Intégration de la transformation de modèles dans un processus de développement itératif

Mikaël Barbero

LINA - Université de Nantes

Sodifrance

Encadrant LINA : Jean Bézivin
Encadrant Sodifrance : Erwan Breton

Mémoire de stage

Master de Recherche

Architectures Logicielles Distribuées

16 mai 2006

Table des matières

Liste des acronymes **v**

1 Introduction **1**

1.1 Motivations et objectifs 1

1.1.1 Les chaînes de transformations 2

1.1.2 Problématique 3

1.2 Contexte du travail 5

1.3 Présentation de Sodifrance 5

1.4 Organisation du rapport 5

2 Notion de modèle **6**

2.1 Introduction 7

2.2 Domaine : représentation des connaissances 7

2.2.1 Réseaux sémantiques 7

2.2.2 Graphes conceptuels 8

2.2.3 sNets 11

2.3 Domaine : systèmes d'informations 12

2.3.1 eXtensible Markup Language, XML 12

2.3.2 OMG/MDA 14

2.3.3 MDE 16

2.4 Synthèse et conclusion 20

3 Transformations de modèles **21**

3.1 Introduction 22

3.2 Transformations d'arbres 22

3.2.1 Principes des transformations XML 22

3.2.2 eXtensible Stylesheet Language . . . 23
3.2.3 XQuery . . . 25
3.2.4 Synthèse . . . 26
3.3 Transformation de modèles MDA/MDE . . . 26
3.3.1 QVT . . . 27
3.3.2 ATLAS Transformation Language . . . 31
3.3.3 MIA-T . . . 32
3.3.4 Synthèse . . . 33
3.4 Transformation de graphes . . . 33
3.4.1 Principes et fondations . . . 34
3.4.2 Outils . . . 35

4 Tissage de modèles **36**
4.1 Introduction . . . 36
4.2 Métamodèle de tissage . . . 37
4.3 Tissage de graphes . . . 39
4.3.1 Techniques niveau élément . . . 39
4.3.2 Techniques niveau structure . . . 40
4.4 Opérateurs sur les tissages . . . 42
4.4.1 Calcul de différences entre modèles . . . 42
4.4.2 Opérateur de composition . . . 43
4.4.3 Opérateur de fusion . . . 43

5 Incrémentalité **44**
5.1 Introduction . . . 45
5.2 Définitions . . . 45
5.2.1 Caractérisation d'un algorithme incrémental . . . 45
5.2.2 Rendre un traitement incrémental . . . 46
5.3 Classement des travaux sur l'incrémentalité . . . 46
5.4 Principales méthodes générales . . . 47
5.4.1 Différence finie (Finite differencing) . . . 47
5.4.2 INC . . . 47
5.4.3 Mise en cache (Functiong caching) . . . 47
5.4.4 Evaluation partielle (Partial evaluation) . . . 48

5.4.5 Renforcement d'invariants (Strenghtening invariants for incrementalization) 49
5.5 Méthodes spécifiques 50
5.5.1 Mise à jour d'entrepôts de données 50
5.5.2 Traduction incrémentale 51
5.5.3 XSLT incrémental 51

6 Traçabilité **55**
6.1 Introduction 55
6.2 Traçabilité des tranformations de modèles 56
6.3 Traçabilité de l'évolution des modèles 57
6.3.1 Traçabilité à priori 57
6.3.2 Traçabilité à posteriori 58

7 Bilan et perspectives **59**
7.1 Bilan 59
7.2 Perspectives 59
7.3 Remerciements 60

Références **70**

Table des figures

1.1 Une chaîne de transformations 1
1.2 Un exemple de chaîne de transformations 3
1.3 Chaîne de transformations et évolutions des modéles 4

2.1 Exemple de graphe conceptuel 9
2.2 Un exemple de document XML 12
2.3 Structure arborescente XML 13
2.4 Métamodèle XML Schema (Partie 1 du standard) 15
2.5 Les couches de modélisation de l'OMG 17
2.6 Modèle et relation de conformité 18
2.7 Le concept de modèle comme entité abstraite 19
2.8 La pile de modélisation et la relation de conformité 19

3.1 Transformation d'arbre XML 23
3.2 Transformation de modèles MDA/MDE 27
3.3 Architecture de QVT 28
3.4 Les 12 points de conformité QVT 30

4.1 Métamodèle abstrait de tissage 38

Liste des acronymes

ADT Atlas Development Tools

API Application Programming Interface

ATL Atlas Transformatin Language

DSL Domain Specific Language

DTD Document Type Definition

EBNF Extended Backus-Naur Form

EMF Eclipse Modeling Framework

HTML HyperText Markup Language

IDM Ingénierie Dirigée par les modèles

JSP JavaServer Pages

KM3 Kernel MetaMetaModel

L4G Langage 4ème Génération

MDA Model Driven Architecture

MDE Model Driven Engineering

MDR MetaData Repository

MIA Model-In-Action

MOF Meta Object Facility

NSDK Nat System Development Kit

OCL Object Constraint Language

OMG Object Management Group

QVT Query/View/Transformation

RFP Request For Proposal

SGML Standard Generalized Markup Language

UML Unified Modeling Language

W3C World Wide Web Consortium

XML eXtensible Markup Language

XSLT eXtensible Stylesheet Language : Transformation

Chapitre 1

Introduction

1.1 Motivations et objectifs

Il existe un fait établi et reconnu par tous : tout système informatique est amené à évoluer [56]. Dans une optique MDE, le système informatique est représenté par ses modèles. Ces derniers sont donc eux aussi amenés à évoluer. Les développements de tels systèmes sont donc nécessairement itératifs, raffinés par incréments successifs.

Au centre de ces développements, les transformations de modèles jouent un rôle critique. Mais les transformations monolithiques sont très peu fréquemment utilisées. On leur préfère de « petites »transformations, chacune s'intéressant à une préoccupation particulière. Par exemple, une transformation qui fait un pont sémantique entre deux espaces technologiques [98] ou encore une transformation qui passe d'un modèle XSLT vers un modèle XQuery [14]. Mais ces transformations ne sont pas utilisées de manière désorganisées : elles s'intègrent dans des chaînes effectuant ainsi des traitements beaucoup plus complexes (voir figure 1.1). Nous pourrions comparer ces chaînes de transformations aux lignes de commandes des systèmes Unix qui, bien souvent, se servent de tuyaux (*pipe*) pour enchaîner des utilitaires de base et ainsi obtenir une fonctionnalité beaucoup plus puissante.

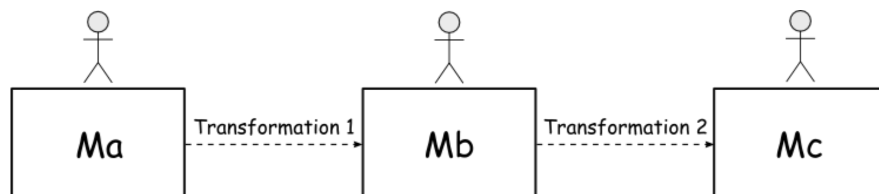

Fig. 1.1 – Une chaîne de transformations

Le but de ce chapitre est de présenter la problématique introduite par les deux faits exposés précédemment à savoir : (1) le développement itératif des systèmes informatiques et (2) le chaînage des transformations de modèles. Nous regarderons tout d'abord les caractéristiques d'une chaîne de transformations. Ensuite nous énumérerons l'ensemble des éléments à prendre en compte pour répondre à la problématique de l'intégration des transformations de modèles dans le contexte d'un développement itératif.

1.1.1 Les chaînes de transformations

Les chaînes de transformations sont nées d'un adage souvent appliqué en informatique : diviser pour mieux régner. En effet, des « monstres »monolithiques, que ce soit des programmes, des modèles ou encore les transformations sont peu maintenables et peu réutilisables. Il est alors naturel de décomposer le problème en petits sous-problèmes.

L'application de transformations successives peut être vue un peu à la manière de raffinements successifs des modèles dont le but final est l'implémentation. Par exemple, dans le cadre d'un projet de migration, il est fréquent de retrouver des chaînes comportant deux, trois, voire quatre transformations successives.

Prenons le cas d'une migration d'une application développée avec un L4G (Langage de 4ème Génération) devant être migrée vers une architecture n-tiers type Java/JSP tel que Sodifrance a pu réaliser.

La figure 1.2 présente ce type de projet. En entrée nous avons le code patrimonial (*legacy*) qui est codé dans un langage de quatrième génération type Visual Basic ou encore NSDK. Celui-ci est chargé dans un modèle conforme à un métamodèle généraliste de L4G[1]. Ce modèle, représentant le programme d'origine, est transformé en un modèle de nouvelle technologie. Ce modèle est nommé ainsi pour rester générique puisqu'il dépend de la tendance du moment. Nous pourrions par exemple l'appeler modèle *n-tiers*. Ce modèle sert à générer le code métier de l'application mais il est aussi transformé en un modèle UML qui va modéliser les enchaînements de pages (la navigation). Enfin, ce modèle est utilisé pour générer le code des JSP qui vont opérationnaliser ces enchaînements.

Mais les chaînes de transformations n'apparaissent pas seulement dans le cadre des migrations. Afin d'assurer l'interopérabilité d'outils, l'emploi de modèles pivots [13] est une technique courante. Elle consiste à définir un métamodèle contenant les concepts critiques des outils et à en faire un métamodèle. Ce métamodèle servira de cible commune aux outils pour l'export de

[1] nous passons volontairement sous silence le chargement de ce modèle car hors du sujet de ce chapitre

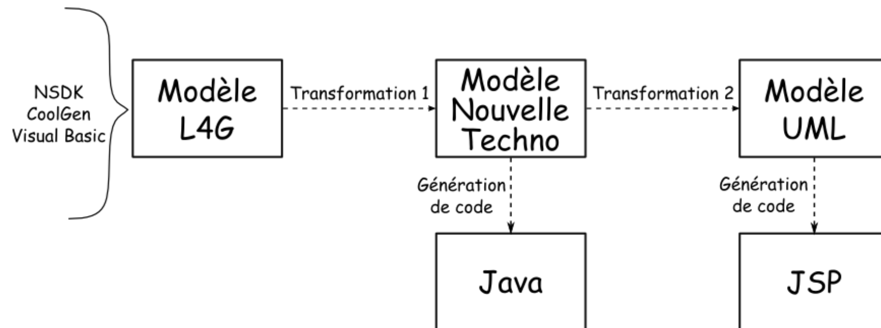

FIG. 1.2 – Un exemple de chaîne de transformations

leurs données mais aussi de source d'informations pour récupérer les données des autres outils. On dit alors que le modèle pivot sert de dépot (*repository*) pour l'architecture mise en place. Imaginons que nous ayons dix outils ou versions d'outils incompatibles à rendre interopérables. Dans le cas classique, il faudrait faire plus de quatre-vingts ponts ($(n-1)^2$) entre les outils. Avec l'utilisation de chaînes de transformations, il en faut vingt ($2n$). Les opérateurs d'import et d'export avec le dépot sont ensuite assemblés afin de former l'ensemble des couples nécessaires. Ces couples forment des chaînes de transformations.

Afin de tirer toute la puissance de l'approche MDE, les chaînes de transformations sont inévitables. Par la segmentation des préoccupations, elles assurent la capitalisation des transformations et ainsi, leur réutilisabilité. Cependant, de manière pragmatique, il ne suffit pas d'appuyer sur un bouton en début de chaîne pour que l'application soit entièrement générée à l'autre bout. Les modèles, où qu'ils se situent dans la chaîne, sont modifiés manuellement au cours du projet. Cela pose des problèmes que nous allons aborder dans la section suivante.

1.1.2 Problématique

Étant soumis à l'imparable loi de l'évolution des systèmes informatique citée en introduction, les éléments d'une chaîne de transformations changent au cours du temps. Mais ils évoluent le plus souvent en parallèle. C'est à dire qu'une fois la première transformation appliquée sur Ma puis la seconde sur Mb, les trois modèles évoluent de manière indépendante (figure 1.3). Le problème est qu'à certains moment critiques, il est nécessaire de ré-appliquer les transormations afin de les re-synchroniser. On doit alors ré-intégrer les modifications de Ma dans Mb et/ou de Mb dans Mc par le biais des transformations. L'idée est de pouvoir conserver les changements effectués entre temps dans Mb et Mc.

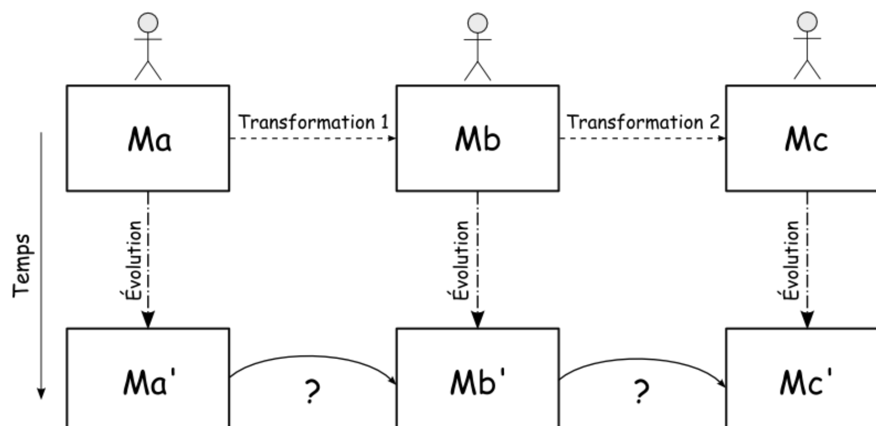

FIG. 1.3 – Chaîne de transformations et évolutions des modéles

Reprenons l'exemple de la figure 1.2. La première transformation génère un modèle conforme à une nouvelle technologie (c'est à dire à un métamodèle de nouvelle technologie). Ce modèle est utilisé ensuite dans deux transformations : une génération de code et une transformation vers un modèle UML. Bien souvent, les informations qui ont été extraites du modèle L4G ne sont pas suffisantes pour réaliser ces deux transformations. Aussi, le modèle « nouvelle technologie »est édité manuellement afin de le compléter par des aspects n'ayant pu, par manque d'information ou par une compléxité trop grande dans la mise en oeuvre, être transformé à partir du modèle L4G. Bien entendu ce scénario se répète autour de la deuxième transformation avec les modèles « nouvelle technologie »et UML.

Ce type de scénario peut se résumer comme sur la figure 1.3. Les éléments de la chaîne évoluent en parallèles.

Imaginons maintenant que le client (dans un contexte industriel) livre une nouvelle version du modèle d'entrée L4G. De votre coté vous avez déjà effectué des modifications sur le modèle nouvelle technologie. Comment synchroniser ce dernier avec la nouvelle version du modèle d'entrée ? Quels sont les impacts sur le troisième modèle ? Les outils actuels ne prennent pas en compte ces problèmes. Il faut donc à présent s'adapter aux processus de développement. Par notre étude nous allons tenter de mettre en exergue les possibilités qui sont susceptibles de répondre à cette problématique.

1.2 Contexte du travail

Ce travail est réalisé lors d'un stage Master de recherche, option Architectures Logicielles Distribuées – ALD. Ce stage est issue d'une collaboration entre le Laboratoire d'Informatique de Nantes Atlantique (Université de Nantes) et Sodifrance. Il est réalisé dans le locaux de Sodifrance, au sein de l'équipe recherche dirigé par Erwan Breton, Docteur en Informatique de l'Université de Nantes. Enfin, ce stage est encadré par Jean Bézivin , professeur en informatique à l'université de Nantes et membre de l'équipe Atlas (LINA et INRIA).

1.3 Présentation de Sodifrance

1.4 Organisation du rapport

Ce rapport constitue la première partie du travail de recherche effectué lors de ce stage. Il s'agit de toute la partie état de l'art. Il sera complété par une seconde partie nommée contribution à la fin de ce stage.

Cette partie état de l'art est organisée en sept chapitres dont nous fournissons un brève aperçu ci-après.

Chapitre 1 Cette introduction.

Chapitre 2 Ce chapitre présente la notion de modèles et de schémas conceptuels dans différents domaines d'ingénierie.

Chapitre 3 Introduction au concept de transformation de modèles. Ce chapitre est organisé selon la typologie des structures à transformer.

Chapitre 4 Le tissage est une extension de la transformation de modèles. Il est introduit ici en présentant tout d'abord un métamodèle minimal puis quelques heuristiques pour réaliser cette opération et enfin des exemples d'utilisation.

Chapitre 5 Nous rentrons dans le cœur du sujet avec le problème de l'incrémentalité. Ce chapitre expose de manière classifiée les différents travaux sur le problème.

Chapitre 6 Ce chapitre présente le problème de traçabilité en tentant de montrer les différences de préoccupations selon l'objet sur lequel il porte.

Chapitre 7 Enfin, ce septième et dernier chapitre conclue la première phase de notre travail, à savoir l'état de l'art.

Chapitre 2

Notion de modèle

Résumé

Les récentes évolutions qui ont eu lieu dans le domaine de l'ingénierie des modèles sont une avancée considérable pour la construction de systèmes complexes à partir de ces modèles. Avant d'étudier ces techniques d'ingénierie et les axes connexes à nos recherches, nous avons étudier la notion de modèles sous deux axes constituant chacun un domaine qui utilise fortement des modèles : le domaine de la représentation des connaissances et celui des systèmes d'informations. Le premier s'attache beaucoup à la sémantique des liens entre concepts tandis que le second s'attache plus à la définition de concepts de manière proche de la conception par objet. Enfin, les derniers courants utilisant les modèles, notamment le MDE, sont une vision conciliant ces deux approches.

Sommaire

2.1 Introduction **7**
2.2 Domaine : représentation des connaissances **7**
2.2.1 Réseaux sémantiques 7
2.2.2 Graphes conceptuels 8
2.2.3 sNets 11
2.3 Domaine : systèmes d'informations **12**
2.3.1 eXtensible Markup Language, XML 12
2.3.2 OMG/MDA 14
2.3.3 MDE 16
2.4 Synthèse et conclusion **20**

2.1 Introduction

La notion de modèle est soumise à interprétation. Étant à la base de notre travail, il est normal que nous en étudions les différentes visions. Nous nous interresserons à deux domaines qui utilisent ces modèles dans un contexte d'ingénierie : le domaine de la représentation des connaissances humaines et le domaine des systèmes d'informations.

2.2 Domaine : représentation des connaissances

2.2.1 Réseaux sémantiques

Les réseaux sémantiques forment un outil qui simule la représentation de notre mémoire. C'est un modèle qui montre comment l'information pourrait être représentée en mémoire et comment on pourrait accéder à ces informations.

Notre mémoire est représentée comme un bassin de données contenant des concepts, des événements, des sensations, ...Tous ces éléments forment un immense réseau en interrelation. Couché sur le papier, un réseau sémantique est composé de nœuds dont les interrelations sont établies par des pointeurs étiquetés. Les nœuds sont les différents types d'information en mémoire. A ces nœuds peuvent être associées des propositions, c'est à dire des énoncés qui caractérisent les propriétés s'appliquant aux nœuds du réseau. L'étiquette associée au pointeur indique quel est le type de relation entre deux nœuds.

Les réseaux sémantiques ont été inventé pour les ordinateurs par Richens (Cambridge Language Research Unit) en 1956 comme une langue intermédiaire pour la traduction automatique des langages naturels. Ils ont été développés par Simmons [53] chez Systems Development Corporation en Californie au début des années 1960 et largement augmentés dans les travaux de Quillian à partir de 1966 [89].

De tels réseaux impliquent des relations sémantiques relativement faibles qui sont néanmoins importantes pour une navigation humaine.

Les principales relations sémantiques utilisées sont :

- Méronymie (A est une partie de B)
- Holonymie (B possède A comme une partie de lui-même)
- Hyponymie (A est du type de B)
- Hyperonymie (A superordonné par rapport à B)
- Synonymie (A dénote la même chose que B)

– Antonymie (A dénote l'opposé de B)

Il est possible de représenter les descriptions logiques en utilisant des réseaux sémantiques tels que les graphes existentiels de Peirce [85] ou les graphes conceptuels de Sowa (voir section 2.2.2). Ceux-ci ont un pouvoir d'expression au moins égal à la logique des prédicats du premier ordre.

Les réseaux sémantiques sont donc des moyens très laxistes de représentation des connaissances. Seuls, ils ne permettent pas une structuration importante des modèles. Nous allons maintenant regarder de plus près un type de réseaux sémantiques plus cadré : les graphes conceptuels.

2.2.2 Graphes conceptuels

Le modèle des graphes conceptuels est un modèle de représentation de connaissances du type réseaux sémantiques qui a donné lieu à un certain nombre de travaux [20, 33, 27] depuis son introduction par John F. Sowa en 1976 [96] puis détaillé dans un livre en 1984 [97].

L'une des particularités de ce modèle est de permettre de représenter des connaissances sous forme graphique. Plus précisément, un graphe conceptuel est un graphe bipartite étiqueté, les deux classes de sommets étant étiquetés respectivement par des noms de « concepts »et des noms de « relations conceptuelles »entre ces concepts. Une telle représentation graphique des connaissances permet aux utilisateurs de comprendre, créer ou modifier directement des objets de ce type, de façon beaucoup plus simple qu'avec une représentation sous forme de formules logiques par exemple.

La facilité d'utilisation du modèle est renforcée par une séparation explicite de différents types de connaissances. Différents types de connaissances sont en effet représentés de différentes façons par des objets distincts et cette structuration entraîne une plus grande clarté lors de l'utilisation de ce modèle pour l'acquisition ou la représentation de connaissances. Le vocabulaire permettant de représenter des connaissances (l'ontologie du domaine) est notamment structuré dans un objet du modèle appelé « support »qui permet de représenter de façon simple des liens « sorte de »et « est un ».

L'autre intérêt du modèle vient du fait que des raisonnements peuvent être effectués sur les connaissances représentées. Ces raisonnements peuvent être vus comme des opérations de graphes et peuvent ainsi se baser sur les travaux d'algorithmique de graphe, puisque les graphes conceptuels sont des graphes étiquetés. Enfin, le modèle est muni d'une sémantique en logique du premier ordre qui est adéquate et complète par rapport à la déduction.

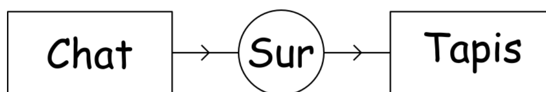

FIG. 2.1 – Exemple de graphe conceptuel

Ce qui suit est tirée de la proposition de standardisation des graphes conceptuels auprès de l'ISO[1].

Un graphe conceptuel est un graphe bipartite G qui possède deux types de nœuds appelés concepts et des relations conceptuelles tel que :

- Tout arc a de G doit lier une relation conceptuelle r de G à un concept c de G. Cet arc a est dit appartenir à la relation r. Il est aussi attaché au concept c, mais il n'appartient pas à c.
- Le graphe conceptuel G peut avoir des concepts qui ne sont liés à aucune relation conceptuelle. Cependant, tout arc qui appartient à n'importe quelle relation conceptuelle de G doit être attaché à exactement un concept de G.
- Les concepts d'un graphe conceptuel ont chacun un type t et un référent r.
- Toute relation conceptuelle r a un type associé t et une valeur entière positive n appelée valence. Le nombre d'arcs qui appartiennent à r est égal à la valence n.

Prenons un exemple afin d'expliciter les définitions ci-dessus. Nous allons prendre une phrase anecdotique et la représenter dans chacune des syntaxes concrètes des graphes conceptuels : *Display Form (DF)*, *Conceptual Graph Interchange Form (CGIF)* et *Linear Form (LF)*. Ces trois syntaxes peuvent-être traduites en formule de calcul des prédicats ou en *Knowledge Interchange Format (KIF)* [39], ce que nous ferons.

Soit la phrase « un chat sur un tapis ». Dans la forme DF (voir figure 2.1), les concepts sont représentés par des rectangles : le concept `[Chat]` représente une instance d'un chat et `[Tapis]` représente une instance d'un tapis. Les relations conceptuelles sont représentées par des cercles ou des ovales : la relation conceptuelle `(Sur)` lie un chat à un tapis. Les arcs qui lient les relations aux concepts sont représentés par des flèches. Si une relation a plus de deux arcs, l'arc est numéroté.

La forme linéaire (LF) représente les concepts entre crochets `[]` à la place des boites et les relations conceptuelles entre parenthèses à la place de cercles : `[Chat]` → `(Sur)` → `[Tapis]`.

Ces deux représentations sont utilisées pour la communication entre humains ou entre humain et machines. La représentation CGIF a une syntaxe utilisée pour la communication de

[1] http://www.jfsowa.com/cg/cgstand.htm

machine à machine. Sa syntaxe associe des étiquettes aux concepts afin de pouvoir y faire référence pour représenter les arcs : `[Chat: *x] [Tapis: *y] (Sur ?x ?y)`.

Il est aussi permis, afin de reduire le nombre d'étiquettes, d'inclure directement les concepts à l'intérieur des relations : `(Sur [Chat] [Tapis])`.

CGIF est prévu pour les échanges entre systèmes utilisant les graphes conceptuels en tant que représentation interne. Pour les échanges entre systèmes utilisant d'autres représentations, CGIF peut-être traduit dans un autre formalisme logique appelé *Knowledge Interchange Format, KIF* : `(exists ((?x Chat) (?y Tapis)) (Sur ?x ?y))`

Tout ces formalismes semblent différents mais ils reposent sur les mêmes fondations logiques. Ils peuvent tous être traduis en formule de calcul des prédicats type :

`(∃x:Chat)(∃y:Tapis)sur(x,y)`.

Maintenant que nous avons défini notre phrase sous le formalisme des graphes conceptuels, il convient aussi de définir les types des concepts et des relations. Ainsi, il conviendrait de définir qu'un chat est un animal qui émet des miaulements[2] :

```
Type Chat(x) is [Animal: *x] -> (Emet) -> [Miaulement]
```

De même, il faudrait alors définir la relation *émet* :

```
[Relation: Emet]->(Def)->[LambdaExpression:
     [Animal: x]<-(Agnt)<-[Emettre]->(Dest)->[Cri: y] ]
```

Nous créons ainsi une hiérarchie de types partiellement ordonnée. Au sommet de cette hiérarchie, le formalisme définit deux étiquettes de types primitifs : *Entity*, le type universel et *Absurdity*, le type absurde. Les relations conceptuelles sont elles-mêmes définies suivant ces types primitifs.

Ce formalisme permet donc d'exprimer à la fois des modèles et leur sémantique. Le problème de ce formalisme est qu'il est très généraliste. En effet nous avons vu qu'un concept était identifié par son type et son référent et qu'il pouvait lui même être constitué d'un graphe conceptuel. Ce graphe est alors considéré comme décrivant le référent. Le référent peut également être constitué d'un simple quantificateur (le quantificateur existentiel identifié par $\exists$ ou le quantificateur universel identifié par $\forall$).

La notion de graphes conceptuels apporte une puissance dans le domaine de l'intelligence artificielle par des opérations de base telles que la copie (copie de graphes conceptuels), la restriction (remplacement dans un graphe d'un type par un sous-type de celui-ci), la jointure

[2] ce qui, dans nos contrées, est une façon quasi certaine d'identifier un chat.

(jointure de deux graphes disposant d'un concept commun) et la simplification (suppression des relations dupliquées dans le graphe). Ces opérations peuvent être utilisées et composées.

2.2.3 sNets

Le formalisme des sNets est basé sur les réseaux sémantiques et défini par Lemesle au cours de sa thèse [57]. Aux réseaux sémantiques ont été ajouté la modularité et le typage, mais sans pour autant s'éloigner de cette base, de telle sorte qu'un modèle sNets puisse toujours être vu comme un simple réseau sémantique. Les termes entité et nœud sont employés indifférement. En effet, ce formalisme est le coeur d'un *framework* de modélisation et de métamodélisation au sein duquel les éléments représentés sont tous des entités. Toutes les entités sont nommées (ce nom pouvant être arbitraire). En terme de représentation graphique, la plus basique consiste à représenter les entités par des cercles avec le nom de l'entité en son centre. Les liens entre les entités sont alors représentés par des flèches nommées entre ces entités.

Un certain nombre de prédicats et de formules logiques ont été introduit afin de définir formellement les sNet. Les prédicats de base sont :

- Node : définissant un nœud (ou entité)
- Link : un lien entre deux nœuds

Les nœuds peuvent être typés par un prédicat *type* qui est équivalent à un lien *meta* entre l'objet et son type.

Un mécanisme de modularité a aussi été défini par un type d'entité appelé *Universe* permettant ainsi de partitionner le sNets. Une entité est alors rattachée à son univers via un lien appelé *partOf*.

Les métaentités sont définies par un type *EntityType* et les métarelations (relations entre métaentités) sont définies par un type *RelationType*.

Les mécanismes d'extensions des sNets sont disponibles à travers l'extension d'univers et l'héritage des types. Le mécanisme d'extension des univers permet de factoriser la définition d'un univers. Pour pouvoir considérer le sNet comme un simple réseau sémantique, il suffit de définir un lien *extends* partant d'un univers et aboutissant sur le ou les univers qu'il étend. Le mécanisme d'héritage des types quant à lui est très proche de l'héritage tel qu'on le trouve dans les différents langages de programmation à objets.

Les règles d'écriture d'un univers sont contraintes par un univers dit « sémantique »[3]

[3] Nous ne sommes pas d'accord sur cette interprétation de l'expression de la sémantique d'un modèle : le métamodèle n'est pas la sémantique d'un modèle. Voir [47] pour plus de précision.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <annuaire>
3.    <personne catégorie="jedi">
4.      <nom>Luke Skywalker</nom>
5.      <description>Un apprenti Jedi</description>
6.      <email>luke.skywalker@sw.foo</email>
7.    </personne>
8.    <personne catégorie="sith">
9.      <nom>Darth Maul</nom>
10      <description>Un apprenti Sith</description>
11.     <email>dm@sw.foo</email>
12.   </personne>
13. </annuaire>
```

FIG. 2.2 – Un exemple de document XML

Enfin, un univers nommé *Semantic* est défini au sommet de la hiérarchie des univers. Cet univers est l'univers sémantique de tous les autres univers sémantiques. Ce métamétamodèle est réflexif, c'est à dire qu'il se définit lui-même. Les concepts et les contraintes précédemment cités sont donc applicables à cet univers.

Les sNets permettent donc de définir des modèles de manière formelle avec un noyau minimal et donc très peu de concepts. Cette extension des réseaux sémantiques est très puissante mais l'utilisation de liens entre univers, et notamment entre liens et nœuds de deux univers différents implique que la totalité des univers est un hypergraphe. Cette contrainte pose des problèmes techniques car le traitement des hypergraphes est encore un sujet d'étude pour de nombreux travaux. Les définitions du MDE (voir section 2.3.3) tentent de résoudre ce problème.

2.3 Domaine : systèmes d'informations

2.3.1 eXtensible Markup Language, XML

Le XML [104] est né du SGML (Standard Generalized Markup Language) et HTML (HyperText Markup Language). Il est défini comme standard au W3C (World Wide Web Consortium) [113].

Le XML est un langage structuré utilisant des balises. Un exemple de document est présenté sur la figure 2.2.

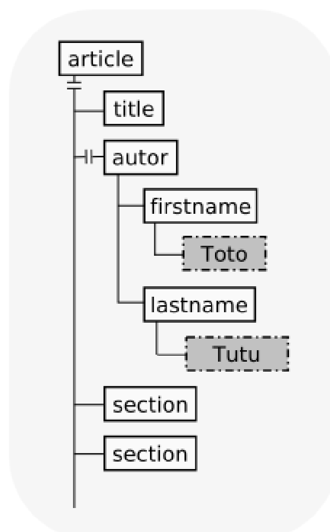

Fig. 2.3 – Structure arborescente XML

La structure d'un document XML est hiérarchique et arborescente et est donc aussi facilement interprétable par un humain que par un programme informatique. Ainsi, il est facile de représenter visuellement un document XML sous forme d'arbre (voir figure 2.3)

XML permet de définir ses propres formats de données. C'est pour cela que l'on considère plutôt XML comme un métalangage permettant de définir des nouveaux langages, ou dialectes. Des langages permettant de décrire la structure d'un document XML ont été repris, comme les DTDs (Document Type Definition) ou créés de toute pièce comme le XML Schema ou Relax-NG. Ainsi, des applications hétérogènes peuvent communiquer avec un format parfaitement adapté aux données qu'elles échangent en se mettant d'accord sur le langage utilisé. De par cette définition de schéma (au sens général du terme), XML apporte une notion essentielle, et qui n'existait pas dans SGML : c'est la définition de deux niveaux de validité à travers les propriétés de document bien-formé et de document valide.

Document bien formé Un document XML est bien-formé s'il respecte la syntaxe concrète définie notamment par les règles suivantes :

- il existe une balise englobant toutes les balises du document,
- toute balise ouverte doit être refermée (à `<balise>` doit être associé `</balise>`),
- lorsqu'un élément est vide, les balises peuvent être simplifiées : `<balise></balise>` est identique à `<balise/>`,

– si un élément contient d'autres éléments alors appelés éléments fils, les balises correspondantes doivent être fermées avant la fermeture de celles de leur parent. Par exemple, `<r><p><a>texte</p></a></r>` n'est pas un document bien formé car la balise `<a>` devrait être fermée avant `</p>`.
– Les valeurs des attributs doivent être entourées d'une paire de guillemets ou d'apostrophes. Ces règles simples permettent d'interpréter le document sous une forme arborescente pour lui appliquer des traitements.

Document valide Un document XML est valide par rapport à un modèle s'il se conforme aux exigences de structure décrites dans le schéma du modèle (c'est à dire s'il est valide par rapport à la syntaxe abstraite définie par le schéma). Un document valide est évidemment bien-formé.

Un point intéressant avec ces deux niveaux de validité est qu'il devient possible de traiter un document dès lors qu'il est bien formé, sans toutefois faire l'hypothèse beaucoup plus forte qu'il soit valide par rapport à un modèle.

Un document valide par rapport à un schéma est dit appartenir à un dialecte XML. Par exemple, un document SVG (Scalable Vector Graphic) valide appartient au dialecte SVG. Cette relation de validité est équivalente à la relation de conformité définie dans le cadre du MDE (voir section 2.3.3). Nous avons dit que ces schémas pouvaient être exprimés avec différents langages tels que les DTD ou encore XML Schema.

Un point intérressant avec XML Schema est qu'il est auto-défini. En effet, il existe dans la norme un document intitulé XML Schema pour XML Schema [106]. Nous pouvons donc considérer XML Schema pour XML Schema comme un métamétamodèle réflexif de XML. Une partie de ce métamodèle est schématisée sur la figure 2.4. Celle-ci est tirée de la spécification officielle de XML Schema.

Les modèles XML permettent donc aussi de représenter des systèmes. Cependant, leur structuration arborescente limite beaucoup leur pouvoir d'expressivité comparativement aux modèles vus précédemment.

2.3.2 OMG/MDA

L'initiative Model Driven Architecture (MDA^{TM}, [71]) a été lancée par l'Object Management Group (OMG, [75]) en 2000. Cette approche a pour but de permettre l'interopérabilité des systèmes informatiques. Afin d'atteindre cet objectif, l'OMG a défini un ensemble de standards industriels pour le développement et la maintenance de systèmes à composantes logicielles.

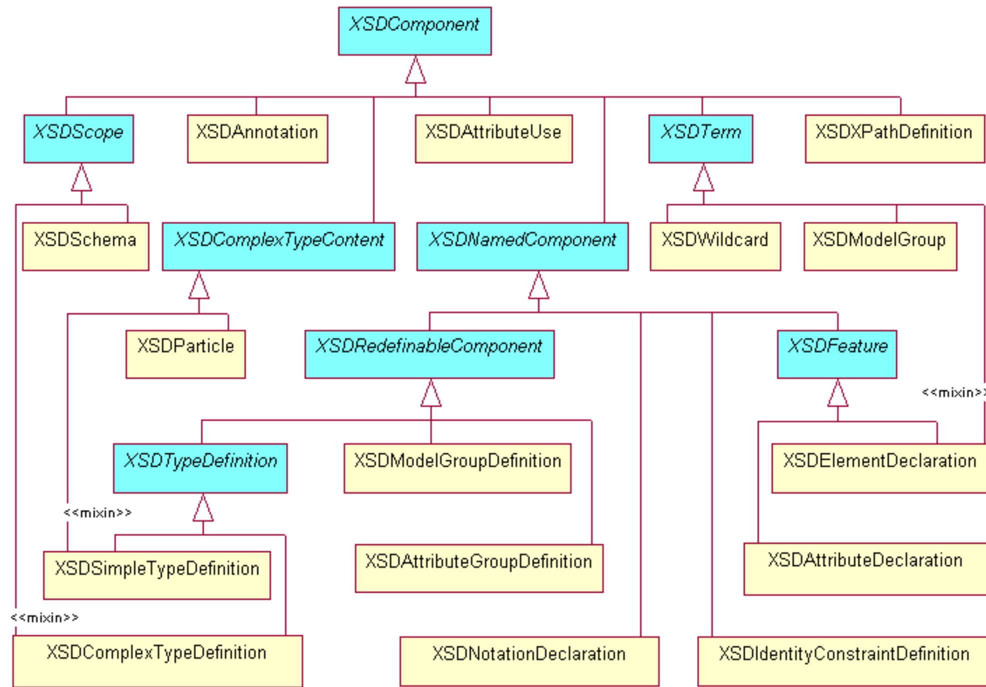

FIG. 2.4 – Métamodèle XML Schema (Partie 1 du standard)

Cette initiative prend Unified Modeling Language (UML, [79]) comme fondation afin de définir séparément les modèles indépendants des plates-formes (PIM, Platform Independant Model) et les modèles liés à une plate-forme spécifique (PSM, Platform Specific Model). Cette idée de départ a connu bien des évolutions.

Afin de pouvoir opérationnaliser ces modèles et ainsi automatiser leurs traitements, il est nécessaire de les définir dans un langage précis. Ce langage est décrit par ce que l'on nomme un métamodèle. Un métamodèle est un modèle qui définit un langage de modélisation. Par exemple, une légende d'une carte urbaine décrit le langage dans lequel la carte est écrit. Contrairement à une idée reçue, ce n'est pas un modèle du modèle. Si l'on reprend l'exemple du globe terrestre, modèle de la terre, un modèle de ce globe pourrait-être une description mathématique de la sphère et un ensemble de coordonnées définissant les contours des continents. Le métamodèle du globe serait plutôt la légende qui définit que les océans sont représentés en bleu et les terres émergées en couleur sable.

Afin qu'il soit compréhensible, le modèle doit donc respecter son métamodèle. On dit qu'il est conforme à ce métamodèle. Le MDA considère cette relation de conformité comme centrale. Nous ne nous attarderons pas sur les différents types de conformités puisqu'ils sont sujets à

controverse mais il est bon de noter qu'il en existent plusieurs : la conformité syntaxique, sémantique... Enfin, il faut noter qu'un métamodèle n'a pas qu'un seul et unique modèle qui lui est conforme.

Tout comme une légende peut être réutilisée pour plusieurs cartes, un métamodèle permet de définir plusieurs modèles. L'ensemble des modèles conforme à ce métamodèle forme le langage de modélisation, c'est à dire le modèle que représente le métamodèle. Pour résumer, un métamodèle est un modèle d'un langage (système dit abstrait) et tout modèle qui lui est conforme est un élément de ce langage. Ce dernier représente lui-même un autre système.

Nous avons vu que le MDA prenait les métamodèle comme entités de premier ordre afin de permettre d'automatiser les traitements sur les modèles. Mais ces entités de premier ordre sont aussi de modèles et sont vraisemblablement amenés à être, eux aussi, manipulés de manière automatique. Il est donc nécessaire de définir un langage pour les décrire. C'est ce que l'on appel les métamétamodèles.

Un métamétamodèle est un modèle qui définit un langage de modélisation pour les métamodèles. Cette définition semble proche de celle d'un métamodèle. Effectivement, on ré-applique les mêmes concepts à ce niveau qu'au niveau inférieur. Il est évident qu'en choisissant une telle stratégie, nous allons continuer à monter dans les méta niveau sans jamais pouvoir redescendre. C'est là où toutes les techniques d'ingénierie se rejoignent. En effet, ce troisième niveau est, la plupart du temps, réflexif.

Le plus haut niveau de modélisation de l'OMG est incarné par le Meta Object Facility (MOF, [80]). Succinctement, c'est un sous-ensemble minimal d'UML dont le but est de fournir des fondations à la définition de métamodèles tel que UML lui-même ou encore le métamodèle de transformation de modèles de l'OMG, à savoir QVT (Query / View / Transformation, [77]).

La pile de modélisation de l'OMG est représentée sur la figure 2.5.

2.3.3 MDE

On assiste aujourd'hui à l'émergence d'une déclinaison plus générale du MDA, connue sous le nom de Modèle Driven Engineering (MDE, ou IDM pour ingénierie Dirigée par les Modèles en français). Cette approche est semblable dans l'architecture et les techniques mises en oeuvre mais ne se restreint pas au MOF pour unique métamétamodèle. Le MDA devient ainsi un sous-ensemble du MDE [35]. Les grands industriels comme Microsoft et IBM ont chacun leur déclinaison du MDE (respectivement les « Software Factories »et le projet « Eclipse Modeling Framework », EMF). Chacun à ses spécificités mais ils permettent tous deux de définir des

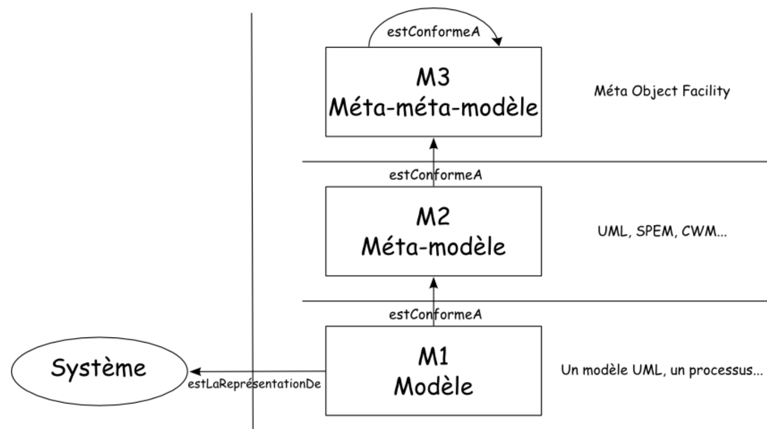

FIG. 2.5 – Les couches de modélisation de l'OMG

métamodèles non monolithiques contrairement à UML dans ses versions actuelles[4]. Microsoft a défini un nom pour ceux-ci que nous reprendrons largement : les « Domain Specific Language »(DSL). Au delà de cette vision élargie du MDA, le MDE tente aujourd'hui de formaliser ses concepts fondamentaux, c'est à dire les modèles et les transformations de modèles, en se fondant sur la théorie des graphes [51]. Nous pensons que cette vision « théorisée »est un bon moyen d'atteindre les objectifs initiaux du MDA/MDE : l'interopérabilité sans concession des outils et des systèmes.

Un modèle est donc en relation avec un système. La relation qui les unit est nommé *représentationDe* dans la littérature ([94], [36] et [12]). Le processus qui permet de représenter un système par un modèle est soumis au savoir de l'expert qui construit ce modèle. Aussi, la relation entre modèle et système est peu formalisable. Cependant, afin de pouvoir automatiser les traitements sur les modèles, il est nécessaire que ceux-ci soient écris dans un langage bien défini. Ce langage est lui-même représenté par un modèle, que nous nommerons indifféremment dans la suite modèle de référence ou métamodèle (voir figure 2.6). La relation qui lie modèle et modèle de référence est nommé « conformeA ».

Maintenant que nous avons défini cette relation de conformité, nous pouvons exprimer formellement la notion de modèle. Avant cela, rappelons la définition d'un graphe que nous allons utiliser.

Définition 1 Un multigraphe orienté $G = (N_G, E_G, \Gamma_G)$ est constitué d'un ensemble de nœuds N_G, un ensemble d'arcs E_G et d'une fonction de mapping $\Gamma_G : E_G \rightarrow N_G \times N_G$.

[4]c'est un des principal reproche qui est fait à UML

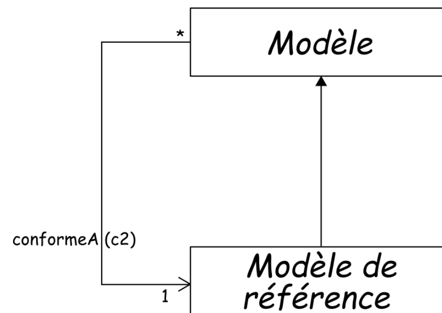

FIG. 2.6 – Modèle et relation de conformité

Un modèle n'est pas qu'un graphe. C'est un triplet comme présenté dans la définition 2.

Définition 2 Un modèle est un triplet $M = (G, \omega, \mu)$ tel que :

- $G = (N_G, E_G, \Gamma_G)$ est un multigraphe orienté,
- ω est lui même un modèle (appelé le modèle de référence ou métamodèle de M tel que M est conforme à ω) associé à un graphe $G_\omega = (N_\omega, E_\omega, \Gamma_\omega)$,
- $\mu : N_G \cup E_G \rightarrow N_\omega$ est une fonction associant les nœuds N_G et les arcs E_G de G aux nœuds de G_ω

Dans la vision MDA de l'OMG et notamment dans le papier « A Proposal for an MDA Foundation Model »[76], l'architecture proposée pour la démarche de modélisation comprend trois couches. Ce choix de trois couches a été défini de manière empirique comparativement à d'autres domaines d'ingénierie qui définissent également de manière générale trois niveaux de modélisation. Nous pouvons citer par exemple :

- Les technologies XML avec XML Schema pour XML Schema, les schémas XML, et les documents XML
- Les technologies des langages avec la grammaire de EBNF, les grammaires EBNF, et les langages.

Dans de nombreux domaines d'ingénierie, nous avons donc trois niveaux de modélisation. Afin de prendre un certaine distance vis à vis de la vision de l'OMG et de se rapprocher de celle du MDE, nous n'appelerons pas les niveaux M1, M2 et M3 mais modèles terminaux, métamodèles et métamétamodèles (voir figure 2.7). La différence tient surtout dans la qualificatif « terminal »attaché à un modèle de niveau M1. Ceci permet de définir un modèle comme une entité abstraite, base de tous les autres. Ces autres modèles ne sont alors que des modèles spécifiques étendant les concepts de base définis par cette entité abstraite.

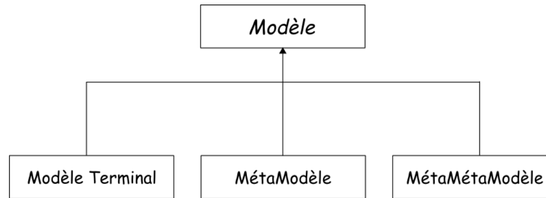

FIG. 2.7 – Le concept de modèle comme entité abstraite

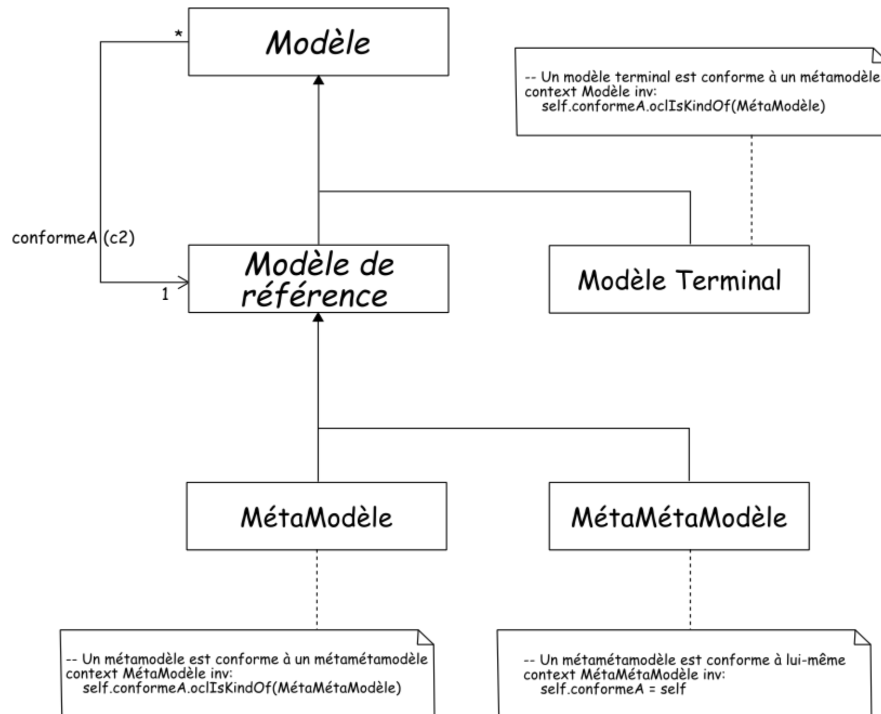

FIG. 2.8 – La pile de modélisation et la relation de conformité

Pour conclure cette section sur la notion de modèle MDE, faisons une synthèse en fusionnant les deux figures 2.6 et 2.7 pour former la figure 2.8. Ce cadre formel permet d'exprimer que toute entité est modèle dans l'approche MDE. De plus, on voit alors que tout modèle doit être conforme à un modèle de référence. Les trois niveaux sont bien représentés et leur relation de conformité avec le niveau supérieur (ou lui-même dans le cas du métamétamodèle) est exprimée via une contrainte OCL.

2.4 Synthèse et conclusion

La notion de modèle est donc utilisée dans des domaines variés. Les modèles tentant de représenter des connaissances sont très sensible à la sémantique des liens unissant les concepts qu'ils définissent. Les modèles d'ingénierie quant à eux présentent une sémantique plus limitée, relativement liée au domaine de la programmation par objet.

Nous retiendrons en conclusion que la vision MDE des modèles est un consensus entre ces deux grands domaines. Ainsi, le MDE tente d'unifier la notion de modèles d'ingénierie à un modèle de connaissance, sans perdre de vue l'objectifs de ceux-ci. La flexibilité et le pouvoir d'expression qui en découlent en font une vision très prometteuse et sera celle que nous retiendrons dans la suite de nos recherches.

Chapitre 3

Transformations de modèles

Résumé

L'opération de transformation est centrale dans toute démarche d'ingénierie. Quelle que soit la structure à transformer : arbre, modèle objet ou graphe, la même démarche est appliquée. A partir d'une sélection d'éléments dans un modèle source, un ensemble d'éléments est créé et initialisé dans le modèle cible. La structure du modèle considéré permet cependant d'appliquer des fondements théoriques et mathématiques afin d'obtenir des performances raisonnables.

Sommaire

3.1 Introduction **22**
3.2 Transformations d'arbres **22**
3.2.1 Principes des transformations XML 22
3.2.2 eXtensible Stylesheet Language 23
3.2.3 XQuery 25
3.2.4 Synthèse 26
3.3 Transformation de modèles MDA/MDE **26**
3.3.1 QVT 27
3.3.2 ATLAS Transformation Language 31
3.3.3 MIA-T 32
3.3.4 Synthèse 33
3.4 Transformation de graphes **33**
3.4.1 Principes et fondations 34
3.4.2 Outils 35

3.1 Introduction

La transformation de modèles est l'opération clé de toute démarche dirigée par les modèles. Cette opération peut avoir des sémantiques différentes en fonction des utilisations : raffinage, traduction, ré-écriture. . .Dans ce chapitre, nous allons traiter de transformations de modèles en fonction de la structure du modèle : arbre ou graphe. Dans le cas d'une transformation d'arbre, nous prendrons le cas de XML dont la notion de modèle est définit dans le chapitre précédent. Nous verrons ensuite les outils de transformations de modèles tels que les approches MDA et MDE les conçoivent. Enfin, nous verrons les techniques de transformation de graphes à proprement parler, considérant les modèles uniquement sous forme de graphes.

3.2 Transformations d'arbres

3.2.1 Principes des transformations XML

Les informations contenues dans un document de type XML sont structurées de manière arborescente, ainsi la transformation XML, autrement dit le passage d'un document de type XML à un autre est en réalité une opération qui transforme un arbre en une autre structure arborescente. Le document d'origine est appelé document source, tandis que le document issu de la transformation est appelé document cible (figure 3.1).

On distingue plusieurs classes de transformations :

- Transformation en bloc : c'est l'approche utilisée par XSLT. On dispose de la totalité de l'arbre source et des règles de transformation qui sélectionnent à chaque fois une partie précise de cet arbre et l'utilise pour générer le document cible.
- En *streaming* : on considère alors les données sources XML comme étant infinies, cela est par exemple utilisé dans le cadre de la TV numérique où l'on diffuse en permanence les informations liées à l'interactivité. La taille totale du flot de données est telle qu'on ne peut envisager de le stocker. Il faut donc réaliser la transformation au fur et à mesure, sans connaître la totalité du document source.
- Transformation incrémentale : sachant que la plupart des modifications apportées dans un document sont très légères, et qu'elles sont souvent à l'origine de légers changements dans le document cible, l'approche incrémentale vise à faire en sorte que l'on puisse calculer le nouveau résultat de manière incrémentale, c'est à dire en mettant à jour les parties adéquates de l'ancien résultat, plutôt que de recalculer à partir de zéro.

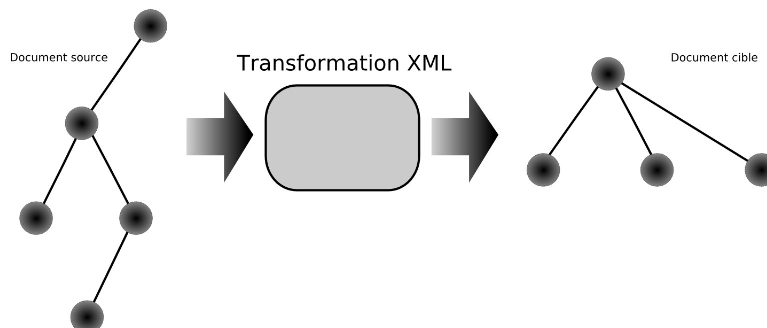

FIG. 3.1 – Transformation d'arbre XML

- Transformation inversible : une transformation est dite inversible si la transformation inverse existe. Autrement dit, si en modifiant le document cible, on est capable de répercuter ces modifications sur le document source. En règle générale les transformations génériques sont rarement inversibles, certaines approchent *ad hoc* fournissent cette fonctionnalité sur des cas très spécifiques.

Ce rapport est un exemple montrant l'intérêt des transformations XML. L'information est présente dans un document DocBook et il peut être tout à fait intéressant dans le cadre de la publication de ce document sur internet, de le transformer en un document XHTML. On peut tout aussi bien imaginer créer à partir de ce document, une représentation graphique vectorielle (SVG par exemple).

Nous allons maintenant décrire le deux principaux langages de transformations XML tels qu'ils sont définis par le W3C : XSLT [112] et XQuery [105].

3.2.2 eXtensible Stylesheet Language

XSL (eXtensible Stylesheet Language) est le langage de description de feuilles de style du W3C associé à XML. Une feuille de style XSL est un fichier qui décrit comment doivent être présentés (c'est-à-dire affichés, imprimés, épelés...) les documents XML basés sur une même DTD ou un même schéma. La spécification est divisée en trois parties :

XPath le langage de navigation dans un document XML [105].

XSLT le langage de transformation [112].

XSL-FO le vocabulaire XML de mise en forme (non détaillé dans ce document).

XPath

XPath est un standard du W3C défini en 1999 [105]. Il a été créé dans le but d'avoir une syntaxe commune pour XPointer et les transformations XSLT. Son nom vient du fait que les expressions XPath sont semblables aux « chemins »que l'on distingue dans les URL par exemple.

XPath permet la sélection d'un ensemble de nœuds dans un arbre XML. XPath fourni quelques fonctions basiques de traitement de chaînes de caractères, de nombre ou de booléens. XPath opère sur la structure logique, abstraite, des données XML et non pas sur la syntaxe de ce dernier. En plus de ces opérations il permet également de définir des règles de filtrage, ou encore matching : pour chaque nœud du source XML on va pouvoir dire s'il est apparié ou non au chemin XPath. Ceci est principalement utilisé dans les transformations XSLT.

XPath opère sur des données XML arborescentes. Cet arbre est composé de nœuds, on en distingue sept types comme définis dans le Document Object Model – DOM [103] (nœuds attributs, textes, éléments, etc.). On distingue également plusieurs types de relation de filliation entre ces nœuds tel que nœuds parents, enfants, etc.

Le principe de XPath est le suivant : on définit un sous ensemble de l'arbre par la concaténation d'un chemin auquel on peut associer un prédicat. Ce principe de base peut ensuite être utilisé autant de fois que nécessaire afin de sélectionner les données désirées.

XSLT

XSLT (eXtended Stylesheet Language Transformations) [112], défini au sein de la recommandation XSL du W3C, est un langage de transformation XML de type fonctionnel. L'objectif principal est la transformation d'un document XML vers un autre, ou un dialecte XML (XHTML, XSL-FO, HTML, etc.). Cependant, le langage XSLT permet aussi les transformations vers tout autre type de document, au format texte ou dans un format binaire (bien que ceci ne soit pas nativement prévu par la recommandation XSLT). XSLT s'appuie sur XPath [105] (une autre partie de la recommandation XSL) pour désigner une partie d'un arbre XML. XSLT est lui-même un dialecte XML. Un programme XSLT est donc, avant tout, un document XML. Le standard XSLT 1.0 est une *W3C Recommendation* du 16 novembre 1999 [112]. A noter que les attributs de certains éléments XML de ce langage contiennent des chaînes de caractères dont la syntaxe et la sémantique obéissent au langage XPath.

Le langage XSLT est de type déclaratif, c'est à dire que, contrairement aux langages impératifs qui sont des séquences d'actions à effectuer, une feuille XSLT est composée d'un ensemble

de règles nommées *templates*. Chacune d'elles spécifie ce qui doit être rajouté dans l'arbre résultat lorsque le processeur XSLT trouve un noeud, suivant un algorithme définie, qui correspond aux conditions de cette règle. De plus, XSLT est un langage sans effet de bord, c'est à dire qu'aucune procédure ne modifie un quelconque état ou valeur en dehors de sa valeur de retour.

La spécification XSLT définie une transformation avec les notions d'arbre source et résultat afin d'éviter d'enfermer les implémentations dans des interfaces spécifiques et dans des problèmes de mémoires, réseau et d'entrée-sortie.

De même, la sortie peut-être dans un langage qui n'a pas été imaginé par les créateurs de XSLT. Cependant, le traitement XSLT commence souvent par la lecture d'un document XML sérialisé et le transforme en arbre source et termine par écrire l'arbre résultat dans un document XML sérialisé lui aussi. Le document de sortie peut-être du XML mais aussi du HTML, RTF, TEX, CSV, texte plein ou n'importe quel format que le processeur XSLT est capable de produire.

XSLT se base sur le langage XPath pour identifier les sous-ensembles dans l'arbre source mais aussi pour effectuer des calculs. XPath fournit un certain nombre de fonctions que XSLT augmente de son coté.

3.2.3 XQuery

Le W3C travail actuellement sur la finalisation de la spécification de XQuery. XQuery est un langage pour interroger et transformer des données XML. Cela signifie que XQuery ne se limite pas seulement aux documents XML, mais peut tout aussi bien traiter des données (telles que les bases données) dont la structure (arborescente avec des attributs) est similaire au XML.

XQuery est le fruit d'une longue gestation, descendant du langage Quilt et inspiré par XQL, XML-QL et SQL. Le composant clé de XQuery qui familié aux utilisateurs de XML est XPath (voir section 3.2.2). XQuery est un sur-ensemble de XPath (dans sa version 2.0) et donc une expression XPath isolée est parfaitement valide suivant XQuery.

La ressemblance entre XQuery et SQL, tant au niveau apparence que des capacités, est frappante mais aussi la bienvenue pour les gens venant du monde des bases de données relationnelles.

Le langage XQuery est en réalité composé de trois langages :

- La syntaxe classique que la plupart des utilisateurs utilisent sans jamais avoir connaissance des deux autres [109].
- Un syntaxe alternative basée sur XML. Nommé XQueryX [108], ce langage est une traduction de la première syntaxe en pur XML pour un traitement machine plus aisé.

- Un langage algébrique formel [111] décrivant en détail le fonctionnement interne d'un processeur XQuery.

Les documents [110, 111] fournissent tous les deux une base théorique précise à XQuery. Ces deux documents détaillent une algèbre de requête, un ensemble de definitions précises qui définissent en termes formels les entités sur lesquelles une requête XQuery travaille, et les formulations de ce que peuvent faire les différents opérateurs du langage avec ces opérandes.

L'algèbre ainsi définie fournit des règles qui détaillent comment optimiser et transformer les expressions complexes en des équivalents plus simples.

L'algèbre fournit aussi des mécanismes pour gérer les types. XQuery est fortement typé : si les données en entrée d'un programme XQuery est associé à un XML Schema, le processeur peut valider ces données par rapport à ce Schema. De plus, il peut fournir au moteur de requêtes des informations *Post-Schema Validation Infoset (PSVI)* à propos des types de données des noeuds du documents, utilisant les types de bases de XML Schema [107] et les types définis par l'utilisateur. L'algèbre a aussi bien des capacités de vérifications de types statiques (grâce aux PSVI) et dynamiques.

3.2.4 Synthèse

XQuery et XSLT sont donc des langages de transformation XML. Le principal grief que l'on pourrait retenir contre XSLT est de ne pas utiliser une relation de conformité sur les documents d'entrée et de sortie de manière explicite. En effet, il est difficile de prouver qu'une transformation écrite en XSLT génère un document valide vis à vis d'un schéma. De même, si le métamodèle du document d'entrée est modifié un tant soit peu dans sa structure, il est très difficile d'en prévoir les conséquences sur la transformation. XQuery prend en compte ces métadonnées.

3.3 Transformation de modèles MDA/MDE

La transformation de modèles est une opération centrale de l'initiative MDA tout comme de l'Ingénierie Dirigée par les Modèles. Consciente de ce problème, l'OMG a fait l'appel à proposition (RFP – Request For Proposal) QVT pour *Query / View / Transformation*. En parallèle, d'autres moteurs de transformation existent étant tous plus ou moins éloignés de la spécification. Cependant, nous retiendrons que tous se basent sur le même schéma d'exécution tel qu'il est présenté en figure 3.2.

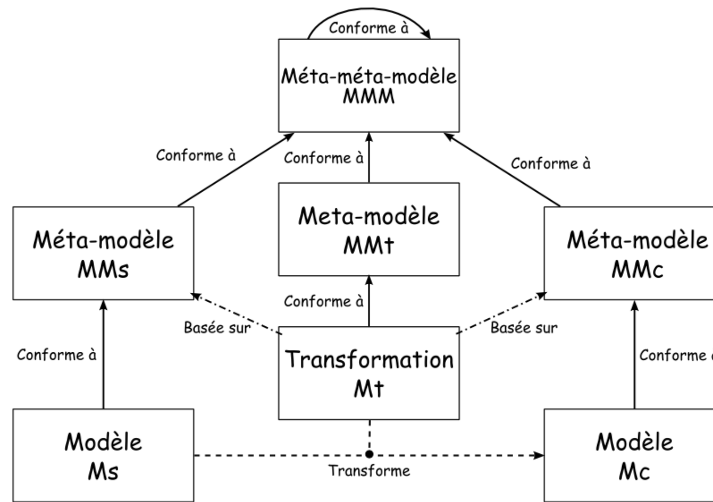

FIG. 3.2 – Transformation de modèles MDA/MDE

3.3.1 QVT

L'OMG a soulevé le problème de la standardisation de la transformation de modèles avec la *Query/View/Transformation (QVT) Request For Proposal (RFP)* [77] afin de chercher une réponse compatible avec sa vision MDA et ses fondations : UML [79], MOF [80], OCL [78], etc.

La spécification QVT est à l'heure actuelle en phase de finalisation. Elle a subit une longue gestation au sein des groupes de travail de l'OMG. Le rapport du groupe de travail concernant la validation de cette spécification sortira le 7 juillet 2006. Elle constitue une avancée importante dans la standardisation des langages et des mises en oeuvre de transformations de modèles.

L'objectif de cette section est de faire une présentation générale des concepts de la spécification MOF QVT telle qu'elle est présentée dans le document de l'OMG [77].

La spécification est divisée en deux composantes ; l'une impérative et l'autre déclarative. Cela lui confère une nature hybride répondant ainsi à la majorité des besoins de transformations de modèles. Au delà de cette séparation déclaratif/impératif, la composante déclarative est elle-même subdivisée en deux niveaux d'abstraction distincts.

La partie déclarative de QVT est structurée selon deux couches de niveaux d'abstraction différents :

- Un metamodèle et un langage nommé Core définie en étendant très peu EMOF et OCL [78].
- Un métamodèle et un langage nommé Relations qui est à un niveau d'abstraction supérieur par rapport à Core.

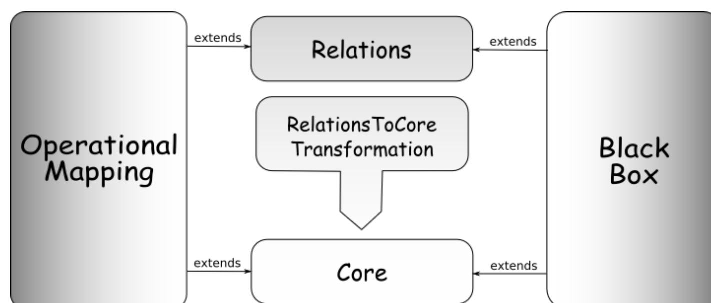

Fig. 3.3 – Architecture de QVT

La partie impérative de QVT est en réalité deux greffons qui viennent étendre à la fois la langage Core et le langage Realtions. Ces deux greffons sont :

- Le langage Operational Mappings est spécifié de manière standard afin de fournir une façon de programmer les transformations de manière impérative.
- Black Box implementation permet de définir des opérations de n'importe quelle implémentation d'une *MOF Operation* avec la même signature. Par exemple, cela permet de :
  - coder des algorithmes complexes dans n'importe quel langage de programmation ayant un *binding* MOF ;
  - utiliser des librairies spécifiques pour calculer des propriétés sur les modèles (par exemple dans les domaines des mathématiques, certains algorithmes seraient difficiles voire impossibles à exprimer avec OCL).

Partie déclarative

Le langage Core Le langage Core possède trois concepts fondamentaux : les mappings, les patterns et les bindings. Ces concepts sont minimaux et le langage Core a donc une sémantique simple. Le but de ce langage est de définir des bases pour la sémantique du langage Relations (bien que cela dépende de l'implémentation choisie).

Une transformation définie avec Core met en correspondance les éléments des modèles grâce à un ensemble de variables et en évaluant des conditions sur ces variables. Une transformation en langage Core est composé d'un ensemble de règles de correspondances nommées mappings. Cette mise en correspondance est réalisée par les domaines qui lui sont associés. En effet, chaque domaine possède deux patterns : un guard pattern et un bottom pattern. Un mapping définit une relation un à un entre tous les bottom pattern de ce mapping sous la condition définie par la garde de chaque domaine.

Un pattern est composé d'un ensemble de variables, de prédicats et d'assignations. Ils peuvent être mis en correspondance (mode matched) ou forcés (mode enforced). Les deux modes ont des comportements différents. Lors de la mise en correspondance de patterns, le résultat est un ensemble de liaisons entre les variables. Si les patterns sont forcés, le résultat est un changement dans le modèle qui change alors les liaisons entre les variables pendant la mise en correspondance. Les prédicats et les assignations sont exprimés via des expressions OCL.

Les classes de trace (définissant les liens de traçabilité) sont définies de manière explicite en tant que modèle MOF et sont manipulées de la même manière que les autres objets durant la transformation. Ainsi, c'est à l'auteur de la transformation de gérer la création, mise à jour et suppression de ces objets.

Le langage Relations Le langage Relations permet de définir une transformation entre des modèles candidats en spécifiant un ensemble de relations qui doivent être maintenues pour que la transformation soit réalisée. Un modèle candidat peut-être n'importe quel modèle qui est conforme à un métamodèle.

Une relation est définie par deux ou plusieurs domaines et une paire de prédicats *when* et *where*. Un domaine est un ensemble de variables typés par des éléments du métamodèles du domaine. Ces domaines plus ces prédicats spécifient l'association qui doit-être maintenue entre les modèles candidats.

Un domaine à un patron (*pattern*) qui peut-être vu comme un ensemble de variables et de contraintes. Les éléments des modèles candidats qui sont associés aux variables de domaines doivent satisfaire ces *pattern*.

Une relation peut aussi être contrainte par deux ensembles de prédicats, une clause *when* et une clause *where*. La clause *when* spécifie les conditions sous lesquelles la relation doit-être maintenue tandis que la clause *where* spécifie la condition que doivent satisfaire tous les éléments qui participent à la relation. Ces deux clauses sont exprimées suivant des expressions OCL.

La langage gère la manipulation des liens de traçabilité de manière automatique et cache tous les détails relatifs à l'utilisateur.

Partie impérative

Operational Mappings Le langage Operational Mappings permet de définir des transformations unidirectionnelles exprimées de manière impératives tout en définissant les mêmes modèles de trace que le langage Relations. Il définie une signature indiquant les modèles impliqués dans

<table>
<tr><td></td><td colspan="5">Dimension intéropérabilité</td></tr>
<tr><td rowspan="4">Dimension langage</td><td></td><td>Syntaxe exécutable</td><td>XMI exécutable</td><td>Syntaxe exportable</td><td>XMI exportable</td></tr>
<tr><td>Core</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Relations</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Operational Mappings</td><td></td><td></td><td></td><td></td></tr>
</table>

FIG. 3.4 – Les 12 points de conformité QVT

la transformation et il définit une opération principale pour son exécution (*main*). Comme une classe, une transformation suivant ce langage possède des propriétés et des opérations.

Ce langage étend le langage Relations avec des construction impératives et des expressions OCL avec la possiblité d'effets de bords.

L'idée de base dans ce langage est que les patterns d'objets spécifiés dans les relations sont instanciés en utilisant des constructions impératives. De cette façon, les relations spécifiés de manière déclarative sont implémentées de manière impératives. La syntaxe concrète du langage Operational Mappings fournit des constructions typiques des langages impératifs (boucles, conditions, etc.). Il peut être utilisé pour fournir une ou plusieurs relations à une transformation en langage Relations quand il est compliqué de définir celle-ci dans style purement déclaratif.

Black Box Le mécanisme dit boite noire (Black Box) permet de connecter et d'exécuter du code externe durant l'exécution de la transformation. Ce mécanisme permet d'implémenter des algorithmes complexes et aussi la réutilisation de bibliothèques externes. Cela rend certaines parties de la transformation opaque ce qui est un danger potentiel puisque les fonctionnalités ainsi fournies ne sont plus controllées par le moteur de transformation.

L'opération appelée par ce mécanisme est responsable de la bonne exécution des changements dans le modèle afin de statisfaire la relation étendue.

Points de conformité QVT

Afin de permettre de classer les outils, la proposition QVT a définit un tableau de conformité vis à vis de ses langages et de ses syntaxes abstraites et concrètes (tableau 3.4).

La dimension langage définit trois niveaux correspondant aux trois langages de QVT. Un outil conforme à un de ces niveaux est capable d'exécuter une transformation dans ce langage.

La dimension interropérabilité définie la façon dont les transformations sont exprimées. Il y a quatre niveaux :

Syntaxe exécutable . L'outil peut lire et exécuter une transformation écrite dans la syntaxe concrète d'un des langages.

XMI exécutable . L'outil peut lire et exécuter une transformation dont le modèle est sérialisé en XMI.

Syntaxe exportable . L'outil peut sérialiser une transformation dans la syntaxe concrète du langage.

XMI exportable . L'outil peut sérialiser une transformation dans le format XMI.

Dans [77], un condition est posée sur ces niveaux : si un outil est compatible avec un niveau d'exécutabilité donné, alors il doit être compatibles avec le niveau d'exportabilité correspondant.

Pour le moment, QVT est un norme qui ne possède pas d'implémentation. Cependant, il y a fort à parier que, dans un futur proche, de nombreuses implémentations se targueront d'être compatible QVT selon un certain niveau de conformité. Étant donné la complexité de cette spécification et les zones de flou qu'elle comporte, de nombreuses révisions vont suivre cette première spécification.

3.3.2 Atlas Transformation Language

ATL est la réponse à la RFP QVT [77] du groupe de recherche Atlas de l'INRIA et du LINA . Le langage ATL fait partie du projet ADT – ATL Development Tools qui a pour but de fournir une plate-forme de transformation de modèle avec un IDE – Integrated Development Environment et un moteur de transformation de modèles.

ATL est un langage de transformation de modèles spécifié à la fois comme un métamodèle et une syntaxe textuelle concrète. Dans le champs du MDE, ATL fournit aux développeurs un moyen de spécifier comment produire un certain nombre de modèles cibles à partir d'un ensemble de modèles sources.

ATL est un langage hybride, à la fois impératif et déclaratif. Le style de prédilection d'écriture d'une transformation est la manière déclarative : ce style permet d'exprimer simplement un mapping entre les éléments des modèles source et cible. Cependant, ATL fournit aussi des constructions impératives pour faciliter l'expression de transformations difficilement constructible déclarativement.

Un programme ATL est composé de règles qui définissent comment les éléments du modèle source sont identifiés et parcourus pour créer et initialiser les éléments des modèles cibles. Au delà des transformations de modèles classiques, ATL définie un modèle de requête supplémentaire qui permet de spécifier des requêtes sur les modèles. ATL permet aussi de factoriser le code redondant en autorisant la définition de bibliothèques.

L'environnement de développement intégré d'ATL est intégré à la plate-forme Eclipse [38]. Il fournit ainsi des outils de développements standards (coloration syntaxique, dévermineur ...) afin de faciliter l'écriture de transformations.

ATL accepte en entrée aussi bien des modèles et des métamodèles définit selon le MOF [80] *via* le MetaData Repository – MDR [74] de NetBeans ou selon Eclipse Modeling Framework – EMF [37]. Cette indépendance vis à vis du métamétamodèle est réalisé grâce à une couche d'abstraction nommé KM3 pour Kernel MetaMetaModel [51] incluant une notation textuelle simple pour spécifier des métamodèles et de nombreux ponts entre cette notation et les représentations correspondantes.

Le moteur de transformation ATL est une machine virtuelle. Comme la *Java Virtual Machine*, la machine virtuelle ATL dispose de son propre jeu d'instruction. La machine virtuelle ATL est indépendante du langage de transformation ATL. Elle ne s'intéresse qu'au code de la transformation compilée. Ce code contient des instructions pour la machine virtuelle ATL, il est aussi appelé *byte code*.

3.3.3 MIA-T

MIA-T – Model-In-Action Transformation [69] est un outil de transformation de modèle développé par Mia-Software, filiale de Sodifrance. C'est à la fois un moteur de transformation, un dépôt de métamodèles et un environnement de développement. Cependant, il est possible d'utiliser le moteur de transformation en dehors de cet environnement *via* l'API disponible.

MIA-T suit le modèle d'exécution tel qu'il est présenté sur la figure 3.2 page 27. Il se base sur un pseudo-MOF [80] maison comme métamétamodèle. MIA-T est livré en standard avec les métamodèles UML 1.3 et 1.4. Il est possible d'en générer des nouveaux grâce au projet MetaModel Generator [69] à partir d'un diagramme de classes UML.

Un projet de transformation est composé d'un ensemble de règles. Ces règles sont décomposées en deux partie : une requête qui sélectionne les éléments dans les modèles sources et une action qui créée et initialise les éléments dans les modèles cibles. Ces deux parties peuvent-être écrites dans des langages différents en fonction des besoins.

MIA-T possède trois langages pour l'écriture des transformations : deux langages déclaratifs, à savoir MIA-TL et RL-TL, ainsi que Java qui peut-être appelé pour des traitements complexes nécessitant un langage impératif. Il est possible d'appeler des services dans chacun de ces trois langages. Un service est une procédure en Java qui est factorisée et qui peut-être appelé en lui passant des paramètres typés. Ces services servent essentiellement de « super-macro fg dans les règles écrites en MIA-TL et RL-TL.

Outre ces capacités de bases, MIA-T est capable de composer des règles, c'est à dire des règles qui ne sont déclenchées que si la requête de la règle composante est activée.

3.3.4 Synthèse

La transformation de modèle est mise en œuvre dans la plupart des outils de manière relativement similaire. Elle se base sur un unique métamétamodèle plus ou moins proche du MOF [80] de l'OMG. Cependant, ce métamétamodèle est très lié à une conception par objets des modèles. Cette sémantique objet est parfois limitée pour l'expression d'une sémantique plus riche. Ainsi, des outils de transformation ont vu le jour en reprenant la notion de modèle telle qu'elle est définie dans le domaine de la représentation des connaissances, c'est à dire des graphes. C'est ce que nous allons voir dans la section suivante.

3.4 Transformation de graphes

La transformation de graphes est apparue à cause du manque d'expressivité des approches classiques de ré-écriture comme les grammaires de Chomsky et la ré-écriture de termes pour gérer les structures non-linéaires. Les premières propositions apparues à la fin des années soixante et au début des années soixante-dix [86, 72, 87]. Ces travaux s'occupaient de traductions de diagrammes et d'implémentations efficaces de λ-réduction basée sur des structures de graphes.

Les approches fondamentales qui sont encore populaires aujourd'hui incluent l'approche algébrique ou *double-pushout* (DPO) [32, 23], l'approche *node-label controlled* (NLC) [49, 34], la logique monadique du second ordre (*monadic second-order* – MSO) [24, 25] et l'approche PROGRES [93] qui représente la première application majeure de transformation de graphes dans le domaine de l'ingénierie des modèles.

3.4.1 Principes et fondations

Rappel sur les graphes

Le principe de la transformation de graphes dépend du type de graphes considéré. Si l'on considère les multi-graphes (qui admettent plusieurs arcs parallèles deux noeuds) alors les formalismes traitant de ces types de graphes sont connus comme des approches algébriques [23].

L'alternative la plus commune est de considérer les graphes comme des structures relationnelles. Dans ce cas, il existe au plus un arc entre une paire donnée de nœuds, ce qui est une restriction de la notion algébrique. D'autres variations incluent, par exemple, les graphes non orientés, ou les hypergraphes (souvent encodés comme des graphes bipartites).

Enfin, les graphes peuvent-être des graphes typés [22] où des contraintes structurelles sur les labels des noeuds et des arcs sont exprimées par un autre graphe.

Règles et transformations

En relation avec les deux principaux types de graphes précédemment cités, à savoir la considération algébrique et relationnelle, il y a deux approches fondamentales à la ré-écriture qui ont été référencées comme l'approche *gluing* et l'approche *connecting*. La différence se situe dans le mécanisme utilisé pour inclure la partie droite de la règle dans le contexte (c'est à dire la structure laissée après les suppressions dans la partie gauche). Dans une approche *gluing* comme [32, 64], la partie droite de la règle est « collée »suivant les noeuds en commun. Dans une approche type *connecting* comme NLC [49], l'inclusion est réalisée par une union disjointe, avec des arcs additionnels connectant la partie droite avec le contexte.

La plupart des approches pratiques utilise une combinaison de ces deux approches (*gluing* et *connecting*). En effet, la première est plus efficace mais la seconde a une expressivité supérieure.

La restriction la plus commune imposée par les règles de transformation de graphes est le degré de liberté du contexte des grammaires de graphe : une règle est hors-contexte si elle a un seul nœud ou arc dans sa partie gauche. De manière identique aux grammaires hors-contexte de Chomsky, les grammaires de graphes hors-contexte sont intéressantes par leur simplicité, combiné avec un bonne expressivité.

Nous avons donc vu que les systèmes de transformation de graphe généralisent les notions de ré-écriture, comme les grammaires de Chomsky, les réseaux de Pétri et la ré-écriture de termes. Dans la section suivante, nous allons faire un tour d'horizon des principaux outils qui utilisent ces notions.

3.4.2 Outils

Le premier exemple est Pogres – PROgrammed Graph REwriting Systems [93]. Conceptuellement, il fournit un langage graphique et textuel pour spécifier des structures de graphes et des transformations de graphes, des requêtes sur les graphes et des règles de ré-écriture avec des conditions complexes. De plus, il fournit un vérificateur de types qui détecte les incohérences vis à vis de la sémantique statique du langage. Le projet AGG – Attributed Graph Grammar citeagg, est basé sur les concepts de Pogres se basant donc sur une approche algébrique.

Un autre outil remarquable est AToM^3 – A Tool for Multi-Formalism and Meta-Modelling [26]. C'est un outil visuel de métamodélisation dont les modèles sont représentés en interne en utilisant une syntaxe abstraite de graphes. Ainsi, il est possible avec AToM^3 de manipuler les modèles avec des modèles de grammaire de graphes. Un système similaire est l'outil GReAT – Graph Rewriting and Transformation Language [52] qui s'inscrit dans la tendance MIC (Model-Integrated Computing) de l'université Vanderbilt. Citons aussi l'outil ViATra [99] faisant partie du projet GMT Eclipse.

Chapitre 4

Tissage de modèles

Résumé

Ce chapitre présente l'opération de tissage de schémas conceptuels. Cette opération de tissage est une extension de l'opération de transformation. Elle permet de mettre en place des opérateurs complexes tels que la composition, l'extension ou encore la fusion de modèles.

Sommaire

4.1 Introduction **36**
4.2 Métamodèle de tissage **37**
4.3 Tissage de graphes **39**
4.3.1 Techniques niveau élément 39
4.3.2 Techniques niveau structure 40
4.4 Opérateurs sur les tissages **42**
4.4.1 Calcul de différences entre modèles 42
4.4.2 Opérateur de composition 43
4.4.3 Opérateur de fusion 43

4.1 Introduction

Nous avons vu dans les chapitres précédents la notion de modèles dans différents domaines et le concept de transformation de ces modèles. Cette opération de transformation est certes centrale, mais il est vraissemblable qu'une autre opération existe : le tissage de modèle qui n'est pas une transformation de modèle au sens exécutable du terme [15].

Cette opération de tissage est souvent nommée *mapping* dans la littérature [66, 95, 65, 7, 48]. La variété des sens attribués à ce terme nous fait préférer celui de tissage puisqu'il s'agit bien de créer un ensemble de liens. De plus, il nous permet d'être indépendant du schéma conceptuel utilisé : graphes, réseaux sémantique, vision MDA/MDE, etc.

L'opération de tissage, étant toujours effectué avec un but précis, elle est rarerement un processus totalement automatisable. La plupart des outils existant sont en effet des outils graphique permettant d'éditer les liens de façon manuelle [45]. Cependant, il existe aujourd'hui des heuristiques permettant de fournir un tissage entre modèle dont les concepts sont similaires [67].

La plupart de ces outils se base sur des représentation des modèles sous forme de graphes comme présentés dans le chapitre 2 section 2.3.3. Cependant, il existe aussi des plate-formes spécifiques aux modèles de la vision MDE comme le projet ATLAS Model Weaver – AMW [28].

Dans ce chapitre nous allons tout d'abord présenter un métamodèle de tissage minimal et consensuel. Ensuite, nous verrons les principales techniques permettant d'établir un tissage entre graphes de façon semi-automatique. Enfin, nous donnerons une définition aux opérateurs qui vont de paire avec le tissage et qui sont souvent énoncés dans la littérature : les opérateurs de composition (*compose*), de fusion (*merge*), etc.

4.2 Métamodèle de tissage

A l'instar de la programmation orientée aspect qui a pour objectif la séparation des préoccupations dans le code, l'ingénierie dirigée par les modèles utilise la métamodélisation afin de représenter les différentes préoccupations des systèmes. Afin d'opérationnaliser ces préoccupations, il est nécessaire de les réunir, de les tisser. Il apparait que ce tissage entre les éléments affiche des liens sémantiques souvent semblables. L'idée est donc de fournir un métamodèle de tissage minimal et extensible permettant de servir de base pour des besoins plus spécifiques [28].

De manière naïve, un modèle de tissage est situé entre deux ensembles de modèles ou de métamodèles. Il définit un ensemble de liens entre les éléments de ceux-ci. De plus, entre certains liens peuvent exister des associations ou des contraintes.

Le métamodèle de la figure 4.1 est présenté dans [28] et décrite ci-dessous.

- *WElement* : élément de base. Tous les autres éléments l'étendent. Il a deux attributs : un nom et une description.

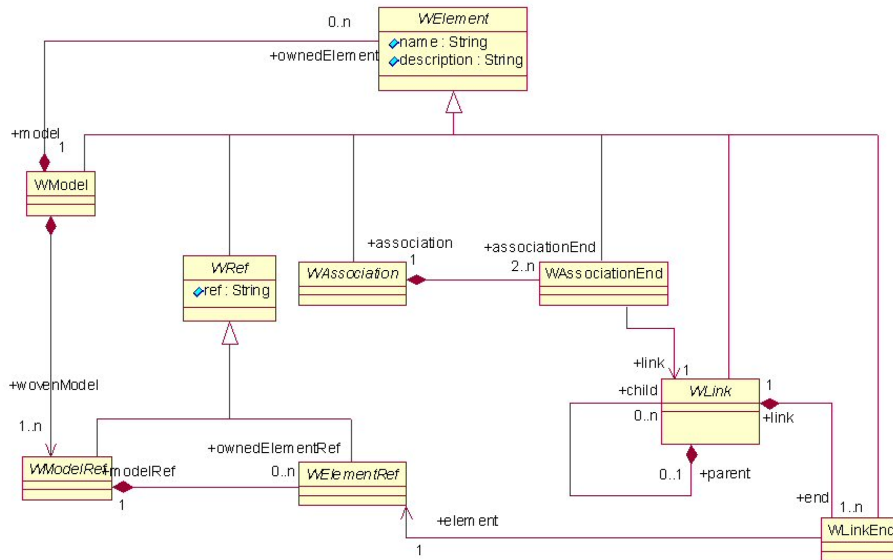

FIG. 4.1 – Métamodèle abstrait de tissage

- *WModel* : élément racine du métamodèle. Il est composé des éléments de tissages et des références vers les modèles tissés.
- *WLink* : représente le lien entre les éléments d'un élément d'un modèle. La référence *end* permet de lié un nombre quelconque d'élément. Les liens de tissages peuvent aussi être en relation avec d'autre liens pour créer une relation de contenance. Cet élément doit-être étendu afin d'ajouter différentes sémantiques au métamodèle.
- *WLinkEnd* : donne l'extrémité d'un lien, référencant le modèle tissé via un *WElementRef*.
- *WRef* : classe abstraite représentant les références.
- *WElementRef* : toute référence d'élément d'un modèle. L'attribut *ref* contient l'identifiant de l'élément tissé. Cet élément doit être étendu pour ajouter différents mécanismes d'identification des éléments.
- *WModelRef* : référence un modèle à tisser. Il est composé de référence d'éléments.
- *WAssociation* : utilisé pour créer des associations entre liens.
- *WAssociationEnd* : extrémité d'une association.

Avec ce métamodèle minimal il est possible de définir des métamodèles spécifiques à un objectif donné. Par exemple, pour tisser deux modèles dont les concepts sont similaires mais qui sont exprimés selon des métamodèles différents, il est possible de définir une classe de liens représentant l'égalité de concept.

Dans le domaine de tissage de graphes [95], un tissage consiste à déterminer un modèle de tissage MW entre deux graphes. Un modèle de tissage est un ensemble d'éléments de tissage W tel que W soit un 5-uplet (id, e, e', R, n) avec :

- id, l'identifiant du tuple,
- e et e' sont des éléments (ou des ensembles d'éléments selon la multiplicité du mapping) des graphes gauche et droit,
- R est le type de la relation qui unie e et e',
- n est un coefficient de confiance sur ce mapping (typiquement entre 0 et 1).

Le type de relation caractérisant un élément de tissage est une information sémantique et dépend du contexte de celui-ci.

4.3 Tissage de graphes

Dans cette section nous allons présenter brièvement quelles sont les techniques mises en oeuvre pour réaliser le tissage entre deux graphes. Nous commencerons par présenter les techniques basiques implémentées au niveau élément puis au niveau structure. Ensuite nous introduirons les processus de mise en correspondance avec leurs types et leurs stratégies. Enfin, nous ferons une liste des implémentations de ces techniques et des plate-formes de tissage de graphes.

4.3.1 Techniques niveau élément

Les techniques de tissage au niveau des éléments de graphes se décomposent en quatre catégories avec, pour chacune d'elles, des méthodes spécifiques.

Traitements des chaines de caractères. Ces techniques se basent sur la détection de préfixes ou de suffixes communs, de distances entre les chaînes, de l'algorithme LCS (Longest Common Substring [102]), ou encore les N-Gram [17].

Traitements au niveau langage. Ces techniques sont complémentaires aux précédentes mais sont dépendantes de la langue d'écriture de ces chaînes. Elles sont donc moins généralisables. Une des méthodes est le principe de découpage des mots (tokenisation) en reconnaissant la ponctuation et la casse. Par exemple, « PointDeVente »sera découpé en « point », « de », « vente », ou encore « ordre_fabrique »sera traduit en « ordre », « fabrique ». Une autre méthode est une analyse morphologique des termes afin d'en trouver des formes basiques : par exemple, enlever le pluriel sur les mots, les mettre tous au masculin. Cette méthode est très dépendante de la langue. Enfin, la technique d'élimination

des mots courants tels que les conjonctions de coordination, les articles et autres mots de liaison fait aussi partie des traitements possibles.

Prise en compte de contraintes. L'utilisation des types de données et des multiplicités des éléments permettent de classer ces éléments.

Traitements linguistiques. L'utilisation de relations léxicales avec des thésaurus (spécifique ou non au domaine), de dictionnaires classiques (en comptant le nombre de mots communs dans la définition par exemple) ou encore de dictionnaires sémantiques utilisant des relations telles que l'antonymie, l' holonymie l'homonymie (homophonie, homographie), l'hypéronymie, l'hyponymie, la méronymie, la pantonymie, la paronymie ou encore la synonymie.

4.3.2 Techniques niveau structure

Les techniques de tissage au niveau structure prennent en compte la structure des graphes afin de mettre en relation les éléments. Elles s'appliquent bien souvent après une première phase d'analyse utilisant les techniques au niveau élément. Elles se décomposent en trois catégories.

Techniques basées sur la taxonomie. Ces techniques prennent en compte l'aspect hiérarchique des éléments en essayant par exemple de mettre en correspondance des chemins bornés à l'intérieur des graphes (grâce à une précédente analyse des éléments de ces chemins). De même, elles utilisent les règles de sur et sous-concepts (si les concepts sémantiquement plus généraux correspondent, alors les sous-concepts correspondent). Enfin, la technique de calcul de ratio d'éléments communs dans la hiérarchie (« Upward Cotopic Distance ») permet d'obtenir une métrique sur les éléments du graphe et ainsi la comparer avec les éléments d'un autre graphe.

Techniques basées sur les graphes. Ces techniques utilisent la notion de noeuds feuilles et fils. Les conditions sont : si deux éléments du graphe sont non terminaux (pas des feuilles), et que leurs enfants sont très similaires, alors ils sont très probablement similaires ; si deux éléments du graphes sont non terminaux et que leurs feuilles sont très similaires, alors ils sont très probablement similaires. Les techniques inter-graphes se posent en ces termes : si deux éléments de deux graphes sont très similaires, il y a de grandes chances que leurs voisins soient similaires.

Techniques basées sur des modèles. Ces techniques se basent sur des modèles de raisonnement tels que SAT (la mise en correspondance des paires de noeuds est transformé

en formule propositionnelle) ou des modèles logiques en posant des relations entre les termes (relation d'équivalence, de généralisation, spécialisation, de disparité et de recouvrement...).

Processus, stratégies et ajustements finaux

Pour conclure cette énumération de techniques de matching, il nous reste à donner les différents processus et stratégies mis en oeuvre lors de cette opération.

Les différentes plate-formes adoptent des processus de matching généralement en deux phases. Ces deux phases sont organisées de manière séquentielle ou parallèle. Un processus séquentiel effectue d'abord un matching entre les deux graphes puis ré-utilise ce premier résultat et les graphes pour effectuer le tissage définitif. Un processus parallèle calcul le tissage entre le premier graphe et le second en parallèle de la même opération mais en inversant l'ordre des paramètres (en prenant d'abord le second et ensuite le premier). Enfin, la deuxième phase consiste à agréger ces deux tissages.

Ces processus de mise en correspondance utilisent des stratégies bien différentes pour la première phase. On retiendra notamment :

- la mise en relation comme une solution à un problème d'optimisation,
- la mise en relation comme un théorème dont le tissage satisfait les hypothèses,
- la production d'indices de similitude pour l'utilisateur final.

Enfin, la deuxième et dernière phase du processus d'alignement permet les ajustements finaux. Ces ajustements sont laissés à la charge de l'utilisateur la plupart du temps. Les principaux types d'ajustements sont :

- l'utilisation de seuils sur les indices de similitude et/ou des techniques de maximisation de vraisemblance,
- Le choix de cardinalité dans le tissage, par exemple que tout élément d'un graphe est mis en relation avec au moins un élément de l'autre graphe,
- le choix d'une direction (c'est à dire que la relation inverse n'est pas obligatoirement correcte).

Nous avons vu différentes techniques utilisées dans les études réalisées précédement sur le sujet du tissage de graphes. Dans les deux sections suivantes, nous allons faire un rapide tour d'horizon des implémentations de ces techniques et des plate-formes les mettant en oeuvre.

Systèmes de tissages de graphes

Nom	Types de techniques	Références
Artemis	Techniques basées sur le langage. Tissage des voisins par thesaurus commun.	[5, 6, 16]
COMA, COMA++	Techniques basées sur les chaînes de caractère. Thesaurus auxiliaires.	[4, 31, 90]
CtxMatch	–	[10, 11]
Cupid	Techniques basées sur les chaînes de caractères et langage. Tissage d'arbres pondérés par les feuilles.	[65]
Similarity Flooding (SF)	Techniques basées sur les chaînes de caractères et sur les types. Calcul itératif de points de difficulté.	[67, 66]
S-Match	Thesaurus externe (WordNet) et SAT.	[42, 43]

Infrastructure de matching/merging

Nom	Lien
Chimaera	http://www.ksl.stanford.edu/software/chimaera/
OntoMerge	http://cs-www.cs.yale.edu/homes/dvm/daml/ontology-translation.html
Rondo	http://www-db.stanford.edu/~melnik/mm/rondo/

4.4 Opérateurs sur les tissages

Nous avons vu dans les sections précédentes la notion de tissage de modèles ainsi que les techniques pour aboutir à ce tissage. Dans cette section, nous allons étudier ce qu'il est possible de faire avec ce tissage.

4.4.1 Calcul de différences entre modèles

Le calcul de différences entre modèles utilise depuis ses début un tissage entre les entités similaires, bien que cette notion de tissage ne soit pas explicite [114, 55]. En effet, la calcul d'un Δ entre deux modèles, passe d'abord par l'identification de ce qui n'à pas changé. Ce tissage peut se faire soit avec des identifiants uniques attribués aux entités comme dans [2], ou alors avec des heuristiques de tissage tentant de mettre en relief les similarités entre les modèles (voir les sections précédentes de ce chapitre ainsi que [66].

Les différences entre modèles identifient les ajouts et les suppressions en effectuant alternativement la différence entre le modèle gauche et le tissage puis entre le modèle droit et le tissage [55]. Les opérations plus complexes sur la structure du modèle sont prises en charges par des heuristiques se basant sur le métamodèle et ses contraintes [114].

4.4.2 Opérateur de composition

Afin de définir l'opérateur de composition, considérons l'exemple suivant : soit un tissage $m_1_m_2$ entre deux schémas conceptuels m_1 et m_2 tel que $m_1_m_2 \subseteq m_1 \times m_2$. Pour un schéma x conforme à m_1, le tissage génère un un schéma y conforme à m_2. Supposons que le schéma m_2 est modifié en m_3 et que nous avons donc $m_2_m_3 \subseteq m_2 \times m_3$. L'opérateur de composition permet la création d'un tissage $m_1_m_3$ noté $Compose(m_1_m_2, m_2_m_3)$ ou encore $m_1_m_2 \circ m_2_m_3$

4.4.3 Opérateur de fusion

Imaginons le scénario suivant. Soit m_1 et m_2 deux schémas conceptuels non disjoints. Par exemple, deux schémas de base de données d'étudiants, deux métamodèles de cinématique pour le Web ou encore deux schémas de documents scientifiques. On souhaite pouvoir fusionner les schémas m_1 et m_2 pour une raison quelconque permettant ainsi d'avoir un unique schéma commun m. Ce schéma se doit d'être minimal, c'est à dire ne retenant que les informations strictement nécessaires.

Soit le tissage $m_1_m_2$ décrivant comment m_1 est en relation avec m_2, c'est à dire que $m_1_m_2$ identifie tous les états mutuellement consistants $x \in m_1, y \in m_2$. En d'autres termes, chaque paire $(x, y) \in m_1_m_2$ représente un seul état valide $z \in m$. Les états x et y doivent-être assemblés dans z d'une telle façon qu'il est possible de les reconstruire à partir de z en utilisant deux tissages m_m_1 et m_m_2. De fait, m_m_1 et m_m_2 ne sont pas totales.

Chapitre 5

Incrémentalité

Résumé

Les traitements incrémentaux sont étudiés depuis quelques dizaines d'années maintenant. Leurs origines remontent aux méthodes de programmation introduite par Dijkstra [29] en 1976. Ensuite, ces techniques ont été surtout utilisées pour des problèmes d'optimisation. Les algorithmes incrémentaux ont ainsi été étudiés dans les domaines de solveurs de contraintes, la compilation de programmes ou encore la transformation de documents structurés. Dans ce chapitre, nous faisons une synthèse des principales méthodes, d'abord générales, c'est à dire applicables à un large spectre de problèmes, puis spécifiques à une problématique donnée.

Note : ce chapitre est largement inspiré de l'excellent travail effectué dans [40].

Sommaire

5.1 Introduction **45**
5.2 Définitions **45**
5.2.1 Caractérisation d'un algorithme incrémental 45
5.2.2 Rendre un traitement incrémental 46
5.3 Classement des travaux sur l'incrémentalité **46**
5.4 Principales méthodes générales **47**
5.4.1 Différence finie (Finite differencing) 47
5.4.2 INC 47
5.4.3 Mise en cache (Functiong caching) 47
5.4.4 Evaluation partielle (Partial evaluation) 48
5.4.5 Renforcement d'invariants (Strenghtening invariants for incrementalization) 49
5.5 Méthodes spécifiques **50**

5.5.1 Mise à jour d'entrepôts de données 50
5.5.2 Traduction incrémentale 51
5.5.3 XSLT incrémental 51

5.1 Introduction

Dans la langue française, le terme « incrémentiel »fait référence à ce « qui permet un traitement séquentiel immédiat des informations ». Dans de nombreux contextes en informatique, les modifications des données d'entrée doivent être traitées rapidement afin d'avoir un effet immédiat sur le résultat en sortie. Les techniques de traitements incrémental posent comme postulat fort qu'une modification minime des données d'entrée implique une faible modification des données de sortie. L'objectif que l'on peut se poser grâce à ce postulat est de calculer le nouveau résultat de manière incrémentale, c'est à dire en utilisant l'ancien résultat et en le mettant à jour plutôt que de recalculer entièrement le nouveau résultat. L'algorithme qui permet cette opération de mise à jour est dit incrémental.

Ainsi, ce postulat posé de façon empirique permet de rechercher des solutions générales au problème d'incrémentalité. Dans le cas de certains traitements où il n'est pas vérifié ou suffisament vérifiable, des techniques spécifiques doivent-être appliquées.

Le but des méthodes incrémentales est bien entendu l'optimisation de programmes informatiques couteux en temps de calcul. Mais cela ne veut pas dire que cette techniques sont sans coût. Par exemple, certaines techniques que nous allons voir plus loin stockent des résultats intermédiaires, ce qui requiert de la mémoire supplémentaire. Enfin, le développement de ces techniques est nettement plus complexe que les versions classiques des algorithmes.

5.2 Définitions

5.2.1 Caractérisation d'un algorithme incrémental

Le problème du traitement incrémental pourrait être exprimé comme suit : le but est de calculer une fonction f sur la donnée d'entrée x, où x est une structure de données bien identifiée et de maintenir le résultat $f(x)$ à jour lorsque la donnée d'entrée subit des modifications. Ainsi, un algorithme incrémental pour le calcul de f prend comme données d'entrée :

- l'entrée du traitement « batch »x ;
- le résultat du traitement « batch »$f(x)$;

- éventuellement des informations auxiliaires résultant de l'analyse statique ou dynamique du traitement « batch », de l'entrée x, ou du résultat $f(x)$
- les modifications Δx de l'entrée

L'algorithme incrémental calcule le nouveau résultat $f(x \oplus \Delta x)$ où $x \oplus \Delta x$ dénote l'entrée modifiée, et met à jour adéquatement les éventuelles informations auxiliaires.

Prenons par exemple le cas d'un programme effectuant la somme de n nombres. Nous avons donc le résultat $r = \sum_{i=1}^{n} x_i$. Imaginons que vous modifiez un nombre x_k. Le plus simple est de relancer le calcul afin de trouver le nouveau résultat r'. Un algorithme incrémental effectuerait la différence entre l'ancien x_k et le nouveau x'_k puis l'additionnerait à r pour donner r' comme ceci : $r' = r + (x'_k - x_k)$. Trois opérations au lieu de n, nous conviendrons que cette méthode est plutôt efficace. Ici, tout se passe pour le mieux dans le meilleur des mondes parce que les éléments que nous traitons sont dans $\mathbb{Z}$ et que cet ensemble muni de l'addition (au sens classique du terme) est un groupe. Dans le cas contraire, c'est à dire ou l'opération et l'ensemble de données en entrée ne forment pas un groupe, trouver un algorithme incrémental est nettement moins aisé.

5.2.2 Rendre un traitement incrémental

Une définition formelle du problème de « rendre un traitement incrémental »est donnée par [61] :

« Étant donné un programme f et une opération $\oplus$, un programme f' est appelé une version incrémentale de f par rapport à $\oplus$ si f' calcule $f(x \oplus \Delta x)$ de façon efficace en utilisant $f(x)$. »

Souvent, des informations auxiliaires (autres que le résultat $f(x)$) doivent être maintenues et sont nécessaires pour un calcul incrémental efficace de $f(x \oplus \Delta x)$. Un programme qui calcule de telles informations est appelé une version étendue de f. Ainsi, le but de calculer efficacement consiste à construire une version étendue d'un programme f, puis de dériver une version incrémentale de la version étendue pour une opération $\oplus$.

5.3 Classement des travaux sur l'incrémentalité

Un aperçu des différents travaux existants est donné par G. Ramalingam et T. Reps dans une bibliographie classifiée sur le traitement incrémental [91]. De nombreux autres travaux sont venus s'ajouter depuis. Les algorithmes incrémentaux sont notamment utilisés pour les environnements de programmation interactifs, pour la compilation de programmes, dans les résolveurs de contraintes, pour la transformation de documents structurés, etc.

Il est possible de diviser tous les travaux existants en deux catégories : les méthodes dites générales et les méthodes spécifiques à un problème. Chacune de ces catégories peut-être classée en deux autres catégories orthogonales : les approches par différence finie et les approches par recalcul sélectif.

5.4 Principales méthodes générales

5.4.1 Différence finie (Finite differencing)

Le terme « finite differencing »a été introduit par Paige et Koenig dans [84]. Leurs idées ont été appliquées par Paige dans son papier sur la programmation avec invariants [83]. Dans la programmation avec invariants, les programmes sont transformés en programmes incrémentaux en remplaçant les fonctions par des fonctions correspondantes incrémentales prises dans une librairie. Pour chaque fonction incrémentale f, un invariant $E = f(x_1, \ldots, x_n)$ est maintenu durant l'exécution du programme. Lorsque la valeur de l'un des arguments x_i change, la valeur de E est mise à jour au moyen d'une fonction de différence $f_{\Delta xi}$. Seule la dernière valeur de E est sauvegardée, ce qui limite la taille des informations stockées.

5.4.2 INC

INC est un langage conçu pour le traitement incrémental, par Yellin et Strom [115]. Il est basé sur la méthode « finite differencing ». Le langage INC comporte un certain nombres de fonctions prédéfinies pour lesquelles une fonction de différence existe. Un programme INC est un réseau de fonctions prédéfinies dans lequel le résultat d'une des fonctions sert de paramètre d'entrée pour la suivante. Lorsque un programme INC est exécuté avec un certain paramètre d'entrée x, le paramètre d'entrée et le résultat de chaque fonction sont sauvegardés. Lors de la prochaine exécution du programme avec la nouvelle donnée d'entrée $x + \Delta x$, la première fonction f du programme utilise f' pour calculer le nouveau résultat et la différence avec le précédent résultat. La différence de résultat est passée à la fonction suivante. Seule l'information du dernier calcul est sauvegardée.

5.4.3 Mise en cache (Functiong caching)

Introduite par Michie dans [70], « function caching »est une technique pour stocker les résultats des applications de fonctions en vue de les réutiliser lorsqu'une fonction est appelée pour la seconde fois avec les mêmes paramètres. Cette technique est souvent utilisée pour améliorer

la performance des fonctions récursives. Pugh et Teitelbaum [88] ont décrit comment utiliser la technique « function caching »pour l'évaluation incrémentale avec des structures de données. Dans le but d'obtenir des algorithmes incrémentaux efficaces, les structures de données doivent être conçues spécifiquement de manière à ce que les traitements sur des instances similaires du type de données aient en commun de nombreux sous-traitements. « function caching »est conceptuellement simple mais difficile à implanter efficacement. Une des difficultés est de trouver des manières de rechercher rapidement dans le cache. Lorsque les fonctions sont appelées avec de nombreux paramètres, comparer les paramètres de l'appel de la fonction en cours avec les informations présentes dans le cache devient très coûteux. Une autre difficulté est de conserver la taille du cache raisonnablement limitée. Il est nécessaire d'implanter des techniques pour purger le cache, qui conservent les résultats ayant une forte probabilité d'être réutilisés dans des traitements ultérieurs, et suppriment les autres informations.

5.4.4 Evaluation partielle (Partial evaluation)

L'évaluation partielle est une technique classique de transformation de programme pour spécialiser une fonction étant donné une partie connue de ses données d'entrée. Le résultat de cette technique s'appelle une fonction résiduelle, et a la propriété de donner les résultats souhaités lorsqu'elle est appliquée aux parties restantes des données d'entrée.

Une définition formelle de l'évaluation partielle au moyen des projections est donnée par Sundaresh dans [100]. Partant de cette définition, Sundaresh montre comment l'évaluation partielle peut être utilisée dans le cadre du traitement incrémental.

Le principe est que le domaine d'entrée d'une fonction peut être partitionné en utilisant un ensemble de projections. Une fois le domaine d'entrée partitionné, les fonctions résiduelles correspondant au partitionnement, c'est à dire à chaque projection, sont calculées. Ceci revient à stocker les portions du calcul qui ne dépendent que de chacune des parties du domaine d'entrée. Ainsi, lorsqu'une partie du domaine d'entrée change, il suffit d'appliquer à la nouvelle partie la fonction résiduelle correspondante. La difficulté principale est de trouver un bon partitionnement du domaine d'entrée pour pouvoir combiner correctement et efficacement les fonctions résiduelles [100].

5.4.5 Renforcement d'invariants (Strenghtening invariants for incrementalization)

Récemment, des efforts ont été menés pour tenter de dériver des programmes incrémentaux automatiquement (ou semi-automatiquement) à partir de programmes non incrémentaux écrits dans des langages de programmation classiques. Cette approche ambitieuse est nommée incrémentalisation [61]. Elle découpe le coeur du problème en trois sous-problèmes :

- l'exploitation du résultat précédent $f(x)$;
- le stockage, le maintien et l'exploitation de valeurs intermédiaires du calcul $f(x)$;
- et enfin la découverte et l'exploitation d'informations auxiliaires, c'est à dire des valeurs non calculées par $f(x)$.

Puisque les informations auxiliaires ne sont pas calculées par l'algorithme original qui calcule $f(x)$, le fait de les ajouter renforce les invariants vérifiés lors du calcul itératif qui utilise $f(x)$, d'où le nom de cette méthode générale.

L'idée basique en réponse au premier problème est d'identifier dans le calcul de $f(x \oplus y)$ les sous-calculs qui sont aussi effectués dans le calcul de $f(x)$ et dont les résultats peuvent être récupérés du cache r de $f(x)$. Le calcul de $f(x \oplus y)$ est symboliquement transformé pour éviter d'effectuer à nouveau ces sous-calculs en les remplaçant par les résultats correspondants [60].

En réponse au second problème, une méthode a été donnée dans [62] pour transformer statiquement les programmes pour stocker dans un cache tous les résultats intermédiaires utiles pour le traitement incrémental. L'idée basique est (I) tout d'abord d'étendre le programme f en un programme g qui renvoie tous les résultats intermédiaires, (II) incrémentaliser le programme g pour $\oplus$ pour obtenir une version incrémentale h de g en utilisant la méthode citéé plus haut pour le premier problème, et (III) d'analyser les dépendances dans h pour élaguer le programme étendu g en un programme i qui retourne seulement les résultats intermédiaires utiles, et enfin élaguer le programme h pour obtenir un programme f' qui maintient incrémentalement seulement les résultats intermédiaires nécessaires.

Concernant le troisème problème, une proposition pour trouver des informations auxiliaires est donnée dans [61].

Les deux idées clefs de cette proposition sont les suivantes :

- tout d'abord, considérer comme information auxiliaire éligible pour f tous les résultats intermédiaires d'une version incrémentale de f qui ne dépend que de x. Une telle version peut être obtenue grâce aux techniques citées précédemment.

– Etendre f avec toutes les informations auxiliaires éligibles, puis appliquer des techniques toujours issues des méthodes citées précédemment pour obtenir une version étendue et une version incrémentale qui ensemble calculent, exploitent et maintiennent seulement les résultats intermédiaires utiles et seulement les informations auxiliaires utiles.

La principale limite de cette technique est liée à son objectif : la méthode s'avère lourde à employer puisqu'elle tente de rendre incrémental un algorithme qui ne l'est pas.

5.5 Méthodes spécifiques

5.5.1 Mise à jour d'entrepôts de données

Les systèmes d'informations des entreprises peuvent aujourd'hui contenir des quantités astronomiques de données réparties dans des bases de données aux formats et aux schémas divers. Afin d'uniformiser ces sources, les données sont souvent aggrégées dans des entrepôts de données (*Data Warehouse*). Elles sont ensuite traitées par des requêtes complexes afin d'en avoir une vue synthétique et adaptée à la prise de décision. Les optimisations traditionnelles de requêtes ne permettent pas d'obtenir de temps de réponse raisonables. Une solution qui marche en pratique est la création de vues matérialisées. Celles-ci calculent et stockent physiquement le résultat des requêtes complexes afin d'éviter le recalcul de celles-ci à chaque demande.

Le problème survient lorsque les données sources de la vue sont modifiées. Cette dernière est alors périmée et nécessite d'être mise à jour. La maintenance incrémentale de ces vues est une technique éprouvée largement étudiée [46, 63, 1, 21, 44, 92, 117]. Ces travaux peuvent être classés selon deux axes : les méthodes algorithmiques et les méthodes algébriques.

Les méthodes algorithmiques dérivent un programme (la plupart du temps, un ensemble de requêtes SQL) dont l'évaluation maintient les vues [54, 18, 46]. Les principaux problèmes avec les techniques algorithmiques de maintenance de vues sont : (1) l'exactitude des algorithmes est dure à prouver et (2), la sortie des algorithmes de maintenance sont difficilement optimisables.

Afin de répondre à ces problèmes, des méthodes algébriques ont été développées. Plus précisément, une approche algébrique [44] prédéfinie un ensemble de règles primitives de propagation de changements pour chaque opérateur. L'opération de maintenance peut alors être construite en propageant les changements grâce à chaque opérateur dans l'arbre algébrique de la requête pour la vue en appliquant récursivement ces primitives. La sortie d'un tel algorithme peut-être optimisée par un optimisateur de requêtes basé sur le coût. De plus, étant basé sur de l'algèbre, le résultat n'est pas couplé à un langage particulier.

La maintenance de vues basée sur l'algèbre a été le sujet de travaux allant au delà des bases de données relationelles et des entrepôts de données. Ainsi, [30] maintient des vues XQuery en se basant sur une algèbre XML [116].

5.5.2 Traduction incrémentale

Un algorithme incrémental a été conçu autour de l'édition documentaire [59]. Il s'agit d'un algorithme incrémental de traduction d'arbre. Habituellement, lorsqu'un utilisateur saisit une chaîne de caractères, un *parsing* est effectué pour reconnaître cette chaîne. Ce parsing peut d'ailleurs lui même être incrémental. Les travaux de Lindén concernent la traduction incrémentale, c'est à dire, étant donné une grammaire d'entrée et une de sortie, traduire la chaîne saisie par l'utilisateur contrainte par une grammaire d'entrée en une chaîne contrainte par une autre grammaire de sortie.

Cette traduction est effectuée de manière incrémentale et réagit donc à chaque modification de la chaîne d'entrée. On pourrait considérer qu'une variante simplifiée de cet algorithme est présent de nos jours sur les téléphones cellulaires. En effet, certains téléphones permettent de saisir rapidement du texte à l'aide d'un nombre de touches inférieur au cardinal de l'alphabet réellement permis. Un contrôle incrémental de la saisie est effectué à l'aide d'un dictionnaire. L'algorithme de Lindén prend en entrée un arbre modifié, une liste des changements, une grammaire d'entrée et une grammaire de sortie et l'ancien arbre de sortie. L'algorithme calcule le nouvel arbre cible par mise à jour de l'ancien. La principale limite de cette approche réside dans le fait qu'elle est difficilement extensible car les grammaires considérées sont très restrictives.

5.5.3 XSLT incrémental

incXSLT

incXSLT est un processeur XSLT incrémental conçu par Lionel Villard et Nabil Layaïda [101]. La conception de ce processeur incrémental repose sur la modélisation du contexte d'exécution de XSLT et l'étude fine du langage de sélection XPath en définissant sa sémantique dénotationnelle.

Lors d'une mise à jour du document source, les principes mis en oeuvre dans le processeur incXSLT sont les suivants : il détecte dans un premier temps les expressions XPath qui ont besoin d'être ré-évaluées, il élabore ensuite un graphe de dépendance afin d'identifier le sous-ensemble de règles de transformation qu'il est nécessaire de ré-appliquer, il restaure le

contexte du document cible et celui du document source puis il peut enfin appliquer les règles sélectionnées auparavant. C'est donc une méthode d'incrémentalisation par recalcul sélectif.

Ces principes ont été validés par la création d'un processeur incrémental basé sur Xalan. Nous allons maintenant les détailler.

Au niveau des feuilles de transformation, on peut voir cela comme la sélection de toutes les instructions qu'il faut ré-exécuter. Ainsi ce processus repose principalement sur l'analyse des chemins XPath. A partir de ces chemins XPath nous pouvons donc sélectionner les instructions qu'il est nécessaire de ré-exécuter.

Pour chacune des instructions ainsi sélectionnées il s'agit dans un premier temps de rétablir son contexte d'exécution, et ce de la manière la plus efficace possible. Dans cet optique, il faut définir les données minimales nécessaires à la ré-execution de la transformation.

Dans le langage XSLT, un certain nombre d'instructions utilisent des expressions exprimées dans des attributs. Ces expressions sont généralement composées d'une ou plusieurs sous expressions de chemin.

L'évaluation de telles expressions dépend d'un contexte statique mais aussi d'un contexte dynamique. Dans le cas de la transformation incrémentale, il n'est nécessaire de restaurer que le contexte dynamique à chaque fois que l'on exécute une instruction. Ce contexte dépend de l'état du processeur au moment où l'expression est évaluée. La syntaxe des expressions est également utilisée afin de définir les règles de ré-évaluation.

Pendant une session incrémentale, les instructions qu'il faut réévaluer sont celles qui utilisent une expression dont la valeur a pu changer. En particulier cela peut arriver lorsqu'un attribut est modifié dans le document source.

Comme il est indiqué plus tôt, une expression de chemin sélectionne un ensemble de noeuds, dans la plupart des cas, les types de noeuds inclus dans cet ensemble peuvent être déterminés sans connaître le contexte dynamique. En fait les noeuds de l'ensemble sélectionné correspondent à une règle de ré-évaluation particulière. Cette règle est déduite de l'expression de chemin. Lorsqu'on la détermine il est indispensable d'y enlever toute référence à des éléments du contexte dynamique.

Durant une session de transformation incrémentale, il est nécessaire de restaurer le contexte dynamique afin de ré-exécuter les règles qui prendront alors en compte les modifications. Le contexte du document cible doit également être restauré.

Les noeuds de production contiennent des données liées au document cible. Ces données serviront à restaurer le contexte cible. Par exemple, l'instruction `element` à un lien vers l'élément

qu'elle génère. Pour un producteur de caractères, tel qu'une instruction `value-of` seul le nombre de caractères générés doit être stocké.

Après ces opérations, incXSLT créé un graphe de dépendances permettant d'identifier les relations entre différentes règles (`template`). En effet, l'exécution de l'instruction apply-template entraîne la recherche des templates qui correspondent à tous les noeuds sélectionnés. Cette recherche (l'opération d'instanciation) doit être reconsidérée à chaque fois que le document source est modifié.

Omega

Omega est un langage de transformation de documents XML permettant d'exécuter ces tranformations de manière incrémentale [40]. Comparé à d'autre langages tels que XSLT, ce langage a pour originialité d'utiliser XPath pour sélectionner les éléments dans l'arbre source mais aussi pour spécifier les éléments à créer dans l'arbre cible. Ceci permet d'obtenir une description précise des noeuds impliqués dans une règle. Outre cette spécificité, Omega se base sur un système de ré-écriture par *matching* des arbres source et cible. L'aspect incrémental du langage Omega se base sur des techniques de différences finies.

Les règles de transformations sont des triplets (N, R, ω) où N est un nom identifiant de manière unique la règle, R est une règle de ré-écriture de la forme suivante $l, r \rightarrow r'$ et enfin ω est un ensemble de contraintes associé à la règle. Les expressions l, r et r' du langage sont conformes à un sous-ensemble de XPath auquel a été ajouté le test d'égalité $=$.

L'opération de matching dans l'arbre source est exprimée par l qui renvoie, non pas un ensemble de noeuds comme une expression XPath classique, mais le premier sous-arbre qui correspond à cette expression.

La génération de l'abre cible utilise les mêmes expressions que l. Une expression r identifie le premier sous-arbre correspondant. r' définie comment ce sous-arbre cible doit-être modifié après l'opération de génération.

L'ensemble ω permet l'expression de contraintes par l'intermédiaire de variables dans les expressions l, r et r' liées par un ensemble d'équations.

Enfin, une règle $(N, (l, r \rightarrow r'), \omega)$ est applicable si et seulement si toutes les conditions suivantes sont vérifiées :

- l identifie au moins un sous-arbre de l'arbre source,
- ce sous-arbre n'a pas déjà été identifié auparavant par la règle N,
- chaque équation associée à la règle de transformation (chaque élément de ω) a une solution,

– r identifie au moins un sous-arbre de l'arbre cible.

Le but d'Omega est de fournir un langage de tranformation XML incrémentale. Pour ce faire, il suppose qu'une première génération a eu lieu en totalité et donc, que l'historique d'exécution est disponible. Voyons maintenant comment il traite les modifications sur l'arbre source.

Tout d'abord, le langage Omega se limite à quelques opérations élémentaires sur le document source :

– l'insertion d'un noeud feuille (élément, attribut ou texte),
– la suppression d'un noeud feuille (élément, attribut ou texte),
– la modification de la valeur d'un noeud texte (en particulier la valeur d'un attribbut).

Afin de savoir quelles sont les règles qui sont susceptibles d'être remises en cause par ces opérations, celles-ci sont analysées statiquement avant la première génération afin de les catégoriser. Ensuite, il s'agit d'identifier un noeud modifié. Omega génère une expression XPath absolue permettant d'identifier de manière unique ce noeud. Enfin, lors de la première transformation, des données auxiliaires sont générées afin de conserver les liens entre sous-arbre source et règles qui les identifient ainsi qu'entre règles et sous-arbres cibles. Il ne reste plus qu'a ré-exécuter les règles ainsi remises en cause et mettre à jour les noeuds cibles. L'algorithme est détaillé dans [40].

Le langage Omega permet donc la transformation incrémentale de documents XML avec une approche par différence finie. Cependant, il ne permet pas d'exprimer des règles très complètes notamment par son manque de support des prédicats de position de XPath. De plus, l'auteur a identifié des ambiguïtés de contexte dans l'arbre cible liées à la façon de spécifier celui-ci. Le problème provient du fait que la spécification d'un contexte dans l'arbre cible peut identifier plusieurs sous-arbres de l'arbre cible.

Chapitre 6

Traçabilité

Résumé

Dans l'état de l'art de la traçabilité, une grande partie du travail est effectué manuellement. De nombreux travaux ont été réalisés au sujet de la traçabilité, mais la plupart cible la traçabilité des besoins. La traçabilité dans le contexte de l'ingénierie des modèles est un problème global où il n'existe pas de solution unique. Nous avons identifié deux types de traçabilité : la traçabilité des transformations de modèles et la transformation des évolutions des modèles. Ces deux points de vue nécessitent un traitement différent.

Sommaire

6.1 Introduction **55**
6.2 Traçabilité des tranformations de modèles **56**
6.3 Traçabilité de l'évolution des modèles **57**
6.3.1 Traçabilité à priori 57
6.3.2 Traçabilité à posteriori 58

6.1 Introduction

La traçabilité est une préoccupation qui s'assure de conserver un ensemble de liens permettant dŠidentifier chronologiquement une entité lors lŠexécution dŠun processus la mettant en oeuvre.

La traçabilité est largement reconnue comme une préoccupation importante en ingénierie des logiciels et des systèmes d'informations reflétée par une littérature exhaustive, de nombreux

outils et un intérêt croissant pour la recherche dans ce domaine. Le bénéfice d'une traçabilité est bien acceptée aujourd'hui mais plusieurs problèmes posent encore des difficultés dans l'adoption à grande échelle dans les domaines d'ingénierie logicielle et notamment en ingénierie des modèles [58]. Il existe un cruel manque de définition communément acceptée en ce qui concerne la traçabilité, qui va au delà de la simple définition du terme comme dans [8]. De plus, il n'existe pas de manière standard de spécification de la traçabilité d'éléments quelconques. Il est intéressant de voir que des standards comme [81] et [82], largement utilisés dans l'industrie, ne fournissent pas de définition de cette notion. Une classification précise des types de traçabilité fait aussi défaut.

Nous tenons à faire remarquer au lecteur que les travaux concernant la traçabilité dans l'ingénierie des modèles se porte surtout sur la traçabilité des exigences [19, 73, 41] ou des fonctionnalités [3].

Malgré tout ces manques, nous allons tenter de fournir dans ce chapitre une analyse des approches utilisées pour traiter la traçabilité dans le domaine de l'ingénierie des modèles et notamment de la traçabilité des transformations de modèles mais aussi de l'évolution des modèles. Nous ferons tout d'abord un tour d'horizon de la gestion de la traçabilité dans le contexte de transformation de modèles. Enfin, nous étudierons la traçabilité de l'évolution de modèles.

6.2 Traçabilité des tranformations de modèles

Afin d'introduire la traçabilité des transformations de modèles, nous allons d'abord étudier comment le futur standard QVT gère celle-ci. Dans le dernier document de l'OMG [77], il est mentionné que le langage relation gère de lui même la traçabilité en créant une classe de trace pour chaque relation. Lors de l'utilisation du langage Core, il est nécessaire de créer soi-même ces classes entre les éléments de la transformation. Lorsque l'on regarde de plus près la transformation *Relation2Core* proposé en annexe, on se rend compte que les classes de trace sont très simples (pour ne pas dire simplistes). En effet, chacune d'entre elle contient uniquement une liste des éléments mis en jeu par la règle de transformation (aussi appelée relation).

Disons le tout net, cette définition ne nous satisfait pas. Le seul point positif que nous voyons est que la traçabilité est considérée comme externe à la transformation, créant un modèle supplémentaire. Cette considération est d'ailleurs une caractéristique qui différencie les approches de traçabilité. Ces approches sont soit fortement, soit faiblement couplées au moteur de transformation.

Le principe d'une traçabilité faiblement couplée est présenté dans le document [50]. Une traçabilité faiblement couplée est indépendant du moteur de transformation. En réalité, elle dépend de la capacité du moteur de transformation à générer plusieurs modèles cibles. Dans ce cas, pour chaque règle de transformation, il est possible de générer un ensemble de classes représentant la trace de l'exécution de cette règle. Cette liberté permet donc de s'adapter au besoin de traçabilité qui varie selon deux axes [50] : format et gamme. Le format est la façon dont la trace est stockée, la gamme est la quantité d'information à conserver pour la trace. Des métamodèles de traçabilité de transformation sont présentés dans [50, 9].

Lorsque la traçabilité est fortement couplée au moteur, sa réutilisation est nécessairement limitée. Elle sert principalement alors à des fins de déverminage. Quand bien même cette trace est récupérable d'une manière ou d'une autre sous la forme d'un modèle, sa modularité est moindre puisque son métamodèle n'est pas modifiable.

6.3 Traçabilité de l'évolution des modèles

La traçabilité est aussi une notion utilisée pour étudier l'évolution des modèles. En effet, quoi de plus naturelle que conserver une trace des modifications apportées à un modèle. Précédemment, nous conservions uniquement des liens entre les élements sources et cibles des transformations. Au delà des transformations, les modèles subissent d'autres opérations : refactoring, évolutions divers et variées.

Ainsi, il peut-être interressant de conserver une trace de toutes ces opérations afin, par exemple de permettre un versionnement des modèles via des opérations de différences [2]. Dans ce cas, deux écoles s'affrontent : soit l'on possède un environnement dédié qui permet de conserver toutes les opérations effectuées sur un modèle (y compris de très haut niveau), soit l'on tente de retrouver ces modifications entre deux versions d'un modèle, aboutissant ainsi à un modèle de *patch*.

6.3.1 Traçabilité à priori

Une traçabilité est dite à priori si les opérations réalisables sur un modèles sont précédement identifiées et leur enregistrement se fait à l'exécution de celles-ci. Cette traçabilité se base sur une formalisation des opérations possibles sur un modèle conforme à un métamodèle précis [68].

Les deux types de d'évolutions de modèles identifiés par ces techniques sont le raffinage de modèles et le *refactoring*. Le raffinement de modèles est utilisé pour transformer un modèle en

un autre modèle plus riche, plus détaillé (rajout de classes, d'attributs...). Le *refactoring* de modèles restructure un modèle en conservant sa « sémantique »(c'est à dire qu'il répond de la même façon aux même questions que l'on pouvait lui poser avant). Le but est d'améliorer le modèle en le rendant plus modulaire, réutilisable...

La formalisation des opérations est nécessaire afin de conserver un modèle conforme au métamodèle et de propager les modifications dans les contraintes des entités.

6.3.2 Traçabilité à posteriori

Une traçabilité à posteriori tente de retrouver les évolutions qui ont eues lieu sur un modèle en ayant auncune autre information que les deux versions de modèles. Cette traçabilité est en réalité un diff et nous renvoyons le lecteur au chapitre 4 page 36.

Chapitre 7

Bilan et perspectives

7.1 Bilan

Nous avons présenté tout au long de cette synthèse de bibliographie les différents domaines de recherche qui nous semblent intéressants pour répondre à notre problématique. Nous avons insisté sur l'incrémentalité et les divers pratiques s'y rattachant. Nous avons à partir de là identifié deux nouvelles problématiques. Tout d'abord, une problématique de traçabilité ; tantôt la traçabilité des transformations de modèles, tantôt de l'évolution des modèles. La traçabilité de l'évolution de modèles a débouché sur la seconde problématique, à savoir le calcul de différences entre modèles.

Nous avons alors étudié plus en détail les travaux de recherches se rapportant à ces problématiques. Nous avons découvert que peu de papiers ont été publiés sur ces sujets (notamment concernant la différence entre modèles), bien qu'au cœur de problématiques actuelles. Il nous parait donc évident de trouver d'abord des réponses à ces questions soulevées.

7.2 Perspectives

La suite de ce travail a déjà débuté. Elle a pour premier objectif la création d'une heuristique de tissage de modèles qui servira à effectuer l'opération de différence. Cette solution, si elle est viable, sera déjà un grand pas en avant vers une réponse à notre problématique initiale.

Nous avons conscience que le sujet est, dès le départ, très ambitieux pour un stage de Master de recherche. Ce que nous souhaitons pour la suite de ce travail est de faire une proposition théorique solide pour l'ensemble de la problématique. Cette proposition sera au fur et à me-

sure validée ou non grâce à de nombreuses expériences d'implémentation. Les résultats de ces expériences seront consignés dans la prochaine partie de ce rapport.

Nous espérons ainsi fournir quelques solutions à des problèmes précis et un scénario global, peut-être moins détaillé, mais qui devra constituer une base solide pour la continuation du projet.

7.3 Remerciements

Je tiens à grandement remercier Jean Bézivin et Erwan Breton pour leur discernement sur le sujet ainsi que leur aide précieuse dans la rédaction de ce rapport.

Références

[1] D. Agrawal, A. El Abbadi, A. Singh and T. Yurek. « Efficient ViewMaintenance at DataWarehouses ». In *Proceedings of SIGMOD*, pages 417–427, 1997. 50

[2] M. Alanen and I. Porres. « Difference and Union of Models ». In Perdita Stevens, Jon Whittle and Grady Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, Oct 2003. 42, 57

[3] G. Antoniol, E. Merlo, Y.G. Guéhéneuc and H. Sahraoui. « On feature traceability in object oriented programs ». In *TEFSE '05 : Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, pages 73–78, New York, NY, USA, 2005. ACM Press. 56

[4] D. Aumüller, H. H. Do, S. Massmann and E. Rahm. « Schema and ontology matching with COMA++ ». In *In Proceedings of the International Conference on Management of Data (SIGMOD), Software Demonstration*, 2005. 42

[5] D. Beneventano, S. Bergamaschi, S. Lodi and C. Sartori. « Consistency checking in complex object database schemata with integrity constraints ». In *IEEE Transactions on Knowledge and Data Engineering*, volume 10, pages 576–598, 1998. 42

[6] S. Bergamaschi, S. Castano and M. Vincini. « Semantic integration of semistructured and structured data sources ». In *SIGMOD Record*, volume 28, pages 54–59, 1999. 42

[7] P. Bernstein, S. Melnik, M. Petropoulos and C. Quix. « Industrial-Strength Schema Matching ». In *SIGMOD Record*, volume 33, 2004. 37

[8] IEEE Standards Board. « IEEE Std 610.12-1990 IEEE standard glossary of software engineering terminology », 1990. 56

[9] L. Bondé, P. Boulet and J-L. Dekeyser. « Traceability and Interoperability at Different Levels of Abstraction in Model Transformations ». In *In FDL05*, 2005. 57

[10] P. Bouquet, B. Magnini, L. Serafini and S. Zanobini.. « A SAT-based algorithm for context matching ». In *In Proceedings of the International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT)*, pages 66–79, 2003. 42

[11] P. Bouquet, L. Serafini and S. Zanobini.. « Semantic coordination : A new approach and an application ». In *In Proceedings of the International Semantic Web Conference (ISWC)*, pages 130–145, 2003. 42

[12] J. Bézivin. « In Search of a Basic Principle for Model Driven Engineering ». *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, V(2) :21–24, 2004. 17

[13] J. Bézivin, H. Brunelière, F. Jouault and I. Kurtev. « Model Engineering Support for Tool Interoperability ». In *Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005)*, Montego Bay, Jamaica, 2005. 2

[14] J. Bézivin, G. Dupé, F. Jouault, G. Pitette and J. E. Rougui. *First experiments with the ATL model transformation language : Transforming XSLT into XQuery*. Atlas Group, INRIA, IRIN and TNI-Valiosys, 2003. 1

[15] J. Bézivin, F. Jouault and P. Valduriez. « First Experiments with a ModelWeaver ». In *Proceedings of the OOPSLA/GPCE : Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004. 36

[16] S. Castano, V. De Antonellis and S. De Capitani di Vimercati. « Global viewing of heterogeneous data sources ». In *IEEE Transactions on Knowledge and Data Engineering*, volume 13, pages 277–297, 2001. 42

[17] W. B. Cavnar and J. M. Trenkle. « N-Gram-Based Text Categorization ». In *Proceedings of the 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, Las Vegas, USA, 1994. 39

[18] S. Ceri and J. Widom. « Deriving Production Rules for Incremental View Maintenance ». In *Proceedings of VLDB*, 1991. 50

[19] J. Champeau and E. Rochefort. « Model Engineering and Traceability ». In *Model Driven Architecture in the Specification, Implementation and Validation of Object-oriented Embedded Systems – Conférence UML 2003*, 2003. 56

[20] M. Chein. « Special Issue on Conceptual Graphs ». *Revue d'Intelligence artificielle*, 10(1), 1996. 8

[21] L. S. Colby, R. L. Cole, E. Haslam, N. Jazayeri, G. Johnson, W. J. Mckenna, L. Schumacher and D. Wilhite. « Redbrick Vista : Aggregate Computation and Management ». In *Proceedings of ICDE*, pages 174–177, 1998. 50

[22] A. Corradini, U. Montanari and F. Rossi. « Graph processes ». In *Fundamenta Informaticae*, volume 26, pages 241–266, 1996. 34

[23] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe. « Algebraic approaches to graph transformation, Part I : Basic concepts and double pushout approach ». In *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1 : Foundations*, pages 163–245, 1997. 33, 34

[24] B. Courcelle. « The monadic second-order logic of graphs I, recognizable sets of finite graphs ». In *Information and Computation*, number 8521, pages 12–75, 1990. 33

[25] B. Courcelle. « The expression of graph properties and graph transformations in monadic second-order logic ». In *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1 : Foundations*, 1997. 33

[26] J. de Lara and H. Vangheluwe. « AToM3 : A Tool for Multi-Formalism Modelling and Meta-Modelling ». In *Lecture Note in Computer Science*, volume 2306, pages 174–188, 2002. 35

[27] L. Dickson, H. Delugach, M. Keeler, L. Searle and J. Sowa. « Conceptual Structures : Fulfilling Peirce's Dream ». *Lecture Notes in AI 1257*, 1997. 8

[28] M. Didonet Del Fabro, J. Bézivin, F. Jouault, E. Breton and G. Gueltas. « AMW : a generic model weaver ». In *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05)*, 2005. 37

[29] E. W. Dijkstra. « A Discipline of Programming ». *Prentice Hall Series in Automatic Computation*, 1976. 44

[30] K. Dimitrova, M. El-Sayed and E. A. Rundensteiner. « Order-Sensitive View Maintenance of Materialized XQuery Views ». In *International Conference on on Conceptual Modeling*, pages 144–157, 2003. 51

[31] H. H. Do and E. Rahm. « COMA - a system for flexible combination of schema matching approaches ». In *In Proceedings of the Very Large Data Bases Conference (VLDB)*, pages 610–621, 2001. 42

[32] H. Ehrig, M. Pfender and H. J. Schneider. « Graph grammars : an algebraic approach ». In IEEE, editor, *14th annual IEEE Symposium on Switching and Automata Theory*, pages 167–180, 1973. 33, 34

[33] Peter W. Eklund, E. Gerard and M. Graham. « Conceptual Structures : Knowledge Representation as Interlingua ». *Lecture Notes in AI 1115*, 1996. 8

[34] J. Engelfriet and G. Rozenberg. « Node replacement graph grammars ». In *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1 : Foundations*, pages 1–94, 1997. 33

[35] J-M Favre. « IDM : Les concepts fondamentaux (PRESENTATION Rapport AS MDA) ». In *In Actes IDM05*, 2005. 16

[36] J.M. Favre. « Towards a Basic Theory to Model Driven Engineering ». In *3rd Workshop in Software Model Engineering*. WiSME, 2004. 17

[37] Eclipse Foundation. « Eclipse Modeling Framework ». http://www.eclipse.org/emf/. 32

[38] Eclipse Foundation. « Eclipse Website ». http://www.eclipse.org. 32

[39] M. Genesereth. « Knowledge Interchange Format - draft proposed American National Standard ». http://logic.stanford.edu/kif/dpans.html, 1998. (dpANS) NCTIS.T2/98-004. 9

[40] P. Genevès. « Langages de transformation incrémentale ». Master's thesis, INRIA Rhône-Alpes, 2003. 44, 53, 54

[41] M. Gills. « Survey of Traceability Models in IT projects ». In *In Proceedings of ECMDA Traceability Workshop (ECMDA-TW)*, Nuremberg, Germany, 2005. 56

[42] F. Giunchiglia and P. Shvaiko. « Semantic matching ». In *The Knowledge Engineering Review Journal (KER)*, volume 3, pages 265–280, 2003. 42

[43] F. Giunchiglia and M. Yatskevich. « Element level semantic matching ». In *In Proceedings of the Meaning Coordination and Negotiation workshop at the International Semantic Web Conference (ISWC)*, 2004. 42

[44] T. Griffin and L. Libkin. « Incremental Maintenance of Views with Duplicates ». In *Proceedings of SIGMOD*, pages 328–339, 1995. 50

[45] J.C. Grundy, J.G. Hosting, R.W. Amor, Mugridge, W.B. and Y. Li. « Domain-Specific Visual Languages for Specifying and Generating Data Mapping Systems ». In *Journal of Visual Languages and Computing*, volume 15, pages 207–209, 2004. 37

[46] A. Gupta, I. S. Mumick and V. S. Subrahmanian. « Maintaining Views Incrementally ». In *Proceedings of SIGMOD*, pages 157Ű–166, 1993. 50

[47] D. Harel and B. Rumpe. « Meaningful modeling : What's the semantics of « semantics » ? ». *Computer*, 37(10) :64–72, oct 2004. 11

[48] W. Hu, N. Jian, Y. Qu and Y. Wang. « GMO : A Graph Matching for Ontologies ». In *Proceedings of K-Cap 2005 Workshop on Integrating Ontologies*, pages 43–50, Banff, Canada, 2005. 37

[49] D. Janssens and G. Rozenberg. « On the structure of node-label controlled graph grammars ». In *Information Science*, volume 20, pages 191–216, 1980. 33, 34

[50] F. Jouault. « Loosely Coupled Traceability for ATL ». In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, Nuremberg, Germany, 2005. 57

[51] F. Jouault and B. Bézivin. « KM3 : a DSL for Metamodel Specification ». In *FMOODS*, Bologne, Italie, jun 2006. 17, 32

[52] Shi F. Karsai G., Agrawal A. « On the Use of Graph Transformations for the Formal Specification of Model Interpreters ». In *Journal of Universal Computer Science*, volume 9, pages 1296–1321, 2003. 35

[53] S. Klein and R. F. Simons. « Syntactic dependence and the computer generation of coherent discourse ». *Mechanical Translation*, 7, 1963. 7

[54] S. Koenig and R. Paige. « A Transformational Framework for the Automatic Control of Derived Data ». In *Proceedings of VLDB*, pages 306–318, 1981. 50

[55] D. Kolovos, R. Paige and F. Polack. « Model Comparison : a Foundation for Model Composition and Model Transformation Testing ». In *to appear in Proc. First International Workshop on Global Integrated Model Management (G@MMA) 2006*, may 2006. 42

[56] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E Perry and W. M. Turski. « Metrics and laws of software evolution - The nineties view ». In *Proc. Int'l Symposium Software Metrics*, San Diego, USA, 1997. IEEE Computer Society. 1

[57] R. Lemesle. « *Techniques de modélisation et de Méta-modélisation* ». PhD thesis, Laboratoire de Recherche en Sciences de Gestion (L.R.S.G.), Université de Nantes, 2000. 11

[58] A. Limťon and J. Garbajosa. « The Need for a Unifying Traceability Scheme ». In *In ECMDA Workshop - Traceability*, 2005. 56

[59] Greger Lindén. « Incremental Updates in Structured Documents », 1994. 51

[60] Y. A. Liu, L. Hongjun, L. Feng and F. Hussain. « Efficient Search of Reliable Exceptions ». In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 194–203, 1999. 49

[61] Y. A. Liu, S. D. Stoller and T. Teitelbaum. « Strengthening invariants for efficient computation ». *Science of Computer Programming*, 41(2) :139–172, 2001. 46, 49

[62] Y. A. Liu and T. Teitelbaum. « Systematic Derivation of Incremental Programs ». *Science of Computer Programming*, 24(1) :1–39, 1995. 49

[63] J. J. Lu, G. Moerkotte, J. Schue and V. S. Subrahmanian. « Efficient Maintenance of Materialized Mediated Views ». In *Proceedings of SIGMOD*, pages 340–351, 1995. 50

[64] M. Löwe. « Algebraic approach to single-pushout graph transformation ». In *Theoret. Comput. Sci.*, volume 109, pages 181–224, 1993. 34

[65] J. Madhavan, P. Bernstein and E. Rahm. « Generic Schema Matching Using Cupid ». In *Proceedings of the 27th International Conferences on Very Large Databases*, pages 49–58, 2001. 37, 42

[66] S. Melnik. « *Generic Model Management : Concepts and Algorithms* ». PhD thesis, University of Leipzig, 2004. 37, 42

[67] S. Melnik, H. Garcia-Molina and E. Rahm. « Similarity Flooding : A Versatile Graph Matching Algorithm and Its Application to Schema Matching ». In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, page 117, 2002. 37, 42

[68] T. Mens and R. Van Der Straeten. « On the Use of Formal Techniques to Support Model Evolution ». In *Proc. Ingénierie Dirigée par les Modèles (IDM 05)*, 2005. 57

[69] MIA-Software. « MIA-Transformation and MIA-Generation website ». `http://www.mia-software.com`. 32

[70] D. Michie. « Memo functions and machine learning ». *Nature*, 218 :19–22, 1968. 47

[71] J. Miller and J. Mukerji. « MDA Guide Version 1.0.1 (omg/2003-06-01) », 2003. 14

[72] U. Montanari. « Separable graphs, planar graphs and web grammars ». In *Information and Control 16*, pages 243–267, 1970. 33

[73] L. Naslavsky, T. A. Alspaugh, D. J. Richardson and H. Ziv. « Using scenarios to support traceability ». In *TEFSE '05 : Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, pages 25–30, New York, NY, USA, 2005. ACM Press. 56

[74] NetBeans. « Metadata Repository (MDR) Project Home ». http://mdr.netbeans.org/. 32

[75] Object Management Group. http://www.omg.org. 14

[76] Object Management Group. « A Proposal for an MDA Foundation Model ». *Object and Reference Model Subcommittee (ORMSC) of the OMG Architecture Board*, August 2001. 18

[77] Object Management Group. « Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (ptc/05-11-01) », 2005. 16, 27, 31, 56

[78] Object Management Group. « OCL 2.0 Specification Version 2.0 ptc/2005-06-06 », 2005, 2006. 27

[79] Object Management Group. « Unified Modeling Language (UML) Version 2 (formal/05-07-05, formal/05-07-04) », 2005, 2006. 15, 27

[80] Object Management Group. « Meta Object Facility (MOF) Core Specification version 2.0 (formal/2006-01-01) », 2006. 16, 27, 32, 33

[81] ISO International Standard Organization. « (ISO/IEC 12207) Information Technology – Software Lifecycle Processes », 1995. 56

[82] ISO International Standard Organization. « (ISO/IEC 15288) Information Technology – Systems Engineering – System life cycle processes », 2002. 56

[83] R. Paige. « Programming With Invariants ». *IEEE Software*, pages 56–69, jan 1986. 47

[84] R. Paige and S. Koenig. « Finite differencing of computable expressions ». *ACM Trans. on Program. Lang. Syst.*, 4(3) :402–454, July 1982. 47

[85] C. S Peirce. *Collected Papers of Charles Sanders Peirce*. Harvard University Press, Cambridge, 1931–1935. 8

[86] J. L. Pfaltz and A. Rosenfeld. « Web grammars ». In *Int. Joint Conference on Artificial Intelligence*, pages 609–619, 1969. 33

[87] T. W. PRATT. « Pair grammars, graph language and string-to-graph translations ». In *Journal of Computer and System Sciences*, volume 5, pages 560–595, 1971. 33

[88] W. PUGH and T. TEITELBAUM. « Incremental computation via function caching ». In *the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, New York, USA, January 1989. 48

[89] M. R. QUILLIAN. « Semantic Memory ». In Marvin MINSKY, editor, *Semantic Information Processing*, pages 216–270, Cambridge, MA, 1968. MIT Press. 7

[90] E. RAHM, H. H. DO and S. MASSMANN. « Matching large XML schemas ». In *SIGMOD Record*, volume 4, pages 26–31, 2004. 42

[91] G. RAMALINGAM and T. REPS. « A categorized Bibliography on Incremental Computation ». In *20th Annual ACM Symposium on Principles of Programming*, pages 502–510, New York, jan 1993. 46

[92] K. SALEM, K. S. BEYER, R. COCHRANE and B. G. LINDSAY. « How To Roll a Join : Asynchronous Incremental View Maintenance ». In *Proceedings of SIGMOD*, pages 129–140, 2000. 50

[93] A. SCHÜRR, A. J. WINTER and A. ZÜNDORF. « The PROGRES approach : Language and environment ». In *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2 : Applications, Languages, and Tools*, pages 487–550, 1999. 33, 35

[94] E. SEIDEWITZ. *What do models mean ?*, volume 20. IEEE Software, sep–oct 2003. 17

[95] P. SHVAIKO and J. EUZENAT. « Tutorial on Schema and Ontology Matching ». In *ESWC'05*, 2005. 37, 39

[96] John F. SOWA. « Conceptual graphs for a database interface ». *IBM Journal of Research and Development*, 20(4) :336–357, 1976. 8

[97] John F. SOWA. *Conceptual Structures : Information Processing in Mind and Machine*. Addison-Wesley, 1984. 8

[98] A. STAIKOPOULOS and B. BORDBAR. *A Metamodel Refinement Approach for Bridging Technical Spaces, a Case Study*. School of Computer Science, University of Birmingham, 2005. 1

[99] GMT SUBPROJECT. « ViATra 2 ». `http://www.eclipse.org/gmt/`. 35

[100] R. S. Sundaresh and P. Hudak. « Incremental computation via partial evaluation ». In *the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, New York, USA, January 1991. 48

[101] L. Villard. « *Modèles de documents pour l'édition et l'adaptation de présentations multimédias* ». PhD thesis, Institut National Polytechnique de Grenoble, 2002. 51

[102] The Free Encyclopedia Wikipedia. « Longest common substring problem ». http://en.wikipedia.org/wiki/Longest_common_substring_problem. 39

[103] World Wide Web Consortium. « Document Object Model Technical Reports ». http://www.w3.org/DOM/DOMTR. 24

[104] World Wide Web Consortium. « Extensible Markup Language (XML) 1.0 (Third Edition) ». http://www.w3.org/TR/REC-xml/. 12

[105] World Wide Web Consortium. « XML Path Language (XPath) Version 1.0 ». http://www.w3.org/TR/SVG12/. 23, 24

[106] World Wide Web Consortium. « XML Schema for XML Schema (normative) ». http://www.w3.org/TR/xmlschema-1/#normative-schemaSchema. 14

[107] World Wide Web Consortium. « XML Schema Part 2 : Datatypes Second Edition ». http://www.w3.org/TR/xmlschema-2/. 26

[108] World Wide Web Consortium. « XML Syntax for XQuery 1.0 (XQueryX) ». http://www.w3.org/TR/xqueryx/. 25

[109] World Wide Web Consortium. « XQuery 1.0 : An XML Query Language ». http://www.w3.org/TR/xquery/. 25

[110] World Wide Web Consortium. « XQuery 1.0 and XPath 2.0 Data Model ». http://www.w3.org/TR/xpath-datamodel/. 26

[111] World Wide Web Consortium. « XQuery 1.0 and XPath 2.0 Formal Semantics ». http://www.w3.org/TR/xquery-semantics/. 26

[112] World Wide Web Consortium. « XSL Transformations (XSLT) Version 1.0 ». http://www.w3.org/TR/xslt. 23, 24

[113] World Wide Web Consortium, W3C. http://www.w3.org/. 12

[114] Z. Xing and E. Stroulia. « UMLDiff : an algorithm for object-oriented design differencing ». In *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on*

Automated software engineering, pages 54–65, New York, NY, USA, 2005. ACM Press. 42

[115] D. YELLIN and R. STROM. « INC : A Language for Incremental Computations ». *ACM Transactions on Programming Languages and Systems*, 13(2) :211–236, April 1991. 47

[116] X. ZHANG, B. PIELECH and E. A. RUNDENSTEINER. « Honey, I shrunk the XQuery ! : an XML Algebra Optimization Approach ». In *Web Information and Data Management (WIDM)*, pages 15–22, 2002. 51

[117] Y. ZHUGE, H. GARCÍA-MOLINA, J. HAMMER and J. WIDOM. « View Maintenance in a Warehousing Environment ». In *Proceedings of SIGMOD*, pages 316–327, May 1995. 50