink** afin de *tracer* ces deux éléments (lignes 6 et 7). Notons que ces transitions nouvellement créées et tracées sont immédiatement utilisées dans la construction des arcs du réseau de Petri.

Le bloc de code des lignes 14 à 23 sert à la gestion des processus hiérarchiques : dans un tel cas, un processus possède un processus père qui est une *WorkDefinition* non `null`, et il existe un traitement particulier. Il s'agit de créer des éléments *resolve* par la construction **%resolve** (lignes 16 et 20) pour jouer le rôle de transitions créées dans une autre *définition*. En effet, ces deux nœuds sont censés être créés par la transformation de *WorkDefinitions* en réseaux de Petri. Ils sont représentés par les deux carrés bleus aux bords pointillés sur la figure 7.5. Les deux éléments *resolve* peuvent être immédiatement utilisés dans la construction d'autres termes (lignes 18 et 22, arcs en pointillés sur la figure 7.5) ou avec Java (lignes 17 et 21).

```
1  definition P2PN traversal ‘TopDown(P2PN(tom__linkClass,pn)) {
2    p@Process[name=name] -> {
3      Place Pp_ready  = ‘Place(name+"_ready", 1);
4      Place Pp_running = ‘Place(name+"_running", 0);
5      Place Pp_finished = ‘Place(name+"_finished", 0);
6      %tracelink(Pt_start:Transition, ‘Transition(name+"_start", pn, 1, 1));
7      %tracelink(Pt_finish:Transition, ‘Transition(name+"_finish", pn, 1, 1));
8
9      ‘Arc(Pt_start, Pp_ready, pn, ArcKindnormal(), 1);
10     ‘Arc(Pp_running, Pt_start, pn, ArcKindnormal(), 1);
11     ‘Arc(Pt_finish, Pp_running, pn, ArcKindnormal(), 1);
12     ‘Arc(Pp_finished, Pt_finish, pn, ArcKindnormal(), 1);
13
14     WorkDefinition from = ‘p.getFrom();
15     if (from!=null) {
16       /* WDt_start et WDt_finish : transitions de l’image d’une activité que
17       décrit le processus, par exemple B dans la figure 7.4 */
18       Transition source = %resolve(from:WorkDefinition, WDt_start:Transition);
19       source.setNet(pn);
20       Arc tmpZoomIn = ‘Arc(Pp_ready, source, pn, ArcKindnormal(), 1);
21
22       Transition target = %resolve(from:WorkDefinition, WDt_finish:Transition);
23       target.setNet(pn);
24       Arc tmpZoomOut = ‘Arc(target, Pp_finished, pn, ArcKindread_arc(), 1);
25     }
26   }
27 }
```

Listing 7.1 – `P2PN` : Code de la définition *ProcessToPetriNet*

WorkDefinitionToPetriNet. Une *WorkDefinition* SimplePDL est traduite par un réseau de Petri de la forme de celui donné par la figure 7.6. Cette transformation élémentaire est implémentée par la *définition* `WD2PN` donnée dans le listing 7.2.

Le réseau de Petri résultant de cette transformation élémentaire ressemble beaucoup à celui obtenu par transformation d'un *Process*. Il se différencie par un arc et une place supplémentaires $WD_{p_{started}}$ après la transition $WD_{t_{start}}$. Ces éléments additionnels par rapport à l'image d'un *Process* permettent l'ajout d'une séquence entre deux *WorkDefinitions*. L'image d'une activité est donc constituée de quatre places ($WD_{p_{ready}}$, $WD_{p_{running}}$ et $WD_{p_{finished}}$, $WD_{p_{started}}$), deux transitions ($WD_{t_{start}}$ et $WD_{t_{finish}}$) et cinq arcs. Dans le cas où il s'agit d'un processus hiérarchique, deux arcs de synchronisation avec le processus parent sont présents : l'un venant de la transition $P_{t_{start}}$ de l'image du processus parent et pointant sur la place $WD_{p_{ready}}$, l'autre partant de $WD_{p_{finished}}$ et pointant sur la transition $P_{t_{finish}}$ de l'image du *Process* parent.

Cette *définition* est implémentée par le bloc `definition WD2PN`, comprenant une règle similaire à celle de la *définition* `P2PN`, la différence étant que nous filtrons des éléments de type *WorkDefinition* et non plus *Process* (ligne 2). Les places et les transitions sont créées grâce à la construction **backquote** (lignes 3 et 5) ou *via* **%tracelink** (lignes 4, 6, 7 et 8). Tous ces termes — tracés ou non — sont immédiatement utilisés pour construire les arcs du réseau de Petri résultant (lignes 10 à 14).

En fin de bloc `definition` (lignes 16 à 25), les éléments intermédiaires *resolve* représentés dans la figure 7.6 par les deux carrés bleus avec les bords pointillés sont créés (lignes 19 et 23). Ils sont utilisés respectivement comme source et destination des arcs de synchronisation

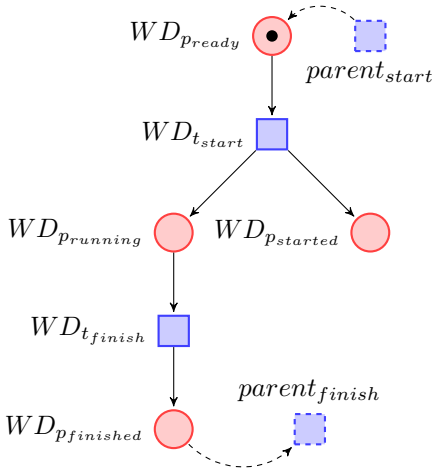


FIGURE 7.6 – Réseau de Petri résultant de la transformation d'une *WorkDefinition*.

avec le processus parent créés lignes 23 et 27.

Le fait de tracer quatre éléments dans cette *définition* aura pour conséquence de générer à la compilation une *ReferenceClass* ayant quatre champs correspondants.

```
definition WD2PN traversal `TopDown(WD2PN(tom__linkClass,pn)) {
  wd@WorkDefinition[name=name] -> {
    Place WDp_ready = `Place(name+"_ready", pn, 1);
    %tracelink(WDp_started:Place, `Place(name+"_started", pn, 0));
    Place WDp_running = `Place(name+"_running", pn, 0);
    %tracelink(WDp_finished:Place, `Place(name+"_finished", pn, 0));
    %tracelink(WDt_start:Transition, `Transition(name+"_start", pn, 1, 1));
    %tracelink(WDt_finish:Transition, `Transition(name+"_finish", pn, 1, 1));

    `Arc(WDt_start, WDp_ready, pn, ArcKindnormal(), 1);
    `Arc(WDp_started, WDt_start, pn, ArcKindnormal(), 1);
    `Arc(WDp_running, WDt_start, pn, ArcKindnormal(), 1);
    `Arc(WDt_finish, WDp_running, pn, ArcKindnormal(), 1);
    `Arc(WDp_finished, WDt_finish, pn, ArcKindnormal(), 1);

    SimplePDLSemantics.DDMMSimplePDL.Process parent = `wd.getParent();
    /* Pt_start et Pt_finish : transitions de l'image d'un processus, par
    exemple P_root dans la figure 7.4 */
    Transition source = %resolve(parent:Process, Pt_start:Transition);
    source.setNet(pn);
    Arc tmpDistribute = `Arc(WDp_ready, source, pn, ArcKindnormal(), 1);

    Transition target = %resolve(parent:Process, Pt_finish:Transition);
    target.setNet(pn);
    Arc tmpRejoin = `Arc(target, WDp_finished, pn, ArcKindread_arc(), 1);
  }
}
```

Listing 7.2 – `WD2PN` : Code de la définition *WorkDefinitionToPetriNet*

WorkSequenceToPetriNet. Une *WorkSequence* SimplePDL est traduite par un réseau de Petri constitué d'un arc, comme illustré par la figure 7.7. Cette transformation élémentaire est implémentée par la *définition* `WS2PN` donnée par le listing 7.3.

Dans cette *définition*, seul un arc est créé à partir de l'élément source filtré (*WorkSequence*). Cependant, tout arc ayant deux extrémités et ces deux extrémités étant des éléments obtenus lors de l'application d'autres transformations élémentaires, il est nécessaire de construire des éléments *resolve*. Les extrémités de l'arc image dépendent du type de la *WorkSequence* filtrée. Nous filtrons donc sur `linkType` (ligne 5) et, compte tenu des règles écrites et du fait que Tom donne toutes les solutions possibles du filtrage, nous avons la garantie que pour un type de contrainte de précédence donné, deux règles seront déclenchées (une parmi celles des lignes 6 et 9, l'autre parmi celles des lignes 13 et 16). Après exécution de ce bloc, les variables `source` et `target` sont bien initialisées et peuvent être utilisées pour construire l'arc image `wsImage` (ligne 23) de la séquence filtrée.

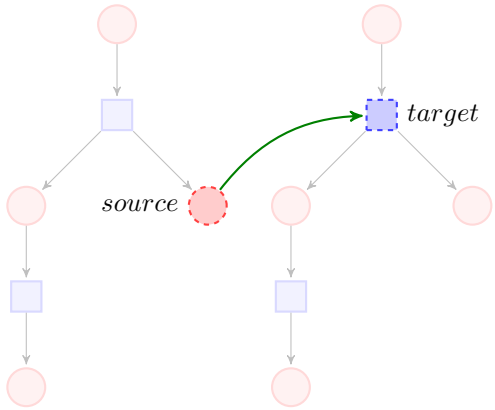


FIGURE 7.7 – Réseau de Petri résultant de la transformation d'une *WorkSequence*.

```
definition WS2PN traversal `TopDown(WS2PN(tom__linkClass,pn)) {
  ws@WorkSequence[predecessor=p,successor=s,linkType=linkType] -> {
    Place source= null;
    Transition target= null;
    %match(linkType) {
      (WorkSequenceTypefinishToFinish|WorkSequenceTypefinishToStart)[] -> {
              source = %resolve(p:WorkDefinition, WDp_finished:Place);
      }
      (WorkSequenceTypestartToStart|WorkSequenceTypestartToFinish)[] -> {
              source = %resolve(p:WorkDefinition, WDp_started:Place);
      }

      (WorkSequenceTypefinishToStart|WorkSequenceTypestartToStart)[] -> {
              target = %resolve(s:WorkDefinition, WDt_start:Transition);
      }
      (WorkSequenceTypestartToFinish|WorkSequenceTypefinishToFinish)[] -> {
              target = %resolve(s:WorkDefinition, WDt_finish:Transition);
      }
    }
    source.setNet(pn);
    target.setNet(pn);

    Arc wsImage = `Arc(target,source, pn, ArcKindread_arc(), 1);
  }
}
```

Listing 7.3 – `WS2PN` : Code de la définition *WorkSequenceToPetriNet*

Transformation globale. Ces blocs `definition` s'intègrent dans une transformation Tom+ Java dont la forme générale du code est donnée par le listing 7.4. Le code complet de la transformation est quant à lui donné dans l'annexe A.1 et est directement accessible dans le dépôt du projet [53]. Notons que cette transformation sert aussi de support pour la documentation sur le site officiel de Tom [54]. Expliquons le reste du code de la transformation dans ses grandes lignes :

début : Les points de suspension de la ligne 1 représentent du code java classique (`package` et `import`).

ancrages : Au sein de la classe `SimplePDLToPetriNet`, nous notons l'usage de plusieurs constructions `%include`. Celle de la ligne 3 sert à charger les ancrages formels de la bibliothèque de stratégies, celle de la ligne 4 charge l'ancrage du modèle de lien (fourni comme bibliothèque) et celle de la ligne 5 permet de charger les types Ecore, eux aussi fournis sous la forme d'une bibliothèque. Les deux ancrages chargés lignes 7 et 8 ont été générés par Tom-EMF. Le bloc de code suivant montre des déclarations de variables et l'écriture d'un *mapping* minimal permettant d'utiliser la classe `SimplePDLToPetriNet` comme un type Tom. Les points de suspension suivants représentent du code Java (déclarations de variables, *etc.*).

transformation : Les trois *définitions* constituent le corps d'un bloc `%transformation` (lignes 18 à 29). Nous avons choisi de les écrire dans l'ordre dans lequel nous les avons présentées, cependant nous rappelons que **cet ordre n'a aucune importance** avec notre approche.

main : Nous avons extrait une partie de `main()`. Avant l'extrait, il s'agit de contrôles ainsi que du code permettant de charger un modèle ou créer un modèle en Tom dans le cas où aucun fichier n'est passé en paramètre. L'extrait comprend quant à lui la création de la stratégie de transformation (ligne 38) ainsi que son appel (ligne 40). La *stratégie de résolution* est appelée à la ligne 42. Étant générée, son nom est construit de manière prédictible pour l'utilisateur, à partir du nom de la transformation ainsi que d'un préfixe explicite (`tom_StratResolve_`). La stratégie appelée à la ligne 44 sert pour l'affichage du résultat de la transformation.

fin : Le reste du code qui n'est pas montré dans le listing 7.4 consiste en des méthodes d'affichage et de sérialisation pour obtenir un réseau de Petri compatible avec le format d'entrée du *model-checker* TINA [55] [BRV04].

Usage de cette transformation. Cette transformation étant bien connue, elle nous a servi de support pour le développement de nos outils. Son intérêt étant de pouvoir vérifier formellement des propriétés de son résultat, nous avons dépassé le simple développement de la transformation pour vérifier nos résultats. C'est pour cette raison que le modèle cible est généré par défaut au format d'entrée de TINA. Cela nous permet de le visualiser et d'en vérifier des propriétés avec le *model-checker*.

Ainsi, nous avons pu exprimer une formule ainsi que des propriétés en logique temporelle linéaire (LTL) telles que la terminaison. Nous les avons ensuite vérifiées avec TINA sur le réseau de Petri résultant de la transformation. La formule, les propriétés ainsi que les résultats sont donnés en annexe A.4 de ce document.

53. `https://gforge.inria.fr/scm/?group_id=78`
54. `http://tom.loria.fr/wiki/index.php5/Documentation:Playing_with_EMF`
55. `http://projects.laas.fr/tina`

```
...
public class SimplePDLToPetriNet {
  %include{ sl.tom }
  %include{ LinkClass.tom }
  %include{ emf/ecore.tom }

  %include{ mappings/DDMMPetriNetPackage.tom }
  %include{ mappings/DDMMSimplePDLPackage.tom }

  private static PetriNet pn = null;
  private static LinkClass tom__linkClass;

  %typeterm SimplePDLToPetriNet { implement { SimplePDLToPetriNet }}
  public SimplePDLToPetriNet() {
    this.tom__linkClass = new LinkClass();
  }
 ...
  %transformation SimplePDLToPetriNet(tom__linkClass:LinkClass,pn:PetriNet) :
            "metamodels/SimplePDL.ecore" -> "metamodels/PetriNet.ecore" {
    definition P2PN traversal `TopDown(P2PN(tom__linkClass,pn)) {
      /* code du listing  7.1 */
    }
    definition WD2PN traversal `TopDown(WD2PN(tom__linkClass,pn)) {
      /* code du listing  7.2 */
    }
    definition WS2PN traversal `TopDown(WS2PN(tom__linkClass,pn)) {
      /* code du listing  7.3 */
    }
  }

  public static void main(String[] args) {
    ...
    SimplePDLToPetriNet translator = new SimplePDLToPetriNet();
    Introspector introspector = new EcoreContainmentIntrospector();
    // processus à transformer
    simplepdl.Process p_root = `Process("root", ...);

    Strategy transformer =
            `SimplePDLToPetriNet(translator.tom__linkClass,translator.pn);
    transformer.visit(p_root, introspector);
    //Appel de la stratégie de résolution générée
    `TopDown(tom__StratResolve_SimplePDLToPetriNet(translator.tom__linkClass,
                        translator.pn)).visit(translator.pn, introspector);
    `TopDown(Sequence(PrintTransition()),PrintPlace()).visit(translator.pn,
                                                        introspector);
    ...
  }
  ...
}
```

Listing 7.4 – Forme générale du code de la transformation *SimplePDLToPetriNet*

7.2 Aplatissement d'une hiérarchie de classes

Cette deuxième étude de cas avait pour but d'évaluer l'intérêt et les limites éventuelles des nouvelles constructions intégrées au langage Tom. Il s'agit d'une transformation endogène très simple : l'aplatissement d'une hiérarchie de classes. Nous disposons de classes, dont certaines héritent d'autres. Nous souhaitons aplatir cette hiérarchie en reportant les attributs des surclasses dans les classes qui sont les feuilles de l'arbre hiérarchique. Nous choisissons aussi de nommer les nouvelles classes reprenant tous les attributs hérités. Les classes qui ne sont pas impliquées dans une relation d'héritage ne changent pas.

Nous présentons un exemple de transformation dans la section 7.2.1 et le métamodèle dans la section 7.2.2. Pour évaluer et comparer, nous avons écrit plusieurs implémentations de cet exemple, que nous décrivons dans la section 7.2.3.

7.2.1 Exemple de transformation

Nous considérons comme modèle source la hiérarchie de classes donnée par le membre gauche de la figure 7.8. La classe *C* possède un attribut *attrC* de type *C* et n'est dans aucune hiérarchie de classes. La classe *B* est quant à elle dans une hiérarchie de classes : elle hérite de *A* qui hérite elle-même de *D*, surclasse de toute la hiérarchie. La classe *B* a deux attributs *attrB1* et *attrB2* de types *C*. Cette transformation aplatit la hiérarchie et doit donc produire le modèle cible illustré par le membre droit de la figure 7.8. Il s'agit d'un modèle constitué de deux classes : la classe *C* — qui reste inchangée par rapport au modèle source — ainsi qu'une classe *DAB* qui rassemble tous les attributs de la hiérarchie de classe aplatie.

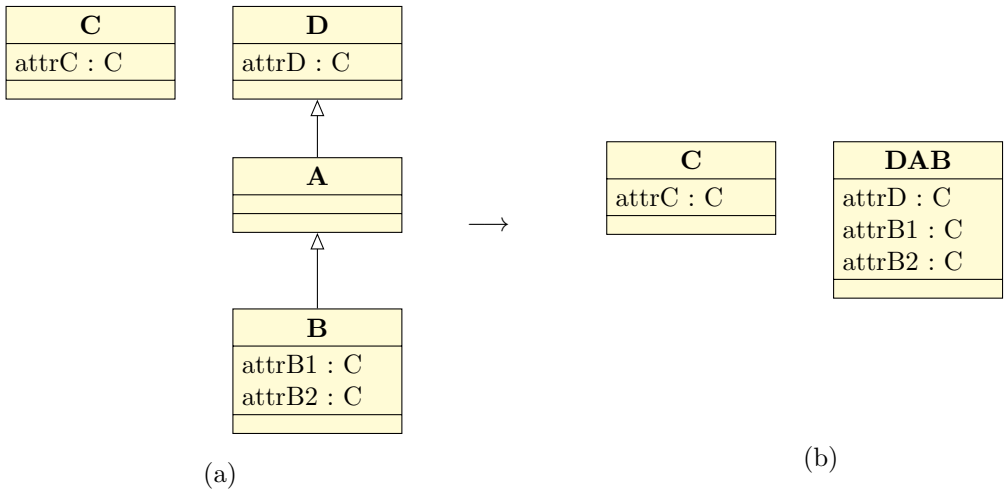


FIGURE 7.8 – Aplatissement d'une hiérarchie de classes.

7.2.2 Métamodèle

S'agissant d'une transformation endogène, le métamodèle source est identique au métamodèle cible. Pour cette transformation, nous utilisons le métamodèle simplifié d'UML donné par la figure 7.9.

Les *Classifiers* sont des éléments ayant un nom et étant de type *DataType* ou de type *Class*. Un élément *Class* peut avoir des attributs (*Attribute*), qui sont eux-mêmes des *Classifiers*. Dans notre contexte technique, nous avons besoin d'une racine afin d'obtenir un arbre de recouvrement. Nous avons donc ajouté une racine virtuelle dans le métamodèle — élément *VirtualRoot* — qui contient tous les *Classifiers* (relation de composition).

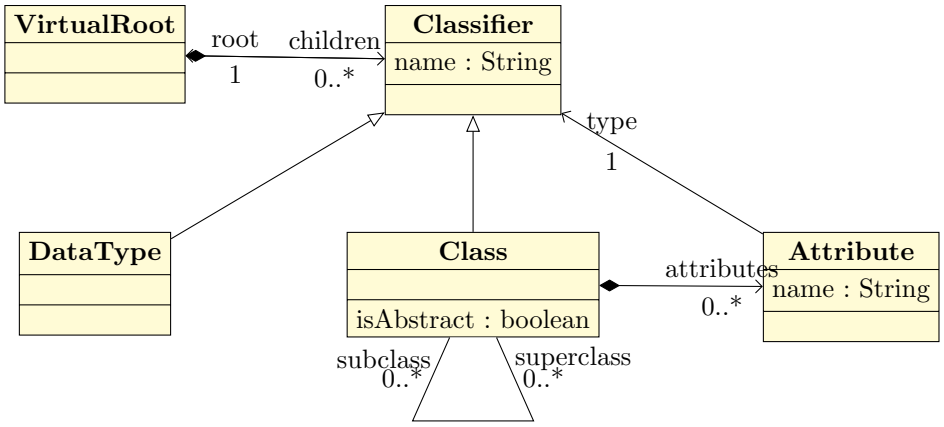


FIGURE 7.9 – Métamodèle d'UML simplifié.

Pour les besoins de l'étude et afin de simplifier les explications et le développement, nous avons considéré une version épurée du métamodèle illustré par la figure 7.10.

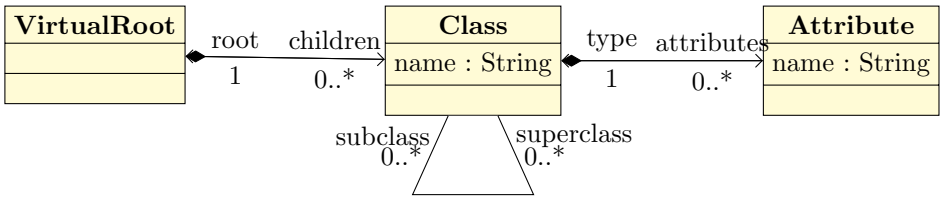


FIGURE 7.10 – Métamodèle considéré pour l'étude de cas.

7.2.3 Implémentation utilisant les outils développés

Nous avons implémenté cet exemple de plusieurs manières afin de comparer l'intérêt d'utiliser les outils développés durant cette thèse pour la transformation de modèles :

1. La première version de cette implémentation est écrite en pur Java (+EMF), sans l'aide de Tom, des nouvelles constructions et des outils liés tels que Tom-EMF. Il s'agit d'une version mêlant récursivité et itération ;
2. la deuxième version est écrite en Tom+Java mais sans user des stratégies ni de la nouvelle construction **%transformation**. En revanche, nous avons utilisé Tom-EMF pour générer les *mappings* ;
3. la troisième implémentation est une application de la méthode présentée dans [BCMP12], à savoir l'écriture d'une transformation en utilisant les stratégies de réécriture, mais sans la construction haut niveau **%transformation** ;
4. la quatrième et dernière version utilise les outils développés dans le cadre de cette thèse.

Pour des raisons de lisibilité, nous faisons uniquement apparaître des extraits significatifs de code de cette transformation dans cette section. Le code complet des implémentations est donné dans l'annexe B.1.

Version 1 : Java-EMF. L'implémentation en Java-EMF d'une telle transformation se révèle sans véritable difficulté. Le principe est de parcourir les classes du modèle source et d'appliquer

un aplatissement récursif sur celles qui sont les feuilles de l'arbre d'héritage. Cette transformation peut être implémentée en environ 40 lignes de code, comme le montre le listing 7.5 (code complet en annexe B.1.1).

```
1  public static VirtualRoot v1_processClass(VirtualRoot root) {
2    org.eclipse.emf.common.util.EList<Class> newChildren =
3      new org.eclipse.emf.common.util.BasicEList<Class>();
4    for(Class cl : root.getChildren()) {
5      if(cl.getSubclass().isEmpty()) {
6        newChildren.add(flattening(cl));
7      }
8    }
9    VirtualRoot result = (VirtualRoot) ClassflatteningFactory.eINSTANCE.create(
10                          (EClass)ClassflatteningPackage.eINSTANCE.getEClassifier("VirtualRoot"));
11   result.eSet(result.eClass().getEStructuralFeature("children"), newChildren);
12   return result;
13 }
14
15 public static Class flattening(Class toFlatten) {
16   Class parent = toFlatten.getSuperclass();
17   if(parent==null) {
18     return toFlatten;
19   } else {
20     Class flattenedParent = flattening(parent);
21     EList<Attribute> head = toFlatten.getAttributes();
22     head.addAll(flattenedParent.getAttributes());
23     Class result = (Class)ClassflatteningFactory.eINSTANCE.
24                     create((EClass)ClassflatteningPackage.eINSTANCE.getEClassifier("Class"));
25     result.eSet(result.eClass().getEStructuralFeature("name"), flattenedParent.getName()+toFlatten.getName());
26     result.eSet(result.eClass().getEStructuralFeature("attributes"), head);
27     result.eSet(result.eClass().getEStructuralFeature("superclass"), flattenedParent.getSuperclass());
28     result.eSet(result.eClass().getEStructuralFeature("subclass"), (new BasicEList<Class>()) );
29     result.eSet(result.eClass().getEStructuralFeature("root"), virtR);
30     return result;
31   }
32 }
33 ...
34 public static void main(String[] args) {
35   ...
36   VirtualRoot translator.virtR = v1_processClass(source_root);
37   ...
38 }
```

Listing 7.5 – Version 1 : Implémentation de la transformation d'aplatissement de hiérarchie de classes en Java.

Les pré-requis pour cette version de la transformation sont de maîtriser Java et de connaître un minimum EMF afin d'être en mesure d'écrire les appels adéquats pour créer un élément. Un défaut de cette implémentation est la lisibilité du code, le langage Java ainsi que le *framework* EMF étant particulièrement verbeux.

Version 2 : Tom+Java-EMF. Pour remédier à ce désagrément — qui peut devenir un enjeu fort dans le cadre de la maintenance logicielle industrielle — nous avons modifié l'implémentation initiale avec Tom, afin d'user de ses facilités d'écriture. L'utilisation de la construction **%match** (filtrage de motif) ainsi que du *backquote* (création et manipulation de termes) permettent notamment d'améliorer la lisibilité du programme. Le listing 7.6 est le code résultant de l'évolution du précédent listing, intégrant du code Tom simple (le code complet est donné dans l'annexe B.1.2).

Cet extrait de code est plus concis que la version en pur Java et EMF (moins de 25 lignes pour la transformation elle-même), mais il est surtout plus lisible. Pour utiliser la construction **backquote** (‘) comme nous le faisons dans ce listing, des ancrages algébriques sont nécessaires. Nous avons bien évidemment utilisé notre générateur d'ancrages formels Tom-EMF plutôt que de les écrire manuellement. L'utilisateur n'a donc pas de travail additionnel à fournir par rapport à une transformation en pur Java et EMF autre que la commande de génération (dans

```
1 public static VirtualRoot v2_processClass(VirtualRoot root) {
2   EList<Class> children = root.getChildren();
3   EList<Class> newChildren = `cfClassEList();
4   %match(children) {
5     cfClassEList(_*,cl@cfClass(_,_,_,cfClassEList(),_),_*) -> {
6       newChildren = `cfClassEList(flattening(cl),newChildren*);
7     }
8   }
9   return `cfVirtualRoot(newChildren);
10 }
11
12 public static Class flattening(Class toFlatten) {
13   Class parent = toFlatten.getSuperclass();
14   if(parent==null) {
15     return toFlatten;
16   } else {
17     Class flattenedParent = flattening(parent);
18     EList<Attribute> head = toFlatten.getAttributes();
19     head.addAll(flattenedParent.getAttributes());
20     return `cfClass(flattenedParent.getName()+toFlatten.getName(), head, flattenedParent.getSuperclass(),
21                                                                                    cfClassEList(), virtR);
22   }
23 }
24 ...
25 public static void main(String[] args) {
26   ...
27   VirtualRoot translator.virtR = v2_processClass(source_root);
28   ...
29 }
```

Listing 7.6 – Version 2 : Implémentation de la transformation d'aplatissement de hiérarchie de classes en Tom+Java.

la précédente version, l'utilisateur doit aussi écrire le métamodèle et générer le code EMF avec Eclipse).

Version 3 : Tom+Java-EMF avec stratégies. Les stratégies étant un aspect important de Tom, nous écrivons une autre version de cette transformation les utilisant. C'est l'occasion de mettre en œuvre la méthode présentée dans [BCMP12]. Dans cette nouvelle implémentation (extrait dans le listing 7.7, code complet en annexe B.1.3), nous utilisons toujours les ancrages algébriques générés par Tom-EMF et nous ajoutons une stratégie Tom. L'usage des stratégies avec des modèles EMF Ecore implique aussi l'utilisation de l'outil *EcoreContainmentIntrospector* présenté dans 6.3.2. Pour rappel, il permet le parcours des modèles EMF Ecore vus sous leur forme de termes.

Habituellement, l'utilisation des stratégies Tom simplifie systématiquement et grandement l'écriture de code ainsi que sa lisibilité, le parcours — traité par les bibliothèques que nous fournissons — étant séparé du traitement. Cependant, après écriture et exécution de la transformation, nous nous apercevons que la transformation n'est ni véritablement plus courte, ni plus lisible, ni plus efficace que les implémentations précédentes.

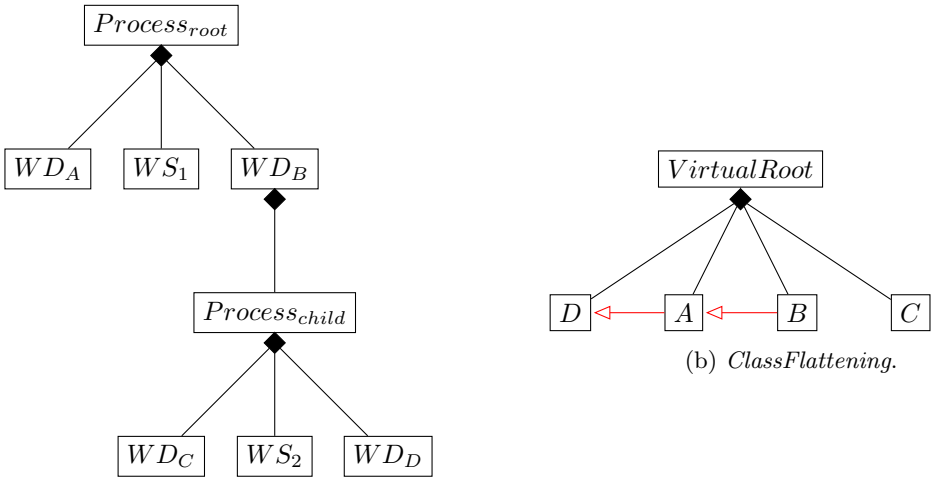
C'est en observant plus précisément le métamodèle de notre exemple, la transformation attendue ainsi que l'outil permettant l'utilisation des stratégies que l'on identifie la raison. Un modèle EMF a une racine unique par la relation de composition et peut donc être représenté sous la forme d'un arbre, comme nous le faisons dans Tom. La figure 7.11 illustre ce mécanisme appliqué aux modèles sources des deux exemples que nous présentons dans ce chapitre.

Dans cette figure, les deux termes sont représentés de manière classique : la racine est en haut, les feuilles en bas. Un losange noir — le symbole de la relation de composition en modélisation — a été ajouté à chaque endroit où la relation est une relation de composition, c'est-à-dire à chaque relation père-fils. L'arbre est bien un arbre par la relation de composition.

Si l'on examine la figure 7.11a, nous nous apercevons que les relations de sa structure correspondent à celles qui nous intéressent dans l'exemple, à savoir les relations de composi-

```
1 %strategy FlatteningStrat(translator:UMLClassesFlattening) extends Identity() {
2   visit cfVirtualRoot {
3     cfVirtualRoot(cfClassEList(_*,cl@cfClass(n,_,_,cfClassEList(),_),_*)) -> {
4       EList<Class> newChildren = translator.virtR.getChildren();
5       translator.virtR = `cfVirtualRoot(cfClassEList(flattening(cl),newChildren*));
6     }
7   }
8 }
9
10 public static Class flattening(Class toFlatten) {
11   Class parent = toFlatten.getSuperclass();
12   if(parent==null) {
13     return toFlatten;
14   } else {
15     Class flattenedParent = flattening(parent);
16     EList<Attribute> head = toFlatten.getAttributes();
17     head.addAll(flattenedParent.getAttributes());
18     return `cfClass(flattenedParent.getName()+toFlatten.getName(), head,
19         flattenedParent.getSuperclass(), cfClassEList(), null);
20   }
21 }
22 ...
23 public static void main(String[] args) {
24   ...
25   VirtualRoot translator.virtR = `cfVirtualRoot(cfClassEList());
26   Strategy transformer = `BottomUp(FlatteningStrat(translator));
27   transformer.visit(source_root, new EcoreContainmentIntrospector());
28   ...
29 }
```

Listing 7.7 – Version 3 : Implémentation de la transformation d'aplatissement de hiérarchie de classes en Tom+Java avec stratégies.



(a) *SimplePDLToPetriNet*.

(b) *ClassFlattening*.

FIGURE 7.11 – Arbres représentant les modèles source des exemples *SimplePDLToPetriNet* (a) et *ClassFlattening* (b).

tion. En revanche, dans le cas de l'exemple de l'aplatissement d'une hiérarchie de classes, la relation entre éléments qui nous intéresse véritablement est la relation d'héritage, modélisée par une relation bidirectionnelle *subclass–superclass* et non par une relation de composition. Nous représentons ces relations « intéressantes » dans la figure 7.11b par des flèches rouges. La seule relation de composition de cet exemple est une relation de composition artificielle que nous avons créée (ainsi que l'élément de type *VirtualRoot*) afin d'avoir une racine et donc de pouvoir écrire ce modèle EMF Ecore. Notre outil *EcoreContainmentIntrospector* descendra

bien dans les arbres, mais dans le cas du second exemple, il ne servira qu'à obtenir tous les fils de cette racine virtuelle qui sont à plat. Ensuite, pour l'aplatissement en lui-même, nous faisons tout de même appel à une fonction `flattening()` récursive, que nous utilisions ou non des stratégies.

Passé ce constat, la dernière version de l'implémentation reposant elle aussi sur les stratégies de réécriture mais avec les nouvelles constructions, nous pouvons supposer qu'elle ne sera pas meilleure (plus lisible, plus concise et plus efficace). Nous constatons effectivement que l'implémentation est moins lisible et moins concise, avec une efficacité similaire. Différents facteurs permettent d'expliquer ce résultat : d'une part, comme pour la version précédente, les relations nous intéressant ne sont pas celles constituant l'arbre de recouvrement, d'autre part, cette transformation est trop simple pour tirer parti de nos outils. Expliquons plus en détail cet aspect. Dans cette transformation, nous ne pouvons extraire plusieurs transformations élémentaires. La transformation globale sera donc constituée d'une seule *définition*, encodée par une stratégie de réécriture, comme dans la version précédente de l'implémentation. Ainsi, le gain habituellement apporté par la construction **%transformation** est complètement absent. Outre ce point, nous constatons qu'aucun élément en cours de transformation ne nécessite le résultat d'un autre élément devant être transformé. Il n'y a donc pas besoin d'introduire d'élément *resolve* dans la transformation. L'un des apports de nos outils étant de gérer l'ordonnancement des pas d'exécution en générant une stratégie de résolution, son intérêt reste limité pour cette transformation.

Finalement, nous pouvons donc déduire de cet exemple que nos outils ne sont pas pleinement adaptés à toutes les transformations. Dans ces quatre implémentations, la deuxième version semble être le compromis le plus judicieux. Le générateur de *mappings* y joue un rôle majeur, nous utilisons une partie du langage Tom, en revanche nous nous passons des constructions plus complexes telles que les stratégies ainsi que les nouvelles constructions intégrées durant cette thèse. Cependant, les nouvelles constructions pour de tels exemples ne sont pas pour autant inintéressantes : en effet, si la résolution (et sa construction associée **%resolve**) n'apporte pas de gain, il subsiste le second aspect de nos travaux, à savoir la traçabilité. L'utilisation de la construction **%tracelink** afin de générer un modèle de lien reste possible.

Une autre conclusion de l'étude de cet exemple est que nous avons développé nos outils en visant la couverture d'un grand nombre de transformations, notamment celles où les relations de composition sont au cœur. Une perspective serait maintenant de travailler à l'élaboration d'outils gérant d'autres relations ou étant plus génériques. Nous pensons notamment à des stratégies de réécriture que nous pourrions paramétrer par des types de relations à suivre.

7.3 Synthèse

Dans ce chapitre, nous avons présenté deux cas d'étude pour deux objectifs distincts : *SimplePDLToPetriNet* et *UMLHierarchyFlattening*. La transformation *SimplePDLToPetriNet* nous a permis de présenter l'utilisation des outils que nous avons développés durant cette thèse en déroulant complètement une transformation. La seconde étude de cas nous a permis de donner une première évaluation de nos outils dans un contexte où ils ne peuvent donner leur pleine mesure.

L'objectif de cette seconde étude de cas était de repérer les points d'amélioration de nos outils, tant dans leur mise en œuvre actuelle qu'envisagée, et de nous donner de nouvelles perspectives techniques et scientifiques. En effet, si cette étude de cas nous a montré que nos outils n'étaient pas tous adaptés dans toutes les situations, elle nous a permis en revanche de relever un point intéressant. La relation de composition dans les modèles est centrale et se retrouve bien souvent au cœur des transformations de modèles. Dans notre contexte, elle nous permet d'avoir la vision arborescente des modèles que nous pouvons parcourir. Cependant, pour certaines transformations comme celle d'aplatissement d'une hiérarchie de classes, la relation d'intérêt n'est pas celle de composition. Partant de ce constat, il est intéressant de

se poser la question de la généralisation des stratégies pour les transformations de modèles. Une piste est la paramétrisation des stratégies par le type de relation à suivre lors de la traversée des termes. Dans un premier temps, pour tester la validité de ce principe, on pourrait implémenter un *introspecteur* dédié à d'autres types de relations (héritage notamment). Cette extension lèverait la limitation révélée par la seconde étude de cas. Ensuite, une seconde question d'intérêt serait de travailler sur la possibilité de paramétrer dynamiquement une stratégie : est-il possible de changer le type de lien à suivre en cours de parcours ? Ce type de mécanisme permettrait de déclencher localement une stratégie avec un autre type de parcours, et donc d'adopter une stratégie de réécriture en fonction du contexte.

Le premier exemple nous a aussi servi de support pour le développement de nos outils, et notre confiance en notre implémentation de cette transformation étant forte, nous nous en sommes aussi servi pour mener des expériences que nous présentons dans le chapitre suivant.

Chapitre 8

Résultats expérimentaux

Dans ce chapitre, nous présentons des résultats expérimentaux obtenus avec nos outils, notamment leurs performances. Nous discutons notre approche et nos choix technologiques, leur intérêt, ainsi que les limitations et perspectives de notre implémentation technique. Nous avons tenté d'évaluer nos outils afin d'améliorer les points limitants et de parfaire nos points forts par rapport à d'autres outils tels qu'ATL ou d'autres outils de transformation utilisant des stratégies de réécriture.

8.1 Utilisabilité

Compte tenu du fait qu'un de nos objectifs était de simplifier le processus de développement pour les utilisateurs, un premier aspect de notre évaluation concerne l'utilisabilité de nos outils. Fortement dépendant du contexte et de l'utilisateur, ce critère est difficilement mesurable et quantifiable. Nous pouvons toutefois exposer des retours utilisateurs.

Nous avons pu tester en conditions réelles une partie de nos outils : il s'agissait de développer une transformation en Tom+Java dans le cadre du projet quarteFt par une entreprise partenaire du projet. Le développeur était un ingénieur en informatique maîtrisant le langage Java ainsi que l'environnement Eclipse. Il ne connaissait pas Tom, ni les langages à base de règles de réécriture, ni le concept de stratégie avant de commencer.

La prise en main du langage Tom sans les nouvelles constructions s'est faite sans aucune difficulté et sans réel besoin de support de notre part hormis la documentation officielle en ligne. Nous avons donné un support informel pour l'usage des stratégies Tom afin d'accélérer son développement. L'outil Tom-EMF a été utilisé pour générer les ancrages formels inclus dans la transformation. Si l'usage en lui-même n'a pas posé de problème à l'ingénieur (peu d'options différentes, documentation centralisée), nous lui avons fourni un support plus important. En effet, l'usage par un non développeur a permis de découvrir des *bugs* et des manques de fonctionnalités (besoin de génération massive de *mappings* en une seule fois qui a donné lieu à la fonctionnalité de génération multiple de *EPackages*, conflits de noms qui a donné lieu à la fonctionnalité de préfixage, génération multiple des ancrages pour les bibliothèques UML2 et Ecore, *etc.*). Ces premiers retours utilisateur ont joué un rôle important dans l'amélioration de l'outil Tom-EMF.

Lors de ce test en conditions réelles, l'utilisateur a suivi la méthodologie avec les stratégies, mais n'a cependant pas utilisé les constructions Tom haut-niveau dédiées aux transformations de modèles. En effet, elles n'étaient pas incluses dans la version stable du moment.

L'aspect hybride de notre approche a permis à l'utilisateur d'être très rapidement opérationnel sans qu'il ait à apprendre un nouveau langage complet. De plus, du fait de son environnement de développement, sa transformation a pu être immédiatement intégrée dans les outils développés, ce qui n'aurait pas été aussi simple avec des langages tels que ATL, Kermeta ou même Stratego. Si notre approche utilisant Tom fournit moins de fonctionnalités

pour la transformation de modèles que ATL, nous avons l'avantage de ne pas nous séparer des fonctionnalités (bibliothèques notamment) du langage généraliste considéré (Java dans notre cas), ce qui permet à l'utilisateur d'exprimer tout de même ce qu'une construction Tom ne pourrait pas fournir nativement.

Le principe d'utiliser des stratégies de réécritures pour encoder les transformations de modèles semble aussi un choix judicieux du point de vue de l'utilisateur : la décomposition d'une transformation en transformations élémentaires est naturelle pour appréhender la complexité et le principe d'écriture de règles est aussi courant dans d'autres langages. Nous ne changeons donc pas fondamentalement les habitudes du développeur. Grâce aux mécanismes en jeu dans notre approche, nous avons une forte expressivité dans les motifs (membre gauche) et nous offrons une grande liberté à l'utilisateur dans les actions (membre droit) grâce à leur nature composite.

Le fait que notre approche se fonde sur un changement d'espace technologique pourrait paraître limitant en première approche (outil supplémentaire pour opérer le changement), cependant cet outil (générateur d'ancrages formels) est extrêmement simple d'utilisation et ne nécessite qu'une seule intervention en début de développement.

Le choix de l'usage d'un outil par rapport à un autre ne peut être définitif et universel : il doit être effectué de manière réfléchie en tenant compte de critère tels que l'environnement de développement, les fonctionnalités respectives des outils envisagés ainsi que les performances des outils. Dans un contexte Java+EMF, compte tenu des premiers retours, le choix de nos outils est pertinent.

8.2 Performances

De manière générale, un outil peut être extrêmement intéressant scientifiquement, mais inutilisable du fait de ses performances (implémentation inefficace ou problème à résoudre trop complexe). Dans un cadre industriel, cette question se pose rapidement, étant donné que des modèles « réels » (généralement de plus grandes tailles que ceux utilisés pour le prototypage initial) sont transformés. Il est donc naturel d'évaluer ses performances, même si n'est pas l'objectif premier de l'outil. Dans le domaine de la vérification du logiciel, les outils de *model-checking* sont généralement des outils consommateurs de ressources (temps et mémoire), ce qui peut constituer un goulet d'étranglement dans une chaîne de développement imposant une vérification de ce type. Il ne faut donc pas que les autres traitements aient des performances moindres (et deviendraient de fait le nouveau goulet d'étranglement), ou que leurs temps d'exécution ajoutent trop de délais avant ce goulet. Un second aspect de l'évaluation des performances vient du fait que beaucoup d'outils de transformation existent, mais leur passage à l'échelle est souvent un facteur limitant. C'est donc dans ce contexte que nous avons effectué une première évaluation des performances de nos outils, d'une part Tom-EMF, d'autre part les constructions haut-niveau dédiées aux transformations de modèles.

8.2.1 Tom-EMF

Nous avons expérimenté Tom-EMF sur divers métamodèles afin de nous assurer de son passage à l'échelle. Très rapidement, nous avons constaté que peu de très gros métamodèles sont utilisés, les métamodèles Ecore et UML étant souvent les plus gros métamodèles utilisés régulièrement. Nous avons donc appliqué notre outil sur ces métamodèles pour générer les *mappings* correspondant, ainsi que sur les parties des deux métamodèles du cas d'utilisation présenté dans le chapitre 7.1.

Nous constatons que les temps de génération donnés dans la table 8.1 sont extrêmement bas, de l'ordre de la seconde. Notre outil de génération d'ancrages est tout à fait capable de gérer un « gros » métamodèle, dans un temps négligeable par rapport à la durée de développement et d'exécution d'une transformation comme nous allons le voir. Considérant ces temps de génération pour de tels métamodèles, nous estimons que les performances de l'ou-

Métamodèle	Ecore	Mappings Tom		Temps moyen de
	#eClassifiers	#sortes	#opérateurs	génération (en ms)
Ecore	52	56	28	24
UML2	264	345	346	1527
SimplePDL	4	8	10	256
PetriNet	2	8	8	254

TABLE 8.1 – Tailles de quelques métamodèles significatifs ainsi que des *mappings* Tom-EMF correspondant.

til Tom-EMF sont largement suffisantes et qu'elles ne sont pas limitantes dans la chaîne de développement.

Outre l'aspect performances que nous jugeons satisfaisant, cet outil s'est avéré simple à utiliser pour un utilisateur complètement extérieur au projet Tom. Cela nous donne un indice positif sur l'utilisabilité de notre outil.

8.2.2 Transformation

Afin de tester les performances du langage, nous avons utilisé la transformation *SimplePDLToPetriNet* sur des modèles de tailles différentes. Nous avons fait ce choix, car ayant servi de support au développement de nos outils, nous en avions une bonne maîtrise tout en ayant une confiance élevée dans son implémentation. Nous l'avons aussi implémentée dans d'autres langages afin d'avoir des points de comparaison avec l'existant. S'agissant d'une transformation connue dans le domaine, nous avons pu corriger et améliorer ces implémentations grâce à celles existantes.

Pour le modèle d'entrée, nous avons défini un processus paramétrable généré de manière déterministe en fonction de deux paramètres :

— le nombre de *WorkDefinitions* présentes (`N`) ;

— le fait qu'il y ait ou non des *WorkSequences*.

Le processus d'entrée n'est pas hiérarchique, mais cela n'a pas d'impact sur la complexité du modèle d'entrée étant donné que :

— l'image d'un *Process* est plus simple que l'image d'une *WorkDefinition*, nous pouvons nous limiter à jouer sur le nombre de *WorkDefinitions* plutôt que sur le nombre de *Process* ;

— un processus père existant, des éléments *resolve* sont dans tous les cas créés pour chaque *WorkDefinition* ;

— l'ajout de séquences entre les activités permet d'ajouter d'autres éléments *resolve* et donc augmenter la complexité du modèle d'entrée et de la transformation.

Il s'agissait surtout d'être en mesure de générer des modèles d'entrée définis simplement, mais parfaitement maîtrisés (dont les éléments sources sont dénombrables de manière fiable) et pouvant atteindre de très grosses tailles (plusieurs millions d'éléments). La figure 8.1 donne la forme générale d'un tel processus donné en entrée ainsi que le réseau de Petri résultant par l'application de la transformation *SimplePDLToPetriNet*.

En fonction des deux paramètres et de la définition du processus, nous pouvons parfaitement dénombrer les éléments constituant le modèle source, ainsi que ceux constituant le résultat. Il est aussi possible de dénombrer précisément les éléments intermédiaires *resolve* créés.

Ainsi, dans notre exemple, pour un processus avec N *WorkDefinitions* ($N > 1$) chaînées, il y a $N - 1$ *WorkSequences*. Les règles de transformation produisant des réseaux de Petri identiques à ceux vus dans le chapitre 7.1 que nous rappelons dans la figure 8.2, nous établissons le tableau 8.2 donnant les règles de production des éléments sources. Dans ce tableau, n et

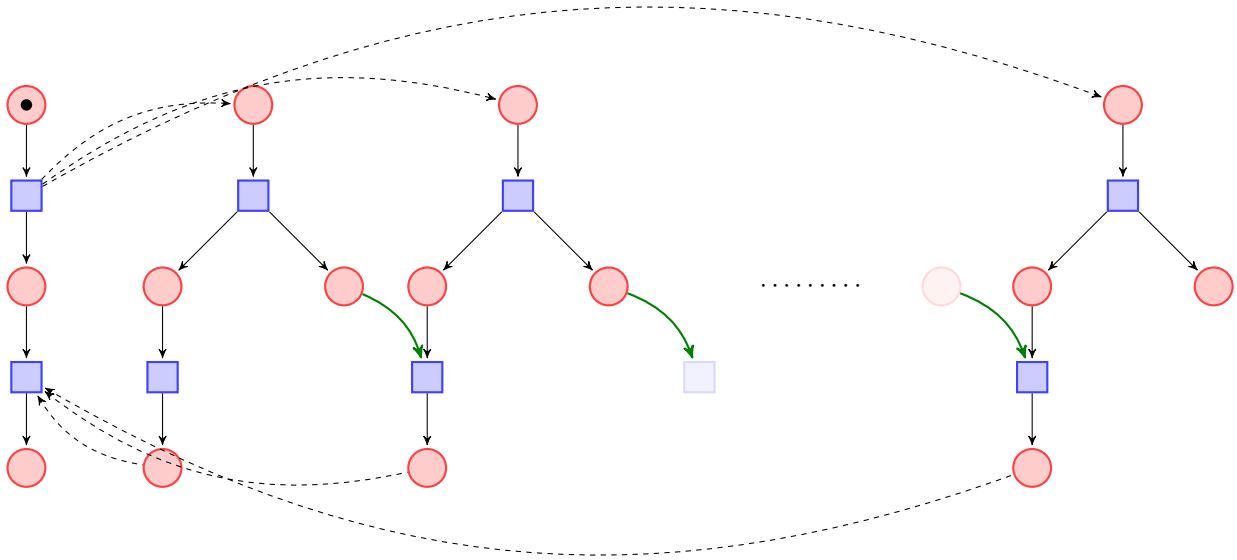


FIGURE 8.1 – Forme générale des modèles d'entrée générés.

r signifient respectivement *normal* et *resolve*, les abréviations Pl, Tr, A, P, WD, WS correspondent respectivement à *Place*, *Transition*, *Arc*, *Process*, *WorkDefinition* et *WorkSequence*. Les trois dernières colonnes du tableau sont les totaux d'éléments *resolve* créés ($T_{resolve}$), d'éléments créés ($T_{intermédiaire}$), et d'éléments dans les versions finales des images des sources après résolution (T_{final}).

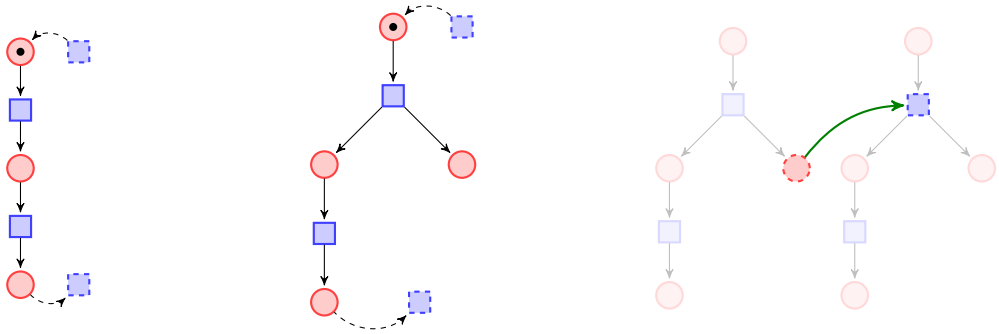


FIGURE 8.2 – Réseaux de Petri images d'un *Process*, d'une *WorkDefinition* et d'une *WorkSequence*.

Compte tenu du fait que nos modèles d'entrée générés n'ont pas de processus hiérarchiques, les arcs de synchronisation ainsi que les éléments *resolve* associés n'existent pas dans l'image du *Process*.

Ainsi, pour un processus composé de N *WorkDefinitions* (avec $N > 1$) on obtient $14N+8$ éléments cibles *normaux* et $4N-2$ éléments *resolve*, comme le résume la table 8.3.

Les valeurs que nous donnons par la suite sont celles d'expériences menées sur un serveur ayant les caractéristiques suivantes :

— système Unix 64 bits ;

Source	Places		Transitions		Arcs	$T_{resolve}$	$T_{intermédiaire}$	T_{final}
	n	r	n	r				
P	3	0	2	0	4	0	9	9
WD	4	0	2	2	7	2	15	13
WS	0	1	0	1	1	2	3	1

Table 8.2 – Dénombrement des éléments cibles créés à partir d'éléments sources.

Places		Transitions		Arcs	$T_{resolve}$	T_{normal}
n	r	n	r			
$4N+3$	$N-1$	$2N+2$	$3N-1$	$8N+3$	$4N-2$	$14N+8$

Table 8.3 – Dénombrement des éléments cibles créés en fonction du nombre de *WorkDefinitions* donné en entrée ($N > 1$).

— RAM : 24 GB ;

— processeurs : 2 × 2,93 GHz Quad-Core Intel Xeon.

Cependant, mis à part pour les transformations à plusieurs millions d'éléments qui demandent des ressources supérieures à celles fournies par un poste de bureau, ces expériences peuvent tout à fait être reproduites sur une machine bien plus modeste.

La table 8.4 donne les temps moyens de la transformation appliquée sur des modèles de tailles différentes. La première colonne donne le nombre d'éléments sources du modèle d'entrée en fonction du paramètre d'entrée N. La deuxième colonne donne le nombre d'éléments constituant le modèle résultant. Les temps moyens sont donnés par les colonnes suivantes, en séparant les deux phases, la dernière colonne donnant les durées totales. Pour donner une idée de la taille des données, la sérialisation `.xmi` d'un modèle à 2 000 000 d'éléments sources (dernière ligne) est un fichier d'environ 320 Mo.

#src (2N)	#tgt	phase 1		phase 2		Total
		temps	%	temps	%	
20	148	16,7ms	97,09	0,5ms	2,91	17,2ms
200	1408	66ms	90,66	6,8ms	9,34	72,8ms
1000	7008	208ms	87,76	29ms	12,24	237ms
2000	14 008	359ms	86,92	54ms	13,08	413ms
4000	28 008	833ms	85,35	143ms	14,65	976ms
6000	42 008	∼1,47s	86,98	223ms	13,02	∼1,69s
8000	56 008	∼2,4s	87,27	342ms	12,73	∼2,75s
10 000	70 008	∼3,61s	88,03	491ms	11,97	4,1s
20 000	140 008	∼13,4s	88,16	∼1,8s	11,84	∼15,2s
40 000	280 008	∼0min 52s	88,14	∼0min 7s	11,86	∼0min 59s
100 000	700 008	∼5min 22s	88,54	∼0min 42s	11,46	∼6min 4s
200 000	1 400 008	∼20min 50s	88,22	∼2min 47s	11,78	∼23min 37s
2 000 000	14 000 008	∼35h 40min	84,87	∼6h 30min	11,78	∼42h 10min

Table 8.4 – Mesures de durées de transformation en fonction des tailles des modèles sources.

La figure 8.3 donne une représentation graphique des temps d'exécution en fonction de la taille du modèle d'entrée. Plus le modèle source est gros, plus la durée de la transformation est élevée. Nous notons que plus le modèle source est important, plus la part de la première phase de la transformation est importante par rapport à la phase de résolution. Cela s'explique par deux facteurs : d'une part par la transformation elle-même qui crée bien plus d'éléments

cibles (*normaux* et *resolve*) qu'elle ne doit en résoudre, d'autre part par les améliorations et optimisations que nous avons apportées au code généré et aux mécanismes développés.

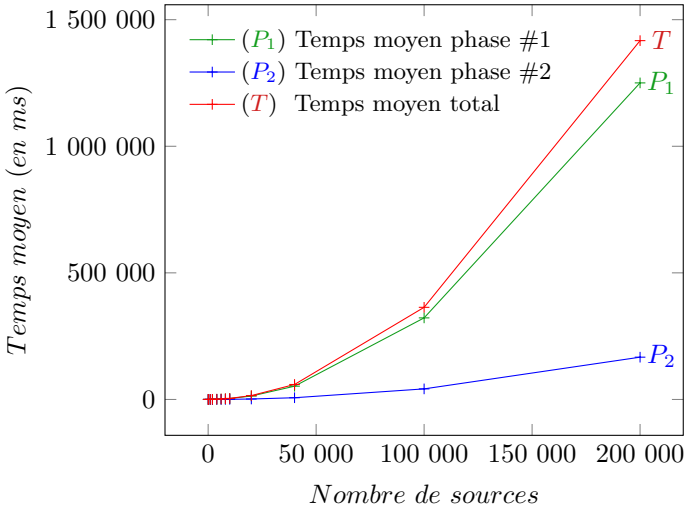


FIGURE 8.3 – Temps moyen de transformation (en ms) en fonction du nombre d'éléments dans le modèle source (phase 1, phase 2, total).

En effet, ces résultats ne sont pas les résultats que nous pourrions obtenir avec la version stable actuelle de nos outils : initialement, la phase de résolution était largement plus consommatrice de ressources et la durée d'une transformation était quasiment celle de la seconde phase (environ 99% du temps d'exécution pour des modèles de taille >100). De plus, dans leur première version stable publiée, outre une certaine lenteur, nos outils consomment une grande quantité de mémoire. Lors des premières expériences, nos transformations étaient beaucoup plus longues à s'exécuter que la même transformation développée en ATL (donnée en C). Nous ne pouvions exécuter une transformation d'un modèle source de 20 000 éléments ou plus, par manque de mémoire (20GB alloués par défaut). De son côté, ATL franchissait ce seuil est bloquait aux alentours de 50 000 éléments sources par manque de mémoire. C'est ce qu'illustre le tableau 8.5 qui donne les temps d'exécution de deux versions de nos outils, que nous comparons à ceux avec ATL. Nous avons donc travaillé en priorité sur l'amélioration du code généré pour la phase de résolution afin de corriger ces lacunes.

Dans un premier temps, constatant que l'essentiel du temps était passé dans l'appel de fonctions du *framework* EMF, nous avons modifié la résolution de liens. Nous nous reposions exclusivement sur les méthodes *emf* gérant la résolution de liens, qui étaient appelées au sein de la stratégie de résolution. Cependant, l'implémentation de cette méthode dans le *framework* EMF n'est pas assez efficace. Nous avons donc implémenté notre propre résolution de liens. Dans un second temps, nous avons continué à optimiser en modifiant notre approche lors de la trace technique des objets créés. Ces améliorations successives ont drastiquement diminué la durée d'exécution de la seconde phase par rapport à la première qui concentre maintenant l'essentiel du temps processeur d'une transformation.

Finalement, nous avons abouti à la version actuelle qui sera intégrée dans la prochaine version stable de Tom. Tout au long de la transformation, nous stockons les éléments tracés servant à la résolution (traçabilité technique) ainsi que les éléments référençant les éléments intermédiaires *resolve* (c'est-à-dire ayant un pointeur vers un élément *resolve*, par exemple un arc image d'une *WorkSequence* dans l'exemple *SimplePDLToPetriNet*).

En début de phase de résolution, nous récupérons l'ensemble des éléments intermédiaires temporaires. Nous le parcourons et pour chaque élément de cet ensemble, nous parcourons l'ensemble des objets le référençant pour leur appliquer la résolution de lien. Avec notre im-

N	#src (2N)	#tgt	Tom		ATL
			v1	v2	Simple
10	20	148	83ms	17ms	-
100	200	1408	638ms	73ms	-
500	1000	7008	∼38s	237ms	-
1000	2000	14 008	∼4min 14s	414ms	-
2000	4000	28 008	∼36min 55s	976ms	-
3000	6000	42 008	∼1h 59min	1,7s	-
4000	8000	56 008	∼4h 24min	2,8s	-
5000	10 000	70 008	∼8h 14min	4,1s	∼28s
10 000	20 000	140 008	N/A	∼0min 15s	∼1min 41s
20 000	40 000	280 008	N/A	∼0min 59s	∼7min 38s
50 000	100 000	700 008	N/A	∼6min 4s	∼1h 39min
100 000	200 000	1 400 008	N/A	∼23min 37s	∼
1 000 000	2 000 000	14 000 008	N/A	∼42h 10min	∼

Table 8.5 – Comparaison des performances entre Tom (première et dernière versions) et ATL.

plémentation, nous parcourons le minimum d'objets possible, ce qui réduit considérablement les parcours (notamment par rapport à EMF). Ces optimisations nous ont apporté un gain conséquent nous permettant de passer d'outils transformant difficilement des petits modèles de 10 000 éléments en des outils capables de transformer des modèles de plusieurs millions d'éléments.

Finalement, ces premiers résultats expérimentaux nous permettent d'exprimer un avis sur nos choix technologiques initiaux, en particulier sur l'utilisation de EMF. Si ce choix paraissait naturel pour pouvoir avoir une couverture initiale maximale de chaînes de développement, d'outils et d'utilisateurs, il paraît beaucoup moins naturel lorsque l'on commence à s'intéresser aux performances. En effet, notre travail de *profiling* et d'optimisation nous a montré que le facteur limitant essentiel résidait dans les appels EMF. Leur limitation puis leur suppression a drastiquement amélioré les performances de nos outils. Dans notre contexte, et pour les tâches demandées, nous n'avions finalement pas un besoin fondamental de toutes les fonctionnalités fournies par EMF. Une version simplifiée et optimisée de la gestion des références inverses suffisait amplement pour notre phase de résolution. Il est donc plus intéressant en termes de performances pures et de mémoire de nous passer de la technologie de support (EMF dans notre implémentation actuelle) pour tout ce qui relève de la mécanique interne de la transformation. De plus, nous affranchir totalement d'une technologie nous permet d'obtenir des outils plus génériques, et donc plus facilement intégrables dans d'autres contextes et chaînes de développement.

Cependant, si le choix initial d'utilisation des services EMF pour le moteur des transformations s'est avéré peu judicieux et a été remis en cause en cours de thèse, il a été fondamental pour avoir une implémentation concrète et un support de travail. De plus, notre travail a mis en avant ce choix peu judicieux pour la mise en œuvre de mécanismes internes de nos transformations, ce qui ne remet pas en cause l'usage d'EMF dans d'autres contextes avec d'autres outils, ni même son usage avec Tom. En effet, au-delà des constructions haut-niveau qui s'appuyaient partiellement sur EMF, subsiste notre outil de génération automatique d'ancrages formels qui permet d'opérer le changement d'espace technologique. Cet outil reste extrêmement intéressant pour faciliter l'écriture des transformations — de manière totalement indépendante du langage de transformation ainsi que de sa mise en œuvre technique — dans un contexte Java-EMF.

8.3 Perspectives

L'ensemble de nos outils peut être actuellement considéré comme un prototype opérationnel. Cependant, dans le cadre d'une utilisation plus large et dans un contexte industriel réel, on pourrait s'orienter vers différents aspects qui ont émergé de ce travail de thèse et des premiers résultats expérimentaux.

Nous avons pu éprouver l'intérêt de notre approche hybride pour développer des transformations de modèles. Cette approche facilite de développement grâce aux outils de génération de code et aux constructions haut-niveau avec une forte expressivité, tout en conservant l'environnement global existant. Il reste donc possible d'utiliser les spécificités du langage généraliste au sein duquel nous nous intégrons pour certaines tâches tout en manipulant des termes dans une autre partie de l'application. Le travail d'adaptation que l'utilisateur doit fournir est de fait plus faible, d'où une plus grande efficacité.

Notre approche a aussi l'avantage de perturber au minimum la chaîne de développement et le choix de l'adopter n'est pas irréversible ou coûteux dans le sens où chaque transformation développée peut être remplacée par sa version dans le langage hôte (ou écrite avec un autre outil) sans changer le reste du logiciel. Dans cette optique d'intégration non-intrusive dans des environnements existants, une voie serait de travailler à la généralisation de nos outils afin d'étendre l'approche à d'autres langages et technologies. Par exemple, un premier pas dans cette direction pourrait être de se pencher sur l'utilisation de Kevoree ainsi que sur l'extension au langage Ada (des travaux sur le sujet ont déjà été entamés).

L'utilisation de constructions haut-niveau et d'outils de génération permet à l'utilisateur de se concentrer sur le cœur de l'application et de déléguer à un outil automatique les tâches répétitives sujettes à erreurs pour un humain. Cela contribue à augmenter la confiance dans le logiciel développé.

Un autre axe de travail consisterait à réfléchir sur les stratégies que nous utilisons pour transformer des modèles. Nous nous sommes aperçus que certaines relations étaient souvent vues comme intéressantes (composition), mais que dans certaines situations, d'autres relations pouvaient être le centre de la transformation. Travailler sur la paramétrisation des stratégies de réécriture par des types de relation à suivre pourrait être une voie à explorer.

Un autre aspect intéressant que d'autres outils couvrent est la transformation de modèles multiples : actuellement notre approche considère un seul modèle source et un seul modèle cible. Cependant, on pourrait gagner en expressivité en permettant d'écrire des transformations à plusieurs sources et cibles.

Bien que les outils soient complètement opérationnels, une optimisation serait probablement nécessaire pour un usage plus large dans un cadre industriel, tant du point de vue de leurs performances que de leur utilisabilité. Nos travaux d'amélioration des performances par réingénierie de la phase de résolution constituent une première étape qui sera concrétisée dans une prochaine *release* du projet Tom.

Du point de vue de la modularité et de la réutilisation du code, les stratégies de réécriture nous fournissent un socle intéressant de par leur modularité intrinsèque. Nos expérimentations sur le sujet ont abouti à un prototype de nos outils avec phase de résolution modulaire. Cependant, pour des raisons de confort d'utilisation pour le développeur, nous avons choisi de ne pas intégrer cette fonctionnalité par défaut dans la version stable actuelle. Nous avons cependant une base de travail pour réfléchir à cet aspect. Il faut toutefois noter que les évolutions de nos outils dues à l'amélioration de leurs performances risquent d'entrer en concurrence avec l'approche adoptée dans notre prototype modulaire. De fait, l'aspect performance est actuellement privilégié.

Nous avons aussi pu apporter une traçabilité au sein d'un langage généraliste tel que Java. Il s'agissait d'apporter une aide substantielle au processus de qualification d'un logiciel. L'aspect traçabilité des transformations semble être un thème particulièrement prometteur. En effet, les industries développant du logiciel critique ont un besoin accru de traçabilité pour qualifier et certifier leurs outils. L'adoption de l'ingénierie dirigée par les modèles a entraîné l'usage de nouvelles méthodes et de nouveaux outils tandis que les contraintes légales

subsistent. Partant de ce constat, les travaux sur la traçabilité à des fins de validation du logiciel offrent des perspectives intéressantes. Dans l'implémentation actuelle de nos travaux, les constructions pour la traçabilité ne sont pas distinctes. Une première étape serait de distinguer clairement (par deux constructions distinctes du langage) la traçabilité utilisée à des fins purement techniques (usage pour la phase de résolution) et la traçabilité métier. Notre approche de trace à la demande permet déjà de limiter la taille des traces et donc de les rendre plus exploitables *a posteriori*. Se pose aussi la question de l'exploitation du modèle de lien, de manière automatique ou non dans le processus de qualification.

Conclusion

La qualification d'outil est une étape fondamentale dans le processus de certification du logiciel. L'objectif principal de cette thèse était de contribuer à l'élaboration de méthodes et outils pour améliorer la confiance dans les outils intégrés dans les chaînes de développement critiques. Le langage Tom développé dans l'équipe a été le support de ces travaux.

Contributions

Une méthode de transformation de modèles par réécriture

Dans cette thèse, nous avons présenté une méthode pour transformer des modèles en nous appuyant sur des stratégies de réécriture. Cette méthode nous permet d'exprimer aisément une transformation à partir de règles, sans aucune notion d'ordre d'application.En nous plaçant dans le cadre du langage Tom, nous avons fait le choix d'être à la frontière entre les langages dédiés et les langages généralistes, afin d'en tirer le meilleur des deux mondes. Nous avons donc proposé un îlot formel dédié aux transformations de modèles qui s'intègre au sein du langage généraliste Java. Du fait du fonctionnement de notre méthode hybride, nos transformations sont modulaires. Cette qualité est fondamentale en ingénierie du logiciel, que ce soit pour la maintenance ou la création de nouveaux outils.

Une représentation de modèles sous la forme de termes algébriques

Pour transformer des modèles selon notre méthode hybride, il est nécessaire d'opérer un changement d'espace technologique entre le monde des modèles et celui des termes. Nous avons donc établi une représentation des modèles sous la forme de termes algébriques. Cette dernière repose sur l'obtention d'un arbre de recouvrement du modèle par la relation de composition. Un outil implémente cette représentation par la génération de la signature algébrique correspondant à un métamodèle donné. Cela permet d'instancier des modèles et de les manipuler en tant que termes algébriques.

Une traçabilité des transformations

L'exigence de traçabilité qu'impose le processus de qualification nous a conduits à proposer un îlot formel permettant de lier un élément source à des éléments cibles. Cette construction dédiée permet d'ajouter une traçabilité au sein d'une transformation écrite dans un langage généraliste tel que Java.

Des outils accessibles pour améliorer la confiance dans le logiciel

Tout au long de ce travail, nous nous sommes appliqués à mettre en œuvre des outils implémentant nos travaux. Nos choix ont aussi été guidés par notre souci d'accessibilité et de diffusion. C'est pourquoi nous nous sommes appuyés sur les langages et *frameworks* très utilisés et ayant déjà percé dans l'industrie. Si certains choix ont pu être remis en cause suite à nos évaluations et aux tests de performance, il n'en demeure pas moins qu'ils nous ont

été très utiles pour obtenir un premier prototype opérationnel. Les outils développés doivent faciliter le développement logiciel en limitant le code écrit manuellement et en maximisant le code généré.

De plus, dans cette optique d'amélioration de la confiance dans le logiciel, il est très important de diffuser des outils de production dont le code source est librement consultable, modifiable et rediffusable afin de pouvoir les vérifier, corriger, améliorer et utiliser. Par conséquent, tous nos outils sont diffusés sous licence libre.

Perspectives

Les perspectives qu'ouvre ce travail sont multiples, les chaînes de développement de systèmes critiques ayant de besoins forts pour la qualification d'outils. Différents travaux peuvent être entrepris pour étendre et prolonger les contributions de cette thèse, que ce soit au niveau de la représentation et du parcours de modèles, de l'expression et de la traçabilité des transformations ou de la généralisation des méthodes et outils.

Stratégies paramétrées

L'une des pistes de recherche consiste à travailler sur les stratégies. Dans le chapitre 7, nous avons mis en avant un cas d'étude qui ne permet pas de donner la pleine mesure de nos outils. Cela s'explique par le principe de notre représentation de modèles sous la forme de termes ainsi que par leur parcours. Nous construisons un arbre de recouvrement par la relation de composition pour obtenir une vue arborescente du modèle. Ensuite, les stratégies les parcourent en suivant ces liens de composition. Bien que ce type d'utilisation couvre la majeure partie des cas, il existe des situations où la relation de composition n'est pas la relation d'intérêt de l'utilisateur, comme nous l'avons montré. Afin de remédier à ce problème, il faudrait pouvoir définir une alternative pour les stratégies de parcours en offrant la possibilité de les paramétrer par les types de liens à suivre. Un second prolongement serait alors de pouvoir paramétrer localement un parcours, en fonction d'un contexte donné.

Modularité des transformations et réutilisation

Un aspect récurrent en génie logiciel est la modularité et la réutilisation du code. Du fait de notre approche construite sur les stratégies de réécriture, nous offrons déjà une certaine modularité des transformations et permettons la réutilisation du code de chaque transformation élémentaire. Cependant, les *définitions* écrites pour une transformation donnée peuvent rarement être réutilisées directement, sans aucune modification. Un axe de travail serait donc de réfléchir à l'expression des *définitions* les rendant plus génériques. Ensuite, le prototype modulaire de la phase de résolution de notre approche pourrait être affiné, étendu et simplifié afin d'accompagner chaque *définition* par une *stratégie de résolution élémentaire*.

Généralisation et ouverture à d'autres technologies

Nous nous sommes appuyés sur la technologie EMF et le langage Java pour développer des outils et tester nos travaux. À terme, il serait intéressant de généraliser notre méthode de représentation et de transformation de modèles en ouvrant nos outils à d'autres langages généralistes et à d'autres technologies. Nous pensons notamment au langage Ada pour lequel nous avons mené quelques expérimentations et dont un *backend* est maintenant présent dans Tom. Les stratégies ont été implémentées et peuvent être utilisées dans un programme Ada. Ce langage étant utilisé pour le développement de systèmes temps réel et embarqués pour son haut niveau de fiabilité et de sécurité, nous pourrions nous concentre sur ce langage. L'une des difficultés de la gestion de Ada réside dans l'absence de *garbage collector*. On ne peut donc aisément implémenter toutes les fonctionnalités de Tom (notamment le partage maximal) qui

existent avec le langage Java. La généralisation de nos outils nous permettrait aussi de nous ouvrir à d'autres technologies que EMF. Nous pensons notamment à Kevoree qui semble être un bon candidat, tant du point de vue fonctionnalités que performances.

Traçabilité : extension et usage

Nous avons fourni une traçabilité qui pourrait être étendue, d'une part à des transformations comprenant des sources multiples, d'autre part à des objectifs de détection et récupération d'erreurs. Cela demanderait de réfléchir à l'évolution du métamodèle de trace. Ensuite, cette traçabilité fournie par Tom pourrait être généralisée afin de dépasser le cadre de l'îlot formel dédié aux transformations de modèles. Enfin, de manière pragmatique et à court terme, nous envisageons de modifier le langage pour séparer la construction de traçabilité en deux constructions distinctes, l'une dédiée à la traçabilité interne, l'autre à la traçabilité de spécification.

Annexe A

Étude de cas : Transformation SimplePDLToPetriNet

A.1 Code de la transformation *SimplePDLToPetriNet*

Ce code est accessible sur le dépôt officiel du projet Tom : `https://gforge.inria.fr/scm/?group_id=78`. Pour plus d'informations sur le sujet, le lecteur pourra aussi se référer à la documentation en ligne accessible sur cette page : `http://tom.loria.fr/wiki/index.php5/Documentation:Playing_with_EMF`.

```
1  import org.eclipse.emf.common.util.*;
2  import org.eclipse.emf.ecore.*;
3  import org.eclipse.emf.ecore.util.ECrossReferenceAdapter;
4  import org.eclipse.emf.ecore.xmi.*;
5  import org.eclipse.emf.ecore.xmi.impl.*;
6
7  import SimplePDLSemantics.DDMMSimplePDL.*;
8  import petrinetsemantics.DDMMPetriNet.*;
9
10 import SimplePDLSemantics.EDMMSimplePDL.*;
11 import petrinetsemantics.EDMMPetriNet.*;
12 import SimplePDLSemantics.SDMMSimplePDL.*;
13 import petrinetsemantics.SDMMPetriNet.*;
14 import SimplePDLSemantics.TM3SimplePDL.*;
15 import petrinetsemantics.TM3PetriNet.*;
16
17 import java.util.*;
18 import java.util.concurrent.ConcurrentMap;
19 import java.util.concurrent.ConcurrentHashMap;
20 import java.io.File;
21 import java.io.Writer;
22 import java.io.BufferedWriter;
23 import java.io.OutputStreamWriter;
24 import java.io.FileOutputStream;
25 import java.io.FileInputStream;
26
27 import tom.library.utils.ReferenceClass;
28 import tom.library.utils.LinkClass;
29 import tom.library.sl.*;
30 import tom.library.emf.*;
31
32 public class SimplePDLToPetriNet {
33
34   %include{ sl.tom }
35   %include{ LinkClass.tom }
36   %include{ emf/ecore.tom }
37
38   %include{ mappings/DDMMPetriNetPackage.tom }
39   %include{ mappings/DDMMSimplePDLPackage.tom }
40
41   %include{ mappings/EDMMPetriNetPackage.tom }
42   %include{ mappings/EDMMSimplePDLPackage.tom }
43   %include{ mappings/SDMMPetriNetPackage.tom }
44   %include{ mappings/SDMMSimplePDLPackage.tom }
```

```
45    %include{ mappings/TM3PetriNetPackage.tom }
46    %include{ mappings/TM3SimplePDLPackage.tom }
47
48    %typeterm SimplePDLToPetriNet { implement { SimplePDLToPetriNet }}
49
50    private static Writer writer;
51    private static PetriNet pn = null;
52    private static LinkClass tom__linkClass;
53
54    public SimplePDLToPetriNet() {
55      this.tom__linkClass = new LinkClass();
56    }
57
58
59    %transformation SimplePDLToPetriNet(tom__linkClass:LinkClass,pn:PetriNet) : "metamodels/SimplePDLSemantics_updated.ecore" -> "met
60
61      definition P2PN traversal `TopDown(P2PN(tom__linkClass,pn)) {
62        p@Process[name=name] -> {
63          Place p_ready = `Place(name + "_ready", pn,ArcEList(), ArcEList(), 1);
64          Place p_running = `Place(name + "_running", pn,ArcEList(), ArcEList(), 0);
65          Place p_finished = `Place(name + "_finished", pn,ArcEList(), ArcEList(), 0);
66          String n1 = `name+"_start";
67          %tracelink(t_start:Transition, `Transition(n1, pn,ArcEList(), ArcEList(), 1, 1));
68          n1 = `name+"_finish";
69          %tracelink(t_finish:Transition, `Transition(n1, pn,ArcEList(), ArcEList(), 1, 1));
70
71          `Arc(t_start, p_ready, pn,ArcKindnormal(), 1);
72          `Arc(p_running, t_start, pn,ArcKindnormal(), 1);
73          `Arc(t_finish, p_running, pn,ArcKindnormal(), 1);
74          `Arc(p_finished, t_finish, pn,ArcKindnormal(), 1);
75
76          WorkDefinition from = `p.getFrom();
77          if (from!=null) {
78            Transition source = %resolve(from:WorkDefinition,t_start:Transition);
79            source.setNet(pn);
80            Arc tmpZoomIn = `Arc(p_ready,source,pn,ArcKindnormal(), 1);
81
82            Transition target = %resolve(from:WorkDefinition,t_finish:Transition);
83            target.setNet(pn);
84            Arc tmpZoomOut = `Arc(target,p_finished,pn,ArcKindread_arc(), 1);
85          }
86        }
87      }
88
89      definition WD2PN traversal `TopDown(WD2PN(tom__linkClass,pn)) {
90        wd@WorkDefinition[name=name] -> {
91          //System.out.println("Je suis un A");
92          Place p_ready = `Place(name + "_ready", pn,ArcEList(), ArcEList(), 1);
93          String n1 = `name+"_started";
94          %tracelink(p_started:Place, `Place(n1, pn,ArcEList(), ArcEList(), 0));
95          Place p_running = `Place(name+"_running", pn,ArcEList(), ArcEList(), 0);
96          n1 = `name+"_finished";
97          %tracelink(p_finished:Place, `Place(n1, pn,ArcEList(), ArcEList(), 0));
98          n1 = `name+"_start";
99          %tracelink(t_start:Transition, `Transition(n1, pn,ArcEList(), ArcEList(), 1, 1));
100         n1 = `name+"_finish";
101         %tracelink(t_finish:Transition, `Transition(n1, pn,ArcEList(), ArcEList(), 1, 1));
102
103         `Arc(t_start, p_ready, pn,ArcKindnormal(), 1);
104         `Arc(p_started, t_start, pn,ArcKindnormal(), 1);
105         `Arc(p_running, t_start, pn,ArcKindnormal(), 1);
106         `Arc(t_finish, p_running, pn,ArcKindnormal(), 1);
107         `Arc(p_finished, t_finish, pn,ArcKindnormal(), 1);
108
109         SimplePDLSemantics.DDMMSimplePDL.Process parent = `wd.getParent();
110         Transition source = %resolve(parent:Process,t_start:Transition);
111         source.setNet(pn);
112         Arc tmpDistribute = `Arc(p_ready,source,pn,ArcKindnormal(), 1);
113
114         Transition target = %resolve(parent:Process,t_finish:Transition);
115         target.setNet(pn);
116         Arc tmpRejoin = `Arc(target,p_finished,pn,ArcKindread_arc(), 1);
117       }
118     }
119
120     definition WS2PN traversal `TopDown(WS2PN(tom__linkClass,pn)) {
121       ws@WorkSequence[predecessor=p,successor=s,linkType=linkType] -> {
122         Place source= null;
123         Transition target= null;
124         WorkDefinition pre = `p;
```

```
125         WorkDefinition suc = ‘s;
126         %match(linkType) {
127           (WorkSequenceTypefinishToFinish|WorkSequenceTypefinishToStart)[] -> {
128                   source = %resolve(pre:WorkDefinition,p_finished:Place);
129           }
130           (WorkSequenceTypestartToStart|WorkSequenceTypestartToFinish)[] -> {
131                   source = %resolve(pre:WorkDefinition,p_started:Place);
132           }
133
134           (WorkSequenceTypefinishToStart|WorkSequenceTypestartToStart)[] -> {
135                   target = %resolve(suc:WorkDefinition,t_start:Transition);
136           }
137           (WorkSequenceTypestartToFinish|WorkSequenceTypefinishToFinish)[] -> {
138                   target = %resolve(suc:WorkDefinition,t_finish:Transition);
139           }
140         }
141         source.setNet(pn);
142         target.setNet(pn);
143
144         Arc wsImage = ‘Arc(target,source, pn,ArcKindread_arc(), 1);
145       }
146     }
147   }
148
149   public static void main(String[] args) {
150     System.out.println("\nStarting...\n");
151
152     XMIResourceImpl resource = new XMIResourceImpl();
153     SimplePDLSemantics.DDMMSimplePDL.Process p_root;
154     Map opts = new HashMap();
155     opts.put(XMIResource.OPTION_SCHEMA_LOCATION, java.lang.Boolean.TRUE);
156
157     if (args.length>0) {
158       DDMMSimplePDLPackage packageInstance = DDMMSimplePDLPackage.eINSTANCE;
159       File input = new File(args[0]);
160       try {
161         resource.load(new FileInputStream(input),opts);
162       } catch (Exception e) {
163         e.printStackTrace();
164       }
165       p_root = (SimplePDLSemantics.DDMMSimplePDL.Process) resource.getContents().get(0);
166     } else {
167       System.out.println("No model instance given in argument. Using default hardcoded model.");
168       WorkDefinition wd1 = ‘WorkDefinition(null,WorkSequenceEList(),WorkSequenceEList(),"A",null);
169       WorkDefinition wd2 = ‘WorkDefinition(null,WorkSequenceEList(),WorkSequenceEList(),"B",null);
170       WorkDefinition wd3 = ‘WorkDefinition(null,WorkSequenceEList(),WorkSequenceEList(),"C",null);
171       WorkDefinition wd4 = ‘WorkDefinition(null,WorkSequenceEList(),WorkSequenceEList(),"D",null);
172       WorkSequence ws1 = ‘WorkSequence(null,WorkSequenceTypestartToStart(),wd1,wd2);
173       WorkSequence ws2 = ‘WorkSequence(null,WorkSequenceTypestartToFinish(),wd3,wd4);
174
175       p_root = ‘Process("root",ProcessElementEList(wd1,wd2,ws1),null);
176       SimplePDLSemantics.DDMMSimplePDL.Process p_child = ‘Process("child",ProcessElementEList(wd3,wd4,ws2), wd2);
177
178       wd1.setParent(p_root);
179       wd2.setParent(p_root);
180       wd2.setProcess(p_child);
181
182       wd3.setParent(p_child);
183       wd4.setParent(p_child);
184
185       ws1.setParent(p_root);
186       ws2.setParent(p_child);
187     }
188     SimplePDLToPetriNet translator = new SimplePDLToPetriNet();
189
190     try {
191       translator.pn = ‘PetriNet(NodeEList(),ArcEList(),"main");
192
193       //System.out.println("Initial Petri net");
194       //‘Sequence(TopDown(PrintTransition()),TopDown(PrintPlace())).visit(translator.pn, new EcoreContainmentIntrospector());
195
196       /*//transformer is equivalent to:
197       Strategy transformer = ‘Sequence(
198           TopDown(Process2PetriNet(translator)),
199           TopDown(WorkDefinition2PetriNet(translator)),
200           TopDown(WorkSequence2PetriNet(translator))
201        );
202        */
203
204       //NOTE: force the user to give the link as first parameter, and target
```

```
205       //model as second one
206       Strategy transformer = `SimplePDLToPetriNet(translator.tom__linkClass,translator.pn);
207       transformer.visit(p_root, new EcoreContainmentIntrospector());
208       `TopDown(tom__StratResolve_SimplePDLToPetriNet(translator.tom__linkClass,translator.pn)).visit(translator.pn, new EcoreContai
209
210
211       //for generation of textual Petri nets usable as input for TINA
212       String outputName = "resultingPetri.net";
213       writer = new BufferedWriter(new OutputStreamWriter(new
214            FileOutputStream(new File(outputName))));
215
216       System.out.println("\nResult");
217       `Sequence(TopDown(PrintTransition()),TopDown(PrintPlace())).visit(translator.pn,
218          new EcoreContainmentIntrospector());
219
220       System.out.println("\nFinish to generate "+outputName+" file, usable as input for TINA");
221       writer.flush();
222       writer.close();
223       System.out.println("done.");
224
225     } catch(VisitFailure e) {
226       System.out.println("strategy fail!");
227     } catch(java.io.FileNotFoundException e) {
228       System.out.println("Cannot create Petri net output file.");
229     } catch (java.io.IOException e) {
230       System.out.println("Petri net save failed!");
231     }
232   }
233
234   %strategy PrintArc() extends Identity() {
235     visit Arc {
236       Arc[source=node1, target=node2] -> {
237         System.out.println(`node1.getName() + " -> " + `node2.getName());
238       }
239     }
240   }
241
242   %strategy PrintTransition() extends Identity() {
243     visit Transition {
244       tr@ResolveWorkDefinitionTransition[tom_resolve_element_attribute_name=name] -> {
245         System.out.println("tr resolve " + `name);
246         return `tr;
247       }
248
249       ptr@ResolveProcessTransition[tom_resolve_element_attribute_name=name] -> {
250         System.out.println("tr process resolve " + `name);
251         return `ptr;
252       }
253
254       Transition[name=name,incomings=sources,outgoings=targets] -> {
255         String s = " ";
256         String t = " ";
257         %match {
258          ArcEList(_*,Arc[kind=k,weight=w,source=node],_*) << sources && Place[name=placename]<< node -> {
259            s += `placename + ((`k==`ArcKindread_arc())?"?":"*") + `w + " ";
260          }
261          ArcEList(_*,Arc[kind=k,weight=w,target=node],_*) << targets && Place[name=placename]<< node -> {
262            t += `placename + ((`k==`ArcKindread_arc())?"?":"*") + `w + " ";
263          }
264         }
265         multiPrint("tr " + `name + s + "->" + t);
266       }
267
268
269     }
270   }
271
272   %strategy PrintPlace() extends Identity() {
273     visit Place {
274       pl@ResolveWorkDefinitionPlace[tom_resolve_element_attribute_name=name] -> {
275         System.out.println("pl resolve " + `name);
276         return `pl;
277       }
278
279       Place[name=name,initialMarking=w] && w!=0 -> {
280         multiPrint("pl " + `name + " " + "(" + `w + ")");
281       }
282
283     }
284   }
```

```
285
286   public static void multiPrint(String s) {
287     System.out.println(s);
288     try {
289       writer.write(s+"\n");
290     } catch (java.io.IOException e) {
291       System.out.println("Petri net save failed!");
292     }
293   }
294
295 }
```

A.2 Modèle source

Cette transformation a été testée avec de nombreux modèles. Le modèle donné par le listing 1 est le modèle donné en exemple dans la section 7.1 du chapitre 7. Il s'agit aussi du modèle par défaut de la transformation dans le cas où aucun modèle n'est passé en paramètre (c'est en fait le modèle équivalent directement construit en Tom sans chargement de fichier).

```
<?xml version="1.0" encoding="ASCII"?>
<simplepdl.ddmm:Process xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:simplepdl.ddmm="simplepdl.ddmm"
    xsi:schemaLocation="simplepdl.ddmm SimplePDLSemantics.ecore#//DDMMSimplePDL" name="root">
  <processElements xsi:type="simplepdl.ddmm:WorkDefinition" linksToSuccessors="//@processElements.2" name="A"/>
  <processElements xsi:type="simplepdl.ddmm:WorkDefinition" linksToPredecessors="//@processElements.2" name="B">
    <process name="child">
      <processElements xsi:type="simplepdl.ddmm:WorkDefinition"
          linksToSuccessors="//@processElements.1/@process/@processElements.2" name="C"/>
      <processElements xsi:type="simplepdl.ddmm:WorkDefinition"
          linksToPredecessors="//@processElements.1/@process/@processElements.2" name="D"/>
      <processElements xsi:type="simplepdl.ddmm:WorkSequence" linkType="startToFinish"
          predecessor="//@processElements.1/@process/@processElements.0"
          successor="//@processElements.1/@process/@processElements.1"/>
    </process>
  </processElements>
  <processElements xsi:type="simplepdl.ddmm:WorkSequence"
      predecessor="//@processElements.0" successor="//@processElements.1"/>
</simplepdl.ddmm:Process>
```

Listing 1 – Modèle source de l'exemple pour la transformation SimplePDLToPetriNet (format `.xmi`)

A.3 Modèle résultant

Le modèle source donné précédemment produit le modèle cible suivant 2, qui est en fait le réseau de Petri résultant de la transformation, sérialisé sous la forme d'un fichier au format compatible avec l'entrée du *model-checker* TINA [BRV04] (il est aussi affiché directement sur la sortie standard) :

```
tr root_start root_ready*1 -> root_running*1 A_ready*1 B_ready*1
tr root_finish root_running*1 A_finished?1 B_finished?1 -> root_finished*1
tr child_start child_ready*1 -> child_running*1 C_ready*1 D_ready*1
tr child_finish child_running*1 C_finished?1 D_finished?1 -> child_finished*1
tr B_start B_ready*1 A_started?1 -> B_started*1 B_running*1 child_ready*1
tr B_finish B_running*1 child_finished?1 -> B_finished*1
tr A_start A_ready*1 -> A_started*1 A_running*1
tr A_finish A_running*1 -> A_finished*1
tr C_start C_ready*1 -> C_started*1 C_running*1
tr C_finish C_running*1 -> C_finished*1
tr D_start D_ready*1 -> D_started*1 D_running*1
tr D_finish D_running*1 C_started?1 -> D_finished*1
pl root_ready (1)
pl child_ready (1)
pl A_ready (1)
pl B_ready (1)
pl C_ready (1)
pl D_ready (1)
```

Listing 2 – Modèle cible résultant de la transformation SimplePDLToPetriNet (format lisible par TINA)

A.4 Vérification du résultat

La boîte à outils TINA [56] (version 3.1.0) a été utilisée pour vérifier le modèle résultant de la transformation de modèle *SimplePDLToPetriNet*.

Procédure

Le résultat de la transformation est un réseau de Petri dans un format lisible par TINA (fichier `result.net`. La procédure ci-après permet de reproduire l'expérience :

1. Visualisation du réseau de Petri :
   `~/tina-3.1.0/bin/nd result.net` (sauvegarde au format `.ndr`)
2. Construction du graphe d'atteignabilité :
   `~/tina-3.1.0/bin/tina -C result.ndr result.ktz` (le format `.ktz` est un format binaire propriétaire)
3. Vérification de propriétés :
   `~/tina-3.1.0/bin/selt  result.ktz prop.ltl >> result.selt` (le fichier `prop.ltl` décrit ci-après contient les formules et les propriétés à vérifier)

Formule et propriétés

```
1  op finished = T /\ A_finished /\ B_finished /\ C_finished /\ D_finished;
2
3  [] (finished => dead);
4  [] (dead => finished);
5  [] <> dead ;
6  - <> finished;
```

Listing 3 – Formule et propriétés LTL à vérifier sur le modèle résulatnt de la transformation SimplePDLToPetriNet

Le listing 3 contient une formule et quatre propriétés exprimées en LTL afin de pouvoir être utilisées avec TINA. Nous les décrivons ci-après, dans l'ordre :

ligne 1 : Il s'agit de la formule définissant l'opérateur *finished* comme $\top \wedge A_{finished} \wedge B_{finished} \wedge C_{finished} \wedge D_{finished}$. Cela signifie que l'on considère le processus complet terminé lorsque toutes les activités le composant sont terminées (lorsqu'il y a un jeton dans chaque place $finished$ des activités A, B, C et D).

ligne 3 : $\Box(finished \Rightarrow dead)$. Cette première propriété signifie que tout processus dans son état final est terminé.

ligne 4 : $\Box(dead \Rightarrow finished)$. Cette deuxième propriété peut être traduite par « tout processus terminé est dans son état final » (correction partielle).

ligne 5 : $\Box\Diamond dead$. Cette troisième propriété est la propriété de terminaison qui assure que « toute exécution se termine ».

ligne 6 : $\neg\Diamond finished$. Cette quatrième propriété peut être traduite par « l'état final n'est jamais atteint » et revient à vérifier la propriété de consistance faible (existence d'au moins une exécution du processus).

Note : `dead` est une propriété prédéfinie vraie uniquement sur les états de blocage.

56. `http://projects.laas.fr/tina`

Résultat fourni par selt

Nous avons vérifié les propriétés données précédemment sur le réseau de Petri généré en utilisant `selt` (partie de TINA). Le résultat est le suivant :

```
Selt version 3.1.0 -- 01/07/13 -- LAAS/CNRS
ktz loaded, 12523 states, 60082 transitions
0.119s
operator finished : prop
0.000s
FALSE
  state 0: A_ready B_ready C_ready D_ready child_ready root_ready
  -A_start ... (preserving T)->
  state 31: A_finished*2 A_started*2 B_finished*2 B_started*2 C_finished*4 C_started*4 D_finished*4
            D_started*4 child_finished*3 root_running
  -root_finish ... (preserving - dead /\ D_finished /\ A_finished /\ B_finished /\ C_finished)->
  state 33: L.dead A_finished*2 A_started*2 B_finished*2 B_started*2 C_finished*4 C_started*4 D_finished*4
            D_started*4 child_finished*3 root_finished
  [accepting all]
0.001s
TRUE
0.141s
TRUE
0.363s
FALSE
  state 0: A_ready B_ready C_ready D_ready child_ready root_ready
  -A_start ... (preserving T)->
  state 32: L.dead A_finished*2 A_started*2 B_finished*2 B_started*2 C_finished*4 C_started*4 D_finished*4
            D_started*4 child_finished*3 root_finished
  -L.deadlock ... (preserving C_finished /\ D_finished /\ A_finished /\ B_finished)->
  state 33: L.dead A_finished*2 A_started*2 B_finished*2 B_started*2 C_finished*4 C_started*4 D_finished*4
            D_started*4 child_finished*3 root_finished
  [accepting all]
0.001s
```

Listing 4 – Résultat fourni par `selt` pour les propriétés et le réseau de Petri fourni dans notre exemple

Ainsi, les propriétés 2 et 3 sont vérifiées. En revanche, les propriétés 1 et 4 sont fausses, `selt` exhibe donc un contre-exemple pour chacune. Dans le cas de la quatrième propriété, une évaluation à *False* permet d'affirmer que le processus est faiblement consistant (le contre-exemple donne une exécution correcte du processus).

Annexe B

Étude de cas : aplatissement d'une hiérarchie de classes

B.1 Code de la transformation

B.1.1 Version 1 : transformation en Java+EMF

```
import org.eclipse.emf.common.util.*;
import org.eclipse.emf.ecore.*;
import org.eclipse.emf.ecore.xmi.*;
import org.eclipse.emf.ecore.xmi.impl.*;

import classflattening.*;
import classflattening.impl.*;

import java.util.*;
import java.io.File;
import java.io.FileInputStream;

/* VERSION0: Version without any Tom code -> pure Java EMF */
public class V1_notom_UMLClassesFlattening {
  private static classflattening.VirtualRoot virtR = null;

  public static classflattening.VirtualRoot v0_processClass(classflattening.VirtualRoot root) {
    org.eclipse.emf.common.util.EList<classflattening.Class> newChildren =
              new org.eclipse.emf.common.util.BasicEList<classflattening.Class>();
    for(classflattening.Class cl : root.getChildren()) {
      if(cl.getSubclass().isEmpty()) {
        newChildren.add(notom_flattening(cl));
      }
    }
    classflattening.VirtualRoot result = (classflattening.VirtualRoot)classflattening.ClassflatteningFactory.eINSTANCE.create(
                                    (EClass)classflattening.ClassflatteningPackage.eINSTANCE.getEClassifier("VirtualRoot"));
    result.eSet(result.eClass().getEStructuralFeature("children"),(Object)newChildren);
    return result;
  }

  public static classflattening.Class notom_flattening(classflattening.Class toFlatten) {
    classflattening.Class parent = toFlatten.getSuperclass();
    if(parent==null) {
      return toFlatten;
    } else {
      classflattening.Class flattenedParent = notom_flattening(parent);
      org.eclipse.emf.common.util.EList<classflattening.Attribute> head =
        toFlatten.getAttributes();
      head.addAll(flattenedParent.getAttributes());
      classflattening.Class result = (classflattening.Class)classflattening.ClassflatteningFactory.eINSTANCE.create(
                                  (EClass)classflattening.ClassflatteningPackage.eINSTANCE.getEClassifier("Class"));

      result.eSet(result.eClass().getEStructuralFeature("name"),(Object)flattenedParent.getName()+toFlatten.getName());
      result.eSet(result.eClass().getEStructuralFeature("attributes"),(Object)head);
      result.eSet(result.eClass().getEStructuralFeature("superclass"),(Object)flattenedParent.getSuperclass());
      result.eSet(result.eClass().getEStructuralFeature("subclass"),(Object)(
                                   new org.eclipse.emf.common.util.BasicEList<classflattening.Class>()) );
      result.eSet(result.eClass().getEStructuralFeature("root"),(Object)virtR);
```

```
49        return result;
50      }
51    }
52
53    public static void main(String[] args) {
54      System.out.println("\nStarting...\n");
55
56      XMIResourceImpl resource = new XMIResourceImpl();
57      classflattening.VirtualRoot source_root = null;
58      Map opts = new HashMap();
59      opts.put(XMIResource.OPTION_SCHEMA_LOCATION, java.lang.Boolean.TRUE);
60
61      if (args.length>0) {
62        //EMF style source model creation (loading .xmi file)
63        classflattening.ClassflatteningPackage packageInstance = classflattening.ClassflatteningPackage.eINSTANCE;
64        File input = new File(args[0]);
65        try {
66          resource.load(new FileInputStream(input),opts);
67        } catch (Exception e) {
68          e.printStackTrace();
69        }
70        source_root = (classflattening.VirtualRoot) resource.getContents().get(0);
71      } else {
72        System.out.println("No model instance given in argument. Bye bye.");
73        return;
74      }
75      V1_notom_UMLClassesFlattening translator = new V1_notom_UMLClassesFlattening();
76      try {
77        //init virtual root
78        translator.virtR = null;
79        System.out.println("\nSource:");
80        printer(source_root);
81
82        System.out.print("\nFlattening");
83        //VERSION0
84        System.out.println(" (VERSION0) ...");
85        translator.virtR = v0_processClass(source_root);
86
87        System.out.println("\nResult:");
88        printer(translator.virtR);
89      } catch(Exception e) {
90        System.out.println("Ooops!");
91      }
92    }
93
94    public static void printer(classflattening.VirtualRoot root) {
95      for(classflattening.Class cl : root.getChildren()) {
96        classflattening.Class sup = cl.getSuperclass();
97        String strSup = (sup!=null)?"\n sup="+sup.getName():"";
98        String strAttr = "";
99        org.eclipse.emf.common.util.EList<classflattening.Attribute> attr = cl.getAttributes();
100       if(!(attr.isEmpty())) {
101
102         classflattening.Attribute head = attr.get(0);
103         strAttr += "\n attr="+head.getName()+":"+head.getType().getName();
104         if(attr.size()>1) {
105           java.util.List<classflattening.Attribute> tail = attr.subList(1,attr.size());
106           for(classflattening.Attribute at : tail) {
107             strAttr += ", "+at.getName()+":"+at.getType().getName();
108           }
109         }
110       }
111       System.out.println("Class: "+cl.getName()+strAttr+strSup/*+strSub*/);
112     }
113   }
114
115 }
```

B.1.2 Version 2 : transformation en Tom+Java simple (+EMF)

Cette version est légèrement plus longue que la version précédente étant donné que nous avons intégré la création d'un modèle par défaut dans le cas où aucun modèle source n'est donné en argument du programme.

```
1  import org.eclipse.emf.common.util.*;
2  import org.eclipse.emf.ecore.*;
```

```
3   import org.eclipse.emf.ecore.util.ECrossReferenceAdapter;
4   import org.eclipse.emf.ecore.xmi.*;
5   import org.eclipse.emf.ecore.xmi.impl.*;
6
7   import classflattening.*;
8   import classflattening.impl.*;
9
10  import java.util.*;
11  import java.io.File;
12  import java.io.FileInputStream;
13
14  import tom.library.emf.*;
15
16  /* VERSION1: recursive version, no %strategy, no %transformation */
17  public class V2_nostrat_UMLClassesFlattening {
18
19    %include{ emf/ecore.tom }
20    %include{ mappings/ClassflatteningPackage.tom }
21
22    %typeterm V2_nostrat_UMLClassesFlattening { implement { V2_nostrat_UMLClassesFlattening }}
23    private static classflattening.VirtualRoot virtR = null;
24
25    public static classflattening.VirtualRoot v1_processClass(classflattening.VirtualRoot root) {
26      org.eclipse.emf.common.util.EList<classflattening.Class> children = root.getChildren();
27      org.eclipse.emf.common.util.EList<classflattening.Class> newChildren = `cfClassEList();
28      %match(children) {
29        cfClassEList(_*,cl@cfClass(_,_,_,cfClassEList(),_),_*) -> {
30          newChildren = `cfClassEList(flattening(cl),newChildren*);
31        }
32      }
33      return `cfVirtualRoot(newChildren);
34    }
35
36    public static classflattening.Class flattening(classflattening.Class toFlatten) {
37      classflattening.Class parent = toFlatten.getSuperclass();
38      if(parent==null) {
39        return toFlatten;
40      } else {
41        classflattening.Class flattenedParent = flattening(parent);
42
43        org.eclipse.emf.common.util.EList<classflattening.Attribute> head = toFlatten.getAttributes();
44        head.addAll(flattenedParent.getAttributes());
45        classflattening.Class result = `cfClass(flattenedParent.getName() +
46            toFlatten.getName(), head, flattenedParent.getSuperclass(),
47            cfClassEList(), null);
48        return result;
49      }
50    }
51
52    public static void main(String[] args) {
53      System.out.println("\nStarting...\n");
54
55      XMIResourceImpl resource = new XMIResourceImpl();
56      classflattening.VirtualRoot source_root = `cfVirtualRoot(cfClassEList());
57      Map opts = new HashMap();
58      opts.put(XMIResource.OPTION_SCHEMA_LOCATION, java.lang.Boolean.TRUE);
59
60      if (args.length>0) {
61        //EMF style source model creation (loading .xmi file)
62        classflattening.ClassflatteningPackage packageInstance = classflattening.ClassflatteningPackage.eINSTANCE;
63        File input = new File(args[0]);
64        try {
65          resource.load(new FileInputStream(input),opts);
66        } catch (Exception e) {
67          e.printStackTrace();
68        }
69        source_root = (classflattening.VirtualRoot) resource.getContents().get(0);
70      } else {
71        System.out.println("No model instance given in argument. Using default hardcoded model.");
72        //Tom style source model creation
73        classflattening.Class clC =
74          `cfClass("C",cfAttributeEList(),null,cfClassEList(), null);
75        clC = `cfClass("C",cfAttributeEList(cfAttribute("attrC", clC)),null,cfClassEList(), null);
76
77        classflattening.Class clD =
78          `cfClass("D",cfAttributeEList(cfAttribute("attrD", clC)), null,
79              cfClassEList(), null);
80
81        classflattening.Class clA = `cfClass("A",cfAttributeEList(), clD,
82            cfClassEList(), null);
```

```
83
84        classflattening.Class clB =
85          `cfClass("B",cfAttributeEList(cfAttribute("attrB1", clC),
86                cfAttribute("attrB2", clC)), clA, cfClassEList(), null);
87
88        clD = `cfClass("D",cfAttributeEList(cfAttribute("attrD", clC)), null,
89            cfClassEList(clA), null);
90        clA = `cfClass("A",cfAttributeEList(), clD, cfClassEList(clB), null);
91
92        source_root = `cfVirtualRoot(cfClassEList(clC,clD,clA,clB));
93        clA.setRoot(source_root);
94        clB.setRoot(source_root);
95        clC.setRoot(source_root);
96        clD.setRoot(source_root);
97      }
98      V2_nostrat_UMLClassesFlattening translator = new V2_nostrat_UMLClassesFlattening();
99      try {
100       //init virtual root
101       translator.virtR = `cfVirtualRoot(cfClassEList());
102
103       System.out.println("\nSource:");
104       `TopDown(Print()).visit(source_root, new EcoreContainmentIntrospector());
105
106       System.out.print("\nFlattening");
107       //VERSION1
108       System.out.println(" (VERSION1) ...");
109       translator.virtR = v1_processClass(source_root);
110
111       System.out.println("\nResult:");
112       `TopDown(Print()).visit(translator.virtR, new EcoreContainmentIntrospector());
113
114     } catch(VisitFailure e) {
115       System.out.println("strategy fail!");
116     }
117   }
118
119   %strategy Print() extends Identity() {
120     visit cfClass {
121       cl@cfClass(n,attr,sup,sub,root) -> {
122         String strSup = (`sup!=null)?"\n sup="+`sup.getName():"";
123         String strAttr = "";
124         if (!(`attr.isEmpty())) {
125           %match(`attr) {
126             cfAttributeEList(head,tail*) -> {
127               strAttr = "\n attr="+`head.getName()+":"+`head.getType().getName();
128               %match(`tail) {
129                 cfAttributeEList(_*,a,_*) -> {
130                   strAttr += ", "+`a.getName()+":"+`a.getType().getName();
131                 }
132               }
133             }
134           }
135         }
136         System.out.println("Class: "+`n+strAttr+strSup/*+strSub*/);
137       }
138     }
139   }
140
141 }
```

B.1.3 Version 3 : transformation en Tom+Java avec stratégies (+EMF)

Comme pour la version précédente, nous avons intégré la création d'un modèle par défaut dans le cas où aucun modèle source n'est donné en paramètre, d'où sa taille de code légèrement supérieure à la première version.

```
1  import org.eclipse.emf.common.util.*;
2  import org.eclipse.emf.ecore.*;
3  import org.eclipse.emf.ecore.util.ECrossReferenceAdapter;
4  import org.eclipse.emf.ecore.xmi.*;
5  import org.eclipse.emf.ecore.xmi.impl.*;
6
7  import classflattening.*;
8  import classflattening.impl.*;
9
10 import java.util.*;
```

```
11  import java.io.File;
12  import java.io.FileInputStream;
13
14  import tom.library.sl.*;
15  import tom.library.emf.*;
16
17  /* VERSION3: %strategy version, no %transformation */
18  public class V3_stratnotransfo_UMLClassesFlattening {
19
20    %include{ sl.tom }
21    %include{ emf/ecore.tom }
22
23    %include{ mappings/ClassflatteningPackage.tom }
24
25    %typeterm V3_stratnotransfo_UMLClassesFlattening { implement { V3_stratnotransfo_UMLClassesFlattening }}
26
27    private static classflattening.VirtualRoot virtR = null;
28
29    public static classflattening.Class flattening(classflattening.Class toFlatten) {
30      classflattening.Class parent = toFlatten.getSuperclass();
31      if(parent==null) {
32        return toFlatten;
33      } else {
34        classflattening.Class flattenedParent = flattening(parent);
35
36        org.eclipse.emf.common.util.EList<classflattening.Attribute> head = toFlatten.getAttributes();
37        head.addAll(flattenedParent.getAttributes());
38        classflattening.Class result = `cfClass(flattenedParent.getName() +
39            toFlatten.getName(), head, flattenedParent.getSuperclass(),
40            cfClassEList(), null);
41        return result;
42      }
43    }
44
45    %strategy FlatteningStrat(translator:V3_stratnotransfo_UMLClassesFlattening) extends Identity() {
46      visit cfVirtualRoot {
47        cfVirtualRoot(cfClassEList(_*,cl@cfClass(n,_,_,cfClassEList(),_),_*)) -> {
48          org.eclipse.emf.common.util.EList<classflattening.Class> newChildren = translator.virtR.getChildren();
49          translator.virtR = `cfVirtualRoot(cfClassEList(flattening(cl),newChildren*));
50        }
51      }
52    }
53
54    public static void main(String[] args) {
55      System.out.println("\nStarting...\n");
56
57      XMIResourceImpl resource = new XMIResourceImpl();
58      classflattening.VirtualRoot source_root = `cfVirtualRoot(cfClassEList());
59      Map opts = new HashMap();
60      opts.put(XMIResource.OPTION_SCHEMA_LOCATION, java.lang.Boolean.TRUE);
61
62      if (args.length>0) {
63        //EMF style source model creation (loading .xmi file)
64        classflattening.ClassflatteningPackage packageInstance = classflattening.ClassflatteningPackage.eINSTANCE;
65        File input = new File(args[0]);
66        try {
67          resource.load(new FileInputStream(input),opts);
68        } catch (Exception e) {
69          e.printStackTrace();
70        }
71        source_root = (classflattening.VirtualRoot) resource.getContents().get(0);
72      } else {
73        System.out.println("No model instance given in argument. Using default hardcoded model.");
74        //Tom style source model creation
75        classflattening.Class clC =
76          `cfClass("C",cfAttributeEList(),null,cfClassEList(), null);
77        clC = `cfClass("C",cfAttributeEList(cfAttribute("attrC", clC)),null,cfClassEList(), null);
78
79        classflattening.Class clD =
80          `cfClass("D",cfAttributeEList(cfAttribute("attrD", clC)), null,
81             cfClassEList(), null);
82
83        classflattening.Class clA = `cfClass("A",cfAttributeEList(), clD,
84            cfClassEList(), null);
85
86        classflattening.Class clB =
87          `cfClass("B",cfAttributeEList(cfAttribute("attrB1", clC),
88                cfAttribute("attrB2", clC)), clA, cfClassEList(), null);
89
90        clD = `cfClass("D",cfAttributeEList(cfAttribute("attrD", clC)), null,
```

```
91          cfClassEList(clA), null);
92      clA = `cfClass("A",cfAttributeEList(), clD, cfClassEList(clB), null);
93
94      source_root = `cfVirtualRoot(cfClassEList(clC,clD,clA,clB));
95      clA.setRoot(source_root);
96      clB.setRoot(source_root);
97      clC.setRoot(source_root);
98      clD.setRoot(source_root);
99    }
100   V3_stratnotransfo_UMLClassesFlattening translator = new V3_stratnotransfo_UMLClassesFlattening();
101   try {
102     //init virtual root
103     translator.virtR = `cfVirtualRoot(cfClassEList());
104
105     System.out.println("\nSource:");
106     `TopDown(Print()).visit(source_root, new EcoreContainmentIntrospector());
107
108     System.out.print("\nFlattening");
109     //VERSION3
110     System.out.println(" (VERSION2) ...");
111     Strategy transformer = `BottomUp(FlatteningStrat(translator));
112     transformer.visit(source_root, new EcoreContainmentIntrospector());
113
114     System.out.println("\nResult:");
115     `TopDown(Print()).visit(translator.virtR, new EcoreContainmentIntrospector());
116   } catch(VisitFailure e) {
117     System.out.println("strategy fail!");
118   }
119 }
120
121 %strategy Print() extends Identity() {
122   visit cfClass {
123     cl@cfClass(n,attr,sup,sub,root) -> {
124       String strSup = (`sup!=null)?"\n sup="+`sup.getName():"";
125       String strAttr = "";
126       if(!(`attr.isEmpty())) {
127         %match(`attr) {
128           cfAttributeEList(head,tail*) -> {
129             strAttr = "\n attr="+`head.getName()+":"+`head.getType().getName();
130             %match(`tail) {
131               cfAttributeEList(_*,a,_*) -> {
132                 strAttr += ", "+`a.getName()+":"+`a.getType().getName();
133               }
134             }
135           }
136         }
137       }
138       System.out.println("Class: "+`n+strAttr+strSup/*+strSub*/);
139     }
140   }
141 }
142
143 }
```

B.1.4 Version 4 : transformation en **Tom+Java** avec les nouvelles constructions (+**EMF**)

```
1  import org.eclipse.emf.common.util.*;
2  import org.eclipse.emf.ecore.*;
3  import org.eclipse.emf.ecore.util.ECrossReferenceAdapter;
4  import org.eclipse.emf.ecore.xmi.*;
5  import org.eclipse.emf.ecore.xmi.impl.*;
6
7  import classflattening.*;
8  import classflattening.impl.*;
9
10 import java.util.*;
11 import java.io.File;
12 import java.io.FileInputStream;
13
14 import tom.library.sl.*;
15 import tom.library.emf.*;
16 import tom.library.utils.LinkClass;
17
18 public class V4_transfo_UMLClassesFlattening {
19
```

```
20    %include{ sl.tom }
21    %include{ LinkClass.tom }
22    %include{ emf/ecore.tom }
23
24    %include{ mappings/ClassflatteningPackage.tom }
25
26    %typeterm UMLClassesFlattening { implement { UMLClassesFlattening }}
27
28    private static classflattening.VirtualRoot virtR = null;
29    private static LinkClass tom__linkClass;
30
31    public UMLClassesFlattening() {
32      this.tom__linkClass = new LinkClass();
33    }
34
35    /* This function is called by the transformation, like in previous
36     * implementations */
37    public static classflattening.Class flattening(classflattening.Class toFlatten) {
38      classflattening.Class parent = toFlatten.getSuperclass();
39      if(parent==null) {
40        return toFlatten;
41      } else {
42        classflattening.Class flattenedParent = flattening(parent);
43
44        org.eclipse.emf.common.util.EList<classflattening.Attribute> head = toFlatten.getAttributes();
45        head.addAll(flattenedParent.getAttributes());
46        classflattening.Class result = `cfClass(flattenedParent.getName() +
47            toFlatten.getName(), head, flattenedParent.getSuperclass(),
48            cfClassEList(), null);
49        return result;
50      }
51    }
52
53    /* VERSION4: %transformation version */
54      %transformation FlatteningTrans(tom__linkClass:LinkClass,virtR:cfVirtualRoot) : "metamodels/Class.ecore" -> "metamodels/Class.ecore" {
55      definition DEF1 traversal `BottomUp(DEF1(tom__linkClass, virtR)) {
56        cfVirtualRoot(cfClassEList(_*,cl@cfClass(n,_,_,cfClassEList(),_),_*)) -> {
57          org.eclipse.emf.common.util.EList<classflattening.Class> newChildren = virtR.getChildren();
58          return `cfVirtualRoot(cfClassEList(flattening(cl),newChildren*));
59        }
60      }
61    }
62
63
64    /* NOTE: no resolve element needed */
65    public static void main(String[] args) {
66      System.out.println("\nStarting...\n");
67
68      XMIResourceImpl resource = new XMIResourceImpl();
69      classflattening.VirtualRoot source_root = `cfVirtualRoot(cfClassEList());
70      Map opts = new HashMap();
71      opts.put(XMIResource.OPTION_SCHEMA_LOCATION, java.lang.Boolean.TRUE);
72
73      if (args.length>0) {
74        //EMF style source model creation (loading .xmi file)
75        classflattening.ClassflatteningPackage packageInstance = classflattening.ClassflatteningPackage.eINSTANCE;
76        File input = new File(args[0]);
77        try {
78          resource.load(new FileInputStream(input),opts);
79        } catch (Exception e) {
80          e.printStackTrace();
81        }
82        source_root = (classflattening.VirtualRoot) resource.getContents().get(0);
83      } else {
84        System.out.println("No model instance given in argument. Using default hardcoded model.");
85        //Tom style source model creation
86        classflattening.Class clC =
87          `cfClass("C",cfAttributeEList(),null,cfClassEList(), null);
88        clC = `cfClass("C",cfAttributeEList(cfAttribute("attrC", clC)),null,cfClassEList(), null);
89
90        classflattening.Class clD =
91        `cfClass("D",cfAttributeEList(cfAttribute("attrD", clC)), null,
92            cfClassEList(), null);
93
94        classflattening.Class clA = `cfClass("A",cfAttributeEList(), clD,
95            cfClassEList(), null);
96
97        classflattening.Class clB =
98        `cfClass("B",cfAttributeEList(cfAttribute("attrB1", clC),
99              cfAttribute("attrB2", clC)), clA, cfClassEList(), null);
```

```
100
101       clD = `cfClass("D",cfAttributeEList(cfAttribute("attrD", clC)), null,
102           cfClassEList(clA), null);
103       clA = `cfClass("A",cfAttributeEList(), clD, cfClassEList(clB), null);
104
105       source_root = `cfVirtualRoot(cfClassEList(clC,clD,clA,clB));
106       clA.setRoot(source_root);
107       clB.setRoot(source_root);
108       clC.setRoot(source_root);
109       clD.setRoot(source_root);
110
111     }
112     UMLClassesFlattening translator = new UMLClassesFlattening();
113
114     try {
115       //init virtual root
116       translator.virtR = `cfVirtualRoot(cfClassEList());
117
118       System.out.println("\nSource:");
119       `TopDown(Print()).visit(source_root, new EcoreContainmentIntrospector());
120
121       System.out.print("\nFlattening");
122       //VERSION4
123       System.out.println(" (VERSION3) ...");
124       Strategy transformer = `TopDown(FlatteningTrans(translator.tom__linkClass,translator.virtR));
125       translator.virtR = transformer.visit(source_root, new EcoreContainmentIntrospector());
126       /* As we do not use any %resolve, there is no Resolve strategy
127        * generation, and no call of it. The following line is only the
128        * instruction we would write if there were a Resolve strategy:
129          `TopDown(tom__StratResolve_FlatteningTrans(translator.tom__linkClass,
130            translator.virtR)).visit(translator.virtR, new
131            EcoreContainmentIntrospector());
132        */
133
134       System.out.println("\nResult:");
135       `TopDown(Print()).visit(translator.virtR, new EcoreContainmentIntrospector());
136     } catch(VisitFailure e) {
137       System.out.println("strategy fail!");
138     }
139   }
140
141   %strategy Print() extends Identity() {
142     visit cfClass {
143       cl@cfClass(n,attr,sup,sub,root) -> {
144         String strSup = (`sup!=null)?"\n sup="+`sup.getName():"";
145         String strAttr = "";
146         if(!(`attr.isEmpty())) {
147           %match(`attr) {
148             cfAttributeEList(head,tail*) -> {
149               strAttr = "\n attr="+`head.getName()+":"+`head.getType().getName();
150               %match(`tail) {
151                 cfAttributeEList(_*,a,_*) -> {
152                   strAttr += ", "+`a.getName()+":"+`a.getType().getName();
153                 }
154               }
155             }
156           }
157         }
158         System.out.println("Class: "+`n+strAttr+strSup/*+strSub*/);
159       }
160     }
161   }
162
163 }
```

Annexe C

Implémentation ATL de SimplePDLToPetriNet

```
-- @path simplepdl=/BenchSimplePDL2PetriNet/SimplePDLSemantics_updated.ecore
-- @path petrinet=/BenchSimplePDL2PetriNet/PetriNetSemantics_updated.ecore

module simplepdltopetrinet;
create OUT: petrinet from IN: simplepdl;

-- Process -> PetriNet
rule Process2PetriNet {
        from
                p: simplepdl!Process
        to
                --Petri net creation (sufficient for flat exemple, should be a bit modified for the nested version)
                pn: petrinet!PetriNet (
                        name <- p.name,
                        nodes <- simplepdl!WorkDefinition.allInstances()->collect(pl | thisModule.resolveTemp(pl,'p_ready')) -- WD2PN nodes
                            .union(simplepdl!WorkDefinition.allInstances()->collect(tr | thisModule.resolveTemp(tr,'t_start')))
                                .union(simplepdl!WorkDefinition.allInstances()->collect(pl | thisModule.resolveTemp(pl,'p_started')))
                                .union(simplepdl!WorkDefinition.allInstances()->collect(pl | thisModule.resolveTemp(pl,'p_running')))
                                .union(simplepdl!WorkDefinition.allInstances()->collect(tr | thisModule.resolveTemp(tr,'t_finish')))
                                .union(simplepdl!WorkDefinition.allInstances()->collect(pl | thisModule.resolveTemp(pl,'p_finished')))
                                .append(p_ready) -- P2PN nodes
                                .append(t_start)
                                .append(p_running)
                                .append(t_finish)
                                .append(p_finished)
                                ,
                        arcs <- simplepdl!WorkDefinition.allInstances()->collect(a | thisModule.resolveTemp(a,'a_ready2start')) -- WD2PN arcs
                                .union(simplepdl!WorkDefinition.allInstances()->collect(a | thisModule.resolveTemp(a,'a_start2started')))
                                .union(simplepdl!WorkDefinition.allInstances()->collect(a | thisModule.resolveTemp(a,'a_start2running')))
                                .union(simplepdl!WorkDefinition.allInstances()->collect(a | thisModule.resolveTemp(a,'a_running2finish')))
                                .union(simplepdl!WorkDefinition.allInstances()->collect(a | thisModule.resolveTemp(a,'a_finish2finished')))
                                .union(simplepdl!WorkDefinition.allInstances()->collect(a | thisModule.resolveTemp(a,'a_distribute')))
                                .union(simplepdl!WorkDefinition.allInstances()->collect(a | thisModule.resolveTemp(a,'a_rejoin')))
                                .union(simplepdl!WorkSequence.allInstances()->collect(a | thisModule.resolveTemp(a,'a_ws'))) -- WS2PN arc
                                .append(a_ready2start) -- P2PN arcs
                                .append(a_start2running)
                                .append(a_running2finish)
                                .append(a_finish2finished)
                ),
                -- Process PLACES
                p_ready: petrinet!Place (
                        name <- p.name + '_ready',
                        initialMarking <- 1
                ),
                p_running: petrinet!Place (
                        name <- p.name + '_running',
                        initialMarking <- 0
                ),
                p_finished: petrinet!Place (
                        name <- p.name + '_finished',
                        initialMarking <- 0
                ),
                -- Process TRANSITIONS
                t_start: petrinet!Transition (
```

```
55                      name <- p.name + '_start',
56                      min_time <- 0,
57                      max_time <- 0
58              ),
59              t_finish: petrinet!Transition (
60                      name <- p.name + '_finish',
61                      min_time <- 0,
62                      max_time <- 0
63              ),
64              -- Process ARCS
65              a_ready2start: petrinet!Arc (
66                      source <- p_ready,
67                      target <- t_start,
68                      weight <- 1,
69                      kind <- #normal
70              ),
71              a_start2running: petrinet!Arc (
72                      source <- t_start,
73                      target <- p_running,
74                      weight <- 1,
75                      kind <- #normal
76              ),
77              a_running2finish: petrinet!Arc (
78                      source <- p_running,
79                      target <- t_finish,
80                      weight <- 1,
81                      kind <- #normal
82              ),
83              a_finish2finished: petrinet!Arc (
84                      source <- t_finish,
85                      target <- p_finished,
86                      weight <- 1,
87                      kind <- #normal
88              )
89              -- nested case
90              --if (p.from!=null)
91              --then
92              --ain: petrinet!Arc (
93              --      source <- thisModule.resolveTemp(p.from, 't_start'),
94              --      target <- p_ready,
95              --      weight <- 1,
96              --      kind <- #read_arc
97              --),
98              --aout: petrinet!Arc (
99              --      source <- p_finished,
100             --      target <- thisModule.resolveTemp(p.from, 't_finish'),
101             --      weight <- 1,
102             --      kind <- #read_arc
103             --)
104             --else
105             --endif
106 }
107
108 -- WorkDefinition -> PetriNet
109 rule WorkDefinition2PetriNet {
110         from
111             wd: simplepdl!WorkDefinition
112         to
113             -- WorkDefinition PLACES
114             p_ready: petrinet!Place (
115                     name <- wd.name + '_ready',
116                     initialMarking <- 1
117             ),
118             p_started: petrinet!Place (
119                     name <- wd.name + '_started',
120                     initialMarking <- 0
121             ),
122             -- place that allows to memorize a task has been started
123             p_running: petrinet!Place (
124                     name <- wd.name + '_running',
125                     initialMarking <- 0
126             ),
127             p_finished: petrinet!Place (
128                     name <- wd.name + '_finished',
129                     initialMarking <- 0
130             ),
131             -- WorkDefinition TRANSITIONS
132             t_start: petrinet!Transition (
133                     name <- wd.name + '_start',
134                     min_time <- 0,
```

```
135                     max_time <- 0
136             ),
137             t_finish: petrinet!Transition (
138                     name <- wd.name + '_finish',
139                     min_time <- 0,
140                     max_time <- 0
141             ),
142             -- WorkDefinition ARCS
143             a_ready2start: petrinet!Arc (
144                     source <- p_ready,
145                     target <- t_start,
146                     weight <- 1,
147                     kind <- #normal
148             ),
149             a_start2started: petrinet!Arc (
150                     source <- t_start,
151                     target <- p_started,
152                     weight <- 1,
153                     kind <- #normal
154             ),
155             a_start2running: petrinet!Arc (
156                     source <- t_start,
157                     target <- p_running,
158                     weight <- 1,
159                     kind <- #normal
160             ),
161             a_running2finish: petrinet!Arc (
162                     source <- p_running,
163                     target <- t_finish,
164                     weight <- 1,
165                     kind <- #normal
166             ),
167             a_finish2finished: petrinet!Arc (
168                     source <- t_finish,
169                     target <- p_finished,
170                     weight <- 1,
171                     kind <- #normal
172             ),
173             -- synchonization ARCS (case of a process described by activities)
174             a_distribute: petrinet!Arc (
175                     source <- thisModule.resolveTemp(wd.parent, 't_start'),
176                     target <- p_ready,
177                     weight <- 1,
178                     kind <- #normal
179             ),
180             a_rejoin: petrinet!Arc(
181                     source <- p_finished,
182                     target <- thisModule.resolveTemp(wd.parent, 't_finish'),
183                     weight <- 1,
184                     kind <- #read_arc
185             )
186 }
187
188 -- WorkSequence -> PetriNet
189 rule WorkSequence2PetriNet {
190         from
191             ws: simplepdl!WorkSequence
192         to
193             -- WorkSequence ARC
194             a_ws: petrinet!Arc (
195                     source <- if (ws.linkType = #finishToStart or ws.linkType = #finishToFinish)
196                                     then
197                                     thisModule.resolveTemp(ws.predecessor, 'p_finished')
198                             else
199                                     thisModule.resolveTemp(ws.predecessor, 'p_started')
200                             endif,
201                     target <- if (ws.linkType = #startToFinish or ws.linkType = #finishToFinish)
202                                     then
203                                     thisModule.resolveTemp(ws.successor, 't_finish')
204                             else
205                                     thisModule.resolveTemp(ws.successor, 't_start')
206                             endif,
207                     weight <- 1,
208                     kind <- #read_arc
209             )
210 }
```

Glossaire

AADL : Architecture Analysis and Design Language, précédemment Avionics Architecture Description Language.
ATL : ATLAS Transformation Language.
CMOF : Complete Meta-Object Facility.
CIM : Computation Independent Model.
DSL : Domain Specific Language.
DSML : Domain Specific Modeling Language.
EBNF : Extended Backus-Naur Form.
EMF : Eclipse Modeling Framework.
EMOF : Essential Meta-Object Facility.
GPL : General Purpose Language.
IDM : Ingénierie Dirigée par les Modèles.
MDA : Model Driven Architecture.
MDD : Model Driven Development.
MDE : Model Driven Engineering.
MOF : Meta-Object Facility.
OCL : Object Constraint Language.
OLAP : OnLine Analytical Processing.
OMG : Object Management Group.
PIM : Platform Independent Model.
PSM : Platform Specific Model.
QVT : Query/View/Transformation.
SPEM : Software Process Engineering Metamodel.
SQL : Structured Query Language.
UML : Unified Modeling Language.
XMI : XML Model Interchange.
XML : eXtensible Markup Language.
XSLT : Extensible Stylesheet Language Transformations.

Bibliographie

[ABB+12] Ali Afroozeh, Jean-Christophe Bach, Mark Brand, Adrian Johnstone, Maarten Manders, Pierre-Etienne Moreau, and Elizabeth Scott. Island grammar-based parsing using gll and tom. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 224–243. Springer Berlin Heidelberg, Dresden, Germany, 2012.

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin : Advanced concepts and tools for in-place emf model transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *MoDELS (1)*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010. 36

[AC93] Andrew W Appel and Marcelo JR Conçalves. Hash-consing garbage collection. Technical report, Princeton University, Department of Computer Science, February 1993. 14

[AI04] Freddy Allilaire and Tarik Idrissi. Adt : Eclipse development tools for atl. In *Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA-2)*, pages 171–178. Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK, 2004. 37

[AK03] C. Atkinson and T. Kuhne. Model-driven development : a metamodeling foundation. *Software, IEEE*, 20(5) :36–41, Sept 2003. 27

[AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. Moflon : A standard-compliant metamodeling framework with graph transformations. In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 2006. 36

[AKS03] Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph transformations on domain-specific models. *Journal on Software and Systems Modeling*, 2003. 37

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained : Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004. 40

[Bac14] Jean-Christophe Bach. Une approche hybride GPL-DSL pour transformer des modèles. *Technique et Science Informatiques*, 33(3) :175–201, 2014.

[Bal09] Emilie Balland. *Conception d'un langage dédié à l'analyse et la transformation de programmes*. These, Université Henri Poincaré - Nancy I, March 2009. 20

[BB04] Jean Bézivin and Jean-Pierre Briot. Sur les principes de base de l'ingénierie des modèles. *L' Objet (Paris)*, 10(4) :147–157, 2004. fre. 27

[BB+13] Jean-Christophe Bach, Émilie Balland, , Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, Marc Pantel, François Prugniel, Antoine Reilles, and Cláudia Tavares. *Documentation of Tom 2.10*, March 2013. 14, 35

[BBB+09] Jean-Christophe Bach, Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom manual. Rapport technique, PAREO - INRIA Lorraine - LORIA - INRIA - CNRS : UMR7503 - Université Henri Poincaré - Nancy I - Université Nancy II - Institut National Polytechnique de Lorraine, 2009. 14

[BBI+04] Grady Booch, Alan W. Brown, Sridhar Iyengar, James Rumbaugh, and Bran Selic. An MDA Manifesto. *Business Process Trends/MDA Journal*, May 2004. 30

[BBK+07] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom : piggybacking rewriting on java. In *Proceedings of the 18th international conference on Term rewriting and applications*, RTA'07, pages 36–47, Berlin, Heidelberg, 2007. Springer-Verlag. 7, 13, 35

[BC04] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development – Coq'Art :the calculus of inductive constructions*. springer, 2004. 41

[BCF+10] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0 : An XML Query Language (Second Edition)*. W3C, December 2010. 34

[BCMP12] Jean-Christophe Bach, Xavier Crégut, Pierre-Etienne Moreau, and Marc Pantel. Model transformations with tom. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 4 :1–4 :9, New York, NY, USA, 3 2012. ACM. 24, 67, 94, 96

[BDH+01] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment : a Component-Based Language Development Environment. In Reinhard Wilhelm, editor, *CC'01 : Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001. 35

[BDJ+03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the atl model transformation language : Transforming xslt into xquery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003. 37

[Bec03] Kent Beck. *Test-driven development : by example*. Addison-Wesley Professional, 2003. 40

[BFS+06] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations : an algorithm and a tool. In *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, pages 85–94, Nov 2006. 40

[BG01] J. Bezivin and O. Gerbe. Towards a precise definition of the omg/mda framework. In *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273 – 280, nov. 2001. 27

[BJRV05] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the large and modeling in the small. In Uwe Assmann, Mehmet Aksit, and Arend Rensink, editors, *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer Berlin Heidelberg, 2005. 37

[BKdV03] Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term rewriting systems*. Cambridge University Press, 2003. 12

[BKK+96] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN : A logical framework based on computational systems. In J. Meseguer, editor, *WRLA'96 : Proceedings of the 1st International Workshop on Rewriting Logic and its Applications*, volume 4, 1996. 13

[BKK+98] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of elan. *Electr. Notes Theor. Comput. Sci.*, 15 :55–70, 1998. 13, 19

[BKM06] Emilie Balland, Claude Kirchner, and Pierre-Etienne Moreau. Formal islands. In *Proceedings of the 11th international conference on Algebraic Methodology and Software Technology*, AMAST'06, pages 51–65. Springer-Verlag, 2006. 13

[BM05] A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '05, pages 1052–1059, Washington, DC, USA, 2005. IEEE Computer Society. 41

[BM09] Artur Boronat and José Meseguer. MOMENT2 : EMF Model Transformations in Maude. In Antonio Vallecillo and Goiuria Sagardui, editors, *JISBD*, pages 178–179, 2009. 36

[BMP12a] Jean-Christophe Bach, Pierre-Etienne Moreau, and Marc Pantel. EMF Models Transformations with Tom. poster, 2012.

[BMP12b] Jean-Christophe Bach, Pierre-Etienne Moreau, and Marc Pantel. Tom-based tools to transform EMF models in avionics context. In *ITSLE*, Dresden, Germany, 2012.

[BMR08] Emilie Balland, Pierre-Etienne Moreau, and Antoine Reilles. Rewriting strategies in java. *Electr. Notes Theor. Comput. Sci.*, 219 :97–111, 2008. 13, 19, 20

[BMR12] Emilie Balland, Pierre-Etienne Moreau, and Antoine Reilles. Effective strategic programming for java developers. *Software : Practice and Experience*, 44(2) :129–162, 2012. 13, 19, 20

[BNvBK07] Daniel Balasubramanian, Anantha Narayanan, Christopher van Buskirk, and Gabor Karsai. The graph rewriting and transformation language : GReAT. *Electronic Communications of the EASST*, 1, 2007. 37

[BÖ10] Artur Boronat and PeterCsaba Ölveczky. Formal real-time model transformations in moment2. In DavidS. Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 29–43. Springer Berlin Heidelberg, 2010. 36

[Bon04] Richard Bonichon. Tamed : A tableau method for deduction modulo. In *IJCAR*, pages 445–459, 2004. 24

[Bra10] Paul Brauner. *Fondements et mise en œuvre de la Super Déduction Modulo.* PhD thesis, Université Henri Poincaré - Nancy I, 06 2010. 24

[Bro04] Alan W. Brown. Model driven architecture : Principles and practice. *Software and Systems Modeling*, 3(4) :314–327, 2004. 31

[BRV04] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool tina – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14) :2741–2756, 2004. 41, 91, 119

[BS05] Xavier Blanc and Olivier Salvatori. *MDA en action : Ingénierie logicielle guidée par les modèles.* Editions Eyrolles, 2005. 31, 33

[Bur09] Guillaume Burel. *Bonnes démonstrations en déduction modulo.* PhD thesis, Université Henri Poincaré (Nancy 1), 2009. 24

[Bé05] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2) :171–188, 2005. 27

[Bé06] Jean Bézivin. Model driven engineering : An emerging technical space. In Ralf Laemmel, Joao Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer Berlin Heidelberg, 2006. 28

[CDE+02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude : specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2) :187 – 243, 2002. Rewriting Logic and its Applications. 13, 35

[CDE+11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. *Maude manual (version 2.6)*, 2011. 13

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of maude. *Electronic Notes in Theoretical Computer Science*, 4(0) :65 – 89, 1996. RWLW96, First International Workshop on Rewriting Logic and its Applications. 13, 35

[CH00] Koen Claessen and John Hughes. Quickcheck : A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9) :268–279, September 2000. 40

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17, 2003. 32, 68

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3) :621 –645, 2006. 32, 68

[CM96] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. *Electronic Notes in Theoretical Computer Science*, 4(0) :126 – 148, 1996. RWLW96, First International Workshop on Rewriting Logic and its Applications. 13, 35

[CM02] Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2) :245–288, 2002. 13

[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. in french. 83

[DFF+10] Zoé Drey, Cyril Faucher, Franck Fleurey, Vincent Mahé, and Didier Vojtisek. *Kermeta Language, Reference Manual*, 2010. 37

[DRB+10] Francisco Durán, Manuel Roldán, Jean-Christophe Bach, Emilie Balland, Mark Van Den Brand, James R. Cordy, Steven Eker, Luc Engelen, Maartje De Jonge, Karl Trygve Kalleberg, Lennart C. L. Kats, Pierre-Etienne Moreau, and Eelco Visser. The third rewrite engines competition. In PeterCsaba Ölveczky, editor, *Proceedings of the 8th international conference on Rewriting logic and its applications*, volume 6381 of *WRLA'10*, pages 243–261, Berlin, Heidelberg, 2010. Springer-Verlag.

[EF13] Clara Benac Earle and Lars-Ake Fredlund. Testing java with quickcheck. 2013. 40

[EMOMV07] Steven Eker, Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Deduction, strategies, and rewriting. *Electronic Notes in Theoretical Computer Science*, 174(11) :3–25, 2007. 13

[FBMT09] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and YvesLe Traon. Qualifying input test data for model transformations. *Software & Systems Modeling*, 8(2) :185–203, 2009. 40

[FEBF06] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino. *L'ingénierie dirigée par les modèles : au-delà du MDA.* Informatique et Systèmes d'Information. Hermes Science, February 2006. 30

[FHN+06] Jean-Rémy Falleri, Marianne Huchard, Clémentine Nebut, et al. Towards a traceability framework for model transformations in kermeta. In *ECMDA-TW'06 : ECMDA Traceability Workshop*, pages 31–40, 2006. 44

[Fow99] Martin Fowler. *Refactoring : Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 31

[FSB04] F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering : testing model transformations. In *Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on*, pages 29–40, Nov 2004. 40

[GLMS11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2010 : A toolbox for the construction and analysis of distributed processes. In ParoshAziz Abdulla and K.RustanM. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin Heidelberg, 2011. 41

[GLR+02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation : The Missing Link of MDA. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer Berlin Heidelberg, 2002. 34

[GP09] Pau Giner and Vicente Pelechano. Test-driven development of model transformations. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 748–752. Springer Berlin Heidelberg, 2009. 40

[GS03] Jack Greenfield and Keith Short. Software factories : Assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 16–27, New York, NY, USA, 2003. ACM. 30

[Hug10] John Hughes. Software testing with quickcheck. In Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók, editors, *Central European Functional Programming School*, volume 6299 of *Lecture Notes in Computer Science*, pages 183–223. Springer Berlin Heidelberg, 2010. 40

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL : A model transformation tool. *Science of Computer Programming*, 72(1-2) :31–39, June 2008. 37

[JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. Grgen.net. *International Journal on Software Tools for Technology Transfer*, 12 :263–271, 2010. 51

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin / Heidelberg, 2006. 37

[Jou05] Frédéric Jouault. Loosely Coupled Traceability for ATL. In *In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, pages 29–37, 2005. 59

[KBA02] I. Kurtev, J. Bézivin, and M. Akşit. Technological spaces : An initial appraisal. In *International Conference on Cooperative Information Systems (CoopIS), DOA'2002 Federated Conferences, Industrial Track, Irvine, USA*, pages 1–6, October 2002. 28

[KHVDB+11] Paul Klint, Mark Hills, Jeroen Van Den Bos, Tijs Van Der Storm, and Jurgen Vinju. Rascal : From algebraic specification to meta-programming. In *Proceedings Second International Workshop on Algebraic Methods in Model-based Software Engineering (AMMSE)*, pages 15–32, Zurich, Suisse, 2011. 35

[KKK+] Jan Korta, Paul Klint, Steven Klusener, Ralf Lämmel, ChrisVerhoef, and Ernst-Jan Verhoeven. Engineering of Grammarware. 28

[KKK+08] Claude Kirchner, Florent Kirchner, Hélène Kirchner, et al. Strategic computation and deduction. *Reasoning in Simple Type Theory. Festchrift in Honour of Peter B. Andrews on His 70th Birthday*, 17 :339–364, 2008. 12

[KLV05] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3) :331–380, July 2005. 28

[KMT09] Claude Kirchner, Pierre-Etienne Moreau, and Cláudia Tavares. A type system for tom. In *RULE*, pages 51–63, 2009. 23, 77

[Kön05] Alexander Königs. Model transformation with triple graph grammars. In *Model Transformations in Practice Satellite Workshop of MODELS*, page 166, 2005. 36

[KV10] Lennart C. L. Kats and Eelco Visser. The spoofax language workbench : rules for declarative specification of languages and ides. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *OOPSLA*, volume 45, pages 444–463, New York, NY, USA, October 2010. ACM. 35

[KvdSV11] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-programming with Rascal. In João Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer Berlin / Heidelberg, 2011. 35

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained : The Model Driven Architecture : Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 27, 28, 31

[Lam07] Maher Lamari. Towards an automated test generation for the verification of model transformations. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 998–1005, New York, NY, USA, 2007. ACM. 40

[LS06] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 139–150. Springer Berlin Heidelberg, 2006. 37

[LZG05] Yuehua Lin, Jing Zhang, and Jeff Gray. A testing framework for model transformations. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-Driven Software Development*, pages 219–236. Springer Berlin Heidelberg, 2005. 40

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005. 37

[MFV+05] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, Jean-Marc Jézéquel, et al. On executable meta-languages applied to model transformations. In *In Model Transformations In Practice Workshop*, 2005. 37

[MG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152 :125–142, 2006. 32

[MKUW04] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA distilled : principles of model-driven architecture*. Addison-Wesley Professional, 2004. 28

[MOMV05] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Towards a strategy language for maude. *Electronic Notes in Theoretical Computer Science*, 117 :417–441, 2005. 13

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 61–76. Springer Berlin / Heidelberg, 2003. 7, 13, 35

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. Springer, 2002. 41

[OMG03] OMG. *Diagram Interchange 2.0.* Object Management Group, Inc., September 2003. 28

[OMG06a] OMG. *Meta Object Facility (MOF) Core Specification Version 2.0.* Object Management Group, Inc., May 2006. 28

[OMG06b] OMG. *Object Constraint Language Specification (OCL) 2.0 Specification.* Object Management Group, Inc., May 2006. 28

[OMG08] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Query/-View/Transformation (QVT) Specification, version 1.0*, April 2008. ix, 33, 36

[OMG09] OMG. *Unified Modeling Language (UML), v2.2.* Object Management Group, Inc., February 2009. 28, 29

[OMG11] OMG. *Unified Modeling Language (UML), v2.4.1.* Object Management Group, Inc., August 2011. 28

[OMG13] OMG. *MOF/XMI Mapping 2.4.1.* Object Management Group, Inc., June 2013. 28

[RCD+11] Jonathan Robie, Don Chamberlin, Michael Dyck, Daniela Florescu, Jim Melton, and J Siméon. *XQuery Update Facility 1.0.* W3C, March 2011. 34, 35

[Rei07] Antoine Reilles. Canonical abstract syntax trees. *Electronic Notes in Theoretical Computer Science*, 176(4) :165 – 179, 2007. Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006). 14

[RRDV07] José Raúl Romero, José Eduardo Rivera, Francisco Durán, and Antonio Vallecillo. Formal and tool support for model driven engineering with maude. *Journal of Object Technology*, 6(9) :187–207, 2007. 35

[Rus11] Vlad Rusu. Embedding domain-specific modelling languages in maude specifications. *ACM SIGSOFT Software Engineering Notes*, 36(1) :1–8, 2011. 35

[RV08] José Rivera and Antonio Vallecillo. Representing and operating with model differences. In RichardF. Paige and Bertrand Meyer, editors, *Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 141–160. Springer Berlin Heidelberg, 2008. 35

[RW03] MatthewJ. Rutherford and AlexanderL. Wolf. A case for test-code generation in model-driven systems. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*, pages 377–396. Springer Berlin Heidelberg, 2003. 40

[SBM09] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In RichardF. Paige, editor, *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 148–164. Springer Berlin Heidelberg, 2009. 40

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009. 31, 35

[Sei03] E. Seidewitz. What models mean. *Software, IEEE*, 20(5) :26 – 32, sept.-oct. 2003. 27

[SK03] S. Sendall and W. Kozaczynski. Model transformation : the heart and soul of model-driven software development. *Software, IEEE*, 20(5) :42 – 45, sept.-oct. 2003. 31, 32

[SK08] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2008. 36

[Spe11a] Special Committee 205 of RTCA. *DO-178C : Software considerations in airborne systems and equipment certification*, 2011. 42

[Spe11b] Special Committee 205 of RTCA. *DO-330 : Software Tool Qualification Considerations*, December 2011. 43

[StOS00] Richard Soley and the OMG Staff. Model Driven Architecture. white paper, Nov 2000. 28

[Tae04] Gabriele Taentzer. Agg : A graph transformation environment for modeling and validation of software. In JohnL. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer Berlin Heidelberg, 2004. 34, 36

[Tae10] Gabriele Taentzer. What algebraic graph transformations can do for model transformations. *ECEASST*, 30 :1–10, 2010. 36

[Tav12] Cláudia Tavares. *Un systeme de types pour la programmation par réécriture embarquée*. PhD thesis, Université Henri Poincaré-Nancy I, 2012. 23

[The04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0. 41

[VB07] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3) :214 – 234, 2007. Special Issue on Model Transformation. 36, 37, 51

[VBT98] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. Building program optimizers with rewriting strategies. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *ICFP*, pages 13–26. ACM, 1998. 13, 19, 35

[Vis01a] Eelco Visser. Stratego : A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5. In Aart Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer Berlin Heidelberg, 2001. 13, 35

[Vis01b] Joost Visser. Visitor combination and traversal control. In Linda M. Northrop and John M. Vlissides, editors, *OOPSLA*, pages 270–282. ACM, 2001. 19

[Vit94] Marian Vittek. *ELAN : Un cadre logique pour le prototypage de langages de programmation avec contraintes*. Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, octobre 1994. 13

[VVP02] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2) :205 – 227, 2002. 51

[W3C99] W3C. *XSL Transformations (XSLT) Version 1.0*, November 1999. 34, 35

[WMV03] L. Williams, E.M. Maximilien, and M. Vouk. Test-driven development as a defect-reduction practice. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 34–45, Nov 2003. 40

Résumé

La certification est un processus durant lequel une autorité approuve un logiciel, imposant des contraintes de qualité strictes pour les systèmes critiques. Or leurs chaînes de développement se complexifient et intègrent de plus en plus couramment des outils issus de l'Ingénierie Dirigée par les Modèles. Ces derniers permettent de générer une partie du logiciel par transformations successives de modèles jusqu'au code. Il est donc nécessaire d'utiliser des outils fiables et de vérifier les transformations opérées. Ils doivent cependant remplir des exigences imposées par la qualification pour pouvoir être intégrés dans les chaînes de développement.

Nous nous intéressons dans cette thèse à fournir des outils et des méthodes utilisant le formalisme de la réécriture pour écrire des transformations de modèles qualifiables.

Dans ce travail, nous nous appuyons sur le langage Tom, qui intègre des fonctionnalités telles que le filtrage et la réécriture dans les langages généralistes comme Java, C ou Ada. Nous présentons une méthode hybride de transformation de modèles par réécriture, à mi-chemin entre les méthodes utilisant des outils généralistes et celles s'appuyant sur des langages dédiés. Nous proposons un outil de représentation de modèles sous la forme de termes algébriques ainsi qu'une nouvelle construction Tom pour implémenter cette méthode. Nous complétons ensuite ces outils par l'ajout d'une construction implémentant la notion de traçabilité. Cette dernière fait partie des exigences de qualification. La trace fournie peut ainsi être réutilisée à des fins de vérification *a posteriori*. Nous détaillons l'utilisation de nos outils sur des cas d'étude et nous les évaluons, en particulier du point de vue des performances lors du passage à l'échelle.

Ces résultats constituent ainsi une avancée vers le développement de logiciels critiques plus fiables.

Mots-clés: réécriture, termes, transformation, modèles, traçabilité, qualification.

Abstract

During the software certification process, a piece of software is approved by an authority. It requires important quality constraints for critical systems. Development chains are getting increasingly complex and integrate more and more tools from Model Driven Engineering. The latter allow to generate a part of software doing successive transformations, from models to source code. Therefore it is necessary to use liable tools and to verify transformations. However, they have to fulfill qualification requirements in order to be integrated within development chains.

In this thesis we study how to provide tools and methods, based on the rewriting formalism, to write qualifiable models transformations.

In this work, we are relying on the Tom language which adds features such as pattern-matching and rewriting capabilities in general purpose languages (for instance in Java, C or Ada). We present an hybrid method of models transformation by rewriting. It fills a gap between methods using general purpose tools and the ones using domain specific languages. We propose tools to represent models as algebraic terms and a new construct within Tom to implement this hybrid method. Then, we complete these tools by adding another construct that implements the notion of traceability. The latter is a requirement for software qualification. The provided trace can be reused for back verification. We also explain the mechanisms on which our tools rely. We detail the use of our tools with practical case study. Then, we evaluate them, particularly from the performances point of view during scaling.

Those results can be seen as an additional step towards more liable critical software development.

Keywords: rewriting, terms, transformation, models, traceability, qualification.