Chapitre 5 : Transformation de Modèles

1. Introduction : Pourquoi transformer des modèles ?

Dans une approche **MDA (Model Driven Architecture)**, le développement logiciel est centré sur les **modèles** plutôt que directement sur le code. Chaque modèle correspond à un **niveau d'abstraction** :

Niveau	Modèle	Description
CIM	Computation Independent Model	Modèle des besoins métier (ex. cas d'utilisation)
PIM	Platform Independent Model	Modèle conceptuel (ex. UML, sans plateforme)
PSM	Platform Specific Model	Modèle adapté à une plateforme donnée (Java, .NET…)
Code	Code source généré	Résultat final de la transformation

La transformation de modèles est donc le mécanisme qui permet de **passer d'un niveau à l'autre**, automatiquement ou semi-automatiquement.

2. Notions de transformation

2.1 Transformation de modèle

Une **transformation de modèle** est un processus qui, à partir d'un ou plusieurs **modèles sources**, produit un ou plusieurs **modèles cibles**, selon un ensemble de **règles** définies.

Formellement :

$$T : Ms \rightarrow Mt$$

avec :

- Ms = **modèle source**, conforme au métamodèle MMs
- Mt= **modèle cible**, conforme au métamodèle MMt
- T = **transformation**, définie par un ensemble de règles

2.2 Types de transformations

Type	Description	Exemple
Transformation verticale	Change le **niveau d'abstraction**	UML (PIM) → Code Java (PSM)
Transformation horizontale	Reste **au même niveau**, modifie ou traduit le modèle	UML → UML simplifié (refactoring, optimisation)

Exemple :

- **Verticale** : un diagramme de classes UML transformé en classes Java.
- **Horizontale** : un diagramme UML enrichi (ex. ajout d'attributs dérivés).

2.3. Transformation endogène et exogène

Une **transformation de modèle** peut être **endogène** ou **exogène** selon les métamodèles impliqués.
La **transformation endogène** s'effectue **au sein du même métamodèle** : le modèle source et le modèle cible partagent la même structure. Elle sert souvent à **raffiner, filtrer ou restructurer** un modèle existant (ex. UML → UML).

La **transformation exogène** relie des **métamodèles différents** : elle permet de passer d'un domaine à un autre, comme la **transformation d'un modèle UML vers un modèle Java ou relationnel**.

3. Règles de transformation

3.1 Définition

Une **règle de transformation** définit **comment** un élément du modèle source est **converti** en élément du modèle cible.

Forme générale :

```
rule NomDeLaRègle {
   from s : SourceMM!TypeSource
   to   t : CibleMM!TypeCible (
        attribut1 <- s.attributA,
        attribut2 <- s.attributB
   )
}
```

3.2 Types de règles

- **Règles de correspondance (mapping)** : établissent les relations entre éléments source et cible.
- **Règles de filtrage** : excluent certains éléments.
- **Règles de création** : ajoutent de nouveaux éléments.
- **Règles de navigation** : parcourent les relations entre éléments du modèle source.

4. Langages de transformation de modèles

4.1.QVT (Query/View/Transformation)

QVT (Query/View/Transformation) est une **spécification standardisée par l'OMG (Object Management Group)** pour décrire des **transformations de modèles** entre métamodèles définis en **MOF (Meta Object Facility)**.

En d'autres termes, QVT permet d'exprimer **comment un modèle source** (par ex. UML) peut être **transformé en un autre modèle cible** (par ex. un modèle de base de données ou un modèle Java).

4.2. Architecture générale de QVT

Le langage QVT repose sur trois sous-langages complémentaires :

Sous-langage	Type	Description	Usage principal
QVT-Relations	Déclaratif	Exprime les correspondances entre modèles	Pour les transformations "bidirectionnelles" (UML ↔ DB)
QVT-Core	Intermédiaire, formel	Noyau sémantique de QVT	Base de QVT-Relations
QVT-Operational (QVT-OM)	Impératif	Décrit les actions de transformation en détail	Pour transformations unidirectionnelles (UML → Java)

4.3. Les trois composants du nom QVT

Élément	Rôle
Query	Extraire ou interroger un modèle (similaire à une requête SQL).
View	Créer une "vue" dérivée d'un modèle existant.
Transformation	Modifier ou générer un nouveau modèle à partir d'un autre.

Ces trois aspects forment la logique complète du **Model Transformation Framework**.

4.4. QVT-Relations (QVT-R)

4.4.1. Nature

Langage **déclaratif**, proche des **contraintes OCL (Object Constraint Language)**.
On déclare **les correspondances** entre les éléments du modèle source et ceux du modèle cible.

4.4.2. Structure d'une transformation QVT-R

```
transformation UML2RelationalModel (uml : UML, relational : RDBMS);

top relation Class2Table {
   domain uml c : UML::Class {};
   domain relational t : RDBMS::Table {
      name = c.name
   };
}

relation Attribute2Column {
   domain uml a : UML::Property {};
   domain relational col : RDBMS::Column {
      name = a.name,
      type = a.type.name
   };
}
```

Explication :

- transformation : définit le nom et les modèles source/cible.
- top relation : point d'entrée principal.
- domain : indique les éléments des deux métamodèles (source et cible).
- name = c.name : correspondance entre les attributs.

Ici, chaque **Classe UML** est transformée en **Table** et chaque **Attribut UML** en **Colonne**.

4.4.3. Correspondance bidirectionnelle

Une relation QVT peut être **utilisée dans les deux sens** :

- UML → Table (génération de base de données),
- Table → UML (ingénierie inverse).

C'est un **avantage majeur** par rapport à ATL (qui est unidirectionnel par défaut).

4.5. QVT-Operational (QVT-OM)

4.5.1. Nature

Langage **impératif**, orienté "procédural".
Il permet de décrire **comment exécuter la transformation étape par étape**, comme un algorithme.

4.5.2. Structure générale

```
transformation UML2Java (uml : UML, java : Java);

main() {
   uml.classes->forEach(c | mapClass(c));
}

mapping UML::Class::mapClass() : Java::ClassDeclaration {
   name := self.name;
   fields := self.attributes->map mapAttribute();
}

mapping UML::Property::mapAttribute() : Java::FieldDeclaration {
   name := self.name;
   type := self.type.name;
}
```

Explication :

- main() : point d'entrée de la transformation.
- mapping : définit la correspondance entre un type UML et un type Java.
- self : fait référence à l'élément courant du modèle source.
- := : affectation.
- On stocke le résultat dans la variable ou propriété fields.
- Donc **fields = collection des attributs transformés**.

Ici, on transforme explicitement des classes UML en classes Java.

4.6. Exemple de résultat

Entrée UML :

Class: Student
- id : Integer
- name : String

Sortie Java générée :

```
public class Student {
   private int id;
   private String name;
}
```

4.7. QVT-Core

Ce sous-langage est rarement écrit directement par le concepteur.
Il sert de **couche intermédiaire formelle** entre QVT-Relations et QVT-Operational.
C'est une sorte de "langage pivot" pour les outils d'exécution.

4.8. Outils supportant QVT

Outil	Description
Eclipse QVTd	Implémentation officielle de QVT (déclaratif et impératif) sous Eclipse.
Medini QVT	Outil Eclipse pour exécuter QVT-Relations (projet de Modelware).
SmartQVT	Exécution de QVT-Operational Mapping (recherche académique).

4.9. Exemple complet : UML → Base de données (QVT-Relations)

Transformation

```
transformation UML2DB (uml : UML, db : RDBMS);

top relation Package2Schema {
   domain uml p : UML::Package {};
   domain db s : RDBMS::Schema {
      name = p.name
   };
}

relation Class2Table {
   domain uml c : UML::Class {};
   domain db t : RDBMS::Table {
      name = c.name
   };
}

relation Attribute2Column {
   domain uml a : UML::Property {};
   domain db col : RDBMS::Column {
      name = a.name,
      type = a.type.name
   };
}
```

Résultat attendu

UML	RDBMS
Package University	Schema University
Class Student	Table Student
Property id : Integer	Column id : INT

4.10. Cycle d'utilisation QVT (dans la chaîne MDA)

1. Papyrus (UML Model)
   ↓
2. Export en XMI
   ↓
3. QVT Transformation
   ↓
4. Nouveau modèle (Java, SQL, etc.)
   ↓
5. Acceleo pour génération de code final

5. ATL (ATLAS Transformation Language)

5.1. Définition

ATL (ATLAS Transformation Language) est un **langage de transformation de modèles** développé par l'équipe **ATLAS (INRIA & LINA, France)**.
Il fait partie de la **plateforme Eclipse Modeling Framework (EMF)**.

ATL permet de transformer automatiquement un modèle **source** (conforme à un métamodèle) en un modèle **cible** (conforme à un autre métamodèle).

5.2. Syntaxe générale d'un programme ATL

```
module NomDuModule;

create OUT : MetaModeleSortie from IN : MetaModeleEntree;

-- (Optionnel) Helpers
helper context <NomMetaclasse> def: <nomHelper> : <Type> =
   <expression>;

-- (Optionnel) Helpers avec paramètres
helper def: <nomHelper>(param : Type) : Type =
   <expression>;

-- Règles de transformation
rule NomDeLaRegle {
```

```
    from
        sourceVar : MetaModeleEntree!MetaClasseSource (optional guard)
    to
        targetVar : MetaModeleSortie!MetaClasseCible (
            -- initialisation des attributs
            attribut1 <- <expression>,
            attribut2 <- <expression>,
            ...
        )
}

-- (Optionnel) Règles "lazy" (évaluées uniquement si appelées)
lazy rule NomLazyRule {
    from
        ...
    to
        ...
}

-- (Optionnel) Règles "unique lazy" (créent un seul élément par entrée)
unique lazy rule NomUniqueLazyRule {
    from
        ...
    to
        ...
}
```

1. Module ATL

Un fichier ATL commence toujours par la déclaration du module :

module NomDuModule;

- module : mot-clé obligatoire
- NomDuModule : nom de votre transformation
- Fichier ATL : .atl

2. Création des modèles source et cible

create OUT : MetamodelCible from IN : MetamodelSource;

- OUT : nom du modèle **cible**
- IN : nom du modèle **source**
- MetamodelCible et MetamodelSource : méta-modèles respectifs
- Cette ligne **déclare le contexte global** de la transformation

3. Helpers (fonctions)

Les **helpers** sont des fonctions auxiliaires utilisées dans les règles.

helper context MetamodelSource!Type def: nomHelper : TypeRetour = expression;

- context : type sur lequel le helper est défini
- nomHelper : nom de la fonction
- TypeRetour : type de retour (Integer, String, Boolean…)
- expression : corps du helper (souvent OCL)

Exemple :

helper context UML!Class def: nbAttributes : Integer = self.attributes->size();

4. Règles de transformation (Rules)

4.1 Règles déclaratives (Matched Rule)

```
rule NomRegle {
   from
      s : MetamodelSource!TypeSource
   to
      t : MetamodelCible!TypeCible (
         prop1 <- s.propSource,
         prop2 <- s.autreProp
      )
}
```

- from : type source et variable
- to : type cible et correspondance des propriétés
- Utilisée pour les transformations **structurelles simples**

4.2 Règles lazy

- Ne s'exécutent que lorsqu'elles sont **appelées explicitement**.
- Syntaxe :

```
lazy rule NomRegleLazy {
   from
      s : MetamodelSource!TypeSource
   to
      t : MetamodelCible!TypeCible (
         prop <- s.autreProp
      )
}
```

4.3 Règles imperatives / do

- Permettent de **contrôler le flux** (boucles, conditions, calculs complexes)

```
rule ExempleImperatif {
   from
      s : MetamodelSource!TypeSource
   to
      t : MetamodelCible!TypeCible
   do {
      t.prop <- s.collection->collect(e | thisModule.RegleLazy(e));
   }
}
```

Exemple pour chaque type de règle

1. Matched Rule (règle classique)

C'est la règle ATL **la plus courante**.
Elle se déclenche automatiquement pour chaque élément du modèle source qui satisfait la condition.

Exemple : UML!Class → Rel!Table

```
rule Class2Table {
   from
      c : UML!Class
   to
      t : Rel!Table (
         name <- c.name
      )
}
```

Pour **chaque UML!Class**, ATL crée une **Table**.

2. Lazy Rule

- Elle **ne s'exécute que si elle est appelée**.
- Elle ne parcourt pas automatiquement le modèle source.

Exemple : transformer un Attribut → Colonne uniquement quand on appelle la règle

```
lazy rule Attribute2Column {
   from
      a : UML!Attribute
   to
      col : Rel!Column (
         name <- a.name,
         type <- a.type
      )
}
```

Usage dans une autre règle :

```
columns <- c.attribute->collect(a | thisModule.Attribute2Column(a))
```

Très utile pour contrôler manuellement la transformation.

3. Unique Lazy Rule

Comme une lazy rule

- Mais **si on l'appelle deux fois sur le même élément**, elle renvoie **toujours le même résultat** évite les duplications.

Exemple : ne créer qu'une seule Table par Class (même si appelée plusieurs fois)

```
unique lazy rule UniqueClass2Table {
   from
      c : UML!Class
   to
      t : Rel!Table (
         name <- c.name
      )
}
```

Appel :

```
tableRef <- thisModule.UniqueClass2Table(c);
```

Même si on appelle la règle plusieurs fois, **la même instance t** sera renvoyée.

4. Called Rule (style impératif)

Ce n'est pas un type officiel, mais ATL permet d'écrire des règles **appelées comme des fonctions**, utiles pour la logique procédurale.

Exemple : transformer un nom de classe en nom de table

```
rule ConvertNameInUpperCase {
   from
      s : String
   to
      out : String (
         value <- s.toUpper()
      )
}
```

5.3. Exemple complet de programme ATL (UML → Relational)

```
module UML2Relational;

create OUT : Relational from IN : UML;

-- Helper
helper context UML!Class def: nbAttributes : Integer = self.attributes->size();

-- Règle principale
rule Class2Table {
   from
      c : UML!Class
   to
      t : Relational!Table (
         name <- c.name,
         columns <- c.attributes->collect(a | thisModule.Attribute2Column(a))
      )
}

-- Règle lazy pour les attributs
lazy rule Attribute2Column {
   from
      a : UML!Attribute
   to
      c : Relational!Column (
         name <- a.name,
         type <- a.type.name
      )
}
```

- Module : UML2Relational
- Modèles : IN : UML, OUT : Relational
- Helpers : nbAttributes
- Règles : Class2Table (principale) et Attribute2Column (lazy)

5.4. Mécanisme d'exécution

Étapes typiques :

1. Créer le **métamodèle source** (UML.ecore)
2. Créer le **métamodèle cible** (Java.ecore)
3. Charger le **modèle source (XMI)**
4. Exécuter le fichier .atl
5. Générer le **modèle cible (XMI)**
6. (Optionnel) Convertir ce modèle en code avec **Acceleo**

5.5. Exemple d'usage

- UML → Java (PIM → PSM)
- UML → SQL (UML Class → Table)
- XML → UML

5.6. ATL dans la chaîne MDA

Papyrus (UML) → EMF (Ecore)
↓
ATL (Transformation)
↓
Acceleo (Code Génération)
↓
Java / SQL / C#

ATL est donc **le cœur de la transformation** entre modèles dans MDA.

5.7. Exemple complet : UML → Java

UML d'entrée :

Class: Person
- name : String
- age : Integer

Transformation ATL :

```
module UML2Java;
create OUT : Java from IN : UML;

rule Class2JavaClass {
   from
      c : UML!Class
   to
      j : Java!ClassDeclaration (
         name <- c.name,
         fields <- c.ownedAttribute->collect(a | thisModule.Attribute2JavaField(a))
      )
}

lazy rule Attribute2JavaField {
   from
      a : UML!Property
   to
      f : Java!FieldDeclaration (
         name <- a.name,
         type <- a.type.name
      )
}
```

Résultat généré :

```
public class Person {
   private String name;
   private int age;
}
```

5.8. Exemple : Transformation UML → SQL

```
module UML2SQL;
create OUT : SQL from IN : UML;

rule Class2Table {
   from
      c : UML!Class
   to
      t : SQL!Table (
         name <- c.name,
         columns <- c.ownedAttribute->collect(a | thisModule.Attribute2Column(a))
      )
}

lazy rule Attribute2Column {
   from a : UML!Property
   to col : SQL!Column (
      name <- a.name,
      type <- if a.type.name = 'String' then 'VARCHAR(255)'
            else if a.type.name = 'Integer' then 'INT'
            else 'TEXT' endif endif
   )
}
```

Résultat :

```
CREATE TABLE Person (
   name VARCHAR(255),
   age INT
);
```

6.**Conclusion :**

La transformation de modèle est un élément clé du Développement Dirigé par les Modèles (MDE), permettant de convertir automatiquement un modèle en un autre tout en garantissant leur cohérence. Les langages dédiés comme **ATL** (pour les transformations déclaratives) et **QVT** (pour les transformations standardisées OMG) facilitent la mise en œuvre de ces transformations de manière formelle et reproductible. Maîtriser ces techniques permet d'automatiser l'évolution des modèles, d'améliorer la productivité et d'assurer la qualité et la réutilisabilité des systèmes développés.