# Une approche hybride GPL-DSL pour transformer des modèles

**Jean-Christophe Bach**

*Inria, Villers-lès-Nancy, F-54600, France*
*LORIA, UMR CNRS 7503, Université de Lorraine*
*Vandœuvre-lès-Nancy, F-54500, France*
*jeanchristophe.bach@inria.fr*

RÉSUMÉ. *L'ingénierie des modèles (IDM) plaide en faveur des transformations de modèles afin d'automatiser au maximum le développement logiciel et sa vérification. Entre approches opérationnelles et relationnelles, à base de langages dédiés ou généralistes, de nombreux outils de transformation de modèles existent dans le domaine. Pour faciliter et accélérer le développement logiciel basé sur les transformations de modèles, tout en assurant la qualité du logiciel, nous proposons une méthode ainsi qu'un langage associé et de l'outillage dédié. Notre approche se situe à la frontière des langages généralistes et des langages dédiés afin de pouvoir bénéficier du meilleur des deux mondes pour une qualité logicielle accrue. Elle s'appuie sur l'usage du langage Tom, qui est une extension de langages généralistes. Notre proposition permet d'écrire des transformations modulaires, dont le code est réutilisable, et qui sont traçables.*

ABSTRACT. *Model Driven Engineering (MDE) advocates the use of model transformations in order to automate software development and its verification. Between operational and relational approaches, based on general purpose or domain specific languages, many models transformations tools have been developed. To ease and to speed up software development based on models transformations, we propose a method, an associated language and dedicated tooling. Our approach aims to bridge the gap between general purpose languages and domain specific ones in order to take benefit from both of the two worlds, increasing software quality. Our approach uses the Tom language which is a shallow extension of general purpose languages. Our proposal allows to write modular transformations whose code is reusable, and which are traceable.*

MOTS-CLÉS : *transformations de modèles, langage, tom, java, langage de transformation, EMF, Ecore.*

KEYWORDS: *models transformations, language, tom, java, transformation language, EMF, Ecore.*

DOI:10.3166/TSI.33.175-201 

## 1. Introduction

Par sa capacité d'abstraction des problèmes, l'ingénierie des modèles (IDM) permet d'appréhender et de créer des systèmes de plus en plus complexes, soumis à des contraintes de fiabilité fortes. Le modèle – ou l'abstraction d'un problème spécifique – est conforme à son métamodèle qui est la spécification dans un langage de modélisation pour ce type de problème. Résoudre un problème donné consiste alors à modéliser ce problème, puis à transformer cette représentation avec les techniques et outils adaptés. Les transformations de modèles – ou transformations de problèmes exprimés dans un métamodèle donné en des problèmes exprimés avec un autre métamodèle – sont donc au cœur de l'IDM. Elles facilitent des activités telles que la construction de squelettes d'applications, l'automatisation des tâches de développement répétitives, la maintenance logicielle (*refactoring* par exemple), la vérification d'applications. L'objectif étant d'améliorer la qualité du logiciel tout en limitant les coûts de développement et de maintenance : il sera souvent plus intéressant de développer un outil qui génère du code à partir d'un modèle donné plutôt que d'attribuer cette tâche à un humain. En effet, outre le fait que ce type de tâche peut être répétitive et sans véritable valeur ajoutée, l'intervention humaine peut être un facteur d'introduction d'erreurs (*bugs*) non négligeable. Utiliser un outil de génération issu de l'IDM permet de limiter ce facteur et de cibler plus facilement la source en cas de *bug*.

Cette problématique de la transformation de modèles étant fondamentale, de nombreux outils dédiés à la transformation de modèles ont vu le jour, avec des approches différentes. En effet, si la vision d'une transformation de modèle comme un programme prenant un modèle en entrée retournant un nouveau modèle – conforme au métamodèle d'entrée ou à un autre métamodèle – paraît simple, exprimer une transformation de modèles évoluée dans un formalisme exécutable peut être complexe.

Pour ce faire, il existe de nombreux outils et langages adoptant généralement l'une des deux approches suivantes : l'utilisation de langages dédiés tels que QVT (OMG, 2008), ATL (Jouault *et al.*, 2008), Kermeta (Jézéquel *et al.*, 2011 ; Muller *et al.*, 2005), *etc.*, ou l'utilisation de langages généralistes (Java par exemple) équipés de *frameworks* tels que EMF (Steinberg *et al.*, 2009). Ces deux types d'approches ont leurs avantages et inconvénients. Il est parfois difficile de choisir un outil, les critères de choix (outillage externe – IDE, débogueur, bibliothèques –, vaste communauté d'utilisateurs, documentation, facilité d'utilisation, intégration à l'environnement courant, outil inconnu des développeurs, etc.) n'étant pas les mêmes selon le contexte (environnement de développement, compétences des utilisateurs, outils historiques, usage des transformations, etc.). De plus, le choix est aussi souvent conditionné par son coût induit.

De notre côté, nous nous positionnons entre ces deux approches pour bénéficier à la fois des avantages de l'approche généraliste et de l'approche dédiée. Pour cela, nous nous appuyons sur le langage Tom [1] qui a été conçu dans l'optique d'enrichir des

1. http://tom.loria.fr

langages généralistes *via* des constructions dédiées totalement intégrées aux langages génénalistes qu'il étend. Nous donnons un aperçu du langage et des constructions dans la section 6.

Nous souhaitons aussi répondre aux problématiques essentielles suivantes qu'induisent les transformations de modèles :

– modularité : notre approche ainsi que les technologies sur lesquelles nous nous appuyons permettent de rendre nos transformations modulaires, et donc de pouvoir réutiliser le code développé pour uni autre transformation ou même de pouvoir changer la représentation des modèles à moindre coût.

– traçabilité : notre approche permet de générer des traces qui peuvent être utilisées à des fins de vérification ou de *debug*. Nous distinguons deux types de traçabilités : la traçabilité dite interne ou technique (Jouault, 2005) – proche de l'implémentation et dont l'usage est pour la transformation elle-même ainsi que pour le *debug* –, et la traçabilité des spécifications – qui peut être plus éloignée de l'implémentation –. C'est cette dernière qui nous intéresse à terme étant donné que c'est un élément central du processus de qualification d'un logiciel.

– vérification : nous souhaitons que les briques logicielles que nous apportons soient une première étape pour permettre à terme la vérification de transformations qualifiables dans un environnement Java. Notons que nous ne vérifions pas les transformations mais que nous tentons de fournir des outils pour aider à la vérification.

Dans cette optique, nous présentons dans cet article notre approche hybride pour transformer des modèles en nous appuyant sur les technologies Tom, Java, et EMF Ecore. Nous proposons une extension haut-niveau dédiée à la transformation de modèles pour le langage Tom, que nous illustrons dans un cas d'étude.

Suite à cette partie introductive, l'article est articulé de la façon suivante. Après des notions préliminaires en section 2, nous expliquons le cas d'étude sur lequel nous appuyons nos exemples dans la section 3. Nous motivons notre choix d'approche hybride dans la section 4, puis nous détaillons la méthode de notre approche dans la section 5. Dans la section 6, nous décrivons rapidement le langage Tom, pour pouvoir ensuite expliquer la mise en œuvre de notre approche dans la section 7. Nous discutons des limitations de nos outils et exposons nos perspectives dans la section 8 avant de conclure dans la section 9.

## 2. Notions préliminaires

Dans cette partie, nous définissons et précisons certains termes et notations que nous utiliserons dans cet article. L'IDM se concentre sur la création et la manipulation de modèles, et plaide pour l'automatisation des processus de développement. Au début des années 2000, l'OMG [2] a proposé l'initiative MDA (*Model-Driven Architecture*) comme réalisation des principes de l'IDM autour de standards et de technologies.

2. *Object Management Group*, http://www.omg.org

DÉFINITION 1 (Modèle). — *Un modèle est une abstraction d'un système, modélisé sous la forme d'un graphe.*

DÉFINITION 2 (Métamodèle). — *Un métamodèle est un modèle qui définit le langage d'expression d'un modèle (OMG, 2006a), c'est-à-dire le langage de modélisation. Nous noterons les métamodèles $MM$. Les métamodèles source et cible seront respectivement notés $MM_s$ et $MM_t$. Nous utiliserons aussi par la suite la notation $MM_{t_{resolve}}$ pour désigner un métamodèle intermédiaire constitué du métamodèle cible étendu.*

La notion de métamodèle a été formalisée par l'OMG dans le standard Meta Object Facility (MOF) 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 que c'est un ensemble d'éléments, d'attributs et de relations entre les éléments se conformant à leurs métadéfinitions.

DÉFINITION 3 (Transformation de modèle). — *Une transformation de modèle $T$ est une relation d'un métamodèle source $MM_s$ vers un métamodèle cible $MM_t$. On la notera : $T : MM_s \rightarrow MM_t$.*

L'OMG a défini le standard *Query/View/Transformation* (QVT) pour fournir des technologies de transformation de modèles. Les langages de modélisation sont définis en utilisant le standard MOF et sont manipulés en utilisant OCL [3], le langage déclaratif d'expression de contraintes pour les modèles qui fait partie du standard QVT (OMG, 2006b). Il existe au moins deux approches principales pour décrire une transformation de modèle proposées notamment par QVT :

– Une transformation de modèle est exprimée comme une séquence de pas d'exécution élémentaires qui construisent le modèle cible pas à pas (instanciation des nouveaux éléments, initialisation des attributs, création des liens, *etc.*) en utilisant les informations du modèle source. Cette approche, généralement appelée *opérationnelle*, est impérative. Elle peut être implémentée en utilisant des outils dédiés tels que Kermeta, QVT-operational… ou en utilisant des bibliothèques réflexives ou du code généré au sein d'un langage généraliste.

– 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, généralement appelée *relationnelle* (ou déclarative), n'est pas directement exécutable, mais peut parfois être traduite en une transformation opérationnelle. Des outils tels que QVT-Relations et ATL permettent de la mettre en œuvre.

3. *Object Constraint Language*

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 cela est géré par le moteur d'exécution pour l'approche relationnelle.

Certains auteurs (Sendall, Kozaczynski, 2003) considèrent même une troisième approche architecturelle basée sur l'introduction d'une représentation intermédiaire comme XML ou n'importe quelle autre syntaxe concrète textuelle. La transformation de modèle peut ensuite être décrite en utilisant un langage tel que XSLT (W3C, 1999) ou tout autre langage de transformation comme Stratego/XT (Visser, 2001 ; 2004) , ASF+SDF (Brand *et al*., 2002), Rascal(Klint *et al*., 2009), etc.

NOTE 4 (Terme). — *À chaque fois que nous parlons de terme – ou terme Tom –, nous nous référons à un objet ayant une structure d'arbre.*

## 3. Cas d'étude

Pour illustrer notre propos, nous nous appuyons sur un cas d'utilisation bien connu de la communauté et décrit en détails dans (Combemale, 2008) : SimplePDLToPetriNet. Le but de ce cas d'étude est de transformer des processus décrits dans le formalisme SimplePDL en leurs réseaux de Petri équivalents. Dans la communauté, cette transformation sert à la vérification des processus : on ne peut vérifier directement le processus décrit avec le formalisme SimplePDL, en revanche sa vue sous forme de réseau de Petri permet l'utilisation d'un *model-checker*.

Nous décidons de modéliser un processus hiérarchique composé d'activités et de contraintes de précédence, que nous donnons en figure 1.

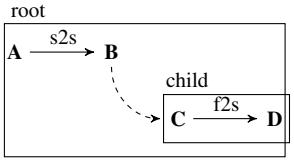



*Figure 1. Processus, instance de SimplePDL*

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 démarrée. `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 que `D` puisse démarrer.

Ce processus est conforme au métamodèle SimplePDL donné par la figure 2. Ce métamodèle définit le concept de *Processus* (*root*, *child*) composé de *ProcessElement*s (*A*, *B*, *s2s*, *C*, *D*, *f2s*). Chaque *ProcessElement* peut être une *WorkDefinition* (*A*, *B*, *C*, *D*) ou une *WorkSequence* (*s2s*, *f2s*). Les *WorkDefinition*s sont des activités qui doivent être effectuées pendant un processus. Une *WorkSequence* définit une relation de dépendance entre deux activités. La deuxième *WorkDefinition* (*successor*) peut être démarrée – ou terminée – uniquement lorsque la première (*predecessor*) est déjà

démarrée – ou terminée – selon la valeur de l'attribut *linkType* (donc quatre valeurs possibles : *start2start*, *start2finish*, *finish2start*, *finish2start*). Enfin, une *WorkDefinition* peut elle-même être définie par un processus imbriqué (référence *process*), ce qui permet de définir des processus hiérarchiques (figure 1 : le processus *child* décrit l'activité *B*).

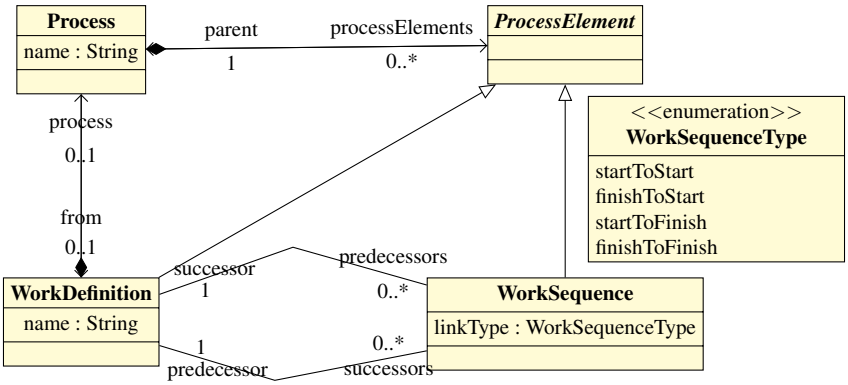



*Figure 2. Métamodèle SimplePDL*

Nous souhaitons transformer le processus décrit et illustré figure 1 afin de vérifier ses propriétés (par exemple s'assurer des contraintes de précédence données par les *WorkSequences*). Cependant, le formalisme SimplePDL n'est pas le plus adapté à la vérification. En revanche, celui des réseaux de Petri est beaucoup plus adapté au monde de la vérification, et du *model-checking*. Il s'agit d'un langage pour spécifier formellement la synchronisation dans les systèmes concurrents pour lequel de nombreux outils de vérification par *model-checking* existent [4]. Le métamodèle des réseaux de Petri est illustré par la figure 3 et permet de décrire le langage des réseaux de Petri. Un *PetriNet* est composé de *Node*s qui sont soit des *Place*s, soit des *Transition*s. Les nœuds sont reliés entre eux par des *Arc*s qui peuvent être de type *normal*, ou de type *read_arc*. Un arc spécifie le nombre de jetons (*weight* – poids –) consommés dans la place source ou produits dans la cible lorsqu'une transition est tirée. Un *read-arc* contrôle uniquement la disponibilité des jetons sans pour autant les supprimer. Le marquage d'un réseau de Petri est défini par le nombre de jetons dans chaque place (*marking*) et il représente l'état du système.

Pour transformer le processus décrit figure 1, on peut isoler aisément trois transformations élémentaires qui composeront la transformation globale. Chacune d'entre elles traitera un type d'élément du modèle source : respectivement `Process2PetriNet`, `WorkDefinition2PetriNet` et `WorkSequence2PetriNet` pour les éléments `Process`, `WorkDefinition` et `WorkSequence` comme l'illustre la figure 4. Dans ces schémas, les places sont représentées par des cercles rouges, tandis que les transitions le sont par des carrés bleus. Les arcs de type *normal* sont quant

4. Notamment TINA : http://www.laas.fr/tina/

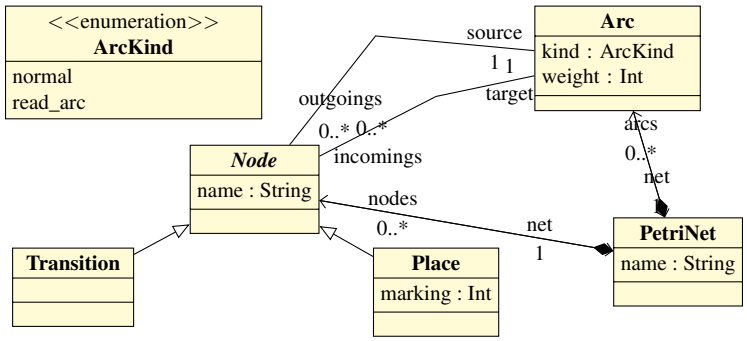



*Figure 3. Métamodèle PetriNet*

à eux matérialisés par des flèches en trait plein – vertes dans le cas d'une séquence –. Celles en pointillés correspondent aux synchronisations entre éléments, c'est-à-dire aux arcs de type *read_arc*.

Ainsi, `Process2PetriNet` traduit un `Process` en un réseau de Petri représenté par le cas *a.* de la figure 4. L'image d'un processus est donc constituée de trois places ($p_{ready}$, $p_{running}$ et $p_{finished}$), deux transitions ($t_{start}$ et $t_{finish}$) et quatre arcs.

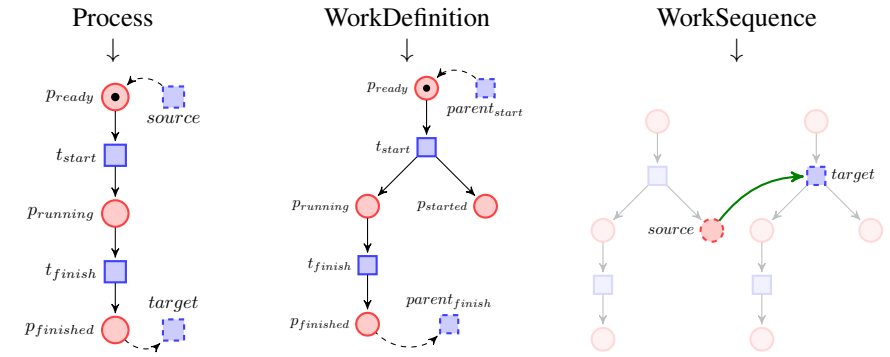
`WorkDefinition2PetriNet` créera tous les éléments du réseau de Petri qui définissent l'image d'une `WorkDefinition` (cas *b.* de la figure 4). Ce réseau de Petri est composé de quatre places ($p_{ready}$, $p_{running}$, $p_{started}$ et $p_{finished}$), deux transitions ($t_{start}$ et $t_{finish}$), et cinq arcs. La seule différence avec la représentation d'un processus et celle d'une *WorkDefinition* est la place supplémentaire $p_{started}$ après la transition $t_{start}$. Elle permet l'ajout d'une séquence entre deux *WorkDefinitions*.

Une `WorkSequence` est traduite par un `Arc` dans la transformation `WorkSequence2PetriNet`, dont la représentation est le cas *c.* de la figure 4, lorsqu'il s'agit d'une séquence *start2start*. Par souci de clarté, nous avons choisi de colorer en vert les arcs résultant d'une transformation de *WorkSequences*.

La transformation traduit chaque processus et activité en un réseau de Petri donné, et chaque séquence en un arc. Ensuite, lorsque les processus et les activités sont traduites, des arcs sont générés pour encoder la synchronisation entre les processus et leurs activités. La figure 5 illustre le réseau de Petri résultant de notre exemple.

## 4. Choix et intérêt de l'approche hybride

Pour écrire une transformation de modèle telle que celle que nous avons présentée, nous avons de nombreux outils et langages à disposition. Parmi eux, on retrouve généralement deux approches : l'utilisation de langages dédiés tels que QVT, Kermeta, ATL, etc., ou l'utilisation de langages généralistes (Java par exemple) équipés de *framework* tels que EMF.




*a. ProcessToPetriNet* *b. WorkDefinitionToPetriNet* *c. WorkSequenceToPetriNet*

*Figure 4. Transformations élémentaires composant* `SimplePDLToPetriNet`

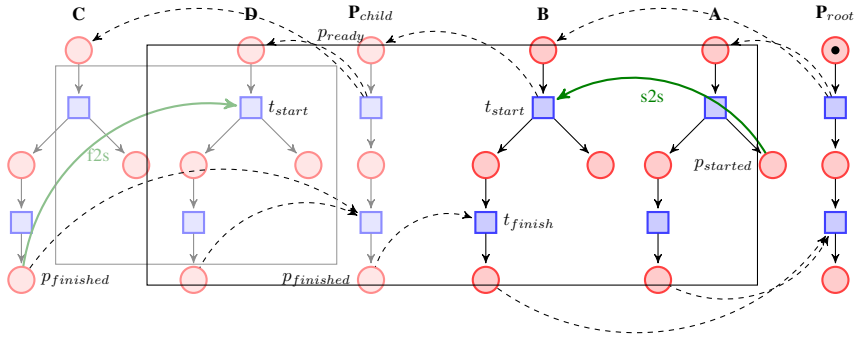



*Figure 5. Réseau de Petri résultant de la transformation*

Ces deux types d'approches ont leurs avantages et leurs inconvénients et il est parfois difficile de choisir. En effet, si l'utilisation de langages généralistes permet de bénéficier d'outils (IDE, débogueur, bibliothèques) ainsi que d'une communauté généralement étendue (et donc de documentation et d'un support possible), certaines tâches précises (retrouver et modifier une information dans la structure de données par exemple) peuvent se révéler fastidieuses ou difficiles à mettre en œuvre. Bien qu'il existe des *frameworks* dédiés à la manipulation de modèles tels que EMF, et offrant de nombreuses possibilités, leur utilisation n'est pas toujours aisée et l'écriture d'une transformation n'est pas toujours concise et naturelle pour un utilisateur.

À l'opposé, on peut choisir d'utiliser une approche par langage dédié aux transformations de modèles. Les problèmes usuels liés au domaine sont alors naturellement traités par des méthodes et constructions dédiées, simplifiant ainsi l'écriture des transformations de modèles. L'inconvénient est alors l'intégration de la transformation dans la chaîne de développement : en effet, l'ajout d'un nouveau langage (et de nouveaux outils) implique une adaptation de la chaîne de développement, le langage dédié ne

s'intégrant pas toujours parfaitement au sein des outils utilisés. À cela s'ajoute le fait que l'équipe de développement doit apprendre un nouveau langage pour être en mesure d'écrire la transformation de modèle, ce qui induit des coûts supplémentaires.

Pour notre part, nous avons choisi de nous positionner entre ces deux approches pour bénéficier à la fois des avantages de l'approche généraliste et ceux de l'approche dédiée. Pour cela, nous nous appuyons sur le langage Tom qui a été conçu dans l'optique d'enrichir des langages généralistes *via* des constructions dédiées. Ce langage est intéressant dans le sens où il s'intègre parfaitement dans le langage généraliste qu'il étend (Java dans notre cas). Nous en donnons un aperçu dans la section 6. Cette approche a pour avantages de ne pas perturber la chaîne de développement par l'introduction de nouveaux outils non intégrés, de ne pas obliger les utilisateurs à apprendre un nouveau langage complet, de fournir des constructions dédiées pour mettre en œuvre certaines tâches et résoudre des problèmes précis. Un utilisateur ne souhaitant pas maximiser l'utilisation du langage Tom et des outils associés pour le développement de son logiciel pourra rester dans le monde Java pour l'essentiel du code, et n'utiliser Tom que de manière ponctuelle.

## 5. Approche en deux temps : transformation puis réconciliation

Dans cette partie, nous décrivons notre approche pour la transformation de modèles : nous donnons dans un premier temps le principe général de l'approche accompagné d'une formalisation simple, puis nous la détaillons.

### *5.1. Principe général et formalisation simple de l'approche*

Le principe de notre approche est de décomposer une transformation $T$ en deux phases distinctes qui peuvent être vues comme des fonctions. Nous verrons par la suite que nous encodons ces transformations par des stratégies de réécriture.

La première $c : MM_s \rightarrow MM_{t_{resolve}}$ consiste à créer les éléments cibles du modèle résultant ainsi que des éléments additionnels que nous appelons *éléments resolve* (équivalent au *resolve* de QVT et au *resolveTemp* de ATL). Ces éléments temporaires permettent de faire référence à des éléments qui n'auraient pas encore été créés à cet instant de la transformation. Des éléments *resolve* étant créés et intégrés au résultat durant cette phase, le modèle résultant est conforme au métamodèle cible étendu $MM_{t_{resolve}}$.

La seconde phase $r : MM_{t_{resolve}} \rightarrow MM_t$, quant à elle, a pour objectif de rendre cohérent le modèle cible résultant, c'est-à-dire conforme au métamodèle cible $MM_t$. Elle consiste donc à éliminer les éléments intermédiaires *resolve* et à les remplacer par des références vers les éléments effectivement créés par la transformation. Cette seconde phase n'ajoute aucun nouvel élément cible au résultat.

Les deux fonctions étant définies et la seconde s'appliquant au résultat de la première, la transformation complète $T : MM_s \rightarrow MM_t$ est définie par leur composée : $T = r \circ c$.

En instanciant notre approche relativement classique dans le domaine avec le cas d'étude ***SimplePDLToPetriNet*** – $MM_{SimplePDL}$ (respectivement $MM_{PetriNet}$ et $MM_{PetriNet_{resolve}}$) étant le métamodèle source (respectivement cible et cible étendu) – nous obtenons :

$$\begin{aligned} SimplePDLToPetriNet &: MM_{SimplePDL} \rightarrow MM_{PetriNet} \\ Transformer &: MM_{SimplePDL} \rightarrow MM_{PetriNet_{resolve}} \\ Resolve &: MM_{PetriNet_{resolve}} \rightarrow MM_{PetriNet} \\ SimplePDLToPetriNet &= Resolve \circ Transformer \end{aligned}$$

### *5.2. Description de l'approche*

Une transformation de modèle $T$ est une relation d'un métamodèle source $MM_s$ vers un métamodèle cible $MM_t$. L'écriture de cette relation peut se faire par une approche procédurale monolithique. L'utilisateur construira sa transformation par étapes (*transformation steps*), dont l'ordre s'imposera naturellement en fonction des besoins des différents éléments : par exemple, pour construire un `Arc` – image d'une `WorkSequence` –, on construira d'abord ses deux extrémités (une `Place` et une `Transition`), puis l'arc lui-même. Cependant, cette approche 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 – par exemple l'ajout de la hiérarchie des processus dans notre cas d'étude – implique donc de repenser la transformation.

Pour 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 faut adopter une méthode permettant une grande modularité du code. Nous décomposons donc d'abord une transformation complexe en transformations plus simples (ce que nous nommons *transformations élémentaires* ou *définitions*). La transformation globale est ensuite construite en utilisant ces transformations élémentaires. De plus, par souci d'utilisabilité, nous ajoutons la contrainte suivante : la gestion de l'ordre d'application des définitions ne doit pas être du ressort de l'utilisateur. Se pose donc le problème de la dépendance des définitions entre elles, ainsi que de la référence à des éléments issus d'une transformation élémentaire dans une autre transformation élémentaire. Il faut donc mettre en œuvre un mécanisme permettant de résoudre ces problèmes.

Un premier élément de réponse vient du fait que nous effectuons des transformations dites *out-place*, c'est-à-dire qui ne modifient pas le modèle source. 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 par exemple les outils VIATRA (Varró *et al.*, 2002)/VIATRA2 (Varró, Balogh, 2007) et GrGen.NET (Jakumeit *et al.*, 2010) –. Partant de ce constat, les transformations élémentaires constituant notre transformation n'entretiennent donc aucune dépendance dans le sens où la sortie d'une définition n'est pas l'entrée d'une autre définition. Dans notre approche, chaque sortie d'une transformation élémentaire est une partie du résultat final.

Le deuxième élément de réponse au problème de l'ordre et de la dépendance entre définitions (au sens où le résultat d'une définition peut référencer un élément du résultat d'une autre définition) est l'introduction d'éléments intermédiaires dits *resolve* dont nous détaillons le fonctionnement ci-après.

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 est de type un sous-type de l'élément final ciblé, auquel nous ajoutons des informations telles que l'élément d'origine et le nom de l'élément cible. Il peut ainsi être manipulé en lieu et place du terme ciblé censé être construit dans une autre définition. À la compilation, le métamodèle cible $MM_t$ est étendu en $MM_{t_{resolve}}$ par l'ajout des métaclasses *resolve*. Pour illustrer le mécanisme, la figure 6 instancie une partie du schéma d'extension du métamodèle cible au cas d'étude : il s'agit de l'extension du métamodèle liée à la transformation élémentaire `WorkDefinition2PetriNet` uniquement. Le métamodèle cible est enrichi d'une métaclasse *ResolvePT* pour pouvoir créer des éléments intermédiaires *resolve* qui jouent temporairement le rôle d'une *Transition* obtenue à partir d'une source *Process*.

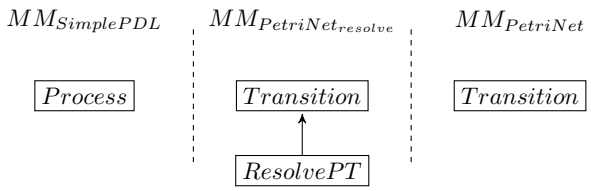



*Figure 6. Instanciation d'une partie du schéma d'extension du métamodèle cible pour le cas d'étude SimplePDLToPetriNet*

Concrètement, dans l'exemple *SimplePDLToPetriNet*, cela se traduit par l'instanciation de *Places* et de *Transitions resolve* ajoutées au résultat. La figure 7 permet d'illustrer ce mécanisme. Il s'agit des réseaux de Petri images de la *WorkDefinition* $WD_A$ et du *Process* $P_{root}$ extraits de notre cas d'étude. Une *WorkDefinition* possède un processus parent qui est transformé dans une définition différente de celle où est

transformée la *WorkDefinition* elle-même. Pour établir les liens de synchronisation (arcs en pointillés) entre ces deux réseaux de Petri, il est nécessaire de créer des éléments *resolve* (carrés en pointillés dans la figure). Le modèle résultant de l'application de toutes les transformations élémentaires – constitué de tous les résultats partiels – contient donc des éléments temporaires *resolve*.

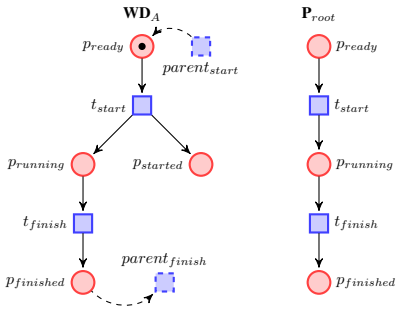



*Figure 7. Illustration du mécanisme* resolve *avec les réseaux de Petri images d'une* `WorkDefinition` *et d'un* `Process`

Durant cette première phase, chaque définition produit un ensemble d'éléments tous disjoints, les éléments censés provenir d'autres définitions ayant été représentés par des éléments *resolve*. Pour obtenir un résultat cohérent conforme au métamodèle cible, ce résultat intermédiaire est traité par la phase de résolution 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. Dans la figure 7, la résolution remplace donc les extrémités des arcs en pointillés par les transitions correspondantes du réseau de Petri de droite.

Ce remplacement est possible grâce aux informations supplémentaires qui enrichissent le type cible (source et nom de la cible) ainsi qu'aux informations que nous sauvegardons durant la transformation. Ces informations additionnelles sont des informations de relations 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. Une autre approche possible eût été de tracer systématiquement tous les termes créés, mais nous avons fait notre choix dans le but d'améliorer la lisibilité de la trace. Toutes ces 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.

Outre son usage interne pour la phase de résolution – traçabilité interne (Jouault, 2005) –, le modèle de lien nous permet d'assurer une forme de traçabilité de la trans-

formation, et peut être utilisé *a posteriori*. Nous en reparlons par la suite dans la description de l'implémentation technique ainsi que dans la section d'évaluation.

## 6. Aperçu du langage Tom

Dans cette section, nous présentons Tom qui est utilisé pour implémenter notre approche. Tom (Moreau *et al*., 2003 ; Balland *et al*., 2007) est un langage conçu pour intégrer des fonctionnalités issues de la réécriture et de la programmation fonctionnelle dans des langages généralistes tels que Java, C, C#, Caml, Python ou plus récemment Ada. Tom n'est pas un langage *stand-alone* : il est l'implémentation du principe des îlots formels (Balland *et al*., 2006) qui sont ajoutés au sein de programmes écrits dans un langage hôte. Les constructions Tom sont transformées et compilées vers le langage hôte. Cette approche présente l'intérêt de continuer à bénéficier des avantages des langage hôtes (système de type, outils, bibliothèques, forte communauté, *etc.*), tout en ayant de nouvelles fonctionnalités à moindre coût.

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*). Nous allons passer en revue les différentes fonctionnalités à travers les constructions Tom.

### *6.1. Filtrage de motif et backquote*

La principale fonctionnalité de Tom est le filtrage de motif qui s'opère grâce à la construction **%match**. Cette fonctionnalité peut être vue comme une généralisation de la construction *switch-case* que l'on retrouve dans de nombreux langages généralistes. Elle occupe une place importante dans les langages fonctionnels et, implémentée dans Tom, permet le développement plus sûr grâce au typage (différence notable avec les langages de la famille Lisp par exemple). La construction **%match** est constituée d'un ensemble de règles dont le membre gauche est un motif (*pattern*) et le membre droit une action. Le *pattern* possède une structure d'arbre et peut contenir des variables, tandis que l'action est un bloc de code hôte qui peut à son tour contenir des constructions Tom. L'extrait de code Listing 1 illustre la construction **%match** :

```
1 %match(node) {
2   Transition[name=n] -> { System.out.println(`n + " est une transition"); }
3   Place[name=n]      -> { System.out.println(`n + " est une place"); }
4 }
```

Listing 1 – Exemple de construction %match

Pour un sujet donné `node` (ligne 1), le *pattern* `Transition[name=n]` (ligne 2) vérifie que le sujet correspond bien à une *Transition*. Lorsque c'est le cas, la variable `n` est initialisée par l'objet correspondant au champ `name` de la structure. Un *pattern* peut contenir d'autres *patterns* imbriqués, ce qui ajoute des contraintes sur la forme des sous-termes. La conjonction (`&&`) et la disjonction (`||`) de *patterns*, ainsi

que le nombre d'occurences d'une variable (motifs non linéaires) peuvent aussi être considérés, comme illustré dans le Listing 2 :

```
1 %match(nodeList) {
2   NodeEList(_*,(Transition|Place)[name=n],_*) -> {
3     System.out.println("le node s'appelle "+`n);
4   }
5 }
```

Listing 2 – Exemple d'utilisation du filtrage associatif

Dans cet exemple, le sujet `nodeList` (ligne 1) est filtré. `NodeEList` (ligne 2) est un opérateur variadique, `_*` étant une variable anonyme ('_' de Prolog) pouvant être instanciée par une sous-liste éventuellement vide ('*' indique une multiplicité `0..n`). On note aussi l'utilisation de la disjonction de symboles de tête : en effet, nous avons une liste de `Node`s, et nous filtrons les `Transition`s ou les `Place`s.

Dans ces exemples, le membre droit (l'*action*) contient du Java ainsi que du Tom. En effet, on note une deuxième construction importante : le terme *backquote* (caractère **`**). Pour un terme algébrique donné, il permet de construire la structure de données en allouant et initialisant les objets en mémoire. Le *backquote* permet à la fois de construire un terme et de récupérer la valeur d'une variable instanciée par le filtrage de motif.

Les définitions des opérateurs `Place`, `Transition` et `NodeEList` ne sont pas internes au langage. Ils sont dérivés du métamodèle des réseaux de Petri donné en figure 3.

### *6.2. Ancrages algébriques*

Une autre fonctionnalité essentielle pour la création de termes et le filtrage de motif est ce que l'on appelle les ancrages algébriques ou *mappings*. En effet, pour pouvoir compiler les constructions de filtrage de motif, le compilateur Tom doit nécessairement connaître la relation entre l'implémentation des objets (les classes Java) et la vue algébrique (les types des *patterns* et les opérateurs). Les constructions **%typeterm** et **%op** (**%oplist** et **%oparray** pour les opérateurs variadiques tels que `NodeEList`) permettent justement de définir cette relation. Le Listing 3 illustre ces constructions. **%typeterm** lie le type de donnée de l'implémentation (`petrinet.Place`) et le type algébrique. La construction **%op** spécifie à Tom la manière dont un objet doit être vu comme un terme algébrique.

```
1 %typeterm Place extends Node {
2   implement     { petrinet.Place }
3   is_sort(t)    { t instanceof petrinet.Place }
4   equals(l1,l2) { l1.equals(l2) }
5 }

7 %op Place Place(name : String, marking : int) {
8   is_fsym(t) { t instanceof petrinet.Place }
9   get_slot(name, t) { t.eGet(t.eClass().getEStructuralFeature("name")) }
10  get_slot(marking, t) { t.eGet(t.eClass().getEStructuralFeature("marking")) }
11  get_default(marking) { 0 }
12  make(name, marking) { constructPlace(...) }
13 }
14 public static <O extends EObject> O constructPlace(O o, Object[] objs) { ... }
```

Listing 3 – Exemple de *mapping*

L'opérateur `Place` a deux arguments, de type `String`, et `int` ; ainsi qu'un codomaine de type `Place`. La construction **is_fsym** (ligne 8) est utilisée par l'algorithme de *pattern matching* pour vérifier si le constructeur courant (*i.e.* `Place`) est la racine de la représentation algébrique de l'objet Java (*i.e.* `t`). Les constructions **get_slot** (lignes 9 et 10) définissent comment récupérer les valeurs des champs dans la structure de données. La construction **get_default** (ligne 11) permet d'attribuer une valeur par défaut à un champ, ce qui évite d'écrire tous les champs lors de la création d'un terme. **make** (ligne 12) décrit comment construire une instance de l'élément. Elle est utilisée par le **‘** (*backquote*) pour construire un terme. En Tom+Java, on retrouvera typiquement un `new Place` dans cette construction. Un *mapping* peut être écrit à la main, mais dans cet exemple, il a été généré à partir du métamodèle présenté en figure 3 en utilisant le framework EMF (la syntaxe d'un *pattern* dépend donc fortement du métamodèle considéré). De ce fait, **make** est un peu plus complexe, ce qui explique que nous passons par une fonction Java intermédiaire `constructPlace` que nous ne détaillons pas ici.

### *6.3. Stratégies*

La programmation par stratégie permet de spécifier le contrôle de l'utilisateur sur l'application des règles de réécriture, en séparant le traitement (règles de réécriture) du parcours (traversée de l'arbre). L'utilisateur se concentrera donc sur l'élaboration de la partie métier du programme, puis il sera en mesure de définir des stratégies plus 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. Cela est rendu possible au sein de Tom *via* la construction **%strategy** illustrée par le Listing 4 ainsi que la bibliothèque de stratégies que nous fournissons.

```
1 %strategy Process2PetriNet() extends Identity() {
2   visit Process {
3     Process[name=n,from=f] -> { <Host+Tom code> }
4   }
5 }
6 ...
7 `TopDown(Process2PetriNet()).visit(root_process);
8 ...
```

Listing 4 – Exemple de construction %strategy

Bien que nous n'utilisions pas explicitement cette construction par la suite, les nouvelles constructions haut-niveau s'appuient dessus pour encoder les transformations de modèle. C'est pourquoi nous l'expliquons tout de même ici.
L'extrait de code du Listing 4 montre l'utilisation d'une stratégie appelée `Process2PetriNet`, dont le comportement par défaut est l'identité (*i.e.* ne fait rien, mais n'échoue pas). Ce qui signifie qu'aucune transformation n'a lieu lorsque la règle ne peut s'appliquer [5]. La construction `visit` (ligne 2) est un premier filtre qui spécifie la sorte des objets (`Process` dans l'exemple) sur lesquels la règle doit s'appliquer. Ensuite, une règle classique Tom composée d'un *pattern* et d'une action est définie. La principale particularité de la construction **%strategy** est qu'elle n'est pas automatiquement déclenchée. Son application doit être contrôlée par une stratégie. Dans l'exemple, l'expression `TopDown(Process2PetriNet())` (ligne 7) signifie que la règle `Process2PetriNet()` est appliquée de manière *top-down* sur le terme `root_process`. Composer des stratégies de réécriture avec des combinateurs permet d'élaborer des stratégies plus complexes. Pour plus d'informations sur le sujet, nous invitons le lecteur à se référer à (Balland *et al.*, 2008) et (Balland *et al.*, 2012), ainsi qu'à la page dédiée du manuel [6].

Le principe de fonctionnement de Tom ainsi que les fonctionnalités qu'il fournit ayant été présentés, nous pouvons expliquer l'implémentation technique de notre approche.

## 7. Outil pour la transformation de modèles

### *7.1. Représentation de modèles*

Un aspect de notre approche concerne le modèle de données utilisé pour représenter et manipuler les modèles. En effet, dans le monde des modèles, la structure de représentation est majoritairement le graphe. Une transformation de modèle revient alors à une transformation de graphe. La réécriture de graphe pour les transformations de modèles est donc une approche possible (Taentzer, 2010 ; Schürr, Klar, 2008) en utilisant des outils catégoriques tels que le *single-pushout* et le *double-pushout*, ainsi que le formalisme de spécification *triple graph grammar* (TGG). Dans le monde des graphes, on trouve des outils variés tels que Moflon (Amelunxen *et al.*, 2006) qui repose sur TGG, ou Henshin (Arendt *et al.*, 2010) qui repose sur AGG.

De notre côté, nous manipulons plutôt des termes algébriques qui sont des structures arborescentes. Compte tenu du fonctionnement de Tom, nous devons utiliser un langage hôte (Java) et nous avons choisi l'espace technologique EMF Ecore car très répandu. Il nous fournit un ensemble de services pour manipuler et reprèsenter les modéles en Java. Se pose donc le problème de la représentation des modèles EMF Ecore sous la forme de termes.

---

5. Par opposition à `Fail` qui spécifie que la transformation échoue si la règle ne peut être appliquée.
6. http://tom.loria.fr/wiki/index.php5/Documentation:Strategies

Un modèle EMF Ecore a une racine unique par la relation de composition. La structure arborescente engendrée par cette relation constitue un arbre de recouvrement du modèle. Nous sommes alors en mesure d'établir un *mapping* entre notre structure de terme algébrique et le modèle EMF Ecore. Nous avons donc pu développer un outil – Tom-EMF – permettant de générer automatiquement la représentation Tom d'un métamodèle sous la forme d'ancrages algébriques. Nous avons donc une vue d'arbre du modèle qui correspond à l'arbre par la relation de composition. Cette vision est naturelle pour l'utilisateur et il peut manipuler le modèle *via* via les mécanismes fournis par Tom. Cependant, il peut utiliser directement Java et les services EMF pour modifier les objets Java qui ont été instanciés.

Le Listing 3 est un *mapping* généré avec cet outil à partir du métamodèle des réseaux de Petri. Tout *EClassifier* (surclasse commune de *EClass* et *EDataType* permettant de spécifier les types d'opérations, de *structural features* et de paramètres) d'un métamodèle donne lieu à un ancrage algébrique. Ainsi, `Place` du métamodèle donné en figure 3 entraîne la génération des constructions **%typeterm** et **%op** correspondantes. Si le métamodèle spécifie qu'un élément *EClass* possède la propriété *abstract*, le constructeur (**%op**) n'est naturellement pas généré. C'est le cas de l'élément `Node`, dont `Transition` et `Place` héritent comme spécifié dans la figure 3. Cet exemple illustre aussi l'intégration du sous-typage simple [7] par Tom-EMF (ligne 1, Listing 3). Cela est possible du fait du moteur d'inférence de type de Tom qui est équipé du sous-typage (Kirchner *et al.*, 2009). Il rentre aussi en jeu dans le mécanisme *resolve*.

La relation de composition entraîne la génération de *mappings* dédiés. Dans notre cas d'étude, la relation de composition `nodes` entre `PetriNet` et `Node` (illustrée par figure 3) est modélisée au niveau des termes par un champ `node` de type `NodeEList` dans `PetriNet`. Le type `NodeEList` ainsi que son constructeur associé (opérateur variadique avec la construction **%oparray**) sont donc naturellement générés.

Cet outil génère une représentation d'un métamodèle dans le monde des termes Tom. Nous sommes donc ensuite en mesure de créer des modèles conformes à ce métamodèle sous forme de termes en utilisant la construction Tom **backquote** (section 6.1). Nous souhaitons pouvoir utiliser toutes les fonctionnalités du langage Tom sur ces modèles, en particulier les stratégies afin d'être capable de parcourir et réécrire un modèle comme n'importe quel autre terme. Pour ce faire, nous avons complété l'outil Tom-EMF par une bibliothèque dédiée à EMF Ecore. Nous ne détaillerons pas ici l'outil – `EcoreContainmentIntrospector` –, nous retiendrons seulement qu'il permet d'appliquer une stratégie sur un modèle. Il suit donc les liens de composition étant donné qu'il s'agit des liens donnant une représentation arborescente des modèles.

7. Par opposition au sous-typage multiple : en effet Ecore supporte l'héritage multiple, mais pas Java (au niveau des classes).

Être capable de générer la signature algébrique (le métamodèle pour les termes) et de pouvoir parcourir un modèle comme un terme constitue un pont entre les deux espaces technologiques. Nous pouvons donc opérer des transformations et des manipulations sur des termes puis éventuellement sérialiser le résultat sous forme d'un fichier .xmi pour repasser dans un environnement plus classique des modèles. Pour manipuler ces modèles, nous proposons aussi une extension du langage Tom que nous présentons dans la section suivante.

### *7.2. Langage de transformation*

En support de notre approche par décomposition et réconciliation décrite section 5, nous avons illustré notre méthode par un prototype écrit en Tom+Java et EMF dans (Bach *et al.*, 2012). Dans ce prototype, toute la transformation était écrite à la main, sans outil de génération additionnel ni de constructions Tom de haut niveau dédiées à la transformation de modèles. Ainsi, les stratégies permettaient d'encoder les transformations élémentaires (*définitions*) et les éléments *resolve* étaient définis explicitement par l'utilisateur (structure de données Java et ancrages algébriques Tom). La phase de résolution était elle aussi totalement définie par l'utilisateur sous la forme d'une stratégie supplémentaire. Il revenait ensuite à l'utilisateur de composer une stratégie à partir de toutes ces stratégies pour former la transformation complète. Ce prototype nous permettait d'implémenter le principe de notre approche, sans pour autant faciliter le développement d'une transformation de modèle.

C'est pourquoi nous proposons d'étendre le langage par des constructions haut niveau permettant d'automatiser un partie de l'écriture de la transformation. Ce langage doit aider l'utilisateur en lui épargnant l'écriture des structures de données des éléments *resolve* et des ancrages algébriques correspondants, de la phase de résolution, ainsi que celle de la stratégie de transformation complète. L'utilisateur peut se concentrer sur l'écriture des *définitions* uniquement.

Trois constructions supplémentaires s'avèrent nécessaires :

- pour exprimer la transformation ;
- pour créer les éléments temporaires *resolve* ;
- pour traçer les éléments créés.

La section 7.2.1 donne succinctement l'intuition et la forme générale d'une transformation de modèle comme présenté dans notre approche, tandis que la section 7.2.2 présente concrètement les constructions instanciées avec notre cas d'étude.

#### *7.2.1. Description d'une transformation de modèle*

Une transformation de modèle est composée de *définitions*. Ces *définitions* sont des ensembles de règles de réécriture Tom : elles sont 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. Les *définitions* sont compilées en des stratégies

élémentaires, puis composées avec des combinateurs pour former la stratégie de transformation globale. Une transformation de modèle avec Tom, sera de la forme :

```
Transformation MyTransfo
  Definition Def1 Traversal s1
    Pattern 1,1 -> Action 1,1
    Pattern 1,2 -> Action 1,2
  Definition Def2 Traversal s2
    Pattern 2,1 -> Action 2,1
    Pattern 2,2 -> Action 2,2
```

Elle donnera lieu à une stratégie $MyTransfo = Sequence(s1, s2)$, où $s1$ et $s2$ sont des stratégies impliquant respectivement `Def1` et `Def2`.

### *7.2.2. Exemple d'utilisation des constructions Tom dédiées aux transformations de modèles*

Nous nous proposons de partir d'un exemple simplifié de notre cas d'étude (Listing 5) pour illustrer l'utilisation de notre langage :

```
1  %transformation SimplePDLToPetriNet(links:LinkClass,pn:PetriNet) :
2   simplepdl.ecore -> petrinet.ecore {
3     definition WD2PN traversal `BottomUp(WD2PN(links,pn)) {
4       wd@WorkDefinition[name=n] -> {
5         ...
6         Process p = `wd.getParent();
7         Transition source = %resolve(p:Process,t_start:Transition);
8         ...
9       }
10    }
11    definition P2PN traversal `TopDown(P2PN(links,pn)) {
12      p@Process[name=n,from=f] -> {
13        ...
14        %tracelink(t_start:Transition,`Transition[name=`n]);
15        ...
16      }
17    }
18    definition WS2PN traversal `TopDown(WS2PN(links,pn)) {
19      ws@WorkSequence -> { ... }
20    }
21  }
```

Listing 5 – Exemple d'utilisation des constructions %transformation, %tracelink et %resolve

La transformation `SimplePDLToPetriNet` prend deux paramètres (ligne 1) : `links` – le modèle de lien – et `pn` – le modèle cible – qui seront peuplés et utilisés durant la transformation. Le modèle de lien permet de maintenir les relations entre les éléments de la source et ceux de la cible. Cette transformation utilise les métamodèles source `simplepdl.ecore` et cible `petrinet.ecore` (ligne 2).

Comme expliqué dans la section 3, la transformation globale est composée de trois transformations élémentaires nommées, définies par des blocs **definition** : `WD2PN` (ligne 3), `P2PN` (ligne 11) et `WS2PN` (ligne 18). Le mot-clef **traversal** permet de spécifier la stratégie encodant la *définition* qui entrera finalement dans la composition de

la stratégie `SimplePDLToPetriNet` : en effet, chaque *définition* est encodée par une stratégie qui est mise en séquence avec les autres stratégies générées. L'utilisateur averti peut moduler la stratégie qu'il spécifie via **traversal** plus largement qu'un simple changement de stratégie de parcours (*TopDown*, *etc.*) : il peut ainsi utiliser une stratégie développée hors de la transformation (et ayant un nom différent de la *définition* ou le compilateur détectera une erreur de définition multiple de stratégie). Dans ce cas, il n'est plus possible de garantir quoi que ce soit sur la transformation. Nous déconseillons cet usage qui ne doit être fait que pour des cas spécifiques bien maîtrisés par l'utilisateur (stratégies identiques mais avec ajout de *debug*, comparaisons d'implémentations de *définitions*, *etc.*). Comme illustré dans le Listing 5, le nom des *définitions* (`WD2PN`, `P2PN` et `WS2PN`) ainsi que les paramètres (`links` et `pn`) peuvent apparaître dans les stratégies. Les parties *action* de chaque *définition* sont des blocs de code Tom+Java dans lesquels les éléments du langage cible (places, transitions et arcs) sont créés.

La première *définition* – `WD2PN` – a besoin d'un élément créé dans une autre *définition* – `P2PN` – pour créer les arcs de synchronisation. Cependant, il n'y a aucune garantie que la précédente *définition* ait déjà été exécutée. Il est donc nécessaire d'introduire un élément intermédiaire *resolve* par la construction **%resolve** (ligne 7). Outre le fait d'instancier un objet *resolve* comme le ferait de manière approximativement équivalente : `Transition source = `ResolveProcessTransition(p, "t_start");` cette instruction permet d'étendre le métamodèle cible par l'ajout d'un élément de type `ResolveProcessTransition`, sous-type de `Transition`. Naturellement, l'extension du métamodèle cible s'accompagne de la génération de la structure de données Java ainsi que de l'ancrage algébrique Tom *ad-hoc*. En plus de jouer le rôle de *placeholder* et de pouvoir être manipulé comme s'il s'agissait d'une transition classique, cet élément *resolve* maintient une relation entre un élément du modèle source (le processus `p` tel qu'obtenu ligne 6) et l'élément du modèle cible qu'il est censé représenter (dans notre cas la transition `t_start` issue de la transformation de `p` dans une autre *définition*).

Dans la deuxième *définition* – `P2PN` –, plutôt que de créer classiquement une transition $t_{start}$, nous traçons la création de ce terme par la construction **%tracelink** (ligne 14). Cela sera équivalent à créer un terme via la construction *backquote* (*e.g.* `Transition t_start = `Transition[name=`n];`), tout en sauvant la référence dans le modèle de lien que nous générons durant la transformation. Ce modèle de lien correspond à une structure de données type *HashMap* permettant de maintenir les relations *éléments sources* → *éléments cibles*. Il fait correspondre une structure référençant les éléments cibles créés dans une *définition* avec les éléments sources dont ils sont issus. Un squelette du modèle de lien est généré en fonction des *définitions*, il est ensuite peuplé en suivant les instructions **%tracelink** de la transformation.

La phase de résolution – matérialisée par une stratégie – est entièrement générée dans le cas où une construction **%resolve** est utilisée. Dans le Listing 5, nous voyons la définition d'une transformation de modèle EMF Ecore. Du fait qu'elle est totalement

générée, la phase de résolution n'apparait pas dans le code Tom+Java. Le Listing 6 ci-dessous montre l'utilisation d'une transformation au sein d'un programme Tom+Java.

```
1 public static void main(String[] args) {
2  ...
3  //source model to transform
4  Process p_root = `Process(...);
5  //links model to keep track of links
6  LinkClass links = new LinkClass();
7  //target resulting model, empty at the beginning
8  PetriNet pn = `PetriNet(...);
9  //transformation is a strategy:
10 Strategy transformer = `SimplePDLToPetriNet(links,pn);
11 //call of transformer on the source model
12 transformer.visit(p_root, new EcoreContainmentIntrospector());
13 //links resolution phase:
14 `TopDown(tom__StratResolve_SimplePDLToPetriNet(links,pn)).visit(pn,
15                                  new EcoreContainmentIntrospector());
16 ...
```

Listing 6 – Exemple d'utilisation de transformation au sein d'un programme Tom+Java

Le modèle source à transformer est créé ligne 4 comme n'importe quel autre terme Tom. Il est aussi possible de charger une instance `.xmi` du métamodèle SimplePDL plutôt que d'utiliser la construction *backquote*. Le modèle de lien (ligne 6) ainsi que le modèle cible (ligne 8) sont initialisés (vides) afin d'être peuplés durant la transformation. La transformation définie dans le Listing 5 étant encodée par une stratégie complexe (composée de stratégies de parcours et des stratégies élémentaires), on l'instancie comme toute stratégie (ligne 10). On applique ensuite cette transformation sur le sujet (processus racine, `p_root`) *via* la méthode `visit`, implémentée par toute stratégie. Pour terminer la transformation, la deuxième phase – ou phase de résolution – doit être exécutée : la stratégie de résolution générée – `tom__StratResolve_SimplePDLToPetriNet` – est donc appliquée sur le résultat obtenu lors de l'application de la transformation précédente (ligne 14). Après cette étape, le réseau de Petri résultant `pn` est une instance conforme au métamodèle PetriNet décrit dans `petrinet.ecore`, ce que l'on peut facilement vérifier en le sérialisant et en l'important dans Eclipse.

### *7.3. Travail connexe*

Dans le même esprit que Tom, des outils de réécriture peuvent être adaptés afin d'implémenter des transformations de modèles. Plusieurs expériences ont été menées avec des encodages de modèles variés, et ont conduit à l'utilisation de méthodes existantes ou de nouveaux outils. On notera le langage Maude (Clavel *et al.*, 1996), qui fournit des services orientés objet pouvant être utilisés pour implémenter des métamodèles et des modèles. Son utilisation a été expérimentée par plusieurs équipes de recherche (Romero *et al.*, 2007 ; Boronat, Meseguer, 2009 ; Rusu, 2011) et implémentée par exemple dans le projet Moment [8]. Comme Tom-EMF, Moment rend

8. http://moment.dsic.upv.es/

interopérable l'environnement de spécification algébriques Maude avec l'espace technologique EMF. Une différence notable entre les environnements Tom et Maude est le fait que Maude est environnement complet tandis que Tom s'intègre dans les langages généralistes et a donc une approche hybride. Si Maude maîtrise totalement son environnement, ce n'est pas le cas de Tom qui n'a pas le contrôle du langage hôte. D'autres outils de réécriture de termes qui ont servi à la transformation de programmes tels que ASF+SDF, Spoofax– basé sur Stratego/XT – (Kats, Visser, 2010 ; Hemel *et al.*, 2010) ou Rascal (Klint *et al.*, 2009) peuvent aussi servir à la transformation de modèles, l'un des soucis étant de passer des termes aux graphes. Nous avons expliqué notre solution pour résoudre ce problème dans la section 7.1.

## 8. Évaluation et perspectives

Dans cette section, nous discutons notre approche, nos choix technologiques, leur intérêt ainsi que les limitations et perspectives de notre implémentation technique.

***Approche hybride.*** Notre choix d'utiliser Tom est intéressant car nous nous plaçons à la frontière de deux espaces technologiques qui sont rarement tous deux connus par les développeurs. Cette approche nous permet d'utiliser des concepts et outils formels tout en restant accessible à des utilisateurs non académiques. Il s'agit d'aider l'utilisateur à améliorer la qualité de son logiciel sans pour autant le pousser à changer radicalement d'environnement, d'outil, de langage et d'habitudes. Cela permet de limiter le coût du développement par rapport à l'adoption d'outils complètement spécifiques et dédiés.

***Langage de transformation.*** Concernant l'expression de la transformation elle-même ainsi que l'implémentation de l'approche en deux temps avec résolution, nous ne devrions pas apporter de changements fondamentaux. Il s'agit d'une approche relativement classique dans le monde des transformations de modèles qui a montré son efficacité. Les évolutions à venir viseront essentiellement l'amélioration de l'expérience utilisateur et la simplification de la prise en main des outils :

– L'introduction de raccourcis syntaxiques permettra de spécifier plus simplement la stratégie après le mot-clef **traversal**. Dans le cas le plus courant, l'utilisateur suit nos recommandations et ne souhaite véritablement spécifier que le type de parcours et non toute la stratégie (avec les paramètres la plupart du temps identiques à ceux de la transformation) ;

– Une simplification de l'utilisation du modèle de lien est aussi à l'étude afin que l'utilisateur n'ait plus à le spécifier explicitement ;

– Nous souhaitons alléger la syntaxe de la construction **%resolve** pour ne plus avoir à donner explicitement les types en les inférant.

***Évaluation et limitations.*** Nous avons essayé de trouver des usages pour lesquels nos outils étaient particulièrement intéressants ou au contraire pour lesquels ils apportaient peu. Nous les avons notamment utilisés pour implémenter l'exemple de l'applatissement d'une hiérarchie de classes (les feuilles de l'arbre d'héritage récupèrent tous les attributs de leurs surclasses). Cet exemple nous semblait intéressant du fait que la

relation de composition n'est pas centrale et que notre outil est calibré pour la relation de composition.

Nous avons constaté que pour un tel exemple dont l'implémentation récursive en Java, Java+Tom classique (sans la partie modèle) ou ATL est triviale, les constructions dédiées aux modèles n'apportent aucun véritable gain. Elles complexifient même la transformation qui devient moins lisible (et donc plus difficilement maintenable), sans véritablement tirer parti des outils développés. Nous constatons aussi que **%transformation** prend beaucoup d'intérêt lors de l'utilisation d'éléments *resolve*, car les structures de données ainsi que la phase de résolution sont générées automatiquement (minimisation du code écrit par le développeur, diminution des sources potentielles de *bugs* et donc plus grande fiablilité du logiciel). En revanche, cet exemple n'en nécessitant pas, le code est plus concis et lisible avec du Tom classique sans construction **%transformation**. Cet exemple nous a aussi amené à réfléchir à l'extension de l'*introspecteur* à d'autres liens que les liens de composition (paramétrisation par les liens).

***Traçabilité.*** Ce faible intérêt apparent pour les transformations les moins complexes peut être compensé par un autre axe de travail actuel : la traçabilité des transformations. Elle est actuellement assurée par le modèle de lien et la construction **%tracelink**, cependant cela comprend deux aspects de la traçabilité que nous souhaitons distinguer : la traçabilité *technique* et de *spécification*.
La traçabilité *technique* est la traçabilité utilisée en interne pour la transformation elle-même (pour la phase de résolution). Bien qu'extrêmement utile à des fins de debug, Cette traçabilité proche de l'implémentation peut être très éloignée d'une spécification. La certification d'un logiciel peut par conséquent être particulièrement ardue avec cette unique traçabilité, qui est d'ailleurs implémentée dans ATL (Jouault, 2005) et Kermeta (Falleri *et al*., 2006).

La traçabilité de *spécification* peut quant à elle être décorrélée de l'implémentation technique. Elle décrit des relations entre des sources et des cibles comme une spécification peut le faire, sans forcément adopter le découpage des règles de l'implémentation. Nous souhaitons séparer clairement ces deux traçabilités et donc découper la construction **%tracelink** en deux constructions distinctes qui joueront chacune un rôle spécifique. Techniquement, il nous faudra lever la limitation actuelle de ne pouvoir lier qu'une seule source à plusieurs cibles. Cela nous permettra de travailler à la généralisation de la traçabilité d'une transformation pour les approches hybrides, et à l'ajout de la traçabilité dans les transformations écrites avec des langages généralistes grand public.

***Enrichissement de l'expressivité du langage.*** Nous nous distinguons d'ATL par notre volonté de nous intégrer totalement à des langages généralistes (et particulièrement Java). Nous souhaitons aussi étendre et généraliser le mécanisme *resolve* à des sources multiples, ce que ne permet pas ATL. Actuellement, un élément *resolve* (ou le *resolveTemp* d'ATL) ne lie qu'un élément source à un élément cible. Outre la déjà très grande expressivité du langage de *patterns* de Tom, nous souhaitons améliorer l'expressivité de notre extension en offrant la possibilité du *resolve* multi-sources

ainsi que des règles multi-*patterns*. Cela passe par l'élaboration de nouveaux mécanismes tels que des *patterns* paramétrés ou des stratégies multi-sujets sur lesquelles nous travaillons actuellement.

***Modularité.*** Ensuite, dans l'optique d'accroître la modularité de notre approche – et donc la réutilisabilité des transformations –, un nouvel axe de travail est de décomposer la phase de résolution jusqu'alors monolithique en *résolutions élémentaires*. Si une définition pouvait être réutilisée dans une autre transformation, il était nécessaire de générer (ou éventuellement développer) une nouvelle phase de résolution adaptée à la nouvelle transformation. En décomposant cette phase en briques élémentaires, nous rendons la phase de résolution modulaire et réutilisable. Une définition – accompagnée des résolutions élémentaires associées – pourra donc être complètement portable dans d'autres transformations.

***Usage industriel.*** Bien que les versions courantes du langage et des outils puissent être encore améliorées, l'ensemble est déjà utilisable dans un contexte non académique. Nos outils ont été utilisés par nos partenaires industriels durant le projet Quarteft [9] financé par la FRAE [10].

***Performances.*** Des améliorations très récentes de nos outils nous ont permis d'obtenir d'excellentes performances sur de très gros modèles. Nous sommes maintenant en mesure de transformer des modèles dont la taille est de l'ordre de plusieurs millions d'éléments sur des machines « raisonnables » (serveurs ou machines de développement accessibles pour une petite structure n'ayant pas les moyens d'avoir une grille ou un serveur de calcul).

## 9. Conclusion

Dans cet article, nous avons décrit notre approche pour transformer des modèles. Nous avons montré comment le filtrage de motif et les vues algébriques peuvent être utilisés pour encoder des transformations de modèles d'une manière plus abstraite qu'en pur Java. Nous avons montré comment nous pouvions exprimer une transformation de modèles en utilisant les stratégies Tom pour encoder les transformations élémentaires la composant.

La méthode adoptée dans notre approche permet de ne plus avoir à gérer l'ordonnancement des pas de transformation. Pour cela nous avons procédé à la décomposition en deux phases distinctes de la transformation, la phase de création des éléments cibles d'une part, et la phase de résolution de liens d'autre part. La première étape est elle-même découpée en transformations plus simples – les transformation élémentaires ou *définitions* –, qui sont appliquées en séquence dans un ordre non géré par l'utilisateur. Durant cette première phase, nous avons introduit des éléments intermédiaires – éléments *resolve* dérivés de QVT – qui viennent temporairement enrichir le

9. http://www.quarteft.loria.fr
10. Fondation de Recherche pour l'Aéronautique et l'Espace : http://www.fnrae.org

modèle cible lorsqu'une transformation élémentaire le nécessite. Nous avons aussi introduit un *modèle de lien* qui permet, d'une part, de maintenir les relations entre les éléments sources et cibles, et, d'autre part, d'assurer la traçabilité de la transformation. La deuxième étape consiste à résoudre les liens entre les éléments cibles créés durant la première phase. Il s'agit de retrouver les éléments *resolve* et de les remplacer par les éléments cibles qu'ils remplaçaient temporairement à l'aide du modèle de lien.Nous avons aussi proposé une extension du langage Tom pour exprimer les transformations haut-niveau en nous appuyant sur le mécanisme des stratégies Tom qui encodent toutes les opérations (définitions, phases de création d'éléments et de résolution ainsi que transformation complète) de la transformation.

Notre approche à mi-chemin entre les langages généralistes et les langages dédiés ainsi que les technologies sur lesquelles nous nous appuyons nous permettent de nous intégrer aisément dans le monde Java. Cependant, ce que nous avons présenté est suffisamment générique et extensible pour s'adapter à d'autres langages et d'autres *frameworks* de modélisation. À ce titre, nous expérimentons la généralisation de nos outils au cas du langage Ada acconpagné de son *framework* GMS basé sur Ecore (en cours d'élaboration).

Outre les améliorations techniques de nos outils, la suite logique et naturelle de ce travail est la traçabilité des transformations et l'utilisation du modèle de lien à des fins de vérification. L'ingénierie des modèles étant adoptée par l'industrie pour la conception et le développement des systèmes critiques, il est nécessaire d'assurer la fiabilité du logiciel. Se pose donc la problématique des transformations de modèles qualifiables et de leur vérification. Lors d'une qualification de logiciel, les tâches étant manuelles, une trace de transformation pouvant être utilisée avec d'autres outils dédiés (ajout de contraintes OCL par exemple) est un gain considérable pour l'utilisateur. Notre approche hybride étant bien intégrée dans le monde Java nous comptons apporter des éléments de confiance utiles pour la qualification de logiciels dans le monde Java.

## Bibliographie

Amelunxen C., Königs A., Rötschke T., Schürr A. (2006). MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink, J. Warmer (Eds.), *Ecmda-fa*, vol. 4066, p. 361-375. Springer.

Arendt T., Biermann E., Jurack S., Krause C., Taentzer G. (2010). Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In D. C. Petriu, N. Rouquette, Ø. Haugen (Eds.), *Models (1)*, vol. 6394, p. 121-135. Springer.

Bach J.-C., Crégut X., Moreau P.-E., Pantel M. (2012). Model transformations with Tom [inproceedings]. In *LDTA*, p. 16. Tallinn, Estonia, ACM. Consulté sur http://doi.acm.org/10.1145/2427048.2427052

Balland E., Brauner P., Kopetz R., Moreau P.-E., Reilles A. (2007). Tom: Piggybacking Rewriting on Java. In *Proceedings of the 18th international conference on term rewriting and applications*, p. 36–47. Springer-Verlag.

Balland E., Kirchner C., Moreau P.-E. (2006). Formal islands. In *Proceedings of the 11th international conference on algebraic methodology and software technology*, p. 51–65. Springer-Verlag.

Balland E., Moreau P.-E., Reilles A. (2008). Rewriting Strategies in Java. *Electr. Notes Theor. Comput. Sci.*, vol. 219, p. 97-111.

Balland E., Moreau P.-E., Reilles A. (2012). Effective strategic programming for java developers. *Software: Practice and Experience*.

Boronat A., Meseguer J. (2009). MOMENT2: EMF Model Transformations in Maude. In A. Vallecillo, G. Sagardui (Eds.), *Jisbd*, p. 178-179.

Brand M. van den, Heering J., Klint P., Olivier P. (2002). Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, vol. 24, n° 4, p. 334-368.

Clavel M., Eker S., Lincoln P., Meseguer J. (1996). Principles of Maude. *Electronic Notes in Theoretical Computer Science*, vol. 4, n° 0, p. 65 - 89. (RWLW96, First International Workshop on Rewriting Logic and its Applications)

Combemale B. (2008). *Approche de métamodélisation pour la simulation et la vérification de modèle – Application à l'ingénierie des procédés*. Thèse de doctorat non publiée.

Falleri J.-R., Huchard M., Nebut C. *et al*. (2006). Towards a traceability framework for model transformations in kermeta. In *Ecmda-tw'06: Ecmda traceability workshop*, p. 31–40.

Hemel Z., Kats L. C. L., Groenewegen D. M., Visser E. (2010). Code generation by model transformation: a case study in transformation modularity. *Software and System Modeling*, vol. 9, n° 3, p. 375-402.

Jakumeit E., Buchwald S., Kroll M. (2010). GrGen.NET. *International Journal on Software Tools for Technology Transfer*, vol. 12, p. 263-271.

Jézéquel J.-M., Barais O., Fleurey F. (2011). Model driven language engineering with Kermeta. In *Proceedings of the 3rd international summer school conference on Generative and Transformational Techniques in Software Engineering III*, p. 201–221. Berlin, Heidelberg, Springer-Verlag.

Jouault F. (2005). Loosely Coupled Traceability for ATL. In *In proceedings of the european conference on model driven architecture (ecmda) workshop on traceability*, p. 29–37.

Jouault F., Allilaire F., Bézivin J., Kurtev I. (2008, 01 juin). ATL: A model transformation tool. *Science of Computer Programming*, vol. 72, n° 1-2, p. 31–39.

Kats L. C. L., Visser E. (2010). The spoofax language workbench: rules for declarative specification of languages and IDEs. In *Oopsla*, p. 444-463.

Kirchner C., Moreau P.-E., Tavares C. (2009). A Type System for Tom. In *Rule*, p. 51-63.

Klint P., Storm T. van der, Vinju J. J. (2009). RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Scam*, p. 168-177. IEEE Computer Society.

Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0 Manuel de logiciel. (2008, avril).

Moreau P.-E., Ringeissen C., Vittek M. (2003). A pattern matching compiler for multiple target languages. In *Proceedings of the 12th international conference on compiler construction*, p. 61–76. Springer-Verlag.

Muller P.-A., Fleurey F., Jézéquel J.-M. (2005). Weaving Executability into Object-Oriented Meta-languages. In L. C. Briand, C. Williams (Eds.), *Models*, vol. 3713, p. 264-278. Springer.

OMG. (2006a). *Meta Object Facility (MOF) Core Specification Version 2.0.* Object Management Group.

OMG. (2006b). *Object Constraint Language Specification (OCL) 2.0 Specification.* Object Management Group.

Romero J. R., Rivera J. E., Durán F., Vallecillo A. (2007). Formal and Tool Support for Model Driven Engineering with Maude. *Journal of Object Technology*, vol. 6, n° 9, p. 187-207.

Rusu V. (2011). Embedding domain-specific modelling languages in maude specifications. *ACM SIGSOFT Software Engineering Notes*, vol. 36, n° 1, p. 1-8.

Schürr A., Klar F. (2008). 15 Years of Triple Graph Grammars. In H. Ehrig, R. Heckel, G. Rozenberg, G. Taentzer (Eds.), *Icgt*, vol. 5214, p. 411-425. Springer.

Sendall S., Kozaczynski W. (2003). Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, vol. 20, n° 5, p. 42–45.

Steinberg D., Budinsky F., Paternostro M., Merks E. (2009). *EMF: Eclipse Modeling Framework 2.0* (2nd éd.). Addison-Wesley Professional.

Taentzer G. (2010). What Algebraic Graph Transformations Can Do For Model Transformations. *ECEASST*, vol. 30.

Varró D., Balogh A. (2007). The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, vol. 68, n° 3, p. 214 - 234. (Special Issue on Model Transformation)

Varró D., Varró G., Pataricza A. (2002). Designing the automatic transformation of visual languages. *Science of Computer Programming*, vol. 44, n° 2, p. 205 - 227.

Visser E. (2001). Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5. In A. Middeldorp (Ed.), *Rewriting techniques and applications*, vol. 2051, p. 357-361. Springer Berlin Heidelberg.

Visser E. (2004). Program Transformation with Stratego/XT. In C. Lengauer, D. Batory, C. Consel, M. Odersky (Eds.), *Domain-specific program generation*, vol. 3016, p. 216-238. Springer Berlin Heidelberg.

W3C. (1999, nov). *XSL Transformations (XSLT) Version 1.0.* Object Management Group.

Article reçu le 15/01/2013
Accepté après révisions le 24/10/2013

Jean-Christophe Bach *termine son doctorat en informatique au LORIA (UMR 7503) à Nancy. Membre de l'équipe Pareo, ses travaux portent sur l'utilisation de la réécriture pour le développement logiciel qualifié, en particulier sur la définition d'une extension du langage Tom pour transformer des modèles par réécriture dans un environnement Java.*