# Transformations de modèles, basées sur XSLT.

Benoit Georges

Computer Science Department, University of Namur,
Rue Grandgagnage 21,
5000 Namur, Belgique
bgeorges@info.fundp.ac.be
benoit.georges@belgacom.net

**Résumé.** The abstract should summarize the contents of the paper and should contain at least 70 and at most 150 words. It should be set in 9-point font size and should be inset 1.0 cm from the right and left margins. There should be two blank (10-point) lines before and after the abstract. …

## 1 Introduction

Il est admis désormais de distinguer dans une application logicielle son modèle (abstrait issu des phases d'analyse et de conception) et le code exécutable qui la réalise. Longtemps, ces modèles sont restés des entités informelles. Mais la révolution que la programmation par objets a provoquée en génie logiciel se poursuit avec l'émergence d'une ingénierie des modèles. Celle-ci a permis de mettre en lumière des formalismes employés pour écrire ces modèles. Chacune des méthodes d'analyse et de conception par objet avait le sien mais UML (Unified Modeling Language), entériné par l'OMG (Object Management Group), s'est imposé en tant que synthèse [13].

Dès lors, les transformations de modèles ont fait l'objet d'une vive attention et ont été un thème récurrent dans la littérature de ces dernières années, dont la bibliographie de cet article en est la preuve. De plus, les transformations de modèles sont au cœur des applications de la méta-modélisation au génie logiciel que fournissent les méta-outils (Meta-Case Tools), depuis GraphTalk, MétaEdit, jusqu'à UMLAUT, et MétaGen.

Cet article a premièrement pour objectif de présenter la problématique des transformations de modèles en définissant les buts qu'elles poursuivent et en décrivant leur différentes classes. Ensuite, le chapitre suivant de cet article permet de mieux comprendre comment les transformations sont intégrées à l'organisation de modélisation en couche et de mettre en évidence la possibilité de baser les transformations de modèles sur les méta-modèles. Les trois chapitres suivants décrivent ensuite une organisation particulière de modélisation, celle de l'OMG, ainsi que la présence de deux niveaux d'abstraction dans l'organisation de modélisation en

générale et plus particulièrement dans celle de l'OMG. Enfin, ils font remarquer la nécessité de conserver ces deux niveaux dans la constitution de transformation. Les trois derniers chapitres détaillent enfin deux langages permettant de spécifier des transformations, XSLT et MTRANS, correspondant respectivement à deux niveaux d'abstraction ; ainsi que la possibilité d'établir une corrélation entre ces deux langages.

## 2 Transformations de modèles : définition, objectifs et types

Les transformations de modèles spécifient les relations entre deux modèles distincts, c'est-à-dire comment créer un modèle à partir des informations fournies par un autre modèle. Alternativement, les relations peuvent être envisagées avec plusieurs modèles à l'origine et plusieurs modèles à l'arrivée [12].
Les transformations de modèles permettent d'accomplir plusieurs objectifs. Premièrement, elles offrent un moyen de traduction entre des modèles représentant le même système selon la même perspective mais respectant des formalismes différents. L'utilisation de formalismes différents est justifiée par le fait que les formalismes ont des caractéristiques différentes et aucun d'eux n'est meilleur que tous les autres dans toutes les situations de modélisation. Les formalismes ont une expressivité, une facilité ( c'est-à-dire la possibilité offerte aux utilisateurs de représenter aisément et directement la réalité grâce à ses concepts ) et une précision différente. Par exemple, un langage peut permettre des analyses de modèles que d'autres langages ne peuvent réaliser ; ou la communication des modèles aux humains peut être plus aisée avec certains formalismes qu'avec d'autres. Ces différences induisent que certains formalismes peuvent être utilisés dans certains cas mais que dans d'autres cas leur mise en œuvre n'est pas possible ou difficile. Une approche multi-formaliste peut être utilisée afin de traiter les problèmes complexes en profitant des caractéristiques complémentaires des différents formalismes [5]. L'emploi de plusieurs formalismes est aussi motivé par le fait que les méta-outils proposent des formalismes différents, tels que GOPRR pour MétaEdit, UML pour UMLAUT, PIR3 pour MétaGen [13].
En outre, les transformations fournissent un moyen de transition entre des modèles représentant le même système mais selon des perspectives différentes et respectant un même formalisme ou des formalismes différents. Par exemple, on peut imaginer des transformations fournissant un moyen d'obtenir un diagramme de collaboration représentant la dynamique d'objets à partir d'un diagramme de séquences représentant un cas d'utilisation.
Enfin, les transformations peuvent également servir lors de la ré-ingénierie. Elles procurent un moyen de transformation entre des modèles représentant des systèmes dont des détails de conception diffèrent. Elles permettent la simplification et l'optimisation des modèles ainsi que tout autre raffinement pouvant être exprimé comme un ensemble de transformations entre des modèles [2].

On peut communément classifier les transformations dans plusieurs catégories selon le gain ou la perte d'informations qu'elles engendrent.
Premièrement, les transformations d'équivalence permettent de s'acheminer d'un modèle source à un modèle cible, de telle manière que chaque instance du modèle

source correspond exactement à une instance du modèle cible. Et inversement, chaque instance du modèle cible correspond exactement à une instance du modèle source. Deuxièmement, les transformations avec gain d'informations mettent en relation un modèle source et un modèle cible mais quelques instances du modèle cible ne peuvent être décrites par le modèle source ; tandis que toutes les instances du modèle source peuvent être décrites par le modèle cible.
Enfin, les transformations avec perte mettent en relation un modèle source et un modèle cible et toutes les instances du modèle cible peuvent être décrites par le modèle source ; cependant quelques instances du modèle source ne peuvent être décrites par le modèle cible [2].

## 3 Organisation de modélisation et transformations de modèles

Depuis que l'ingénierie des logiciels s'est développée, des modèles sont écrits lors de l'analyse et de la conception d'un système, en respectant certains formalismes. Cet esprit de modélisation s'est poursuivi avec l'élaboration de méta-modèles pouvant être utilisés afin de définir les relations qui existent entre les concepts de ces formalismes. Ces relations représentent explicitement les transformations entre les formalismes [11]. Si ces méta-modèles sont formellement définis, il peut être possible de déterminer des règles de transformation basées sur la sémantique des méta-modèles, afin de transformer un modèle en un autre [3]. De tels méta-modèles nécessitent donc à leur tour des formalismes suffisamment riches afin de pouvoir exprimer tous les concepts que ces méta-modèles peuvent être amenés à représenter [11].

Par exemple, l'article [3] propose de formaliser un modèle orienté objet, un modèle relationnel et leur méta-modèle respectif grâce à sNets.

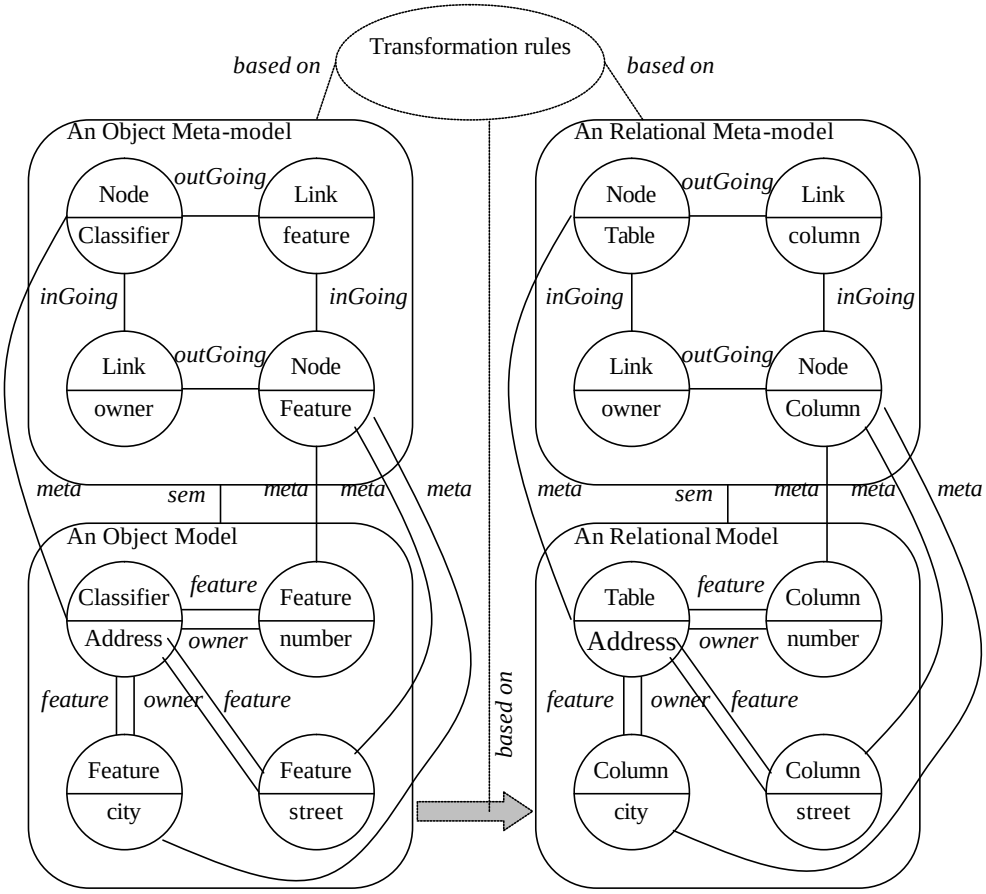



**Fig. 1.** Un modèle objet, un modèle relationnel et leur méta-modèle en sNets.

Dans cet exemple, une règle de transformation pourrait être : un nœud 'Classifier' devient un nœud 'Table' et un lien 'feature' entre un nœud 'Classifier et un nœud 'Feature' devient un lien 'column' entre un nœud 'Table' et un nœud 'Column'. Une telle règle est clairement basée sur les méta-modèles.
Lorsque de telles règles de transformations sont formulées formellement, la transformation d'un modèle source à un modèle cible consiste simplement à appliquer ces règles sur le modèle source jusqu'à ce que plus aucune règle ne soit applicable.
Enfin, il faut encore signaler que les formalismes des méta-modèles peuvent être aussi modélisés par un méta-méta-modèle. Ce méta-méta-modèle spécifie les différents éléments qui peuvent être utilisés pour établir un formalisme de méta-modèle [11].

## 4 Organisation de modélisation de l'OMG

L'OMG a proposé un langage de modélisation appelé UML. Le consensus établi vis-à-vis de la standardisation de UML a été déterminant pour l'évolution de la modélisation dans le génie logiciel. Maintenant, un rôle clef est joué par les méta-modèles. Cependant, d'autres langages similaires se sont développés. Pour faire face au danger d'avoir une variété de méta-modèles incompatibles différents, l'OMG a mis sur pied une structure générale d'intégration pour tous les méta-modèles, le MOF (Meta-Object Facility) fournissant un langage pour définir les méta-modèles et aboutissant à un méta-méta-modèle [17]. Cette organisation conduit à une architecture à quatre niveaux : méta-méta-modèle, méta-modèle, modèle et logiciel exécutable. Dans cette architecture, chaque niveau entretient une relation d'instanciation avec le niveau supérieur [8].

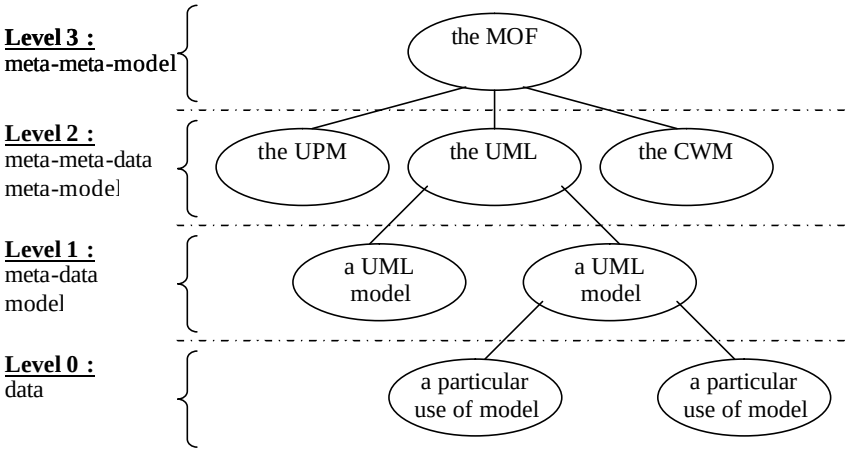



**Fig. 2.** L'architecture de modélisation standard à quatre couches de l'OMG [17].

## 5 Organisation de modélisation et niveau d'abstraction

Les articles [8] et [9] proposent de séparer chaque couche de l'organisation de modélisation en deux niveaux. Un niveau concret fournit le support qui doit être bien accepté (standard international), bien défini (spécification complète) et supporté par le maximum d'outils de modélisation, afin de présenter les modèles, méta-modèles et méta-méta-modèles. Par exemple, on peut concevoir une DTD XML permettant de satisfaire au rôle de ce niveau. Ensuite, un niveau abstrait est distingué. Ce dernier permet d'exprimer et définir les concepts utilisés dans les modèles et méta-modèles indépendamment de tout support de présentation des informations. Par exemple, la spécification MOF permet de satisfaire au rôle de ce niveau.

Ces articles indiquent également, la volonté de distinguer ces deux niveaux d'abstraction dans la problématique de transformations de modèles. Une vision pratique des transformations est différenciée d'une vision conceptuelle de ces mêmes transformations. Si l'exemple de XML est retenu, la vision pratique des

transformations tend à se diriger vers XSLT qui est au cœur des transformations de documents XML. En ce qui concerne la vision conceptuelle, elle peut être envisagée, par exemple, à l'aide MTRANS.

Enfin, ces articles ont montré que la problématique de transformations est plus simple si tous les modèles et méta-modèles sont basés sur le même méta-méta-modèle et si les transformations sont basées sur les méta-modèles.

## 6 Organisation de modélisation de l'OMG et niveau d'abstraction

XMI (XML Metadata Interchange Format) a été proposé comme format par l'OMG pour permettre l'échange des données et méta-données entre les outils de modélisation UML dans des environnements distribués hétérogènes. XMI est le produit de deux standards MOF et XML. La spécification MOF apporte la compatibilité entre les modèles et XML procure une amélioration dans l'échange des données. En fait, XMI permet de créer des DTD représentant des méta-modèles et les documents XMI qui sont conciliables avec une DTD, formulent des modèles instanciant le méta-modèle décrit par la DTD [8]. Le XMI peut donc être considéré comme le représentant du niveau concret dans l'organisation de l'OMG.

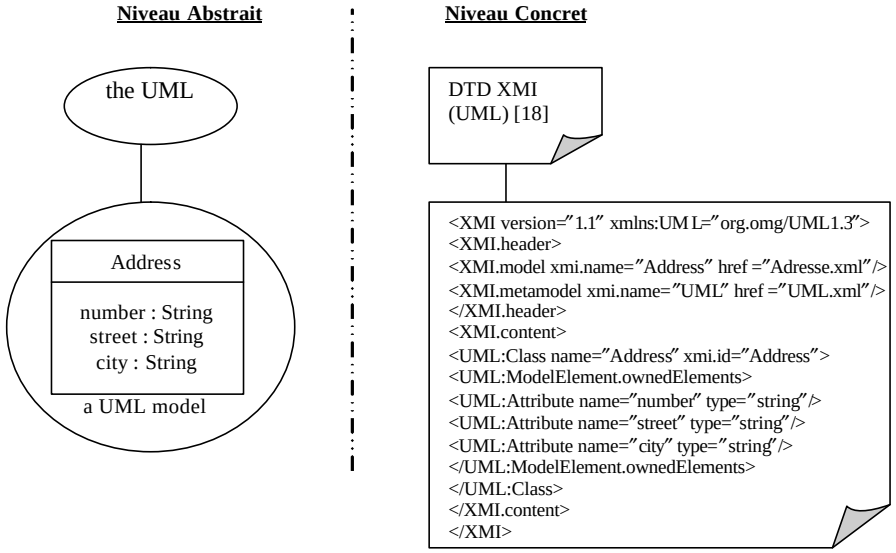



**Fig. 3.** Le niveau abstrait et concret d'un modèle UML et du méta-modèle UML.

Mais on peut également se servir de XMI pour réaliser les transformations aussi bien parmi les modèles UML qu'entre les modèles UML et les autres notations. Puisque XMI est fondé sur la spécification XML, on peut exploiter XSLT pour l'implémentation de transformations entre documents XMI. En effet, XSLT et XPath ont été créés particulièrement pour la transformation de documents XML. XSLT est un langage simple mais puissant de déclaration de règles de transformations. XPath offre un large éventail de fonctions permettant de naviguer dans un document XML,

ainsi que des opérations sur les string et booléen [15]. L'utilisation du langage XSLT est idéale pour le niveau concret des transformations, surtout depuis que XMI est devenu un standard d'échange de modèles entre outils de modélisation. Cependant, la formulation et la rédaction manuelle de règles de transformations sont difficiles et provoquent de nombreuses erreurs. Une vision abstraite de ces règles de transformations basées sur les méta-modèles respectant un autre langage permet l'expression de règles de transformations, de manière plus compacte et plus simple. En outre, à partir de ces règles abstraites basées sur les méta-modèles, on peut imaginer générer automatiquement les règles à appliquer aux modèles [8], [9].

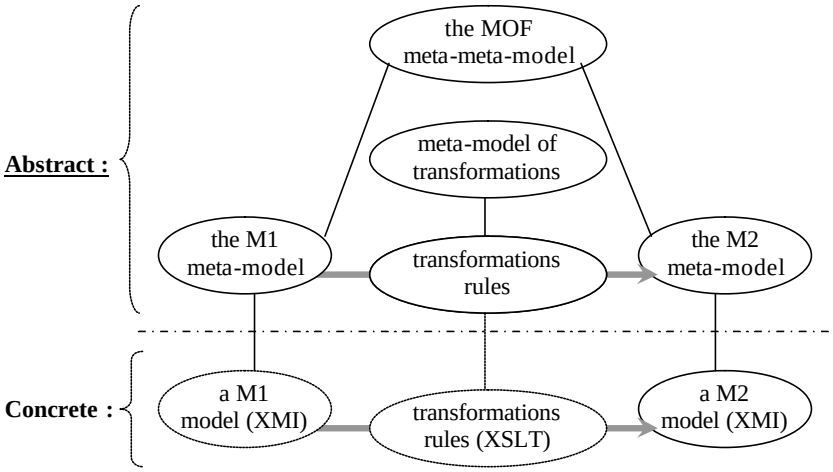



**Fig. 4.** Une structure de transformations de modèles à deux niveaux.

## 7 Technologie XSLT et XPath

Le langage XSLT permet de définir des règles de transformations destinées à transformer un document XML bien formé (valide pour une DTD particulière) en un autre. Cela signifie que chaque document XSLT, appelé feuille de style, regroupe un ensemble de règles de transformations permettant de passer d'une DTD à une autre. Pour cette raison, elles peuvent d'ailleurs être utilisées en chaîne : une transformation A de la DTD1 à la DTD2 et une transformation B de la DTD2 à la DTD3.

Le langage XSLT est lui-même défini avec le formalisme XML, cela signifie qu'une feuille de style XSLT est un document XML bien formé. Cette réalité permet d'imaginer la transformation via XSLT de feuilles de style XSLT, ce qui correspond à une transformation de transformation. On peut également imaginer obtenir à partir d'un document XML une feuille de style XSLT, ce qui correspond à une génération de transformation. Par exemple, dans le cadre de la problématique de transformations de modèles, cette dernière possibilité permet, si les règles abstraites peuvent être exprimées sous forme de document XML, de transformer ce document XML via XSLT pour obtenir une feuille de style XSLT correspondant aux règles concrètes à appliquer.

La différence principale entre les transformations réalisées avec XSLT par rapport à d'autres transformations, telles que celles intégrées immédiatement dans un code exécutable, est qu'il suffit de décrire le résultat à obtenir dans une feuille de style, puis fournir la description à un processeur XSLT, plutôt que de devoir spécifier comment obtenir le résultat. C'est donc le processeur XSLT qui se charge d'effectuer les transformations proprement dites. Ce processeur XSLT lit la feuille de style puis le document source (XML) et enfin applique les règles de transformations contenues dans la feuille de style afin de produire le document cible.

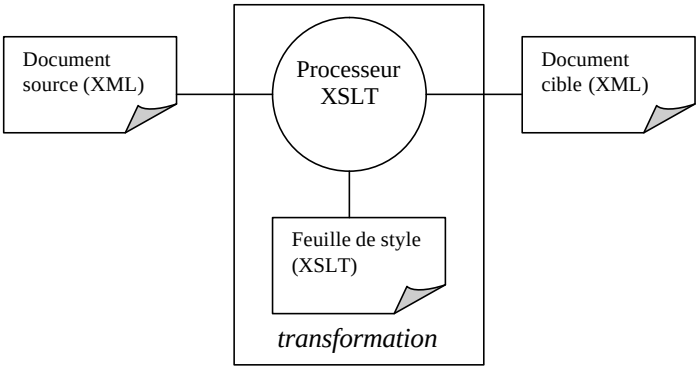



**Fig. 5.** Le principe de transformation d'un document XML bien formaté.

Pour pouvoir appliquer les règles de transformations, un processeur XSLT doit composer avec le principe selon lequelle les informations sont organisées dans un document XML. L'organisation de base des informations peut être décrite de manière abstraite comme un ensemble de nœuds assemblés hiérarchiquement sous forme d'arbre, chaque nœud contenant une partie des informations.

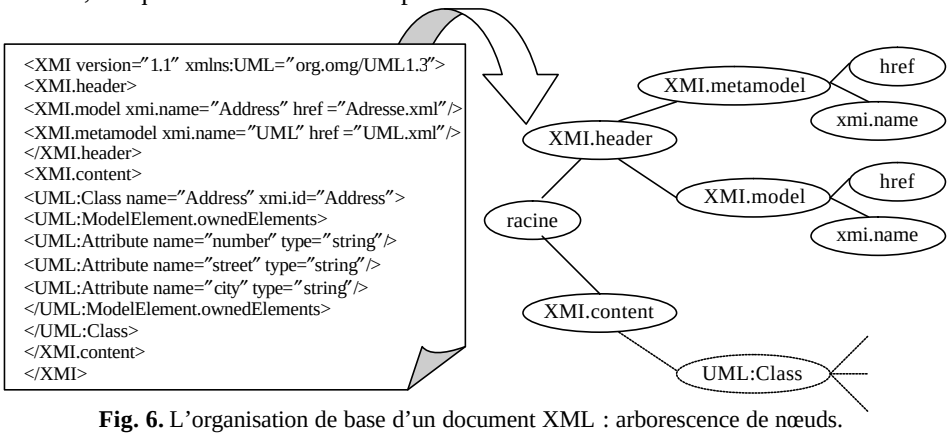



**Fig. 6.** L'organisation de base d'un document XML : arborescence de nœuds.

Le processeur XSLT crée donc une structure logique arborescente à partir du document XML et lui fait subir des transformations selon les règles contenues dans la feuille de style, pour produire un arbre résultat correspondant au document cible. Pour un processeur XSLT, un document XML peut être constitué de sept types de nœuds :

racine, élément, texte, attribut, espace de noms, instruction de traitement et commentaire.
Chaque règle de transformations définit donc des traitements à effectuer sur un nœud de l'arbre source. Une règle de transformation est composée de balises dont la première est `<xsl:template>` et la dernière est `</xsl:template>`. La première balise est complétée par un attribut `match` dont la valeur permet de définir le ou les nœuds du document XML sur lesquels s'applique la transformation :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet>
        <xsl:template match="nœud1 ">
                traitements sous forme de balises
        </xsl:template>                                  } Règle 1.

        <xsl:template match="nœud2 ">
                traitements sous forme de balises
        </xsl:template>                                  } Règle 2.
        …
</xsl:stylesheet>
```

**Fig. 7.** Un exemple de feuille de style.

La composition d'une règle de transformations est réalisée à l'aide de balises qui permettent de sélectionner et d'effectuer des opérations sur les éléments du document XML. Leur syntaxe est la suivante : `<xsl:`*`nom [attributs]`*`/>`. Voici une liste des balises de transformations les plus couramment utilisées :

| Nom | Attributs | Signification |
|---|---|---|
| `apply-templates` | `select` `mode` | Permet d'appliquer les règles correspondant à l'attribut `mode` sur les nœuds sélectionnés par l'attribut `select`. |
| `call-template` | `name` | Permet d'invoquer une règle dont le nom est `name`. |
| `with-param` | `name` | Permet de définir un paramètre dont le nom est `name` lors d'un appel de règle `xsl:call-template` et `xsl:apply-template`. La règle appelée doit déclarer ce paramètre à l'aide de `xsl:param`. |
| `param` | `name` | Permet de définir un paramètre dont le nom est `name`. |
| `variable` | `name` | Permet de définir une variable dont le nom est `name`. |
| `choose` | | Permet de choisir entre plusieurs alternatives de traitement. Cette balise est suivie de `xsl:when` ou `xsl:otherwise` qui permettent de séparer les alternatives en fonction de certaines conditions. |
| `if` | `test` | Permet de choisir si un traitement doit être effectué en fonction d'une condition spécifiée par l'attribut `test`. |
| `for-each` | `select` | Permet l'itération d'un traitement pour tous les nœuds sélectionnés par l'attribut `select`. |
| `value-of` | `select` | Permet d'évaluer, comme un string, la valeur du nœud sélectionné par l'attribut `select`. |
| `copy` | | Permet de copier le nœud courant dans le document cible. |
| `copy-of` | `select` | Permet de copier la valeur des nœuds sélectionnés par l'attribut `select` dans le document cible. |

| | | |
|---|---|---|
| `element` | `name` | Permet de créer un nouvel élément dont le nom est `name` dans le document cible. |
| `attribute` | `name` | Permet d'ajouter un attribut dont le nom est `name` à un élément. |
| `text` | | Permet d'écrire un texte dans le document cible. |
| `document` | `href` | Permet de diriger le résultat de la transformation vers un autre document cible dont l'URL est spécifiée par l'attribut `href`. Un document source peut donc être à l'origine de plusieurs documents cibles. |
| `include` | `href` | Permet d'inclure les règles de transformations d'une autre feuille de style dont l'URL est spécifiée par l'attribut `href`. |

**Fig. 8.** Liste des balises principales de transformations XSLT.

La composition d'une règle de transformations nécessite donc un moyen d'expression de sélection de nœuds, pouvant être utilisé afin de fournir une valeur aux attributs `select` des différentes balises de transformations. De plus, certaines balises requièrent la possibilité de formuler des conditions. Ces exigences sont satisfaites par XPath qui définit une syntaxe fournissant les outils pour localiser les nœuds et les informations contenues dans ces nœuds, à partir de la structure abstraite : un ensemble de types de données utilisées pour représenter les informations et les nœuds, et un ensemble de fonctions pour manipuler ces types de valeurs. L'ensemble des types de valeurs inclut les strings, les entiers, les valeurs booléennes et des expressions de nœuds telles que : le nœud fils, le nœud père, l'attribut du nœud courant, etc. La syntaxe des expressions de nœud ressemble à celle utilisée dans les systèmes pour adresser un répertoire.

<table>
<tr><th>Expression-Nœud</th><th colspan="2">Signification</th></tr>
<tr><td>.</td><td colspan="2">nœud courant</td></tr>
<tr><td>/</td><td colspan="2">nœud racine</td></tr>
<tr><td>/ chemin-relatif</td><td colspan="2">nœuds de chemin relatif par rapport au nœud racine</td></tr>
<tr><td>expression-nœud1 | expression-nœud2</td><td colspan="2">union de deux ensembles de nœuds</td></tr>
<tr><td>expression-nœud1 [predicat]</td><td colspan="2">sous-ensemble de l'ensemble de nœuds expression-nœud1 qui respecte le predicat (prédicat exprimé grâce à XPath)</td></tr>
<tr><td rowspan="6">expression-nœud1 / chemin-relatif</td><td colspan="2">nœuds de chemin relatif par rapport aux nœuds de l'ensemble expression-nœud1</td></tr>
<tr><td>chemin-relatif</td><td>Signification</td></tr>
<tr><td>*</td><td>tous les nœuds fils</td></tr>
<tr><td>name</td><td>tous les nœuds fils dont le nom est name</td></tr>
<tr><td>@*</td><td>tous les attributs</td></tr>
<tr><td>@name</td><td>tous les attributs dont le nom est name</td></tr>
<tr><td>..</td><td>tous les nœuds parents</td></tr>
</table>

**Fig. 9.** La syntaxe (incomplète) des expressions de nœuds de XPath.

# 8 Langage MTRANS

MTRANS est un langage dédié à la transformation d'un méta-modèle vers un autre. Les méta-modèles source et destination doivent être conformes à la spécification MOF, car l'outil de génération associé au langage MTRANS doit vérifier certaines propriétés par rapport aux règles de transformations. La spécification MOF a été choisie comme méta-méta-modèle car elle apporte une uniformisation standardisée dans la représentation des modèles et des méta-modèles [19]. Le langage MTRANS permet d'exprimer de manière plus commode, les règles définissant la méthode pour transformer les entités d'un méta-modèle vers les entités d'un autre méta-modèle. Dans ce contexte, une entité représente simplement un concept du méta-modèle source ou cible. Par exemple, les concepts ″Classifier″, ″Feature″ du méta-modèle UML ou les concepts ″Class″, ″Field″ du méta-modèle Java peuvent être considérés comme des entités dans le langage de transformation MTRANS. Une règle de transformations d'entité est divisée en deux parties : la première est utilisée pour spécifier l'entité source et la seconde pour l'entité destination. En appliquant des restrictions, la règle peut être limitée aux entités respectant une certaine condition. Une restriction est également composée de deux parties : une partie droite qui détermine un rôle ou un attribut de l'entité et une partie gauche qui correspond à la valeur acceptée pour ce rôle ou cet attribut. MTRANS permet également de créer de nouvelle entité.
De plus, MTRANS permet d'énoncer les règles spécifiant le procédé de transformation des attributs et des rôles d'une entité. Ces règles prennent la forme d'une assignation dont la partie gauche correspond au nom de l'attribut ou du rôle transformé et la partie droite définit la valeur de cet attribut ou ce rôle dans le méta-modèle de destination.
La valeur assignée à un attribut peut être de quatre types : un attribut du méta-modèle source, un attribut quelconque exact, une valeur conditionnelle ou une concaténation d'attributs contenus dans le méta-modèle source.
La valeur assignée à un rôle peut être de trois types : un rôle du méta-modèle source, une valeur conditionnelle ou un rôle du méta-modèle source réduit à une entité.

*rules_transformation* ::= *Entity* **[***restriction***]**? **to** *Entity*
**{ attributes :** [*Attribute* = *attributes_transformation* **,**]*
**roles :** [*Role* = *roles_transformation* **,**]* **}**
| *Entity*
**{ attributes :** [*Attribute* = *attributes_transformation* **,**]*
**roles :** [*Role* = *roles_transformation* **,**]* **}**

*restriction* ::= *Attribute operator value* | *Role operator value*
*attributes_transformation* ::= *Attribute* | *Entity***.***Attribute* | ²*Literal*²
**| (** *condition* **?** *resultIfTrue* **:** *resultIfFalse* **)**
| *Attribute* **#** *Attribute*

*resultIfTrue* ::= *attributes_transformation*
*resultIfFalse* ::= *attributes_transformation*

*roles_transformation* ::= *Role* | **(** *condition* **?** *resultIfTrueBis* **:** *resultIfFalseBis* **)**
| *Role***[***Entity***]**

*resultIfTrueBis* ::= *roles_transformation*
*resultIfFalseBis* ::=*roles_transformation*

**Fig. 10.** La syntaxe de MTRANS.

## 9 Correspondance entre MTRANS et XSLT

On a déclaré qu'à partir de règles abstraites basées sur les méta-modèles, on peut imaginer générer automatiquement les règles concrètes à appliquer aux modèles. Dans le cas précis où on emploie le formalisme MTRANS pour exprimer les règles abstraites et XSLT pour énoncer les règles concrètes, on peut déjà établir des correspondances entre les règles de chaque niveau :

- *Entity1* **[***attORrole operator value***] to** *Entity2*

  devient,

```
<xsl:template match="Entity1">
   <xsl:if test="attORrole operator value">
      <xsl:element name="Entity2">
         <!--attributes mapping-->
         <!--roles mapping-->
         <xsl:apply-templates select="./child::node()" mode="Entity1"/>
           </xsl:element>
      </xsl:if>
</xsl:template>
```

A la fin de cette règle de transformations d'entité, on doit encore ajouter les règles de correspondances des attributs et des rôles de cette même entité. Cette opération est réalisée grâce à l'appel de règles (`xsl:apply-templates`) sur les nœuds enfants. Cependant, toutes les entités doivent avoir des règles spécifiques pour leurs attributs et leurs rôles, c'est pourquoi, l'appel de règles est effectué selon le mode de l'entité. De plus, les nœuds enfants peuvent être de deux types : des nœuds éléments ou des nœuds attributs. Dès lors, les correspondances entre les règles de transformations des attributs sont les suivantes :

- *Attribute* = ²*value*²

  devient,

```
<xsl:attribute name="Attribute">
   <xsl:value-of select="'value'"/>
</xsl:attribute>
```

- *Attribute1* = *Attribute2*

devient,

```
<xsl:variable name="typenode">
   <xsl:value-of select="@Attribute2"/>
</xsl:variable>
<xsl:if test="$typenode !=''">
   <xsl:attribute name="Attribute1">
      <xsl:value-of select="$typenode"/>
   </xsl:attribute>
</xsl:if>
<xsl:if test="$typenode ==''">
   <xsl:element name="Attribute1">
      <xsl:value-of select="./owner.Attribute2"/>
      </xsl:element>
</xsl:if>
```

- *Attribute1* = **(** *Attribute2 operator* **'***value***' ?** *Attribute3* **:** *Attribute4* **)**

devient,

```
<xsl:variable name="typenode">
     <xsl:value-of select="@Attribute2"/>
</xsl:variable>
<xsl:if test="$typenode ==''">
     <xsl:variable name="typenode">
     <xsl:value-of select="./owner.Attribute2"/>
   </xsl:variable>
</xsl:if>
<xsl:if test="$typenode operator 'value'">
   < !--mapping based on kind of Attribute3-->
   <xsl:variable name="typenode">
      <xsl:value-of select="@Attribute3"/>
   </xsl:variable>
      <xsl:if test="$typenode !=''">
           <xsl:attribute name="Attribute1">
                <xsl:value-of select="$typenode"/>
           </xsl:attribute>
      </xsl:if>
      <xsl:if test="$typenode ==''">
           <xsl:element name="Attribute1">
                <xsl:value-of select="./owner.Attribute2"/>
      </xsl:element>
      </xsl:if>
</xsl:if>
<xsl:if test="not($typenode operator 'value')">
   < !--mapping based on kind of Attribute4-->
   <xsl:variable name="typenode">
      <xsl:value-of select="@Attribute4"/>
   </xsl:variable>
```

```
      <xsl:if test="$typenode !=''">
      <xsl:attribute name="Attribute1">
         <xsl:value-of select="$typenode"/>
      </xsl:attribute>
      </xsl:if>
      <xsl:if test="$typenode ==''">
      <xsl:element name="Attribute1">
         <xsl:value-of select="./owner.Attribute2"/>
      </xsl:element>
      </xsl:if>
</xsl:if>
```

Enfin, on peut conclure par les correspondances entre règles de transformations des rôles :

- *Role1* = *Role2*

  devient,

```
<xsl:variable name="typenode">
   <xsl:value-of select="@Role2"/>
</xsl:variable>
<xsl:if test="$typenode !=''">
   <xsl:attribute name="Role1">
      <xsl:value-of select="$typenode"/>
      </xsl:attribute>
</xsl:if>
<xsl:if test="$typenode ==''">
   <xsl:element name="Role1">
      <xsl:apply-templates select="./owner.Role2" mode="single"/>
      </xsl:element>
</xsl:if>

<xsl:template match="owner.Role2" mode="single">
   <xsl:element name="Role1">
      <xsl:apply-templates select="./child::node()" mode="single"/> (1)
      </xsl:element>
</xsl:template>
```

- *Role1* = *Role2***[***Entity1***]**

  devient,

  le même code que la transformation ci-dessus dans lequel (1) est remplacé par :
  `<xsl:apply-templates select="./child::Entity1"/>`

- *Role1* = **(** *Role2 operator* **‘***value***’ ?** *Role3* **:** *Role4* **)**

devient,

```
<xsl:attribute name="Role1">
<xsl:variable name="typenode">
   <xsl:value-of select="@Role2"/>
</xsl:variable>
<xsl:if test="$typenode =='">
   <xsl:variable name="typenode">
      <xsl:value-of select="./owner.Role2"/>
   </xsl:variable>
</xsl:if>
<xsl:if test="$typenode operator 'value'">
   < !--mapping based on kind of Role3-->
   …
</xsl:if>
<xsl:if test="not($typenode operator 'value')">
   < !--mapping based on kind of Role4-->
   …
</xsl:if>
</xsl:attribute>
```

## 10 Conclusion

L'utilisation de méta-modèles contenant les concepts des modèles facilite la composition des règles de transformations et minimise leur nombre et leur complexité. L'emploi de méta-modèles pour engendrer les transformations permet également d'obtenir un modèle transformé compatible avec un méta-modèle de destination. Enfin, l'approche de la transformation d'un méta-modèle vers un autre permet de construire des règles de transformations très génériques puisque ces règles de transformation peuvent s'appliquer à tous les modèles pourvu qu'il respecte le méta-modèle source.
Le langage XSLT est approprié pour transformer un document XML en un autre. Mais l'écriture d'une feuille de style est longue et pénible car XSLT n'est pas réservé uniquement à la transformation de modèles. Par contre, MTRANS est adapté à la transformation de méta-modèle et permet d'exprimer naturellement les règles de transformations. L'utilisation d'un tel langage est donc nécessaire surtout que comme le montrent les correspondances entre MTRANS et XSLT, il semble aisé de générer automatiquement, à partir des règles du niveau abstrait en MTRANS, des règles XSLT plus concrètes adaptées aux modèles à transformer.
De plus, contrairement aux solutions de transformation basées sur un code exécutable rigide dans lesquelles le code devrait être remodelé pour pouvoir faire évoluer une transformation, l'architecture de transformation utilisant des feuilles de style XSLT et un processeur pour les interpréter est pleinement évolutive. En effet, pour pouvoir faire évoluer une transformation entre deux modèles, il suffira simplement de changer la feuille de style utilisée, sans pour autant changer la manière dont elle est utilisée. Cependant, cette architecture de transformation est loin d'être complète et nécessite encore de nombreuses recherches afin de l'améliorer et d'en dégager tout le potentiel.

## Bibliographie


[1] Henry Tirri and Greger Lindén ; *ALCHEMIST – an object-oriented tool to build transformations between Heterogeneous Data Representations* ; Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences (HICSS'94), Maui, Hawaii, volume II, IEEE Computer Society Press ; January 1994 ; 226-235.

[2] Michael Blaha and William Premerlani ; *A Catalog of Object Model Transformations* ; Presented at 3rd Working Conference on Reverse Engeneering, Monterey, California ; November 1996.

[3] R. Lemesle ; *Transformation Rules Based on Meta-modeling* ; EDOC'98, San Diego ; November 1998.

[4] Michael H. Kay ; *XPath Expression Syntax* ; April 1999 ; http://users.iclway.co.uk/mhkay/saxon/saxon6.3/expressions.html.

[5] Michael Petit ; *Formal Requierements Engeneering of Manufacturing Systems : A Multi-Formalism and Composant-Based Approach* ; A thesis submitted to The Computer Science Departement, University of Namur, Belgium ; October 1999.

[6] W3C ; *Extensible Stylesheet Language Transformation (XSLT) version 1.0* ; November 1999 ; http://www.w3.org/TR/xslt.

[7]W3C ; *XML Path Language (XPath) version 1.0* ; November 1999 ; http://www.w3.org/TR/xpath.

[8] Mikaël Peltier, François Ziserman & Jean Bézivin ; *On levels of model transformation* ; XML Europe Conference, Paris, France ; June 2000.

[9] Mikaël Peltier, François Ziserman & Jean Bézivin ; *Concrete and Abstract Spaces in Model Engineering* ; Information Systems Engineering : Conceps and Industrial Applications World (ISICA), Multiconference on Systemics, Cybernetics and Informatics, Orlando ; July 2000.

[10] Ken Holman ; *What is XSLT?* ; August 2000 ; http://www.xml.com/pub/a/2000/08/holman/.

[11] Pieter Mosterman and Hans L. Vangheluwe ; *Computer automated multi paradigm modeling in control system design* ; International Symposium on Computer-Aided Control System Design, Anchorage, Alaska, IEEE Computer Society Press ; September 2000 ; 65-70.

[12] David H Akehurst ; *Model Translation : A UML-based specification technique and active implementation approach* ; A thesis submitted to The University of Kent, Canterbury ; December 2000.

[13] N. Revault, X. Blanc and J-F. Perrot ; *Traduction de méta-modèles* ; Langage et Modèle à Objets 2001 (LMO'01), France ; Janvier 2001.

[14] Michael H. Kay ; *Standard XSLT Elements* ; February 2001 ; http://users.iclway.co.uk/mhkay/saxon/saxon6.4/xsl-elements.html

[15] Birgit Demuth, Heinrich Hussmann and Sven Obermaier ; *Experiments With XMI Based Transformations of Software Models* ; WTUML: Workshop on Transformations in UML, ETAPS 2001 Satellite Event, Genova, Italy ; April 2001

[16] Mikaël Peltier, Jean Bézivin & Gabriel Guillaume ; *MTRANS: A general framework, based on XSLT, for model transformations* ; Workshop on Transformations in UML, Genova, Italy ; April 2001.

[17] Jean Bézivin ; *From Object Composition to Model Transformation with MDA* ; The proceedings of TOOLS'USA, Santa Barbara, volume I ; August 2001.

[18] OMG ; *eXtensible Metadata Interchange (XMI) specification version 1.2* ; Document formal/02-01-01 ; January 2002 ; http://www.omg.org/cgi-bin/doc?formal/2002-01-01.

[19] Mikaël Peltier ; *Transformation entre un profil UML et un méta-mod èle MOF* ; Langages et Modèles à Objets, volume I ; 2002.

## Annexe : Exemple de transformation entre un modèle orienté objet et un modèle relationnel

Reprenons en détail l'exemple proposé à la figure 1. Voici les méta-modèles détaillés sous une autre forme :

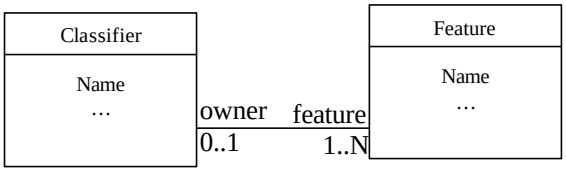



**Fig. 11.** Méta-modèle orienté objet.

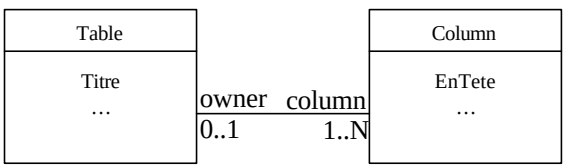



**Fig. 12.** Méta-modèle relationnel.

A partir de ces méta-modèles, on peut composer des règles de transformation en respectant la syntaxe MTRANS :

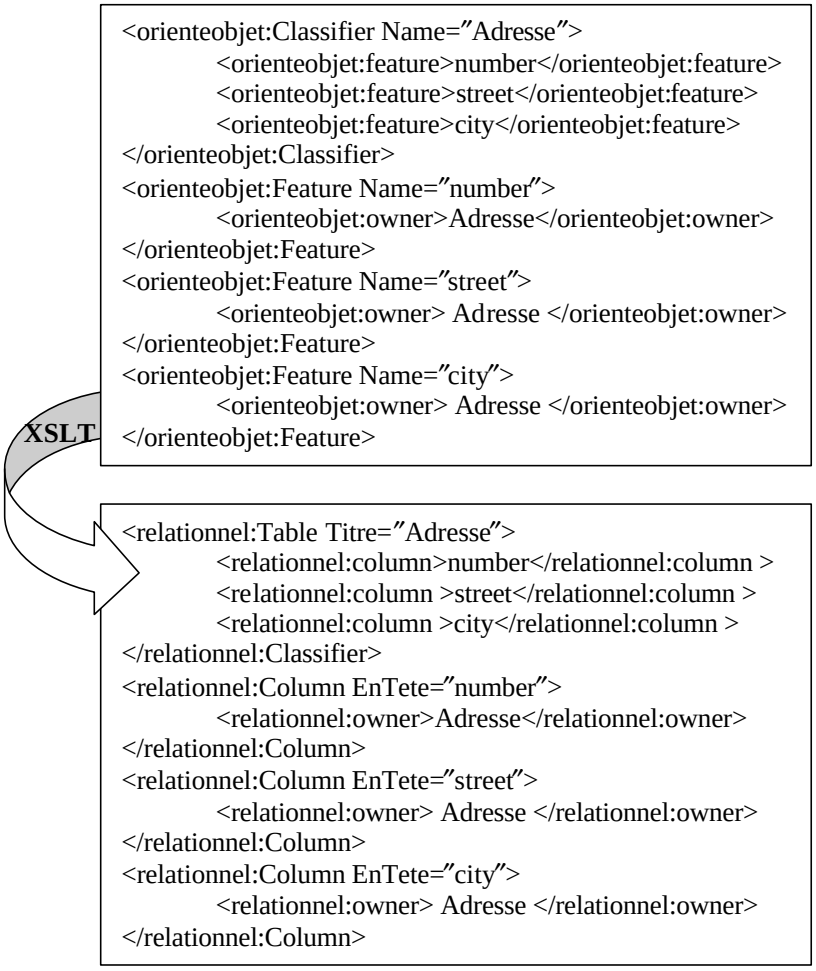
**setOfRules**
**{** Classifier **to** Table
**{** **attributes :** Titre = Name, …
**roles :** owner = owner
**}**
Feature **to** Column
**{** **attributes :** EnTete = Name, …
**roles :** column = feature
**}**
**}**

Ces règles de transformation du niveau abstrait peuvent trouver une correspondance au niveau concret sous forme de règles de transformation respectant la syntaxe XSLT :

```
<xsl:template match="Classifier">
       <xsl:element name="Table">
              <xsl:variable name="typenode">
                 <xsl:value-of select="@Name"/>
              </xsl:variable>
              <xsl:if test="$typenode !=''">
                 <xsl:attribute name="Titre">
                    <xsl:value-of select="$typenode"/>
                 </xsl:attribute>
              </xsl:if>
              <xsl:if test="$typenode ==''">
                 <xsl:element name="Titre">
                    <xsl:value-of select="./owner.Name"/>
                 </xsl:element>
              </xsl:if>
              <xsl:apply-templates select="./child::node()"
              mode="single"/>

              <xsl:variable name="typenode">
                 <xsl:value-of select="@owner"/>
              </xsl:variable>
              <xsl:if test="$typenode !=''">
                 <xsl:attribute name="owner">
                    <xsl:value-of select="$typenode"/>
                    </xsl:attribute>
              </xsl:if>
              <xsl:if test="$typenode ==''">
                 <xsl:element name="owner">
                      <xsl:apply-templates select="./owner.owner"
                      mode="single"/>
                  </xsl:element>
```

```
            </xsl:if>
        </xsl:element>
</xsl:template>

    <xsl:template match="owner.owner" mode="single">
       <xsl:element name="owner">
          <xsl:apply-templates select="./child::node()" mode="single"/>
          </xsl:element>
    </xsl:template>

<xsl:template match="Feature">
        <xsl:element name="Column">
            <xsl:variable name="typenode">
               <xsl:value-of select="@Name"/>
            </xsl:variable>
            <xsl:if test="$typenode !=''">
               <xsl:attribute name="EnTete">
                  <xsl:value-of select="$typenode"/>
               </xsl:attribute>
            </xsl:if>
            <xsl:if test="$typenode ==''">
               <xsl:element name="EnTete">
                  <xsl:value-of select="./owner.Name"/>
               </xsl:element>
            </xsl:if>
            <xsl:apply-templates select="./child::node()"
            mode="single"/>

            <xsl:variable name="typenode">
               <xsl:value-of select="@feature"/>
            </xsl:variable>
            <xsl:if test="$typenode !=''">
               <xsl:attribute name="column">
                  <xsl:value-of select="$typenode"/>
                  </xsl:attribute>
            </xsl:if>
            <xsl:if test="$typenode ==''">
               <xsl:element name="column">
                    <xsl:apply-templates select="./owner.feature"
                    mode="single"/>
                </xsl:element>
            </xsl:if>
        </xsl:element>
</xsl:template>

    <xsl:template match="owner.feature" mode="single">
       <xsl:element name="column">
          <xsl:apply-templates select="./child::node()" mode="single"/>
          </xsl:element>
    </xsl:template>
```

Au niveau concret, les méta-modèles peuvent être décrits dans des documents XML. On suppose que ces deux documents sont conformes à deux DTD correspondant chacune à un méta-modèle. On peut constater que les règles de transformation XSLT s'appliquent parfaitement au passage d'un document à l'autre, et donc d'une DTD à l'autre.




```
<orienteobjet:Classifier Name="Adresse">
        <orienteobjet:feature>number</orienteobjet:feature>
        <orienteobjet:feature>street</orienteobjet:feature>
        <orienteobjet:feature>city</orienteobjet:feature>
</orienteobjet:Classifier>
<orienteobjet:Feature Name="number">
        <orienteobjet:owner>Adresse</orienteobjet:owner>
</orienteobjet:Feature>
<orienteobjet:Feature Name="street">
        <orienteobjet:owner> Adresse </orienteobjet:owner>
</orienteobjet:Feature>
<orienteobjet:Feature Name="city">
        <orienteobjet:owner> Adresse </orienteobjet:owner>
</orienteobjet:Feature>
```

```
<relationnel:Table Titre="Adresse">
        <relationnel:column>number</relationnel:column >
        <relationnel:column >street</relationnel:column >
        <relationnel:column >city</relationnel:column >
</relationnel:Classifier>
<relationnel:Column EnTete="number">
        <relationnel:owner>Adresse</relationnel:owner>
</relationnel:Column>
<relationnel:Column EnTete="street">
        <relationnel:owner> Adresse </relationnel:owner>
</relationnel:Column>
<relationnel:Column EnTete="city">
        <relationnel:owner> Adresse </relationnel:owner>
</relationnel:Column>
```