Evolution des profils UML : vers une migration automatisée et une optimisation assistée des modèles

Fadoi Lakhal[*], Hubert Dubois[*], Dominique Rieu[**]

** CEA, LIST*
Laboratoire d'Ingénierie dirigée par les modèles pour les Systèmes Embarqués
91191 Gif-sur-Yvette CEDEX, France
{fadoi.lakhal, hubert.dubois}@cea.fr

*** Laboratoire d'informatique de Grenoble, Equipe SIGMA*
220, Rue de la Chimie, BP 53
38041 Grenoble Cedex 9, France
dominique.rieu@imag.fr

RÉSUMÉ. La syntaxe abstraite d'un langage de modélisation évolue au fil des versions de ce langage et ces évolutions peuvent avoir des répercussions importantes sur les modèles décrits en ce langage. Nous étudions les évolutions de la syntaxe abstraite lorsque celle-ci est définie par un profil UML. Plus particulièrement, nous nous sommes intéressés à l'impact des évolutions d'un profil UML sur les modèles instances et à minimiser l'effort pour faire migrer ces modèles vers une version conforme à la nouvelle version d'un profil. En effet, le coût d'une migration manuelle des modèles peut parfois être supérieur au coût d'une redéfinition complète de ces derniers. La classification des évolutions d'un profil UML en fonction de leur impact sur les modèles instances semble être l'étape indispensable pour offrir une approche de migration automatisée qui soit adaptée à chaque type d'évolution. L'objectif étant que l'intervention manuelle du concepteur soit la plus réduite possible.

ABSTRACT. The abstract syntax of a modeling language evolves all along the life-cycle of this language with its successive versions and these evolutions can have important repercussions on the models described in this language. We study the evolutions of the abstract syntax when this syntax is defined by a UML profile. More particularly, we are interested in the UML profile evolutions impacts on the instance models and in minimizing the efforts to migrate models to a new version that has to be compliant with the new version of the profile. Indeed, the cost of a models manual migration can sometimes be higher to the cost of a complete redefining. Defining a UML profile evolutions classification according to their impacts on the instance models is the indispensable step to offer an automated migration adapted to each evolution. The objective is to reduce as much as possible the different interventions of the designer and to guide the evolution process.

MOTS-CLÉS : syntaxe abstraite, évolution de profil UML, classification d'évolution, classification d'impact, migration de modèle.

KEYWORDS: abstract syntax, modeling language, UML profile evolution, evolutions classification, impact classification, models migration.

1. Introduction

Dans le contexte de l'ingénierie dirigée par les modèles (IDM) (Estublier et al., 2005), deux solutions sont largement utilisées pour décrire la syntaxe abstraite d'un langage de modélisation : la définition d'un métamodèle (Kleppe, 2007) ou la définition d'un profil UML (Selic, 2011). La définition d'un métamodèle est la solution la plus adaptée lorsqu'il s'agit de définir des concepts d'un domaine particulier tout en garantissant une certaine indépendance envers d'autres langages de modélisation existants. Avec cette solution, pour chaque nouveau langage, le problème de la redéfinition de concepts de base se pose : doit-on redéfinir ce qu'est une classe ou un diagramme état-transition ? Pour répondre partiellement à cette problématique, le langage UML propose son propre mécanisme d'extension : les profils (OMG, 2012). Définir un profil UML pour décrire la syntaxe abstraite d'un langage dédié à un domaine spécifique consiste à étendre les concepts du langage UML (comme *Class*, *Operation*, *Association*, …) possédant leur propre sémantique et de les spécialiser en leurs associant un (des) stéréotype(s) (*Stereotype*) correspondant(s) à un (des) concept(s) propre(s) au domaine. Ce mécanisme de définition de profil offre un double avantage qui est d'une part l'économie de redéfinir des concepts de base permettant ainsi de se concentrer sur le cœur du langage directement ; et d'autre part un avantage pratique en utilisant les outils UML déjà disponibles et offrant la possibilité de définir et de manipuler des profils UML. Ces raisons justifient le nombre croissant de profils disponibles pour des domaines aussi variés que les systèmes complexes (SysML) (OMG, 2012) ou le temps réel embarqué (MARTE) (OMG, 2012). Rien que pour ces profils, force est de constater que ces langages évoluent très régulièrement et ce, pour diverses raisons : émergence d'un nouveau concept, évolution des concepts décrivant la syntaxe abstraite ou toute autre modification apportée au profil (structure, relations, etc.). De plus, du fait qu'il n'existe que peu de travaux (Lagarde, 2007; Selic, 2011) aidant à la bonne conception de profil, de nombreuses versions successives sont généralement produites avant d'obtenir un profil stable. La gestion de ces évolutions peut alors devenir problématique en fonction du type des modifications apportées au profil et en fonction de l'impact de ces modifications sur les modèles instances. Dans de nombreux cas, ces évolutions vont entraîner une perte de conformité des modèles vis-à-vis de la nouvelle version du profil. Pour permettre un usage encore plus large des profils, il est primordial d'avoir une meilleure gestion des impacts des évolutions de profils sur les modèles instances pour éviter des ré-définitions trop coûteuses de ces modèles. Nous nous sommes donc intéressés au support des évolutions des profils et essentiellement à leurs impacts sur les modèles instances. De plus, les concepteurs des modèles utilisent souvent des profils « standards » qui ne sont pas développés dans leurs propres équipes mais livrés sous forme de nouvelle version de référence du langage, ne participant pas ainsi à l'évolution du profil. L'impact des évolutions du profil est donc plus difficile à gérer par le concepteur de modèles qui ne possède pas l'historique des évolutions (motivations, versions intermédiaires du profil). Nous distinguons dans notre approche les deux

acteurs que sont l'utilisateur des profils (le concepteur des modèles) et le concepteur des profils, ces deux acteurs ne sont pas ainsi systématiquement une seule et même personne. Notre travail fournit un support d'évolution de profil pour les concepteurs de modèles ne disposant que des versions diffusées d'un profil sans participer aux étapes d'évolutions de ce profil.

L'objectif de cet article est de proposer une approche facilitant la co-évolution, (Mens, 2008) des modèles instances d'un profil suite à une évolution de ce dernier. L'enjeu est :

1/ d'adapter les modèles pour qu'ils restent conformes à leur profil, un modèle étant conforme à une syntaxe abstraite ; dans notre cas, celle-ci est décrite par un métamodèle puis implémentée comme profil UML. De ce fait, les modèles décrits en utilisant un profil sont conformes à la syntaxe abstraite que traduit ce profil (et sont donc conformes au profil).

2/ d'améliorer les modèles pour qu'ils représentent au mieux les systèmes modélisés. Une évolution de profil correspond le plus souvent à une amélioration du langage de représentation d'un domaine d'application ; amélioration dont les modèles doivent bénéficier.

Le processus de co-évolution que nous proposons repose sur deux opérations :

1/ une opération de **migration** qui permet d'adapter un modèle de manière à ce qu'il reste conforme à son profil. La migration d'un modèle doit être la plus automatisée possible et des alertes et des préconisations sont générées vers le concepteur des modèles.

2/ une opération d'**optimisation** qui permet d'améliorer un modèle de manière à ce qu'il soit le plus représentatif du système modélisé (ceci améliore l'acceptation du modèle par le concepteur). Cette opération ne peut pas être entièrement automatisée : des alertes et des préconisations sont alors émises vers le concepteur des modèles.

Cet article est organisé comme suit : après un positionnement par rapport aux approches existantes en section 2, nous proposons en section 3 une classification des évolutions en fonction du type des modifications faites sur un profil et en fonction de leur impact sur les modèles instances. En section 4, nous décrivons le processus de transformation que nous proposons qui est basé sur une migration la plus automatique possible. La section 5 présente l'approche expérimentale suivie pour catégoriser les différents types d'évolution. Nous concluons et ouvrons nos perspectives de travail en dernière section.

2. Positionnement scientifique

Les travaux menés sur les évolutions de la syntaxe abstraite d'un langage de modélisation ne sont abordés que pour le cas qu'une description de cette syntaxe sous la forme de métamodèle. Aucune de ces approches n'étant directement proposée pour gérer l'évolution d'une syntaxe abstraite basée sur des profils UML, nous avons regardé si elles étaient néanmoins adaptables à cet usage.

2.1 Processus défini pour la gestion de l'évolution d'un métamodèle

D'après les différents travaux étudiés sur ce thème, nous observons que la gestion de l'impact d'une évolution d'un métamodèle sur les modèles instances peut se résumer à un processus en trois étapes illustré en Figure 1.

1/ **Identifier les modifications entre deux versions d'un métamodèle** dans un modèle dédié appelé « *Delta model*». Deux méthodes peuvent être utilisées pour l'établir : la méthode de différenciation (*Diffing*) qui consiste à chercher les différences entre deux versions d'un métamodèle *a posteriori* (l'évolution n'est pas observée pas à pas mais statiquement entre deux versions du métamodèle) et la méthode d'enregistrement des changements (*Change Recorder*) qui consiste à enregistrer pendant la phase d'évolution du métamodèle toutes les opérations d'évolution effectuées (ajout, suppression, etc.).

2/ Exploiter ce *Delta model* pour **classifier les modifications** selon le type de l'opération d'évolution mais aussi selon leurs impacts sur les modèles instances.

3/ Exploiter cette classification afin de proposer des **stratégies de migration réalisant l'adaptation des modèles afin de préserver leur conformité** à la version de la syntaxe abstraite qui a évoluée. Les approches étudiées (paragraphe 2.2) sont focalisées sur l'adaptation des modèles. Elles ne prennent pas en compte leurs améliorations possibles afin d'obtenir des meilleures représentations des systèmes.

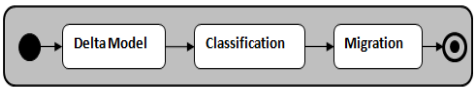

Figure 1: *Processus pour la gestion de l'évolution d'un métamodèle*

2.2 Approches existantes

Pour chacune des approches existantes, à l'issue de l'étape « *Delta model* », des questions se posent : quelle méthode est utilisée pour déterminer les différences entre deux versions d'un métamodèle ? Le *Delta model* est-il obtenu automatiquement ou spécifié manuellement ? Quel est l'acteur principal de cette étape ? Une étape de classification est-elle proposée par l'approche ? De la même manière, à l'issue de l'étape de migration, on se demande si l'opération de migration est obtenue manuellement ou automatiquement et aussi quel est l'acteur principal durant cette étape ? Afin de répondre à ces questions, nous avons établi cinq critères permettant d'évaluer les approches existantes. Ces critères sont :

- -la méthode de **détermination** des différences : *Diffing* ou *Change Recorder*.
- -la méthode d'**obtention** du *Delta model* résultat : obtention manuelle ou obtention automatique.
- -l'**acteur** principal de la co-évolution : concepteur de métamodèles ou concepteur de modèles.
- -la proposition ou non d'une **classification**.
- -la méthode d'**obtention** de **l'opération de migration** : obtention manuelle ou automatique.

Burger et al. (Burger et al., 2010) proposent un métamodèle de différence (métamodèle *Delta*) qui regroupe les éléments susceptibles d'évoluer et ayant un impact sur les modèles instances. Toute évolution se résume à la combinaison d'opérations de suppression ou d'addition. Le métamodèle de différence proposé ne traite pas les éléments pouvant évoluer dans l'ensemble de leurs caractéristiques (attributs dérivés, typage, etc.). Enfin, il n'est pas adapté à l'évolution d'un profil car les notions de stéréotype ou d'extension ne sont notamment pas traitées.

Dans (Cicchetti et al., 2008), les auteurs proposent un enchaînement de deux transformations : 1/ définition des différences entre deux versions d'un métamodèle ; 2/ migration des modèles instances. La première consiste à définir un métamodèle de différence (métamodèle *Delta*) qui est utilisé comme grammaire pour décrire des modèles de différence (*Delta model*). Le concepteur du métamodèle *Delta* (qui est également le concepteur des modèles) doit déterminer les différences (par un *Diff*) entre deux versions puis consigner ces différences manuellement dans un modèle *Delta*. La seconde transformation consiste alors à appliquer sur les modèles instances des règles de transformation implémentant les différences obtenues précédemment. Cette approche ne définit pas de classification particulière mais réutilise la classification proposée par Gruschko et al (Grushko et al., 2007).

Levendovszky et al. définissent (Levendovszky et al., 2010) un langage appelé *Model Change Language* (MCL) avec lequel le concepteur du métamodèle et des modèles peut définir des règles pour mettre en relation des concepts entre deux versions d'un même métamodèle (par ex., un renommage d'une métaclasse se traduit par une relation « MapsTo »; la règle correspondante spécifie que le nom est à modifier). La détermination des différences et la spécification du modèle *Delta* correspondant consistent à l'établissement de ces relations de *mapping*. L'algorithme de migration qui est défini manuellement sert d'entrée à un outil de migration de modèle qui les interprète et les exécute. Cette approche ne propose pas d'étape de classification. Cette approche ne correspond pas à nos objectifs car elle est difficilement applicable à des métamodèles important. En effet, il est très difficile de faire manuellement un mapping précis entre les concepts en évitant l'introduction d'erreurs dans la rédaction des règles de mapping. De plus, comme pour (Cicchetti et al., 2008), l'approche repose sur une spécification du modèle de différence faite par le concepteur des modèles qui n'est pas forcément apte à identifier les différences entre deux versions du métamodèle puis de les impactés sur les modèles instances.

Dans (Hermandosfer et al., 2008), durant l'évolution d'un métamodèle, l'outil enregistre chaque modification sur le métamodèle et attache à chacune d'elle une opération de migration (ajout d'une propriété, renommage d'un élément, etc.). L'ensemble de ces couples d'opérations est regroupé dans un modèle *Delta* qui est obtenu automatiquement. Puis, ces couples d'opérations sont exécutés séquentiellement sur les modèles instances à faire évoluer : le modèle *Delta* est donc

également utilisé en tant qu'algorithme de migration. De plus, ils proposent une classification des évolutions en quatre catégories. Une évolution est dite :

1. « spécifique à un modèle » si les modifications apportées ne font évoluer qu'un seul modèle instance.
2. « indépendante du modèle » si les modifications apportées impactent l'ensemble des modèles instances.
3. « spécifique à un métamodèle » si les changements apportés au métamodèle sont dus à un besoin spécifique du domaine que décrit le métamodèle.
4. « indépendante du métamodèle » si les changements apportés au métamodèle ne sont pas spécifiques au domaine décrit. Par exemple, si les langages de définition des métamodèles ont évolué, l'ensemble des métamodèles décrits dans ce langage doivent évoluer.

Enfin, cette approche n'est pas adaptée à nos objectifs car le modèle de différence ne peut être obtenu a posteriori.

Dans (Grushko et al., 2007) une approche est proposée pour migrer les modèles instances conformément à une évolution du métamodèle. Le modèle *Delta* est obtenu automatiquement (par *Diff* ou par enregistrement). Puis, ils établissent une classification des ajouts ou des suppressions d'éléments d'un métamodèle en fonction de leur impact sur les modèles instances mais aussi en fonction du niveau d'automatisation nécessaire pour faire migrer ces modèles. La classification se décompose en trois catégories en fonction du degré d'automatisation :

1. changements non cassants (*Non breaking changes*) : leur impact fait que les modèles instances restent des instances de la nouvelle version du métamodèle.
2. changements cassants mais pouvant être résolus (*breaking and resolvable*) : l'impact de ces changements fait que les modèles instances ne sont plus des instances du métamodèle évolué mais la migration de ces modèles vers une version conforme à la nouvelle version du métamodèle est complètement automatisable.
3. changements cassants et ne pouvant être résolus (*breaking and non resolvable*) : cette catégorie se différencie du cas précédent par le fait que la migration n'est pas possible automatiquement et qu'elle nécessite obligatoirement l'intervention du concepteur des modèles.

La stratégie de migration proposée dans (Grushko et al., 2007) consiste alors à exécuter une transformation spécifiée manuellement par le concepteur des modèles. Or, spécifier cette transformation peut prendre autant de temps que de redéfinir les modèles complètement.

Le Tableau 1 confronte les approches étudiées aux critères décrits précédemment. Le signe « = » indique que le concepteur du métamodèle et des modèles sont les mêmes personnes.

2.3 Synthèse et Positionnement

Rappelons que l'objectif est de maintenir la conformité des modèles instances entre deux versions successives d'un profil de la manière la plus automatisée possible. Les approches de Burger, Cicchetti et Levendovszky reposent sur une spécification du modèle de différence faite par le concepteur des modèles. Ce dernier doit également spécifier manuellement l'opération de migration. Or, cette spécification des règles de migration peut prendre autant de temps que de redéfinir l'ensemble des modèles. L'approche proposée par Hermandosfer ne correspond pas à nos objectifs du fait que les couples d'opérations de migration sont créés durant l'évolution des profils alors que nous voulons intervenir après celle-ci. Pour l'étape de classification, celle proposée par Hermandosfer détermine un indice de généricité d'une évolution qui ne nous intéresse pas ; alors que celle proposée par Grushko est la plus adaptée à notre approche du fait qu'elle s'applique sur un *Delta model* qui peut être obtenu *a posteriori*.
Par contre, aucune de ces approches n'aborde le cas spécifique de l'évolution des profils. Nous désirons également classer les évolutions d'un profil en fonction du niveau d'automatisation de l'adaptation des modèles instances (soit de l'opération de migration associée). Ceci, tout en conservant à l'issue de l'opération de migration un modèle le plus représentatif du système modélisé. Enfin, nous souhaitons également améliorer la représentativité des modèles en tirant partie des évolutions de profil : cette optimisation n'est pas prise en compte dans les approches existantes. Le Tableau 1 est complété avec notre approche pour illustrer notre positionnement.

CRITERE / APPROCHE	Delta Model							Classification	
	Determination		*Obtention*		*Acteur*				
	Diff	Enregistrement	Spécification manuelle	Génération automatique	MM concepteur	Modèle concepteur	Système	oui	non
Burger et al.	√				√				√
Ciccheti et al.	√		√		√	√			√
Levendovsky et al.	√		√		√	√			√
Hermandosfer et al.		√		√			√	√	
Gruschko et al.	√	√		√				√	
Lakhal et al.	√			√			√	√	

CRITERE / APPROCHE	Migration				
	Algorithme détemination		*Acteur*		
	Generation automatique	Spécification manuelle	MM concepteur	Modèle concepteur	Système
Burger et al.					
Ciccheti et al.		√	√	√	
Levendovsky et al.		√	√	√	
Hermandosfer et al.	√		√		
Gruschko et al.		√		√	
Lakhal et al.	√				√

Tableau 1: *Synthèse des approches et positionnement*

3. Classification des évolutions de profil UML

3.1 Eléments évolutifs pour la construction d'un profil UML

Un profil UML est constitué d'éléments qui étendent le métamodèle UML et qui permettent de spécialiser les notions de UML. En plus d'expliquer chaque étape de construction d'un profil, (Selic, 2011) explique qu'une extension du langage UML

implique une proximité de la sémantique entre le stéréotype créé et la métaclasse étendue. Il précise également qu'il ne doit pas exister de conflit entre les caractéristiques du stéréotype créé et celles héritées de la métaclasse étendue. Dans ces conditions, nous avons commencé par déterminer l'ensemble des éléments d'un profil pouvant évoluer et pouvant avoir un impact sur les modèles instances. Le sous-ensemble du métamodèle UML décrivant le mécanisme d'extension UML par profil est rappelé dans la Figure 2 (éléments grisés).

D'après la norme UML, l'élément *Stereotype* est « *un type limité de métaclasse* ». Il peut donc définir des propriétés, des opérations, des relations vers d'autres éléments du profil. Ce sont ces caractéristiques qui permettent de décrire un concept d'un domaine particulier. Ces caractéristiques n'ayant pas un nombre fixe ou une valeur par défaut, ce sont justement leurs évolutions qui feront que l'évolution d'un stéréotype impactera les modèles instances. Il est donc important d'étudier les évolutions d'un *Stereotype* mais aussi celles de ses caractéristiques.

L'élément *Extension* est une association permettant de relier par ses extrémités (*ExtensionEnd*) une métaclasse à un stéréotype. Il est utilisé pour attribuer à un stéréotype le concept adéquat du métamodèle UML (via la métaclasse UML correspondante). Le stéréotype hérite des caractéristiques de la métaclasse étendue mais aussi de son implémentation. Les évolutions peuvent venir du stéréotype ou des caractéristiques héritées.

L'élément *Profile* est une spécialisation de l'élément *Package*. Un *Profile* évolue car les éléments qu'il contient évoluent ou ces éléments intègrent le nom du profil pour construire leurs propre nom qualifié. Ainsi, un renommage du profil peut avoir des effets de bord sur l'utilisation des éléments contenus. Un profil peut contenir des sous profils ou paquetages, une hiérarchie est alors créée. Une modification de cette hiérarchie ou le déplacement d'un élément dans la hiérarchie peut avoir un impact sur les modèles instances. Un profil autorise l'import d'éléments (*ElementImport*) ou de paquetages (*PackageImport*) externes au profil. Il est donc possible de composer un profil en intégrant d'autres profils existants ou de réutiliser des concepts définis dans un autre langage de modélisation. Par ce lien d'import, un profil peut évoluer suite à l'évolution d'un élément importé.

La Figure 3 illustre un exemple d'évolution de profil. A la version N, le profil est composé d'un stéréotype *StereotypeA* qui étend une métaclasse *MetaClasseA*. Le lien d'extension (flèche à fond noir) signifie que le concept défini par *StereotypeA* se comporte comme *MetaclasseA* (proximité sémantique) mais avec une sémantique différente. *MetaClasseA* ne possédant pas de caractéristique (propriété, relation, contrainte, opération), *StereotypeA* ne possède donc pas de caractéristique héritée et n'en définit pas de nouvelle. L'évolution a consisté à ajouter une nouvelle propriété au *StereotypeA* et à définir un nouveau stéréotype (*StereotypeB*). Une relation entre *StereotypeA* et *StereotypeB* a également été ajoutée avec une multiplicité minimale de 0 (donc optionnelle) signifiant qu'à chaque instance de *StereotypeB* peut être associée à une instance de *StereotypeA*. La Figure 3 ne représente qu'un exemple simple d'évolution d'un profil ; cependant nous avons mené une expérimentation approfondie (section 5) pour établir un catalogue regroupant toutes les évolutions possibles entre deux versions d'un profil.

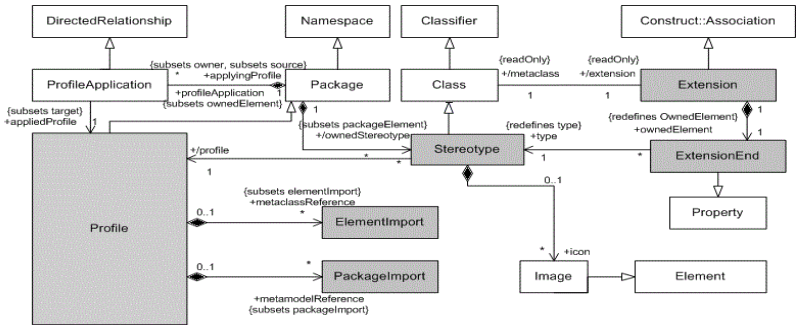

Figure 2: *Fragment UML: éléments de construction avec impact sur les modèles.*

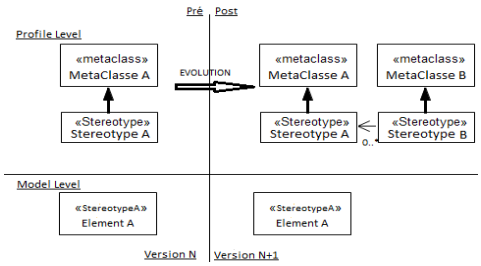

Figure 3 : *Exemple d'évolution de profil*

3.2 Classification des évolutions de profils

Les expérimentations menées sur des évolutions de profils de grandes tailles (section 5) ont mis en évidence deux aspects complémentaires de l'évolution : le niveau d'automatisation de l'opération de migration est à considérer ainsi que les opérations d'optimisation des modèles instances. En effet, la qualité de représentation du système est également liée au type de l'évolution. Nous déduisons de ces expérimentations quatre catégories d'évolutions possibles d'un profil.

CATEGORIE 1 : ***Impact-free category***. Cette catégorie correspond aux évolutions dont l'impact sur la conformité des modèles est nul. Les modèles n'ont pas à s'adapter à la nouvelle version du profil, ils restent des instances du profil évolué (aucune opération de migration n'est nécessaire).
Alors que Grushko considère ces évolutions comme étant *non-breaking*, nous considérons que ce type d'évolution est bien sans impact sur la conformité des modèles instance, mais qu'elles ont un impact sur la qualité de représentation des modèles. Certains ajouts de concepts ou de caractéristiques dans le profil, même s'ils ne nécessitent en rien une opération de migration sur les modèles instances, ne sont pas anodins pour le concepteur des modèles. Le type de messages transmis au concepteur des modèles sont alors ***des préconisations d'amélioration*** qui permettent d'identifier les éléments des modèles pouvant être améliorés. L'exemple d'une évolution sans impact est l'ajout d'un stéréotype actif (instanciable) comme illustré en Figure 3. La métaclasse étendue par *StrereotypeB* ne possède pas de relation vers une autre métaclasse étendue dans le reste du profil. *StrereotypeB* possède un

ensemble de propriétés et d'opérations héritées de la métaclasse qui est vide. *StrereotypeB* possède un ensemble de propriétés, d'opérations et de relations vers le reste du profil qui peut être vide ou non. L'utilisation de ce stéréotype n'est pas indispensable dans le modèle pour maintenir sa conformité.

Intéressons-nous maintenant aux trois catégories évolutions dont l'impact rend les modèles non conformes à la nouvelle définition du profil.

CATEGORIE 2 : ***Automatic evolution category***. Dans cette catégorie, une opération de migration est nécessaire pour maintenir la conformité des modèles avec le profil évolué. Celle-ci peut être faite de manière complètement automatisée.

Reprenons le profil précédent et faisons évoluer le stéréotype *StereotypeA* en lui ajoutant une propriété *propA* de type *String* et de multiplicité minimale égale à 1 (Figure 4). Sa valeur d'initialisation est renseignée (égale à « *Ini* »). La propriété ajoutée ne doit pas être dérivée d'une autre propriété (ni être dérivée par une autre). Il ne doit exister aucune dépendance de ou vers cette propriété. L'opération de migration sera complètement automatisable car toutes les caractéristiques d'une propriété (multiplicité, type, valeur d'initialisation, conteneur) sont définies pendant l'évolution du profil.

La migration assurant la conformité des modèles est donc automatisable. Cependant, pour ce type d'évolution, des préconisations d'amélioration sont transmises au concepteur des modèles afin de l'informer sur des améliorations potentielles ultérieures (la valeur d'initialisation n'étant pas forcément adaptée à toutes les instances).

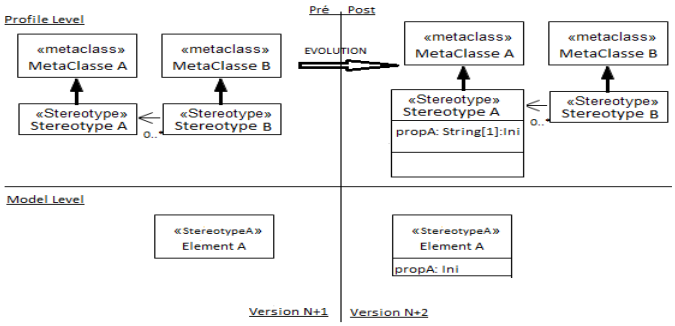

Figure 4 : *Evolution de type Automatic*

CATEGORIE 3 : ***Monitored evolution category***. Ce sont les évolutions qui nécessitent une opération de migration assistée (semi-automatisée).

Ce type d'évolution contraint l'opération de migration car certaines informations utiles manquent pour compléter l'opération. Ces contraintes bloquantes ne peuvent être résolues que par interaction avec le concepteur des modèles. Les messages d'amélioration pour ce type d'évolution correspondent aux échanges d'informations entre le système et le concepteur des modèles pour résoudre ses contraintes bloquantes et terminer l'opération de migration.

Repartons de la version précédente du profil et faisons-le évoluer (Figure 5) de telle sorte que le type de la propriété *propA* devienne une énumération. La valeur d'initialisation utilisée dans l'ancienne version du profil n'est pas incluse dans

l'énumération. A l'issue de cette évolution, aucune nouvelle valeur d'initialisation n'est renseignée. La migration sera semi-automatique car le concepteur des modèles devra cette fois donner une valeur d'initialisation aux instances de cette propriété.

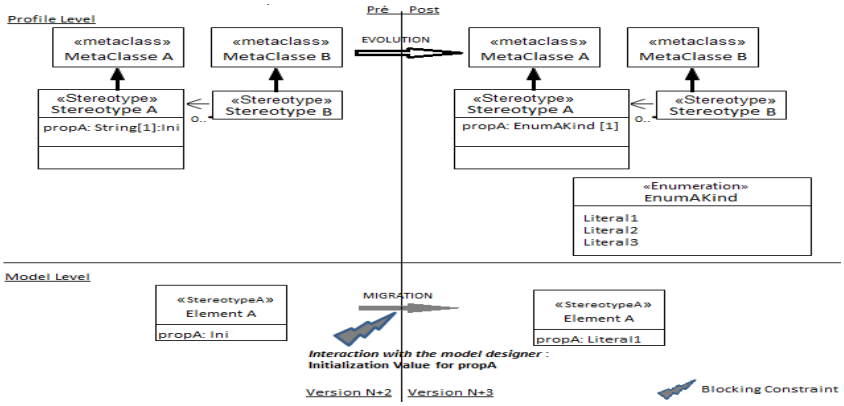

Figure 5 : *Evolution de type Monitored*

CATEGORIE 4 : ***Manual evolution category***. Cette catégorie concerne les évolutions dont la migration ne peut pas être automatisée et qui nécessitent une opération de migration manuelle effectuée par le concepteur des modèles.
Pour illustrer ce cas, partons de la version précédente du profil et faisons-le évoluer (cf. Figure 6) en modifiant la cible de l'association entre les deux stéréotypes (la cible devient *StereotypeB*) et sa multiplicité a désormais une valeur minimale. *StereotypeB* n'est plus actif mais devient abstrait. Un lien de généralisation est ajouté entre *StereotypeB* et deux nouveaux sous stéréotypes (*StereotypeC* et *StereotypeD*). Ces derniers sont actifs et possèdent un ensemble de propriétés non vide. Pour ces deux derniers stéréotypes, l'ensemble des opérations et des relations vers d'autres stéréotypes est vide. L'opération de migration doit être en mesure d'attribuer à chaque élément de type *StereotypeA* une référence vers un élément de type *StereotypeB*. Etant donné que *StereotypeB* n'est pas instanciable (actif), les éléments stéréotypés StereotypeA doivent référencer soit des éléments stéréotypés *StereotypeC,* soit stéréotypés *StereotypeD*. Ces informations constituent l'ensemble des alertes transmises au concepteur des modèles qui devra intervenir manuellement car il s'agit dans ce cas de créer un nouvel élément. Cette création ne peut pas être automatique car la description du modèle diffère selon l'utilisation d'un concept plutôt qu'un autre.

4. Adaptation des modèles instances en fonction des évolutions de profils

Notre approche d'adaptation des modèles instances vers une version conforme à la nouvelle définition du profil se décompose en quatre grandes phases : nous commençons par déterminer les différences entre deux versions d'un profil pour établir un modèle de différence. De ce modèle, nous effectuons une classification selon les catégories décrites précédemment. La troisième étape consiste à générer

automatiquement les opérations de migration spécifiques à chaque catégorie et les alertes et préconisations nécessaires pour une amélioration ultérieure des modèles.

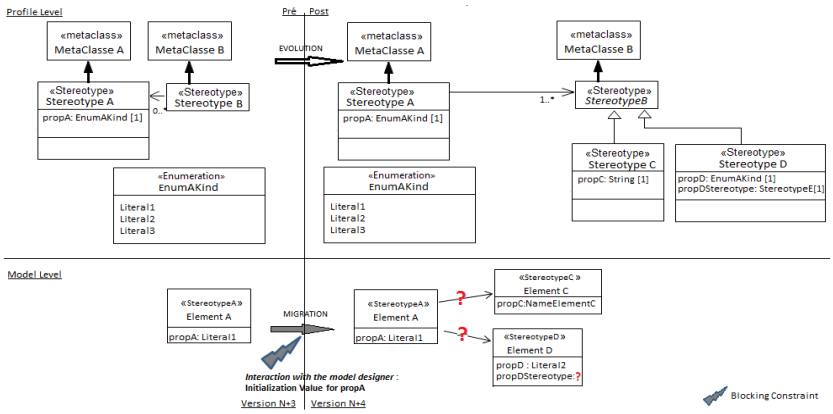

Figure 6 : *Exemple d'évolution de type Manual*

Enfin, l'étape d'optimisation consiste à assister le concepteur dans ses choix d'amélioration des modèles. Comme l'illustre la Figure 7, le concepteur des modèles n'intervient que trois fois durant le processus de migration. Tout d'abord, lorsqu'un changement de profil est détecté (cette détection est faite automatiquement) l'utilisateur lance le processus d'adaptation des modèles instances. Il intervient une seconde fois pour fournir le(s) modèle(s) instance(s) à faire évoluer. La dernière fois est durant l'opération de migration qui nécessite cette interaction pour résoudre les contraintes bloquantes. L'étape d'optimisation repose sur une interaction continue avec le concepteur.

4.1 Obtention du modèle de différence

Nous avons fait le choix d'utiliser la technologie *EMFCompare*[1] (Eclipse EMFCompare, 2009). Celui-ci possède un moteur de différence suffisant pour déterminer l'ensemble des différences entre deux modèles issus d'un même métamodèle. Les dernières améliorations de l'outil permettent désormais de comparer deux profils UML. *EMFCompare* étant un outil open-source, nous avons étendu son mécanisme pour obtenir une comparaison de modèles adaptée à notre approche. Ainsi, *EMFCompare* est maintenant en mesure d'identifier les opérations de modifications (ajout, suppression, modification) ainsi que le type de l'élément qui a évolué (stéréotype, propriété, etc.) contenues en sortie dans un modèle *Delta*.

[1] EMFCompare : *http://wiki.eclipse.org/ index.php/EMF_Compare*, (2011).

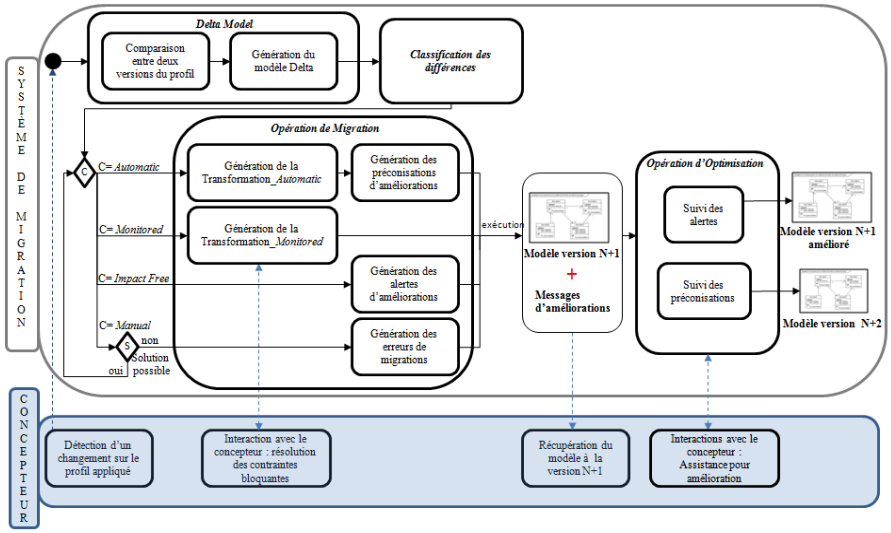

Figure 7 : *Processus d'évolution*

4.2 Classification et préparation à l'évolution

L'étape suivante consiste à classer les évolutions selon les quatre catégories que nous avons définies. Puis, pour chaque catégorie, une action spécifique est générée automatiquement.

Pour les évolutions de type « *Impact-free* », des messages de préconisations sont générés pour signaler au concepteur les éléments du système dont la description peut être optimisée.

Pour gérer les évolutions de type « *Automatic* » et « *Monitored* », la méthode que nous proposons repose sur l'exploitation du modèle de différence. En effet, nous produisons un catalogue regroupant l'ensemble des évolutions possibles entre deux versions d'un profil sous forme de patrons d'adaptation (*patterns*). De ce catalogue de patrons, nous définissons un ensemble de règles de migration correspondant à chaque patron. Dans notre cas, une règle consiste à décrire : 1/ l'état des éléments à évoluer ; 2/ les éventuelles conditions à respecter pour avoir une migration automatisée ; 3/ l'opération de migration à réaliser ; 4/ l'état des éléments attendus après migration. A partir du modèle de différence, nous générons alors automatiquement la transformation de migration propre à chaque catégorie. Cela consiste à instancier automatiquement la règle de migration correspondante (issue du catalogue de patrons). Les règles décrites dans ce catalogue prennent également en compte les contraintes bloquantes devant être résolues par interaction avec le concepteur des modèles. A l'issue de chaque transformation générée, les messages d'alertes et de préconisations sont alors créés.

Etant donné que notre approche propose une aide à la gestion de l'évolution des profils de la manière la plus automatisée possible, nous avons pour but que les évolutions de la catégorie « *Manual* » disparaissent et qu'il n'y ait à terme que des évolutions appartenant aux trois premières catégories. Pour cela, nous proposons

d'offrir des solutions permettant de revenir à un cas où la migration serait « *Automatic* » ou « *Monitored* ». Si aucune solution n'est envisageable, les éléments du modèle qui ne peuvent être migrés seront identifiés (génération de messages d'alerte), afin que le modèle instance puisse être complété.

4.3 Etapes de migration et d'optimisation

La migration complète du modèle instance est soit le résultat d'une seule migration si le modèle de différence ne possède que des évolutions d'une seule catégorie ; soit le résultat d'une somme de migrations si le modèle de différence possède des évolutions appartenant à différentes catégories. L'exécution de l'opération de migration suit l'ordre de classification des catégories, à savoir : des évolutions les plus automatisables (catégorie 1) aux moins automatisables (catégorie 4). Dans ce cas, le modèle résultat de chaque action devient un modèle intermédiaire qui servira d'entrée à l'action suivante. Dans le cas contraire, le modèle résultat devient le nouveau modèle instance. Pour des raisons de lisibilité, nous n'avons pas représenté sur la Figure 7 les modèles intermédiaire possibles ni le chemin d'exécution entre les différentes catégories.
Le traitement des alertes (en interaction avec le concepteur) finalise l'opération de migration en permettant de valider le modèle résultat par rapport à la spécification du système. Le traitement des préconisations a pour but d'aider le concepteur en permettant de créer une nouvelle version du modèle instance qui améliore la représentativité du système modélisé.

5 Approche expérimentale

Pour valider notre classification, nous avons défini une approche expérimentale en quatre étapes pour laquelle nous avons fait évoluer un profil UML défini pour la gestion des exigences (Dubois et al., 2010) (le profil DARWIN) et un modèle instance de ce profil (cf. Figure 8). Ce profil est utilisé pour décrire les exigences d'un système, celui-ci étend le métamodèle UML et se base sur le paquetage *Requirement* proposé par le profil SysML. Le profil DARWIN définit ainsi 4 sous-paquetages, 17 stéréotypes, 17 liens de généralisations, 37 propriétés, 2 opérations, 5 stéréotypes SysML spécialisés, 5 métaclasses UML étendues. La première étape consiste à faire évoluer un élément du profil en ajoutant, supprimant ou modifiant des éléments du profil ainsi que leurs caractéristiques. Puis (étape 2), d'appliquer la nouvelle version du profil sur le modèle instance initial. L'étape de validation (étape 3) consiste à vérifier si le modèle instance est toujours conforme à la nouvelle version du profil (si aucun message d'erreur n'apparaît, le modèle reste en effet conforme). La dernière étape (étape 4) consiste à identifier les évolutions rendant le modèle instance non conforme à la nouvelle version du profil ainsi que les types d'évolutions nécessitant une migration du modèle.
En ajoutant, supprimant ou modifiant chaque élément en fonction de ses propriétés (multiplicité, type, métapropriétés, …), nous avons créé 84 patrons d'évolutions pour 6 éléments évolutifs d'un profil (*Property, Association, Operation, Generalization, Extension, Stereotype*).

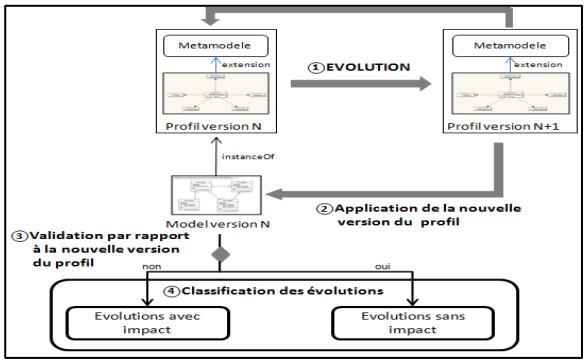

Figure 8 : *Approche utilisée pour classer évolutions de profils et leurs impacts*

6. Conclusion et Perspectives

Nous avons présenté dans cet article, une approche dédiée à la gestion des impacts des évolutions d'un profil UML avec pour objectif d'avoir un processus de migration et d'optimisation le plus automatisable possible. Nous avons déterminé une classification des évolutions d'un profil UML selon deux critères. Tout d'abord selon leur impact sur les modèles instances (préliminaire à l'étape de migration) puis selon le type de messages à transmettre au concepteur des modèles (préliminaire à l'étape d'optimisation). L'étape de migration consiste alors à générer automatiquement (selon la catégorie de l'évolution détectée) une transformation à partir du modèle de différence pour adapter les modèles instances à la nouvelle version du profil UML utilisée. La phase d'optimisation établit une interaction avec le concepteur des modèles pour améliorer la représentativité des modèles. Une implantation complète de cet outil de migration est en cours d'implémentation dans le modeleur Papyrus[2] (Papyrus, 2012). Notre méthodologie devra aussi formaliser l'ensemble des patrons d'évolutions (et implicitement de migration) les plus courant d'un profil et répartir ces patrons selon la catégorie adéquate, afin d'avoir une offre exhaustive outillée.

Par la suite, nous complèterons notre étude préliminaire effectuée ici sur le profil DARWIN à d'autres profils représentatifs et utilisés dans le domaine de la modélisation (EAST-ADL et SysML notamment) ainsi que d'autres métamodèles comme différents *benchmarks* qui nous permettront d'évaluer notre processus et de le comparer aux approches existantes.

Notre approche devra également être complétée par la prise en compte de l'ensemble des éléments de constructions utilisés dans les profils UML. Rappelons en effet qu'un profil peut définir des relations d'import ou de composition d'un ou vers un autre profil. Cette relation d'import implique qu'il existe deux types d'impacts à gérer : un impact sur le profil importateur puis un impact sur les modèles instances du profil importateur. Il s'agira alors de mesurer l'impact d'une évolution d'un import sur un profil et d'offrir une (des) stratégie(s) de migration

[2] Papyrus : *www.papyrusuml.org*

pour faire évoluer le profil importateur de manière automatisée. Puis, il faudra évaluer si une évolution d'un élément importé dans un profil impacte différemment les modèles instances. Pour les même raisons, la relation de composition (consistant à fusionner deux profils au minimum) est à prendre en considération.
Un objectif à plus long terme sera de proposer une méthodologie d'aide pour l'évolution des profils. L'objectif est d'être préventif puisqu'il s'agit de guider le concepteur du profil afin d'assurer que les modifications appartiennent bien aux catégories d'évolutions qui ne sont pas « indésirables ».

Références

Burger, E., Gruschko, B., « A Change Metamodel for the Evolution of MOF-Based Metamodels », In: *Modellierung,* vol. 161, 2010.

Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A., « Automating Co-evolution in Model-Driven Engineering », In: *IEEE Enterprise Distributed Object Computing Conference,* Washington, 2008, p. 222–231.

Dubois, H., Peraldi, M.A., Lakhal, F., « A model for requirements traceability in a heterogeneous model-based design process », In*: 15th IEEE international conference on engineering of complex computer systems,* Oxford*,* 2010*.*

Estublier, J., Favre, J.M., et al., Action Spécifique CNRS sur l'Ingénierie Dirigée par les modèles, Rapport de recherche, http://idm.imag.fr, 2005.

Gruschko, B., Kolovos, D., Paige, R., « Towards Synchronizing Models with Evolving Metamodels », In: *Processing of the Work MODSE'07,* 2007.

Herrmannsdoerfer, M., Ratiu, D., « Limitations of Automating Model Migration in Response to Metamodel Adaptation », In: *MoDSE-MCCM Workshop,* Denver*,* 2009.

Kleppe, A.G., « A Language Description is More than a Metamodel », In: *Workshop on Software Language Engineering,* Nashville*, USA,* 2007.

Koegel, M., Herrmannsdoerfer, M., Helming, J., Yang Li., « State-based vs. Operation-based Change Tracking », In: *MODELS '09 MoDSE-MCCM Workshop,* Denver*,* 2009.

Lagarde, F., Espinoza, H., Terrier, F., Gérard, S., « Improving UML profile design practices by leveraging conceptual domain models», In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering,* 2007.

Levendovszky, T., Rumpe, B., Sch¨atz, B., Sprinkle, J., « Model evolution and management », In: *Model-Based Engineering of Embedded Real-Time System,* vol. 6100, Heidelberg, LNCS Springer, 2010, pp. 241–270.

Mens, T. Demeyer, S., *Software Evolution,* Springer, 2008.

OMG, Available from World Wide Web: http://www.omg.org/spec/, 2012

Selic, B., « The theory and practice of modern modeling language design for model-based software engineering», In: *AOSD*, 2011.