Une approche hybride GPL-DSL pour transformer des modèles

Jean-Christophe Bach

Une approche hybride GPL-DSL pour transformer des modèles

Jean-Christophe Bach[1,2,3]

[1] Inria, Villers-lès-Nancy, F-54600, France
[2] Université de Lorraine, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54500, France
[3] CNRS, LORIA, UMR 7503, 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.

Keywords: transformations de modèles, langage, tom, java, langage de transformation, EMF, Ecore

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.

Keywords: models transformations, language, tom, java, transformation language, EMF, Ecore

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 – parait simple, exprimer un 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 [1], ATL [4] [2], Kermeta [5] [3,4], *etc.*, ou l'usage de langages généralistes (Java par exemple) équipés de *frameworks* tels que EMF [6] [5]. 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, déboggueur, 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

4. `http://www.eclipse.org/atl/`
5. `http://www.kermeta.org/`
6. `http://www.eclipse.org/emf/`

cela, nous nous appuyons sur le langage Tom [7] qui a été conçu dans l'optique d'enrichir des 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 5.

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 ;
- traçabilité : notre approche permet de générer des traces qui peuvent être utilisées à des fins de vérification ou de debug ;
- 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.

Dans cette optique, nous présentons donc 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 en 6 autres sections. Après des remarques préliminaires en Section 2, nous expliquons le cas d'étude sur lequel nous appuyons nos exemples dans la Section 3. Puis nous détaillons notre approche dans la Section 4. Dans la Section 5, nous décrivons rapidement le langage Tom, pour pouvoir ensuite expliquer la mise en œuvre de notre approche dans la Section 6. Nous concluons et exposons nos perspectives dans la Section 7.

2 Notions préliminaires

Dans cette partie, nous définissons et précisons certains termes et notations que nous utiliserons dans cet article.

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

Definition 2 (Métamodèle). *Un métamodèle est un modèle qui définit le langage d'expression d'un modèle [6], 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.*

La notion de métamodèle a été formalisée par l'OMG [8] dans le standard Meta Object Facility [9] (MOF) comme un sous-ensemble du diagramme de classe UML. Intuitivement, un métamodèle est composé d'un ensemble de métaclasses

7. `http://tom.loria.fr`
8. Object Management Group, `http://www.omg.org`
9. `http://www.omg.org/mof`

qui contiennent des *attributs* et des *opérations* comme les classes en programmation orientée object. Les métaclasses peuvent être liées par héritage, et par une métarelation (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.

Definition 3 (Transformation de modèle). *Une transformation de modèles 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 [7]. Il existe au moins deux approches principales pour décrire une transformation de modèles (que QVT propose d'ailleurs) :

- 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 modèles 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.

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 [8] 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 [9] ou tout autre langage de transformation comme Stratego/XT, ASF+SDF, Rascal, etc.

Note 1 (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 [10] : 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 du 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 Figure 1.

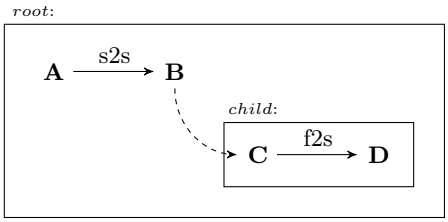

FIGURE 1: Processus, instance de SimplePDL.

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* composé de *ProcessElement*s. Chaque *ProcessElement* peut être une *WorkDefinition* ou une *WorkSequence*. 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). Enfin, une *WorkDefinition* peut elle-même être définie par un processus imbriqué (référence *process*), ce qui permet de définir des processus hiérarchiques.

Nous souhaitons transformer le processus décrit et illustré Figure 1 afin de vérifier ses propriétés. 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*. Le métamodèle des réseaux de Petri est illustré par la Figure 3. 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*, soit de type *read_ arc*. Un arc spécifie le nombre de jetons (*weight* – poids –) consommé dans la place source ou produit 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*).

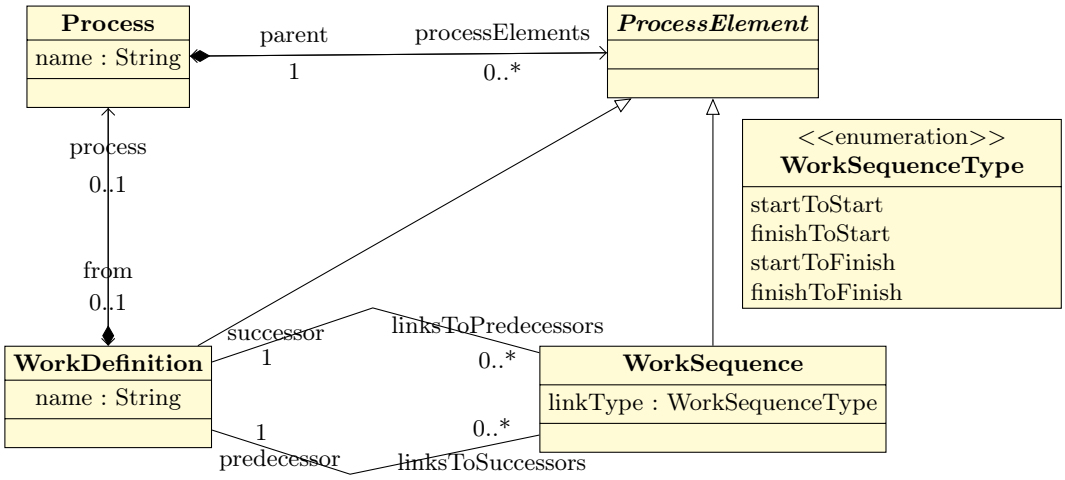

FIGURE 2: Métamodèle SimplePDL.

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 sont quant à eux matérialisés par des flèches – vertes dans le cas d'une séquence –. Celles en pointillés correspondent aux synchronisations entre éléments.
Ainsi, `Process2PetriNet` traduit un `Process` en un réseau de Petri représenté par le premier cas de la Figure 4. L'image d'un process est donc constituée de trois places (p_{ready}, $p_{running}$ and $p_{finished}$), deux transitions (t_{start} and t_{finish}) et quatre arcs.
`WorkDefinition2PetriNet` créera tous les éléments du réseau de Petri qui définissent l'image d'une `WorkDefinition` (deuxième cas 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 le fait qu'il y a une place supplémentaire.
Une `WorkSequence` est traduite par un `Arc` dans la transformation `WorkSequence2PetriNet`, dont la représentation est le troisième cas de la Figure 4, lorsqu'il s'agit d'une séquence *start2start*.

La transformation traduit chaque processus et activité en un réseau de Petri défini, et chaque séquence en un arc. Dans un second temps, lorsque les processus et les activités sont traduites, des arcs sont générés pour encoder la synchronisa-

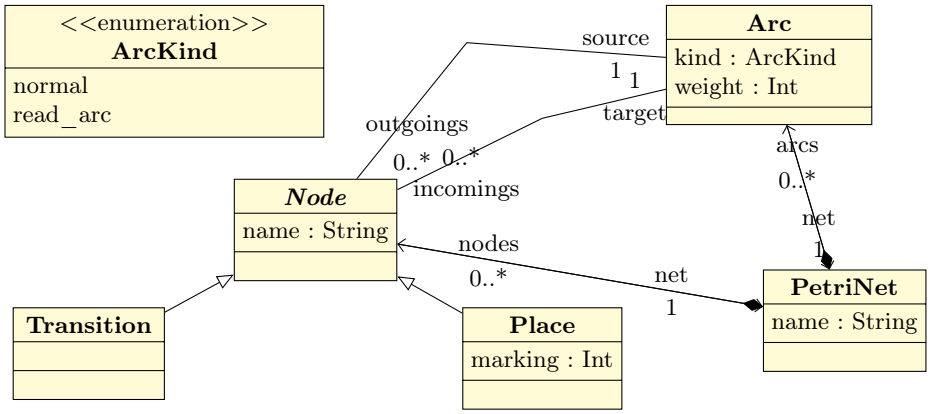

FIGURE 3: Métamodèle PetriNet.

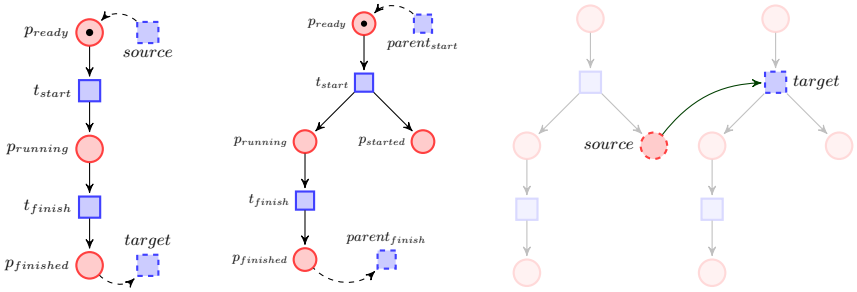

FIGURE 4: Réseaux de Petri résultant des transformations des éléments `Process`, `WorkDefinition` et `WorkSequence`.

tion entre les processus et leurs activités. La représentation graphique Figure 5 illustre le réseau de Petri résultant de notre exemple.

Pour écrire cette transformation de modèle, 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.

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, déboggueur, bibliothèques) ainsi que d'une communauté généralement étendue (et donc de documentation et d'un support possible), certaines tâches (retrouver et modifier une information dans la structure de données par exemple) précises peuvent se révéler fastidieuses ou difficiles à mettre en œuvre. Bien qu'il existe des *frameworks* dédiés à la transformation de modèles tels que EMF, et offrant de nombreuses possibilités, leur utilisation n'est

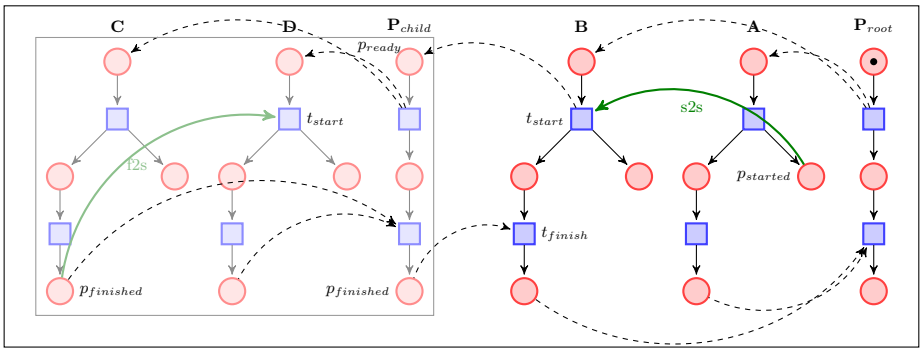

FIGURE 5: Réseau de Petri résultant de la transformation.

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 forcément 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 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 détaillons un peu le langage dans la section suivante. 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.

4 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 une intuition que nous formalisons ensuite, puis nous la détaillons.

Le principe de notre approche est de décomposer une transformation en deux phases distinctes. La première consiste à créer les éléments cible du modèle résultant ainsi que des éléments additionnels que nous appelons *éléments resolve*. Ces éléments permettent de faire référence à des éléments qui n'ont pas encore étét 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.

4.1 Formalisation de l'approche

Dans notre approche, une transformation T est donc constituée de deux phases distinctes qui peuvente être vues comme des fonctions. La première phase $c : MM_s \rightarrow MM_{t_{resolve}}$ consiste à créer des éléments cible à partir des éléments du modèle source. Des éléments *resolve* étant créés et intégrés au résultat durant cette phase, le modèle résultant est conforme au métamodèle cible étendu, noté $MM_{t_{resolve}}$.
Cette extension du métamodèle cible passe par un enrichissement du type cible (ajout d'informations additionnelles). Ainsi, tout élément *resolve* $e^i_{t_{resolve}}$ du modèle intermédiaire enrichi $m_{t_{resolve}}$ sera de type un sous-type de l'élément associé 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 6), 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.
La seconde phase $r : MM_{t_{resolve}} \rightarrow MM_t$ consiste à éliminer ces éléments intermédiaires.
La transformation complète $T : MM_s \rightarrow MM_t$ est définie par $T = r \circ c$.
En instanciant notre approche avec le cas d'étude SimplePDLToPetriNet, nous obtenons :

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

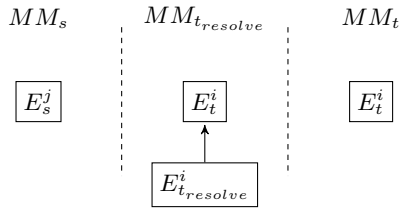

FIGURE 6: Schéma d'extension du métamodèle cible par l'ajout d'éléments intermédiaires *resolve*.

Appliquée au processus p_{root} (illustré Figure 1) conforme à $MM_{SimplePDL}$ (Figure 2), cette transformation produira un réseau de Petri pn (illustré Figure 5) conforme à $MM_{PetriNet}$ (Figure 3), en passant par le résultat intermédiaire $pn_{resolve}$ conforme au métamodèle $MM_{PetriNet_{resolve}}$. On obtient donc :

$SimplePDLToPetriNet(p_{root}) = Resolve(Transformer(p_{root}))$, avec
$Transformer(p_{root}) = pn_{resolve}$ et
$Resolve(p_{resolve}) = pn$

La Figure 7 instancie le schéma d'extension du métamodèle cible au cas d'étude : 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*.

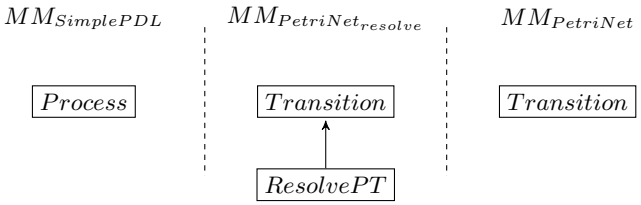

FIGURE 7: Instanciation du schéma d'extension du métamodèle cible pour le cas d'étude SimplePDLToPetriNetPetriNet.

4.2 Description de l'approche

Détaillons maintenant cette 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 la `Place` et la `Transition` le constituant. 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.

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 tout d'abord de décomposer une transformation complexe en transformations les plus simples (ce que nous nommons *transformations élémentaires* ou *définitions*). 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 de 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 [11]/VIATRA2 [12] et GrGen.NET [13] 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, chaque sortie d'une transformation élémentaire est une partie du résultat final.

Reste 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. Cela fait évidemment référence au *resolve* de QVT et au *resolveTemp* de ATL.

Le principe est que, du fait que toutes les transformations élémentaires peuvent être déclenchées indépendamment dans n'importe quel ordre (voire en parallèle), il faut tout de même 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 (comme décrit Section 4.1, Figure 6), 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. La Figure 8 permet d'illustrer le mécanisme des éléments *resolve*. 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 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*.

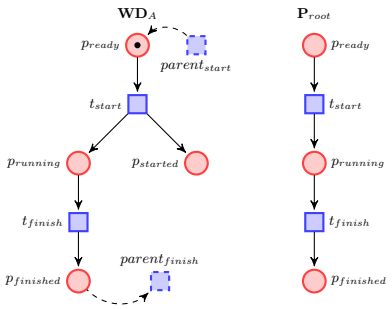

FIGURE 8: Illustration du mécanisme *resolve* avec les réseaux de Petri images d'une `WorkDefinition` et d'un `Process`.

Durant cette première phase, chaque définition a 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, il est nécessaire d'effectuer un traitement sur ce résultat intermédiaire. C'est ce que nous appelons 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.

Ce remplacement est possible grâce aux informations supplémentaires qui enrichissent le type cible (et qui font qu'un élément *resolve* est un élément *resolve*) ainsi qu'aux informations que nous sauvegardons durant la transformation. Ces informations additionnelles sont des informations de relations entre les éléments cible et les éléments source 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 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 source et les éléments cible qui en sont issus. Outre son usage interne pour la phase de résolution, le modèle de lien nous permet d'assurer une forme de traçabilité de la transformation, et peut être utilisé *a posteriori*.

5 Aperçu du langage Tom

Dans cette section, nous présentons Tom, le langage qui va être utilisé pour rendre notre approche effective. Tom [14,15] est un langage basé sur le calcul de réécriture conçu pour intégrer de nouvelles fonctionnalités 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 [16] 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*). Nous allons passer en revue les différentes fonctionnalités à travers les constructions Tom.

5.1 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. 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 constructeurs `Place`, `Transition` et `NodeEList` ne sont pas internes au langage. Ils sont dérivés du métamodèle des réseaux de Petri donné en Figure 3.

5.2 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 constructeurs). 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 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 }

15 public static <O extends EObject> O constructPlace(O o, Object[] objs) {
16   int i=0;
17   EList<EStructuralFeature> sfes = o.eClass().getEAllStructuralFeatures();
18   for(EStructuralFeature esf : sfes) {
19     if(esf.isChangeable()) {
20       o.eSet(esf, objs[i]);
21       i++;
22     }
23   }
24   return o;
25 }
```

Listing 3: Exemple de *mapping*

Le constructeur `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`). Elle est aussi utilisée pour la génération de code. 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) est la construction qui 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 intermédiaire `constructPlace`.

5.3 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

Les stratégies encodent la notion de règle de transformation élémentaire.

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 [10]. 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 [17] et [18], ainsi qu'à la page dédiée du manuel [11].

5.4 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 [19], 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 recherches [20,21,22] et implémentée par exemple dans le projet Moment [12]. D'autres outils de réécriture de termes qui ont servi à la transformation de programmes tels que ASF+SDF, Spoofax [13] – basé sur Stratego/XT – [23,24] ou Rascal [25] peuvent aussi servir à la transformation de modèles, l'un des soucis étant de passer des termes aux graphes. Nous expliquons notre solution pour résoudre ce problème dans la Section 6.1.

10. Par opposition à `Fail` qui spécifie que la transformation échoue si la règle ne peut être appliquée.
11. `http://tom.loria.fr/wiki/index.php5/Documentation:Strategies`
12. `http://www.cs.le.ac.uk/people/aboronat/tools/moment2/`
13. `http://strategoxt.org/Spoofax`

6 Outillage pour la transformation de modèles

6.1 Représentation de modèles

Un aspect de notre approche concerne le modèle de données utilisé pour représenter et manipuler les modèles. En effet, lorsque l'on parle de modèles, on parle de graphes. 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 [26,27] 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 [14][28] qui repose sur TGG, ou Henshin [15][29] qui repose sur AGG.

EMF Ecore étant un standard de fait, nous avons choisi de nous appuyer en premier lieu sur cette technologie. Se pose donc le problème de la représentation des modèles EMF Ecore comme des termes. Cependant, grâce à la relation de composition ainsi qu'à la racine nécessaire d'un modèle EMF, la représentation d'un tel modèle comme un arbre est naturelle. Il nous a donc été possible de développer un outil – Tom-EMF – permettant de générer automatiquement la représentation Tom d'un métamodèle sous la forme de *mappings*. Nous manipulons donc des termes qui sont des vues des modèles que nous transformons.

Ainsi, tout *EClassifier* d'un métamodèle (qu'il soit un *EClass* ou un *EDataType* comme les *EEnum*) donne lieu à un ancrage algébrique. Le Listing 3 a été généré par Tom-EMF. À partir d'une `Place`, les constructions **%typeterm** et **%op** correspondantes sont générées. Si le métamodèle spécifie qu'un élément *EClass* possède la propriété *abstract*, le constructeur 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 [16] 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.

La relation de composition entraîne la génération de *mappings* dédiés. Dans notre cas d'utilisation, 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 de 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 5.1). 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

14. `http://www.moflon.org/`
15. `http://www.eclipse.org/modeling/emft/henshin/`
16. Par opposition au sous-typage multiple : en effet Ecore supporte l'héritage multiple, mais pas Java (au niveau des classes).

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'être interopérable avec les stratégies Tom.

6.2 Langage de transformation

En support de notre approche décrite Section 4, nous avons illustré notre méthode par un prototype écrit en Tom+Java et EMF dans [30]. Dans ce prototype, toute la transformation était écrite à la main, sans outil facilitateur. 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 métastratégie 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 plus 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, de la métastratégie de transformation. L'utilisateur doit 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.

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. Cette transformation utilise les métamodèles source `simplepdligne 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 qui composera la métastratégie `SimplePDLToPetriNet`.
Comme illustré 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 *élément source* → *éléments cible*. Il fait correspondre une structure référenc cant les élément cible créés dans une définition avec les éléments source 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.

Syntaxe abstraite du langage De cet exemple, nous constatons qu'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 les constructions Tom vu précédemment dans la Section 6.2, 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`.
De ce qui précède, nous pouvons déduire la syntaxe abstraite de notre langage, comme donné par le Listing 7 :

```
Transformation = Transformation(Name, Definition*)
Definition        = Definition(Name, TraversalStrategy, Rule*)
Rule              = Rule(Pattern, Action)

Resolve = Resolve(Identifier, Identifier, Identifier, Identifier)

Trace     = Trace(Identifier, Identifier, BackQuoteTerm)

Pattern : terme de la forme a(), g(x) ou f(x,g(a())) par
            exemple, se référer à la syntaxe officielle de Tom
Action : bloc de code Java+Tom
TraversalStrategy : stratégie de parcours paramétrant la
                        définition
Identifier : dans le cas de Resolve, ces 4 identifiants sont
               les noms et types des éléments source et cible.
               dans le cas de Trace, ces 2 identifiants sont
               le nom et le type de l'éléments à tracer.
BackQuoteTerm : terme Tom précédé d'un backquote, se référer
                    à la syntaxe de Tom
```

Listing 7: Syntaxe abstraite du langage de transformation

Note : La syntaxe concrète des constructions Tom dédiées aux transformations de modèles est donnée en A. Elle est à utiliser en conjonction de la syntaxe complète de Tom, disponible en ligne sur la page du site officiel [17].

7 Conclusions et perspectives

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 utilisées pour encoder des transformations de modèles d'une manière plus abstraite qu'en pur Java. Nous avons montré comment nous pouvions

17. `http://tom.loria.fr/wiki/index.php5/Documentation:Tom`

exprimer une transformation de modèles en utilisant les stratégies Tom pour encoder les transformations élémentaires la composant.

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 cible 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 arbitraire. Durant cette première phase, nous avons introduit des éléments intermédiaires – éléments *resolve* dérivés de QVT – qui viennent temporairement enrichir le 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 source et cible, 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 cible 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. Nous travaillons d'ailleurs sur le sujet de la généralisation de nos outils, et un nouveau cas d'utilisation pour valider l'extension des outils est le langage Ada accompagné de son *framework* de modélisation GMS basé sur Ecore (en cours d'élaboration).

Notre approche est proche de la solution apportée par ATL. Cependant, outre notre volonté de nous intégrer totalement à des langages généralistes (et particulièrement Java), nous souhaitons 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. Nous souhaitons améliorer l'expressivité de notre langage 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.

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 réuti-

lisable. Une définition – accompagnée des résolutions élémentaires associées – pourra donc être complètement portable dans d'autres transformations.

La suite logique et naturelle de ce travail est l'utilisation du modèle de lien à des fins de vérification. En effet, l'ingénierie des modèles se répandant dans un contexte industriel critique, il est nécessaire de 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.

Références

1. Object Management Group, Inc. : Meta Object Facility (MOF) 2.0 Query/View/-Transformation (QVT) Specification, version 1.0. (April 2008)
2. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I. : Atl : A model transformation tool. Science of Computer Programming **72**(1-2) (June 2008) 31–39
3. Jézéquel, J.M., Barais, O., Fleurey, F. : Model driven language engineering with kermeta. In : Proceedings of the 3rd international summer school conference on Generative and Transformational Techniques in Software Engineering III. GTTSE'09, Berlin, Heidelberg, Springer-Verlag (2011) 201–221
4. Muller, P.A., Fleurey, F., Jézéquel, J.M. : Weaving executability into object-oriented meta-languages. In Briand, L.C., Williams, C., eds. : MoDELS. Volume 3713 of Lecture Notes in Computer Science., Springer (2005) 264–278
5. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E. : EMF : Eclipse Modeling Framework 2.0. 2nd edn. Addison-Wesley Professional (2009)
6. OMG : Meta Object Facility (MOF) Core Specification Version 2.0. Object Management Group, Inc. (2006)
7. OMG : Object Constraint Language Specification (OCL) 2.0 Specification. Object Management Group, Inc. (2006)
8. Sendall, S., Kozaczynski, W. : Model transformation : the heart and soul of model-driven software development. Software, IEEE **20**(5) (2003) 42–45
9. W3C : XSL Transformations (XSLT) Version 1.0. Object Management Group, Inc. (nov 1999)
10. Combemale, B. : 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.
11. Varró, D., Varró, G., Pataricza, A. : Designing the automatic transformation of visual languages. Science of Computer Programming **44**(2) (2002) 205 – 227
12. Varró, D., Balogh, A. : The model transformation language of the viatra2 framework. Science of Computer Programming **68**(3) (2007) 214 – 234 Special Issue on Model Transformation.

13. Jakumeit, E., Buchwald, S., Kroll, M. : Grgen.net. International Journal on Software Tools for Technology Transfer **12** (2010) 263–271
14. Moreau, P.E., Ringeissen, C., Vittek, M. : A pattern matching compiler for multiple target languages. In : Proceedings of the 12th international conference on Compiler construction. CC'03, Springer-Verlag (2003) 61–76
15. Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A. : Tom : Piggybacking Rewriting on Java. In : Proceedings of the 18th international conference on Term rewriting and applications. RTA'07, Springer-Verlag (2007) 36–47
16. Balland, E., Kirchner, C., Moreau, P.E. : Formal islands. In : Proceedings of the 11th international conference on Algebraic Methodology and Software Technology. AMAST'06, Springer-Verlag (2006) 51–65
17. Balland, E., Moreau, P.E., Reilles, A. : Rewriting strategies in java. Electr. Notes Theor. Comput. Sci. **219** (2008) 97–111
18. Balland, E., Moreau, P.E., Reilles, A. : Effective strategic programming for java developers. Software : Practice and Experience (2012)
19. Clavel, M., Eker, S., Lincoln, P., Meseguer, J. : Principles of maude. Electronic Notes in Theoretical Computer Science **4**(0) (1996) 65 – 89 RWLW96, First International Workshop on Rewriting Logic and its Applications.
20. Boronat, A., Meseguer, J. : Moment2 : Emf model transformations in maude. In Vallecillo, A., Sagardui, G., eds. : JISBD. (2009) 178–179
21. Rusu, V. : Embedding domain-specific modelling languages in maude specifications. ACM SIGSOFT Software Engineering Notes **36**(1) (2011) 1–8
22. Romero, J.R., Rivera, J.E., Durán, F., Vallecillo, A. : Formal and tool support for model driven engineering with maude. Journal of Object Technology **6**(9) (2007) 187–207
23. Kats, L.C.L., Visser, E. : The spoofax language workbench : rules for declarative specification of languages and ides. In Cook, W.R., Clarke, S., Rinard, M.C., eds. : OOPSLA, ACM (2010) 444–463
24. Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E. : Code generation by model transformation : a case study in transformation modularity. Software and System Modeling **9**(3) (2010) 375–402
25. Klint, P., van der Storm, T., Vinju, J.J. : Rascal : A domain specific language for source code analysis and manipulation. In : SCAM, IEEE Computer Society (2009) 168–177
26. Taentzer, G. : What algebraic graph transformations can do for model transformations. ECEASST **30** (2010)
27. Schürr, A., Klar, F. : 15 years of triple graph grammars. In Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G., eds. : ICGT. Volume 5214 of Lecture Notes in Computer Science., Springer (2008) 411–425
28. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A. : Moflon : A standard-compliant metamodeling framework with graph transformations. In Rensink, A., Warmer, J., eds. : ECMDA-FA. Volume 4066 of Lecture Notes in Computer Science., Springer (2006) 361–375
29. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G. : Henshin : Advanced concepts and tools for in-place emf model transformations. In Petriu, D.C., Rouquette, N., Haugen, Ø., eds. : MoDELS (1). Volume 6394 of Lecture Notes in Computer Science., Springer (2010) 121–135

30. Bach, J.C., Crégut, X., Moreau, P.E., Pantel, M. : Model transformations with Tom. In : LDTA, Tallinn, Estonia, ACM (2012) 16 To appear.

A Syntaxe des nouvelles constructions Tom dédiées aux transformations de modèles

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

Listing 8: Syntaxe de la construction %transformation

Afin d'être lisible, plusieurs éléments ne sont pas explicitement définis ici. `TransformationName`, `FileName`, `SubjectName`, `AlgebraicType` et `DefinitionName` sont des identifiants (`Identifier`) dont les noms sont suffisamment explicites pour comprendre leur rôle. `Strategy` représente une stratégie Tom (par exemple `‘TopDown(MyStrategy())`). `Pattern` est un motif tel que décrit dans la grammaire Tom du site officiel [18]. `BlockList` est un bloc constitué à la fois de code hôte et de code Tom, comme décrit dans la grammaire Tom. Nous invitons le lecteur à se référer au guide de référence du langage pour comprendre l'intégration des nouvelles constructions Tom dans la langage.

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

Listing 9: Syntaxe de la construction %tracelink

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

Listing 10: Syntaxe de la construction %resolve

Dans ces deux blocs de syntaxe, `VarName` et `TypeName` sont des identifiant (`Identifier`) représentant des noms de variables et de types.

18. http ://tom.loria.fr/wiki/index.php5/Documentation :Tom (15 janvier 2013)