وزارة التعليم العالي و البحث العلمي

Université Badji Mokhtar-Annaba

Badji Mokhtar-Annaba- University

جامعة باجي مختار– عنابـــة

Année 2017

Faculté des Sciences de L'Ingéniorat
Département d'Informatique

T H È S E

Pour obtenir le diplôme
de Docteur en Sciences

Contribution à la Gestion de la Coévolution des Modèles et des Métamodèles

Option : Informatique

Par
M^me^ ANGUEL Fouzia

Jury :

Président: M^r^ Djamel Meslati	**Pr.**	Université Badji Mokhtar -Annaba
Directeur de thèse : M^r^ Abdelkrim Amirat	**Pr.**	Université Med Chérif Messaidia-Souk Ahras
Co-directrice de thèse : M^me^ Nora Bounour	**MCA.**	Université Badji Mokhtar -Annaba
Examinatrice : M^elle^ Fadila Atil	**Pr.**	Université Badji Mokhtar- Annaba
Examinateur : M^r^ Zine-Eddine Bouras	**MCA**.	E. S. Sciences et Techniques -Annaba
Examinateur : M^r^ Yacine Lafifi	**Pr.**	Université 08 Mai 1945- Guelma

Remerciements

Au terme de cette thèse, je tiens en premier lieu à exprimer ma profonde gratitude à mon dieu qui m'a donné la force, la volonté, le courage, et surtout la patience pour terminer ce travail.

Nombreuses sont les personnes qui ont contribué à réaliser ce travail, auxquelles je présente avec plaisirs mes sincères remerciements.

Tout d'abord je remercie vivement Mr Amirat Abdelkrim professeur à l'université Med Chérif Messaadia, Souk Ahras, qui n'a pas hésité à me faire confiance et d'avoir accepté de diriger cette thèse. Il m'a fait part de son expérience et de son savoir-faire dans le domaine de l'ingénierie dirigée par les modèles. Je tiens aussi à le remercier énormément pour ses conseils ciblés et son soutien. J'ai été très heureuse de travailler sous sa direction.

Je tiens à exprimer mes reconnaissances à Mme Bounour Nora maître de conférences A à l'université Badji Mokhtar, Annaba, d'avoir co-encadré ce travail de thèse, je la remercie pour ses orientations, ses conseils avisés, ses précieuses relectures du manuscrit de thèse et des différents papiers de recherches ainsi que sa compréhension.

J'adresse également mes sincères remerciements aux membres du jury :

Mr Meslati Djamel professeur à l'université Badji Mokhtar, Annaba et directeur du laboratoire LISCO pour l'honneur qu'il m'a fait de présider le jury.

Melle Atil Fadila professeur à l'université Badji Mokhtar, Annaba, Mr Lafifi Yacine professeur à l'université 08 mai 1945 de Guelma et Mr Bouras Zine-Eddine maître de conférences A à ESST de Annaba, pour l'intérêt qu'ils ont bien voulu porter à ce travail en acceptant de l'examiner et l'évaluer. Je suis très honorée de leur participation dans ce jury.

J'aimerais aussi exprimer ma profonde reconnaissance à tous ceux qui ont collaboré de prés ou de loin dans la réalisation de ce travail.

Enfin, j'adresse un remerciement chaleureux à toute ma famille. Plus particulièrement mon époux et mes enfants dont j'ai pris beaucoup de leur temps afin d'élaborer cette thèse.

Dédicaces

Je dédie ce modeste travail :

A mon mari Sofiane pour ses encouragements, son aide et son soutien ;

A mes enfants Lina, Houssem et Sirine ;

A ma mère pour tout ce qu'elle m'a fait ;

A ma belle mère pour son aide ;

A toute ma famille et ma belle famille ;

A la mémoire de mon père.

Résumés

Résumé

L'évolution est inhérente aux systèmes logiciels en raison du progrès rapide des technologies. Comme les métamodèles sont la pierre angulaire de l'ingénierie dirigée par les modèles (IDM) et ils évoluent de manière itérative, leur évolution affecte tout le reste des artefacts qui en dépendent, à savoir les modèles ou les règles de transformation. De ce fait, les outils pour la coévolution des modèles et des autres artefacts sont indispensables. Dans cette thèse, nous proposons une approche intelligente pour l'adaptation des modèles à l'évolution de leurs métamodèles (coévolution) à l'aide de techniques basées états et basées opérateurs. Le processus de coévolution défini se compose en quatre phases. Initialement, les changements entre deux versions du métamodèle sont détectés. Ensuite, le scénario d'évolution est reconstruit en utilisant une méthode basée sur la programmation logique. Le scénario d'évolution est d'abord calculé à partir du modèle de différence et représenté par un ensemble d'opérations d'évolution primitives, puis transformé en utilisant un moteur d'inférence pour inclure des éventuelles opérations d'évolution composites ; de ce scénario les solutions d'adaptation sont générées selon l'impact de l'opérateur d'évolution au niveau modèle. Finalement, le scénario de migration est appliqué sur un modèle utilisateur en entrée conformant à l'ancienne version du métamodèle pour l'adapter à la nouvelle version de son métamodèle. Cette approche intègre également une activité de formalisation d'un catalogue d'opérateurs d'évolution en utilisant la programmation logique.

Mots clés

Evolution du métamodèle; Migration du modèle; Programmation Logique; Operateur Primitif; Operateur composite.

Abstract

Evolution is inherent in software systems because of the rapid improvement of technologies. As metamodels are the cornerstone of model driven engineering and they evolve iteratively, their evolution affects the rest of artefacts involved in a development process, e.g. models or transformation rules. Therefore, co-evolution tools for models and other artefacts are indispensable. We present in this thesis an intelligent approach to adapt models by means of state based and operator based techniques. The defined co-evolution process consists of four phases. Initially, changes between two metamodel versions are detected. After that, evolution scenario is reconstructed using logic programming method. Evolution scenario is first calculated from difference model and represented by a set of primitive evolution operations, after that it is transformed using an inference engine to include eventual composite evolution operations; from this scenario adaptation solutions are generated according to the evolution operator impact in model level. Finally, migration scenario is applied on an input model conforming to the old metamodel version to adapt it to the newer metamodel version. Furthermore this approach integrates a formalization activity of a catalogue of operators using Logic Programming.

Keywords

Metamodel evolution; Model migration; Logic programming; Primitive operator; Composite operator.

ملـخـص

إن التطور متأصل في أنظمة البرمجيات بسبب التقدم السريع للتكنولوجيات. وبما أن النماذج الفوقية تمثل محور الهندسة الموجهة بالنماذج وهي خاضعة للتطور بصفة متكررة ، فتطورها يؤثر على بقية الأدوات المتعلقة بها، على سبيل المثال،النماذج أو قواعد التحول. لذلك، فإن وجود وسائل من أجل تكييف النماذج والأدوات الأخرى لا غنى عنه. نقدم في هذه المذكرة طريقة ذكية لتكييف النماذج عن طريق التقنيات التي تعتمد على الحالات أو العمليات. تتكون الطريقة المقترحة للتطور المشترك من أربع مراحل. في البداية، يتم الكشف عن التغييرات بين إصدارين للنموذج الفوقي بعد ذلك، يتم إعادة بناء سيناريو التطور باستخدام طريقة البرمجة المنطقية. يتم احتساب سيناريو التطور أولا من نموذج الفرق ويمثل بواسطة مجموعة من عمليات التطور الأولية، بعد ذلك يتم تحويله باستخدام محرك الاستدلال ليشمل عمليات التطور المركب المحتملة. من خلال هذا السيناريو يتم إنشاء حلول التكيف وفقا لتأثير عمليات التطور في مستوى النموذج. وأخيرا، يتم تطبيق سيناريو الهجرة (التكيف) على نموذج مدخل مطابقا للنسخة القديمة للنموذج الفوقي لتكييفه مع النسخة الأحدث لهذا النموذج. وعلاوة على ذلك هذه الطريقة تحتوي على نشاط لتشكيل فهرس للعمليات باستخدام البرمجة المنطقية.

الكلمات المفتاحـية

تطور النموذج الفوقي – تكييف النموذج - البرمجة المنطقية- العمليات الأولية –العمليات المركبة.

Table des matières

Liste des figures

Liste des tableaux

Introduction Générale

Introduction Générale

Dans cette introduction, nous établissons le contexte de notre approche qui est l'évolution des langages de modélisation et leurs métamodèles. Puis, nous positionnons le problème motivant l'approche, ce problème étant la migration des modèles instances en réponse à l'évolution de leurs métamodèles. Nous présentons ensuite les contributions majeures de cette thèse et finalement nous fournissons l'organisation générale de cette thèse.

1. Contexte de recherche

Idéalement, les modèles sont construits en utilisant des langages de modélisation adéquats (Bézivin et Heckel, 2006) permettant à leurs utilisateurs d'exprimer directement les abstractions de leur domaine de problème. Le code d'implémentation peut être généré automatiquement à partir de ces modèles, en utilisant des générateurs basés sur les langages de modélisation (Czarnecki et Eisenecker, 2000). A cause du niveau plus élevé d'abstraction par rapport au développement traditionnel de logiciels basé sur le code, le développement basé sur les modèles promet d'améliorer la productivité et la qualité (Kelly et Tolvanen, 2007). La productivité du développement peut être améliorée, car le niveau plus élevé d'abstraction des langages de modélisation permet à leurs utilisateurs d'exprimer la même partie du logiciel avec moins de constructeurs. La qualité du logiciel peut être améliorée, puisque le niveau d'abstraction plus élevé des langages de modélisation permet à leurs utilisateurs et leurs outils de mieux analyser les modèles.

Différentes méthodes permettent de définir un nouveau langage de modélisation et, plus particulièrement, la syntaxe abstraite d'un langage. Les approches récentes telles que l'architecture dirigée par les modèles (Kleppe et al, 2003), les usines de logiciels [Greenfield et al. 2004] et la modélisation spécifique du domaine (Kelly et Tolvanen, 2007) préconisent de définir également les langages de modélisation dans une manière basée sur les modèles. Ainsi, un modèle représentant la syntaxe du langage de modélisation est construit, ce modèle est appelé un métamodèle. Sur la base de ce métamodèle des outils sont développés. Grâce à ces outils, des modèles seront construits conformément au métamodèle, à savoir vérifiant les règles syntaxiques définies par le métamodèle. La définition d'un nouveau métamodèle permet de définir les concepts essentiels d'un domaine, indépendamment des autres langages de modélisation. Avec l'intégration des

langages de modélisation dans les pratiques de développement industriel, leur évolution prend de plus en plus d'importance.

2. Problématique

Au cours du développement initial d'un langage de modélisation, il est souvent difficile de déterminer le niveau approprié d'abstraction (France et Rumpe, 2007). Si le niveau l'abstraction est faible, ceci implique une diminution de la productivité et de la qualité. Par contre, si le niveau d'abstraction est trop élevé, le langage de modélisation peut ne pas être suffisamment expressif pour spécifier tous les aspects nécessaires du logiciel.

Par conséquent, un langage de modélisation, comme tout autre artefact logiciel peut subir de nombreux changements (Favre, 2005). Ceci concerne à la fois les langages généraux ainsi que les langages de modélisation dédiés à un domaine. Par exemple, le langage de modélisation général UML (OMG, 2009) a déjà une riche histoire d'évolution, bien qu'étant relativement jeune. Les langages de modélisation dédiés à un domaine sont encore plus enclins à changer, puisque ils doivent être adaptés, à chaque fois que leurs domaines changent en raison des progrès technologiques ou de l'évolution des besoins (Sprinkle, 2003).

Généralement, un langage de modélisation et par conséquent son métamodèle évolue pour différentes raisons : un changement provenant d'une évolution du domaine, ou encore, une évolution due aux nombreuses corrections itératives issues des retours utilisateurs,… etc. Supposons que les concepteurs du langage ont construit une première version d'un langage de modélisation par la définition d'un métamodèle et la création d'éditeurs et générateurs de code et d'autres outils autour du métamodèle. Lorsque les utilisateurs du langage emploient les éditeurs pour construire des modèles conformes au métamodèle, ils identifient souvent de nouveaux besoins. Les concepteurs évoluent le langage de modélisation à une seconde version en adaptant son premier métamodèle aux nouveaux besoins. Dans de nombreux cas, L'adaptation du métamodèle peut invalider des artefacts existants qui dépendent du métamodèle (Sprinkle, 2003). Plus important encore, les modèles existants construits par les utilisateurs du langage peuvent perdre leur conformité vis-à-vis de la nouvelle version de ce métamodèle.

Les artefacts existants doivent migrer pour se conformer au métamodèle à nouveau, de sorte qu'ils peuvent être utilisés avec le langage de modélisation évolué. Dans cette thèse, nous nous concentrons sur la migration des modèles qui est probablement la plus difficile, car les modèles sont typiquement plus nombreux que les autres artefacts et ne sont pas généralement sous le contrôle des concepteurs des langages.

Conduire l´évolution tout en maintenant la cohérence des modèles est une tâche cruciale et coûteuse en termes de temps et de complexité. Cette complexité varie selon le type d'évolution appliquée au métamodèle et l'impact de celle-ci sur les modèles utilisant l'ancienne version du métamodèle. Par exemple, Il est relativement plus facile d'adapter les modèles à la nouvelle version du métamodèle lorsqu'il s'agit d'un changement des noms des concepts que lorsqu'il s'agit de la suppression d'une partie des concepts du métamodèle.

Différents acteurs sont impliqués dans un processus d'évolution à savoir le concepteur du métamodèle et les concepteurs de modèles. Pour des projets de petites tailles, ces rôles peuvent être joués par la même personne. De ce fait, la personne chargée de l'évolution du métamodèle connaît les motivations et les changements appliqués pour arriver au résultat qui est la seconde version du métamodèle. L'effort pour adapter les modèles instances sera moins important et plus rapide. Mais pour de grands projets, plusieurs personnes collaborent pour définir un métamodèle et le faire évoluer. Le concepteur des modèles n'est pas forcément la personne qui fait évoluer le métamodèle et n'est pas en mesure de comprendre et d'identifier correctement les changements appliqués entre deux versions du métamodèle. Pour adapter ses modèles, l'effort de la part du concepteur des modèles est donc plus élevé.

La problématique à laquelle s'intéresse cette thèse est la gestion des impacts des évolutions d'un métamodèle sur les modèles instances.

La communauté du génie logiciel a adopté, adapté et exploité plusieurs algorithmes pratiques, méthodes et techniques qui ont émergé de la communauté d'intelligence artificielle (IA). Ces algorithmes de l'IA et ces techniques trouve des applications importante et effectives dans la majorité des domaines des activités du génie logiciel (Harman, 2012). Les algorithmes IA ont déjà donné des logiciels intelligent pour l'analyse, le développement, le test ainsi que des systèmes de décision. Ces outils cherchent à supporter les méthodes et les processus de développement.

Dans cette optique, nous proposons dans ce travail de thèse d'exploiter les techniques de l'IA pour résoudre un problème qui se rapporte au domaine du génie logiciel, plus précisément l'étude des impacts de l'évolution des langages de modélisation sur les artefacts qui leur sont dépendants.

3. Objectifs et Contributions

L'objectif général de notre approche est de fournir, une aide pour la gestion des impacts

des évolutions sur les modèles instances. Le processus automatisé de gestion des impacts que nous proposons s'intéresse à trois aspects particuliers pour la gestion de la migration des modèles : l'identification des différences, la reconstruction des opérations d'évolution et finalement, la définition des solutions d'adaptation des modèles instances à la nouvelle version du métamodèle. Nous voulons dans notre approche réduire au maximum l'intervention du concepteur des modèles durant la gestion des impacts. Cette section présente les contributions suivantes à l'état de l'art actuel :

- L'encodage des métamodèles et des modèles dans le langage des prédicats.
- Formalisation d'une bibliothèque d'opérateurs d'évolution en langage des prédicats pour permettre le raisonnement et faciliter toute éventuelle extension de cette librairie.
- Proposition d'un mécanisme de raisonnement intelligent pour la reconstruction du scénario d'évolution qui constitue le noyau du processus de coévolution, ce qui assure une réelle séparation entre l'évolution du métamodèle et la migration des modèles, et par conséquent entre le concepteur du métamodèle et concepteur du modèle.

4. Contenu de la thèse

Après cette introduction générale le chapitre 1 «Modélisation et méta modélisation » introduit les termes et les concepts nécessaires à la compréhension des chapitres qui suivent. Il présente les principes fondamentaux de l'ingénierie dirigée par les modèles (IDM) et des langages de méta modélisation.

Le chapitre 2 «Evolution des métamodèles et coévolution des modèles : Etat de l'art » présente les caractéristiques des approches existantes de la coévolution des modèles ainsi que des travaux connexes à ce problème, plus précisément la gestion de l'évolution dans le domaine des schémas de bases de données et des schémas XML. L'analyse de ces caractéristiques motive notre proposition et nous permet de soulever quelques défis.

Dans le chapitre 3 « Encodage des modèles en langage des prédicats » nous présentons un langage formel c'est le langage des prédicats que nous avons utilisé pour l'encodage des modèles.

Dans le chapitre 4 « Une Approche intelligente pour la coévolution des modèles et des métamodèles » nous présentons un aperçu général du processus proposé pour l'automatisation de la coévolution des métamodèles et des modèles et nous détaillons la démarche de recherche suivie. La méthode proposée est basée sur l'encodage des modèles et des métamodèles en logique des prédicats et une bibliothèque d'opérateurs formulé en

logique ainsi que l'utilisation d'un mécanisme de raisonnement intelligent pour la reconstruction du scénario d'évolution. Dans le chapitre 5 « Implémentation et expérimentation » nous proposons les détails d'implémentation d'un prototype pour l'évaluation de notre approche. Enfin, nous concluons cette thèse en résumant les contributions de notre travail dans le domaine de la gestion de la coévolution des modèles instances. Nous abordons également les perspectives de notre travail.

Chapitre 1
Modélisation et Méta Modélisation

CHAPITRE 1
MODÉLISATION ET MÉTA-MODÉLISATION

Introduction

L'ingénierie dirigée par les modèles, est une approche de développement qui se concentre sur des préoccupations plus abstraites que la programmation classique pour maîtriser la complexité des systèmes qui ne cesse de croître. Dans l'IDM tous les artefacts manipulés sont des modèles mais on les distingue selon trois grands rôles : le modèle, le métamodèle et le modèle de transformation. Tout modèle quel qu'il soit doit toujours être conforme à son métamodèle. Dans ce chapitre nous définissons les concepts de base de l'IDM tout en détaillant la relation de conformité entre les modèles et quelques outils utilisés.

1.1. Ingénierie dirigée par les modèles (IDM)

Due aux progrès technologique, les systèmes logiciels sont devenus de plus en plus complexes. Le développement classique basé sur le code ne parvient pas à faire face à la complexité croissante menant à de plus en plus d'erreurs. L'Ingénierie Dirigée par les Modèles (IDM) (Bézivin, 2004) (en anglais Model Driven Engineering, abrégé en MDE, parfois Model Driven Development, abrégé en MDD) propose des pistes pour permettre aux organisations de surmonter les exigences du développement de logiciels.

L'IDM se réfère à l'utilisation systématique de modèles comme entités de première classe tout au long du cycle de vie du logiciel. Les approches d'IDM orientent le développement logiciel des langages de programmation de troisième génération à des modèles exprimés dans des langages de modélisation. L'objectif est d'accroître la productivité et de réduire le délai, permettant ainsi le développement des systèmes complexes au moyen de modèles définis avec des concepts qui sont beaucoup moins liés à la technologie d'implémentation sous-jacente et sont beaucoup plus proches du domaine du problème. Cela rend les modèles plus faciles à spécifier, à comprendre et à maintenir (Selic, 2003). Ceci aide à la compréhension des problèmes complexes et de leurs solutions potentielles par le bais des abstractions.

L'IDM est une forme d'ingénierie générative, par laquelle tout ou une partie d'une application informatique est générée à partir de modèles (Estublier et al, 2005). Les idées de base de cette approche sont voisines de celles de nombreuses autres approches du génie logiciel, comme la programmation générative, les langages spécifiques de domaines (DSL pour domain-specific language) (Deursen et al, 2000), le MIC (Model Integrated Computing) qui a été conçu initialement pour le développement de systèmes embarqués complexes (Karsai et al, 2005), les usines à logiciels (Software Factories) pour l'industrialisation du développement de logiciels et s'explique par analogie à une chaine de montage industrielle classique (Greenfield et al, 2004), etc. L'IDM est née d'un besoin simple : répondre aux exigences grandissantes du génie logiciel en termes de qualité des systèmes, de portabilité et d'interopérabilité (Estublier et al, 2005). Pour cela, l'IDM propose comme solution de regrouper des concepts au sein d'abstractions de systèmes : les modèles. Dans cette nouvelle perspective, les modèles doivent en contrepartie être suffisamment précis afin de pouvoir être interprétés ou transformés par des machines. Le processus de développement des systèmes peut alors être vu comme un ensemble de transformations de modèles partiellement ordonné, chaque transformation prenant des modèles en entrée et produisant des modèles en sortie, jusqu'à obtention d'artefacts exécutables. Pour donner aux modèles cette dimension opérationnelle, il est essentiel de décrire de manière précise les langages utilisés pour les représenter. On parle alors de méta-modélisation (Bézivin et al, 2001).

En résumé, les deux étapes sur lesquelles repose un processus IDM « complet » sont donc la spécification (i.e. la définition) de modèles dans un premier temps, à l'aide de formalismes de modélisation reposant eux-mêmes sur des méta-formalismes, et dans un second temps, leur transformation en d'autres modèles, en code, ou en d'autres artefacts, au moyen de langages de spécification dédiés aux transformations (Favre, 2006).

Dans les sections suivantes, nous nous proposons de définir les concepts de base de l'Ingénierie Dirigée par les Modèles afin d'appréhender de manière optimale cette approche.

a) Système

Un système est un ensemble d'éléments interagissant entre eux selon certains principes ou règles (Estublier et al, 2005).

b) Modèle

Dans de nombreuses approches d'ingénierie et disciplines, les modèles sont utilisés,

différentes sciences peuvent avoir un avis spécialisé sur le concept de modèle. Même si ’IDM s'appuie sur des modèles considérés comme des «artefact de première classe», A ce jour aucune définition de ”modèle” n’est acceptée de manière universelle. Dans (Seidewitz, 2003), un modèle est défini comme «un ensemble d'énoncés sur un système étudié". Par ailleurs, dans (Bézivin et al, 2001) un modèle est défini comme «une simplification d’un système construit avec un objectif visé à l'esprit. Le modèle devrait pouvoir répondre aux questions au lieu du système actuel". Le guide MDA (OMG, 2003) définit un modèle du système comme «une description ou spécification de ce système et de son environnement pour un certain usage. Un modèle est souvent présenté comme une combinaison de dessins et de texte. Le texte peut être dans une langue de modélisation ou dans une langue naturelle».

La notion commune émergeante est qu’un modèle est une abstraction d’un système construite dans un but précis. Un modèle est une abstraction ou une simplification dans la mesure où il contient un ensemble restreint d’informations sur un système. Il est construit dans un but précis dans la mesure où les informations qu’il contient sont choisies pour être pertinentes vis-à-vis de l’utilisation qui sera faite du modèle. Le modèle d’un système est la spécification formelle des fonctions, de la structure et/ou du comportement de ce système dans son environnement, dans un certain but. Un modèle est souvent représenté par des schémas et du texte. Le texte peut être exprimé dans un langage de modélisation ou en langage naturel. A titre d’exemple, dans la figure (1.1) une carte routière est un modèle d’une zone géographique conçu pour circuler en voiture dans cette zone. Pour être utilisable, une carte routière ne doit inclure qu’une information très synthétique sur la zone cartographiée : une carte à une certaine échelle, bien que très précise, ne serait d’aucune utilité. La relation entre un système et le modèle qui le représente (notée «Représente») est illustrée dans la figure (1.2).

Or dans le cadre de l’IDM, il est nécessaire que cette représentation soit manipulable par une machine. Donc un modèle «compatible IDM » doit être une représentation décrite dans un langage de modélisation bien défini (Estublier et al, 2005). C’est dans ce cadre qu’intervient la notion de métamodèle.

En fait, à partir d’un même système peuvent être (et sont souvent) tirés un ou plusieurs modèles de nature et de complexité différentes en fonction de la ou des caractéristiques du système qui nous intéressent, du formalisme de modélisation choisi, et même de l’appréciation personnelle des concepteurs qui établissent ces modèles.

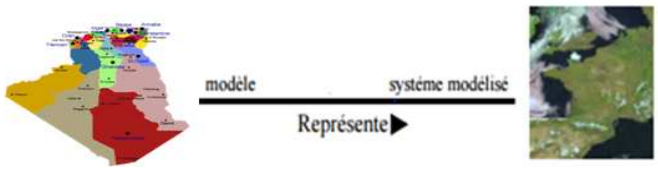

Figure 1.1. Exemple d'un système représenté par un modèle.

c) Métamodèle

Un métamodèle est un modèle qui définit le langage d'expression d'un modèle (i.e. langage de modélisation) (Favre et al, 2006). Il fournit un cadre et des concepts permettant de définir des modèles bien formés vis-à-vis de ce dernier. Dans l'exemple d'une carte routière, le métamodèle utilisé est donné par la légende qui précise, entre autres, l'échelle de la carte et la signification des différents symboles utilisés. Il existe une relation de conformité, notée «Conforme à» entre le modèle et son métamodèle comme illustré dans la figure (1.2). On dit alors qu'un modèle est conforme à un métamodèle si l'ensemble des éléments du modèle sont définis par le métamodèle (Estublier et al, 2005). Cette notion de conformité est essentielle à l'IDM mais n'est pas nouvelle : un texte est conforme à une orthographe et une grammaire, un programme JAVA est conforme à la syntaxe du langage JAVA et un document XML est conforme à sa DTD, etc.

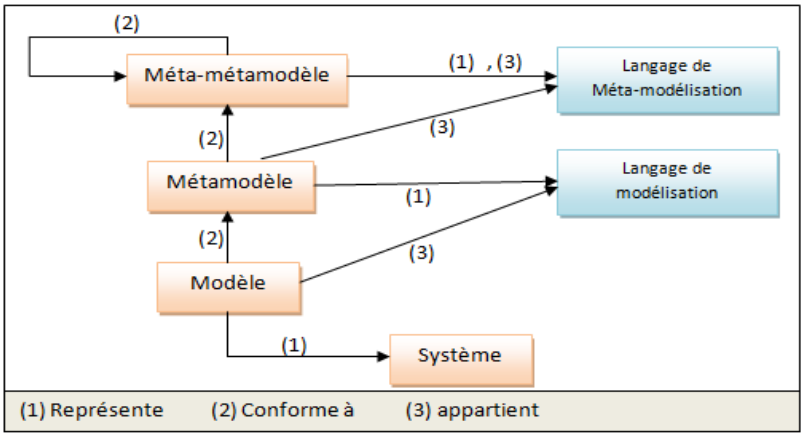

Figure 1.2. Relations entre modèle, métamodèle, méta métamodèle et langage (Fleurey, 2006).

d) Langage de modélisation

Pour être utilisé par une machine, le métamodèle décrivant un domaine particulier doit exprimer ses concepts dans un formalisme de modélisation compréhensible avec une syntaxe abstraite, une syntaxe concrète et une sémantique. Un langage de modélisation décrit ce formalisme, il est défini par une syntaxe abstraite et une ou des syntaxes

concrètes, la correspondance entre la ou les syntaxes concrètes et la syntaxe abstraite, et finalement, une sémantique (Kleppe, 2007). La syntaxe abstraite définit la structure du langage tel qu'elle est manipulée par l'ordinateur. Elle définit les concepts décrits par le langage de modélisation tels que les classes, les paquetages, les types de données, etc. Elle permet également de décrire les relations existantes entre ces concepts, par exemple les compositions de concepts autorisées, les références obligatoires, etc. La syntaxe abstraite est importante car selon la bonne construction de celle-ci, les modèles le seront à leur tour. De la même manière, une évolution de cette syntaxe entraînera une évolution des modèles l'utilisant. La syntaxe concrète définit l'apparence du langage, elle est manipulée par le développeur. Les métamodèles peuvent être utilisés pour définir la syntaxe abstraite d'un langage (Favre, 2004). Finalement, la sémantique définit un moyen permettant de comprendre la signification et le comportement de l'ensemble des modèles que peut produire le langage.

A l'opposé du langage UML[1] (Unified modelling Language) qui est souvent caractérisé de généraliste, il existe une classe de langages appelée « langage dédié» qui répond davantage aux exigences d'un domaine. À l'instar de la mouvance des Langages spécifiques de domaine (Domain Specific languages /DSL) dans les langages de programmation, les langages de modélisation dédiés (ou Domain Specific Modelling Languages /DSML), sont des langages de modélisation spécialisés pour des domaines métiers ou des problèmes particuliers (Favre et al, 2006). La syntaxe abstraite d'un DSML peut être spécifiée à l'aide d'un métamodèle. Dans le premier cas le processus de développement est plus long car on part d'un modèle plus abstrait, le second est plus ciblé car les concepteurs peuvent s'appuyer sur des technologies propres à un domaine. Néanmoins, en débutant le processus avec UML, les modèles pourront être plus facilement réutilisables. Cependant, il existe une autre solution au travers des mécanismes d'extension et de restriction du langage UML appelé profil. Cette approche n'est viable que dans le cas où le nouveau langage possède des concepts en commun avec UML. Un profil est un mécanisme qui permet d'étendre les méta-classes d'un métamodèle. Il ne peut exister s'il ne découle pas d'un métamodèle. Le profil UML permet d'étendre ou de restreindre le métamodèle d'UML afin de l'adapter à un domaine spécifique. Le recours à un profil peut être motivée par différentes raisons telles que :

- définir une syntaxe propre à un domaine qui n'existe pas dans les éléments UML.
- ajouter ou raffiner de la sémantique à un élément du métamodèle.

[1] http://www.uml.org/

- centrer le champ d'action de certains éléments trop permissifs.

Nous citons différent exemples des profils UML : « UML pour Corba », « UML pour C++ », «UML pour l'ordonnancement de la performance et du temps » (applications temps réelles), « UML pour EJB », « Test UML », « UML pour QoS et la tolérance aux pannes ». Quelques profils sont standardisés par OMG et d'autres sont définis de manière indépendante. (Bézivin, 2003).

e) Méta métamodèle

De la même manière qu'il est nécessaire d'avoir un métamodèle pour interpréter un modèle, pour pouvoir interpréter un métamodèle il faut disposer d'une description du langage dans lequel il est écrit : un métamodèle pour les métamodèles, désigné par le terme de méta métamodèle (Bézivin et al, 2001). De ce fait, le choix d'un méta-métamodèle est un choix important et l'utilisation de différents méta métamodèles conduit à des approches très différentes du point de vue théorique et du point de vue technique. A titre d'exemple, on peut citer deux approches différentes. La première c'est EBNF qui permet de définir des grammaires qui jouent le rôle de métamodèles et dont les mots jouent le rôle de modèles. La seconde c'est XML pour lequel la DTD joue le rôle de métamodèle pour des documents XML pouvant alors être considérés comme des modèles. L'idée généralement retenue est de concevoir les méta métamodèles de sorte qu'ils soient auto-descriptifs, c'est-à-dire capables de se définir eux-mêmes.

Pour résumer, sur la figure (1.2) on retrouve les trois relations de base de l'ingénierie des modèles : la conformité, la représentation et l'appartenance. Cela permet de mettre en évidence les relations qui existent entre modèles et langages : un modèle est exprimé dans un langage de modélisation (c'est-à-dire qu'il appartient à ce langage) et il est conforme au modèle qui représente ce langage. Cela permet également de préciser la notion d'autodéfinition pour le méta métamodèle : le méta métamodèle représente le langage dans lequel il est exprimé, il est donc conforme à lui même.

f) Transformation de modèle

En plus de la méta modélisation, la transformation de modèles est également une opération centrale en IDM. La transformation d'un modèle est le processus par lequel ce modèle est converti en un autre modèle (relatif au même système). La transformation de modèle est une opération qui sert à produire un modèle (appelé modèle cible) à partir d'un modèle (appelé modèle source). Elle est constituée de deux étapes, la première est une

spécification des règles de transformation décrivant la correspondance entre les concepts des métamodèles. La deuxième étape est une application des règles pour générer la transformation du modèle source en modèle cible (Bézivin, 2001). Ainsi, Les transformations de modèles sont le biais permettant de rendre les modèles « productifs » (Favre et al, 2006). Il existe différentes techniques de transformation, dont notamment des techniques basées sur l'utilisation des métamodèles qui peuvent être complètement automatisées. Les transformations sont au cœur de l'approche IDM : elles permettent d'obtenir différentes vues d'un modèle, de raffiner ou d'abstraire un modèle, ou encore de réécrire un modèle dans un langage différent. Il existe plusieurs classes de transformations de modèles, chacune remplissant différents objectifs. Une taxonomie de ces transformations est proposée dans (Mens et al, 2005). Dans cette classification, deux caractéristiques principales sont identifiées. La première concerne la relation entre le métamodèle source et destination. S'ils sont identiques alors la transformation est qualifiée d'endogène, dans le cas contraire, elle sera qualifiée d'exogène. La deuxième caractéristique concerne la variation du niveau d'abstraction. Une transformation de modèle est qualifiée d'horizontale si le modèle source et le modèle produit possède le même niveau d'abstraction, dans le cas contraire, elle sera qualifiée de verticale.

1.2. Architecture dirigée par les modèles (MDA)

1.2.1. Le principe du MDA

L'objectif principal d'une approche d'ingénierie dirigée par les modèles (Bézivin, 2004) est de reporter la complexité d'implémentation d'une application au niveau de sa spécification. Cela devient alors un problème d'abstraction du langage de programmation en utilisant un processus de modélisation abstrait.

L'architecture dirigée par les modèles (MDA) est une démarche de développement à l'initiative de l'OMG (Object Management Group) rendue publique fin 2000. MDA est basée sur les problématiques suivantes (Soley, 2000) :

- Spécification d'un système indépendamment de la plateforme sur laquelle ce système doit être déployé,
- Spécification de plateformes,
- Définition d'une plateforme pour un système,
- Transformation des spécifications d'un système en un système associé à une plateforme particulière.

L'idée de base du MDA est de séparer les spécifications fonctionnelles d'un système des

spécifications de son implémentation sur une plateforme donnée. Pour cela, le MDA définit une architecture de spécifications structurée en modèles indépendants des plateformes (Platform Independent Models, PIM) et en modèles spécifiques (Platform Specific Models, PSM) (Bézivin et al, 2003). L'approche MDA permet de réaliser le même modèle sur plusieurs plateformes grâce à des projections normalisées. Elle permet aux applications d'inter-opérer en reliant leurs modèles et supporte l'évolution des plates-formes et des techniques. La mise en œuvre du MDA est entièrement basée sur les modèles et leurs transformations. La figure 1.3 donne une vue générale d'un processus MDA en faisant apparaître les différents niveaux d'abstraction associés aux modèles, depuis les modèles de besoins jusqu'au code qui s'exécute.

a) **Le modèle d'exigence (CIM)** : Le CIM (Computation Independent Model) Aussi appelé modèle de domaine ou modèle métier. Ce modèle ne présente aucune trace informatique, il pourrait s'agir d'un texte. Il capture les exigences en termes de besoins et décrit la situation dans laquelle le système sera utilisé. Son but est d'aider à la compréhension du problème mais aussi de fixer un vocabulaire commun pour un domaine particulier. Dans la pratique, l'appellation « CIM » est très peu utilisée. Le modèle CIM sert de base à la définition d'un modèle PIM.

b) **Le modèle indépendant de toute plateforme (PIM)** : Le PIM (Platform Independent Model) qui est un modèle d'analyse et de conception devant, en théorie, découler, au moins partiellement, des CIM. Il décrit le système indépendamment de la plate-forme cible sur laquelle il s'exécutera. Il présente donc une vue fonctionnelle détaillée du système, sans détails techniques. Il peut être raffiné progressivement jusqu'à intégrer des détails d'architecture spécifiques à un type de plate-forme (machine virtuelle, système d'exploitation, etc.) mais il doit rester technologiquement neutre.

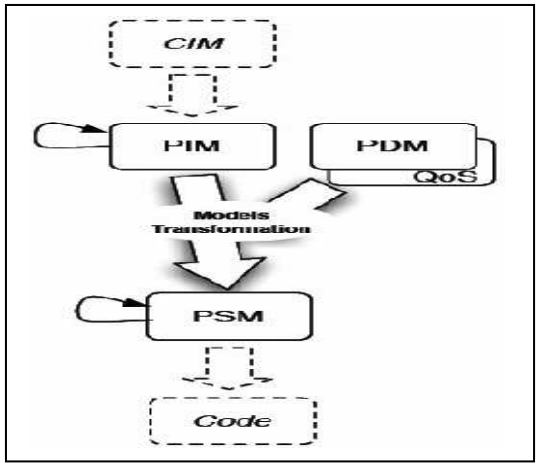

Figure 1.3. Principe du MDA (Bézivin et al, 2001).

c) **Le modèle de description de la plateforme (PDM)** : Le PDM (Platform Description Model) est le modèle qui décrit une plate-forme d'exécution. Il fournit un ensemble de concepts techniques représentant les différentes parties de la plate-forme et/ou les services qu'elle fournit. Un PDM peut représenter par exemple, des plates-formes à base de composants comme CCM (OMG, 2011) ou EJB.

d) **Le modèle spécifique à une plateforme (PSM)** : Le PSM (Platform Specific Model) qui est un modèle spécialisé pour une plateforme d'exécution particulière. Il est obtenu par des transformations combinant les PIM avec les informations des plateformes contenues dans les PDM. Il représente une vue technique détaillée du système. Il peut exister avec différents niveaux de détails. Dans sa forme la plus détaillée, il sert de base à la génération de l'implémentation. A l'opposé des PIM, les PSM sont considérés comme des modèles jetables qui ne jouent qu'un rôle d'intermédiaire pour faciliter la génération du code de l'application finale.

1.2.2. Avantages du processus MDA

Le processus MDA présente quatre avantages (Selic, 2003):

Réutilisation : Chaque modèle augmente la réutilisabilité en factorisant des informations de plus en plus proches de la réutilisation. Le modèle CIM servira à toute réalisation informatique ; le modèle PIM servira et pourra être le point de départ à toute réalisation adoptant le même paradigme ; le modèle PSM servira à toute implémentation sur des plateformes d'une même technologie pour un certain nombre de version de cette dernière.

Capitalisation : Les mécanismes de transformation automatique (PIM-PSM) ou les consignes de retranscription (CIM-PSM) sont «standards » et elles s'enrichissent de manière continue. Cet aspect présente un haut degré de capitalisation.

Intuitivité : Le modèle PSM permet d'appréhender plus facilement la réalisation finale.

Progressivité : Le modèle PSM va permettre une génération de code assez efficace.

1.2.3. Les niveaux de modélisation

Dans l'architecture MDA, l'OMG définit quatre niveaux d'abstraction où la notion de modèle est utilisée. La figure (1.4) présente cette architecture (Bézivin et al, 2001). Entre chaque niveau, la relation « conforme à » s'applique, chaque niveau inférieur est conforme au modèle défini au niveau supérieur. Plusieurs variantes d'un système réel peuvent exister en fonction des valeurs réelles d'implémentation (niveau M0). Pour capturer l'essentiel du système indépendamment des valeurs d'implémentation, les fonctions et l'architecture de

ce dernier doivent être modélisées au niveau M1. Le langage de modélisation employé pour définir ce système est classé au niveau M2. Les entités permettant de créer ce nouveau langage doivent être spécifiées au niveau M3. Dans la figure (1.4) nous retrouvons le MOF au sommet de la pyramide. Le MOF est son propre métamodèle. Il est réflexif car il est capable de se décrire à partir de ses propres concepts pour éviter la nécessité d'une infinité de niveaux. En dessous, au niveau M2 chaque métamodèle définit un langage par domaine d'intérêt. On y retrouve par exemple UML qui permet de modéliser les artefacts d'un système logiciel orienté objet. C'est également à ce niveau que sont définies les extensions des concepts généraux du métamodèle UML, les relations entre les nouveaux concepts du langage, ainsi que les contraintes potentielles sur le langage UML. En somme, c'est à ce niveau que sont définis les profils UML (Bézivin et al, 2003); ceci justifiant d'autant plus leur définition comme une alternative aux métamodèles. Les profils UML et les métamodèles sont en effet deux méthodes distinctes pour décrire un nouveau langage de modélisation et elles se situent toutes les deux au même niveau M2. Un modèle au niveau M1 ne définit aucun nouveau concept du langage, il n'est que l'instance d'un langage défini en M2 et est conforme à celui-ci. De manière générale, un modèle au niveau M1 est conforme à une syntaxe abstraite qui peut être soit un métamodèle soit un profil UML. Enfin, le système réel est défini au niveau M0, il ne s'agit plus d'un modèle mais d'une réelle conception potentiellement exécutable d'un système. La structure et les interactions étant déjà modélisées au niveau supérieur, le concepteur n'a plus qu'à instancier ce modèle en le complétant de valeurs réelles pour chaque élément.

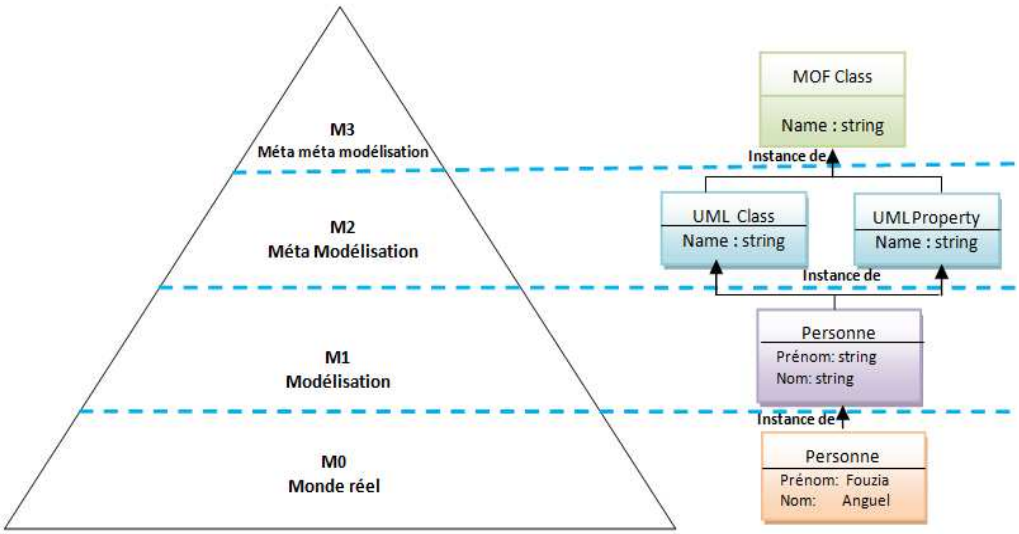

Figure 1.4. Les niveaux d'abstraction de l'architecture MDA.

1.2.4. Les langages de méta-modélisation

a) Meta-Object Facility (MOF)

Malgré son succès, il était clair qu'UML possédait des limites et que d'autres métamodèles adressant des besoins différents seraient nécessaires. Afin d'éviter une prolifération de métamodèles incompatibles évoluant indépendamment, il était nécessaire de mettre en place un cadre permettant de formaliser la définition des métamodèles (Bézivin et al, 2001). La solution fut de proposer un langage pour définir des métamodèles, c'est-à-dire un méta-métamodèle, appelé Meta-Object Facility (MOF) (OMG, 2011). MOF est basé sur les concepts fondés dans le paradigme de programmation orientée objet et il est décrit en utilisant un sous ensemble des concepts d'UML.

Le langage MOF a été défini par l'OMG et standardisé en novembre 1997 (OMG, 1997). Il permet de définir des métamodèles en spécifiant l'ensemble des concepts et des relations existantes entre eux. Comme vu précédemment, définir un métamodèle revient à définir le modèle d'un modèle, Il s'agit de faire de même à un niveau d'abstraction supérieur. Le langage MOF est un méta métamodèle car il est le métamodèle d'un métamodèle. L'une des particularités du MOF est qu'il est réflexif, c'est-à-dire qu'il est son propre métamodèle.

OMG a proposé MOF (OMG, 2003) comme un standard pour spécifier les métamodèles. Par exemple, le métamodèle UML est défini en termes de MOF. Un standard qui supporte MOF est XMI (OMG, 2002), qui définit un format d'échange de modèles à base d'XML.

Le langage MOF est en fait la partie infrastructure du langage UML. Ils partagent les mêmes notions de classes, de paquetages, etc. UML Infrastructure est défini en trois sous niveaux internes L1, L2 et L3. L1 est le sous-ensemble minimum pour décrire un diagramme de classe, il possède tous les concepts de base d'un langage orienté objet excepté la notion d'association entre classes. Ce niveau est regroupé dans un paquetage nommé "BasicPackage". Le niveau L2 regroupé dans le paquetage "ConstructPackage" enrichit "BasicPackage" en lui ajoutant le concept association et en le raffinant du même coup. Ces deux paquetages dans le langage MOF sont appelés respectivement EMOF (Essential MOF) et CMOF (Complete MOF) (OMG, 2011). EMOF et CMOF sont deux évolutions de MOF 1.4 définies lors de la normalisation d'UML 2.0. Comme leurs noms l'indiquent, la différence entre ces deux langages réside dans le nombre de concepts qu'ils contiennent. CMOF fournit des concepts supplémentaires pour définir des métamodèles qui imposent des contraintes supplémentaires sur les modèles, EMOF quant à lui ne contient que les concepts essentiels à la méta-modélisation (OMG, 2011).

Parmi les langages de méta-modélisation existants, nous avons retenu EMOF pour les

travaux présentés dans ce document. Les deux raisons du choix de EMOF sont, d'une part, le fait que ce langage soit normalisé et d'autre part le fait qu'il soit bien supporté par des outils comme Eclipse/EMF. La figure (1.5) présente le méta métamodèle EMOF dans son ensemble. EMOF permet de définir des packages qui contiennent des types. Les types EMOF sont soit des DataType (types primitifs ou types énumérés), soit des classes. Les classes EMOF peuvent avoir un nombre quelconque de super-classes (La propriété superClass de la classe Class) et sont composées de propriétés et d'opérations. En tant que langage de description de structures de données, EMOF ne permet pas de spécifier le corps pour les opérations. Le système de type pour EMOF est très simple : une classe définit un type et le sous-typage n'est possible que grâce au sous-classage. Une des raisons pour lesquelles EMOF présente peu de concepts par rapport à d'autres langages de métadonnées est la représentation "compacte" qui est utilisée pour les attributs, les références, les associations et les compositions. En effet, la notion de propriété (la classe Property sur la figure 1.5) permet de représenter tous ces éléments. Les modèles EMOF peuvent être représentés par des diagrammes d'objets UML et les métamodèles EMOF peuvent être représentés par des diagrammes de classes UML. Nous présentons dans ce qui suit une description détaillée des éléments constructeurs des métamodèles EMOF qui sont intégrés dans le champ de notre étude.

- ***Classe Abstraite « Abstract classes » :*** Si une classe est abstraite, aucune instance représentant cette classe ne doit être présente dans le modèle. Par conséquent, les classes abstraites fournissent un moyen pour extraire les caractéristiques communes dans une super classe qui ne nécessite pas d'avoir des instances.
- ***Types primitifs « Primitive types »*** : Les instances des types primitifs sont également représentées par des nœuds comme les classes, cependant, par rapport aux classes, les types primitifs ne possèdent pas des références.
- ***Types de données « Data types »*** : Les types de données représentent les types primitifs prédéfinis comme booléen, entier, chaîne,… Pour chaque littérale de ces types de données, il ya un nœud dans les modèles instances. Les types de données définissent un nombre infini de littéraux, e.x : type chaîne. Mais, un modèle peut seulement utiliser un nombre fini de littéraux d'un type de données.
- ***Les énumérations et les littéraux « Enumerations and literals » :*** Une énumération peut définir un ensemble fini de littéraux. Chaque littéral utilisé dans le modèle est représenté par un nœud. Un littéral possède son propre nom pour le distinguer des autres littéraux.
- ***Les types « types » :*** Type est une super classe abstraite commune entre « class » et

« Data type ». Pour distinguer les types, un type a un nom. Le nom d'un type doit être unique entre tous les types dans un même package.

- ***Les propriétés « Property » :*** Dans les métamodèles MOF, une propriété définit soit un attribut ou une référence. Les attributs possèdent un type primitif comme type. Par contre les références sont typées par des classes. Une propriété est définie par :
  - un nom (name) qui doit être unique parmi les propriétés de la classe à laquelle elle appartient y compris les propriétés héritées
  - un type (type) qui spécifie le type des éléments que peut contenir la propriété. Le type d'une propriété peut être une classe, un type primitif ou une énumération.
  - une multiplicité (lower, upper, isOrdered et isUnique) qui précise combien d'éléments peut contenir la propriété et si ces éléments doivent être uniques et/ou ordonnés.
  - un booléen isComposite qui spécifie si la propriété correspond à une composition.
  - un booléen isDerived qui spécifie que la propriété est dérivée. Dans ce cas la propriété ne contient pas directement de valeur mais permet de calculer une valeur.
  - Un booléen isIdentifier une propriété peut servir comme identificateur pour les instances d'une classe. Les valeurs d'un identificateur doivent être uniques entre toutes les instances de la classe de cette propriété. Une classe possède au plus une propriété dont Isidentifier est à vrai.
  - La valeur par défaut « default value » : un attribut peut avoir une valeur par défaut pour faciliter l'initialisation de l'attribut
  - une éventuelle propriété opposée (opposite). Un couple de propriétés opposées permet de représenter une association bidirectionnelle entre deux classes.
- ***Les packages « packages » :*** Pour regrouper les types reliés, ils peuvent être organisés dans des packages. Un package consiste en un nombre de types et peut avoir des sous packages. Les packages à leur tour sont distingués par des noms. Le nom d'un sous package doit être unique entre les packages qui appartiennent au même super package. Le nom d'un package racine doit être unique entre les packages n'ayant pas des super packages.

Bien qu'EMOF ne permette pas de décrire le comportement des opérations, le modèle de la figure (1.5) comporte des opérations dont la sémantique est décrite en anglais dans la norme définissant le langage.

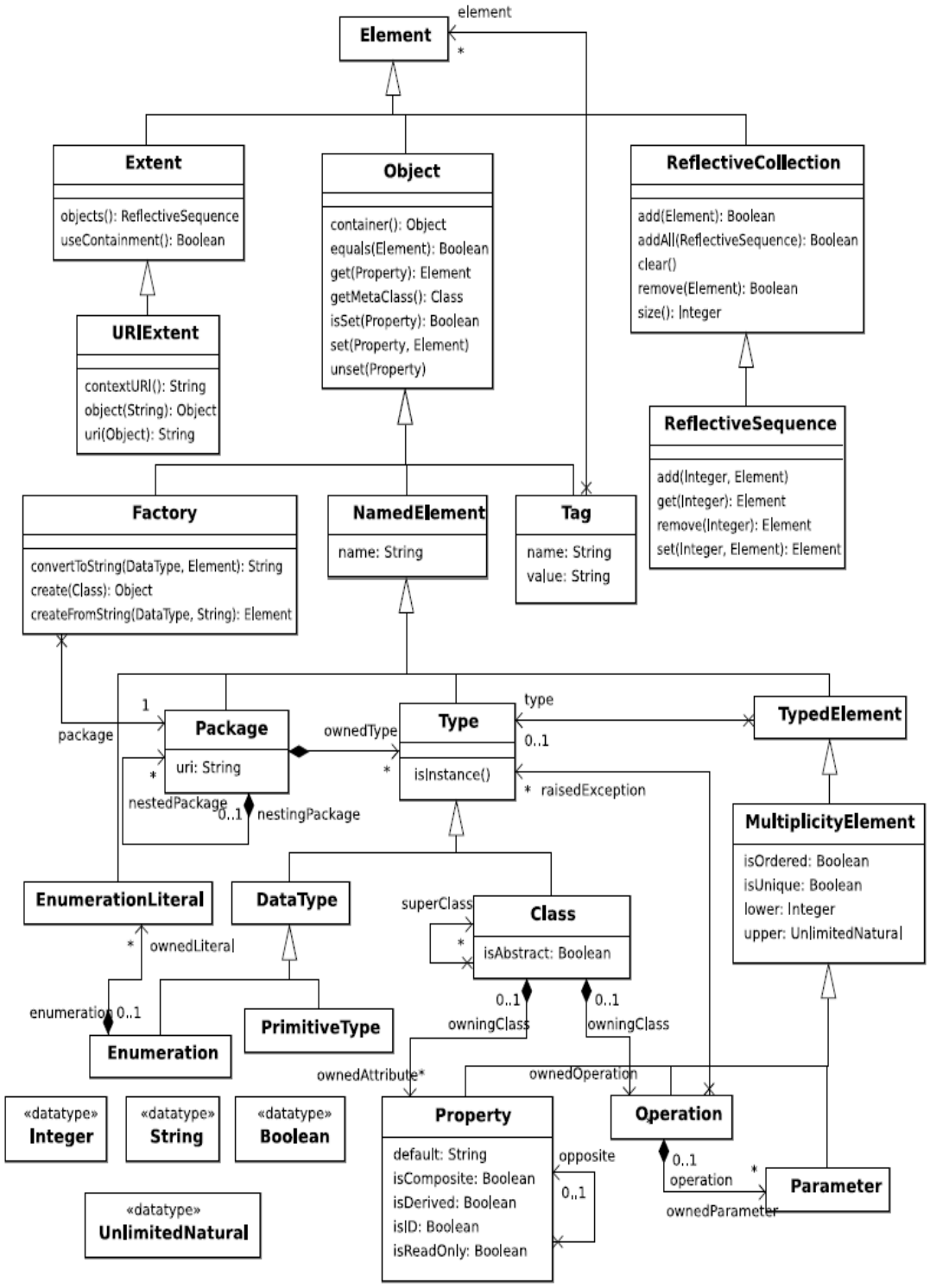

Figure 1.5. Essential Meta-Object Facilities (EMOF) (OMG, 2015).

b) Ecore de Eclipse Modelling Framework

Pour pouvoir construire des modèles avec un langage de modélisation, des outils sont nécessaires pour éditer et interpréter les modèles. Les ateliers des langages (Fowler, 2005)

tel que l'« Eclipse modelling Framework » (EMF), « Metacase » (MetaEdit+), « Generic Modelling Environnement» (GME) (GME, 2016) ou « outils DSL de Microsoft » réduisent de manière significative les efforts pour la construction des outils supports des langages de modélisation autour des métamodèles.

Dans cette thèse nous utilisons EMF, puisqu'il représente la plateforme de modélisation la plus utilisée pour l'implémentation de EMOF. Essentiellement le langage Ecore (Ecore, 2013) définit un environnement de modélisation EMF basé sur EMOF. Son but est de permettre une génération de code automatique des interfaces et des classes d'un métamodèle pour pouvoir mettre à disposition des éditeurs réflexifs (permettant les manipulations des concepts du métamodèle).

Le langage Ecore se distingue du langage MOF et EMOF en particulier par deux notions. Les associations ne sont pas représentées mais remplacées par des références directes entre les concepts. Pour générer du code, Ecore intègre également les notions de Java nécessaires par le concept d'EAnnotation. Les métaclasses Ecore sont toutes préfixées par la lettre 'E' (EClass, EAttribute, EString, etc.). La figure 1.6 présente un extrait du métamodèle Ecore utilisé lors des définitions de nouveaux métamodèles. Les concepts d'un nouveau langage de modélisation sont des EClass qui peuvent définir des propriétés (EAttribute), des opérations (EOperation) ou des relations (EReference) vers d'autres EClass. Un des principaux avantages de Ecore est sa simplicité et le grand nombre d'outils disponible. Actuellement Ecore est devenu le standard dans les plateformes de l'ingénierie dirigée par les modèles.

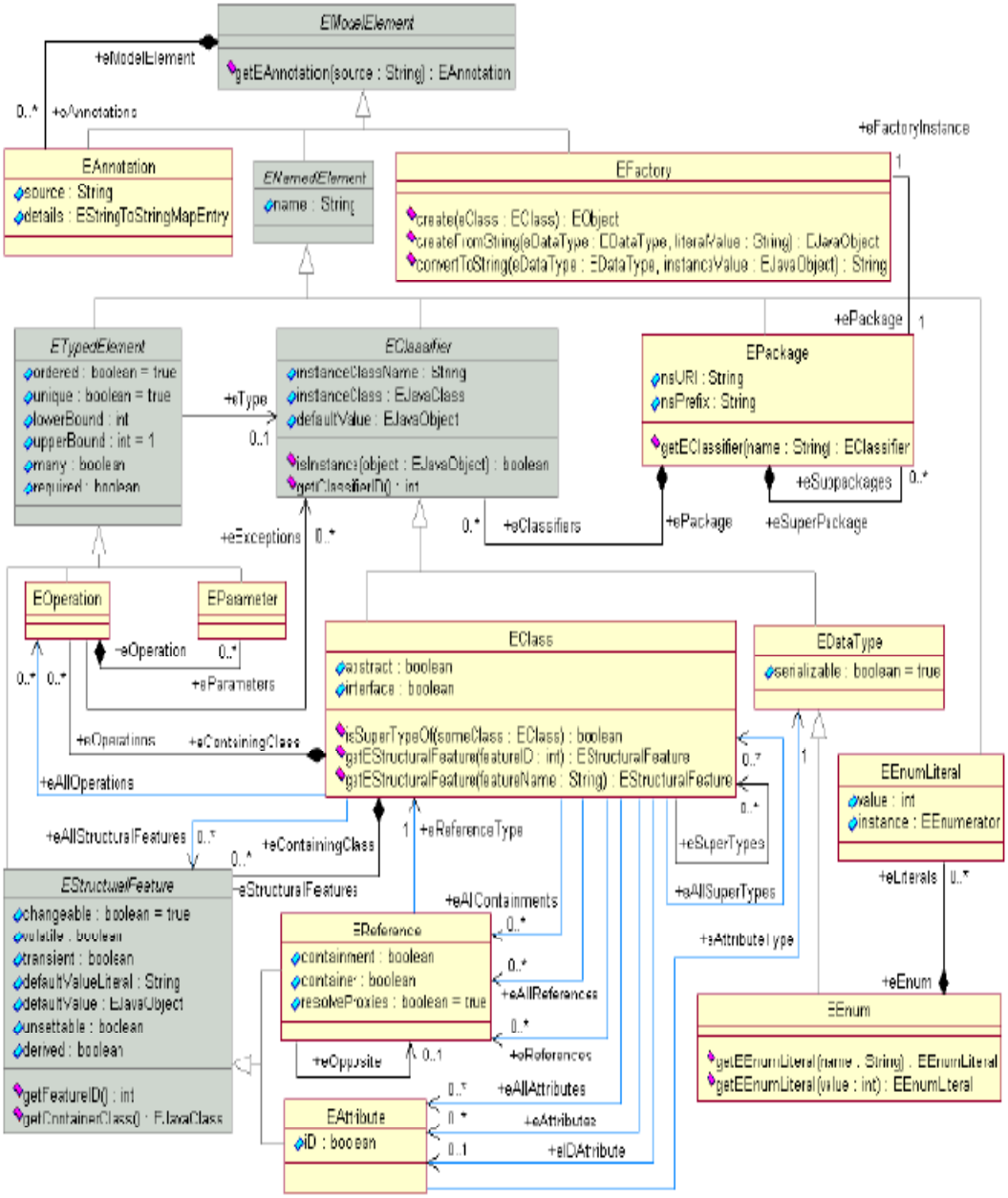

Figure 1.6. Architecture des composants d'Ecore (Ecore, 2013)

Conclusion

Dans ce chapitre, nous avons introduit les concepts de base de l'ingénierie dirigée par les modèles. Les notions de modèle, métamodèle, langage de modélisation, méta métamodèle, transformation de modèle ont été fournies. Nous nous sommes attachés à mettre en évidence le principe du MDA et l'outillage sous-jacent dans le processus de conception des systèmes tel qu'il est vu dans le contexte de l'IDM.

Chapitre 2

EVOLUTION DES MÉTAMODÈLES ET COÉVOLUTION DES MODÈLES : ETAT DE L'ART

CHAPITRE 2
EVOLUTION DES MÉTAMODÈLES ET COÉVOLUTION DES MODÈLES : ETAT DE L'ART

Introduction

L'exactitude d'une application est principalement déterminée par la satisfaction de l'utilisateur plutôt que par le respect de la spécification donnée. En fait, la spécification ne décrit que les besoins initiaux, et il n'est tout simplement pas réaliste de supposer la possibilité de concevoir une application complète depuis le début. Le changement des environnements, changement des besoins, développement des concepts et les nouvelles technologies exigent une adaptation continue pour préserver la satisfaction des utilisateurs. Si l'application n'évolue pas, elle sera inévitablement abandonnée. Par conséquent, l'évolution est un besoin incontournable dans le cycle de vie du logiciel. Il y a plusieurs travaux de recherche pour l'évolution des logiciels. Les publications dans ce domaine existent avant 1970 (Lehman, 1969). Avec l'émergence de l'IDM, la complexité des logiciels est mieux maitrisée et la gestion de l'évolution est du même coup simplifiée. En effet, la notion de modèle est à un niveau d'abstraction élevé qui permet l'indépendance vis-à-vis du code et de la plateforme cible. Les considérations du langage, de l'architecture ou des contraintes matérielles sont dissociées de la définition du système et donc de son évolution. Les concepteurs peuvent ainsi se focaliser sur le développement même du système et n'avoir à considérer l'évolution que d'un point de vue de l'analyse d'impact et de la propagation des modifications. Comme tout Logiciel un langage de modélisation est sujet à l'évolution en raison du changement des besoins ou des progrès technologiques. Ainsi, un langage de modélisation et par conséquent son métamodèle devraient évoluer au fil du temps pour étendre et adapter les objectifs (Favre, 2003). A cause de l'évolution du métamodèle, les modèles existants peuvent ne pas être conformes au métamodèle évolué et nécessite donc d'être mis à jour.

2.1. Définition de l'évolution

Dans le dictionnaire français (Dic_OLF, 2016) le terme évolution désigne une transformation progressive, une suite de transformations graduelles ou une suite de petits changements

successifs. Par exemple, dans le domaine de la médecine, ce terme indique les différents stades par lesquels passe une maladie. En biologie c'est une transformation des espèces vivantes qui se manifeste par des changements de leurs caractères génétiques pouvant aboutir à de nouvelles espèces. En informatique on parle d'évolution des logiciels, suite à des changements technologiques, l'apparition de nouveaux besoins fonctionnels et non fonctionnels, et l'évolution des modèles pour rester en adéquation avec les logiciels qu'ils modélisent.

Pour l'évolution des systèmes, les travaux dans (Cicchetti et al, 2008) distinguent deux types d'évolution. Les évolutions internes à l'application (création ou destruction d'instances, création ou destruction de liaisons) appelées aussi évolutions structurelles, et les évolutions externes à l'application qui permettent d'ajouter de l'information sur une préoccupation de l'application (déploiement, sécurité …etc.) au niveau de la spécification du schéma d'instance appelées aussi évolutions comportementales. Dans la littérature certains auteurs utilisent le terme maintenance, d'autres parlent d'adaptation pour désigner l'évolution logicielle. Elle peut être réalisée à l'étape de spécification ou de conception du système «évolution statique ». Elle peut être réalisée à l'exécution de ce système « évolution dynamique ».

Trois activités adressant le changement des besoins, l'adaptation aux nouvelles technologies et la restructuration architecturale ont été identifiées comme les principales causes de l'évolution des logiciels (Sjoberg, 1993). Ces activités sont les motivations pour trois types communs de l'évolution du logiciel selon la norme ISO/IEC (ISO/IEC, 2006), ces types d'évolution sont :

- Les évolutions de type correctives sont généralement dues aux évolutions des exigences. Elles correspondent également à des corrections d'un comportement défectueux ou inadapté.
- Les évolutions de type adaptatives sont généralement dues aux évolutions des technologies utilisées par le système. Elles correspondent à des adaptations du système face aux nouvelles contraintes de la plateforme ou du langage utilisé.
- Les évolutions de types perfectives sont généralement dues aux contraintes de l'architecture cible du système. Elles correspondent à une amélioration de la qualité du système en termes de performances, de robustesse, etc.

Les évolutions que nous considérons dans ce travail et auxquelles nous proposons des solutions sont les évolutions de type adaptatives. Plus particulièrement, nous considérons les évolutions d'un métamodèle qui ont un impact sur les modèles décrivant un système.

Depuis les années 80, l'évolution ne s'est pas limitée aux corrections directes de bugs ou

aux modifications des fonctionnalités simples d'un système elle s'est intégrée plutôt comme une étape clé dans le processus de développement des logiciels. C'est avec les travaux de Lehmann (Lehmann, 1980) qu'une activité pour la gestion de l'évolution sera créée et considérée tout au long du processus de développement des systèmes logiciels. Lehmann soutient que sans une phase de maintenance adaptative, la qualité du système diminue de manière significative. Avec l'émergence de l'IDM, la complexité des logiciels est mieux maitrisée et la gestion de l'évolution est du même coup simplifiée. En effet, la notion de modèle est à un niveau d'abstraction élevé qui permet l'indépendance vis-à-vis du code et de la plateforme cible. Les considérations du langage, de l'architecture ou des contraintes matérielles sont dissociées de la définition du système et donc de son évolution. Les concepteurs peuvent ainsi se focaliser sur le développement même du système et n'avoir à considérer l'évolution que d'un point de vue de l'analyse d'impact et de la propagation des modifications.

2.2 . Définition de la coévolution

Dans le dictionnaire français, le terme coévolution désigne une évolution simultanée, parallèle et/ou interdépendante de deux entités de même espèce. Par exemple, les plantes à fleurs et les insectes qui en assurent la pollinisation.

Dans (Levendovszky et al, 2010) la coévolution est définit, dans son sens le plus large, comme une adaptation évolutive qui se produit chez plusieurs éléments (gènes ou espèces) à la suite de leurs influences réciproques.

Dans le domaine de la modélisation, un métamodèle représente la définition d'un langage de modélisation et un modèle est une instance du métamodèle. Les métamodèles sont exprimés dans un formalisme de méta-modélisation tel que MOF standardisé par OMG (OMG, 2006) et Ecore (Ecore, 2013). Tous ces formalismes fournissent des moyens similaires au diagramme de classe UML.

Il arrive souvent que les métamodèles évoluent au cours du processus de conception ou de maintenance d'un système logiciel. Dans de tels cas, il est souvent demandé d'adapter les modèles conformes au métamodèle initial, de telle sorte qu'ils soient conformes au métamodèle évolué. Ce processus est dénommé coévolution des modèles (ou évolution couplée des métamodèles et modèles). Dans la littérature, ce problème est appelé évolution du métamodèle et coévolution des modèles (Wachsmuth, 2007), adaptation des modèles (Garcès et al, 2009) ou migration des modèles (Rose et al, 2010). Dans notre travail nous utiliserons ces termes de manière indifférente.

En prenant l'exemple de la figure (2.1), si le métamodèle évolue le modèle doit aussi

évoluer afin de rester conforme au nouveau métamodèle. Donc le modèle A, dans ce cas, doit évoluer afin d'atteindre le modèle B de plus il doit aussi évoluer pour être conforme au nouveau métamodèle B. Ce qui est appelé une coévolution.

Le travail présenté dans cette thèse se concentre sur cette dernière activité de coévolution dans le cadre de l'IDM.

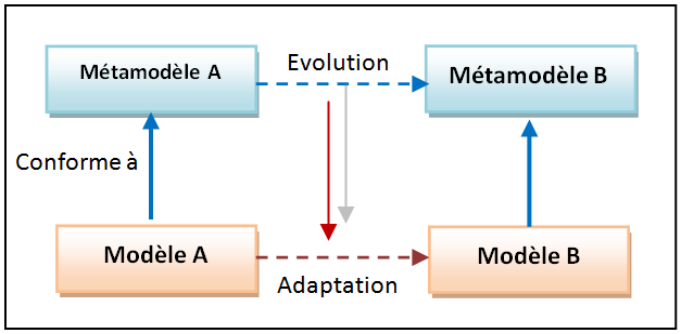

Figure 2.1. Processus d'évolution et de coévolution.

Afin d'effectuer la coévolution des modèles, deux sous-problèmes doivent être résolu. Le premier problème est de savoir comment calculer les différences entre le métamodèle initial et le métamodèle évolué. Le deuxième problème est de savoir comment adapter les modèles en se basant sur les différences calculées. Les approches existantes pour la coévolution diffèrent grandement selon que les différences entre les modèles sont connues (approche basée sur les opérateurs) ou ne sont pas connues à l'avance (approches à base d'états)

Dans cette thèse nous utilisons l'exemple de la littérature qui illustre un scénario d'évolution du métamodèle représentant des réseaux de Petri (Wachsmuth, 2007). La figure (2.2.a) et figure (2.2.b) représentent le métamodèle avant et après son évolution sous forme d'un diagramme de classes UML, (MM1) et (MM2) respectivement. Dans le métamodèle du réseau de Petri (MM1), Un réseau consiste en un certain nombre de « Places » et de « Transitions ». Chaque « Transition » a au minimum une « Place » entrée et une « Place » sortie. Comme un réseau de Petri sans aucune « Place » et aucune « Transition » n'a pas de sens, la nouvelle version du métamodèle (MM2), représenté par la figure (2.2.b) restreint le réseau pour comprendre au minimum une « Place » et une « Transition », en changeant les multiplicités inférieures dans les relations de composition dans (MM1). En plus, (MM2) rend les arcs entre « Places » et « Transitions » explicites en introduisant les classes « PTArc » et « TPArc ». Par conséquent, les relations entre « Place » et « Transition » disparaissent puisque ces relations sont explicitement représentées par

« PTArc » et « TPArc ». De même puisque « PTArc » et « TPArc » les deux représentent des arcs, ils sont généralisés par la classe « Arc ». Par ailleurs, les arcs peuvent être annotés par des poids. Cette extension est reproduite par l'introduction d'un nouveau attribut « Weight » dans la classe « Arc ». Finalement, (MM2) incorpore une nouvelle classe « Token » ainsi que la relation de composition relative avec la classe « Place » pour couvrir l'aspect dynamique des réseaux de Petri et ceci en marquant les places avec des jetons.

Le tableau (2.1) résume les différences observables par un concepteur de modèles. Une comparaison visuelle entre les deux modèles nous a permis de déterminer ces différences sans utiliser un moteur de différence automatique.

La nouvelle version du métamodèle du réseau de Petri rend les modèles existants invalides. Ainsi, nous illustrerons dans ce document, les modifications faites sur le métamodèle du réseau de Petri MM1 et leurs impacts sur les modèles qui lui sont conformes.

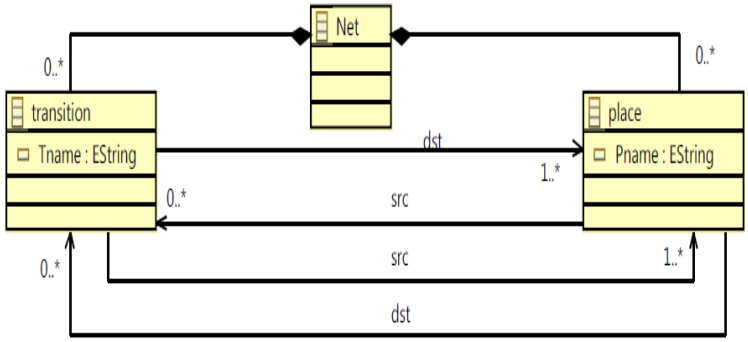

Figure 2.2.a. Métamodèle initial du réseau de Petri (MM1) (Wachsmuth, 2007).

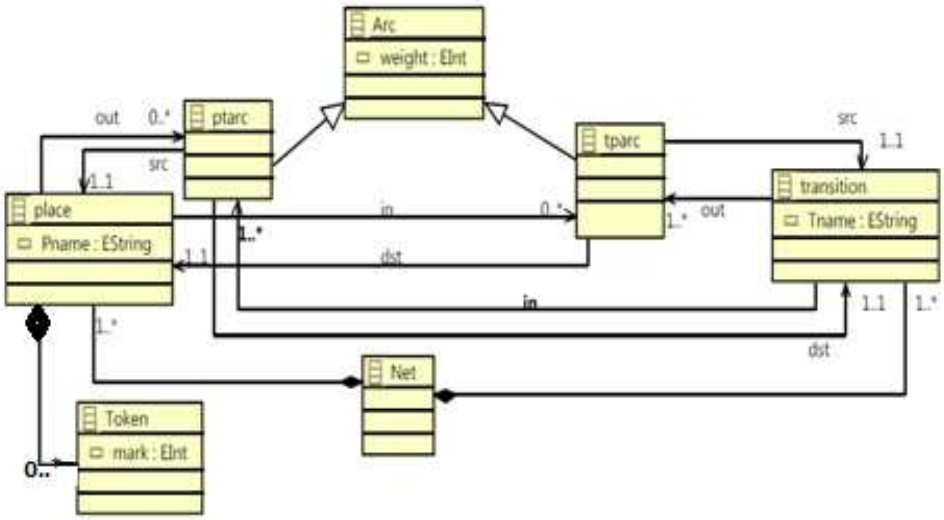

Figure 2.2.b. Métamodèle évolué du réseau de Petri (MM2) (Wachsmuth, 2007).

Version1	Version2
Modifications	
Multiplicité composition (place,net) 0..*	Multiplicité composition (place,net) 1..*
Multiplicité composition (transition,net) 0..*	Multiplicité composition (transition,net) 1..*
Ajouts	
ϕ	Classe Tparc
ϕ	Classe Tparc
ϕ	Classe Arc
ϕ	Classe Token
ϕ	Propriété weight (Arc)
ϕ	Propriété mark (Token)
ϕ	Superclasse (Arc →Tparc)
ϕ	Superclasse (Arc →ptarc)
ϕ	Association in (0..*) place →tparc
ϕ	Association dst (1..1) tparc → place
ϕ	Association src (1..1) tparc → transition
ϕ	Association out (1..*) transition →tparc
ϕ	Association in (0..*) place →ptarc
ϕ	Association src (1..1) ptarc → place
ϕ	Association dst (1..1) ptarc → transition
ϕ	Association in (1..*) transition →ptarc
ϕ	composition (0..*) place →token
Suppressions	
Association src (0..*) place →transition	ϕ
Association dst (1..*) transition → place	ϕ
Association src (1..*) transition → place	ϕ
Association dst (0..*) place →transition	ϕ

Tableau 2.1. Récapitulatif des changements entre les versions MM1 et MM2 du métamodèle représentant les réseaux de Petri.

2.3. Classification des changements d'un métamodèle

Les métamodèles peuvent changer de différentes manières. L'évolution d'un métamodèle d'une ancienne à une nouvelle version est décrite par un ensemble de changements. Plusieurs travaux dans le domaine de l'évolution et la coévolution ont proposé des classifications différentes des changements effectués sur un métamodèle. Dans (Wachsmuth, 2007), l'auteur propose une classification des opérations de changement selon leur impact sur un métamodèle. Ces impacts sont de trois types : les opérations qui préservent la sémantique (refactoring) ; les opérations qui augmentent la taille du métamodèle (construction) et les opérations qui réduisent la taille du métamodèle (destruction). Par ailleurs dans (Cicchetti et al, 2008) La classification des changements proposée distingue entre les changements additifs faisant référence aux additions des éléments dans un métamodèle ; les changements soustractifs qui consistent en la suppression de quelques éléments du métamodèle et les

changements modificatifs représentant quelques modifications et mises à jour des éléments déjà existants dans le métamodèle. Une autre classification est celle proposée dans (Herrmannsdoerfer et al, 2008a) où les changements sont classés en changements primitifs structurels qui affectent la structure du métamodèle ; les changements primitifs non structurels qui préservent la structure du métamodèle tout en changeant uniquement les éléments existants et les changements composites qui regroupent une séquence de changements primitifs.

Ces classifications considèrent l'effet des changements sur les métamodèles. Une autre classification est proposée dans (Gruschko et al, 2008). Cette classification projette les changements sur le niveau M1 en se basant sur l'impact des changements sur les modèles. En fait, Il y a des changements qui ne cassent pas les modèles et par conséquent ne nécessitent aucune ou une légère révision de ces modèles. D'autres changements introduisent des incompatibilités et des incohérences entre les versions, donc invalident (ou cassent) la conformité des modèles. Pour adresser cette issue les instances M1 du métamodèle changé doivent être migrées à la nouvelle version du métamodèle. L'effort nécessaire pour performer cette migration dépend de la nature des changements effectués sur le métamodèle. Elle peut se faire automatiquement ou nécessite une intervention manuelle.

Dans ce travail nous avons effectué une synthèse des changements des métamodèles en s'appuyant sur plusieurs travaux (Wachsmuth, 2007), (Cicchetti et al, 2008), (Herrmannsdoerfer et al, 2008a), (Gruschko et al, 2008). Nous présentons dans ce qui suit un ensemble de ces changements.

- ***Ajout d'une méta-classe « Create class » :*** L'introduction des nouvelles classes est un changement fréquent dans les métamodèles, c'est un moyen de leurs extensions. L'ajout des nouvelles méta-classes pose un problème de coévolution seulement si les nouveaux éléments sont obligatoires en respectant les cardinalités spécifiées. Dans ce cas des nouvelles instances de la classe ajoutée doivent être intégrées dans le modèle.
- ***Ajout d'une méta-propriété « Add Property ,Create Attribute, Create Reference » :*** Ceci est similaire au cas précédent selon que la propriété et obligatoire ou non selon la cardinalité spécifiée. Les modèles existants maintiennent leur conformité au métamodèle considéré si l'addition se fait dans des méta-classes abstraites sans sous classes ou si les propriétés ne sont pas obligatoires. Si la méta-propriété est un attribut obligatoire et son initialisation se fait par une valeur par défaut ou elle est dérivée d'autres propriétés l'adaptation des modèles est automatique. Dans les autres cas, l'intervention humaine est demandée pour spécifier la valeur de la

propriété.

- ***Supprimer une méta-classe « Delete class » :*** une méta-classe est éliminée. Généralement ce changement implique la suppression de toutes les instances de cette classe (les objets) au niveau du modèle M1. En plus, si la méta-classe possède des sous-classes ou référencée par d'autres méta-classes, la suppression affectera aussi les entités en relation. La suppression des classes peut causer des liens à des objets non existants. Ainsi, les classes peuvent être supprimées sans impacts sur les modèles quand elles sont en dehors de toute hiérarchie et elles ne sont pas la cible ni par des références non-composites ni par des références composites obligatoires.
- ***Supprimer une méta-propriété (Delete property):*** Une propriété est supprimée d'une méta-classe. Ce changement est similaire au changement précédent. La propriété est soit un attribut ou une référence. La suppression d'une référence implique la suppression des liens au niveau des modèles M1. Les références peuvent être supprimées uniquement si elles ne sont pas composites et n'ont pas des opposites.
- ***Création et suppression des références opposites « Create/Delete Opposite »***: La création et la suppression des références qui ont un opposite sont différentes des autres opérations de création et de suppression. « *Create opposite reference »* limite l'ensemble des liens valides donc il produit une perte des informations. Par contre, « *Delete opposite reference »* élimine une contrainte du modèle et donc il sera préservé.
- ***Création et suppression des énumérations et des types de données « Create/Delete Data Type , Create/Delete Enumeration » :*** Les énumérations et les types de données peuvent être supprimés uniquement quand ils ne sont plus utilisés dans le métamodèle. Supprimer des énumérations et des types de données préservent les modèles.
- **Création et suppression des littéraux « Create /Merge literal »** : M*erge literal* supprime un littérale dans une énumération au niveau métamodèle, l'adaptation au niveau modèle consiste à remplacer les occurrences de ce littérale par un autre littérale.
- ***Généralisation d'une méta-propriété « Generalize property, Generalize Attribute, Generalize Reference » :*** une méta propriété est généralisée quant sa multiplicité ou son type est élargi. La généralisation exige que l'ancienne multiplicité soit incluse dans la nouvelle. Par exemple si une cardinalité (3..n) d'une méta-classe simple est modifiée à (0..n) aucune action d'adaptation n'est requise.
- ***Specialiser une méta-propriété « Restrict property, Specialize Property, Specialize***

Attribute, Specialize Reference » : une propriété est restreinte ou limitée quand sa multiplicité ou son type sont réduits. Restreindre la borne supérieure d'une multiplicité nécessite la sélection de certaines valeurs à supprimer. Incrémenter la borne inférieure nécessite l'ajout de certaines valeurs. Généralement ces valeurs sont fournies manuellement. Restreindre le type d'une propriété nécessite une conversion de type pour chaque valeur.

- ***Renommer un méta-élément « Rename » :*** c'est un changement simple qui nécessite d'être propagé aux instances existantes et peut être exécuté de manière automatique.
- ***Déplacer une méta-propriété « Move property » :*** consiste à déplacer une propriété d'une méta-classe A à une méta-classe B. Ce changement est résoluble et les modèles existants peuvent coévoluer en déplaçant la propriété p de toutes les méta-classes A aux instances de la méta-classe B.
- ***Remonter une méta-propriété « Pull up property »*** : une propriété (attribut ou référence) est remontée dans une super classe et l'ancienne est supprimée de la sous classe (figure 2.3, 2.4).

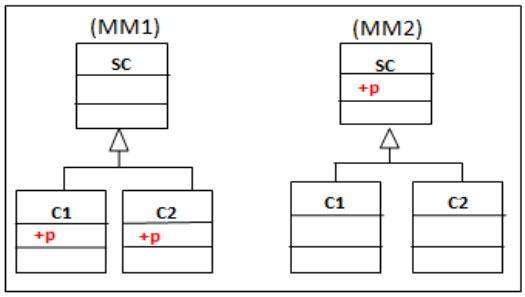

Figure 2.3. Opération du changement Pull Up meta Property (Attribut).

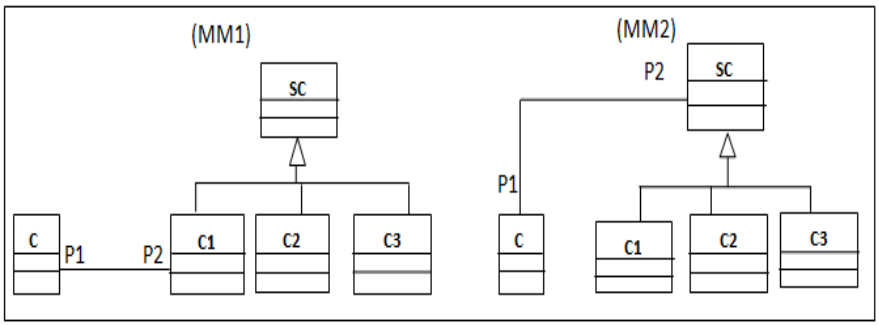

Figure 2.4. Opération du Changement « Pull Up meta Property » (Référence).

- ***Pousser une méta-propriété « Push-down property » :*** pousser une propriété dans une méta-classe signifie que cette propriété est supprimée d'une super classe initiale A et après elle copie dans toutes les sous-classes c de la classe A (figure 2.5). si A est une classe abstraite alors ce genre de changement ne nécessite aucune adaptation

des modèles, autrement toutes les instances de A et ses sous classes doivent être modifiées en conséquences.

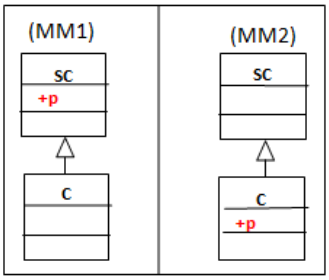

Figure 2.5. Opération du changement **«** Push down property**».**

- ***Extraire une super-class « Extract superclass » :*** une super classe est extraite dans une hiérarchie (figure 2.6) et un ensemble de propriétés est remonté. Si la classe est abstraite les instances d'un modèle seront préservées. Sinon l'effet est celui de « Pull up property ».

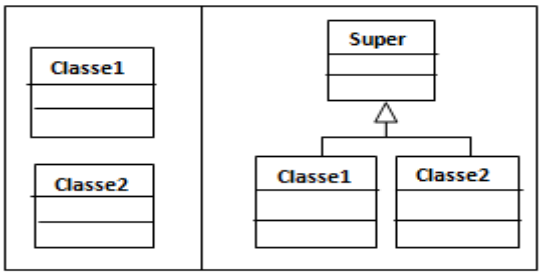

Figure 2.6 Opération du Changement « Extract super class ».

- ***Extraire une méta-classe*** **« Extract class »** : Extraire une méta-classe signifie la création d'une nouvelle classe, déplacer les champs pertinents de l'ancienne classe à la nouvelle classe et la relier à l'ancienne méta-classe par une association dont les deux extrémités doivent avoir la cardinalité (1-1). Cette opération est utilisée en conjonction avec l'opération Move Property pour diminuer le nombre des propriétés d'une méta-classe.
- ***Inline une méta-classe*** **« Inline class »** : c'est l'opération inverse de Extract class. Elle consiste à supprimer une méta-classe liée à une autre par une association dont les deux extrémités sont de cardinalité (1-1). Cette opération signifie le déplacement de tous ses attributs dans une autre classe et la supprimer. Ces deux modifications induisent des coévolutions automatiques des modèles.
- ***Changer une association en une classe « AssociationToClass, ReferenceToClass » :*** L'opération *AssociationToClass* est une opération de modification, qui consiste à extraire une méta-classe à partir d'une association entre deux méta-classes déjà existantes (figure 2.7). Les deux propriétés déjà existantes p1 et p2 qui faisaient

référence respectivement à C2 et C1 vont faire référence à C3. Les nouvelles propriétés p3, p4 qui sont contenues dans C3 feront référence respectivement à C1 et C2. Elles doivent être de cardinalité (1-1).

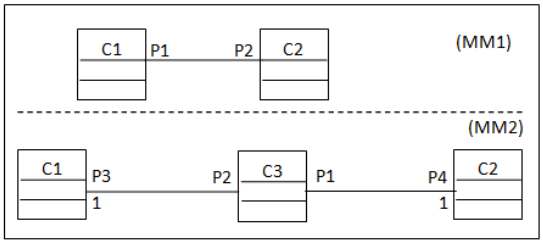

Figure 2.7. Exemple d'application du changement AssociationToClass.

- ***Changer une classe en une association (ClassToAssociation, ClassToReference) :*** c'est l'opération inverse de l'opération AssociationToClass, elle supprime une méta-classe qui comporte une référence p3 de cardinalité (1-1) vers une méta-classe C1, et une autre référence p4 de cardinalité (1-1) vers une autre méta-classe C2. Cette méta-classe est remplacée par une association entre les méta-classes C1 et C2.
- ***Changer l'héritage en une composition (InheritanceToComposition)*** **:** cette opération consiste à convertir une relation d'héritage entre deux méta-classes en relation de composition. La sous méta-classe devient la classe conteneur de la super-méta-classe (figure 2.8). La cardinalité de l'extrémité d'association à côté de la classe contenue (ancienne super-méta-classe) est de (1-1).

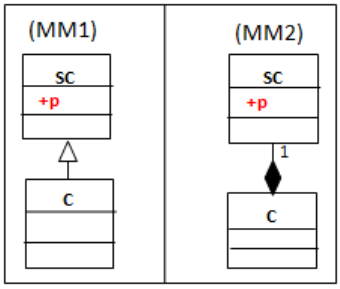

Figure 2.8. Opération du changement « InheritanceToComposition ».

Un récapitulatif des changements est présenté dans le tableau (2.2). Nous spécifions pour chaque changement son impact sur les modèles instance à savoir :

- non cassant (NC) : dans le cas ou le changement n'affecte pas la conformité des modèles instances et par conséquent aucune action de migration n'est nécessaire.
- cassant et automatiquement résoluble (CR) : dans le cas ou la conformité des modèles est affectée, mais une action de migration peut être automatiquement appliquée pour résoudre le problème.
- cassant non résoluble (CNR) : dans le cas ou la conformité des modèles est affectée, mais une action de migration ne peut pas être complètement

automatique.

Nous ajoutons aussi quelques travaux dans le domaine de l'évolution des métamodèles qui ont utilisé ce changement (Becker et al, 2007 ; Waschmuth, 2007 ; Cicchetti et al, 2008 ; Herrmandoerfer et al, 2010).

Changement		Impact	(Becker et al, 2007)	(Whachsmuth, 2007)	(Cicchetti et al, 2008)	(Herrmandoerfer et al, 2010)
Changements primitifs	Create Package Créer un package	NC	x			X
	Delete Package Supprimer un package	NC	x			X
	Create class Créer une classe	NC	x	x	x	X
	Delete class Supprimer une classe	CNR	x	x	x	X
	Create attribute Créer un attribut	NC	x	x	x	X
	Create reference Créer une référence	NC	x	x	x	X
	Delete property Supprimer une propriété	NC	x	x	x	X
	Create opposite reference Créer une référence opposite	NC	x	x	x	X
	Delete opposite reference Supprimer une référence opposite	NC	x			X
	Create data type Créer un type de données	NC	x			X
	Delete data type Supprimer un type de données	NC	x	x	x	X
	Create enumeration Créer une énumération	NC	x			X
	Delete enumeration Supprimer une énumération	NC	x			X
	Create literal Créer un littérale	NC	x			X
	Merge literal Fusionner un littérale	CNR	x			X
	Rename Renommer	CR	x	x	x	X
	Make class abstract Rendre la classe abstraite	CNR	x			
	Drop class abstract Rendre la classe non abstraite	NC	x			X
	Add super-type Ajouter un super- type	NC	x			X
	Remove super-type Enlever le super-type	CNR	x			X

	Make attribute Identifier Rendre l'attribut identificateur	CNR	x			X
	Drop attribute Identifier Rendre l'attribut non identificateur	NC	x			X
	Make reference composite Rendre la référence composite	CNR	x			X
	Switch reference composite Rendre la référence non composite	CR	x			X
	Make reference opposite Rendre la référence opposite	CNR	x			X
	Drop reference opposite Rendre la référence opposite	NC	x			X
Changements composites	Generalize Attribute Généraliser un attribut	NC	x	X	x	X
	Specialize Attribute Spécialiser un attribut	CNR	x	X	x	X
	Generalize Reference Généraliser une référence	NC	x	X	x	X
	Specialize Reference Spécialiser une référence	CNR	x	X	x	X
	Specialize Composite reference Spécialiser une référence composite	CNR				X
	Generalize super-type Généraliser un super-type	CNR	x			X
	Specialize super-type Spécialiser un super-type	CR	x			X
	Pull-up property Remonter une propriété	NC		X	x	X
	Push down property Pousser une propriété	CNR		X	x	X
	Extract super- class Extraire une super- classe	NC		x	x	X
	Inline super-class	CNR		x	x	X
	Fold super-class Plier une super-classe	NC				X
	Unfold super-class Déplier une super-classe	CNR				X
	Extract sub-class Extraire une sous-classe	NC				X
	Inline Sub-class Enfoncer une sous-classe	CNR				X
	Extract class Extraire une classe	CR		x	x	X
	Inline class Enfoncer une classe	CR		x	x	X
	Fold class Plier une classe	CR				X
	Unfold class Déplier une classe	CR				X
	Move feature over reference Déplacer une propriété sur une référence	CR		x	x	X
	Collect feature over reference Rassembler propriété sur une référence	CNR				X

	Subclasses to enumeration Sous- classes en énumération	NC				X
	Enumeration to subclasses Enumération en Sous- classes	NC				X
	Reference to class Référence en classe	NC		x		X
	Class to reference Classe en référence	NC		x		X
	Inheritance to delegation Héritage en composition	NC				X
	Delegation to inheritance Composition en héritage	NC				X
	Reference to identifier Référence en identificateur	NC				X
	Identifier to reference Identificateur en référence	CNR				X

Tableau 2.2. Synthèse des changements d'un métamodèle.

2.4. Evaluation des approches existantes

Les langages logiciels existent dans différents espaces techniques (Herrmannsdorfer, 2010), exemple : les langages de programmation, les langages de modélisation, les formats XML et les modèles de données. Tous ces espaces sont affectés par l'évolution. Dans la littérature, différentes approches ont été proposées pour la gestion de l'évolution des langages. Nous nous focalisons dans cette thèse sur l'étude de l'évolution des langages de modélisation. Nous résumons les approches proposées dans les domaines connexes et nous éclaircissons les relations qui existent entre elles en mettant en évidences les similarités et les différences des approches des différents espaces techniques.

La figure (2.9) compare les niveaux d'abstraction des espaces techniques étudiés à ceux de l'architecture MDA et des langages de modélisation. Comme décrit sur cette figure, nous avons une architecture plus proche des schémas XML. Si l'on compare les schémas de bases données aux langages de modélisation, le schéma contient dans une seule spécification le métamodèle utilisé par les bases de données (définition des concepts et de la structure) et le modèle décrivant une base de données, l'instanciation de ce dernier est la création de base de données réelle.

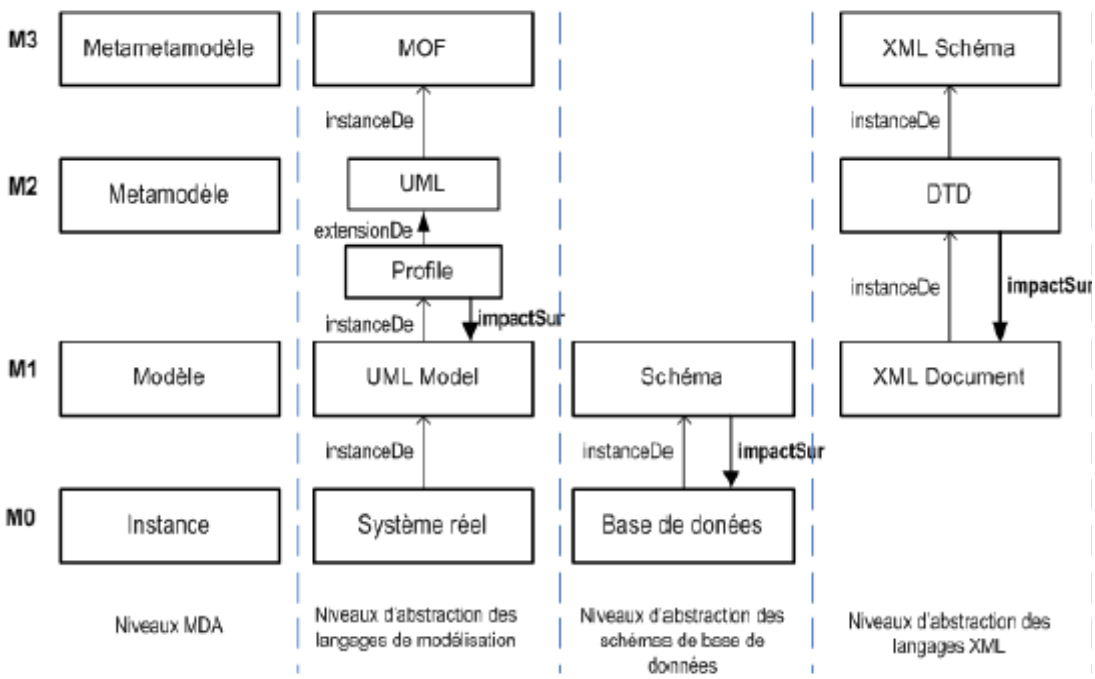

Figure 2.9. Comparaison des niveaux d'abstraction dans des espaces techniques différents (Herrmannsdorfer, 2010)

2.4.1. Evaluation des approches dans les domaines connexes

L'évolution et la coévolution n'est pas restreinte aux évolutions des modèles. Plusieurs travaux ont été menés pour la gestion des évolutions dans des domaines connexes, tel que les bases de données, les schémas XML, les grammaires…etc. Nous nous intéressons aux domaines les plus proches de l'IDM plus particulièrement la gestion de l'évolution dans le domaine des schémas de base de données et des schémas XML.

2.4.1.1. Evolution des schémas des bases de données

Une base de données représente des données du monde réel en rapport avec un thème ou une activité. Un schéma de base de données ou modèle de données décrit la structure de la base de données. Nous distinguons les schémas relationnels et les schémas orientés objets.

Pour les bases de données orientées objets, ORION (Kim et al, 1987) est un système de base de données qui fournit un support à l'évolution dynamique des schémas. La structure d'évolution se base sur un ensemble de propriétés du schéma appelées invariants qui doivent être préservées pendant l'évolution et un ensemble de règles qui servent à la sélection de la méthode la plus adéquate pour préserver les invariants. Les invariants guident la définition de la sémantique de chaque changement significatif du schéma, en

assurant que le changement ne conduit pas le schéma à un état d'inconsistance. Le système d'évolution d'ORIEN est basé sur un catalogue fixe d'opérateurs primitifs. Ce catalogue est suffisamment expressif pour assurer l'évolution de tout schéma. Mais, certaines restrictions dues aux invariants font que quelques évolutions ne sont pas permises dans ces systèmes. Comme par exemple, la redirection d'un lien d'héritage à un Super-type de la hiérarchie n'est pas autorisée dans le système ORION. Les changements complexes ne sont également pas pris en compte dans cette approche, le concepteur du schéma doit soit décomposer un changement complexe en une combinaison d'ajouts et de suppressions, soit développer les programmes de conversion pour traiter ces changements. Une approche similaire à ORIEN est GemStone (Prenney et al, 1987). Dans GemStone, un ensemble d'invariants plus étendu que celle d'ORIEN est défini. Les invariants sont plus simples à cause de la structure de classes qui est sous forme d'arbre et contrairement à ORIEN qui nécessite douze règles pour faire l'évolution des schémas, dans ce système ces règles ne sont pas indispensables. GemStone propose à la différence d'ORION de conserver et de réutiliser les spécifications des changements complexes en étendant directement le moteur d'adaptation de son système. SERF (Claypool et al, 1999) considère que les opérations définies par ORION sont simples et ne permettent pas de définir toutes les évolutions possibles d'un schéma. SERF aborde l'évolution des relations et considère également qu'il est impossible de décrire à l'aide d'opérateurs d'évolution définis de manière statique toutes les évolutions possibles d'un schéma. Dans ce cas, le système propose de prendre en compte ce critère en combinant un langage de requête ainsi que les opérations d'évolution de base pour créer des transformations de schéma et ainsi passer d'une version à une autre de manière automatisée. Les transformations de SERF sont généralisées par des patrons de transformations qui sont réutilisables et paramétrées. Les instances de ces patrons sont des transformations appliquées sur le schéma. Les informations décrites traitent de l'invocation des opérations de base à impliquer durant la transformation. TransformGen (Garlan et al, 1994) est un système dédié au support de l'évolution de la syntaxe abstraite utilisée dans les arbres syntaxiques de bases de données. TransformGen reprend les opérations d'évolution définies par ORION et GemStone et automatise l'adaptation des schémas lors de leur application. TransformGen va au delà de ces systèmes en proposant un langage pour la construction d'opérations complexes. Il propose au concepteur de définir les opérateurs complexes à l'aide de ce langage qui servent alors d'entrée à une transformation. Cette dernière est chargée de définir, de valider et de générer les solutions d'adaptation correspondantes. OTGen (Object Transformer Generator) (Lerner, 1990) est une extension de TransformGen. OTGen enregistre les changements réalisés sur un schéma

en modifiant la spécification d'une transformation à partir de laquelle les solutions d'adaptation sont générées. Pour chaque changement simple appliqué sur le schéma.

O2 est une autre approche (Brèche, 1996), (Ferrandina et al, 1996). Son objectif est d'automatiser au maximum l'application des opérations d'évolution et l'adaptation des instances du schéma. La spécification des solutions peut être faite par le concepteur comme par le système. Si ce dernier ne désire pas les spécifier. O2 génère des solutions de base minimales en traitant chaque changement de manière indépendante avec des valeurs par défaut. Afin de garantir la consistance entre le schéma et la base de données, dans O2 une fonction de migration par défaut est associée à chaque classe modifiée. Cette fonction peut être modifiée en attachant une fonction de migration personnalisée spécifiée manuellement. O2 propose deux stratégies d'adaptation : directe ou différée. Dans le premier cas, l'ensemble des instances est immédiatement adapté après une évolution. Dans le second, les instances des modèles sont adaptées au fur et à mesure de leur utilisation. Le concepteur choisit quelle partie de la base de données doit être adaptée à la nouvelle version du schéma.

Les évolutions dirigées par les versions (Skarra et al, 1986 ; Clamen, 1992 ; Tresch et al, 1992) consistent à établir plusieurs versions d'un schéma de base de données. Il existera autant de versions qu'il existe de nouvelles versions du schéma. Le but est qu'un élément défini dans une version puisse être accessible via une autre version du schéma. L'avantage est qu'il est inutile d'adapter les bases de données aux nouvelles versions du schéma. Les bases de données sont toujours valides, les différentes versions sont toujours disponibles et accessibles. En plus de conserver un historique des évolutions, ceci permet de garder utilisables les anciennes versions des programmes. Cependant, cette approche peut avoir un coût important sur l'espace et le temps, mais nécessite également une gestion d'accès non négligeable vers les éléments des différentes versions. Or, cette mise en place doit être prise en charge par le concepteur qui doit fournir les procédures d'accès vers les différentes versions du schéma.

Le mécanisme de vues (Rundensteiner, 1992) est très utilisé dans le domaine des schémas de bases données pour offrir des préoccupations différentes en fonction des utilisateurs. La gestion de l'évolution des schémas de base de données dirigée par les vues consiste à créer autant de vues que d'évolutions réalisées. Une vue est assimilable à une version du schéma. Pour chaque évolution faite sur une vue particulière, celle-ci est dérivée par la création d'une nouvelle vue, qui devient indépendante de la précédente. Aucune vue n'est supprimée, les instances des schémas restent toujours valides par rapport à une vue et peuvent être améliorées pour correspondre à une autre vue dérivée. Le traitement

d'évolution est donc minimisé, il s'agit uniquement de créer une nouvelle vue.

2.4.1.2. Evolution dans des schémas XML

Un schéma XML est une définition d'un langage ou un format. Une instance d'un langage est appelé un document. Les schémas sont exprimés par des langages de schéma tels que DTD (W3C, 2008) ou XML Schema (Walmsley, 2001), les deux recommandés par « World Wide Web Consortium ». Nous distinguons dans ce domaine, les approches manuelles et les approches à base d'opérateurs.

Les approches manuelles nécessitent que l'utilisateur spécifie manuellement la migration des documents d'une version du schéma à une autre. Tan et Goh proposent une extension du schéma XML pour spécifier de manière déclarative les différences des anciennes versions directement dans le schéma (Tan and Goh, 2005). Les additions, les suppressions, les déplacements des éléments ainsi que le changement des noms sont supportés. L'information est ensuite utilisée pour la migration des documents entre différentes versions du schéma.

Quant à eux les approches à base d'opérateurs fournissent un ensemble de couples d'opérations réutilisable. L'utilisateur effectue l'évolution du schéma XML de manière impérative par l'application d'une séquence d'opérations. Comme les opérations travaillent au niveau schéma et document. Une telle séquence spécifie à la fois l'évolution du schéma et la migration du document

XEM (XML Evolution Manager) aborde l'évolution des schémas DTD (Su et al, 2001). Elle fournit une suite complète, forte et minimale d'opérations primitives. Au niveau schéma, ces opérations travaillent sur les schémas DTD représentés par des graphes. Au niveau document, elles opèrent sur des arbres étiquetés et ordonnés.

Lämmel and Lohmann propose des opérations de transformation pour les schémas DTD à partir desquelles des migrations de documents sont déduites (Lämmel et Lohmann, 2001). Les opérations sont décrites par un texte informel au niveau schéma. Tandis que les migrations sont spécifiées par XSLT. Les opérations préservent la bonne forme des schémas DTD et les documents XML. En plus, Les opérations sont classées selon si elles préservent, étendent ou réduisent la structure des documents XML.

2.4.2. Evolution des langages de modélisation

L'évolution dans le domaine de l'IDM est compliquée car l'IDM implique la combinaison de plusieurs Types d'artefacts interdépendants (Deursen et al, 2007). Plus spécifiquement, il y a trois types d'artefacts de développement spécifiques à IDM (modèles, métamodèles et la

spécification des opérations de gestion des modèles). Le changement d'un type d'artefact peut affecter d'autres artefacts (éventuellement de types différents) (Rose, 2011).

Les classifications actuelles des approches d'évolution et de coévolution se concentrent soit sur le type d'informations considérées au cours de l'évolution, soit sur la technique utilisée pour la stratégie de migration. Quelques travaux classent les types d'évolution du modèle en deux catégories: évolution du modèle syntaxique et sémantique (Sprinkle et Karsai, 2004).

- Évolution du modèle syntaxique: Cette méthode modifie les modèles du domaine existants de sorte que les modèles obéissent aux règles syntaxiques du nouveau métamodèle. Dans ce cas, aucune information n'est connue sur l'intention du changement évolutif. Un inconvénient de l'évolution syntaxique est que les nouvelles données ne reflètent pas nécessairement la sémantique voulue de l'ancien domaine. Cependant, la migration syntaxique peut être entièrement automatisable.
- L'évolution du modèle sémantique est une transformation ou un ensemble de transformations qui réécrivent un modèle pour avoir le même sens dans le nouveau langage qu'il avait dans son langage original. La migration sémantique nécessite une adaptation manuelle. Le manque d'informations sur la sémantique de l'évolution peut réduire la possibilité d'automatiser la propagation des changements. Par exemple, si on considère le cas où une classe est supprimée d'un métamodèle. Les questions suivantes doivent être posées pour faciliter l'évolution :
- Les sous-types de la classe supprimée doivent-ils également être supprimés? Sinon, leur hiérarchie d'héritage doit elle changée ?
- Est-ce que les instances d'une classe supprimée d'un métamodèle devraient être supprimées également où leurs données devraient être déplacées vers une autre classe ?

Une deuxième classification des approches pour automatiser la migration des modèles en réponse à l'évolution des métamodèles est proposée dans (Rose et al, 2009). Les auteurs identifient trois catégories des approches : les spécifications manuelles, les approches à base d'états (matching des métamodèles) et les approches à base d'opérateurs.

2.4.2.1. Les approches manuelles

Dans la spécification manuelle, la stratégie de migration est encodée manuellement par le concepteur du métamodèle, en utilisant un langage de programmation général (par exemple Java) ou un langage de transformation modèle-à-modèle (tel que QVT (MOF, 2005), ou ATL (Jouault, 2005). Certaines approches fournissent des langages de

transformation de modèles personnalisés pour spécifier manuellement la migration des modèles. Ainsi, des constructeurs de migration spécifiques sont fournis pour réduire l'effort de spécification de la migration. Par exemple les éléments du modèle dont la définition du métamodèle correspondante n'a pas changé sont automatiquement copiés dans le nouveau modèle. L'utilisateur ensuite remplacera ce comportement défini par défaut par la migration prévue. Les spécifications manuelles sont essentiellement l'approche de Sprinkle (Sprinkle, 2003), l'approche MCL [Narayanan et al, 2009] et l'approche Flock (Rose et al, 2009).

a) *Un langage visuel spécifique au domaine (DSVL) pour l'évolution des modèles du domaine :*

L'approche de Sprinkle, définit un Langage visuel spécifique au domaine (DSVL) basé sur GME (Sprinkle , 2003 ; Sprinkle et Karsai, 2004). Il fournit une interface spécialisée pour décrire un algorithme permettant de transformer des modèles de domaine d'un DSVL à un autre. La migration des modèles de domaine est effectuée en utilisant des patrons syntaxiques qui sont comparés avec les concepts des modèles du domaine évolués à l'aide de règles de mappage. Ces règles sont de la forme "patron implique conséquence". Un ensemble fondamental d'opérations telles que «Créer», «Créer avec", " Devient " et " Supprimer ". Les différences entre deux versions de métamodèles sont spécifiées de manière déclarative. Dans cette approche, si l'utilisateur est familier avec les concepts de méta modélisation, c'est avec très peu de guidance qu'il peut créer une transformation d'évolution du domaine qui va servir à évoluer les modèles du domaine dans le DSVL évolué (Sprinkle et Karsai, 2004). Le point fort de ce langage c'est la possibilité offerte pour créer les patrons syntaxique sous n'importe quelle forme, puisque le langage des patrons est dérivé du langage de méta_modélisation. Toutefois, l'efficacité de l'algorithme de transformation conçu dépend de la capacité du concepteur. Nous notons également que le langage n'est pas en mesure de vérifier si les modèles du domaine transformés sont correctes (Sprinkle, 2003).

b) *MCL : un langage de changement de modèles*

Dans (Levendovszky et al, 2004 ; Narayanan et al, 2009) les auteurs proposent MCL (Model change language) pour la gestion de l'évolution des métamodèles conformant au MOF. MCL est un langage visuel de haut niveau. Comme pour le DSVL de Sprinkle (Sprinkle, 2003), avec MCL l'utilisateur n'a pas à spécifier uniquement les différences entre les métamodèles mais aussi il définit la migration des modèles basée sur ces différences. MCL

définit un ensemble de règles (« idioms ») pour mettre en relation des concepts correspondants entre deux versions d'un même métamodèle, ainsi qu'une approche de composition pour spécifier la migration. Ces règles sont censés couvrir les cas de migration les plus communs (ex. ajout d'un nouveau concept, modification d'un élément, suppression d'un élément, ajout de nouveaux sous-types). MCL utilise des patrons basiques pour des scénarios de migration typique qui consiste en un élément de l'ancien métamodèle (LHS), un élément du nouveau métamodèle (RHS) et une relation « MapsTo » entre les deux. Un autre lien spécifique dans les patrons est appelé «WasMappedTo », utilisé avec un nœud qui a été migré auparavant avec une règle de migration antérieure.

Une relation de correspondance comme décrit sur la figure (2.10) se décompose en deux parties : "Condition de correspondance" et "Commande additionnelle". La première partie permet de faire correspondre deux concepts ou groupes de concepts grâce à la validation de pré et de post conditions. La seconde partie de la règle permet de spécifier une solution d'adaptation spécifique à cette règle et qui sera exécutée sur les modèles instances. Les algorithmes de migration qui ne sont pas couvert par MCL peuvent être spécifiés impérativement en utilisant des API C++. Par exemple, le changement des noms des éléments se traduit en MCL par une règle « MapsTo » entre ces deux éléments. La solution d'adaptation associée spécifie que le nom des instances de l'ancien élément source est à modifier par le nom du nouvel élément cible. Le nom de la relation appliquée entre deux concepts dépend du type des éléments à faire correspondre. Les concepts seront reliés par une relation « MapsTo », les attributs d'une méta-classe sont reliés par la relation « AttrToAttr », les évolutions des associations et des liens d'héritage utilisent également la relation « MapsTo » au travers des méta-classes auxquelles ils sont attachés. Les auteurs définissent trois manières d'utiliser une relation de correspondance qui représentent trois opérations d'évolution distinctes :

- Ajout de concept : Lors de l'ajout d'un nouveau concept, celui-ci est lié à un élément déjà existant, la représentation en MCL consiste à mettre en relation l'élément auquel il est relié et son ancienne version, puis à spécifier la correspondance.
- Suppression d'un concept. La suppression d'un élément utilisé par d'autres éléments du métamodèle n'est pas autorisée. MCL exprime l'opération de suppression par la relation « MapsTo » ou « AttrToAttr » entre le concept de l'ancienne version et un élément NULL dans la nouvelle version. La solution d'adaptation qui complète cette règle spécifie que toutes les instances de l'élément doivent être supprimées.
- Modification d'un concept. Une correspondance entre les deux versions d'un élément ou d'un groupe d'éléments décrit comment les instances de ces éléments

doivent être exprimées.

A la différence de l'approche de Sprinkle qui fournit une interface de transformation générale (Sprinkle 2003 ; Sprinkle et Karsai, 2004). MCL fournit un langage de modélisation spécifique au domaine (DSML) comme langage de spécification ce qui le rend plus efficace. En plus, MCL fournit une syntaxe graphique puissante et une sémantique simple, MCL est modulaire, expressive, et permet la réutilisation des connaissances de migration. Il permet également de spécifier des relations complexes entre les méta-entités. Mais en MCL quelques règles doivent être résolues manuellement et il y a des cas qui dépendent de l'intention du développeur de la transformation.

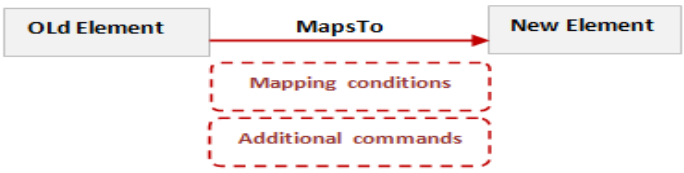

Figure 2.10. Représentation de la règle « MAPS To » (Narayanan et al, 2009)

c) Migration de modèles avec Epsilon Flock

Flock est un langage spécifique de domaine pour spécifier et exécuter les stratégies de migration. Flock est un langage de migration textuel pour les modèles basés sur EMF (Rose et al, 2010). Flock possède une syntaxe compacte. Une grande partie de sa conception et son implémentation est basée sur des «runtime ». Flock automatiquement fait correspondre chaque élément du modèle original à un élément équivalent dans le modèle migré en utilisant un nouveau algorithme de conservation de copie ainsi que des règles de migration définies par l'utilisateur. Ici, seule la migration du modèle est spécifiée. Flock copie automatiquement les éléments du modèle qui sont toujours conformes au métamodèle évolué au lieu d'expliciter les différences entre les versions du métamodèle. L'utilisateur par la suite redéfinit itérativement la spécification de la migration afin d'adapter les éléments non conformes. Les stratégies de migration sont organisées en modules qui comportent n'importe quel nombre de règles. Le contrôle de conformité est à la charge d'EMC. L'approche prend sa puissance de la couche de connectivité d'Epsilon qui permet à Flock d'utiliser des modèles représentés en MDR, XMI, et CZT et l'extension de Flock est capable pour supporter d'autres technologies de modélisation. Par contre la stratégie de migration devient plus difficile pour des métamodèles plus larges puisque il n'y a pas d'outil qui supporte l'analyse des changements entre la version originale et évoluée.

2.4.2.2 Les approches à base d'états

Ces approches reposent sur le matching des métamodèles. Les différences entre deux versions du métamodèle sont calculées et stockées de manière déclarative. A partir desquelles la spécification de la migration sera générée. Ces approches utilisent une des deux catégories de l'historique du métamodèle, soit le métamodèle original (approches de calcul des différences) ou les changements effectués sur le métamodèle original pour produire le métamodèle évolué (approches d'enregistrement des différences). Dans cette catégorie nous trouvons plusieurs approches comme l'approche de Gruschko (Gruschko et al, 2007), l'approche de Cicchetti (Cicchetti et al, 2008) et l'approche de Garcés (Garcès et al, 2009a).

a) Vers la synchronisation des modèles avec les métamodèles évolués : L'approche de Gruschco

Cette approche utilise une transformation de modèle à modèle. Elle supporte la détection automatique des changements simple dans les métamodèles EMF/Ecore (Gruschko, 2006 ; Gruschko et al, 2007 ; Becker et al, 2007). Ils proposent des étapes de migration automatiques pour les changements résolubles, et envisage de supporter l'utilisateur pour spécifier la migration concernant les changements non résolubles. L'approche se divise en cinq étapes 1) détection des changements ou traces, 2) classification, 3) entrées utilisateur, 4) détermination des algorithmes et 5) l'étape de migration. Ces cinq étapes sont successives, les résultats de chacune servent d'entrée à l'étape suivante (figure 2.11).

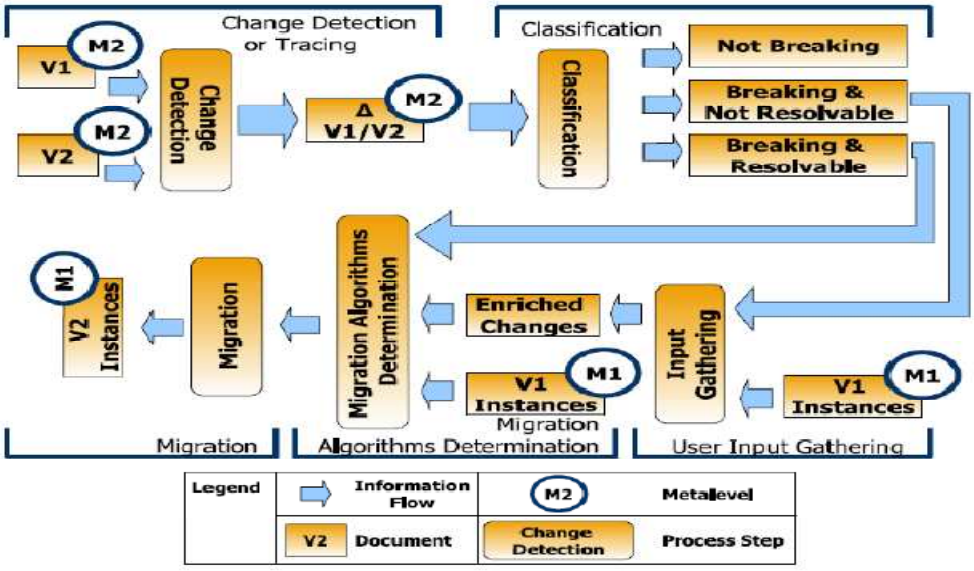

Figure 2.11 : Description de l'approche de synchronisation des modèles (Gruschko et al, 2007)

- Etape détection des changements ou traces. Cette étape a pour but de déterminer les différences unitaires entre deux versions d'un métamodèle EMF. Pour ce faire les auteurs utilisent deux méthodes automatiques : l'enregistrement des différences et la méthode de comparaison de deux versions d'un métamodèle. Le résultat de cette étape est l'obtention des différences de modèle "delta" contenant l'ensemble des différences identifiées. Les auteurs ne proposent pas l'opérationnalisation de ces différences. C'est-à-dire que ces différences ne sont pas représentées comme une opération issue d'un métamétamodèle et que l'on peut appliquer à n'importe quel métamodèle
- Etape classification : Les auteurs proposent une classification en trois catégories des différences contenues dans le modèle Delta : les changements non cassants (Non breaking changes), dont l'impact fait que les modèles instances restent des instances de la nouvelle version du métamodèle. Les changements cassants mais pouvant être résolus (breaking and resolvable), dont l'impact fait que les modèles instances ne sont pas des instances du métamodèle évolué mais la migration des modèles vers une version conforme à la nouvelle version du métamodèle est complètement automatisable. les changements cassants et ne pouvant être résolus (breaking and non resolvable), ils se différencient du cas précèdent par le fait que la migration n'est pas possible automatiquement et qu'elle nécessite l'intervention du concepteur des modèles.
- Etape entrée utilisateurs : durant cette étape, l'utilisateur choisit les modèles qu'il désire adapter. C'est également durant cette étape qu'il prend connaissance des changements non résolubles qui nécessitent son intervention.
- Etape détermination d'algorithme : Cette étape prépare l'opération de migration des modèles. C'est durant cette phase que se construit l'algorithme d'adaptation.
- Etape Migration. La transformation est exécutée sur les modèles à adapter.

Cette approche propose une solution d'adaptation pour cinq types d'évolutions possibles. Les solutions des autres différences sont spécifiées manuellement par l'utilisateur de l'approche (concepteur des modèles ou du métamodèle) en utilisant le langage Epsilon (Rose et al, 2010). L'approche est implémentée par un prototype mais elle n'est pas évaluée. L'approche minimise l'effort manuel nécessaire pour réaliser la migration du modèle. Mais, les changements sont supposés se produisant individuellement et l'utilisation des relations au lieu des modèles de différence ne permet pas de distinguer la modification des méta-éléments des patrons suppression/ajout.

b)- Une approche transformationnelle pour la coévolution des modèles

Cette approche est une approche de coadaptation donnée comme une transformation de modèle de haut niveau (higher-order model transformation). Cette transformation prend le modèle de différence enregistrant l'évolution du métamodèle et génère une transformation capable de produire la coévolution des modèles (Cicchetti et al, 2008). Dans cette approche le modèle de différence consiste en des changements simples interprétés en termes de changements complexes. La spécification de la migration consiste en un ensemble de transformation de modèles à exécuter consécutivement en tenant compte des dépendances caractérisant les changements complexes (Cicchetti et al, 2009). L'approche comporte les étapes suivantes : initialement le modèle de différence est automatiquement décomposé en deux sous-modèles disjoints qui dénotent les changements cassants et résolubles et non résolubles. Si ces sous modèles sont parallèles et indépendants alors les coévolutions correspondantes sont générées séparément, par contre si ces deux sous-modèles sont dépendants ils seront raffinés pour identifier et isoler les interdépendances. Notons que cette isolation n'est pas toujours possible.

Cette approche ne spécifie pas explicitement comment calculer le modèle de différence, seulement il peut être obtenu en utilisant un outil tel que EMFCompare ou SiDiff. Elle se concentre plutôt sur la représentation et la définition des différences entre deux versions du métamodèle. L'objectif est de s'abstraire des techniques de détection des différences qui imposent le format des métamodèles d'entrée et ont une représentation des différences spécifiques à un outil. Les auteurs ici proposent une méthode de représentation de différences qui soit indépendante du format du métamodèle d'entrée (Cicchetti et al, 2008). Les auteurs proposent ainsi la succession de deux transformations. La première transformation (transformation MM2MDD sur la figure (2.12) permet d'identifier les différences. Elle consiste à définir un métamodèle de différence à partir d'un métamodèle source. Le métamodèle de différence est construit en spécialisant chaque méta-classe de la nouvelle version du métamodèle en trois sous méta-classes préfixées : Add MC, Delete MC ou Change MC où MC sont les noms des méta-classes. Ce métamodèle Delta est utilisé pour spécifier des modèles de différences. A chaque nouvelle version du métamodèle source, la transformation est ainsi réappliquée et un nouveau métamodèle Delta est crée et enregistré (versionning). Avant de pouvoir exécuter des solutions d'adaptation sur l'ancienne version des modèles les auteurs proposent de classifier les méta-classes du métamodèle Delta en trois catégories : les opérations dont l'impact est nul, celles dont l'impact peut être résolu automatiquement et celles dont l'impact est non résoluble. Le

résultat de la classification est la division en deux catégories du métamodèle Delta (MM « resoluble » et MM « unresoluble »). La dernière étape de l'approche consiste à reprendre les modèles conformes à l'ancienne version du métamodèle et à renseigner manuellement dans un modèle de différences, les éléments impactés par les évolutions. Un modèle de différence correspond à une instance du métamodèle Delta, il instancie les opérations du métamodèle Delta directement sur les éléments impactés par l'évolution. Les solutions d'adaptation sont alors associées à ces opérateurs d'évolution grâce à la seconde transformation.

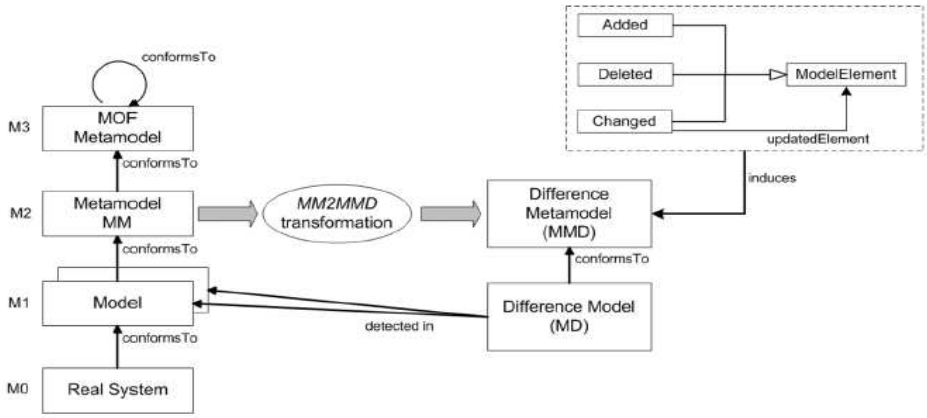

Figure 2.12. Utilisation d'un métamodèle de différence (Cicchetti et al, 2008).

Les opérations complexes et leurs solutions sont également crées durant cette étape. Pour les évolutions non cassantes, aucune solution d'adaptation n'est spécifiée.

c)- Gestion de l'adaptation des modèles par la détection précise des changements du métamodèle

Cette approche consiste en trois étapes pour l'adaptation des modèles à leurs métamodèles évolués (Garcès et al, 2009a). Initialement un processus de mise en correspondance (matching) est utilisé pour calculer automatiquement les équivalences et les différences entre deux versions du métamodèle en appliquant des heuristiques de manière incrémentale. Le résultat est stocké dans un modèle de correspondance. Cette approche propose AML (Atlas Matching Language) qui permet à l'utilisateur la paramétrisation de la détection des changements complexes (Garcès et al, 2009b). Par conséquent, l'utilisateur combine des heuristiques existantes ou définies par l'utilisateur pour définir un algorithme de correspondance. Ensuite, à partir du modèle de différence obtenu par un tel algorithme, une transformation spécifiant la migration est automatiquement générée par une transformation de haut niveau, la transformation produite est écrite en un langage particulier de transformation (ex . ATL, XSLT, SQL-like). Cette

transformation préserve les éléments du modèle qui n'ont pas changés. Finalement, la transformation de l'adaptation est exécutée. L'approche est évaluée sur l'exemple du réseau de Petri (Wachsmuth, 2007), ainsi que sur le métamodèle Java. Les auteurs ont démontré que leur approche proposée donne des résultats exacts pour la détection des changements simples et complexes ainsi que les stratégies de mise en correspondance ont des bonnes performances. Nous trouvons cette approche puissante, puisque elle permet le calcul de différences et de correspondances entre chaque paire de métamodèle. En plus les heuristiques de mise en correspondance s'exécutent de façon modulaire et à la demande. L'approche est générique, les heuristiques sont décrites en termes des concepts de KM3 et peuvent être implémentées par d'autres formalismes tels que MOF ou EMF-Ecore. Mais l'assistance de l'utilisateur est nécessaire dans quelques stratégies et une combinaison sémantique invalides des heuristiques peut causer une erreur d'exécution, cependant une combinaison incorrecte engendre la génération d'une transformation de migration incorrecte. L'utilisation des heuristiques est également ambigüe.

2.4.2.3 Les approches à base d'opérateurs

Les approches à base d'opérateurs spécifient l'évolution des métamodèles par l'application d'une séquence d'opérateurs. Un opérateur peut être couplé à la stratégie de migration correspondante du modèle. Ces approches fournissent une bibliothèque d'opérateurs de coévolution couplés et réutilisable qui s'exécutent sur les deux niveaux métamodèle et modèle. En composant les opérateurs, l'évolution du métamodèle peut être réalisée et la stratégie de migration peut être générée sans l'écriture d'aucun code. Les approches significatives dans cette catégorie sont celle de Wachsmuth (Wachsmuth , 2007) et l'approche Cope (Herrmansdoerfer et al, 2008).

a) adaptation des métamodèles et coadaptation des modèles

L'approche de Wachsmuth est une approche transformationnelle qui assiste l'évolution du métamodèle par des adaptations incrémentale présente une suite d'opérations pour le formalisme de méta-modélisation MOF (Wachsmuth, 2007) basée sur les idées d'évolution des grammaires (Lammel, 2001), Les opérations sont classées selon leur sémantique et leurs propriétés de préservation des modèles et des langages. Pour la migration, la spécification de l'évolution est traduite en transformation de modèle exprimée dans le langage QVT. Cette approche est caractérisée par la possibilité de réutiliser les scripts d'adaptation dans des scénarios d'adaptation similaires. Cependant cette approche est limitée à cause de l'atomicité des changements, ce qui n'est pas réaliste.

b)- COPE- automatisation de l'évolution couplée des métamodèles et des modèles

Cette approche enregistre l'évolution sous forme d'une séquence d'opérations couplées dans un modèle d'historique explicite.

La gestion de l'évolution selon la l'approche COPE (Hermmandoerfer et al, 2008) repose sur les quatre principes suivants (Figure 2.13):

- Les évolutions d'un métamodèle et des modèles instances doivent être réalisées conjointement. Cela garantirait une adaptation plus sécurisée des modèles en évitant notamment des oublis d'éléments à adapter ou des erreurs d'incompréhension des évolutions. Ainsi, COPE crée une liaison entre le métamodèle qui évolue et les modèles à adapter. Pour cela, COPE enregistre chaque modification du métamodèle et lui associe l'opération d'adaptation qui lui correspond. L'association entre ces deux opérations forme ce qui est appelé un couple d'opérations. Des couples d'opérations atomiques sont prédéfinis et proposés dans la librairie de COPE.
- Regrouper l'ensemble des couples d'opérations crées entre deux versions d'un métamodèle dans un document nommé "Migrator". L'adaptation consiste à l'exécution séquentielle des couples d'opérations crées sur l'ancienne version des modèles instances.
- Offrir la possibilité au concepteur (du modèle ou métamodèle) de créer des couples d'opérations personnalisés. En effet, le concepteur peut enrichir un couple d'opérations ou créer un complexe et ainsi gérer une évolution plus large.

Dans cette approche, les auteurs ont proposé une classification des évolutions d'un métamodèle en quatre catégories :

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 doit évoluer.

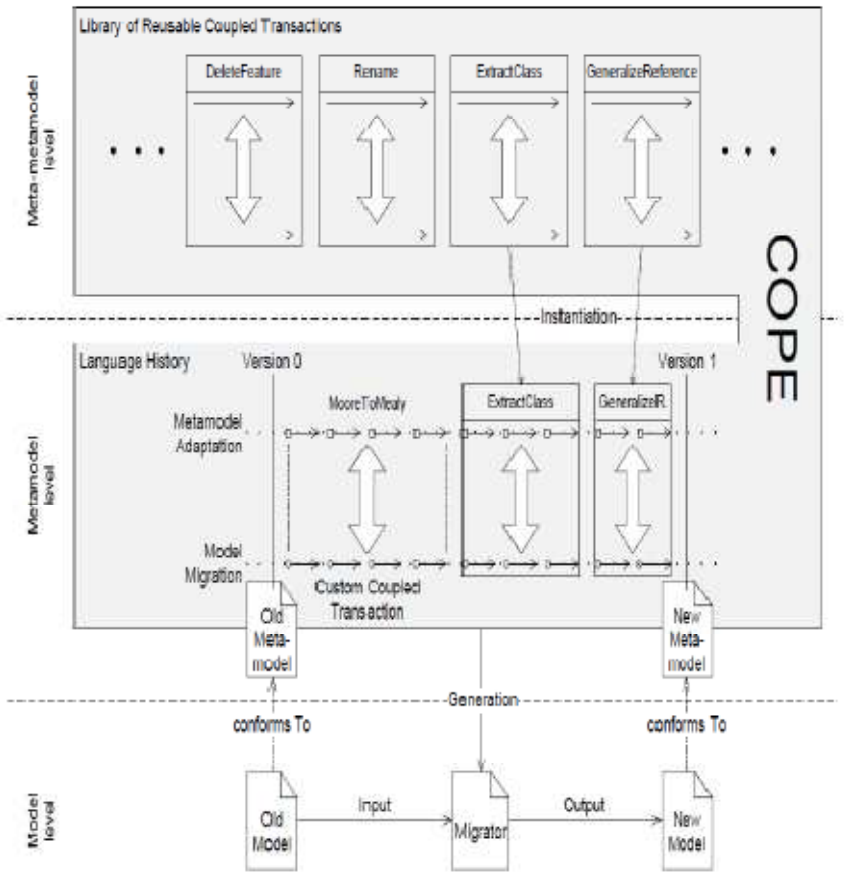

Figure 2.13. Description de l'approche COPE (Hermmandoerfer et al, 2008).

L'approche COPE facilite l'analyse du métamodèle, et offre un haut degré de réutilisation des opérations couplées récurrentes dans la migration des modèles. Elle utilise aussi une large librairie des opérateurs de coévolution. Cependant déterminer la séquence des opérations qui produit une migration correcte n'est pas possible en particulier pour les métamodèles volumineux. La migration personnalisée dans COPE est spécifiée dans un langage de programmation général.

2.4.3. Analyse comparative des approches existantes

Après cette description des approches existantes dans le domaine de la coévolution des modèles et des métamodèles. Une analyse comparative de ces approches est effectuée en tenant compte de quelques critères généraux (Anguel et al, 2015). Ces critères de comparaison sont les suivants :

- Spécification de l'évolution : Soit l'évolution est spécifiée implicitement en fournissant la version originale et évoluée du métamodèle, comme elle peut être spécifiée explicitement selon deux styles (Herrmmansdoerfer, 2010) : les spécifications impératives de l'évolution sont spécifiées par une séquence des opérations de changement, au contraire la spécification déclarative modélise

l'évolution par un ensemble de différences entre deux versions du métamodèle.

- Source de l'évolution : Quand la spécification de l'évolution est explicite, elle peut avoir différentes sources. L'évolution peut être définie par l'utilisateur lorsque ce dernier spécifie manuellement l'évolution. Une autre manière c'est l'enregistrement de l'évolution pendant que l'utilisateur édite les changements du métamodèle. Une importante source c'est la détection automatique de l'évolution, deux types de détection sont à distinguer : détection des changements simples seulement comme l'addition et la suppression et la détection des changements complexes, par exemple l'extraction des constructeurs.
- La cible de la migration : La migration peut être réalisée soit sur le modèle originale lui-même (in-place) en le modifiant, soit un nouveau modèle sera crée pour supporter la migration et le modèle originale sera préservé.
- Langage de migration : La migration peut être définie comme un langage de migration spécifique au domaine, comme il est possible de réutiliser un langage de transformation (TL) existant, une autre manière est d'ajouter un support de migration à un langage général (GPL) comme une API.
- Extensibilité de migration : Ce critère définit si les extensions sont supportées. Il y a trois types d'extensibilité pouvant être supportés. La migration peut être fixe et définie totalement par le développeur et seulement lui qui peut ajouter des nouvelles parties dans la stratégie de migration. Un autre type de migration permet à l'utilisateur de réécrire et de personnaliser la stratégie de migration. Un troisième type où la migration est extensible, dans ce cas l'utilisateur peut ajouter complètement des nouvelles parties dans la stratégie de migration.

Le tableau suivant liste les approches présentées dans les sections précédentes et montre leur comportement vis-à- vis des critères étudiés.

Approche	**Evolution**		**Migration**		
	Spécification	**Source**	**Target**	**Langage**	**Extensibilité**
Sprinkle	Déclarative	Défini-utilisateur	Out	personnalisé	Réécriture
Narayanan	Déclarative	Défini-utilisateur	Out	Personnalisé	Réécriture
Rose	Implicite	-	Out	Personnalisé	Réécriture
Gruschko	Déclarative	Détecté-simple	Out	TL/ETL	Réécriture
Cicchetti	Déclarative	Détecté-complex	Out	TL/ATL	Fixed
Garcès	Impérative	Détecté-complex	Out	TL/ATL	Extensible
Wachsmuth	Impérative	Défini-utilisateur	Out	TL/QVT	Fixed
Herrmansdoerfer	Impérative	Enregistré	In	TL/Groovy	Extensible

Tableau 2.3. Comparaison des approches de coévolution.

De façon générale, nous discernons que les approches manuelles fournissent des langages de transformation de modèles personnalisés pour spécifier la migration manuellement. Les éléments qui n'ont pas changé sont copiés automatiquement. Par la suite, l'utilisateur réécrit le comportement par défaut de la migration. Dans ces approches les différences ne sont pas explicites. Les approches manuelles ne fournissent pas des constructeurs pour la réutilisation des connaissances de migration à travers les métamodèles (Herrmansdoerfer, 2010). La caractéristique essentielle de ces approches c'est le langage de migration personnalisé pour la réécriture manuelle de la migration ce qui augmente l'expressivité de ces approches. Ainsi, les approches manuelles favorisent l'exactitude de la migration des modèles, mais en contre partie exigent un effort plus important.

Les approches à base d'états détectent automatiquement les différences entre deux versions du métamodèle. Elles sont stockées de manière déclarative dans un modèle de différence à partir duquel la spécification de la migration est générée. Ainsi, la caractéristique essentielle de ces approches c'est la spécification déclarative de l'évolution qui est soit enregistrée ou détectée. Ce qui permet d'augmenter l'automaticité en générant automatiquement la stratégie de migration. Ces approches essayent d'automatiser la totalité du processus, ceci ne mène pas toujours à un modèle correcte. Les approches de Gruschko (Gruschko et al, 2007 ; Cicchetti et al, 2008) ne traitent que les opérations atomiques car elles se basent sur des technologies automatiques de détection de différences mais ils fournissent un guide pour le traitement des évolutions complexes non automatisées et il n'y a pas encore une approche mûre dans cette catégorie (Herrmannsdorfer et Wachsmuth, 2014).

Les approches à base d'opérateurs d'évolution fournissent un ensemble d'opérateurs d'évolution couplés qui opère sur le niveau métamodèle ainsi que le niveau modèle. La caractéristique de ces approches c'est la spécification impérative de l'évolution comme une séquence d'opérations. Ces caractéristiques favorisent la réutilisation qui est un moyen pour réduire l'effort. En fait, ces approches conduisent à une migration correcte en enregistrant les traces des changements durant l'évolution du métamodèle. Néanmoins, elles exigent l'intégration du métamodèle et des modèles dans le même éditeur. L'approche COPE propose un ensemble d'opérateurs atomiques riches et compréhensibles. Nous pouvons nous en inspirer pour définir des opérateurs d'évolution.

Conclusion

Le présent chapitre passe en revue les travaux existants sur l'évolution du logiciel. En particulier, ce chapitre explore les méthodes dont l'évolution du logiciel est identifiée et

gérée dans le contexte de l'IDM, que nous avons discuté dans le chapitre 1. Nous avons présentés également la gestion de l'évolution dans un ensemble de domaines connexes, y compris les bases de données, les schémas XML. Par la suite, une analyse comparative des approches examinées est effectuée afin de mettre en évidence les besoins à satisfaire par une approche de coévolution dans le contexte de L'IDM.

Chapitre 3

Encodage des modèles en langage des prédicats

CHAPITRE 3
ENCODAGE DES MODÈLES EN LANGAGE DES PREDICATS

Introduction

Nous proposons dans ce chapitre une méthode pour la formalisation des modèles dans un langage formel basé sur la logique des prédicats. L'objectif étant d'utiliser un mécanisme de raisonnement intelligent pour gérer les connaissances relatives à l'évolution des métamodèles ainsi que l'étude de l'impact sur les modèles instances.

3.1. Encodage des modèles

Les modèles conformes au MOF sont transformés ou encodés dans un formalisme basé sur un langage formel. On obtient ainsi un modèle formel. Une distinction est réalisée entre encodage et transformation du modèle vers le modèle formel : l'encodage peut rendre compte de toutes les constructions des modèles MOF alors qu'une transformation est la source de perte d'informations (Malgouyers, 2006).

3.1.1. Encodage

La différence entre encodage et transformation vient du niveau auquel sont définies les relations entre modèle MOF et modèle formel. Comme le représente la figure (3.1), l'encodage est défini au niveau méta-métamodèle (ou méta-langage). Nous verrons plus tard que pour la transformation, cette définition est effectuée au niveau métamodèle (ou langage).

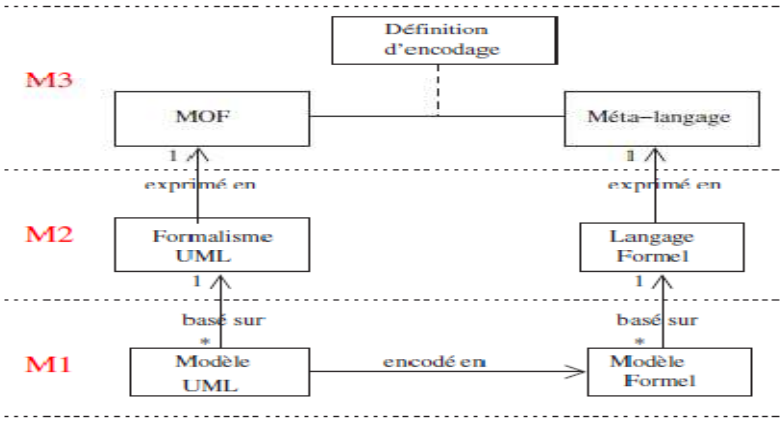

Figure 3.1. Encodage des modèles (Malgouyres, 2006).

L'encodage le plus courant des modèles UML normalisé par l'OMG est celui qui consiste à représenter un modèle UML en XMI (XML Metadata Interchange). D'après l'OMG, "XMI provides a mapping from MOF to XML". La définition du MOF étant réalisée au niveau M3, cette correspondance l'est aussi. Ceci permet d'encoder tout modèle basé sur un langage exprimé en MOF par un document XML. Concrètement, cette correspondance est définie par des règles EBNF (Extended Backus-Naur Form) permettant de générer des schémas XML pour tout métamodèle exprimé en MOF (niveau M2). Au niveau M1, XMI définit la génération de document XML qui encode le modèle considéré. Notons que (OMG, 2005) spécifie explicitement que retranscrire l'ensemble des informations d'un modèle est primordial : "All significant aspects of the metadata are included in the XML document and can be recovered from it. No information is lost".

3.1.2. Transformation

Au contraire la définition d'une transformation est définie au niveau métamodèle (figure 3.2). Le principe des transformations consiste en effet à définir des relations entre les concepts de deux langages pour pouvoir passer de l'un à l'autre.

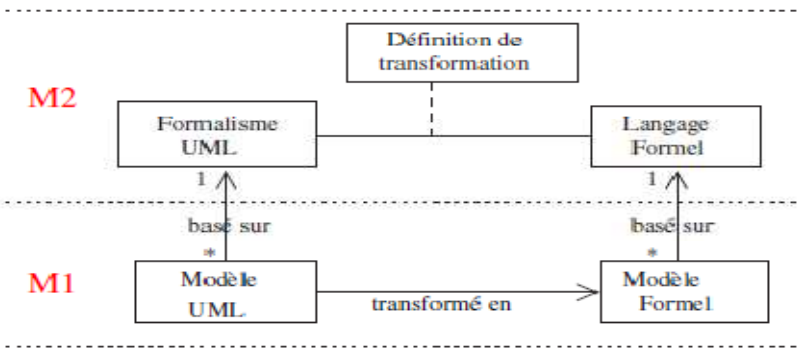

Figure 3.2. Transformation des modèles (Malgouyres, 2006).

3.2. Introduction à la programmation logique

Nos travaux ont pour but d'assurer la coévolution des modèles dans leur intégralité. Nous choisissons donc de mettre en œuvre la technique de l'encodage plutôt que celle de la transformation. Cette technique est mise en œuvre par la formalisation des métamodèles MOF. De plus, nous désirons manipuler les opérateurs d'évolution. Le langage d'expression des opérateurs choisi doit donc pouvoir formaliser l'ensemble des opérateurs associés. Le choix du langage d'encodage des modèles et du langage d'expression des opérateurs a abouti à la sélection de LP (*Logic Programming*).

L'idée centrale de la programmation déclarative (dont fait partie la programmation logique), est que la programmation se résume à la partie logique du problème sans se

préoccuper de la partie contrôle. On ne s'intéresse donc pas à comment le problème est résolu (aspect procédural) mais à ce qu'est la solution (aspect déclaratif). Au contraire, en programmation classique le *quoi* (logique) n'est pas séparé du *comment* (contrôle) : un algorithme est constitué de composants logiques (qui représentent la connaissance vis-à-vis d'une application) et de structures de contrôle (qui déterminent comment cette connaissance est acquise). IL s'agit donc de définir des règles de logique mathématique au lieu de fournir une succession d'instructions que l'ordinateur exécuterait. Ces règles permettent d'expliciter le modèle d'inférence défini par un programme logique (Fages, 1996).

Un programme LP est un ensemble fini de clauses de la forme H ← B1,....,Bn où H,B1,....,Bn sont des atomes. Un atome est de la forme p(t1,...., tn) où p est un symbole de prédicat et ti sont des termes (Spivey, 1996).

3.2.1. Notion de terme

Soit V un ensemble des symboles appelés variables, C un ensemble de symboles appelés constantes et F un ensemble de symboles appelés fonctions. L'ensemble T des termes du premier ordre est défini inductivement comme le plus petit ensemble vérifiant les trois règles de fermeture suivante :

- $V \subset T$;
- $C \subset T$;
- *si f* symbole de fonction n-aire et *t*1,...., *tn* sont des termes alors $f(t1,...., tn) \in T$

Un symbole de prédicat est défini dans un programme s'il apparaît en tête d'une clause. Deux types de clauses sont à différencier : les faits sont des clauses pour lesquelles $n = 0$ et les règles sont des clauses telles que $n > 0$.

L'exécution d'un programme logique est la résolution d'un but de la forme *B*1,....,*Bn*.

le but vide ou but succès est noté T et le but échec est noté ⊥. Nous verrons par la suite quand est-ce qu'un but est en échec.

L'interprétation déclarative d'une clause H← B1,....,Bn stipule que H est vrai si B1,....,Bn sont vrais. La même clause de programme peut également être interprétée comme une règle de réécriture sur les buts : pour prouver H il suffit de prouver successivement les buts B1,....,Bn. Cette interprétation est appelée interprétation procédurale.

Afin d'expliciter ces deux interprétations d'un même programme logique, nous présentons l'interprétation déclarative au moyen d'un exemple et présentons ensuite le principe de

résolution SLD[2] qui est le fondement de l'interprétation procédurale.

3.2.2. Exemple

L'interprétation déclarative des programmes logiques est maintenant présentée à travers un exemple. Cet exemple s'appuie sur la syntaxe particulière des systèmes de programmation logique. Une variable commence par une majuscule (ou par un underscore ('_') si sa valeur est indifférente). Une constante commence par une minuscule. Nous présentons la syntaxe des faits, des règles et des buts ainsi que leur interprétation déclarative.

trans(a,b). (f1)
trans(a,c). (f2)
trans(b,d). (f3)
trans(c,d). (f4)
trans(d,e). (f5)

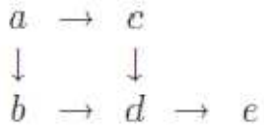

Figure 3.3. Exemple de graphe (acyclique) et de sa représentation en fait

Les faits sont la base de connaissance du système. Ils formalisent ce qui est connu. Considérons, les faits fournis dans la partie gauche de la figure (3.3). Ceux-ci représentent le graphe de la partie droite de la figure 3.3. Le fait (f1) signifie par exemple qu'il existe un arc entre les nœuds a et b.

Les règles représentent la base de déduction du système. Par exemple, les chemins directs et indirects entre deux nœuds sont donnés par les règles :

reach(Start,End) :- trans(Start,End). (R1)

reach(Start, End) :- trans(Start, Node), reach(Node, End). (R2)

Elles expriment qu'il est possible d'atteindre un nœud End à partir d'un nœud Start :

- s'il existe un arc (représenté par un fait trans) entre ces deux nœuds (règle R1) ;
- ou s'il existe un nœud intermédiaire « Node » tel qu'il existe un arc entre le nœud « Start » et le nœud « Node » et qu'il existe un chemin entre ce dernier et le nœud « End » (règle R2).

Au sein d'une règle, une virgule est équivalente au « *et* » logique (conjonction). L'expression d'une disjonction est réalisée par l'écriture de plusieurs règles.

Les buts sont une manière d'interroger la base de connaissance et de déduction représentée par un programme logique. La réponse fournie est une substitution qui s'applique aux variables du but telle que le but devient une conséquence logique de la base

[2] SLD signifiant Sélectionné, Linéaire, Défini

de connaissance et de déduction du programme. Par exemple, l'exécution du but reach(b,Y) permet de connaître les différents nœuds qui sont accessibles à partir de b. L'interpréteur LP donne donc successivement les réponses suivantes : Y=d et Y=e.

3.2.3. Principe de résolution SLD

La résolution- SLD est un algorithme servant à prouver une formule de logique du premier ordre à partir d'un ensemble de clauses de Horn[3]. Elle est basée sur une résolution linéaire, avec une fonction de sélection sur les clauses définies. La SLD-résolution est mieux connue par son extension, SLDNF (NF signifiant negation as failure, la négation par l'échec), qui est l'algorithme de résolution employé par le langage Prolog.

Nous présentons maintenant le principe de résolution SLD qui est le fondement de la sémantique opérationnelle des programmes logiques (Fages, 1996 ; Spivey 1996).

Un programme LP exécute un but *G* de la forme *B*1,....,*Bn*. L'état de résolution est un doublet < *G*,θ > où *G* est une conjonction de buts et θ une substitution. L'état initial est de la forme < *G*, ε > où ε est la substitution identité. Un état est un état final succès s'il est de la forme < *T*, θ >, c'est un état final en échec s'il est de la forme <$\perp$, ε >.

Le principe de résolution est constitué d'un ensemble de transitions élémentaires appelées « pas 0» d'inférence. Un pas d'inférence modifie l'état de la résolution à la suite de l'application d'une clause. Appliquer une clause H ← G consiste à:

- essayer d'unifier la tête H d'une clause du programme avec le sous-but Bi à réduire ;
- remplacer le sous-but *Bi* à réduire par le corps *G* de la clause sélectionnée, remarquons que lorsque la clause sélectionnée est un but, *G* est vide ;
- composer la substitution courante avec la substitution obtenue lors de l'unification de la tête de clause.

Ce principe est appelé principe de résolution SLD. Il est défini formellement de la manière suivante :

$$\text{SLD} : \frac{(H \leftarrow G)\ \sigma \in P \quad \theta \in U\ P(Bi, H)}{< (B1,....,Bi,....,Bn), \beta > \rightarrow < (B1,....,Bi\text{-}1, G, Bi+1,....,Bk), \theta\beta >}$$

où :

- (H ← G) est la clause du programme sélectionnée ;
- < (*B*1,....,*Bi*,....,*Bn*), β > est l'état de la résolution avant application de la transition ; (*B*1,....,*Bi*,....,*Bn*) est le but et β la substitution avant l'étape de résolution ;
- *Bi* est le sous-but à résoudre pour l'étape de résolution SLD ;

[3] Une clause de Horn est une clause comportant au plus un littéral positif.

- $\theta \in UP(Bi, H)$ signifie que θ est un unificateur principal de *Bi* et de *H* ;
- $< B1,....,Bi\text{-}1,G,Bi+1,....,Bk), \theta\beta >$ est l'état résolvant, i.e. le nouvel état à résoudre après application de la clause de programme sélectionnée sur le sous but *Bi* ;
- enfin, σ est une substitution de renommage destinée à éviter les conflits de variables entre la clause de programme et le but sélectionné.
- Une dérivation est un ensemble de ces pas SLD. Les dérivations peuvent amener à trois cas de figures: à un succès dans le cas où l'état final est l'état succès, c'est le cas lorsque le sous but est vide, une telle dérivation est notée $< G, \varepsilon > *\rightarrow <T, \theta >$;
- à un échec dans le cas où l'état final est l'état échec, c'est le cas lorsqu'un état est atteint avec un but non vide dans lequel un sous-but ne s'unifie avec aucune tête des clauses du programme logique ;
- à une dérivation infinie si aucun état final n'est atteint.

Conclusion

Le contexte de cette thèse concerne l'adaptation des modèles à l'évolution de leurs métamodèles. Le but est de fournir un moyen qui permet de :

- Détecter les changements qu'a subit un métamodèle par rapport à une version antérieure.
- Déduire les opérations d'évolution appliquées sur le métamodèle et explorer les possibilités de réorganisation de ces opérations en utilisant une méthode souple et évolutive, c'est-à-dire qui peut être adaptée en fonction du langage de modélisation considéré.
- Générer le scénario de migration pour l'adaptation des modèles instances.

Dans ce chapitre nous avons présenté le principe de la programmation logique (PL) qui est un des paradigmes de programmation de l'intelligence artificielle. La PL est utilisée dans ce travail pour l'encodage des modèles et des métamodèles avec leurs changements ainsi que pour la formalisation d'une bibliothèque d'opérateurs d'évolution. Ensuite nous avons décrit la méthode de résolution des problèmes adoptée par ce paradigme sur laquelle repose le principe de raisonnement utilisé pour la reconstruction du scénario d'évolution.

CHAPITRE 4
UNE APPROCHE INTELLIGENTE POUR LA GESTION DE LA COEVOLUTION DES MODÈLES

CHAPITRE 4
UNE APPROCHE INTELLIGENTE POUR LA GESTION DE LA COEVOLUTION DES MODÈLES

Introduction

Ce chapitre présente l'approche que nous proposons pour adapter les modèles à l'évolution de leurs métamodèles. Dans la première partie de ce chapitre, nous débutons par la présentation de l'approche en décrivant les quatre phases composant le processus de coévolution, à savoir la détection des différences, la reconstruction du scénario d'évolution et de migration ainsi que l'exécution de la migration. Puis nous abordons l'étape préliminaire qui est défini à priori et qui consiste à l'encodage des modèles et des changements ainsi que la formalisation d'une bibliothèque d'opérateurs atomiques et un ensemble d'opérateurs composites. Cet encodage supporte le processus de coévolution des modèles. La seconde partie du chapitre détaille la démarche que nous avons suivie pour la réalisation des différentes phases de notre approche. Ainsi, nous présentons la méthode pour la construction des opérations d'évolution à partir d'une liste de différences calculées entre deux versions du métamodèle. Cette étape se base sur l'ensemble des opérateurs atomiques et composites définis.

4.1. Positionnement de notre approche

Les approches de coévolution présentées précédemment montrent que l'impact des évolutions sur les modèles instances se base en grande majorité sur la classification proposée par (Gruschko et al, 2007). Cette classification est en effet la plus adaptée à notre étude, tout d'abord du fait qu'elle s'applique sur un modèle de différences qui peut être obtenu à posteriori et également car elle est la seule à considérer l'impact des évolutions sur les modèles et l'évaluation du niveau d'automatisation nécessaire pour adapter ces derniers à la nouvelle version du métamodèle. Malheureusement, des évolutions sont considérées comme non résolubles alors qu'avec une aide de la part du concepteur des modèles, la résolution de l'impact sur les modèles serait automatisable.

En fait, après une analyse détaillée effectuée sur les approches existantes (Anguel et al, 2015) et en se référant à d'autres travaux comparant ces approches (Rose et al, 2010 ; Rose

et al, 2012), nous avons identifié quelques besoins à satisfaire par une nouvelle approche à savoir :

- Augmenter l'automatisation de la coévolution des modèles et des métamodèles, ceci peut être possible par le principe de réutilisation des opérations comme dans les approches à base d'opérateurs. Ainsi, que le principe de copiage des éléments inchangés comme dans les approches manuelles et basées états (Herrmansdoerfer et al, 2014).
- Augmenter la clarté et la compréhension de la stratégie de migration en utilisant des langages de migration matures (Rose et al, 2012). Par exemple il est bénéfique d'utiliser EMF « Eclipse Modelling Framework » (Steinberg et al, 2009)
- Augmenter l'expressivité de la stratégie de migration en assurant des solutions dirigées par l'utilisateur d'un côté et de l'autre côté permettre l'extensibilité.

Ces besoins nous ont aidés à motiver notre approche. Ainsi, Dans ce travail de thèse, nous proposons une solution alternative au problème de la coévolution des modèles et des métamodèles où nous utilisons une mixture des techniques utilisées dans les approches à base d'états et à base d'opérateurs. Notre approche diffère des approches existantes dans les points suivants :

1- les approches existantes ne différencient pas le rôle des concepteurs de modèles (utilisateurs du métamodèle) et le concepteur du métamodèle. Ainsi, le concepteur de modèle est fortement impliqué dans le processus d'évolution, par ailleurs il n'est pas toujours connaisseur de la conception du métamodèle et il ne peut pas toujours comprendre l'intention du processus d'évolution. Notre approche est orientée concepteur de modèle (utilisateur) et vise à fournir aux utilisateurs un outil pour qu'ils puissent réaliser avec succès la migration de leurs modèles sans participer au processus d'évolution du métamodèle. L'objectif est de réduire au maximum les différentes interventions du concepteur de modèle et de le guider durant le processus de coévolution. C'est pourquoi, nous considérons que les changements du métamodèle sont calculés à posteriori comme dans les approches basées états.

2- Comme les approches à base d'opérateurs donnent des bons résultats (Herrmannsdorfer et Wachsmuth, 2014) et spécialement l'approche COPE avec son catalogue couvrant un large ensemble d'opérateurs (Herrmannsdorfer et al, 2011). Nous envisageons d'utiliser des opérateurs dans notre approche, mais l'approche COPE est orientée implémentation où les évolutions sont décrites par des codes de fonctions exécutables et les opérateurs n'ont pas une définition formelle et ne sont pas bien documenté. Au contraire, dans notre approche nous proposons une

formalisation de l'ensemble des opérateurs atomiques et composites afin de les utiliser de manière plus efficace et de faciliter l'éventuelle extension de ces opérateurs. La bibliothèque proposée fournit un haut niveau d'expressivité en ajoutant des structures logiques.

3- Les approches à base d'opérateurs étudiées dans le deuxième chapitre étaient descendantes. Les opérateurs d'évolution sont explicites et appliqués par le concepteur du métamodèle. Le système propose des opérateurs d'évolution dans une librairie, le concepteur sélectionne et applique un opérateur sur une version du langage. Une fois l'évolution complétée, le concepteur évalue alors l'impact des instances de ces opérateurs sur ses modèles et procède à leur adaptation. A l'inverse, nous voulons suivre une approche ascendante. Les opérateurs appliqués sont implicites et leurs instances doivent être découvertes en examinant le résultat d'une comparaison entre deux versions d'un métamodèle.

4- la gestion correcte de l'évolution nécessite la prise en compte des opérations d'évolution atomiques et composites. La majorité des approches existantes traitent uniquement les changements atomiques (Rose et al, 2009). Dans notre proposition l'originalité la plus significative c'est l'utilisation d'un mécanisme de raisonnement intelligent pour la reconstruction des opérations d'évolution composites afin d'obtenir un scénario d'évolution. Actuellement, à nos connaissances il n'ya aucune approche qui utilise un raisonnement intelligent pour résoudre le problème de coévolution sujet de notre étude.

4.2. Aperçu générale de l'approche proposée

Nous décrivons d'abord la démarche générale pour réaliser la coévolution des modèles avec leurs métamodèles qui ont évolué. La figure (4.1) illustre ce processus ainsi que les entrées et les sorties de chaque phase. Notre approche est hybride, elle importe des techniques des approches basées états et des approches basées opérateurs et en plus elle utilise un mécanisme de raisonnement de l'intelligence artificielle. L'approche se décompose en quatre phases:

- la détection des changements entre deux versions d'un métamodèle;
- la reconstruction et la validation du scénario d'évolution : cette phase consiste à reconstruire à partir d'une liste désordonnée de différences indépendantes, les opérations d'évolution conformes aux métamodèles MOF ;
- la détermination du scénario de migration : cette phase permet la spécification des

solutions d'adaptation pour une évolution selon son impact ;

- l'exécution de la migration du modèle.

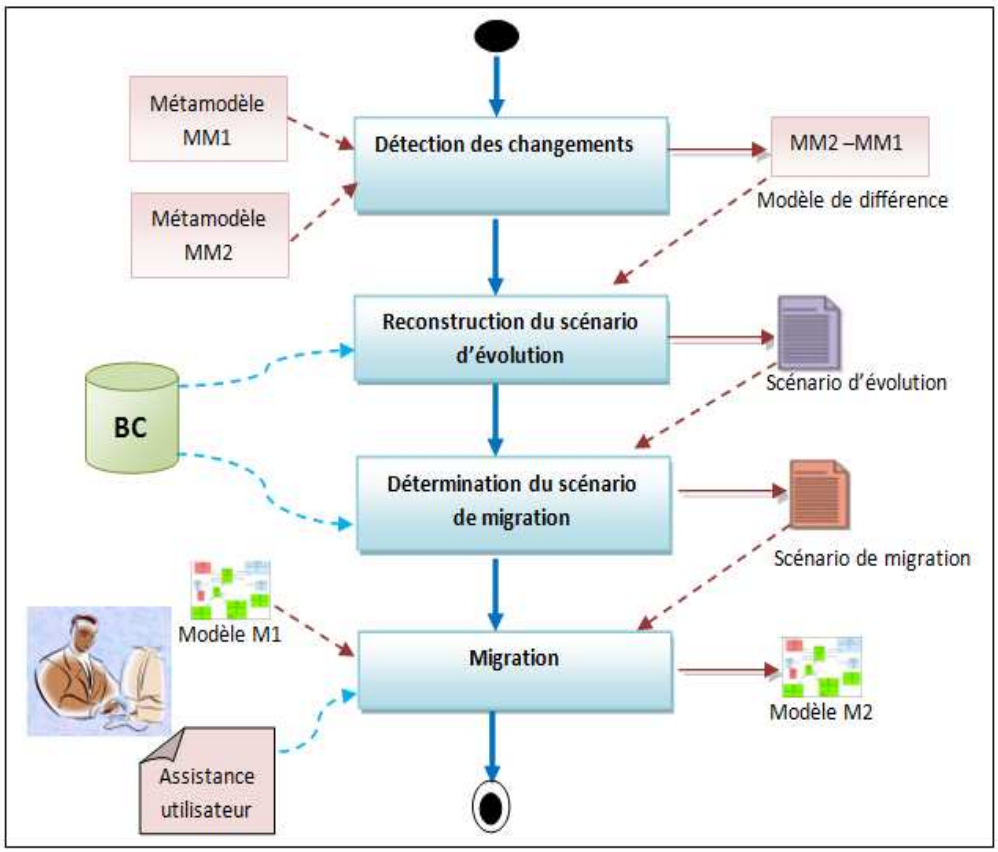

Figure 4.1. Aperçu général du processus de coévolution des modèles.

Dans la première étape; les différences entre deux versions du métamodèle (MM1, MM2) sont déterminées. Dans la deuxième étape nous reconstruisons les opérations d'évolution atomiques à partir du modèle de différence (MM2-MM1), nous utilisons ensuite un moteur d'inférence pour générer les scénarios d'évolution en assemblant les opérations atomiques en opérations composites.

La troisième étape consiste en l'exploration d'une bibliothèque d'opérateurs (i.e. base de connaissances "BC") pour obtenir les différentes procédures de migration qui seront assemblées dans un scénario de migration.

Dans la dernière étape, le scénario de migration sera appliqué sur un modèle instance spécifique (M1) qui est conforme à l'ancienne version du métamodèle (MM1) afin d'obtenir la nouvelle version du modèle (M2) qui soit conforme à la nouvelle version du métamodèle (MM2). Pendant cette étape les utilisateurs peuvent faire des choix entre les solutions alternatives proposées pour la migration de certains éléments.

Avec le processus de coévolution, nous définissons une phase préliminaire où nous procédons à l'encodage des métamodèles, des changements ainsi que la bibliothèque

d'opérateurs. Le résultat de cette phase étant une base de connaissances qui constitue le cœur de notre approche.

Dans les sections suivantes, nous présentons chaque phase du processus de coévolution en définissant tout d'abord les concepts indispensables pour faciliter sa compréhension. Puis nous décrivons cette phase en l'illustrant sur un exemple.

4.2.1. La phase préliminaire

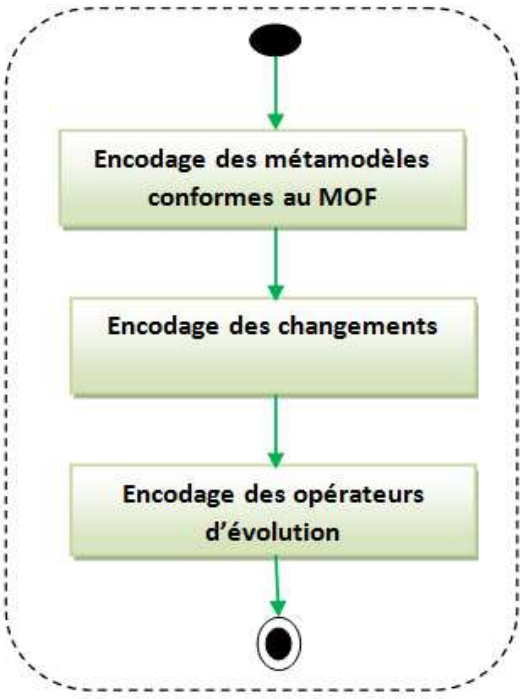

Figure 4.2. Description de la phase préliminaire.

L'étape préliminaire consiste à préparer l'environnement pour l'exécution du processus de coévolution. Elle nous a permis en plus de l'encodage des métamodèles et leurs changements dans un langage formel, de proposer des opérateurs d'évolution dont l'application sur une ancienne version d'un métamodèle garantit de produire une nouvelle version conforme au méta métamodèle MOF. Ainsi, cette phase est réalisée en trois étapes décrites par la figure (4.2) et détaillées dans les sections suivantes.

4.2.1.1. Encodage des métamodèles et des modèles

Pour la gestion des métamodèles et leurs évolutions, la représentation textuelle ou graphique est généralement insuffisante. Pour cela, nous proposons une représentation formelle des modèles et des métamodèles basée sur la logique des prédicats, ce formalisme est utilisé en programmation logique (PL). L'encodage proposé est inspiré de celui défini pour la détection des incohérences structurelles et comportementales dans les modèles UML (Magouyres, 2007). Ainsi, comme chaque modèle est une instance du

métamodèle, un modèle formel peut être déduit de la formalisation d'un métamodèle. Néanmoins, dans cette étape nous nous focalisons uniquement sur les constructeurs essentiels du métamodèle, nous éliminons tous les constructeurs qui n'ont pas d'instances au niveau modèle, comme les packages, les annotations, les opérations. Pour cela, il est nécessaire de proposer une démarche systématique. Ainsi, nous définissons un format d'encodage des modèles en logique des prédicats respectant la hiérarchie de modélisation à trois niveaux : méta métamodèle (M3), métamodèle (M2) et modèle (M1), où chaque niveau est une instance du niveau le plus haut comme illustré par la figure (1.4). Les constructeurs au niveau métamodèle sont représentés par des méta-faits. L'instanciation d'un méta-fait c'est un fait qui consiste à l'attribution des valeurs aux paramètres de ce méta-fait. Ainsi, les faits permettent la représentation des instances (ex : objets) dans un modèle. Un méta-fait est défini par son nom, ses paramètres et la signification de chacun de ses paramètres. Cette définition d'un méta-fait peut être représentée comme un méta-méta-fait.

a) Niveau méta-métamodèle

Les constructeurs du MOF sont représentés par des méta-méta-faits et des méta-méta-règles. Les méta-méta-faits représentent les méta-classes et les méta-associations qui sont des constructions concrètes puisque elles peuvent avoir des instances au niveau plus bas, à l'inverse des constructions abstraites. L'instanciation des méta-méta-faits en méta-faits est faite selon le métamodèle à analyser.

La représentation générale des méta-classes du MOF est comme suit :

Meta_Classe_Name(Identificateur,A1,….,An). (4.1)

Où « *Meta_Classe_Name* » est le nom de la méta-classe, « *Identificateur* » est l'identificateur qui nous sert à identifier l'instance de la méta-classe et « *A1,…,An* » représentent les attributs de la méta_classe avec prise en compte des attributs hérités.

En prenant les méta-classes du MOF permettant de définir les métamodèles qui lui sont conformes, nous définissons un ensemble de méta-méta-fait représentés dans le tableau (4.1) par l'instanciation du méta-méta-fait général par (4.1) par les caractéristiques des méta-classes. Ces méta-classes sont principalement : *Class, Property, Classifier, StructuralFeature, TypedElement, Feature, NamedElement, Enumeration, EnumerationLiteral, DataType, PrimitiveType*.

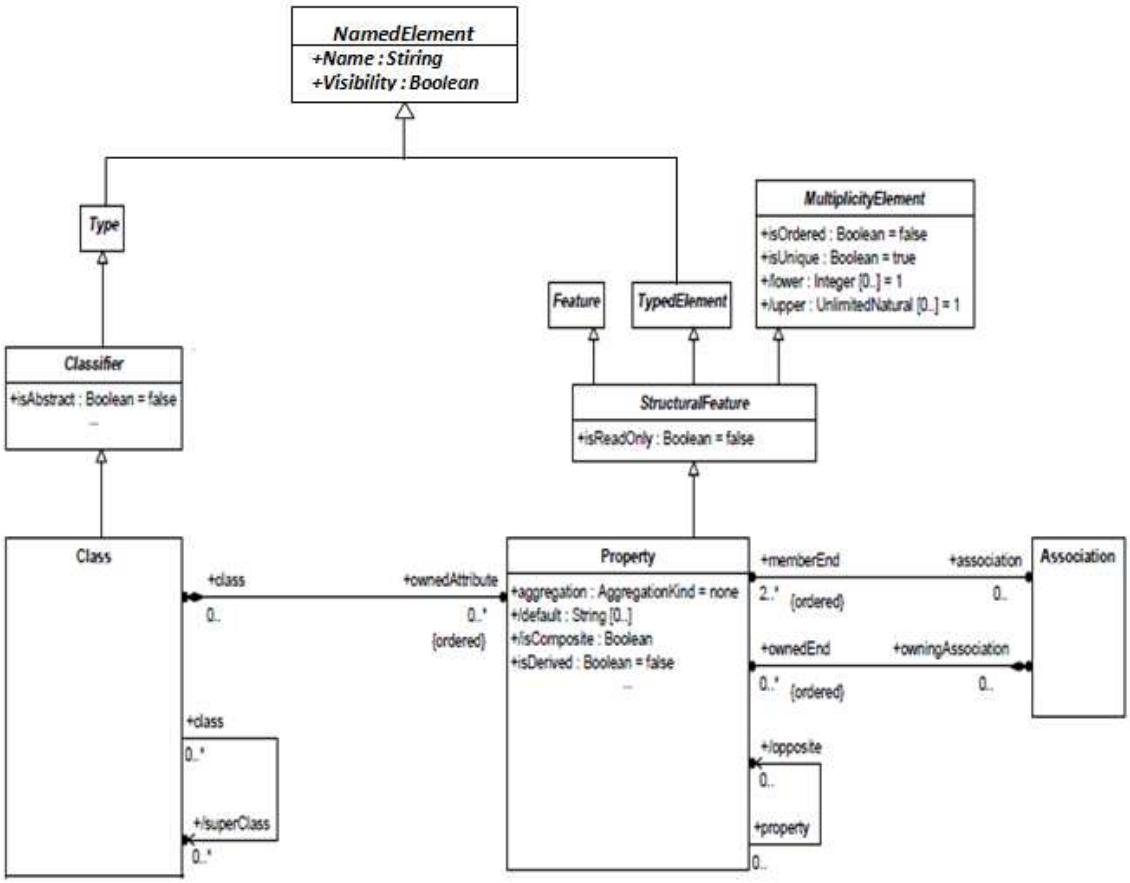

Figure 4.3. Sous ensemble du MOF (OMG, 2011a).

Considérons le sous-ensemble du modèle MOF représenté par la figure (4.3). Ce modèle exprime qu'une classe (méta-classe *Class*) est un classificateur (méta-classe « Classifier »), ce dernier est un type (méta-classe « *Type* ») qui est à son tour un élément nommé (méta-classe « *NamedElement* »). Une classe est donc caractérisée par un nom, une visibilité (par héritage de « *Visibility* »de la méta-classe *NamedElement*) et peut être abstraite (par héritage de la méta-classe *Classifier*). Une substitution au niveau du méta-méta-fait (4.1) par les caractéristiques de la méta-classe « *Class* » résulte le méta-méta-fait (4.2) qui permet de déduire une représentation en méta-faits de toutes les classes d'un métamodèle.

class(IdClasse,Name, Visibility, IsAbstract). (4.2)

« Name,visibility , Is abstract » sont les caractéristiques de la classe.Où *« Name »* est le nom de la classe, *« Visibility »* est la visibilité de la classe et *« IsAbstract »* indique si la classe est abstraite ou non.

Une classe peut contenir des attributs. Les attributs sont représentés par la méta-classe *Property* et ont un ensemble de caractéristiques obtenues par héritage (*Name, Visibility, Isreadonly, Isordered, Isunique, Lower, Upper*) et un ensemble de caractéristiques

spécifiques (*Default,Iscomposite, Aggregation, Isderived*). L'instanciation du méta méta-fait général (4.1) par les caractéristiques de La méta-classe « *Property* » résulte le méta-méta-fait suivant :

property(IdProp,Name, Visibility, Isreadonly, Isordered, Isunique, Lower, Upper, Default,Iscomposite, Aggregation, IsDerived). (4.3)

Ainsi, La relation d'instanciation entre le méta-méta-fait général et les méta-méta-faits consiste à définir le nom, la dimension et les noms des différents paramètres du prédicat relatif au méta-méta-fait.

Selon MOF, un autre type de constructeurs est manipulé pour représenter les relations entre méta-classes. Ainsi, les fins des méta-associations navigables entre méta-classes sont formalisées de manière générale par un méta-méta-fait qui prend en argument les identificateurs des méta-classes participantes à l'association :

Association_End_Name(Id1, Id2). (4.4)

Où « *Association_End_Name* » est le nom de la méta-association, « *Id1* » *et* « *Id2* » sont les identificateurs des méta-classes.

En considérant le sous ensemble du modèle MOF de la figure (4.3). Par exemple Les relations entre classes (méta-classe *Class*) et attributs (méta-classe *Property*) peuvent être représentées par les méta-méta-faits instanciés à partir du méta-méta-fait général présenté en (4.4) :

$$\begin{cases} \text{ownedAttribute(IdClass, IdAttribute).} \\ \text{class(IdAttribute, IdClass).} \end{cases} \quad (4.5)$$

Où « IdClass » et « IdAttribute » sont les identificateurs de la classe et de l'attribut contenu dans cette classe.

Les autres fins d'associations sont formalisées par les méta-méta-faits résumés dans le tableau (4.1).

D'un autre côté, des méta-méta-règles sont utilisées pour représenter les constructions abstraites qui sont ensuite instanciées en méta-règles en fonction du métamodèle à analyser. Ainsi, Les méta_généralisations mettant en relation deux méta-classes doivent assurer la projection des caractéristiques de la méta-classe spéciale vers la méta-classe générale selon le principe d'héritage, les identificateurs des deux méta-classes liées par la relation de méta-généralisation doivent être identiques. Toute méta-généralisation pourra donc être représentée par une règle qui suit le format :

General_Class(Id,A_1,.....,A_n) **:-** Special_Class(Id,B_1,…,B_m). (4.6)

Où « *General_Class* » et « *Special_Class* » sont respectivement les noms de la méta-classe

générale et de la méta-classe spécifique, « *Id* » est l'identificateur des deux classes générale et spéciale. « A_1,…..,A_n » représentent les attributs de la classe générale et « B_1,…,B_m » représentent les attributs de la classe spécifique. Notons que si un méta-attribut appartient aux deux classes, il doit apparaître dans la méta-règle. Dans le cas où le méta-attribut ne figure que dans la classe spécifique alors sa valeur est indifférente dans la règle. De manière plus formelle, Soit les méta-faits suivants de la classe générale et spécifique :

General_Class(Id,A1,….,An).

Special_Class(Id,C1,….,Cm).

Puisque tout attribut de la classe générale est hérité par la classe spécifique, on a dans (4.6) :

$m \geq n \wedge \forall i \in \{1..n\} \quad \exists j \in \{1..n\}$ tel que $Ci = Bj$

On a si $\exists$ i,j tel que Ai = Cj alors dans (4.6) Bj = Cj sinon Bj = _.

Les méta-généralisations présentes dans le modèle MOF (figure 4.3) sont représentées par des méta-méta-règle dans le tableau (4.1).

Par exemple, les éléments nommés sont représentés par la méta-classe *NamedElement* et n'ont pas d'instances propres dans les modèles. Les éléments nommés peuvent cependant apparaître sous plusieurs formes (Classe, propriété, .. etc) qui correspondent aux méta-classes qui spécialisent *NamedElement*. Ainsi, à partir de la présence d'une classe « *Class* » on peut déduire la présence d'un élément nommé « *NamedElement* », puisque chaque instance d'une classe est aussi une instance d'un élément nommé. De manière similaire la méta-classe « *StructuralFeature* » n'a pas d'instance directe au niveau (M1) cependant elle peut apparaître à travers les instances de la méta-classe « *Property* ». En termes de prédicats cette dernière déduction est réalisée par la règle :

structuralFeature(Id,Name, Visibility, Isreadonly, Isordered, Isunique,Lower, Upper) **:-**
property(Id,Name, Visibility, Isreadonly, Isordered, Isunique,Lower, Upper,
_Default, _Iscomposite, _Aggregation, _Isderived) (4.7)

Notons que les caractéristiques de l'instance de la méta-classe générale doivent correspondre aux caractéristiques de la méta-classe spécifique. C'est pourquoi dans (4.7) tous les méta-attributs de la méta-classe « *Property* » qui sont aussi des méta-attributs de la méta-classe « *StructuralFeature* » sont projetés sur cette dernière, c'est le cas des attributs (*Name*, *Visibility*, *Isreadonly*, *Isordered*, *Isunique*,*Lower*, *Upper*). Cependant, certains attributs de la classe spécifique n'appartiennent pas aux attributs de la classe générale, c'est le cas par exemple de la valeur par défaut de la propriété représenté par l'attribut «*Default* ». La valeur de ce genre d'attributs n'a aucun impact pour la règle et commence donc par (_).

D'autre part, les méta-associations accessibles par des méta-classes générales doivent également l'être par la méta-classe spécifique. Or, puisque les identificateurs des méta-classes générales et spécifiques sont identiques, la méta-classe spécifique aura accès aux fins des méta-associations accessibles depuis la méta-classe générale. Ceci met en œuvre de manière naturelle l'héritage des méta-associations.

MetaClass_Name(Identificateur, A1,…., An).	
Méta méta-faits	namedElement(Id,Name,Visibility). classifier(Id,Name, Visibility, IsAbstract). class(Id,Name, Visibility, IsAbstract). feature(Id,Name, Visibility). structuralFeature(Id,Name,Visibility, Type, Isunique, Isordered, Lower,Upper,Isreadonly). property(Id,Name,Visibility, Type, Isreadonly, Isordered ,Isunique, Lower, Upper, , Default, Iscomposite, Aggregation, IsDerived). association(Id).
Association_End_Name(id1, id2).	
Méta méta-faits	ownedAttribute(IdClass, IdProperty). class(IdProperty,IdClass). superClass(IdClass, IdClass). memberEnd(Idassociation, IdProperty). owningassociation(IdProperty, Idassociation). opposite(Idproperty, Idproperty).
General_Class(Id,A1, ...,An) : - Special_Class(Id,B1,...,Bm).	
Méta-méta-faits	multiplicityFeature(Id, Isordred, Isunique, Lower, Upper) :- structuralFeature(Id,Name, Visibility, Type, Isreadonly, Isordred, Isunique, Lower, Upper). typedElement(Id,Name, Visibility,type) : - structuralFeature(Id,Name, Visibility, Type, Isreadonly, Isordred, Isunique, Lower, Upper). structuralFeature(Id,Name, Visibility, Type, Isreadonly Isunique, Isordered, Lower, Upper) :- Property(Id,Name, Visibility, Type, Isunique, Lower, Upper, Isreadonly, _Default, _Iscomposite, _IsDerived).

Tableau 4.1. Extrait de l'encodage des métamodèles MOF en PL.

b) Niveau métamodèle

Les modèles doivent être encodés par des faits (PL). Nous décrivons dans cette section la démarche générale d'obtention de ces faits. Pour chaque constructeur du métamodèle qui représente un type d'élément qui peut se trouver dans un modèle, nous déduisons un format que devra respecter les faits représentant un élément de ce type. Ce format de fait est appelé méta-fait. La définition d'un méta-fait consiste à définir son nom, sa dimension et la signification de chacun de ses arguments. Ces méta-faits sont eux-mêmes une instanciation des définitions du niveau supérieur. Ainsi, l'instanciation des méta-méta-faits représentés dans le tableau (4.1) résulte des méta-faits qui permettent de représenter le

métamodèle à analyser.

Soit l'exemple de la figure (4.4) représentant le métamodèle des réseaux de Petri (Wachsmuth, 2007), Ce métamodèle est conforme au MOF. Par exemple (*Net*, *Transition* et *Places*) sont des classes « type *Class* », (*pname* et *tname*) sont deux propriétés « type *Property* » par contre (*psrc,tsrc,pdst,tdst*) sont des associations *« type Property, Association »*. L'ensemble des constructeurs contenus dans ce métamodèle avec leurs caractéristiques sont explicités dans le code XML de ce métamodèle illustré par la figure (4.5). L'encodage en PL du métamodèle des réseaux de petri consiste à l'instanciation des méta-méta-faits présentés dans le tableau (4.1) en tenant compte du type des éléments : classe, propriété, association, …etc. Le résultat de cette opération d'encodage est résumé dans le tableau (4.2).

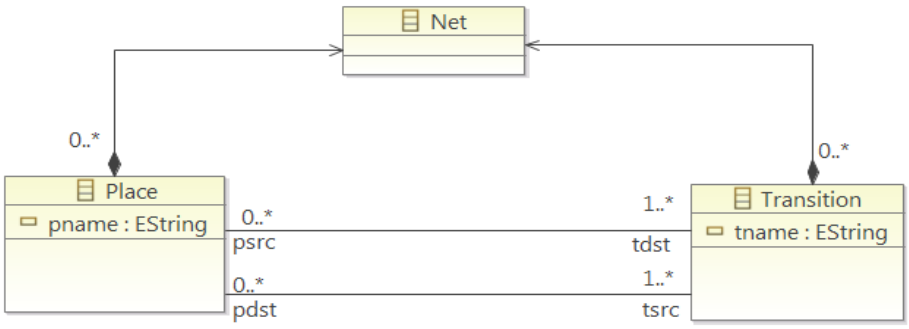

Figure 4.4. Métamodèle des réseaux de Petri (Wachsmuth, 2007).

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="petri1"
    nsURI="http://petri1/1.0" nsPrefix="petri1">
  <eClassifiers xsi:type="ecore:EClass" name="Net" abstract="true"/>
  <eClassifiers xsi:type="ecore:EClass" name="Place">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="pname" eType="ecore:EDataType
    http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="Ref0" upperBound="-1" eType="#//Net"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="psrc" upperBound="-1" eType="#//Transition"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="pdst" upperBound="-1" eType="#//Transition"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="tdst" eType="#//Transition"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Transition">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="tname" eType="ecore:EDataType
    http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="Ref1" upperBound="-1" eType="#//Net"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="tsrc" upperBound="-1" eType="#//Place"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="tdst" upperBound="-1" eType="#//Place"/>
  </eClassifiers>
</ecore:EPackage>
```

Figure 4.5. Code XML du métamodèle des réseaux de Petri.

Méta méta-fait	class(Id,Name, Visibility, IsAbstract).
Méta-faits	class(IdNet,Net, public, false). class(IdTransition,Transition, public, false). class(IdPlace,Place, public, false).
Méta-méta-fait	property(Id,Name,Visibility, Type, Isunique, Isordered, Lower, Upper, Isreadonly, Iscomposite, IsDerived).
Méta-faits	property(IdTname,Tname,public, string, false, true, 1, 1, false, false, false). property(IdPname,Pname,public, string, false, true, 1, 1, false, false, false). property(IdRef0, Ref0,public, Net, false, true, 0, -1, false, true, false). property(IdRef1, Ref1,public, Net, false, true, 0, -1, false, true, false). property(IdPsrc,Psrc,public, Transition, false, true, 0, -1, false, false, false). property(IdPdst,Pdst,public, Transition, false, true, 0, -1, false, false, false). property(IdTsrc,Tsrc,public, Place, false, 1, -1, true, false, false, false). property(IdTdst,Tdst,public, Place, false, 1, -1, true, false, false, false).
Méta-méta-fait	association(Id).
Méta-faits	association(Idasso1). association(Idasso2). association(Idasso3). association(Idasso4). association(Idasso5). association(Idasso6).
Méta-méta-fait	ownedAttribute(IdClass, IdProperty).
Méta-faits	ownedAttribute(IdTransition, IdTname). ownedAttribute(IdPlace, IdPname). ownedAttribute(IdTransition, IdTsrc). ownedAttribute(IdTransition, IdTdst). ownedAttribute(IdPlace, IdPsrc). ownedAttribute(IdPlace, IdPdst). ownedAttribute(IdNet, IdRef0). ownedAttribute(IdNet, IdRef1).
Méta-méta-fait	owningassociation(IdProperty, Idassociation).
Méta-faits	owningassociation(Idpsrc, Idasso1). owningassociation(Idpdst, Idasso2). owningassociation(Idtsrc, Idasso3). owningassociation(Idtdst,Idasso4). owningassociation(IdRef0,Idasso5). owningassociation(IdRef1,Idasso6).
Méta-méta-fait	opposite(Idproperty, Idproperty).
Méta-faits	opposite(IdPsrc,IdTdst). opposite(IdTsrc,IdPdst). opposite(IdTdst,IdPsrc). opposite(IdPdst,IdTsrc).

Tableau 4.2. Exemple d'encodage en PL du métamodèle des réseaux de Petri.

L'encodage des modèles instances est obtenu par instanciation des méta-faits avec les

caractéristiques des objets et des liens dans ce modèle.

4.2.1.2. Encodage des changements d'un métamodèle en PL

Une différence est la représentation d'un changement réalisé entre deux versions d'un métamodèle, elle peut décrire le changement sur un ou plusieurs éléments. Tout dépend de la méthode utilisée pour identifier ces différences. Plus l'algorithme est spécifique à un domaine ou à un langage, plus les différences identifiées sont riches. Les méthodes classiques d'identification de différences se veulent indépendantes du format à différencier (langage, modèle, texte, pages web, etc.), elles ne sont donc pas représentatives des évolutions réellement appliquées sur le métamodèle. De même, la méthode visuelle pour comparer deux métamodèles est une méthode qui prend du temps, il est nécessaire de bien comprendre le métamodèle pour déterminer les différences. L'introduction d'erreurs ou d'oublis est non négligeable. Les méthodes de comparaison automatiques en utilisant un algorithme de différences classique fournissent quant à elles une liste de différences indépendantes et non ordonnée. Elles offrent cependant l'avantage de ne pas exiger l'intervention du concepteur des modèles. Une différence doit contenir au minimum les deux informations suivantes : le type de l'opération qu'elle représente, et l'élément qu'elle référence.

Nous modélisons les différences entre un métamodèle originale MM1 et une version évoluée MM2 par un ensemble de changements. Ces derniers sont de trois types : changements additifs, où le métamodèle évolué contient des éléments qui n'ont pas été présents dans le métamodèle originale. Les changements soustractifs, où le métamodèle évolué ne contient pas des éléments qui ont étaient présents dans le métamodèle originale. Les changements modificatifs, dans ce cas le métamodèle évolué contient un élément qui correspond à un élément du métamodèle originale mais les valeurs de ses caractéristiques (méta-features) sont différentes.

Pour l'encodage de ces changements en PL, nous proposons deux méta-méta-fait. Le premier (4.8) pour représenter les changements portant sur les méta-classes et le second (4.9) représente les changements qui affectent les relations entre méta-classes :

$$\begin{cases} \text{Type\_Change(Type\_élément, Id , [P1,…,Pn]).} & (4.8) \\ \text{Type\_Change} \in \{\text{add, delete, update}\} \\ \text{Type\_élément} \in \{\text{class, property, association,…}\} \end{cases}$$

Où « Type_Change » indique le type du changement qui peut être un des trois types primaires, à savoir : addition, suppression ou modification. «Type_élément» et «Id» indiquent respectivement le type et l'identificateur de la méta-classe sujet du changement

à savoir classe, propriété ou association; [P1,…,Pn] est une liste de paramètres qui varient selon le type du changement ainsi que les caractéristiques de la méta-classe. Chaque paramètre étant un couple qui représente le nom de l'attribut sujet du changement et sa valeur [Attribut, Valeur].

$$\begin{cases} \text{Type\_Change(Type\_élément, [Id1,Id2]).} & (4.9) \\ \text{Type\_Change} \in \{\text{add, delete, update}\} \\ \text{Type\_élément} \in \{\text{superType, ownedAttribute owningassociation,..}\} \end{cases}$$

Où « Type_Change » indique le type du changement qui peut être un des trois types primaires, à savoir : addition, suppression ou modification. «Type_élément» le type et de la méta-relation sujet du changement; [Id1,Id2] représentent les identificateurs des méta-classes participantes à la relation.

Méta-faits des changements	Description
add(class, Idc,[[name,VNom]]).	Ajout d'une nouvelle classe avec le nom «VNom». Les valeurs par défaut sont attribuées à « visibility » et « isAbstract».
add(class, Idc, [[name, VNom], [visibility, Vvisibility],[isAbstract, VisAbstract]]).	Ajout d'une nouvelle classe avec le nom «VNom » et des valeurs spécifiques de « visibility » et « isAbstract »
add (property,Idp, [name, Vnom]). add(ownedAttr,[Idc, Idp]).	Ajout d'une nouvelle propriété avec le nom « VNom » et Attribution de cette propriété à la classe « Idc ».
add(property, Idp, , [[name, VNom], [isreadonly, Visreadonly],[isIdentifier, VisIdentifier]]).	Ajout d'une nouvelle propriété avec le nom « VNom » avec des valeurs spécifique de « IsReadOnly », « IsIdentifier ».
add (property, Idpa, , [[name, Vnom], [type, Idc],[isComposite , VisComposite], [lower,Vlower] ,[upper,Vupper]]). add(association, ida) add(ownedAttr, [Ids, Idpa]). add (owningass, [Idpa,Ida]).	Ajout d'une nouvelle association avec le nom « VNom » de la classe « Ids » à la classe « Idc » avec des attributs spécifique « IsComposite », « Lower » et « Upper ».
add(superType,[Ids,Idg])	Définir la classe « Idg » comme super type de la classe « Ids ».
delete(class,Idc, []).	Supprime la classe « Idc ».
delete(property,Idp, []).	Supprime la propriété « Idp ».
delete(association,Ida, []).	Supprime l'association « Ida ».
update(class,idc, [[name, VNom]]).	Changer le nom de la classe « Idc » avec la nouvelle valeur « VNom ».
update(association,Ida, [[lower,VLower]]).	Modifie la multiplicité inférieure « lower » de l'association « ida » à la nouvelle valeur « VLower».

Tableau 4.3. Des clauses PL pour l'encodage de quelques changements des métamodèles.

Les méta-méta-faits (4.8) et (4.9) correspondent au niveau M3. Dans le niveau M2, nous définissons des méta-faits en se basant sur les changements possibles des métamodèles

décrits dans la littérature (Wachsmuth, 2007; Gruschko et al, 2007; Becker et al 2007 ; Cicchetti et al, 2008; Herrmannsdorfer et al, 2008 ; burger et al, 2010) et présentés dans le chapitre 2. Par exemple « add(class, Idc, [[name,VNom]]) » représente le méta-fait du changement qui permet la création d'une nouvelle classe dont le nom est « VNom ».

Le tableau 4.3 résume l'encodage proposé pour quelques changements.

4.2.1.3. Encodage des opérateurs d'évolution

L'encodage des opérateurs en PL consiste d'abord à identifier les éléments évolutifs d'un métamodèle puis à proposer les opérateurs d'évolution, qui lorsque ils sont appliqués sur une version valide d'un métamodèle résulte une nouvelle version qui soit valide. Cette étape est réalisée uniquement une fois mais constitue le cœur de notre approche. Elle comporte trois tâches illustrées par la figure (4.6) :

- Identification des éléments évolutifs d'un métamodèle MOF.
- L'analyse des métamodèles MOF pour identifier et formaliser les opérateurs d'évolution atomiques ;
- La définition et la formalisation des opérateurs d'évolution composites applicable sur les métamodèles MOF.

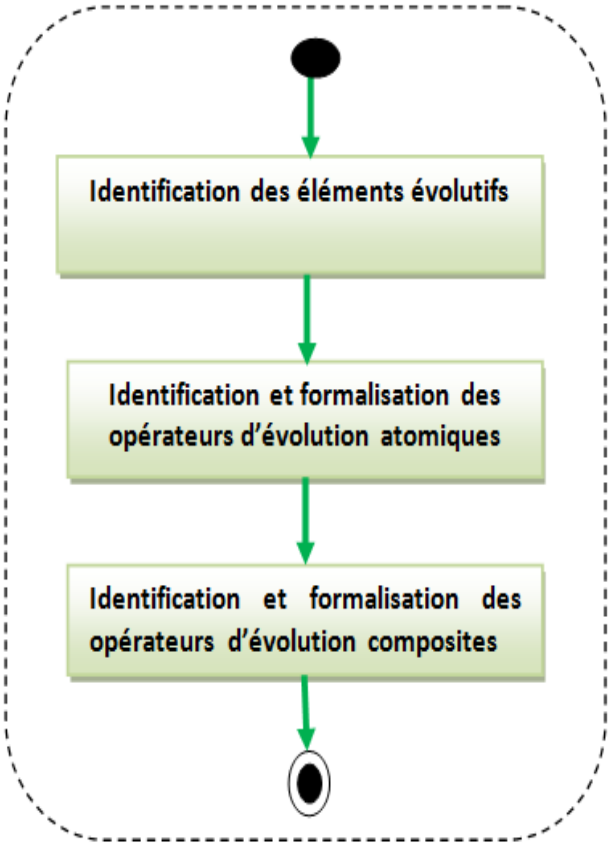

Figure 4.6. Description de l'étape d'encodage des opérateurs d'évolution.

a) Détermination des éléments évolutifs d'un métamodèle MOF

Le méta métamodèle MOF tel qu'il est défini n'explicite ni les opérateurs d'évolution qu'il autorise ni les éléments évolutifs des métamodèles qui lui sont conformes. Un élément évolutif étant un élément du métamodèle qui peut subir une opération d'évolution (ajout, suppression, modification) indépendamment des autres éléments (Lakhal, 2015). Dans le cas contraire, l'élément est considéré comme paramètre d'évolution dans le contexte d'un autre élément. La première étape est justement d'analyser les métamodèles MOF pour reconnaître ces éléments évolutifs. Ainsi, si une différence référence un élément autre que ceux appartenant à cet ensemble, aucun opérateur conforme au métamodèle MOF n'est réellement applicable sur cet élément et il s'agit donc d'une erreur de comparaison ou d'une opération spécifique à l'outil de comparaison.

Selon le méta métamodèle MOF, l'élément « *Package* » est lié par un lien de composition à l'élément « *Type* » qui est une généralisation des éléments : « *Class* » *et* « *Datatype* » (figure 4.7). Le lien de composition indique que l'ajout et la suppression de ces éléments sont invoqués par l'élément « *Package* », il est donc le contexte d'évolution de ces éléments. La multiplicité infinie du lien de composition indique que les éléments « *Class* » et « *Datatype* » peuvent aussi évoluer sans nécessiter ou entraîner l'évolution de l'élément « *Package* ». La modification de l'élément « *Package* » est liée d'une part aux modifications des éléments qu'il contient et d'autre part aux valeurs du reste de ses paramètres d'évolution (méta-informations). Nous considérons les éléments « *Class* » et « *Datatype* » des éléments évolutifs à part entière d'un métamodèle MOF.

Figure 4.7. Liens de composition entre Package et Type.

La figure (4.8) montre que l'élément « *Enumeration* » est un type de « *Datatype* », il permet d'attribuer une liste de valeurs possibles à une propriété ou à un paramètre. L'utilisation d'une énumération est importante dans un métamodèle. Nous limitons son utilisation comme type possible d'une propriété ou d'un paramètre. L'élément « *Enumeration* »

possède un lien de composition de multiplicité infinie vers l'élément « *EnumerationLiteral* » signifiant qu'un littéral peut être ajouté, supprimé ou modifié sans préalablement ajouter, supprimer ou modifier une énumération. Les éléments « *Enumeration* » et « EnumerationLiteral » sont considérés des éléments évolutifs d'un métamodèle MOF.

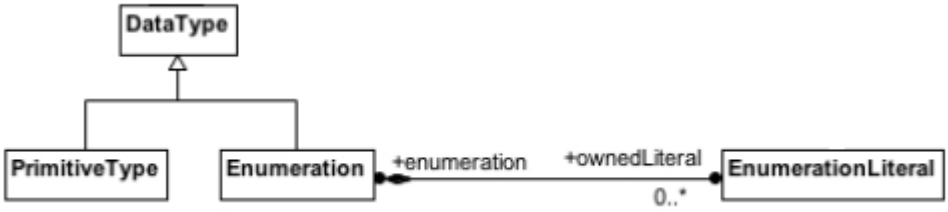

Figure 4.8. Liens de composition entre Enumeration et Enumerationliteral.

L'élément « *Class* » possède un lien de composition vers les éléments « *Property* » et « *Operation* » dont la multiplicité est infinie (figure 4.9). Cette multiplicité signifie qu'une propriété peut être ajoutée dans une classe sans ajouter au préalable une nouvelle classe. De la même manière, une propriété peut être supprimée ou modifiée sans respectivement supprimer ou modifier la classe elle-même. Il en est de même pour l'élément « *Operation* ». Nous en déduisons donc que « *Property* » et « *Operation* » sont des paramètres du contexte *Class* mais également des éléments évolutifs d'un métamodèle.

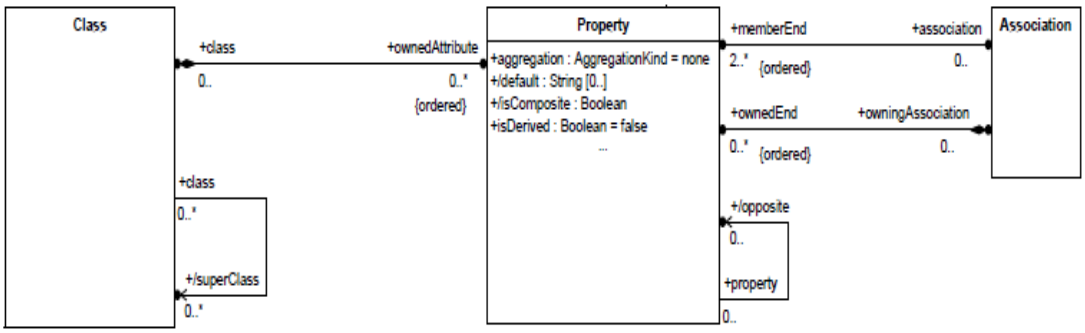

Figure 4.9. Liens de composition entre Class et Property.

Selon le méta-métamodèle MOF, La définition d'une association est représentée par l'élément « *Association* » possédant une relation simple navigable (non composite) avec l'élément « *Property* » L'extrémité « memberEnd » de cette relation est de multiplicité [2..*] (figure 4.9). Ces extrémités étant issues entre deux éléments du métamodèle, elles sont considérées comme des paramètres de l'élément Association. La définition de l'élément Association oblige également à définir une propriété dans une classe qui référence l'association, c'est ce type de propriété qu'une association peut ajouter sans en définir une nouvelle. La multiplicité de cette relation confirme que l'élément « Property » est un élément évolutif dans un métamodèle.

Donc l'ensemble des éléments évolutifs utilisés dans la définition d'un métamodèle et dont les modèles sont impactés sont essentiellement : « Class », « Enumeration »,

« EnumerationLiteral », « Property » et « Association ». C'est à partir de cette liste uniquement que nous proposons des opérateurs d'évolution.

b) Description des opérateurs d'évolution

En pratique, un langage de modélisation et par conséquent son métamodèle évolue par des adaptations incrémentales. Il existe un certain nombre de changements de métamodèle qui sont primitifs, tels que créer un élément, supprimer un élément,...etc. Un ou plusieurs de ces changements composent une adaptation spécifique du métamodèle. Dans les approches à base d'opérateurs, une bibliothèque d'opérateurs est utilisée où des informations relatives à la migration des modèles sont attachées avec l'adaptation du métamodèle. De cette manière, la migration du modèle peut être déjà capturée pendant l'adaptation de son métamodèle. Ainsi, Les opérateurs sont utilisés à la fois pour évoluer les métamodèles et ils sont capables de faire une migration automatique des modèles existants. Dans notre approche nous utilisons également une bibliothèque d'opérateurs où les opérateurs sont également couplés en regroupant l'évolution du métamodèle avec la migration des modèles. Néanmoins, ces opérateurs ne sont pas utilisés dans le but d'évoluer le métamodèle. Ils sont utilisés pour reconstruire le scénario d'évolution et sur sa base migrer les modèles.

Un opérateur d'évolution couplé «Cop» est donc défini comme étant un couple (evol, mig)

$$\begin{cases} \text{Cop= (evol,mig)} \\ \text{où} \quad \text{evol : représente l'action d'évolution du métamodèle (MM).} \\ \text{Mig : représente l'action de migration du modèle (M)} \end{cases}$$

Tel que l'application d'un opérateur « Cop » sur un métamodèle MM résulte un métamodèle valide. L'application successive de plusieurs opérateurs sur un métamodèle constitue une composition séquentielle de ces opérateurs :

Soit Cop1= (evol1,mig1) et Cop2=(evol2,mig2) La composition de Cop1 et Cop2 notée (Cop1 **o** Cop2) résulte l'opérateur Cop= (evol,mig)

Tel que $\begin{cases} \text{evol =evol1 } \mathbf{o} \text{ evol2 et mig = mig1 } \mathbf{o} \text{ mig2} \end{cases}$

Les opérateurs d'évolution sont définis de manière modulaire c'est-à-dire la migration est définie indépendamment des autres opérateurs.

Un opérateur est défini de manière générale pour être appliqué sur un type d'élément constructeur du métamodèle (méta-classe, méta-propriété,….). L'application de l'opérateur sur un élément du métamodèle considéré (instance d'une méta-classe) constitue une instance de cet opérateur que nous appelons une opération d'évolution. Par exemple, nous définissons un opérateur « *Renommer* » pour changer le nom d'une méta-classe, les

principaux paramètres de cet opérateur étant la classe sujet du changement et la nouvelle valeur de son nom. L'application de cet opérateur sur une classe particulière d'un métamodèle constitue une opération qui consiste à renommer une classe par exemple « *Arc* » par le nouveau nom « *PTArc* ».

De plus, nous différencions les opérateurs atomiques ou primitifs qui ne font évoluer qu'un seul élément et réalisent une étape d'évolution atomique du métamodèle qui ne peut être subdivisée encore et les opérateurs composites qui sont des compositions d'opérateurs atomiques.

Chaque opérateur d'évolution proposé est défini par un ensemble de méta-informations que l'on nomme paramètres d'évolution. Un paramètre d'évolution représente une méta-information utilisée par le langage MOF pour définir un élément constructeur d'un métamodèle. Par exemple, l'élément «*Property* » est un élément constructeur qui possède entre autre une multiplicité minimale et maximale. Ces deux multiplicités servent à définir ce qu'est une propriété dans un métamodèle, elles sont donc des paramètres d'évolution de cette propriété.

c) Aperçu sur la bibliothèque d'opérateurs

Une fois la détermination des éléments évolutifs achevée, nous savons désormais sur quel élément peut s'appliquer un opérateur d'évolution. Le but de cette seconde étape n'est pas de proposer de nouveaux opérateurs d'évolution atomiques mais d'extraire à partir des métamodèles MOF les opérateurs qui sont implicitement autorisés.

Dans notre approche nous avons proposé d'utiliser une liste la plus exhaustive que possible pour les opérateurs d'évolution, tout en les formalisant en langage des prédicats pour permettre le raisonnement. Nous proposons en perspective d'étudier les possibilités de factorisation de ces opérateurs.

Dans la bibliothèque des opérateurs proposée nous avons inclus l'ensemble des opérateurs trouvés dans la littérature et qui s'adaptent aux métamodèles conformes au MOF 2 (OMG, 2015) avec une possibilité d'extension en définissant des nouveaux opérateurs qui apparaissent selon le besoin. Pour l'organisation de cette bibliothèque nous considérons deux grandes catégories à savoir les opérateurs primitifs qui réalisent une étape d'évolution atomique du métamodèle qui ne peut être subdivisée encore et les opérateurs composites. Ces derniers peuvent être décomposés en une séquence d'opérateurs primitifs qui ont le même effet au niveau métamodèle mais non nécessairement au niveau modèle. Les opérateurs atomiques eux même peuvent être divisés en opérateurs d'ajouts qui créent des nouveaux éléments dans le métamodèle ; opérateurs de suppression en qui élimine des éléments existants dans le métamodèle, ainsi que les opérateurs de

modifications qui modifient les éléments déjà existants dans le métamodèle. La figure (4.10) illustre l'organisation générale de cette librairie d'opérateurs.

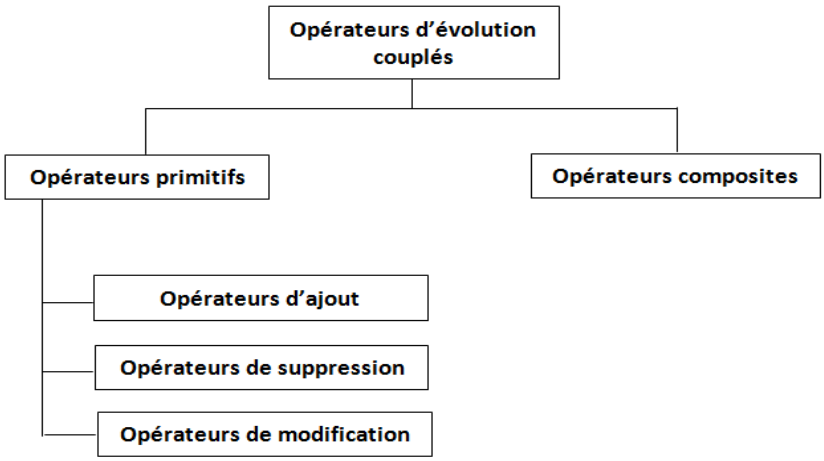

Figure 4.10. Organisation de la librairie des opérateurs.

Les opérateurs seront organisés d'une manière où les parties d'évolution et de migration seront clairement définies pour faciliter l'extension de cette bibliothèque. Les opérateurs seront utilisés après l'évolution du métamodèle.

D) Formalisation des opérateurs atomiques en langage des prédicats

La formalisation des opérateurs atomiques est établie en considérant les principaux éléments évolutifs. Chaque opérateur possède un nombre de paramètres formels, par exemple le nom d'une classe ou d'une propriété. Tout opérateur dans la bibliothèque est décrit par un ensemble d'informations :

- *Description de l'opérateur* : Cette section présente informellement le rôle et la motivation de l'opérateur.
- *Définition de l'opérateur* : Dans cette rubrique nous proposons une définition formelle de l'opérateur en utilisant le formalisme PL.
- *Pré- conditions de l'opérateur* : représentent les conditions qui doivent être vérifiées pour exécuter l'opérateur.
- *Post-conditions de l'opérateur* : résultent après l'exécution d'un opérateur.
- *Action d'évolution de l'opérateur* : utilisée seulement pour tester la validité du scénario d'évolution.
- *Action d'adaptation de l'opérateur* : utilisée pour générer le scénario de migration des modèles.

Les tableaux (4.4), (4.5) et (4.6) représentent la structure de trois opérateurs atomiques différents : ajout d'une propriété, suppression d'une classe et suppression d'une propriété.

Avant d'aborder la définition des opérateurs, nous présentons quelques notations qui

seront utilisées par la suite :

TypeOf(E) : retourne le type de E.

OldMMElements : L'ensemble des instances des éléments évolutifs définis dans l'ancienne version du métamodèle.

NewMMElements : L'ensemble des instances des éléments évolutifs définis dans la nouvelle version du métamodèle.

Ownedmembers(E) : L'ensemble des éléments contenu dans E est un ensemble d'éléments dont E est le conteneur.

Add-collection (S, E): Ajouter l'élément E à l'ensemble S.

Opérateur Ajout d'une propriété « Create Property »	
Description	L'opérateur "Create property" permet d'ajouter un nouvel élément "property" à une classe qui existe déjà dans le métamodèle.
Définition	create_property (Idproperty, Idclass, Name, Visibility, Type, Isunique, Lower, Upper, Isreadonly, Iscomposite, IsDerived, Default).

Pré-condition	**Post-condition**
(Idclass ∈ oldMMElements) and Typeof(idclass) = class and Idproperty ∉ ownedmembers(Idclass)	Idproperty ∈ ownedmembers(Idclass). Idproperty ∈ NewMMElements .
Evolution du métamodèle	**Adaptation du modèle**
Ajoute l'élément *property* au métamodèle avec des caractéristiques spécifiques. property a pour conteneur "Idclass" : property(Idproperty, Idclass, visibility, type, Isunique, lower, upper, Isreadonly, Iscomposite, IsDerived, default) **If** (Idclass ∈ oldMMElements) **then** **If** Typeof(idclass) = class and Idproperty ∉ owndemembers (Idclass) **then** Add-collection (ownedmembers(Idclass), Idproperty)	**If** (lowerValueof(Idproperty) = 0) **then** property est optionnel aucune adaptation n'est nécessaire. **If** (lowerValueof(Idproperty) > 0) and defaultValue_of(Idproperty)< >null **Then** property est instanciée automatiquement , et l'adaptation du modèle est automatique. **If** (lowerValueof(Idproperty) > 0) and defaultValue_of(Idproperty) = null **Then if** typeof(Idproperty) ∈ {String, Boolean, Float, Enumeration } **Then** les options de valeurs sont affichées, l'utilisateur choisit une valeur pour initialiser property. L'adaptation du modèle est dirigée par l'utilisateur. **if** typeof(Idproperty) = class and defaultValue of(Idproperty) = null **then** La création de l'instance de classe ne peut être faite automatiquement. L'adaptation du modèle est manuelle.

Tableau 4.4. Description de l'opérateur ajout d'une propriété.

Opérateur suppression d'une classe « Delete class »		
Description	L'opérateur de suppression « Delete_class » permet de supprimer un élément de type Class qui existe déjà dans le métamodèle. Le résultat est la suppression d'une instance de l'élément class contenu dans la collection "OwnedMembers" d'une instance de l'élément Package	
Définition	delete_class(Idc)	
Pré-condition		**Post-condition**
Typeof(Idc) = class Idc ∈ OldMMElements		Idc ∉ NewMMElements
Evolution du métamodèle		**Adaptation du modèle**
Si Idc ∈ OldMMElements **Si** Typeof(Idc) = class Then Delete-collection (ownedmembers(conteneur), Idc) %% Retire l'élément IdCName de la collection "ownedMembers" appartenant au conteneur.		- l'ensemble des éléments référençant ou reliés à la classe supprimée doivent être soit supprimé, soit relié différemment au métamodèle.

Tableau 4.5. Description de l'opérateur suppression d'une classe.

Opérateur suppression d'une association « Delete_property»		
Description	L'opérateur « Delete_property » permet la suppression d'une propriété de type association appartenant à une classe qui est la source de cette association.	
Définition	delete_property(Idproperty,idclass)	
Pré-condition		**Post-condition**
idproperty , idclass ∈ OldMMElements TypeOf(idproperty) = class		idproperty ∉ New MMElements Id class ne contient plus le lien d'association ou la propriété idproperty.
Evolution du métamodèle		**Adaptation du modèle**
Si idproperty , idclass ∈ OldMMElements **Si** TypeOf(idproperty) = class **alors** **Faire** : RetirerCollection(idclass, idproperty) %% Retire l'association du métamodèle ou de la classe dont elle est la source.		Les liens correspondants à cette association doivent être supprimés.

Tableau 4.6. Description de l'opérateur suppression d'une propriété de type association.

e) Formalisations des opérateurs composites en langage des prédicats

Pour la définition d'un opérateur composite plusieurs opérateurs primitifs sont assemblés. En se basant sur la définition des méta-faits représentant le métamodèle analysé, il est possible d'accéder facilement aux différentes caractéristiques d'un métamodèle. Par exemple pour obtenir le nom des éléments nous définissons une fonction nommée

« getName » définie par la clause suivante :

getName(Id,Name) :- namedElement(Id,Name, _).

Définition des opérateurs primitifs	
create_class(Name, Visibility, IsAbstract)	make_composite(Idass).
delete_class(Id)	drop_composite((Idass)
create_property (Idproperty, Idclass)	make_opposite(Idass1 , Idass2).
create_asso(Name_a, Idclass_s, Idclass_T)	drop_opposite (Idass 1, Idass2)
delete_property(Idproperty)	generalize_lower(Idass, L).
rename_class(Idclasse1, Name_c)	specialize_lower(Idass, L).
rename-property(Idproperty, Name_p)	generalize_upper(Idass, U).
make_abstract(Idclass)	specialize_upper(Idass, U).
drop_abstract(Idclass)	make_identifier(Idproperty).
add_super(Idclass_s, Idclass_G).	drop_identifier(Idproperty).
drop_super((Idclass_s, Idclass_G)	make_composite(Idasso)

Tableau 4.7. Extrait des clauses PL représentant des opérateurs d'évolution primitifs.

Le prédicat «getName» retourne le nom d'un élément dont on fourni son identificateur. De manière similaire nous définissons un ensemble de prédicats pour récupérer les valeurs des caractéristiques des éléments manipulés (getId, getIsAbstract, getlower,….).

D'un autre côté pour tester le type d'un élément nous utilisons des prédicats du genre isClass/1, isProperty/1, isassociation/1 pour tester respectivement si l'élément est une classe, une propriété ou une association. Ainsi pour tester si l'élément est une classe nous proposons la clause suivante :

isClass(Id) :- class(Id,_,_,_).

Ces fonctions intermédiaires (getName, isClass,…) sont insérées dans les clauses des opérateurs composites pour formuler les conditions nécessaires à l'exécution des opérations.

Quelques opérateurs composites sont donnés dans le tableau (4.8). Par exemple, nous considérons l'opérateur "extract super class" où une classe est généralisée dans l'hiérarchie en ajoutant une nouvelle classe générale et des références à ses sous classes. L'ensemble complet des opérateurs primitifs et composites représente une base de connaissances qui sera utilisée par le moteur d'inférence pour la reconstruction du scénario d'évolution.

Définition des opérateurs composites
Pull_up_feature(Idclass, Idproperty) :- typeof(Idproperty, property), findall(Cs, super(Idclass, Cs), Ls), delete_property(Idproperty, Ls) , create_property(Idproperty, Idclass).
Pull_up_feature(Idclass, Idproperty) :- typeof(Idproperty, association), findall(Cs, super(Idclass, Cs), Ls), getName(Idproperty, Name_a), getType(Idproperty,Type), getlower (Idproperty,L), getUpper(Idproperty,U), getComposite(Idproperty,Comp), delete_property(Idproperty, Ls) , create_asso(Name_a, Idclass, Type), specialize_lower(Idproperty, Idclass, L), generalize_upper(Idproperty, Idclass,U).
extract_superclass (Name_c, [Idclass1,..., Idclassk],[Idproperty1, …., Idpropertyj]) :- create_class(Name_c,public,true), getId (Idclass, Name_c), add_super(Idclass1, Idclass),… , add_super(Idclassk, Idclass), pull_up_feature (Idclass, Idproperty1),…, pull_up_feature (Idclass, Idpropertyj).

Tableau 4.8. Exemples des opérateurs d'évolution composites définis par des clauses PL.

4.2.2. Détection des changements

L'évolution d'une version d'un métamodèle vers une autre peut être décrite par une séquence de changements. Il ya deux méthodes pour découvrir les changements (Cicchetti, 2008). Dans la première, les changements peuvent être détectés postérieurement par des méthodes de comparaison en utilisant soit des algorithmes génériques soit des algorithmes de comparaison spécifiques à un langage. Une autre méthode c'est l'enregistrement des opérations pendant l'évolution par l'environnement de modélisation. Cette dernière méthode peut être uniquement utilisée si les métamodèles sont évolués par le même outil pour capturer la trace d'évolution. Dans notre approche nous utilisons la première méthode.

La méthode utilisée pour obtenir ces différences doit permettre de déterminer exhaustivement les différences entre deux versions d'un métamodèle. De plus la différence doit au minimum contenir les trois informations suivantes :

1. Le type de l'élément qui est représenté par la différence (propriété, classe, etc),
2. Le type de changement qui lui est appliqué (ajout, suppression, modification, etc)
3. Un accès possible aux paramètres d'évolution de la différence.

Dans notre approche, nous ne sommes pas intéressés à cette étape car il existe dans la littérature de nombreux outils pour le calcul de différences. L'outil EMFCompare établit ces différences entre deux versions de modèles. Le moteur de différenciation de l'outil détermine la liste de différences comme une liste de changements unitaires indépendants.

Chaque différence n'est décrite que par le nom de l'élément, son type et l'opération générale appliquée. Le choix d'utiliser l'outil EMFCompare n'est que pour des raisons de compatibilité technologique, Indépendamment de l'outil de comparaison utilisé, nous supposons qu'à l'issue de cette étape nous possédons une liste de différences exhaustives et désordonnées.

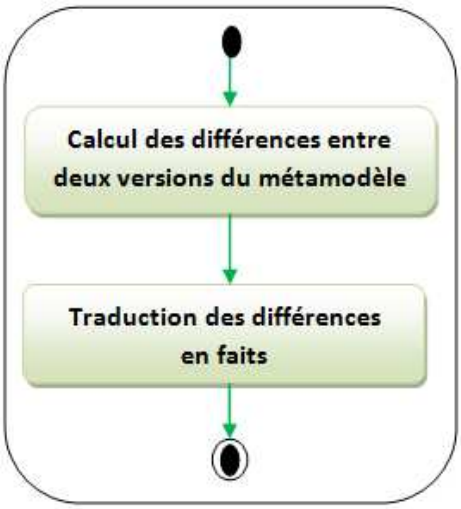

Figure 4.11. Description de l'étape Détection des changements.

Les différences primitives entre les versions du métamodèle sont classées dans trois catégories de base : les ajouts, les suppressions et les modifications des éléments d'un métamodèle (Cicchetti, 2008). Ces différences représentent les changements atomiques. Un ou plusieurs de ces changements compose une évolution spécifique du métamodèle. Chaque différence inclut essentiellement des informations sur le type de changement, l'identité de l'élément sujet du changement ainsi qu'un ensemble de paramètres. L'ensemble des différences détectées est appelé le modèle delta (le modèle de différence). Ce dernier est traduit dans le formalisme PL en instanciant les méta-faits des changements décrits précédemment (tableau 4.3). Ce résultat est utilisé dans la phase suivante pour reconstruire le scénario d'évolution.

4.2.3. Reconstruction du scénario d'évolution

La reconstruction du scénario d'évolution est faite en deux étapes : initialement, nous reconstruisons à partir du modèle delta, un scénario d'évolution correctement ordonné en transformant les faits des changements en opérations d'évolution primitives. Dans la seconde étape, nous utilisons la base de connaissances des opérateurs pour reconstruire les opérateurs composites au moyen d'un moteur d'inférence.

4.2.3.1. Reconstruction des opérations d'évolution primitives

Dans cette étape les changements atomiques calculés et codés sous forme de faits sont transformés en une séquence d'opérations d'évolution primitives appelée scénario d'évolution primitif. Une fois la liste de différences obtenue par comparaison de deux versions du métamodèle, nous procédons à la reconstruction du scénario d'évolution en déduisant les opérations d'évolution qui ont été appliquées sur l'ancienne version du métamodèle.

D'abord les différences sont encodées en PL en instanciant les méta-faits représentant les changements (tableau 4.3). Pour faire correspondre une différence à une opération d'évolution nous récupérons d'abord le type du changement que décrit cette différence (un ajout, une suppression ou une modification). Puis il s'agit de récupérer le type de l'élément sujet du changement (ex : classe, propriété,..) et nous recueillons les paramètres d'évolution (ex : nom, identificateur, multiplicité, …) pour instancier un opérateur d'évolution primitif en une opération d'évolution. Nous obtenons par conséquent un scénario d'évolution. La prise en compte des conditions d'exécution de chaque opération donne une certaine priorité à l'opération dans le scénario par exemple l'ajout d'une propriété ne peut se faire que si sa classe existe déjà ou une opération d'ajout de la classe s'exécute avant. Ainsi le scénario d'évolution est ordonné sur la base de la priorité des opérations. En fin, il s'agit de vérifier la validité du scénario en l'appliquant sur l'ancienne version du métamodèle, s'il résulte la version évolué alors le scénario est bien ordonné sinon le processus de réorganisation est répété.

4.2.3.2. Reconstruction des opérations d'évolution composites

La granularité des changements d'évolution du métamodèle n'est pas toujours appropriée. Parfois, l'intention des changements peut être exprimée à un niveau plus haut. Ainsi, un ensemble de changements atomiques peuvent avoir ensemble l'intention d'un changement composite. Par exemple, la génération d'une super-classe "sc" des deux classes "c1" et "c2" sans attributs communs peut être faite à travers l'application successive d'une liste de changements primitifs :

add(class, Idsc, [[name,'sc']]).	Création d'une classe "sc" ;
add(superType,[Idc1,Idsc]).	Ajouter une référence de "c1" à "sc" ;
add(superType ,[Idc2,Idsc]).	Ajouter une référence de "c2" à "sc" .

Dans cette étape nous appliquons un raisonnement intelligent pour déduire les opérations composites de l'évolution d'un métamodèle. Les faits du scénario d'évolution primitif sont passés en entrée au moteur d'inférence qui utilisera la base de connaissances des

opérateurs pour détecter les opérations composites. Une fois l'opération composite est détectée, la séquence des opérations primitives est fusionné en ajoutant cette nouvelle opération dans la base (assert fact) et par conséquent en supprimant les opérations primitives correspondantes (retract fact). Ceci fournira un nouveau scénario qui sera validé en l'exécutant sur l'ancienne version du métamodèle. Des réorganisations éventuelles des opérations peuvent avoir lieu pour permettre la déduction des opérations composites. Ce processus est répété jusqu'à ce qu'aucune déduction ne puisse être faite. Le scénario d'évolution finale valide peut contenir des opérations primitives ainsi que des opérations composites.

4.2.4. Détermination du scénario de migration

La bibliothèque d'opérateurs regroupe des opérateurs d'évolution associés avec des informations de migration des modèles. Les opérateurs d'évolution ont des impacts différents sur la conformité des modèles (Gruschco et al, 2007). Nous définissons dans la bibliothèque d'opérateurs proposée, quatre catégories d'opérateurs selon cet impact : opérateur sans impact, automatique, dirigé par l'utilisateur et manuel.

- Un opérateur est sans impact quand le changement réalisé en appliquant cet opérateur est non cassant c'est-à-dire le modèle est toujours conforme au métamodèle après son changement, ainsi l'impact de l'évolution de cette catégorie sur la conformité des modèles est nul. Par conséquent aucune adaptation de ces derniers n'est requise, les modèles restent des instances du métamodèle évolué.
- L'opérateur est automatique, si le changement correspondant est cassant et peut être automatiquement résolu. Cette catégorie correspond aux évolutions qui violent la conformité des modèles avec le métamodèle évolué. Par conséquent, Ils requièrent une migration pour redevenir des instances du métamodèle évolué. L'adaptation des modèles dans ce cas est complètement automatisée, elle n'exige pas l'intervention du concepteur des modèles. Pour cette catégorie l'opérateur est réutilisable et à chaque fois la procédure de migration est automatiquement obtenu en instanciant les paramètres.
- L'opérateur est dirigé par l'utilisateur, dans ce cas les changements correspondants ont également un impact sur les modèles, qui ne sont plus des instances du métamodèle évolué. Par conséquent, une migration est requise pour que les modèles redeviennent des instances du métamodèle évolué. La différence avec la catégorie précédente est qu'une partie de l'information demandée par l'adaptation manque pour automatiser complètement l'adaptation. Ces informations ne peuvent

être fournies que par interaction par le concepteur des modèles, car il est le seul à pouvoir compléter l'instanciation de l'élément correctement. Cette catégorie n'existe pas dans la classification de Gruschko (Gruschko et al, 2007), les auteurs considèrent ce type de changement comme cassant et non résoluble. Ainsi, pour cette catégorie d'opérateurs les procédures de migration sont définies en spécifiant explicitement les solutions alternatives pour assister le concepteur de modèles.

- L'opérateur est considéré manuel quand la migration ne peut être définie automatiquement et la solution d'adaptation doit être fournie manuellement.

4.2.5. La migration

Cette phase prend en entrée un modèle instance (ou modèle de l'utilisateur) conforme au métamodèle initial. Le scénario de migration sera appliqué sur ce modèle pour obtenir la version conforme au métamodèle évolué. Ce scénario contient des parties qui seront exécutées automatiquement et des parties où le système assistera les utilisateurs (concepteurs de modèles) pour la résolution des changements en présentant des solutions alternatives spécifiées dans le scénario de migration. L'utilisateur sélectionne à travers une interface utilisateur graphique entre différentes alternatives pour chaque élément affecté, et fournit toute information additionnelle demandée par l'alternative sélectionnée. Par exemple, si l'évolution du métamodèle inclut l'addition d'une nouvelle propriété, les solutions alternatives peuvent inclure l'initialisation de cette propriété avec une valeur prédéfinie ou avec une valeur dérivée basée sur d'autres propriétés existantes. En plus, les utilisateurs peuvent fournir des informations supplémentaires pour compléter le changement du modèle si nécessaire. Par exemple, si une nouvelle propriété doit être initialisée, l'utilisateur doit être aussi sollicité pour la valeur initiale.

Conclusion

Dans ce chapitre, nous venons de poser le cadre théorique utilisé pour la détection des différences entre deux versions du métamodèle et leur reconstruction en scénario d'évolution ainsi que la génération d'un scénario de migration pour l'adaptation des modèles instances. Avec cette démarche nous avons détaillé une étape clé dans notre approche qui consiste à l'encodage des connaissances manipulés en logique pour pouvoir appliquer un raisonnement intelligent lors de la phase de reconstruction des opérations d'évolution primitives et composites. Pour la migration des modèles nous avons procédé à une classification des opérateurs d'évolution en quatre catégories en étudiant les impacts de chaque opérateur sur les modèles instances.

CHAPITRE 5

IMPLEMENTATION ET EXPERIMENTATION

CHAPITRE 5
IMPLEMENTATION ET EXPERIMENTATION

Introduction

Dans le chapitre précédent, nous avons présenté notre approche pour la gestion de la coévolution des modèles en réponse à l'évolution de leur métamodèle.

Dans ce chapitre, nous évaluons notre travail en proposant un prototype implémenté qui sert de support à l'approche proposée. Il a pour objectif d'illustrer la faisabilité logicielle de la démarche. Ce prototype est testé sur l'exemple théorique de l'évolution du métamodèle représentant les réseaux de Petri et l'adaptation d'un modèle du réseau de Petri conforme à l'ancienne version de son métamodèle. Ceci a permis l'évaluation de la méthode d'encodage proposée ainsi que la base de connaissances et le mécanisme de reconstruction des opérations d'évolution.

5.1. Outils utilisés

Le prototype de gestion des coévolutions des modèles a été développé en se basant sur plusieurs technologies qui s'imbriquent et qui étendent d'autres technologies existantes.

- EMF (EMF, 2007) : EMF (Eclipse Modelling Framework) est un outil de méta-modélisation, il est le méta-modeleur officiel de l'environnement ECLIPSE, son avantage est la génération automatique des classes JAVA du métamodèle (grâce à l'outil GENMODEL et Ecore).
- XSD (XSD, 2013) : XSD est un langage de schéma XML, écrit en XML, qui permet de définir un type de sérialisation pour un métamodèle. La sérialisation des listes de différences et des opérations d'évolution repose sur une sérialisation définie grâce à l'outil XSD.
- EMFCOMPARE (EMFCompare, 2013) : Le projet EMFCompare fait partie d'EMF. Il est utilisé pour la différenciation des métamodèles cet outil étend l'outil COMPARE d'ECLIPSE. Il fournit un support générique et personnalisable de la comparaison et de la fusion de modèles. Il fonctionne avec n'importe quel type de modèles EMF tels que Ecore, UML ,..etc.

- Dom4j (Dom4j, 2016) : Dom4j est un framework open source pour manipuler des données XML, XSL et Xpath. Il est entièrement développé en Java. Dom4j n'est pas un parser mais propose un modèle de représentation d'un document XML et une API pour en faciliter l'utilisation. Pour obtenir une telle représentation, dom4j utilise soit SAX, soit DOM. Comme il est compatible JAXP, il est possible d'utiliser tout parser qui implémente cette API (Doudoux, 2016).
- PROLOG (Spivey, 1996) : Le langage Prolog est un Langage d'expression des connaissances fondé sur le langage des prédicats du premier ordre. C'est un langage de programmation déclarative. L''utilisateur définit une base de connaissances, l'interpréteur Prolog utilise cette base de connaissances pour répondre à des questions.

5.2. Description du Prototype

5.2.1. La phase préliminaire

Pour l'encodage automatique d'un métamodèle ou d'un modèle dans le formalisme PL, l'outil développé utilise le format XMI du métamodèle (ou du modèle) considéré. L'analyse de ce fichier permet la construction d'un ensemble de clauses PL représentant le métamodèle. Cet outil est développé sous la plateforme Eclipse en Java et utilise Dom4j pour la manipulation des fichiers XML (Dom4j, 2005). Le processus global d'encodage des métamodèles utilisé dans cet outil est représenté dans la figure (5.1).

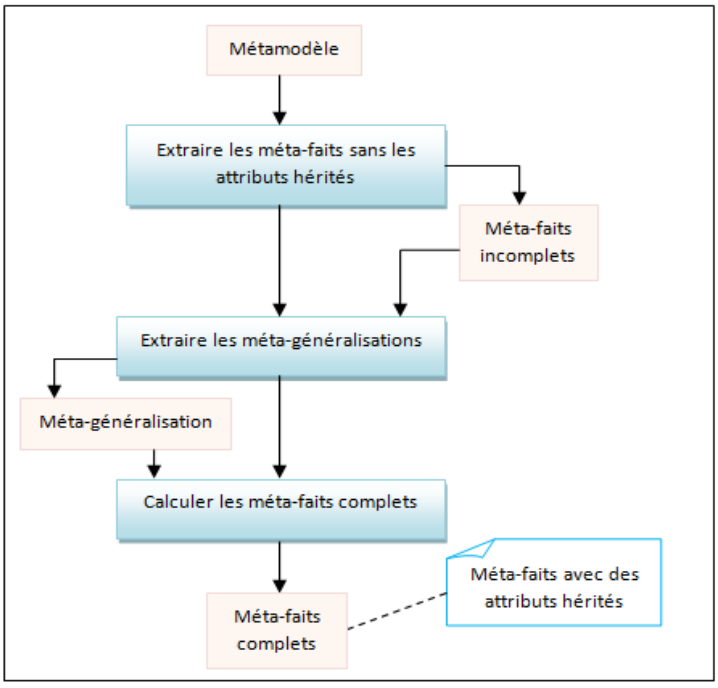

Figure 5.1. Processus d'encodage des métamodèles.

Nous prenons comme exemple un sous-ensemble du métamodèle du réseau de Petri MM2 (figure 2.2b) pour montrer les étapes d'analyse du métamodèle. Le métamodèle analysé est encodé en XMI, le listing de la figure (5.2) est un sous ensemble qui inclut des définitions de quelques classes, associations et des relations de généralisation/spécialisation. Nous notons que dans ce code les propriétés héritées ne sont pas représentées. Ainsi, pour avoir l'ensemble des attributs d'une méta-classe nous devons considérer aussi les méta-généralisations entre les méta-classes.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  <eClassifiers xsi:type="ecore:EClass" name="transition">
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="in" lowerBound="1" upperBound="-1"
        eType="#//ptarc" eOpposite="#//ptarc/dst"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="out" lowerBound="1" upperBound="-1"
        eType="#//tparc" eOpposite="#//tparc/src"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ptarc"
  eSuperTypes="#//Arc">
    <eStructuralFeatures xsi:type="ecore:EReference" name="src"
    lowerBound="1" eType="#//place" eOpposite="#//place/out"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="dst"
    eType="#//transition" eOpposite="#//transition/in"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
  name="tparc" eSuperTypes="#//Arc">
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="src" lowerBound="1" eType="#//transition"
    eOpposite="#//transition/out"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="dst" lowerBound="1" eType="#//place"
        eOpposite="#//place/in"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Arc">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="weight"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Token"/>
</ecore:EPackage>
```

Figure 5.2. Listing d'un sous ensemble du métamodèle du réseau de Petri (MM2).

Les principales étapes pour l'exécution du processus d'encodage sont :

1- Lecture du métamodèle avec l'outil Dom4j (dom4j, 2005) et récupération de l'élément racine du métamodèle. Notons que dom4j utilise un outil de lecture appelé SAX XML.

2- Construction pour chaque méta-classe le méta-fait correspondant comme suit:

- Extraction du nom du méta-fait qui correspond au nom de la classe.
- Extraction des arguments du méta-fait qui sont construits à partir des attributs de

la classe qui ne référence pas une association.

- Addition du méta-fait à l'ensemble des méta-faits incomplets. Ces méta-faits sont incomplets car les attributs hérités ne sont pas pris en compte Par exemple, l'attribut "weight" n'est pas représenté dans les classes "Ptarc" et "Tparc". En fait, cet attribut est hérité de la classe "Arc".

3- Une méthode spécifique est utilisée pour extraire les méta-généralisations afin de calculer automatiquement les méta-faits complets. Pour l'exemple précédent les méta-faits finaux calculés par cet outil sont représentés par la figure (5.3).

```
class(idc21,net, public, true). class(idc22,place, public,false).
class(idc23,transition, public,false). class(idc24,ptarc, public,false).
class(idc25,tparc, public,false). class(idc26,arc, public,false).
class(idc27,token, public, false).
property(idp21,pname, public,string, false,0,1,false, false, false,null).
property(idp22,tname, public,string, false,0,1,false, false, false,null).
property(idp23,null, public,place,false, 0 ,-1,false, true, false, null).
property(idp24,null, public,transition,false, 0,-1,false, true, false, null).
property(idp25,dst, public,place,false, 1 ,1,false, false, false,null).
property(idp26,in, public,tparc,false, 0 ,-1,false, false, false,null).
property(idp27,dst, public, transition,false, 1 ,1 ,false, false, false,null).
property(idp28,in, public, ptarc, 1 ,-1 ,false, false, false,null).
property(idp29,null,public, token, false,0,-1 ,false, true, false,null).
property(idp210,out,public, ptarc, false,0,-1 ,false, false, false,null).
property(idp211,out,public, tparc, false,1,-1 ,false, false, false,null).
property(idp212,src,public, place, false,1,1 ,false, false, false,null).
property(idp213,src,public, transition,false,1,1,false, false, false,null).
property(idp214,weight,public,int,false,0,1,false,false,false,null).
ownedAttribute(idc21, idp23). ownedAttribute(idc21, idp24).
ownedAttribute(idc22, idp21). ownedAttribute(idc22, idp26).
ownedAttribute(idc22, idp29). ownedAttribute(idc22, idp210).
ownedAttribute(idc23, idp22). ownedAttribute(idc23, idp28).
ownedAttribute(idc23, idp211). ownedAttribute(idc24, idp27).
ownedAttribute(idc24, idp212). ownedAttribute(idc25, idp213).
ownedAttribute(idc25, idp25). ownedAttribute(idc24, idp214).
ownedAttribute(idc25, idp214).
superclass(idc26,idc24). superclass(idc26,idc25). association(idp23,idass21).
association(idp24,idass22). association(idp25,idass23). association(idp26,idass24).
association(idp27,idass25). association(idp28,idass26).
association(idp29,idass27). association(idp210,idass28).
association(idp211,idass29). association(idp212,idass210).
association(idp213idass211). opposite(idp25,idp26).
opposite(idp27,idp28). opposite(idp10,idp12). opposite(idp11,idp13).
```

Figure 5.3. Listing des méta-faits du métamodèle du réseau de Petri (MM2).

5.2.2. La phase détection des changements

Pour la comparaison et le calcul de différences entre deux versions d'un métamodèle nous avons utilisé dans notre solution le plug-in d'Eclipse EMFCompare (EMF, 2015b). Cet outil fournit des outils pour le calcul du modèle delta qui représente l'ensemble de différences détectées entre deux modèles, et le visualise sous forme arborescente. Le moteur d'EMFCompare représente les différences comme des changements primitifs. Chacune inclut essentiellement des informations sur le type de changement, L'élément qui a subit le changement ainsi que quelques paramètres d'évolution. En plus, avec chaque changement élémentaire sont fournies deux listes qui la sont pré-liste et la post-liste. La première contient un ensemble de changements qui doivent être réalisés avant le changement en question, cette liste est appelée « required list ». Cet ensemble constitue les pré-conditions

pour exécuter un changement spécifique. La seconde liste appelée “required by list” dans laquelle sont regroupés les changements qui nécessite la réalisation du changement considéré avant eux. EMFcompare est capable de détecter les types de changements suivants : ajout, suppression, modification et déplacement.

Le modèle delta est traduit dans le format XMI et passé en entrée à l’outil d’encodage. Des méthodes spécifiques sont utilisées pour extraire automatiquement les types des changements et leurs paramètres et les traduire en faits. La figure (5.4) montre le modèle de différence calculé avec EMFcompare (EMF, 2015b) en utilisant les deux versions du métamodèle du réseau de Petri MM1 (figure 2.2.a) et MM2 (figure 2.2.b). Ce modèle est ensuite transformé en PL par l’outil développé comme expliqué ci-dessus ce qui résulte les faits représentés en figure (5.5).

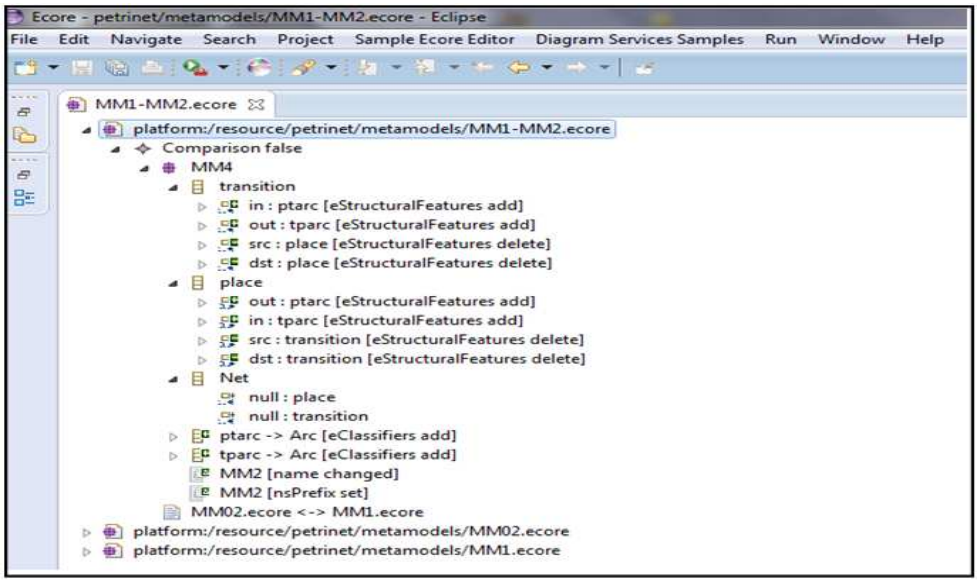

Figure 5.4. Le modèle delta des deux versions du métamodèle du réseau de Petri.

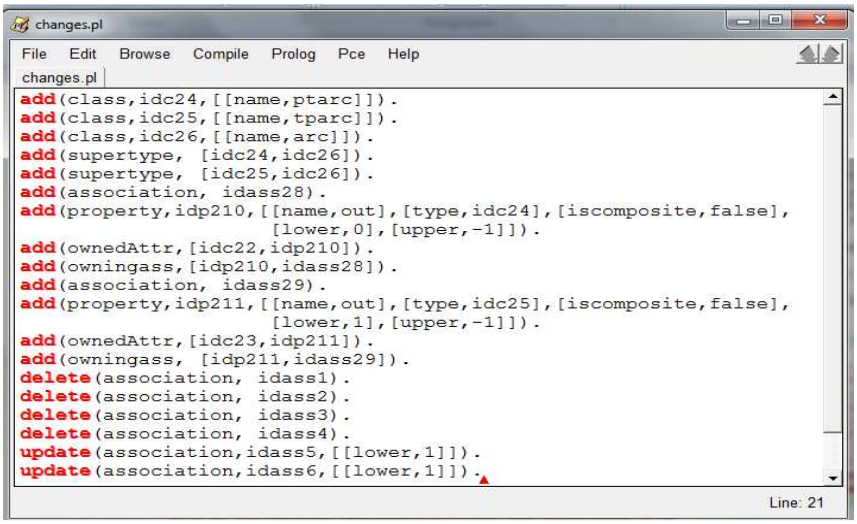

Figure 5.5. Extrait des faits représentant les changements entre les deux versions du métamodèle du réseau de Petri.

5.2.3. La phase reconstruction du scénario d'évolution

La première étape de cette phase consiste à convertir les différences en opérations atomiques conformes à la définition des opérateurs définis dans la bibliothèque des opérateurs. Dans ce but, nous utilisons un moteur d'inférence PL avec une base de règles afin d'obtenir pour chaque changement atomique dans le modèle delta, l'opérateur d'évolution correspondant. Des exemples des règles d'inférences utilisées pour la déduction des opérations d'évolution sont présentés dans le listing suivant :

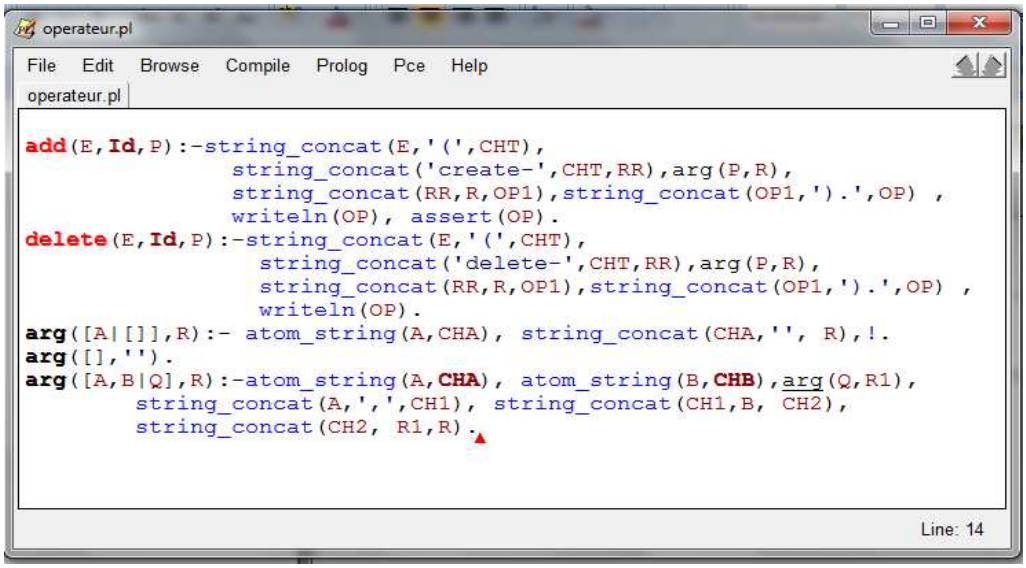

Figure 5.6. Exemples de règles pour traduire les changements en opérations d'évolution.

Dans l'exemple précédent, en utilisant le moteur d'inférence Prolog avec la base des règles et la base de faits qui contient les faits des changements appliqués sur le métamodèle des réseaux de Petri, nous déduisons un ensemble d'opérations d'évolution primitives comme illustré par la figure (5.7).

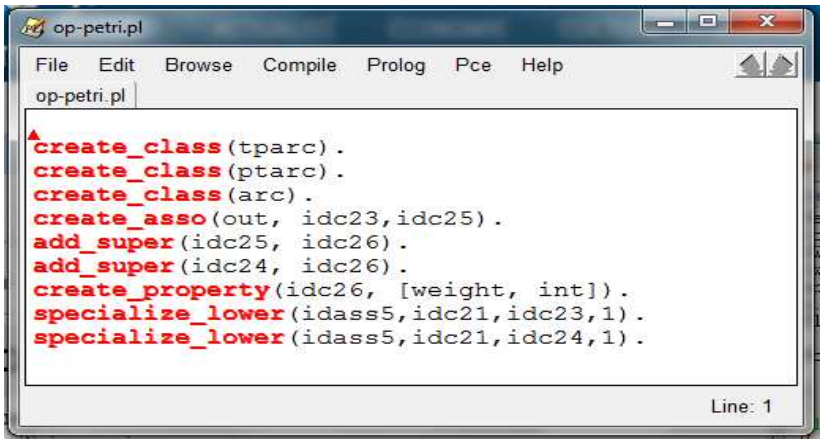

Figure 5.7. Un extrait des opérations d'évolution primitives déduites du modèle delta entre les deux versions du métamodèle des réseaux de Petri.

En effet, le scénario d'évolution est stocké comme une base de faits où chaque fait

représente une instance d'un opérateur d'évolution. Ensuite, les opérations d'évolution seront réorganisées car ces différences ne sont pas toujours présentées dans le bon ordre ce qui rend invalide le scénario d'évolution. Ainsi, nous utilisons les pré-conditions et les post-conditions pour définir les dépendances entre les opérations ce qui permet de réorganiser le scénario d'évolution primitif. Finalement cette étape vérifie si le scénario d'évolution est valide, en appliquant le scénario résultant sur l'ancienne version du métamodèle, s'il résulte la nouvelle version du métamodèle alors le scénario d'évolution primitif est bien ordonné sinon le processus de réorganisation est répété.

La base de faits regroupant les opérations atomiques d'évolution du métamodèle du réseau de Petri représente le scénario d'évolution primitif. Cette base de faits constitue une nouvelle entrée au moteur d'inférence Prolog pour la déduction des éventuelles opérations composites en utilisant la base de connaissances définissant les opérateurs d'évolution composites.

Dans l'exemple précédent le fait illustré par (5.1) peut être déduit. Ce fait indique qu'une nouvelle classe ayant l'identificateur "idc26" est créée et les deux classes avec les identificateurs "idc24" et "idc25" prennent la classe "idc26"comme leur super classe.

Ainsi, nous pouvons voir que l'opération d'évolution composite "extract superclass" dans (5.1) contient trois opérations primitives (une « create class » et deux « create super class ») illustrées par (5.2).

extract_superclass (idc26, [idc24,idc25]) (5.1)

Create-class(arc,public,true).
add_super(Idc24, Idc26). (5.2)
add_super(Idc25,Idc26).

La valeur de "idc26" est obtenu après la création de la classe dont le nom est « arc » et application de la fonction « getid(arc, Idc). » qui permet de récupérer l'identificateur de la classe.

5.2.4. Génération du scenario de migration et migration des modèles

A partir du scénario d'évolution valide, nous prenons les procédures de migration définies avec les opérateurs et nous les exécutons sur un réseau de Petri spécifique représenté par un modèle équivalent (M1) conforme à (MM1) comme représenté sur les figures (5.8, 5.9). Pour adapter le modèle M1 avec la nouvelle version du métamodèle (MM2), des nouvelles instances d'objets sont créées selon les nouvelles méta-classes (*TParc, PTarc, Token*) ainsi que les nouvelles associations et attributs. En plus, quelques instances seront supprimées. Durant cette étape, les valeurs de marquage des jetons (propriété "mark" de la

classe « Token ») seront demandées de l'utilisateur par contre les valeurs de la propriété « weight » des instances de la classe « Arc » (les poids des arcs) prennent la valeur par défaut. Par exemple, les instances des "places" P1, P2, P3 et P4 seront marqués respectivement par les valeurs (4,2,1,0) . Tandis que les différents arcs auront un poids à un en initialisant la propriété « weight » avec la valeur par défaut qui vaut 1. Le résultat d'adaptation du modèle précédent à la nouvelle version du métamodèle est représenté par la figure (5.10).

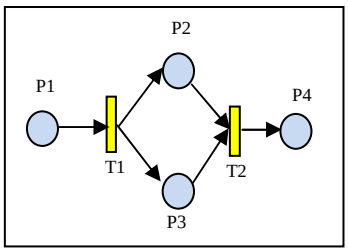

Figure 5.8. Réseau de Petri simple (wachsmuth, 2007).

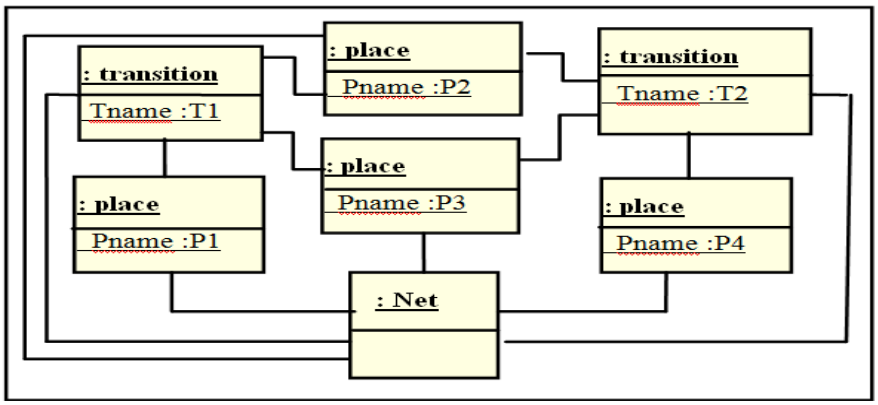

Figure 5.9. Modèle d'un réseau de Petri simple.

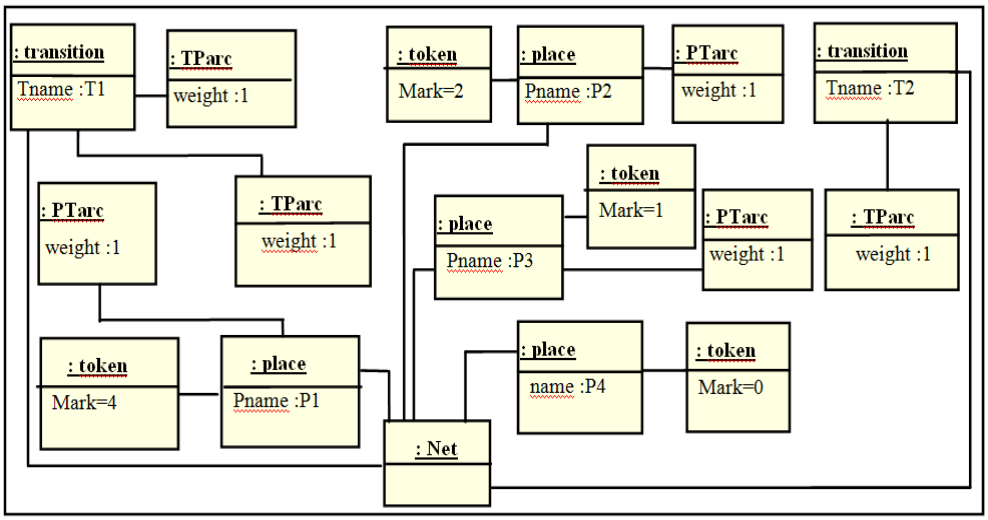

Figure 5.10. Nouveau modèle instance M2 conforme au métamodèle évolué MM2

Conclusion

Dans ce chapitre, nous avons présenté l'évaluation de l'approche proposée pour la coévolution des modèles. Elle a été réalisée sur deux versions du métamodèle représentant le réseau de Petri. D'abord la phase d'encodage a été expérimentée puis les deux versions du métamodèle ont été comparées pour déduire le modèle delta qui a été exploité par le moteur d'inférence Prolog pour inférer les opérations d'évolution atomiques et composites. Sur leur base les procédures de migration sont générées selon la classification accordée à l'opération d'évolution en question. L'application de ces procédures sur un modèle instance représentant un réseau de Petri conforme à la première version du métamodèle a permis d'obtenir un modèle conformant à la deuxième version du métamodèle ce qui concrétise l'approche proposée.

CONCLUSION GENERALE

CONCLUSION GENERALE

Dans cette thèse nous avons soulevé le défi d'automatisation de la coévolution des modèles et des métamodèles, pour cela nous avons proposé une solution alternative à ce problème. Ainsi notre proposition illustre une approche hybride pour guider les concepteurs des modèles à résoudre les problèmes de coévolution. Elle tire profit des avantages des deux classes d'approches : basées états et basées opérateurs en ajoutant une nouvelle dimension pour traiter ce problème qui est l'utilisation d'un raisonnement intelligent. Le processus de résolution du problème de coévolution des modèles repose sur trois principaux axes qui sont :

1. La détection des changements entre deux versions d'un métamodèle.
2. La reconstruction des scénarios d'évolution et de migration
3. La migration des modèles

La solution proposée est indépendante de tout langage de modélisation. Elle consiste à utiliser une bibliothèque d'opérateurs mais à la différence des approches existantes les changements nécessitant l'intervention des utilisateurs sont également intégrés dans la bibliothèque avec des solutions alternatives afin d'assister automatiquement les utilisateurs durant l'activité de migration. En plus, les avantages de cette approche sont nombreux :

- L'encodage des métamodèles avec le langage des prédicats.
- Formalisation des opérateurs d'évolution pour permettre le raisonnement et faciliter toute éventuelle extension de la librairie des opérateurs.
- Le mécanisme du raisonnement intelligent utilisé pour la reconstruction du scénario d'évolution et le noyau du processus de coévolution, il assure une réelle séparation entre l'évolution du métamodèle et la migration des modèles, et par conséquent entre le concepteur du métamodèle et le concepteur du modèle.
- L'utilisation du mécanisme logique intelligent pour inférer les opérateurs d'évolution composites augmente l'efficacité de notre proposition et rend cette solution différente des travaux existants.

Pour l'évaluation de notre proposition nous avons développé un prototype pour tester le noyau de l'approche. Ceci nous a permis de tester en premier lieu le processus d'encodage

sur lequel repose les quatre phases de l'approche. Nous avons également testé le mécanisme de reconstruction de l'évolution sur lequel repose la définition de la migration en question. Au cours de l'évaluation de notre approche, nous avons utilisé l'exemple du scénario d'évolution du métamodèle des réseaux de Petri.

Les résultats de validation sont prometteurs et prouvent la faisabilité de l'approche proposée. Nous concluons donc que Le processus d'encodage en PL ainsi que la spécification des opérateurs sont satisfaisantes et améliorent la compréhension de l'évolution du métamodèle par le concepteur de modèle.

Perspectives de recherche

En arrivant à ces résultats nous visons quelques perspectives pour ce travail de recherche qui sont principalement :

- L'implémentation complète des opérateurs d'évolution ainsi que la validation de l'approche sur un cas d'étude réel.
- Factorisation des opérateurs pour créer des classes d'opérateurs selon les types des éléments partageant des caractéristiques en commun afin de réduire la taille de la bibliothèque et permettre un accès plus rapide aux connaissances.
- En plus, nous pensons à l'extension de l'approche pour supporter la coévolution d'autres artefacts dépendants du même métamodèle tel que les règles de transformation. Et comme les approches actuelles, ainsi que la notre se limitent à la prise en compte de la coévolution syntaxique des modèles nous voulons préserver également la sémantique des modèles en utilisant des ontologies comme introduit dans (Cicchetti and Ciccozzi, 2013).

REFERENCES

Références

(Anguel et al, 2015a) F. Anguel, A. Amirat, N. .Bounour. Comparison Study of Metamodels and Models Co-Evolution Approaches . in *Symposium on Complex Systems and Intelligent Computing (CompSIC), Avril 2015 SoukAhras, Algeria* . http://www.univ-soukahras.dz/en/publication/article/413

(Anguel et al, 2015b) F. Anguel, A. Amirat, N. .Bounour . Hybrid Approach for Metamodel and Model Co-evolution, 5th IFIP International Conference, (CIIA 2015), Saida, Algeria, May 20-21 2015, Proceedings. IFIP Advances in Information and Communication Technology 456, Springer 2015, ISBN 978-3-319-19577-3, pp 563-573.

(Anguel et al, 2016) F. Anguel, A. Amirat, N. .Bounour . (In press) Using Logic programming for adapting models to metamodel evolution, International Journal of Intelligent Information and Data base Systems , 2016 , (ISSN online: 1751-5866).

(Baral et Gelfond, 1994) Baral, C., Gelfond, M.). Logic programming and knowledge representation, Journal of Logic Programming. 1994, 19-20, pp.73–148.

(Becker et al, 2007) Becker, S., Goldschmidt, T., Gruschko, B., and Koziolek, H. A process model and classification scheme for semi-automatic meta-model evolution. In Proc. 1st Workshop MDD, SOA und IT-Management (MSI'07), 2007, pp 35–46. GiTO-Verlag.

(Bézivin et al, 2003) Jean Bézivin, Mireille Blay, Mokrane Bouzhegoub, Jacky Estublier, JeanMarie Favre, Sébastien Gérard, Jean Marc Jézéquel. Rapport de Synthèse de l'AS CNRS sur le MDA (Model Driven Architecture), 2003.

(Bézivin , 2005) Bézivin, J. On the Unification Power of Models, Journal of Software and systems Modeling (SoSyM), vol. 4(2), pp. 171–188, 2005.

(Bézivin et al, 2001) Bézivin, J. and Gerbé, O. Towards a Precise Definition of the OMG/MDA Framework. In Proc. International Conference on Automated Software Engineering (ASE), pages 273-280. IEEE Computer Society, 2001.

(Bézivin et al, 2001) Bézivin, J. and Gerbé, O. Towards a Precise Definition of the OMG/MDA Framework. In Proc. International Conference on Automated Software Engineering (ASE), pages 273-280. IEEE Computer Society, 2001.

(Bézivin et al, 2003) J.Bézivin, S.Gérard, PA.MullerL.Rioux. MDA components : Challenges and opportunities dans le proceeding 1st international workshop , York,UK, November 2003, pp 23-41.

(Bézivin et Gerbé, 2001) J. Bézivin , O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In Proc Automated Software Engineering (ASE'01), IEEE Computer Society Press, San Diego, USA. 2001.

(Bézivin et Heckel, 2006) Bézivin, J. et Heckel, R. Guest editorial to the special issue on language engineering for model-driven software development. Software and Systems Modeling, 5(3) 2006, pp 231–232.

(Bézivin, 2004) Jean Bézivin. In search of a Basic Principle for Model-Driven Engineering. Novatica – Special Issue on UML (Unified Modeling Language), 5(2), 2004.

(Bézivin, 2004b) Bézivin J. Sur les principes de base de l'ingénierie des modèles, L'Objet,10(4):145–157, 2004. DOI: 10.3166/objet.10.4.145-157

(Booch et al, 2007) Grady Booch Robert A. Maksimchuk Michael W. Engle Bobbi J. Young, Jim Conallen Kelli A. Houston Object-Oriented Analysis and Design with Applications Third Edition The Addison-Wesley Object Technology Series, Grady Booch, Ivar Jacobson, and James Rumbaugh, Series Editors, 2007.

(Brèche, 1996) Brèche, Advanced Primitives for Changing Schemas of Object Databases. In Conference on Advanced Information Systems Engineering, pp.476-495, 1996.

(Burger et al, 2007) Burger, E. et Gruschko, B. A change metamodel for the evolution of MOF based metamodels. Modellieurung, pp 285-300, 2010.

(Cicchetti et al, 2008) A., Cicchetti, D.Di., Ruscio, R., Eramo and A., Pierantonio, Automating co-evolution in MDE. In Proc. EDOC'08 IEEE Computer Society. pp 222-231, 2008.

(Cicchetti et al, 2009) Cicchetti, A., Ruscio, D. D., and Pierantonio, A. Managing dependent changes in coupled evolution. In ICMT2009 - International Conference on Model Transformation. Springer LNCS, 2009.

(Clamen, 1992) Clamen, S. Type evolution and instance adaptation. Technical Report CMU-CS-92-133, Carnegie Mellon University, (1992).

(Claypool et al, 1999) Claypool, K., Rundensteiner, E., and Heineman, G. Extending Schema Evolution To Handle Object Models with Relationships. Technical Report, WPI-CS-99-15, Worcester Polytechnic Institute, 1999.

(Czarnecki et Eisenecker, 2000) Czarnecki, K. et Helsen, S. Feature-based survey of model transformation approaches. IBM Syst. J., 45(3) pp 621–645, 2006.

(Deursen et al, 2000) A. van Deursen, P. Klint, and J. Visser. Domain-specific languages : An annotated bibliography. *ACM SIGPLAN Notices*, 35(6) :26–36, juin 2000.

(Deursen et al, 2007) A. van Deursen, E. Visser, and J. Warmer. Model-driven software evolution: A research agenda. In Proc. Workshop on Model-Driven Software Evolution, co-located with the European Conference on Software Maintenance and Reengineering (CSMR), pages 41{49, 2007.

(Diaw et al, 2010) Samba Diaw, Redouane Lbath, Bernard Coulette. «Etat de l'art sur le développement logiciel basé sur les transformations de modèles ». Dans : Technique et Science Informatiques, Hermès Science , Numéro spécial Ingénierie Dirigée par les Modèles, Vol. 29, N. 4-5, p. 505-536, juin 2010

(Dom4j, 2016) dom4j. Disponible en ligne. URL :http ://www.dom4j.org/ (consulté mars 2016)

(Doudoux, 2016) Doudoux J.M. Cours Java et Eclipse, les modèles de documents. Disponible en ligne : www.w3c.org/DOM, (consulté mars 2016).

(Ecore, 2013) Ecore. Disponible en ligne : http ://www.eclipse.org/ecore, (consulté juin 2013).

(EMF, 2007) EMF. Eclipse Modeling Framework. Reference site: http://www.eclipse.org/emf, (consulté janvier 2013)

(EMFCompare, 2013) EMFCompare. Disponible en ligne : www.eclipse.org/emfcompare, (consulté janvier 2013).

(Estublier et al, 2005) J. Estublier, J-M. Favre, J. Bézivin, L. Duchien, R. Marvie, S. Gérard, B.

Baudry M. Bouzhegoub, J-M. Jézéquel, M. Blay, and M. Riveil. Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles. Rapport de synthèse 1.1.2, CNRS, janvier 2005.

(Fages, 1996) F. Fages. Constraint logic programming. Cours de l'Ecole Polytechnique. Ellipses, Paris, 1996.

(Favre et al, 2006) Jean-Marie Favre, Jacky Estublier, Mireille Blay-Fornarino L'ingénierie dirigée par les modèles, au-delà du MDA. Editor, Hermes Science publications – Lavoisier

(Favre et al, 2006) Jean-Marie Favre, Jacky Estublier, Mireille Blay-Fornarino L'ingénierie dirigée par les modèles au-delà du MDA sous la direction de Hermes Science publications Lavoisier, 2006

(Favre, 2004) Jean-Marie Favre. Foundations of Meta-Pyramids : Languages vs. Metamodels - Episode II : Story of Thotus the Baboon1. In Language Engineering for Model-Driven Software Development, 2004.

(Ferrandina et Lautemann, 1996) Ferrandina, F. and Lautemann, S. An Integrated Approach to Schema Evolution for Object Databases. In OOIS, pp.280-294, 1996.

(Fleurey , 2006) Fleurey, F. Langage et méthode pour une ingénierie des modèles fiable, thèse de doctorat de l'Université de Rennes 1, 218 p, 2006.

(Fowler, 2005) Fowler, M. (2005). Language workbenches: The killer-app for domain specific languages? http://martinfowler.com/articles/languageWorkbench.html.

(France et Rumpe, 2007) France, R. et Rumpe, B. (2007). Model-driven development of complex software: A research roadmap.Dans FOSE '07: Future of Software Engineering, pp 37–54, USA. IEEE Computer Society, 2007.

(Garcès et al, 2009b) K., Garcès, F., Jouault, P., Cointe and J., Bézivin. Managing Model Adaptation by Precise Detection of Metamodel Changes. In Proc ECMDA-FA'09. LNCS Springer, vol. 5562. pp 34-49, 2009.

(Garcès et al, 2009a) K., Garcès, F., Jouault, P., Cointe and J., Bézivin. A Domain Specific Language for Expressing Model Matching. In Proc. IDM09 ,5ième journée sur l'ingénier dirigée par les modèles, Nancy 25-26 mars 2009 .

(Garlan et al, 1994) Garlan, D., Krueger, C., and Lerner, B. TransformGen : Automating the Maintenance of Structureoriented Environments. ACM Trans. Program. Lang. Syst., volume 16, pp.727-774, 1994.

(GMF, 2013) GMF. Graphical Modeling Framework. Disponible en ligne : www.eclipse.org, http://www.eclipse.org/modeling/gmp/downloads/?project=gmf-runtime (consulté janvier 2013)

(Greenfield et al, 2004) J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley ; 1st edition, 2004. *ISBN : 0471202843*.

(Gruschko et al, 2007) B., Gruschko, D.S., Kolovos and R.F, Paige, Towards synchronizing models with evolving metamodels. In Proc. The International Workshop on Model-Driven Software Evolution, 2007.

(Harman , 2012) Mark Harman (2012).The Role of Artificial Intelligence in Software Engineering. In The International Workshop on Realizing AI Synergies in Software Engineering (RAISE'12), Zurich, Switzerland, 2012.

(Herrmannsdoerfer et al, 2008a) Herrmannsdoerfer, M., Benz, S., and Juergens, E. Automatability of coupled evolution of metamodels and models in practice. In Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., and Volter, M., editors, Model Driven Engineering Languages and Systems (MODELS 2008), volume 5301/2008 of Lecture Notes in Computer Science, pages 645–659. Springer Berlin / Heidelberg, 2008.

(Herrmannsdoerfer et al, 2008b) Herrmannsdoerfer, M., Benz, S., and Juergens, E. COPE: A language for the coupled evolution of metamodels and models. In 1st International Workshop on Model Co-Evolution and Consistency Management, 2008.

(Herrmannsdoerfer et al, 2009a) Herrmannsdoerfer, M., Benz, S., and Juergens, E. COPE - automating coupled evolution of metamodels and models. In ECOOP 2009 - Object-Oriented Programming, volume 5653 of Lecture Notes in Computer Science, pages 52–76. Springer Berlin / Heidelberg. 2009.

(Herrmannsdoerfer et al., 2010b) Herrmannsdoerfer, M., Vermolen, S., et Wachsmuth, G. An extensive catalog of operators for the coupled evolution of metamodels and models. In Malloy, B., Staab, S., and van den Brand, M., editors, Software Language Engineering, volume 6563 of Lecture Notes in Computer Science, pages 163–182. Springer Berlin / Heidelberg, 2010.

(Herrmannsdoerfer, 2010a) Herrmannsdoerfer, M. Migrating UML activity models with COPE. In Transformation Tool Contest, 2010.

(Herrmannsdoerfer, 2011a) Herrmannsdoerfer, M. COPE - a workbench for the coupled evolution of metamodels and models. In Malloy, B., Staab, S., and van den Brand, M., editors, Software Language Engineering, volume 6563 of Lecture Notes in Computer Science, pages 286–295. Springer Berlin / Heidelberg, 2011.

(Herrmannsdorfer, 2011b) Herrmannsdorfer, M. Evolutionary Metamodeling, PHD Thesis, Université Technique De Munich, 303p , 2011.

(ISO/IEC, 2006) ISO/IEC. Ingénierie du logiciel : Processus du cycle de vie du logiciel : Maintenance, 14764 :2006. Disponible en ligne : www.iso.org, 2006 , (consulté juin 2015).

(JDOM, 2013) JDOM. Java Document Object Model. Disponible en ligne : www.jdom.org, (consulté janvier 2016).

(Jouault et Kurtev, 2005) F. Jouault and I. Kurtev. Transforming models with ATL. In Proc. Satellite Events at MoDELS. LNCS Springer, vol. 3844, pp. 128-138, 2005.

(Karsai et al, 2003) Gabor Karsai, Janos Sztipanovits, Ákos Lédeczi, et Ted Bapty. Model integrated development of embedded software. *Proceedings of the IEEE*, 91(1) :145–164, 2003.

(Kelly et Tolvanen, 2007) Kelly, S. et Tolvanen, J.-P. Domain-Specific Modeling. (Eds) John Wiley & Sons, 2007 .

(Kim et al, 1987) Kim, H. and Korth, K. PSYCHO : A graphical langage for supporting Schema Evolution in Object-Oriented Databases, Technical Report, 1987.

(Kleppe et al, 2003) Kleppe, A. G., Warmer, J., et Bast, W. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co.,Inc., Boston, MA, USA, 2003.

(Kleppe, 2007) Anneke G. Kleppe. A language description is more than a metamodel. In Fourth

International Workshop on Software Language Engineering, Grenoble, France, 2007.

(Kurtev et al, 2002) Kurtev, I., Bézivin, J., and Aksit, M. Technological spaces: An initial appraisal. In CoopIS, DOA Federated Conferences, Industrial track. 2002.

(Kurtev, 2004) Kurtev, Ivan. Adaptability of Model Transformations. PhD thesis, University of Twente, Netherlands, 2004.

(Lämmel, 2001) Lämmel, R. Grammar adaptation. In FME 2001: Formal Methods for Increasing Software Productivity, volume 2021/2001 of Lecture Notes in Computer Science, pages 550–570. Springer Berlin / Heidelberg. 2001.

(Lehman, 1980) Lehman, M. M. "Programs, life cycles and laws of software evolution". Dans Proceeding of the IEEE, volume 68, pp.1060-1076, (septembre 1980).

(Lerner, 1990) Lerner, B. and Habermann, A. Beyond schema evolution to database reorganization. In : OOPSLA,pp.67-76, (1990).

(Levendovszky et al, 2010] Levendovszky, T., Balasubramanian, D., Narayanan, A., and Karsai, G. A novel approach to semi-automated evolution of DSML model transformation. In Software Language Engineering, volume 5969 of Lecture Notes in Computer Science, pages 23–41. Springer Berlin / Heidelberg, 2010.

(Malgouyres, 2006) Hugues Malgouyres. Définition et détection automatique des incohérences structurelles et comportementales des modèles UML - Couplage des techniques de métamodélisation et de vérification basée sur la programmation logique PHD thesis, Institut National des Sciences Appliquées de Toulouse, 151p, 2006.

(Mens et al, 2005) Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., et Jazayeri, M. Challenges in software evolution. In 8th International Workshop on Principles of Software Evolution (IWPSE), pages 13–22, 2005.

(MOF, 2005) OMG, MOF QVT Final Adopted Specification," Available: www.omg.org/docs/ptc/05-11-01.pdf , (consulté janvier 2013).

(Narayanan et al, 2009) A. Narayanan, T. Levendovszky, D. Balasubramanian and G. Karsai, "Automatic domain model migration to manage metamodel evolution," in Proc. MODELS'09, 2009, LNCS Springer, vol. 5795, pp. 706-711, 2009.

(OMG, 2003) Object Management Group (2003). Model Driven Architecture (MDA) guide version 1.0.1. http://www.omg.org/cgi-bin/doc?omg/ 03-06-01.pdf, (consulté janvier 2013).

(OMG, 2005) Object Management Group. MOF 2.0/XMI Mapping Specification, v2.1, URL http ://www.omg.org/cgi-bin/doc ?formal/2005-09-01, September 2005, (consulté janvier 2013).

(OMG, 2009) Object Management Group. Unified Modeling Language (UML) superstructure version 2.2. http://www.omg.org/spec/UML/2.2/, 2009, (consulté janvier 2013)

(OMG, 2011a) Object Management Group. MOF 2.4 Specification. Technical report, Disponible en ligne http://www.omg.org/spec/MOF/2.4/, 2011, (consulté janvier 2013).

(OMG, 2011b) Object Management Group. OMG Query/View/Transformation (OMG QVT), V1.1. Technical report,. http://www.omg.org/spec/QVT/1.1/PDF/. , 2011, (consulté janvier 2013)

(OMG, 2012) Object Management Group. OMG Object Constraint Language (OMG OCL), V2.3.1. Technical report, http://www.omg.org/spec/OCL/2.3.1/PDF/, 2012, (consulté janvier 2013)

(OMG, 2013) Object Management Group. OMG XML Metadata Interchange (OMG XMI), V2.4.1.Technical report, 2013. http://www.omg.org/spec/XMI/2.4.1/PDF/. (consulté janvier 2016)

(OMG, 2014) OMG MDA Guide rev. 2.0 June 2014, (consulté janvier 2016).

(OMG, 2015) OMG. Unified Modeling Language (OMG UML) technical report 2015 http://www.omg.org/spec/UML/2.5 . (consulté janvier 2016).

(Penney et al, 1987) Penney, D. J. and Stein, J. Class Modification in the GemStone ObjectOriented DBMS. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 111-117, 1987.

(Rose et al, 2009) L.M. Rose, D.S. Kolovos, R.F. Paige, and F.A.C. Polack. An analysis of approaches to model migration. In Proc. Joint MoDSEMCCM Workshop, 2009.

(Rose et al, 2010) L.M. Rose, D.S.Kolovos, R.F.Paige and F.A.C. Polack. Model migration with Epsilon Flock. In Proc ICMT'10, LNCS Springer, vol. 6142, pp. 184-198, 2010.

(Rose, 2011) Louis Mathew Rose, Structures and Processes for Managing Model-Metamodel Co-evolution, these de doctorat, université de York, juillet 2011, 306p.

(Rundensteiner, 1992) Rundensteiner, E. A MultiView : A Methodology for Supporting Multiple View Schemata in OODBs. Proc. of the 18th VLDB Conf., pp.187-198, 1992.

(Savoy, 2006) Jacques Savoy. Introduction à la programmation logique Prolog, http://members.unine.ch/ jacques.savoy /lectures/SemCL/Prolog.pdf. (consulté janvier 2013).

(Selic, 2003) Selic, B. The pragmatics of Model Driven Development. IEEE software,20(5),19-25, (2003).

(Sjoberg, 1993) D.I.K. Sjoberg. Quantifying schema evolution. Information & Software Technology, 35(1):35{44, (1993).

(Skarra et Zdonik , 1986) Skarra, A. and Zdonik, S. The Management of Changing Types in OODB. In Proc of the OOPSLA Conference, (1986).

(Soley, 2000) Richard Soley. Model driven architecture. Technical report, Object Management Group (OMG), (2000).

(Spivey, 1996) Michael Spivey. An introduction to logic programming through Prolog. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

(Sprinkle et Karsai, 2004) J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. Journal of Visual Languages and Computing, 15(3-4):291{307, 2004.

(Sprinkle, 2003) J. Sprinkle. Metamodel driven model migration. Phd. thesis, Vanderbilt University, 2003.

(Su et al, 2001) Su, H., Kramer, D., Chen, L., Claypool, K. T., and Rundensteiner, E. A. XEM: Managing the evolution of XML documents. In Eleventh InternationalWorkshop on Research Issues in Data Engineering on Document Management for Data Intensive Business and Scientific Applications, pages 103–110, Washington, DC, USA. IEEE Computer Society, 2001.

(Tan et Goh, 2005) Tan, M. and Goh, A. Keeping pace with evolving XML-Based specifications. In Current Trends in Database Technology - EDBT 2004 Workshops, volume 3268 of Lecture Notes in Computer Science, pages 280–288. Springer Berlin / Heidelberg,

2005.

(Topcased, 2013) Topcased. Toolkit in Open-source for Critical Apllications and Systems Development. Disponible en ligne : http ://www.topcased.org, 2013, (consulté juin 2013)

(Tresch et Scholl, 1992) Tresch, M. and Scholl, M. Meta Object management and its application to database evolution. Proc. Of the 11th Int., Entity Approach Conference., pp.299-321, 1992.

(UML, 1997) UML, Unified Modeling Language, http://www.uml.org, 1997, (consulté janvier 2013).

(W3C, 2008) Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C. http://www.w3.org/TR/REC-xml/, 2008.

(Wachsmuth, 2007) G. Wachsmuth, Metamodel adaptation and model co-adaptation. In Proc. ECOOP'07.LNCS Springer, vol. 4609, pp. 600-624, 2007.

(Wiképédia, 2015) Wiképédia . Système. disponible sur https://fr.wikipedia.org/wiki/Syst%C3%A8me, 2015, (consulté janvier 2016).

(XSD, 2013) XSD. Introduction to the XSD Editor. Disponible en ligne : http ://wiki.eclipse.org/ Introduction to the XSD Editor, 2013.

A propos de l'auteur

Biographie

Fouzia Anguel épouse Labar a obtenu son baccalauréat en mathématiques en 1991. Elle rejoignit l'université de Badji Mokhtar – Annaba (UBMA) la même année, pour suivre une formation en informatique et obtenir le diplôme d'ingénieur option système d'information en 1996. Elle obtient son Magistère option Intelligence artificielle en 2004. Depuis 2011, elle a commencé de préparer sa thèse sous la direction du Pr Abdelkrim Amirat et Dr Nora Bounour, afin d'obtenir le diplôme de Doctorat en sciences. Actuellement, elle est enseignante (Maître assistant classe A) à l'université Chadli Bendjedid –El Tarf et membre du Laboratoire d'Ingénierie des Systèmes COmplexes LISCO, qui s'active autour les thèmes de l'évolution et de la réutilisation du logiciel. Ses intérêts de recherche comprennent: l'ingénierie dirigée par les modèles, la méta-modélisation et l'évolution des modèles.

Courriel: fanguel@yahoo.fr

Adresse:

Département d'Informatique, Université Chadli Bendjedid , BP 73,36000, El Tarf.

Laboratoire LISCO, Département d'informatique, Université Badji Mokhtar, BP 12, 23000 Annaba.

Publications Internationales

Fouzia Anguel, Abdelkrim Amirat , Nora Bounour , (2016) **(In press)** "Using Logic programming for adapting models to metamodel evolution" , International Journal of Intelligent Information and Data base Systems , (ISSN online: 1751-5866).

Communications Internationales

- ANGUEL .F, AMIRAT .A, BOUNOUR .N, «Hybrid Approach for Metamodel and Model Co-evolution», 5th IFIP TC 5 International Conference, (**CIIA 2015)**, Saida, Algeria, May 20-21, 2015, Proceedings. IFIP Advances in Information and Communication. Technology 456, Springer 2015, ISBN 978-3-319-19577-3 , PP 563-573.

- ANGUEL .F, AMIRAT .A, BOUNOUR .N, «Using Weaving Models in Metamodel and Model Co-Evolution Approach», In proceeding of 6th International Conference on computer

science and information technology **(CSIT 2014)** ISBN:987-1-4799-3999-2 IEEE Computer Society. ,Amman, Jordan 26-27 March 2014, PP 142-147.

- ANGUEL .F, AMIRAT .A, BOUNOUR .N, «Towards models and metamodels co-evolution approach »,In proceeding of 11th International Symposium of Programming and Systems **(ISPS 2013)**, 22-24 April 2013, PP 163-167.

Communications Nationales

- ANGUEL .F, AMIRAT .A, BOUNOUR .N, «Towards automatic models and metamodels co-evolution approach», In 3ième journées Doctorales en Informatique **(JDI'2013)**, Guelma 3-4 décembre 2013, PP 18-24.

- ANGUEL .F, AMIRAT .A, BOUNOUR .N, «Comparison Study of Metamodels and Models Co-Evolution Approaches», In proceeding of Symposium on Complex Systems and Intelligent Computing (**CompSIC**) Souk-Ahras, Algéria, avril 2015.