Un framework de traçabilité pour des transformations à caractère impératif

Bastien Amar*, Jean-Rémy Falleri**
Marianne Huchard**, Clémentine Nebut**, Hervé Leblanc*

*IRIT, Université Paul Sabatier, 118 Route de Narbonne, F-31062 Toulouse Cedex 9
{amar, leblanc}@irit.fr,
**LIRMM, CNRS et Univ. Montpellier II, 161 rue Ada 34392 Montpellier Cedex 5 - France
{falleri, huchard, nebut}@lirmm.fr

Résumé. Cet article s'inscrit dans le cadre de l'ingénierie dirigée par les modèles et apporte une contribution au problème de la traçabilité des artefacts de modélisation durant une chaîne de transformations écrites dans un langage impératif. L'approche que nous proposons nécessite peu d'interventions de l'utilisateur. Nous introduisons un métamodèle générique des traces qui permet entre autres d'apporter une dimension multi-échelles aux traces grâce à l'application du patron de conception composite. Le principe de notre approche est de surveiller certaines catégories d'opérations intéressantes pour la génération de traces pertinentes. Ces catégories sont définies à l'aide du type des objets manipulés par les opérations. Une fois les catégories définies, la trace est générée par du code dédié qui est injecté automatiquement dans la transformation, autour des opérations caractérisées par les catégories définies. Un prototype a été réalisé pour les transformations de modèles écrites en Java, sur le *framework* EMF. L'injection du code dédié à la traçabilité est réalisée à l'aide de la programmation par aspects.

1 Introduction

Cet article s'inscrit dans le cadre de l'ingénierie dirigée par les modèles (IDM) (Bézivin et al., 2004), paradigme dans lequel la notion de métamodèle est placée au centre du cycle de développement. Elle permet de considérer des modèles comme des données et donc de les utiliser comme entités de première classe dans les langages dédiés à la transformation de modèles. La traçabilité des artefacts de modélisation durant une chaîne de transformations de modèles constitue la problématique abordée dans cet article.

Avec l'apparition des langages et outils dédiés à la transformation de modèles (e.g. ATL[1], Kermeta[2], EMF[3]), ainsi que des méta-métamodèles de référence (MOF, Ecore) les processus

[1] Atlas Transformation Language, http ://www.eclipse.org/m2m/atl/.
[2] http ://www.kermeta.org.
[3] http ://www.eclipse.org/modeling/emf/.

de développement dirigés par les modèles prennent corps. De ce fait, les problématiques récurrentes liées à la production logicielle apparaissent dans ce nouveau contexte de développement. Une de ces problématiques concerne la traçabilité, qui entre en jeu notamment dans le cadre de l'ingénierie des exigences : il faut en effet s'assurer que les exigences d'un projet se retrouvent dans les modèles, le logiciel et les cas de tests. Un groupe de travail de la communauté internationale IDM (ECMDA : European Conference on Model Driven Architecture) se consacre depuis plusieurs années à l'étude de la traçabilité. Celle-ci peut-être traitée principalement selon deux points de vue :

- la gestion des liens de traçabilité par interception d'événements dans un outil de développement dirigé par les modèles, voire un modeleur UML ;
- la création des liens de traçabilité lors d'une transformation automatisée de modèles pour laquelle le langage est soit impératif, soit déclaratif, soit mixte.

C'est cette dernière approche que nous avons choisi d'étudier et nous proposons dans cet article un *framework* dédié à la traçabilité pour les langages impératifs de transformation de modèle.

Un *framework* de traçabilité doit identifier les modèles sources et cibles lors d'une transformation de modèles, ainsi que le contexte de leurs transformations. Ainsi, elle permettra de (Limòn et Garbajosa, 2005) :

- maintenir et justifier une opération de transformation ;
- obtenir l'évolution des modèles au cours du processus de développement ;
- faciliter la maintenance et le contrôle d'un projet ;
- présenter les décisions prises lors du développement d'un projet.

La contribution de cet article est un mécanisme de gestion de traces emboîtées et générées de manière semi-automatique pour des langages de transformation impératifs. Pour illustrer notre contribution, nous allons considérer tout au long de cet article une transformation volontairement simple[4] illustrée figure 1. Elle réalise deux opérations de restructuration :

- la première est illustrée par les classes A et B de l'exemple. Deux attributs de même nom se retrouvent, après la transformation, factorisés dans une superclasse[5] commune qui sera créée si besoin est. Cette opération correspond au *refactoring* nommé *Extract Super Class* dans Fowler et al. (1999).
- la seconde est illustrée par les classes C de notre transformation exemple. Lorsque deux classes ont le même nom, elles sont fusionnées et leurs attributs respectifs se retrouvent dans le résultat de la fusion. On peut aussi voir cette opération comme la composition du *refactoring* nommé *movefield* (Fowler et al., 1999) ($attrC1$ déplacé dans la classe possédant $attrC2$ ou vice-versa) avec une opération de renommage de classe suivie d'une opération de nettoyage de classe vide.

Le résultat de notre approche sur cet exemple est un modèle représentant la trace dont une vue est présentée figure 2. Nous allons voir dans cet article la conception du *framework* permettant de l'obtenir. En premier lieu, nous proposons un métamodèle de traces emboîtées utilisant le patron de conception *composite*. Il permet d'apporter un comportement multi-échelles aux traces. Nous nous focaliserons ensuite sur la manière d'instancier ce métamodèle lors de l'exécution d'une transformation implémentée dans un langage impératif. Nous décrirons la mise

[4] L'intégralité du code de cette transformation est disponible sur
http ://www.lirmm.fr/∼nebut/Publications/ArticleSupplements/LMO2008/Traceability/ .

[5] Le nom de cette superclasse, dans l'exemple considéré (figure 1), provient d'une interprétation intensionnelle de l'héritage.

en œuvre de ce *framework* et notamment l'utilisation d'AspectJ. Enfin, une étude des travaux existants et une conclusion viendront terminer cet article.

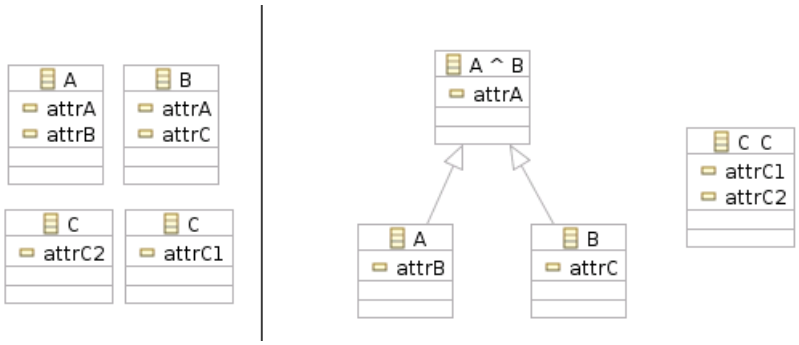

FIG. 1 – *Modèle source (à gauche) et cible (à droite) de la transformation exemple.*

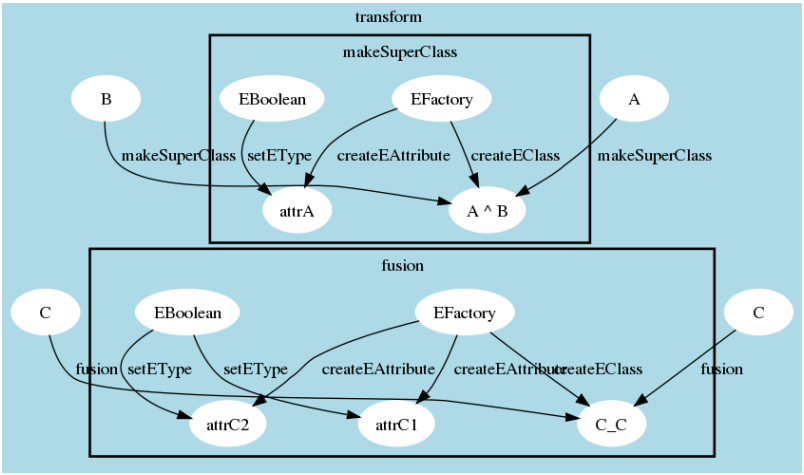

FIG. 2 – *Vue de la trace obtenue par notre* framework *de traçabilité. Les artefacts du modèle sont représentés par les nœuds. Les arcs annotés représentent certaines des opérations utilisées au cours de la transformation.*

2 Un métamodèle de traces emboîtées

Un *framework* de traçabilité permet de garder des informations sur le devenir des éléments des modèles au cours des différentes transformations qu'ils subissent. Dans un contexte d'ingénierie dirigée par les modèles, il est normal que l'information relative à la traçabilité soit

considérée comme un modèle. Un modèle est donc associé à chaque exécution d'une transformation tracée. La définition d'un métamodèle des traces nous permet de structurer les traces qui seront générées par le *framework* de traçabilité, et ainsi de mieux les manipuler.

Plusieurs métamodèles généralistes de traces ont déjà été proposés (Jouault, 2005; Falleri et al., 2006; Kolovos et al., 2006). Ils permettent de supporter la plupart des transformations envisageables en IDM (Mens et Gorp, 2006). Nous étudierons plus précisément ces différentes approches dans la section 5. Le métamodèle proposé ici (Figure 3) se base sur celui de Falleri et al. (2006) (qui gère les transformations en chaîne – plusieurs transformations exécutées successivement sur un même modèle – qui se produisent fréquemment en IDM), que nous avons étendu en y ajoutant les notions de types de liens (LINKTYPE) et de liens composites (COMPOSITELINK). Une trace (ETRACE) est une sorte de COMPOSITELINK, c'est-à-dire composée de plusieurs liens (ABSTRACTLINK) référençant deux objets appartenant aux modèles manipulés (SOURCE et DESTINATION).

La notion de types de liens de traçabilité a été introduite par Limòn et Garbajosa (2005). L'ajout des « types de liens » à ce métamodèle permet de garder l'intention d'une transformation dans le lien, et permet à l'utilisateur de raisonner sur les traces créées.

Il est utile, pour les transformations impératives aussi bien que déclaratives, d'avoir une représentation multi-échelle des traces. En effet, le fait qu'une opération puisse en appeler une autre (ou que des règles puissent en déclencher d'autres) crée des niveaux d'imbrication qu'il est utile de pouvoir représenter. C'est la raison pour laquelle le patron de conception composite (Gamma et al., 1995) est appliqué sur les liens. De cette manière, il est possible de manipuler des « paquets » de liens et l'utilisateur choisit le niveau de détail des traces qu'il visualise. De plus, l'application de ce patron permet d'encapsuler la complexité des transformations mises en jeu. La figure 4 nous montre une instanciation du métamodèle pour l'exemple de transformation que nous considérons. Pour plus de clarté, seule la partie concernant les classes A et B de notre transformation montrée sur la figure 1 a été gardée. Ce métamodèle est indépendant du type de langage de transformation utilisé et nous allons voir comment il est possible de l'instancier pour générer des traces dans le cas de transformations programmées dans un langage impératif.

3 Traçabilité des langages de transformation impératifs

Dans cet article, nous nous intéressons au problème de la gestion de la traçabilité pour les langages impératifs de transformation de modèles. Dans cette section, nous étudions les moyens d'obtenir un modèle de trace défini par le métamodèle proposé à la section précédente, au cours de l'exécution de transformations de modèles. Pour obtenir un modèle de trace, on peut envisager deux types de solution : une solution manuelle où l'utilisateur ajoute dans son code de transformation des instructions pour remplir le modèle de trace (c'est la solution adoptée dans Falleri et al. (2006)) ou une solution automatique où l'on génère la trace automatiquement par analyse du code de la transformation. La solution purement manuelle ne nous paraît pas adaptée : elle nécessite d'une part une trop grande intervention du programmeur et d'autre part une trop grande intrusion dans le code de la transformation. Nous pensons également qu'il est illusoire de générer de manière complètement automatique l'intégralité du modèle de trace. C'est pourquoi nous proposons une approche semi-automatique. Nous proposons de catégoriser les opérations d'une transformation et d'associer à chaque catégorie une

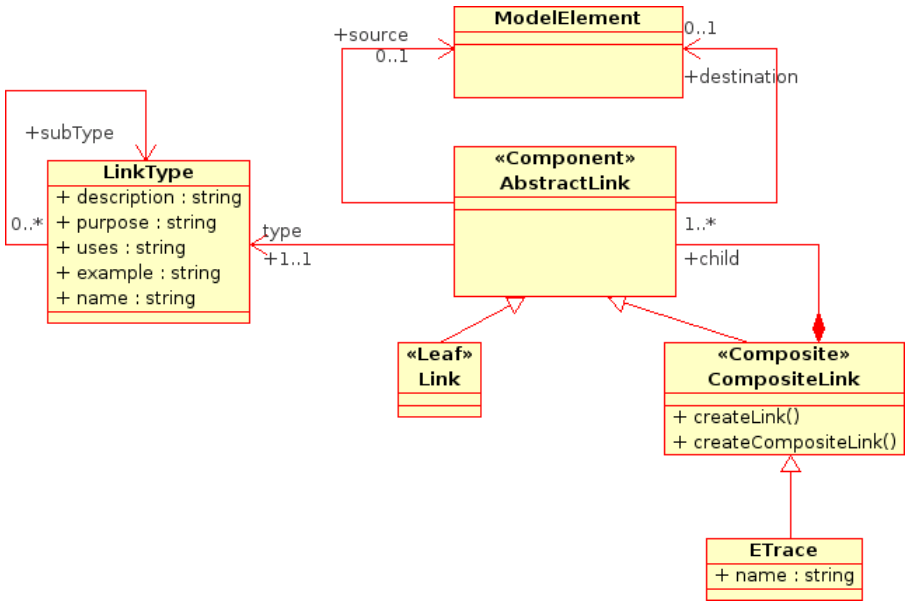

FIG. 3 – *Le métamodèle des traces.*

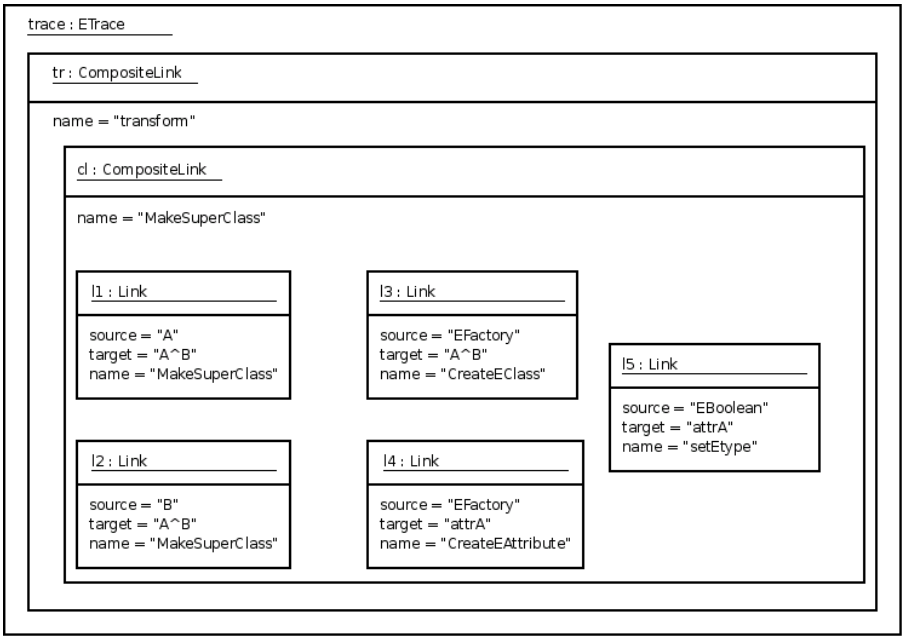

FIG. 4 – *Une partie de l'instanciation du métamodèle correspondant à la trace de la transformation décrite Fig. 1.*

manière de les tracer. Persuadés que cette approche ne permettra pas de capturer toutes les informations de traçabilité, nous laissons également la possibilité au programmeur de décrire ses propres types d'opérations traçables.

3.1 Catégories d'opérations traçables

Dans le reste de l'article, nous nous plaçons dans le cadre d'EMF (Budinsky et al., 2003) et des transformations de modèles réalisées en Java, mais ces travaux peuvent être appliqués pour n'importe quel langage impératif tel que, par exemple, Kermeta (Fleurey et al., 2006).

EMF est un *framework* de modélisation facilitant la génération de code pour le développement d'outils basés sur un modèle structuré. Il permet le chargement de modèles sérialisés en *Ecore* et leur transformation *via* l'API fournie par EMF, en Java. EMF est donc, entre autres, une plate-forme de transformation de modèles.

Dans un cadre IDM, les modèles manipulés par les transformations sont tous définis par un métamodèle. La racine des classes dans *Ecore* est EOBJECT[6], comme l'est OBJECT pour Java. Ainsi, tous les objets d'un modèle conforme à *Ecore* sont de type EOBJECT. On en déduit que les opérations intéressantes à tracer lors d'une transformation de modèles manipulent des EOBJECT. En effet, les transformations étant programmées en Java, il faut pouvoir distinguer les opérations du programme effectuant une transformation des autres opérations éventuellement présentes dans un programme Java classique.

Pour implémenter la transformation exemple présentée en introduction, nous définissons en Java, entre autres, l'opération de transformation :

$$EClass \quad makeSuperClass(EClass\, c1,\ EClass\, c2)$$

Elle prend en paramètre deux ECLASS, les fusionne et retourne une superclasse factorisant les attributs communs à $c1$ et $c2$. Tracer cette opération consiste à établir un lien entre l'objet retourné (la destination) et les objets $c1$ et $c2$ (les sources). Nous proposons l'heuristique suivante : chaque fois qu'une opération a pour paramètre au moins un EOBJECT et retourne un EOBJECT, les paramètres constituent les sources d'un lien de traçabilité, et l'objet retourné (une ECLASS dans notre exemple[7]) la destination du lien de traçabilité. Cela constitue une première catégorie d'opérations, et nous avons donc défini une manière de tracer ce type d'opération, en indiquant ce qui est source et ce qui est destination. Dans la suite de cet article, on note $A \preceq: B$ si B est spécialisé par A (*i.e.* A hérite de B) ou si $A = B$. Si A ne spécialise pas B et $A \neq B$, on note : $A \not\preceq: B$. Notre première catégorie d'opérations peut être exprimée comme suit :

$$meth(T_1, T_2, ..., T_n) : T_{retour} \mid T_{retour} \preceq: EObject \wedge (\exists T_i, T_i \preceq: EObject) \qquad (1)$$

Catégoriser le plus d'opérations possibles permet d'automatiser au maximum la création de la trace, et donc de minimiser l'intervention de l'utilisateur. Ci-dessous, nous présentons deux autres catégories d'opérations, associées à la manière de les tracer :

– deuxième catégorie : dans EMF, les créations d'objets se font *via* le patron de conception *factory* (Gamma et al., 1995). Tout objet est créé par l'appel de méthodes particulières

[6] Toutes les classes EMF ont un nom préfixé par *E* : EClass, EPackage, etc.

[7] Une ECLASS est de type EOBJECT car elle est l'une de ses sous-classes.

sur la classe singleton EFACTORY. L'objet nouvellement créé doit être lié dans une trace au singleton EFACTORY. Cela constitue une autre catégorie d'opérations. Elle nous permet d'identifier les objets nouvellement créés lors de la transformation. Dans la notation $C.meth(..)$, C représente la classe de l'objet sur lequel est appelée l'opération :

$$C.meth() : T_{retour} \mid T_{retour} \preceq: EObject \ \wedge C \preceq: EFactory \tag{2}$$

– troisième catégorie : les classiques accesseurs en écriture. Ceux-ci sont facilement détectables car ils n'ont pas de type de retour (*void*) ou leur type de retour n'est pas une sous-classe de EObject. Le receveur de l'envoi de message est un EOBJECT et ils n'ont qu'un seul paramètre, de type EOBJECT. Pour cette catégorie d'opération, l'objet passé en paramètre constitue la source de la trace, et l'objet receveur la destination.

$$C.meth(T) : T_{retour} \mid T_{retour} \not\preceq: EObject \ \wedge \ C \preceq: EObject \ \wedge \ T \preceq: EObject \tag{3}$$

On remarque que le type de l'élement retourné ne doit pas être EOBJECT pour ne pas que cette catégorie interfère avec la première catégorie que nous avons définie.

Ces catégories sont des heuristiques permettant de faciliter la génération automatique de la trace. Comme il a déjà été dit, il est illusoire, dans le cadre d'un langage impératif, de vouloir tout tracer automatiquement et la liste de catégories n'est pas exhaustive. Elle est liée au langage de transformation de modèle utilisé : par exemple, le patron de conception *factory* n'est pas commun à tous les langages et notre deuxième catégorie est donc une catégorie non transposable à tous les langages. C'est la raison pour laquelle il faut prévoir l'intervention de l'utilisateur, et lui donner les moyens de définir ses propres catégories pour adapter la génération de traces à ses besoins.

3.2 Opérations définies par l'utilisateur

Pour des langages impératifs, on ne peut pas proposer de mécanisme générique et entièrement automatique pour générer la trace. Nous avons donc laissé à l'utilisateur la possibilité d'ajouter ses propres catégories d'opérations à tracer, afin qu'il puisse tracer des opérations non prises en compte dans les catégories que nous avons définies ou pour qu'il puisse restreindre les catégories.

Pour les trois catégories d'opérations définies dans la section précédente, trois critères rentrent en jeu :

– la classe de l'objet receveur du message ;
– le type de retour de l'opération ;
– le type des paramètres de la méthode.

Néanmoins, pour que l'utilisateur puisse définir correctement une catégorie, d'autres critères sont à prendre en considération :

– le nom de la méthode ;
– l'artefact qui sera source dans la trace ;
– l'artefact qui sera cible.

Pour définir une catégorie personnalisée d'opération, l'utilisateur en spécifie la structure en instanciant un extrait du métamodèle *Ecore*, donné figure 5. Considérons l'opération suivante :

$$void \quad C.addAttribute(EClass, EAttribute)$$

Elle n'est pas prise en compte par les catégories que nous avons déjà définies. L'instanciation du métamodèle proposé permettra de créer une nouvelle catégorie restreinte à une opération dont le nom est *AddAttribute*, qui possède deux paramètres : l'un typé ECLASS, l'autre EATTRIBUTE. Ce type d'opérations est très difficile à tracer automatiquement, sans intervention de l'utilisateur.

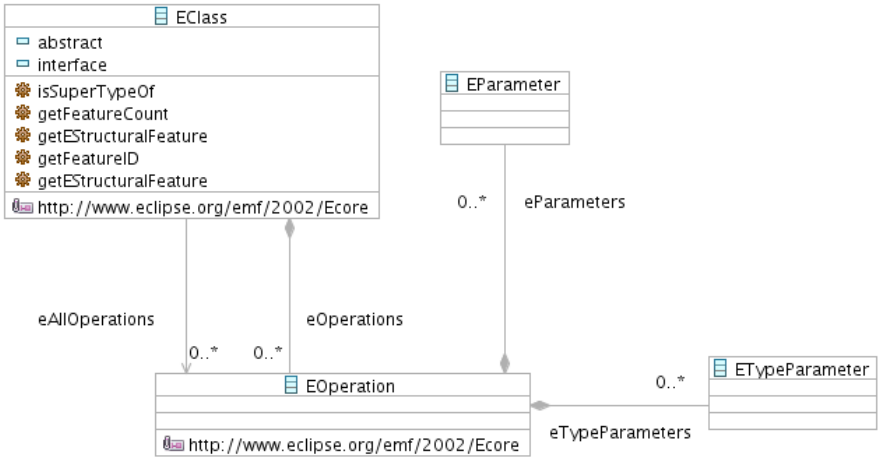

FIG. 5 – *Extrait de la partie « EOperation » d'*Ecore.

Ces instances du métamodèle de *Ecore* nous permettent donc de définir syntaxiquement de nouvelles catégories d'opérations. On peut considérer les catégories déjà définies comme des instances de ce métamodèle pré-enregistrées pour faciliter la démarche de l'utilisateur.

4 Génération des traces au cours d'une transformation de modèles

Nous avons vu dans les sections précédentes comment définir des catégories d'opérations intéressantes à tracer lors d'une transformation de modèles écrite dans un langage impératif. Cette partie explique comment ces catégories ont été utilisées afin d'instancier une trace durant l'exécution de la transformation.

4.1 Un point d'entrée : l'opération *transform*

Nous obligeons l'utilisateur à avoir un point d'entrée pour le début de la génération de la trace : une opération nommée *transform*. Considérons maintenant la transformation comme un arbre, représentant la décomposition fonctionnelle d'une transformation avec une racine imposée qui est cette opération *transform*. Une représentation d'un tel arbre est donnée par la figure 6. Elle représente la transformation que nous avons prise comme exemple. Chaque opération traçable appelée par *transform* est ensuite considérée comme un sommet fils direct

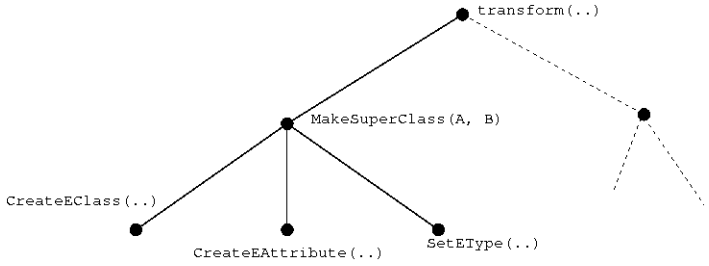

FIG. 6 – *Décomposition fonctionnelle d'une partie de la transformation exemple.*

de la racine de l'arbre. Si elle appelle des opérations traçables, le sommet la représentant aura lui-même des fils, et ainsi de suite. Les feuilles de l'arbre représentent donc le dernier niveau de granularité fonctionnelle voulu.

4.2 Organisation des traces

Nous avons vu que le patron de conception *Composite* a été appliqué sur le métamodèle des traces que nous avons défini. Nous allons voir dans cette section comment les traces sont générées, en considérant l'arbre de décomposition présenté sur la figure 6.

La sémantique que nous attribuons au patron *Composite* dans notre cas est d'avoir des traces emboîtées reflétant l'appel des fonctions au cours de la transformation. Le choix effectué pour obtenir une trace exploitable par l'utilisateur est le suivant : l'appel de l'opération *transform* provoque la création d'un *CompositeLink*. Les feuilles de l'arbre de décomposition fonctionnelle sont des feuilles du « Composite ». Les nœuds internes sont des composants composites du « Composite ». On obtient ainsi une instance du métamodèle isomorphe à l'arbre de décomposition fonctionnelle. L'utilisateur pourra ainsi choisir, lors de la visualisation de la trace, le degré voulu de granularité.

4.3 Intercepter les opérations traçables

Nous étudions maintenant le moyen d'intercepter les opérations correspondant à une catégorie afin de générer la trace. Nous avons choisi d'utiliser pour cela la programmation par aspects (Kiczales et al., 1997), et plus particulièrement AspectJ[8]. Cette approche permet de construire la trace sans modification du code de la transformation. La programmation par aspects permet de regrouper dans un module (un *aspect*) les préoccupations transverses à une application dont le code est habituellement disséminé dans plusieurs classes, comme, par exemple, la persistance. Un *aspect* est principalement défini de :

- *pointcut* : ils servent à définir les endroits où du code source va être greffé dans le code de l'application originale. Ils permettent éventuellement de définir quand ces greffes doivent être appliquées.
- *advice* : ce sont des fragments de code source destinés à être greffés sur l'application. Ils sont de trois types :

[8] http ://www.eclipse.org/aspectj .

- les *before* sont greffés avant le code source indiqué par le *pointcut* auquel ils sont associés ;
- les *after* sont greffés après ;
- les *around* sont greffés autour de l'opération originale. Le mot-clé *proceed* indique l'exécution de la méthode lors de la définition du *around*.

Dans notre cas, pour chaque catégorie d'opérations, un pointcut est créé. Une méthode de type *around* est exécutée avant et après l'appel de l'opération catégorisée et instancie le métamodèles des traces de façon adéquate. La table 1 présente les catégories que nous avons définies, associées à leur *pointcut*.

$meth(T_1, T_2, ..., T_n) : T_{retour}$ | $T_{retour} \preceq$: $EObject \wedge (\exists T_i, T_i \preceq$: $EObject)$	**pointcut** categorie1(): **call**(EObject+ * (.., EObject+,..))
$C.meth() : T_{retour}$ | $T_{retour} \preceq$: $EObject \wedge C \preceq$: $EFactory$	pointcut categorie2(): **call**(EObject+ * (..)) && **target**(receveur) && **if**(receveur **instanceof** EFactory)
$C.meth(T) : T_{retour}$ | $T_{retour} \not\preceq$: $EObject \wedge C \preceq$: $EObject \wedge T \preceq$: $EObject$	**pointcut** categorie3(): **call**(!EObject+ * (.., EObject+,..))

TAB. 1 – *Les catégories et leur* pointcut *associé.*

Par exemple, pour la première catégorie, il faut créer un *pointcut* qui intercepte les opérations dont le type de retour est un EOBJECT et dont au moins l'un des paramètres est un EOBJECT.

Le mot clé *call* indique que le pointcut fait référence à un appel de méthode. En paramètre du *call*, les informations sont organisées comme il suit :

- le type de retour, EOBJECT+, signifie que le type de retour doit être un EOBJECT ou l'une de ses sous-classes (+) ;
- le nom de l'opération est remplacé par * ce qui permet d'intercepter toutes les méthodes, sans tenir compte de leur nom ni de la classe des objets receveurs ;
- les paramètres de la méthode sont représentés par la liste (..,EOBJECT+,..) : peu importe le nombre ou le type des paramètres (« ..»), il doit juste y avoir un EOBJECT ou l'une de ses sous-classes (+).

D'autres mécanismes servant à définir des *pointcuts* sont utilisés, notamment, pour la deuxième catégorie :

- l'objet passé en paramètre de *target* est celui sur lequel sera appelée la méthode ;
- *if (receveur instanceof EFactory)* signifie que l'objet *receveur* doit être de type *EFactory*

Un *advice around* est associé à ces *pointcuts*. Le code exécuté autour de l'opération catégorisée génère la trace. Une partie de l'algorithme de génération implémenté dans l'*advice* est donnée par la figure 7.

Les catégories d'opération que nous avons définies ont chacune un *pointcut* qui leur est associé dans l'aspect. Nous avons vu que notre approche est semi-automatique. Les catégories

Données : lcc, le CompositeLink Courant
CompositeLink $sauv$ = lcc;
lcc = **new** CompositeLink();
Object $retour$ = **proceed**();
si *le lien composite courant a été rempli* **alors**
  Ajouter lcc au CompositeLink $sauv$;
pour *Tout argument* a *de la méthode* **faire**
  CompositeLink e;
  $e.source$ = a;
  $e.destination$ = $retour$;
  Ajouter e au CompositeLink $sauv$

FIG. 7 – *Algorithme de génération de traces*

définies par l'utilisateur doivent aussi avoir leur propre *pointcut* associé. En l'état actuel, l'utilisateur doit lui-même étendre l'aspect pour prendre en charge les opérations qu'il veut tracer. Néanmoins, grâce à l'instanciation du métamodèle *Ecore* nous possédons la syntaxe de ces catégories et nous pourrons donc générer les *pointcuts* adéquats pour les catégories nouvellement créées.

La vue du modèle sous forme de graphe annoté proposée en introduction est obtenue à l'aide d'une transformation *model-to-text*. En effet, la trace est projetée vers un format de représentation graphique de graphe nommé *dot*. Il permet une première visualisation rapide des traces de la transformation.

5 Travaux connexes

Jouault (2005) présente un métamodèle contenant l'essence de la notion de trace : une trace (TRACELINK) a comme source et destination un ou plusieurs ANYMODELELEMENT, que nous avons appelés MODELELEMENT dans le métamodèle que nous proposons. Il est adapté aux langages de transformations de modèles déclaratifs, tels que ATL, pour lequel il a été conçu. La génération de traces se fait grâce à l'application de règles concernant la traçabilité sur les règles de la transformation de modèle. Cette transformation du langage peut être vue comme une précompilation, et le code original de la transformation ne s'en trouve pas modifié. Néanmoins, cette approche n'est que peu applicable aux langages de transformations impératifs.

Dans Falleri et al. (2006), une trace est définie comme un ensemble ordonné de graphes bipartis avec une intersection commune. Le métamodèle proposé inclut à quelques différences près celui proposé par Jouault (2005). En effet, dans Falleri et al. (2006), une trace a une unique source, et une unique destination. En ce qui concerne la génération des traces, le programmeur doit lui-même entrer le code concernant la traçabilité et aucune automatisation n'est proposée. L'approche a été implémentée en Kermeta, et les perspectives indiquent une future gestion de marqueurs dans le code pour pouvoir automatiser la génération de traces. Notre approche est différente : nous essayons d'éviter la présence des informations concernant la traçabilité dans

la transformation originale. De plus, notre métamodèle est plus abouti et permet de créer des liens composites.

Pons et Kutsche (2004) proposent d'utiliser l'artefact *Abstraction* et son méta-attribut *mapping* d'UML pour maintenir la traçabilité. Cette approche permet de gérer les transformations endogènes concernant les modèles UML. Néanmoins, la norme UML est utilisée, et ces artefacts peuvent être utilisés pour d'autres usages, ce qui constitue un risque de mélange des informations.

D'autres métamodèles des traces ont été proposés dans la littérature (Kolovos et al., 2006; Oldevik et Neple, 2006; Bondé et al., 2005) mais sont destinés à des transformations plus spécifiques. Nous avons essayé, dans notre cas, d'établir un métamodèle assez générique permettant de gérer au mieux la plupart des transformations envisageables (Mens et Gorp, 2006) en IDM.

D'une façon plus macroscopique, dans Diaz et al. (2007), les auteurs proposent un métamodèle général de traçabilité s'instanciant en modèles permettant d'assurer la traçabilité des fonctionnalités ou des préoccupations transversales ainsi que la traçabilité des transformations. L'instanciation de ces modèles résulte en graphes de trace. Le graphe de trace d'une fonctionnalité (ou d'une préoccupation transversale) montre quelles entités logicielles sont impliquées dans sa mise en œuvre et de quelle manière (dépendance, composition, documentation, etc.). Les graphes de trace des transformations exposent à une granularité élevée les unités logicielles et les artefacts (par exemple les classes) et les relations de dépendance, d'évolution ou de documentation.

6 Conclusions et perspectives

Au cours d'un développement guidé par les modèles, les artefacts de base sont les modèles et les transformations de modèles permettant par exemple de refactoriser, spécialiser pour une plate-forme donnée, ou encore fusionner des modèles. Le processus de développement est alors largement basé sur l'application des transformations aux modèles. Il est donc nécessaire de garder une trace des différentes transformations opérant sur les modèles d'un projet.

Dans cette optique, nous proposons dans cet article un *framework* de traçabilité dédié aux langages de transformation impératifs et permettant de générer un modèle de trace des différentes modifications des éléments de modèle au cours de l'exécution d'une série de transformations de modèles. Notre *framework* se base sur un métamodèle de trace permettant de définir des traces imbriquées : la modification d'un élément peut ainsi se décomposer en plusieurs sous-modifications plus élémentaires et l'analyse de la trace se trouve ainsi simplifiée, puisqu'on peut la regarder à la granularité choisie. Nous avons de plus proposé la définition de catégories syntaxiques d'opérations à tracer et fourni le moyen de générer la trace à partir de ces catégories et d'une exécution donnée, grâce à l'utilisation de la programmation par aspects. Nous avons proposé de telles catégories pour des transformations écrites en Java/EMF, ainsi que le mécanisme permettant à un utilisateur de définir ses propres catégories. Notre *framework* permet d'obtenir une visualisation des liens de traçabilité à différentes granularités lors de l'exécution de transformations de modèles.

Une amélioration à apporter au *framework* de traçabilité est de lui permettre de gérer la suppression d'éléments au cours d'une transformation de modèle, en utilisant le fait que nos traces n'ont pas forcément de destination. Du côté de la manipulation des liens, il faudrait pou-

voir accéder aux éléments des modèles directement à partir de la trace. De plus, nous projetons de sauvegarder une partie de la sémantique et les conditions des opérations de transformation en gérant les types de liens définis dans notre métamodèle de traces. Une piste de poursuite de ces travaux concerne la définition de catégorie à tracer par l'utilisateur. Il faudrait générer automatiquement les *pointcut* correspondant aux catégories définies *via* le métamodèle des opérations. Enfin, il est prévu de gérer plus finement la définition de catégorie avec la possibilité d'ajouter de la sémantique dans la création de règles, par exemple avec l'utilisation d'un interpréteur OCL.

Références

Bézivin, P., M. Blay, M. Bouzhegoub, J. Estublier, J. Favre, S. Gérard, et J.-M. Jézéquel (2004). Rapport de Synthèse de l'AS CNRS sur le MDA (Model Driven Architecture).

Bondé, L., P. Boulet, et J.-L. Dekeyser (2005). Traceability and interoperability at different levels of abstraction in model transformations. In *Forum on Specification and Design Languages,FDL'05, Lausanne, Switerland*.

Budinsky, F., T. Grose, D. Steinberg, R. Ellersick, E. Merks, et S. Brodsky (2003). *Eclipse Modeling Framework : a developer's guide*. Addison-Wesley Professional.

Diaz, D., L. Seinturier, L. Duchien, et P. Flament (2007). Une aide à la réalisation des évolutions logicielles grâce aux modèles de traçabilité des fonctionnalités. *RSTI-L'Objet 13*(1), pp. 117–145.

Falleri, J.-R., M. Huchard, et C. Nebut (2006). Towards a traceability framework in Kermeta. In *ECMDA-TW 2006 Proceedings, Bilbao, July 11th 2006*, pp. 31 – 40.

Fleurey, F., Z. Drey, D. Vojtisek, et C. Faucher (2006). *Kermeta language Reference manual, Internet : http ://www.kermeta.org/docs/KerMeta-Manual.pdf.* IRISA.

Fowler, M., K. Beck, J. Brant, W. Opdyke, et D. Roberts (1999). *Refactoring : Improving the Design of Existing Code*. Addison-Wesley Professional.

Gamma, E., R. Helm, R. Johnson, et J. Vlissides (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Jouault, F. (2005). Loosely coupled traceability for ATL. In *ECMDA Workshop on traceability. November 8th 2005, Nuremberg Germany*, pp. 29–37.

Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, et J. Irwin (1997). Aspect-oriented programming. In M. Akşit et S. Matsuoka (Eds.), *Proceedings European Conference on Object-Oriented Programming*, Volume 1241, pp. 220–242. Berlin, Heidelberg, and New York : Springer-Verlag.

Kolovos, D. S., R. F. Paige, et F. A. Polack (2006). On-demand merging of traceability links with models. In *ECMDA-TW 2006 Proceedings, Bilbao, July 11th 2006*, pp. 7 – 15.

Limòn, A. E. et J. Garbajosa (2005). The need for a unifying traceability scheme. In *ECMDA Workshop on traceability. November 8th 2005, Nuremberg Germany*, pp. 47 – 56.

Mens, T. et P. V. Gorp (2006). A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci. 152*, pp. 125–142.

Oldevik, J. et T. Neple (2006). Traceability in model to text transformation. In *ECMDA-TW 2006 Proceedings, Bilbao, July 11th 2006*, pp. 67 – 72.

Pons, C. et R.-D. Kutsche (2004). Traceability across refinement steps in UML modeling. In *3rd Workshop in Software Model Engineering WiSME, 7th International Conference on the UML*.

Summary

This paper deals with Model-Driven Engineering and contributes to the issue of model artefact traceability for imperative model transformations. The approach we propose requires few interventions from the user. We introduce a generic trace metamodel that enables someone to define multi-scaled traces thanks to the use of the Composite design pattern. The principle of our approach is to monitor categories of interesting operations in order to generate relevant traces. Those categories are defined based on the type of the objects handled by the operations. Once the categories are defined, the trace is generated by dedicated code that is automatically injected in the transformation, around the operations characterized by the defined categories. A prototype has been implemented for model transformations written in Java, on the EMF framework. The injection of the code dedicated to traceability is implemented using aspect-oriented programming.