CHAPITRE 8 ATLAS TRANSFORMATION LANGUAGE « ATL »

1. Introduction

Les langages de transformations viennent de différents espaces technologiques et ont été proposés par des communautés séparées. Ils sont représentatifs de différents types de transformation et ont chacun leur propre historique de développement. Leurs domaines d'applications originels sont généralement séparés. Il ya plusieurs langages de transformations par exemple : **TXL**, **XSLT, ATL, UMLX…etc.**

2. Historique d'ATL

Deux équipes de l'INRIA « **TRISKELL** à Rennes et **ATLAS** à Nantes » mènent des recherches sur l'ingénierie des modèles. Elles ont exploré deux pistes différentes et complémentaires autour de la transformation de modèles pour l'adapter aux utilisateurs et ont développé deux langages de transformation de modèles, en quelque sorte des implémentations de : **ATL** « **A**TLAS **T**ransformation **L**anguage » développé par **ATLAS** et MTL « **M**odel **T**ransformation **L**anguage » par **TRISKELL**.

ATL est issu de recherches menées en collaboration entre l'INRIA, l'université de Nantes et une société brestoise, TNI-Software. **ATL** et son environnement de programmation sont officiellement disponibles en logiciel libre sur la plate-forme Eclipse d'IBM depuis octobre 2004. Cette même année, **ATL** a reçu le prix innovation Eclipse d'IBM. Parallèlement, l'équipe met en œuvre les mêmes principes selon la démarche cette fois de Microsoft, dans le cadre de conventions de recherche et étend les fonctionnalités des opérateurs au-delà de la transformation en prenant en compte par exemple le tissage de modèles. Ces travaux se font principalement dans des projets de coopération industrielle comme le projet européen ModelWare piloté par la société Thales. Des partenariats forts avec d'autres industriels français comme le groupe Sodifrance ont permis de développer d'autres outils complétant **ATL** dans le cadre d'une plate-forme de recherche en ingénierie des modèles « **AMMA** ».

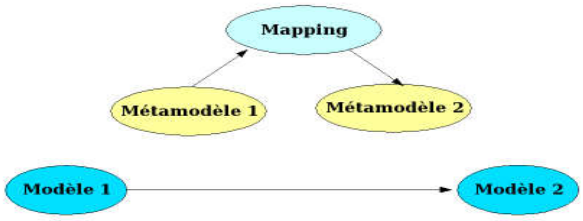
3. Définition d'ATL

ATL « **A**TLAS **T**ransformation **L**anguage » est un langage de transformation de modèles destiné à être appliqué plus particulièrement à l'ingénierie des données, autrement dit aux systèmes d'information et aux systèmes de gestion de base de données. De fait, leur taille ne cesse de s'accroître, les données qu'elles contiennent sont de plus en plus hétérogènes et complexes et les applications qui les utilisent de plus en plus variées.

Donc **ATL** est un langage de transformation qui implémente le principe **QVT**. **QVT** « **Q**uery **V**iew **T**ransformation » qui est une spécification de l'**OMG** pour les langages de transformation et de manipulation de modèles.

Il est développé sur la plateforme Eclipse et plus particulièrement sur sa branche **EMF** « **E**clipse **M**odeling **F**ramework ». Il est accompagné de nombreux outils pour faciliter son utilisation. La mise en forme des mots clés du langage est assuré dans l'éditeur de code ATL, un debugger est fourni, et une notation textuelle simple appelée km3 « **K**ernel **M**eta **M**odel) permet la spécification de méta-modèles.

La transformation de modèle **MDA** se fait par **mapping** entre un modèle initial et un modèle cible. Chaque modèle doit être d'écrit par un méta-modèle, qui recense les caractéristiques de ce modèle. Le **mapping** est alors définit comme une traduction entre le méta-modèle initial et le méta-modèle cible. La figure suivante présente le principe de la transformation.



4. Caractéristiques d'ATL

Le langage ATL présente les caractéristiques suivantes :

- **Modularité**. Chaque transformation en ATL est dans un module. Plusieurs modules peuvent s'inclure mutuellement et se composer pour former des modules de transformation plus grande.
- **Interopérabilité**. Nous n'avons pas d'information concernant le fait qu'une règle ATL puisse ou non appeler un autre langage.

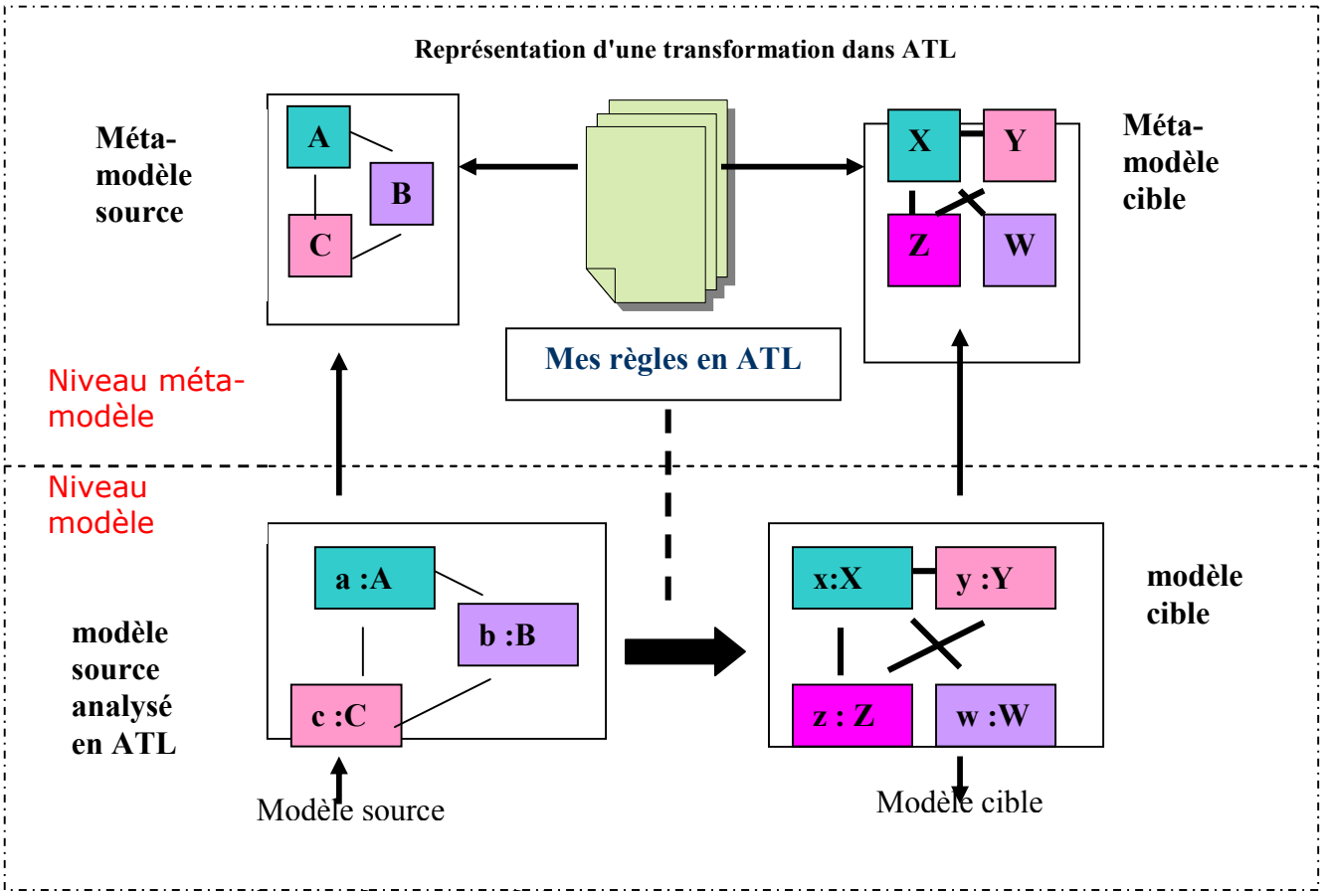
- **Extensibilité**. ATL peut être considéré comme un langage de programmation. La possibilité de définir des fonctions ou des parties impératives et de les stocker dans des modules permet de définir des bibliothèques réutilisables. La syntaxe d'ATL est simple et ne peut pas être étendue.
- **Traçabilité.** La traçabilité entre le modèle source, le modèle cible et la transformation est gérée par l'environnement d'exécution ATL car des liens sont créés dynamiquement entre ces différents éléments. Chaque élément du modèle source est associé à un élément du modèle cible. Cette relation est conservée pour gérer les transformations inverses.

5. Caractéristiques d'une transformation ATL

Les principales caractéristiques d'une transformation **ATL** sont les suivantes :

- Un environnement supposant que les méta-modèles soient représentés en **MOF**.
- Les modèles sources et cibles conformes aux méta-modèles sources et cibles.
- Des règles de transformation **ATL** construites à partir des éléments du méta-modèle.
- La syntaxe d'**ATL** est textuelle. Elle est basée sur le langage **OCL**.

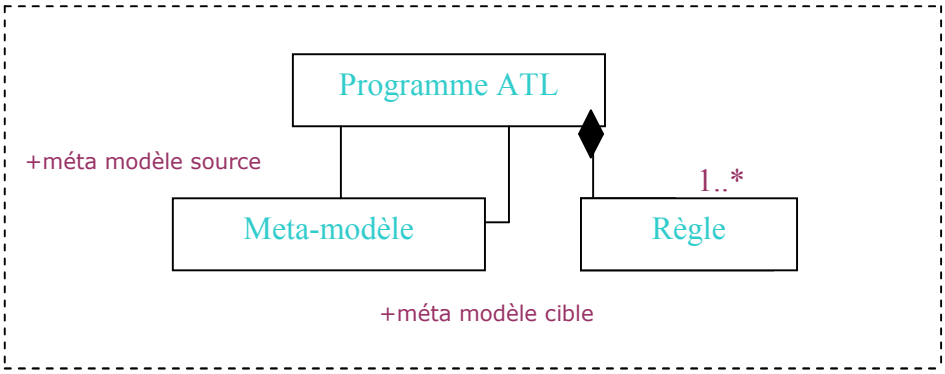
Un Programme **ATL** est formé d'une collection de règles et référence un méta-modèle source et un méta-modèle cible.



5.1 Syntaxe abstraite

ATL est un langage de transformation hybride, semi-déclaratif et semi-impératif. Comme différents autres langages de l'espace MDA, ATL est basé sur le langage OCL pour l'écriture des expressions. OCL est le langage de contraintes associé à UML. Il fait partie de la collection des standards MDA. Le rôle joué par OCL dans ATL est similaire au rôle joué par XPath dans XSLT.

Un programme ATL est formé d'une collection de règles et référence un méta-modèle source et un méta-modèle cible.



Structure d'un programme ATL

Forme générale d'une règle :

```
rule rule_name {
        from
                in_var : in_type [( condition )]?
        [using {        var1 : var_type1 = init_exp1;
                ...
                varn : var_typen = init_expn;}]?

           to
                   out_var1 : out_type1 (bindings1),
                   ...
                   out_varn : out_typen (bindingsn        )
           [do {   action_block }]?
   }
```

Les règles peuvent être de différents types, selon la manière dont elles sont appelées et la manière dont elles spécifient leur résultat. Il existe plusieurs sortes de règles en ATL :

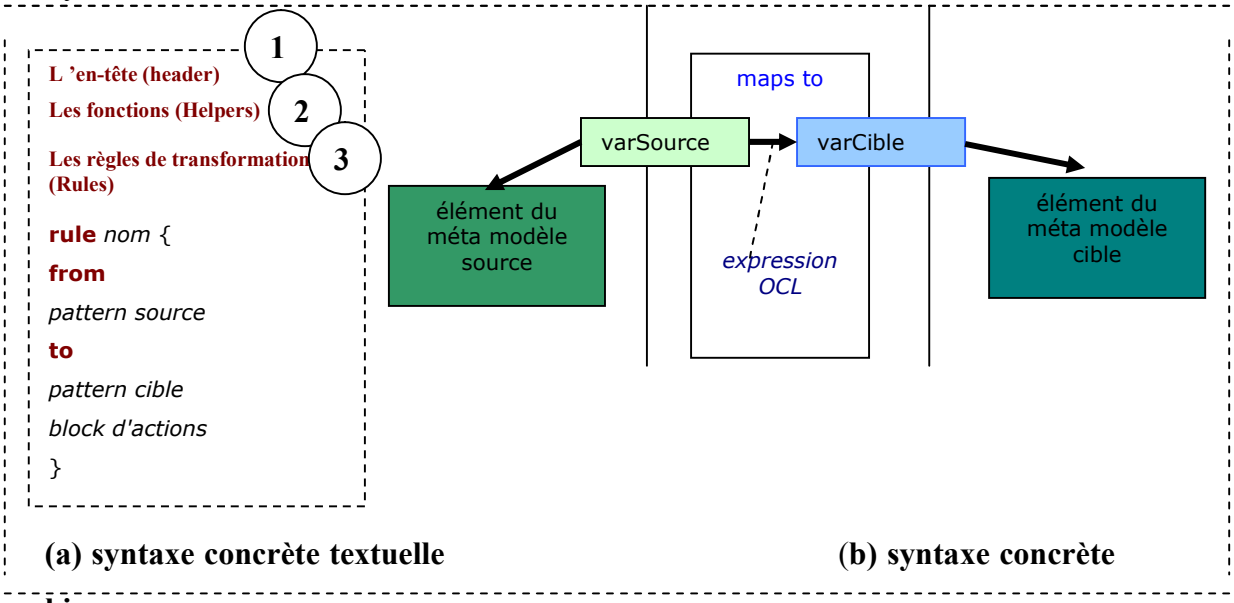
- **Fonction (CalledRule).** Elles ont des paramètres. Ce sont des opérations au sens OCL et peuvent donc être appelée comme des fonctions dans les expressions OCL.

- **Règle à pattern (MatchedRule).** Contrairement aux fonctions dont le profil est définit par une liste de paramètres, les règles à pattern ont généralement un pattern source. Elles sont appelées automatiquement dans le processus d'exécution lorsque des correspondances auront lieu.

De plus les règles peuvent avoir ou non une section impérative sous la forme d'un block d'actions. Les règles peuvent également avoir un pattern cible.

5.2 Syntaxe concrète

ALT a deux syntaxes concrètes : l'une est textuelle et l'autre est graphique. La syntaxe graphique est utilisée uniquement pour la visualisation car elle n'est pas complète et ne permet d'avoir qu'une vision globale de la transformation. La correspondance entre les deux syntaxes est illustrée dans le schéma suivant :

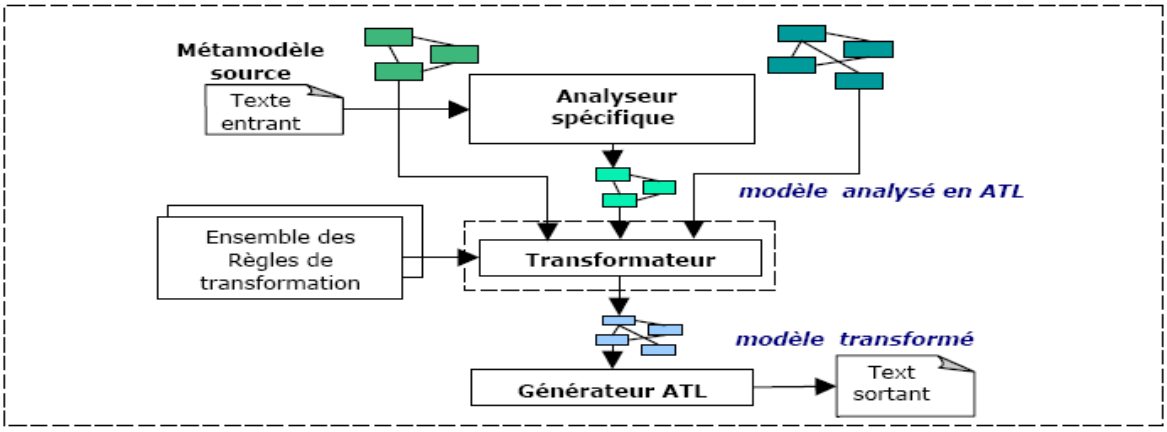


Syntaxe concrète textuelle et graphique d'ATL

L 'en-tête « header », Les fonctions « Helpers » et **Les règles de transformation « Rules »** vont être expliqué dans le point suivant.

6 Environnement d'exécution

Le schéma ci-dessous donne une vision globale d'une transformation en ATL. L'environnement d'exécution ATL comporte le Transformateur mais il faut fournir un analyseur pour chaque langage source ainsi qu'un générateur pour chaque langage cible. Bien évidemment certains langages sont fournis. Des projets sont en cours pour générer des analyseurs à partir de méta-modèles.

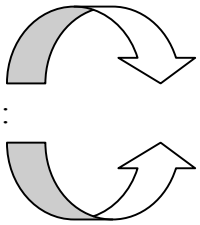


Vision conceptuelle d'un processus ATL

7 Les type de transformation d'ATL

Une transformation est une opération qui prend un modèle source en entrée fournit un modèle cible en sortie

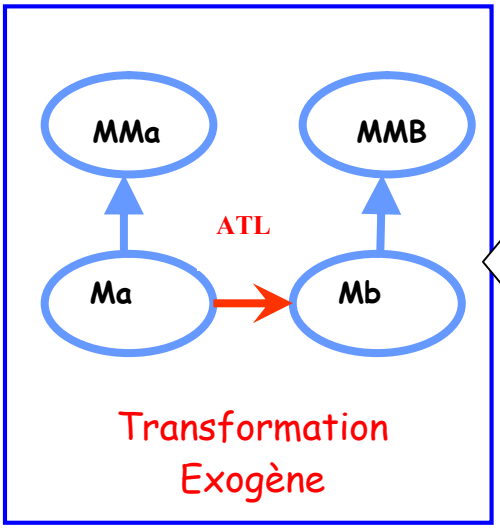
Il existe deux sortes de transformation d'ATL :



1- Transformation de modèle à modelé dans le même contexte

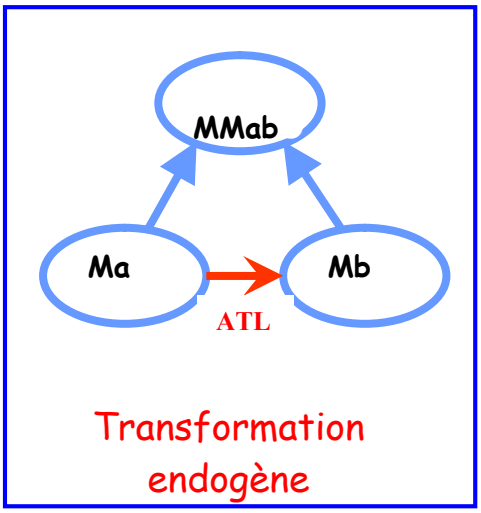
2 -Transformation de modèle vers un autre modèle

7.1 Transformation de modèle vers autre modèle



- Entre 2 espaces technologiques différents
- Les modèles source et cible sont conformes à des méta-modèles différents
- Transformation d'un modèle UML en programme Java
- Transformation d'un fichier XML en schéma de BDD
- modèle UML en un autre modèle UML

7.2 Transformation de modèle à un modèle dans le même contexte



- Dans le même espace technologique
- Les modèles source et cible sont conformes au même méta-modèle
- Transformation d'un modèle UML en un autre modèle UML

7.3 Module ATL

La transformation de notre modèle Source vers notre modèle Cible consiste à :

- Chaque paquetage Source donne un paquetage Cible.
- Chaque classe Source donne une classe Cible :

- Chaque Data Type Source donne un type primitif correspondant en Cible :
- Chaque attribut Source donne un attribut Cible respectant le nom, le type, la classe d'appartenance et les modifies.

Le module ce compose de cinq point suivant :

7.3.1 Entête de Transformation de modèle à modelé

```
module Source2Cible;
create OUT : Cible from IN : Source;
```

La convention veut qu'une transformation d'un modèle à un autre, soit nommée d'après les méta-modèles avec un 2 (to) ajout´e entre les deux.

OUT et IN sont les noms données aux modèles. Ils ne sont pas utilisés par la suite.

7.3.2 Helpers

Un helper est l'équivalent d'une fonction auxiliaire ; Il est défini sur un contexte et pourra être appliqué sur toute expression ayant pour type de ce contexte.

Un helper peut prendre des paramètres et possède un type de retour. Le code d'un helper est une expression OCL.

7.3.3 Règles déclaratives

1-règle déclenchée sur un élément du modèle :

```
rule P2P {
        from e: UML!Package (e. oclIsTypeOf(UML!Package))
        to out: JAVA!Package (
        name <- e.getExtendedName()
      )
}
```

Une règle est caractérisée par deux éléments obligatoires :

- Un motif sur le modèle source **from** avec une éventuelle contrainte.
- Un ou plusieurs motifs sur le modèle cible **to** qui expliquent comment les éléments cibles sont initialisés à partir des éléments sources correspondant.

Une règles peut aussi définir : une contrainte sur les éléments correspondant au motif source, une partie impérative et des variables locales.

2-Lien entre éléments cibles et sources:

```
rule C2C {
        from e: UML!Class
        to out: JAVA!JavaClass (
        name <- e.name,
        isAbstract <- e.isAbstract,
        isPublic <- e.isPublic (),
        package <- e.namespace
)
}
```

Lors de la création d'un élément cible à partir d'un élément source, ATL conserve un lien de traçabilité entre les deux.

Ce lien est utilisé pour initialiser un élément cible dans la partie to.

3-partie impérative (do):

```
rule C2C {

    from e: UML!Class
    to out: JAVA!JavaClass (
)
do { package <- e.namespace
}
}
```

La clause **do** est optionnelle ; do contient des instructions (partie impérative d'une règle) .ces instructions sont exécutées après l'initialisation des éléments cibles.

4-Variables locales (using) :

```
from
c: GeometricElement!Circle
using { pi : Real = 3.14;
area : Real = pi _ c. radius . square ();
}
```

La clause **using** est optionnelle ; elle permet de déclarer des variables locales à la règle.

Les variables peuvent utilisées dans les clauses **using, to** et **do.**

Une variable est caractérisée par son nom, son type et doit être initialisée avec une expression OCL.

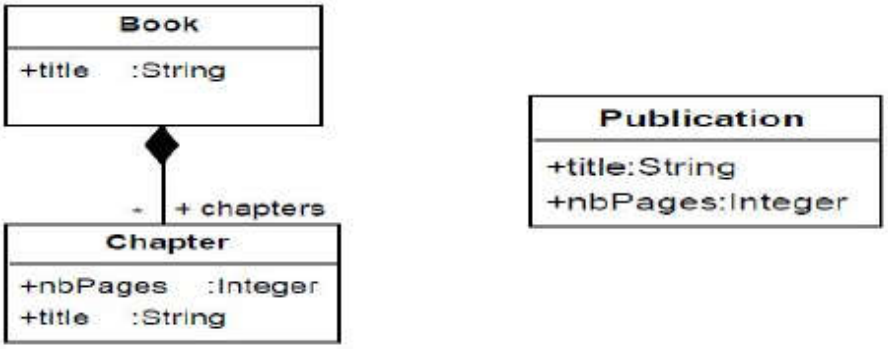
7.3.4 Règles impératives

```
rule  newPackage(qualifiedName: String) {
to p: JAVA!Package (
name <- qualifiedName
)
```

Equivalent d'un helper qui peut créer des éléments dans le modèle cible doit être appelée depuis une règles déclaratives ou une autre règles impératives ne peut pas avoir de partie **from** peut avoir des paramètres.

Exemple :

Si nous considérons l'exemple de la transformation du méta-modèle Book vers le méta-modèle Publication (ATL) présentés par la figure suivante :



Example: Book → Publication

Nous pouvons définir un *helper* opération qui permet de calculer la somme des pages du livre en fonction de la somme des pages de ces chapitres, ceci se présente comme suit :

```
helper context Book!Book def : getSumPages() : Integer = self.chapters-
>collect(f|f.nbPages).sum();
```

7.3.5 Exécution d'un module ATL

L'exécution d'un module se fait en trois étapes :

1. Initialisation du module :

   Initialisation des attributs définis dans le contexte du module.

2. mises en correspondance des éléments sources des règles déclaratives :
   Quand une règle correspond à un élément du modèle source, les éléments cibles correspondants sont créés.
3. initialisation des éléments du modèle cible.

Le code impératif des règles **do** est exécuté après l'initialisation de la règle correspondante. Il peut appeler les règles impératives. [Benoit et al ,07]

8 Avantages et inconvénients

On peut dire qu'ATL a comme avantage :

- Le langage ATL a une syntaxe de base très simple. Un développeur ayant une bonne compréhension des concepts typiques d'UML pourra utiliser ATL relativement naturellement.
- La syntaxe graphique ne permet de fournir qu'une vision globale de la transformation, mais celle-ci est néanmoins intéressante en complément avec le texte.
- ATL semble prometteur pour réaliser des transformations flexibles, bidirectionnelles et paramétrées et la notion de traçabilité.

Et inconvénient :

- OCL est complexe à première vue et n'est pas évident à lire au début. OCL est plus complexe que XPath dans XSLT. En fait pour écrire une transformation en ATL, il est nécessaire de connaître OCL, MOF et ATL lui-même. Le fait que le ATL et OCL soient intégrés rend difficile la lecture lorsque l'on n'est pas habitué.

EXEMPLE D'UNE TRANSFORMATION DE MODELE VERS UN AUTRE MODELE

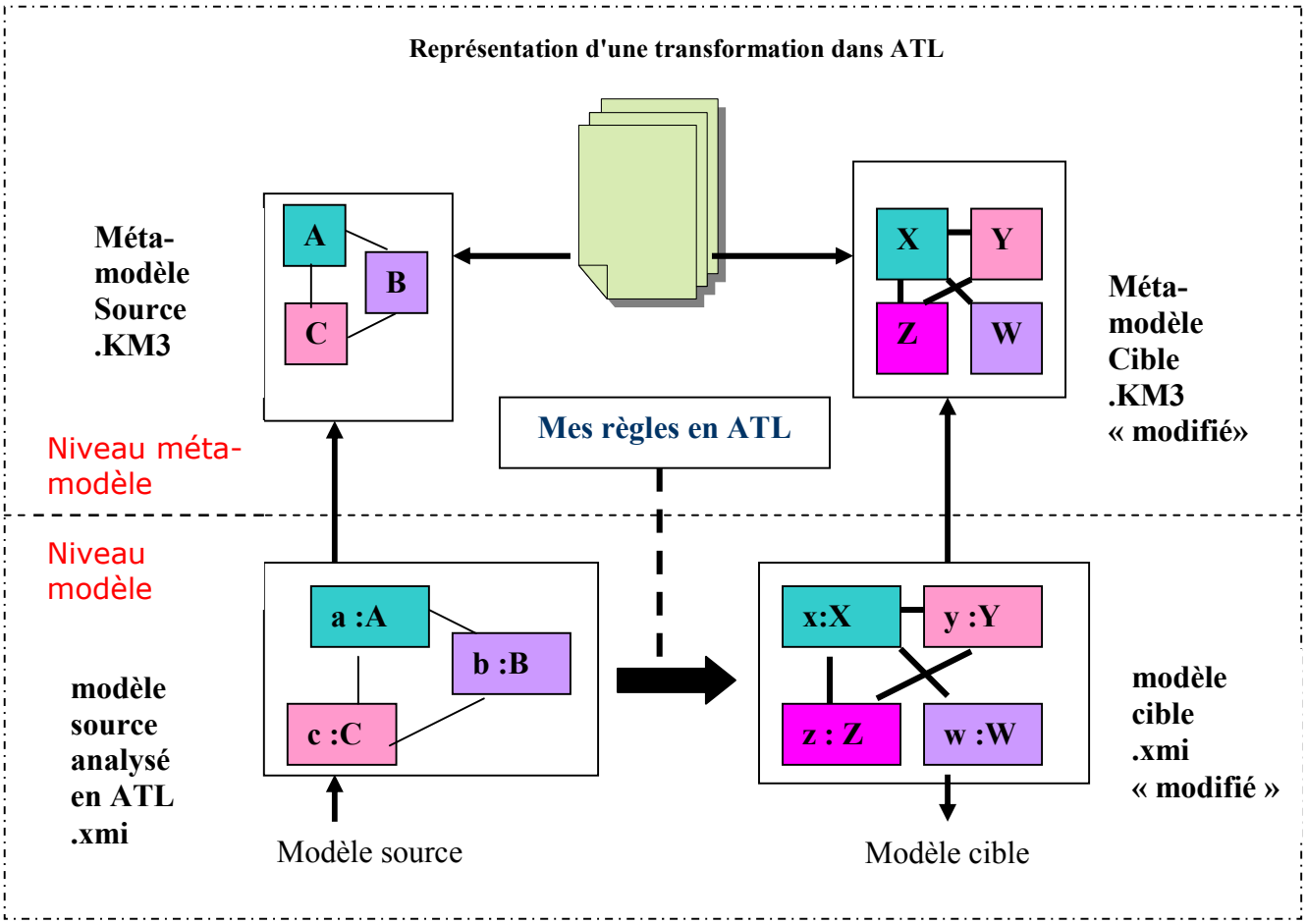
3.1 Description générale de l'exemple

Considérons notre exemple qui permet de générer un modèle cible à partir de d'un model source.

Les informations que nous devons apporter au système pour nourrir le moteur de traduction ATL sont :

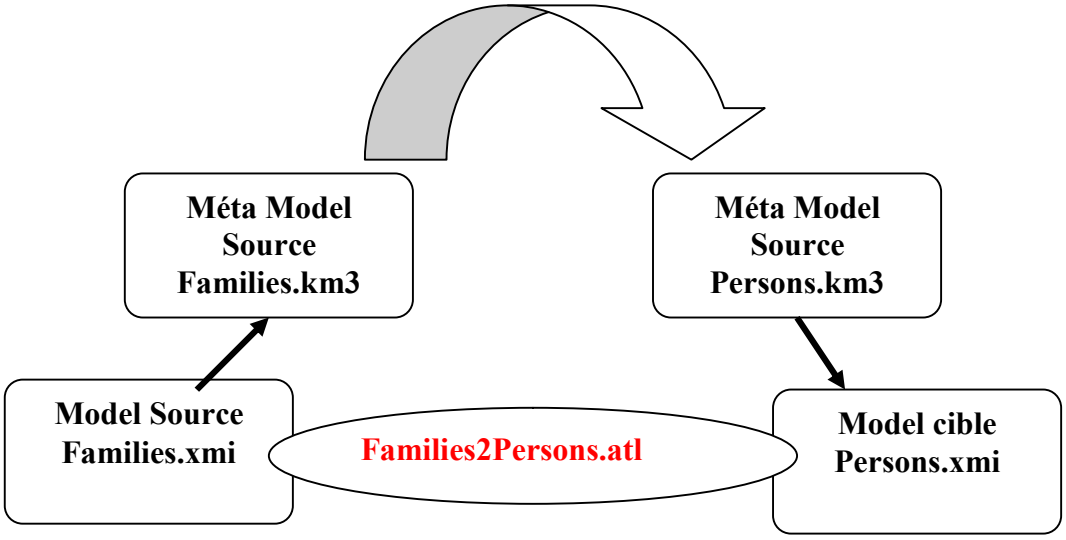
- Le méta-modèle en KM3.
- Le méta-modèle KM3 « modifié ».
- Le modèle en XMI.
- Le code ATL de transformation.

On peu présenter cette transformation avec ce schéma :



3.2 Description de l'exemple Families2Persons

La description de l'exemple Families2Pernons peut être décrite comme suit :



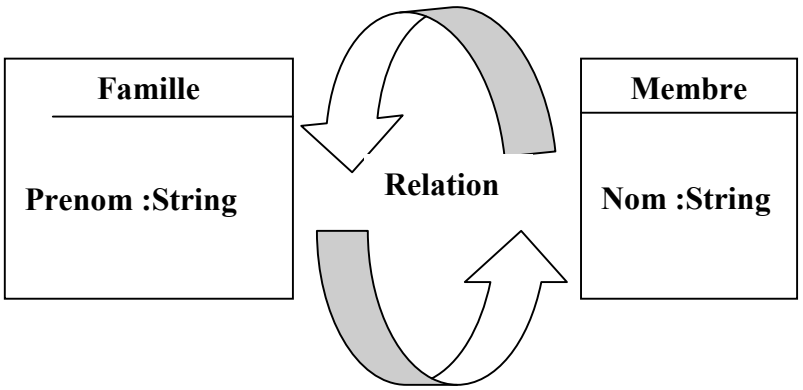
Pour faire une transformation ATL en a besoin de :

1. Source méta-model en KM3« families » .
2. Source model en XMI conforme a « Families»
3. Cible meta-model in KM3 « Persons».
4. Une transformation model en ATL « « Families2Persons ».

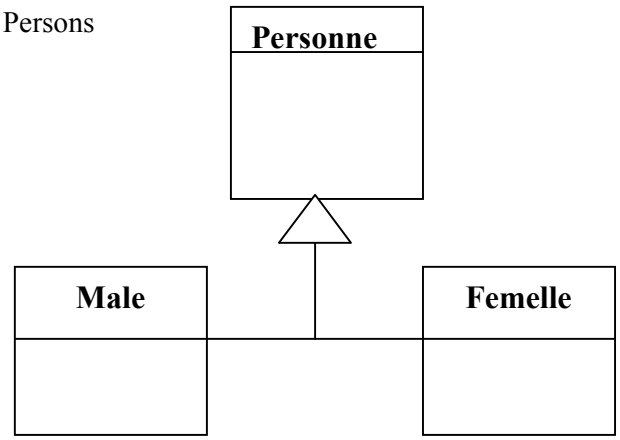
Quand une transformation ATL est exécutée on obtiens :

5. Cible model en XMI conforme a « Persons».

- Le modèle Families est une collection de familles ou chaque une a un Nom et elle se compose de membre « père, mère, plusieurs fils et plusieurs filles ».



- Le modèle Persons

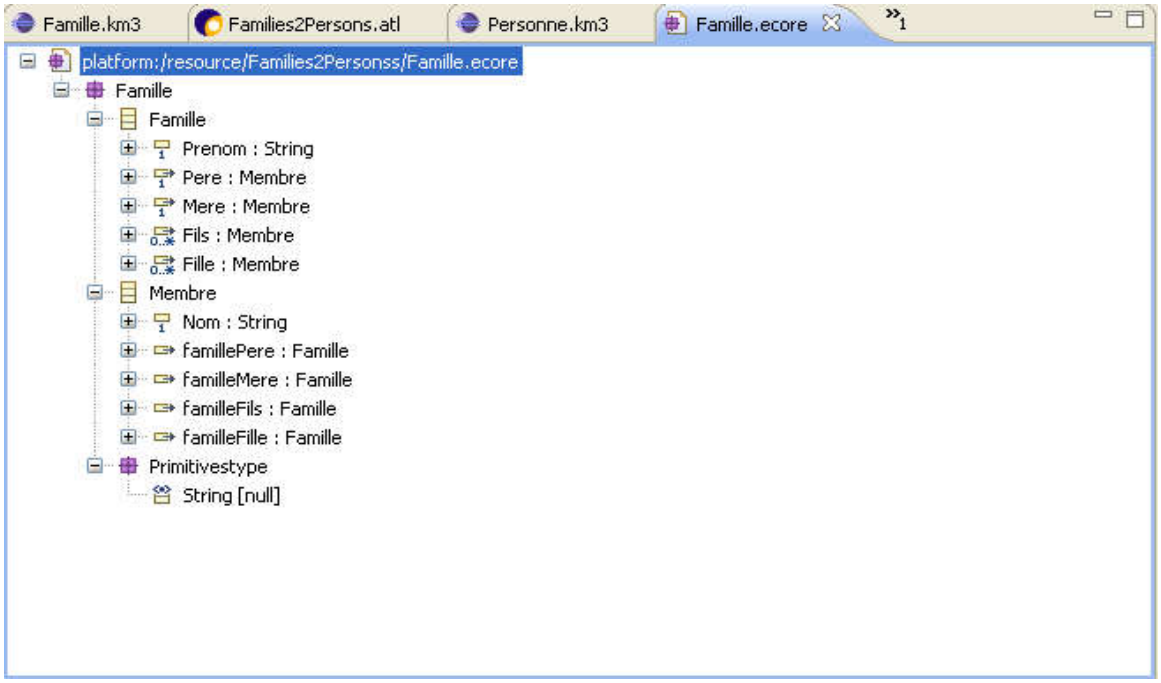


3.3 Les outils utilisés pour la transformation

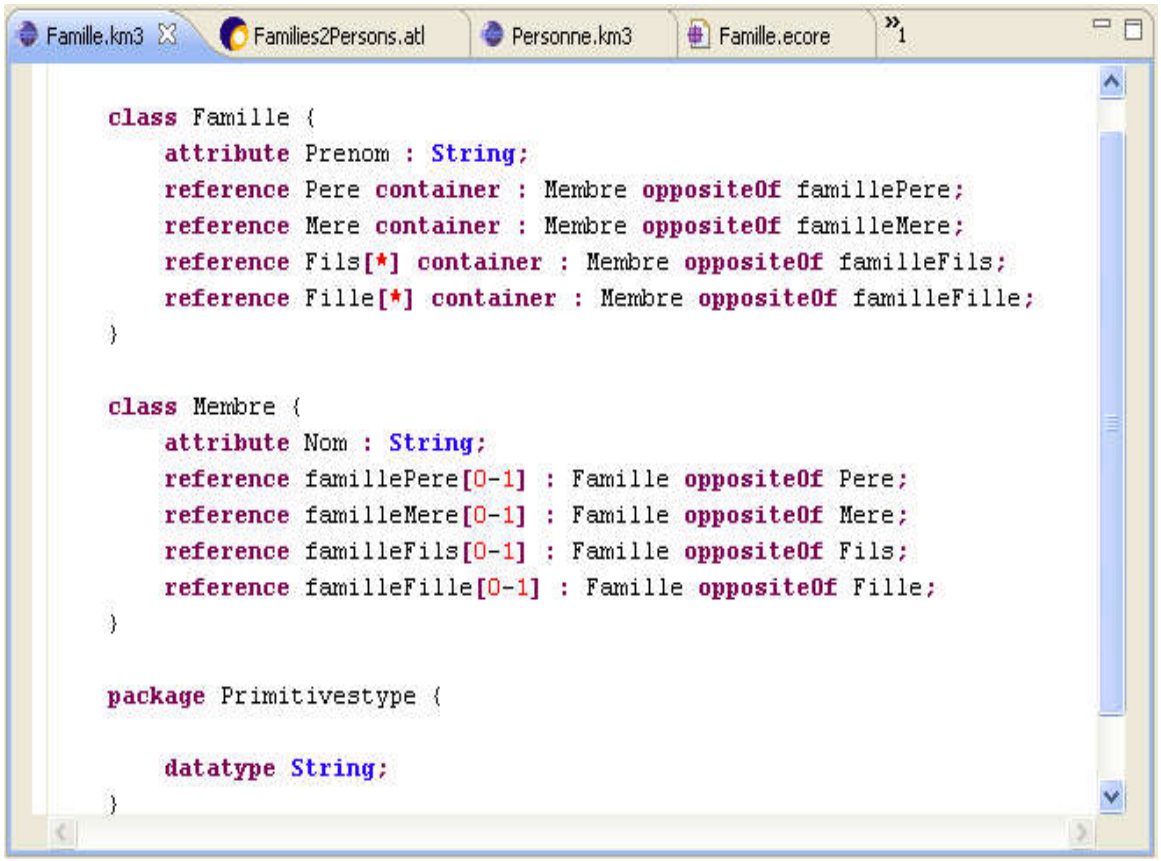
Il ya plusieurs outils qui nous assure la transformation ATL on a le Papyrus, Atl_UML2_Bundle_2.0.ORC_Windows. Tous ces outils sont implantés dans la plateforme Eclipse. on a utilisé Atl_UML2_Bundle_2.0.ORC_windouws qui nous a idée a faire une transformation de model comme il a été expliqué lors de la présentation oral de ce travail.

Voici quelque capture écran qui permet de décrire la transformation avec l'outil utilisé :

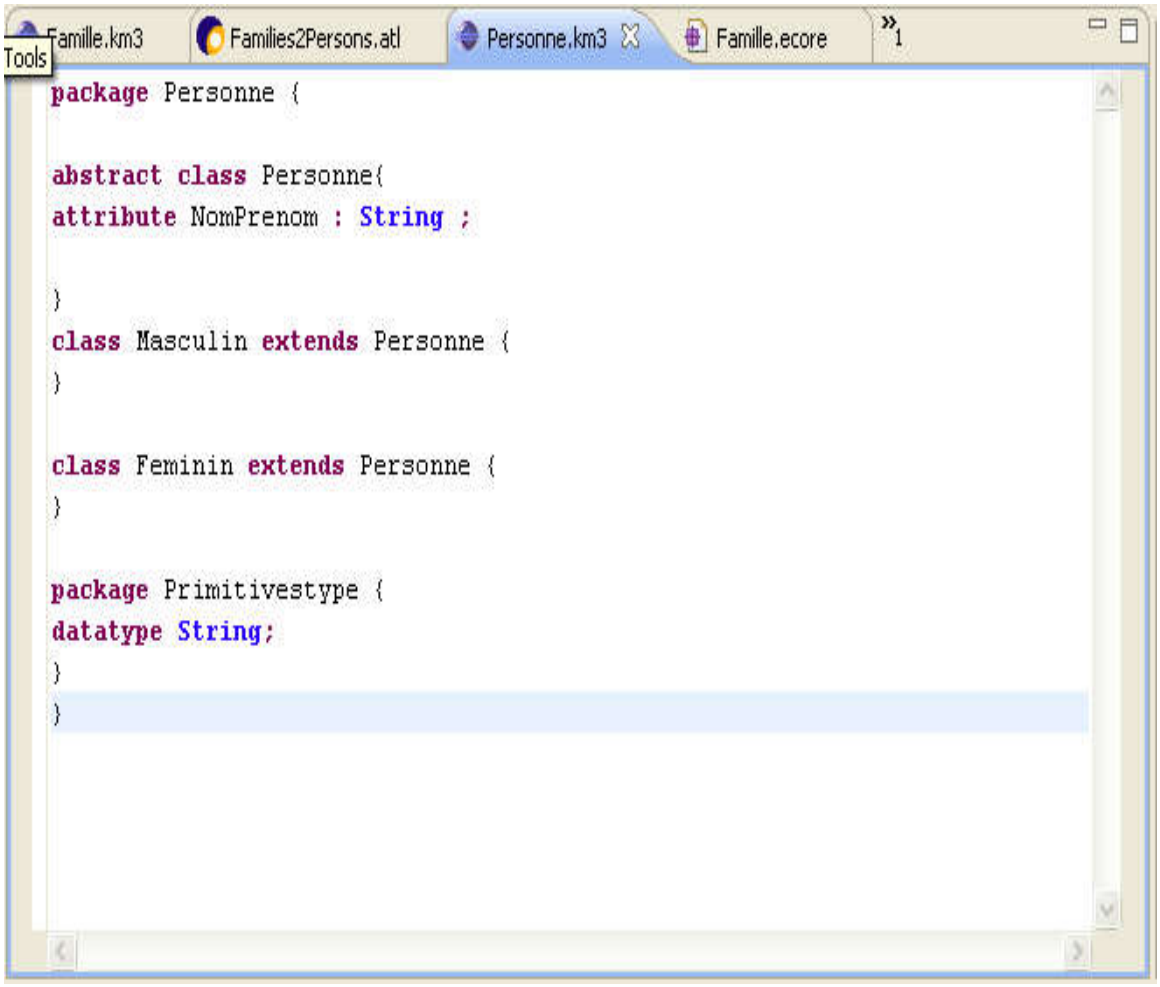
Modèle Source Famille.encore :



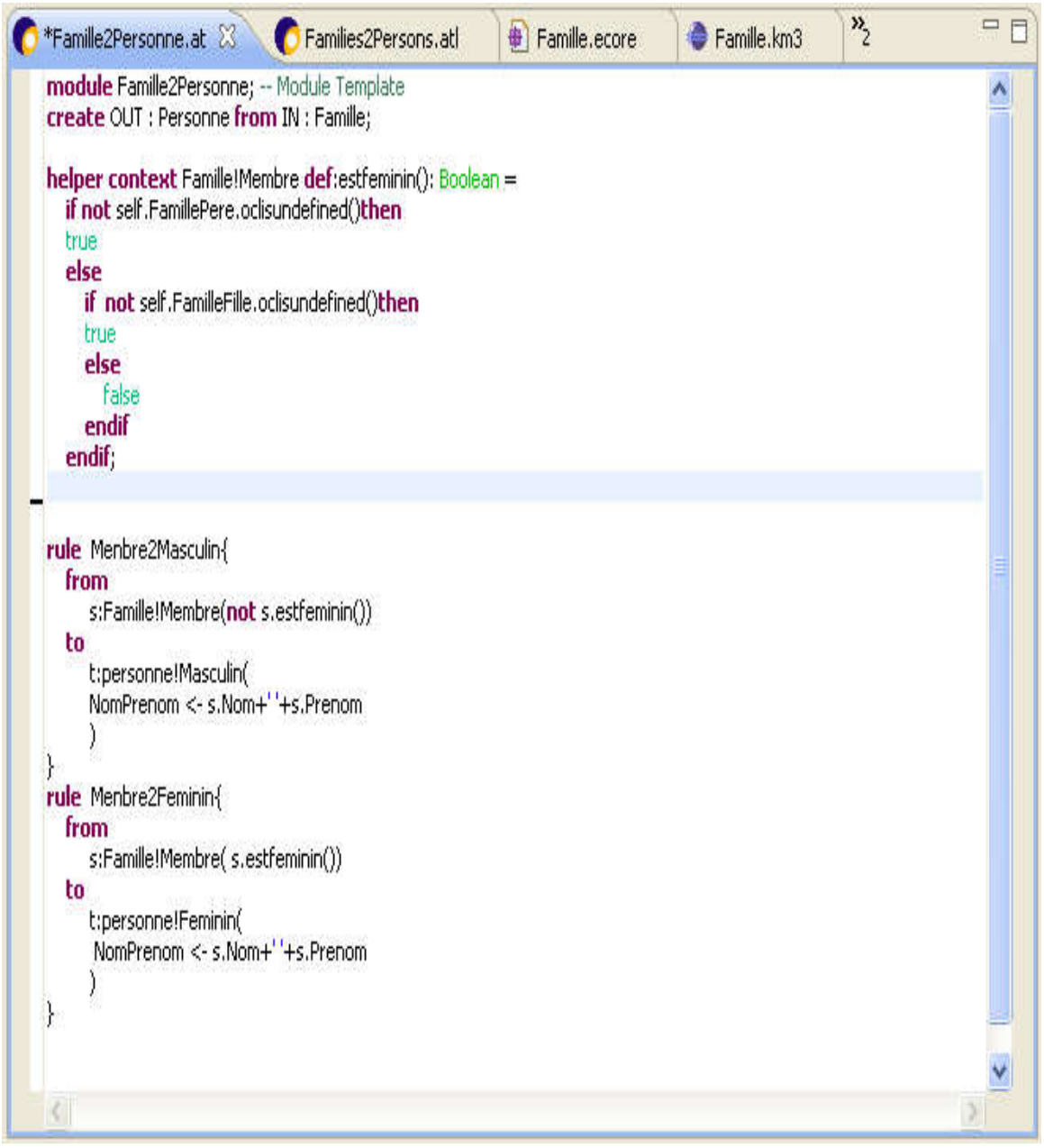
Meta-modèle Source Famille.km3



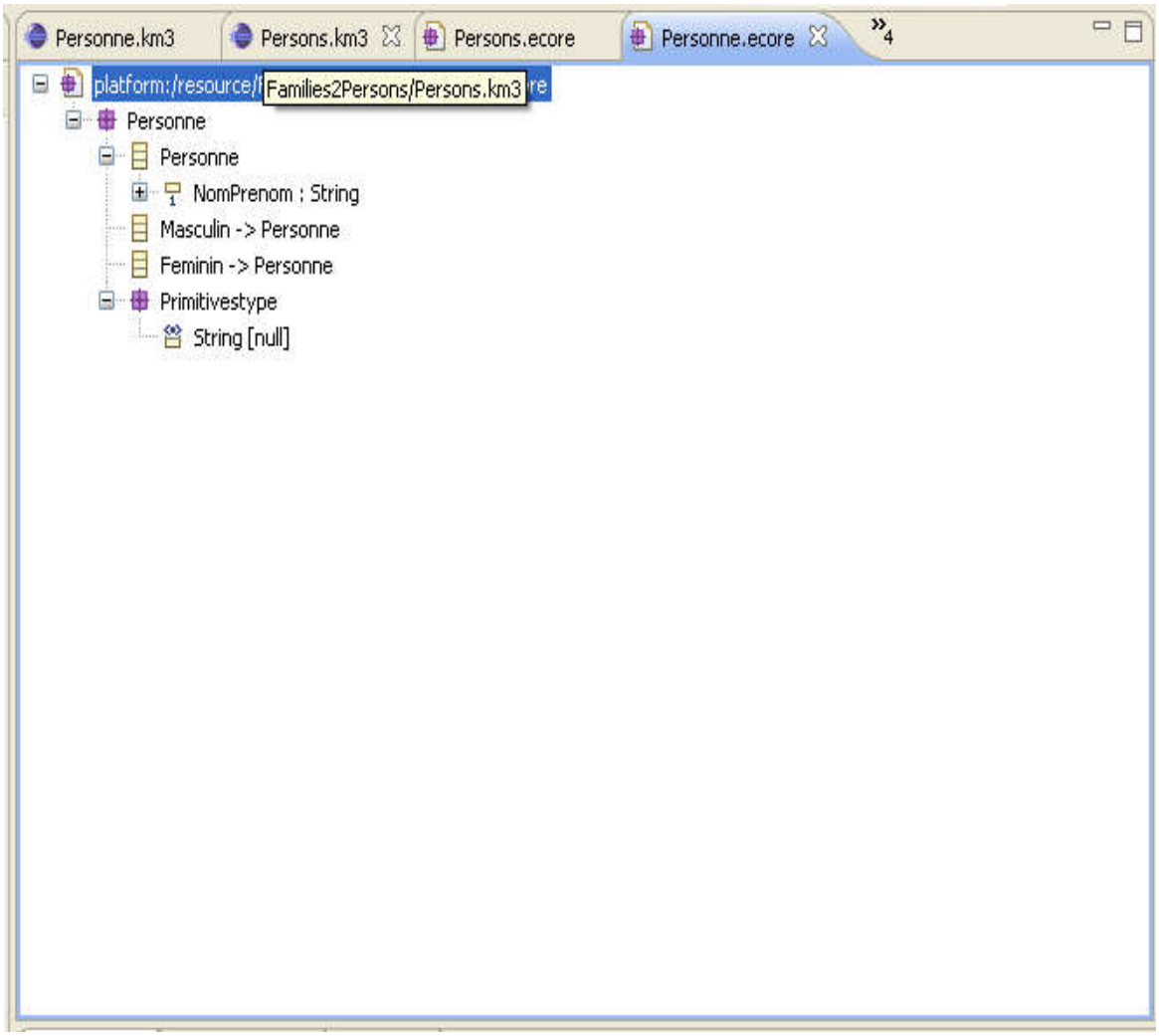
Meta-modèle Cible personne.km3



Le module ATL :



Le résultat c'est le model Cible personne.encore


Personne.km3
Persons.km3
Persons.ecore
Personne.ecore
platform:/resource/
Families2Persons/Persons.km3
Personne
Personne
NomPrenom : String
Masculin -> Personne
Feminin -> Personne
Primitivestype
String [null]

CONCLUSION

ATL a pleinement révélé son adéquation avec notre besoin de transformer les modèles qui nous intéressent, et aucun verrou technique ne laisse présager de difficulté à exprimer dans une transformation automatique . Comme il a déjà été évoqué, plus que sur le codage, le travail du programmeur portera sans doute sur un patient dialogue avec les concepteurs du processus de transformation pour précisément spécifier les méta-modèles sources et cibles de la transformation, qui devront être assez complets et précis pour permettre d'exprimer l'ensemble des concepts utilisés dans les deux mondes de modélisation.

ATL souffre cependant d'une mise en œuvre fastidieuse, en tout cas sur le plan des outils techniques à utiliser pour sa programmation. Son intégration sous forme de plugin dans la plateforme Eclipse oblige en effet le programmeur à procéder à l'installation de plusieurs éléments de sources diverses, dont les versions évoluent de manière permanente et souvent incompatibles les unes avec les autres. Ainsi, trouver à un instant t une combinaison des versions de ces éléments qui permettent d'obtenir un environnement de développement à la fois fonctionnel, robuste et permettant de profiter de toutes les fonctionnalités de développement peut parfois relever de la gageure. La situation a cependant tendance à s'arranger avec le souci des développeurs Nantais du langage de livrer des « bundle » complets, intégrant dans un seul paquet d'installation tous les éléments nécessaires et compatibles entre eux. Un de ces bundle est déjà disponible, mais pour l'instant exclusivement pour le système d'exploitation Microsoft Windows.

REFERENCES

[Antoine,06] Antoine Wiedeman,« Approche MDA pour la transformation d'un diagramme de classes conforme UML 2.0 en un schéma relationnel conforme CWM et normalisé_L »,Juin 2006.

[NGUYEN,04] NGUYEN Thi Thanh Tam**,** « Transformations dans de multiples espaces technologiques pour l'Ingénierie Dirigée par les Modèles »,Septembre 2004.

[Benoit et al ,07] Benoit Combemale-Xavier Cregut-Marc Pantel ,« Transformation de Modèles Introduction à ATL »,Novembre 2007.