

# *Ingénierie des Modèles*

## *Transformation de modèles*

Eric Cariou

Master Technologies de l'Internet 2<sup>ème</sup> année

*Université de Pau et des Pays de l'Adour  
UFR Sciences Pau – Département Informatique*

Eric.Cariou@univ-pau.fr

# *Transformations*

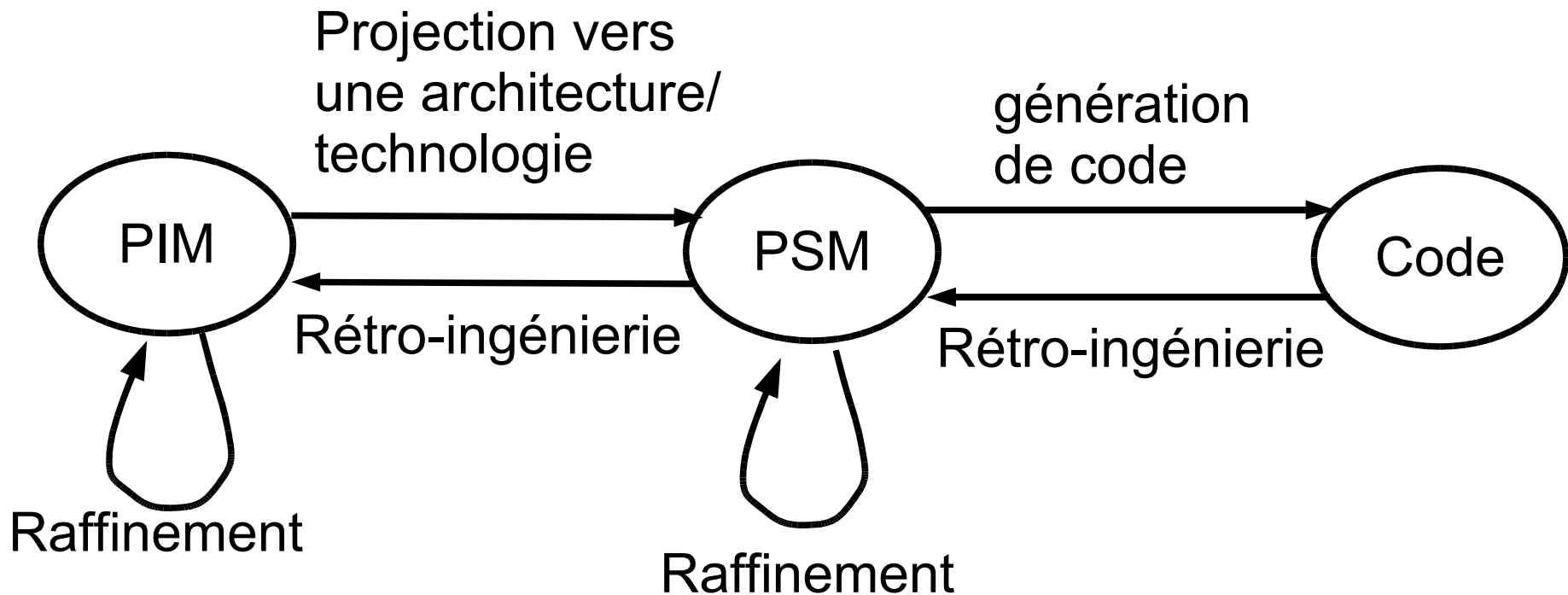
- ◆ Une transformation est une opération qui
  - ◆ Prend en entrée des modèles (source) et fournit en sortie des modèles (cibles)
    - ◆ Généralement un seul modèle source et un seul modèle cible
- ◆ Transformation endogène
  - ◆ Dans le même espace technologique
  - ◆ Les modèles source et cible sont conformes au même méta-modèle
    - ◆ Transformation d'un modèle UML en un autre modèle UML
- ◆ Transformation exogène
  - ◆ Entre 2 espaces technologique 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

# *Model Driven Architecture*

- ◆ Le MDA définit 2 principaux niveaux de modèles
  - ◆ PIM : Platform Independent Model
    - ◆ Modèle spécifiant une application indépendamment de la technologie de mise en oeuvre
    - ◆ Uniquement spécification de la partie métier d'une application
  - ◆ PSM : Platform Specific Model
    - ◆ Modèle spécifiant une application après projection sur une plate-forme technologique donnée

# Model Driven Architecture

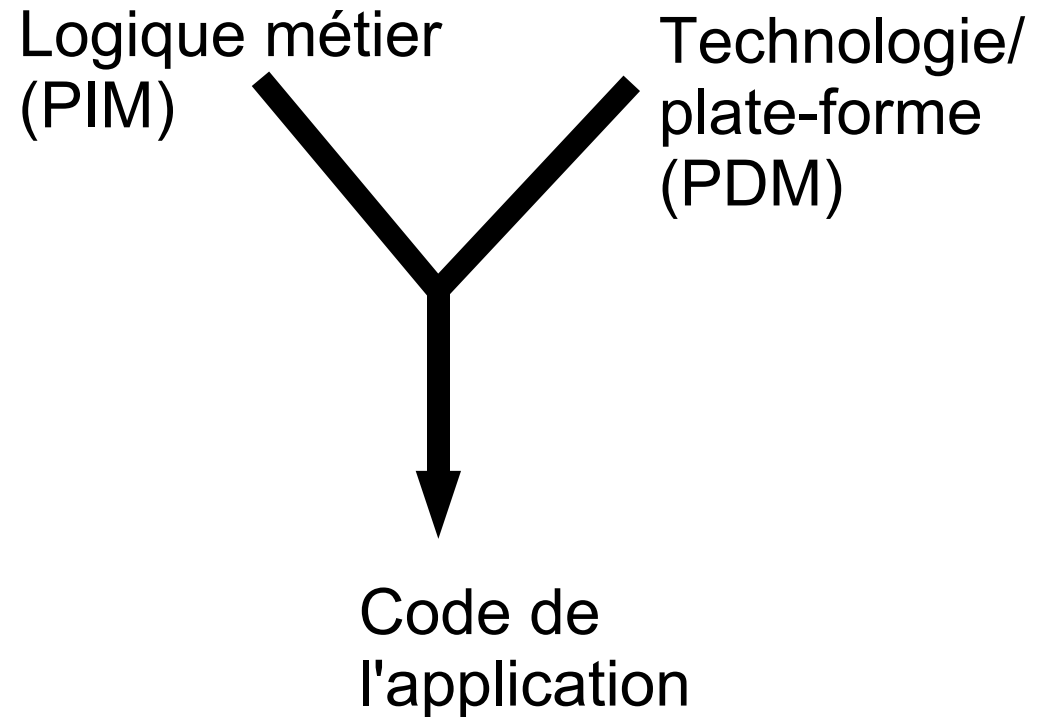
- ◆ Relation entre les niveaux de modèles



# *Model Driven Architecture*

- ◆ Cycle de développement d'un logiciel selon le MDA

- ◆ Cycle en Y
- ◆ Plus complexe en pratique
- ◆ Plutôt un cycle en épi



# *Niveaux de modèles*

- ◆ Les niveaux PIM et PSM du MDA peuvent se généraliser dans tout espace technologique
- ◆ Modèles de niveau abstrait : indépendamment d'une plateforme de mise en oeuvre, d'une technologie
- ◆ Modèles de niveau concret : par rapport à une plateforme, technologie de mise en oeuvre
- ◆ Nécessité de modéliser une plateforme de mise en oeuvre
  - ◆ PDM : Platform Deployment Model
  - ◆ Peu de choses sur ce sujet ...

# *Transformations en série*

- ◆ Réalisation d'une application
  - ◆ Processus basé sur une série de transformations de modèles
- ◆ Exemple
  1. Modèle de l'application au niveau abstrait, avec un modèle de composant abstrait : modèle PIM
  2. Projection du modèle vers un modèle de composant EJB : modèle PSM
  3. Raffinement de ce modèle pour ajouter des détails d'implémentation : modèle PSM
  4. Génération du code de l'application modélisée vers la plateforme EJB

# *Autre vision des transformations*

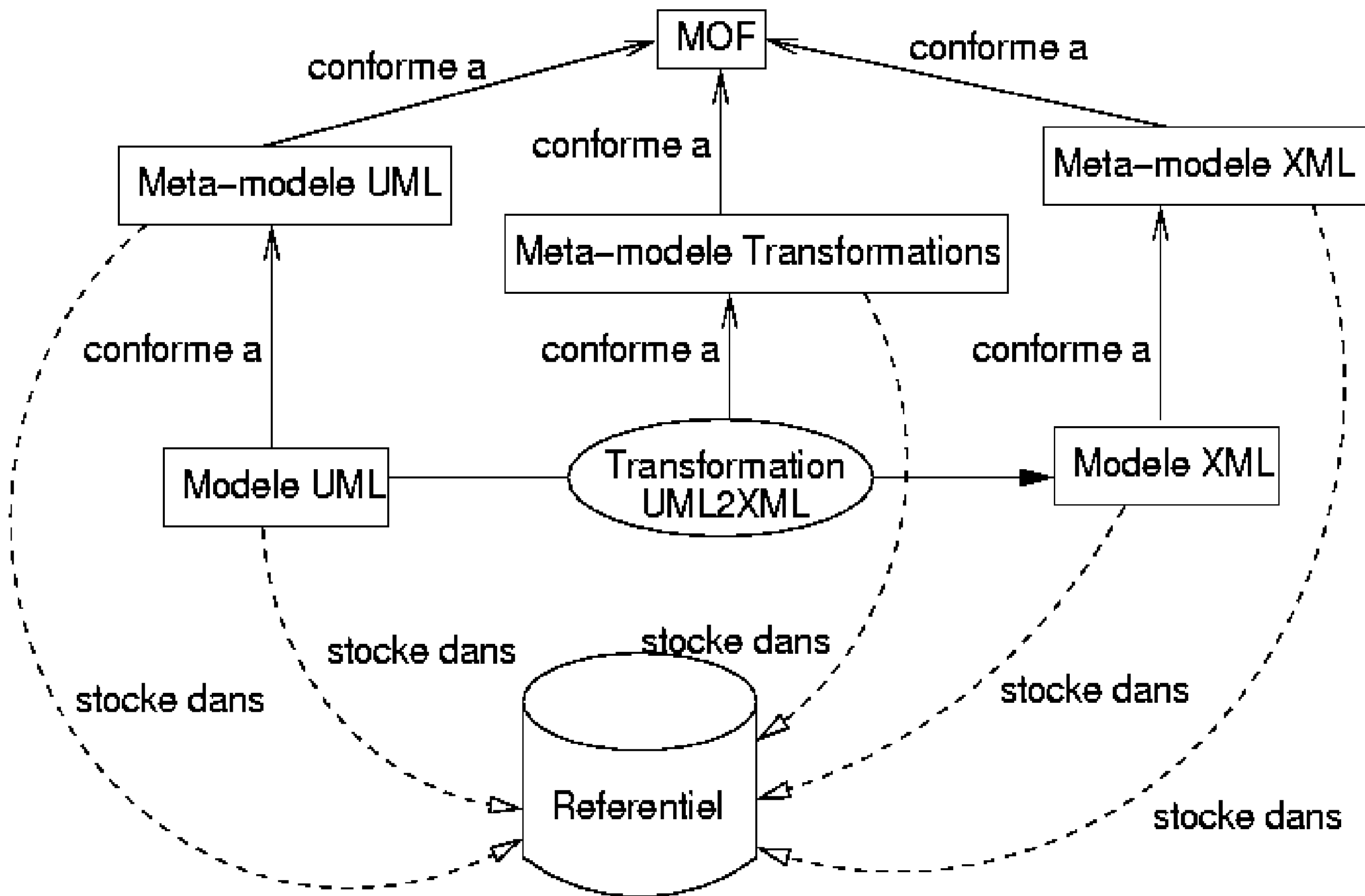
- ◆ Depuis longtemps on utilise des processus de développement automatisé et basé sur les transformations
  - ◆ Rien de totalement nouveau
    - ◆ Adaptation à un nouveau contexte
  - ◆ Exemple : compilation d'un programme C
    - ◆ Programme C : modèle abstrait
    - ◆ Transformation de ce programme sous une autre forme mais en restant à un niveau abstrait
      - ◆ Modélisation, représentation différente du programme C pour le manipuler : transformation en un modèle équivalent
      - ◆ Exemple : arbres décorés
    - ◆ Génération du code en langage machine
      - ◆ Avec optimisation pour une architecture de processeur donnée



# ***Outils pour réaliser des transformations***

- ◆ Outils de mise en oeuvre
  - ◆ Exécution de transformations de modèles
    - ◆ Nécessité d'un langage de transformation
    - ◆ Qui pourra être défini via un méta-modèle de transformation
  - ◆ Les modèles doivent être manipulés, créés et enregistrés
    - ◆ Via un *repository* (dépôt, référentiel)
    - ◆ Doit pouvoir représenter la structure des modèles
      - ◆ Via des méta-modèles qui devront aussi être manipulés via les outils
      - ◆ On les stockera également dans un repository
- ◆ Il existe de nombreux outils ou qui sont en cours de développement (industriels et académiques)
  - ◆ Notamment plusieurs moteurs/langages de transformation

# Modèles/méta-modèles/repository



# *Transformations : types d'outils*

- ◆ Langage de programmation « standard »
  - ◆ Ex : Java
  - ◆ Pas forcément adapté pour tout
  - ◆ Sauf si interfaces spécifiques
    - ◆ Ex : JMI (Java Metadata Interface) ou framework Eclipse/EMF
- ◆ Langage dédié d'un atelier de génie logiciel
  - ◆ Ex : J dans Objecteering
  - ◆ Souvent propriétaire et inutilisable en dehors de l'AGL
- ◆ Langage lié à un domaine/espace technologique
  - ◆ Ex: XSLT dans le domaine XML, AWK pour fichiers texte ...
- ◆ Langage/outil dédié à la transformation de modèles
  - ◆ Ex : standard QVT de l'OMG, langage ATL
- ◆ Atelier de méta-modélisation avec langage d'action
  - ◆ Ex : Kermeta

# *Transformations : types d'outils*

- ◆ 3 grandes familles de modèles et outils associés
  - ◆ Données sous forme de séquence
    - ◆ Ex : fichiers textes (AWK)
  - ◆ Données sous forme d'arbre
    - ◆ Ex : XML (XSLT)
  - ◆ Données sous forme de graphe
    - ◆ Ex : diagrammes UML
    - ◆ Outils
      - ◆ Transformateurs de graphes déjà existants
      - ◆ Nouveaux outils du MDE et des AGL (QVT, J, ATL, Kermeta ...)

# *Techniques de transformations*

- ◆ 3 grandes catégories de techniques de transformation
  - ◆ Approche déclarative
    - ◆ Recherche de certains patrons (d'éléments et de leurs relations) dans le modèle source
    - ◆ Chaque patron trouvé est remplacé dans le modèle cible par une nouvelle structure d'élément
    - ◆ Ecriture de la transformation « assez » simple mais ne permet pas toujours d'exprimer toutes les transformations facilement
  - ◆ Approche impérative
    - ◆ Proche des langages de programmation usuels
    - ◆ On parcourt le modèle source dans un certain ordre et on génère le modèle cible lors de ce parcours
    - ◆ Ecriture transformation peut être plus lourde mais permet de toutes les définir, notamment les cas algorithmiquement complexes
  - ◆ Approche hybride : à la fois déclarative et impérative
    - ◆ La plupart des approches déclaratives offrent de l'impératif en complément car plus adapté dans certains cas

# *Repository*

- ◆ Référentiel pour stocker modèles et méta-modèles
  - ◆ Les (méta)modèles sont stockés selon le formalisme de l'outil
    - ◆ XML par exemple pour les modèles
    - ◆ Et DTD/Schema XML pour les méta-modèles
  - ◆ Forme de stockage d'un modèle
    - ◆ Modèle est formé d'instances de méta-éléments, via la syntaxe abstraite
      - ◆ Stocke ces éléments et leurs relations
    - ◆ L'affichage du modèle via une notation graphique est faite par l'AGL
  - ◆ Les référentiels peuvent être notamment basés sur
    - ◆ XML
      - ◆ XMI : norme de l'OMG pour représentation modèle et méta-modèle
    - ◆ Base de données relationnelle
    - ◆ Codage direct dans un langage de programmation

# *Modèle en Java pour diag. états*

- ◆ 

```
public class StateMachine {  
    protected Vector states = new Vector();  
    protected Vector transitions = new Vector();  
  
    public void addState(State s) {  
        states.add(s);    }  
  
    public void addTransition(Transition t) {  
        transitions.add(t); }  
    ... }  
  
public class Transition {  
    protected State from, to;  
    protected String event;  
  
    public Transition(  
        State from, State to, String evt) {  
        this.from = from;  
        this.to = to;  
        event = evt; }  
}
```

# Modèle en Java pour diag. états

- ◆ 

```
public class State {  
    protected String name;  
    public State(String name) {  
        this.name = name; }  
}
```
- ◆ Définition d'un modèle : instances et liaisons de ces 3 classes

```
...  
StateMachine sm;  
State s1,s2;  
...  
sm = new StateMachine();  
s1 = new State("Ouvert");  
s2 = new State("Ferme");  
sm.addState(s1);  
sm.addState(s2);  
sm.addTransition(new Transition(State.Initial, s1, ""));  
sm.addTransition(new Transition(s1, s2, "fermer"));  
sm.addTransition(new Transition(s2, s1, "ouvrir"));
```



# *Transfo. basique de modèles en Java*

- ◆ Transformation de modèles
  - ◆ Ajout de méthodes pour lire les éléments du modèle
    - ◆ Ex. pour StateMachine : `getStates()`, `getTransitions()`
  - ◆ Parcours du modèle via ces méthodes
    - ◆ Approche impérative
  - ◆ Utilisation possible du patron Visiteur
  - ◆ Génération d'un nouveau modèle à partir de ce parcours
    - ◆ En utilisant éventuellement des méthodes des différentes classes pour gérer la transformation des éléments un par un

```
public class State {  
    ...  
    public String toXML() {  
        return new String("<state>\n\t<name>" + name  
            + "</name>\n</state>\n"); } ...
```

# *Transfo. basique de modèles en Java*

- ◆ Parcours de type « impératif » du graphe d'objet pour sérialisation en XML

```
String xml = "<statemachine>";

Iterator it = sm.getStates().iterator();
while(it.hasNext())
    xml += ((State)it.next()).toXML();

it = sm.getTransitions().iterator();
while(it.hasNext())
    xml += ((Transition)it.next()).toXML();

xml += "</statemachine>";

System.out.println(xml);
```

# *Transfo. basique de modèles en Java*

## ◆ Résultat sérialisation XML

```
<statemachine>
  <state>
    <name>Ouvert</name>
  </state>
  <state>
    <name>Ferme</name>
  </state>
  <transition>
    <from>_initial</from>
    <to>Ouvert</to>
    <event></event>
  </transition>
  <transition>
    <from>Ouvert</from>
    <to>Ferme</to>
    <event>fermer</event>
  </transition>
  <transition>
    <from>Ferme</from>
    <to>Ouvert</to>
    <event>ouvrir</event>
  </transition>
</statemachine>
```

# *Query/View/Transformation*

- ◆ Langage(s) de transformation et de manipulation de modèles normalisé par l'OMG
  - ◆ Query/View/Transformation ou QVT
  - ◆ Query : sélectionner des éléments sur un modèle
    - ◆ Le langage utilisé pour cela est OCL légèrement modifié et étendu
    - ◆ Avec une syntaxe différente et simplifiée
  - ◆ View : une vue est une sous-partie d'un modèle
    - ◆ Peut être définie via une query
    - ◆ Une vue est un modèle à part, avec éventuellement un méta-modèle restreint spécifique à cette vue
  - ◆ Transformation : transformation d'un modèle en un autre

# *Langages de transformation dans QVT*

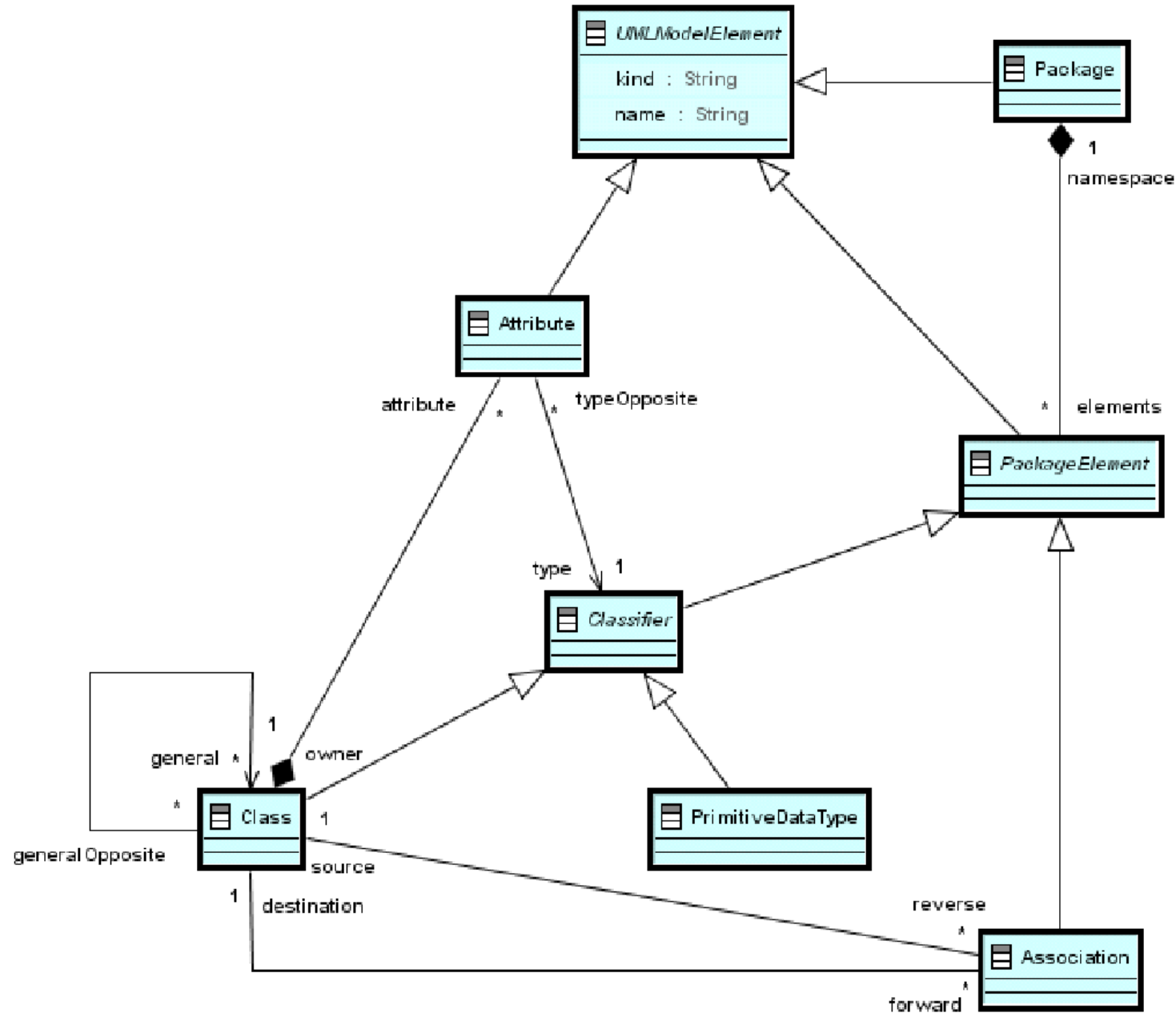
- ◆ 3 langages/2 modes pour définir des transformations
  - ◆ Mode déclaratif
    - ◆ *Relation*
      - ◆ Correspondances entre des ensembles/patrons d'éléments de 2 modèles
      - ◆ Langage de haut niveau
    - ◆ *Core*
      - ◆ Plus bas niveau, langage plus simple
      - ◆ Mais avec même pouvoir d'expression de transformations que *relation*
  - ◆ Mode impératif
    - ◆ *Mapping*
      - ◆ Impératif, mise en oeuvre/raffinement d'une relation
      - ◆ Ajout de primitives déclaratives inspirées en partie d'OCL
        - ◆ Manipulation d'ensembles d'objets avec primitives à effet de bords
- ◆ Plusieurs syntaxes pour écriture de transformation selon les langages
  - ◆ Syntaxe textuelle
  - ◆ Syntaxe graphique

# ***QVT : langage « relation »***

- ◆ Une transformation est définie par un ensemble de relations
- ◆ Une relation fait intervenir 2 domaines
  - ◆ Domaine = ensemble d'éléments d'un modèle
  - ◆ Relation = contraintes sur dépendances entre éléments de 2 domaines
    - ◆ Domaine du modèle source
    - ◆ Domaine du modèle cible
- ◆ Une relation peut s'appliquer
  - ◆ Par principe quand on applique la transformation
  - ◆ En dépendance d'application d'une autre relation

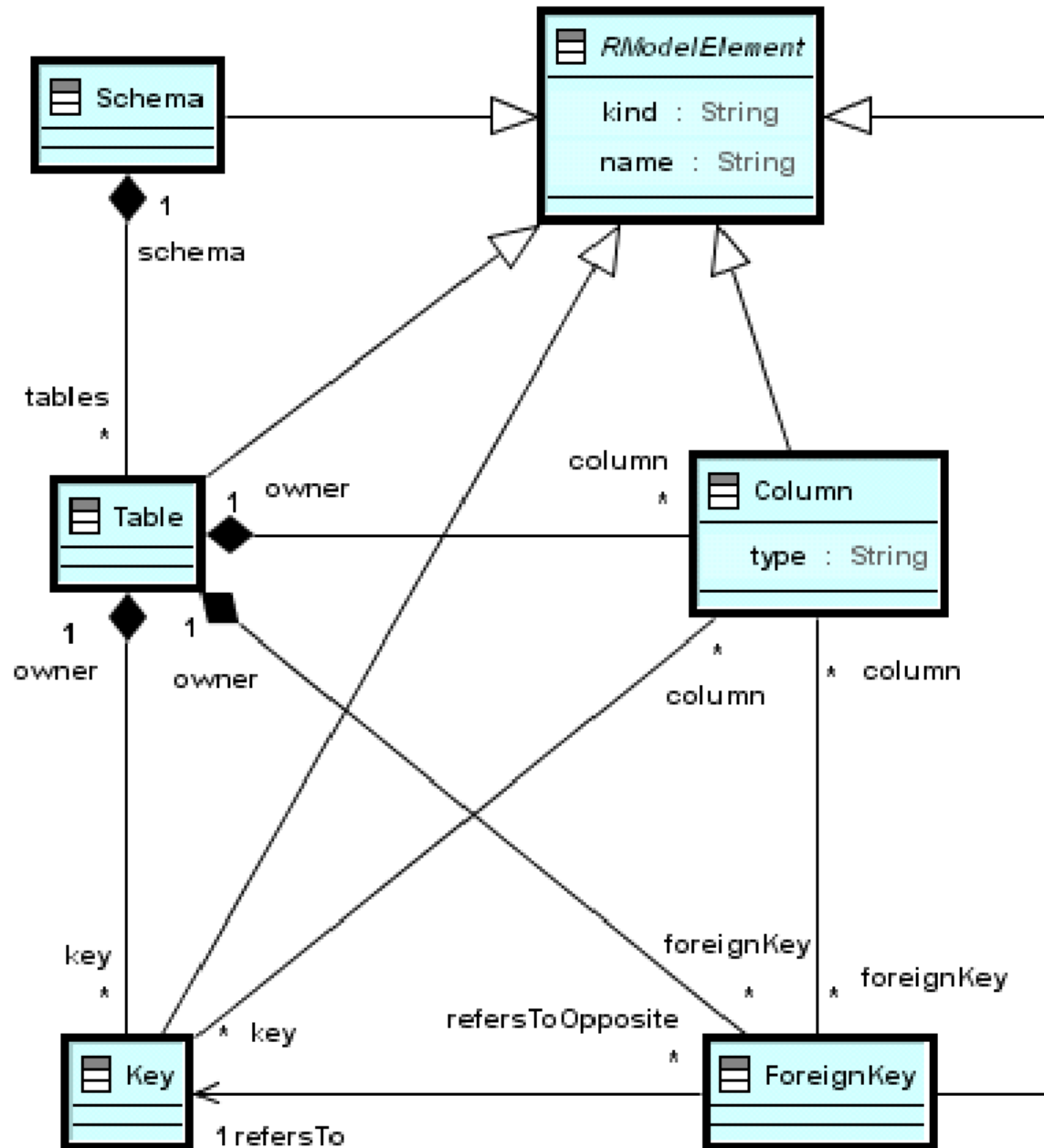
# Exemple : UML vers SGBDR

## ◆ Méta-modèle UML simplifié



# Exemple : UML vers SGBDR

## ◆ Méta-modèle relationnel simplifié





# Exemple : UML vers SGBDR

- ◆ But de la transformation
  - ◆ Modèle de données en UML vers équivalent schéma de données relationnel (et inversement)
  - ◆ transformation `umlToRdbms(uml : SimpleUML, rdbms : SimpleRDBMS) {`

```
    top relation PackageToSchema {...}
    top relation ClassToTable {...}
    relation AttributeToColumn {...}
    ...
}
```
- ◆ Etapes de la transformation
  - ◆ Chaque package UML correspond à un schéma de BDD, chaque classe persistante à une table, chaque attribut de classe à une colonne de table ...
- ◆ Relations marquées avec « top »
  - ◆ S'appliquent par principe
  - ◆ Les autres s'appliquent si dépendantes d'autres relations

# *Exemple : UML vers SGBDR*

- ◆ top relation PackageToSchema  
{  
    domain uml p:Package {name=pn}  
    domain rdbms s:Schema {name=pn}  
}
- ◆ Pour chaque package UML, on a un schéma de données portant le même nom
  - ◆ Les attributs `name` correspondent à la même variable `pn`

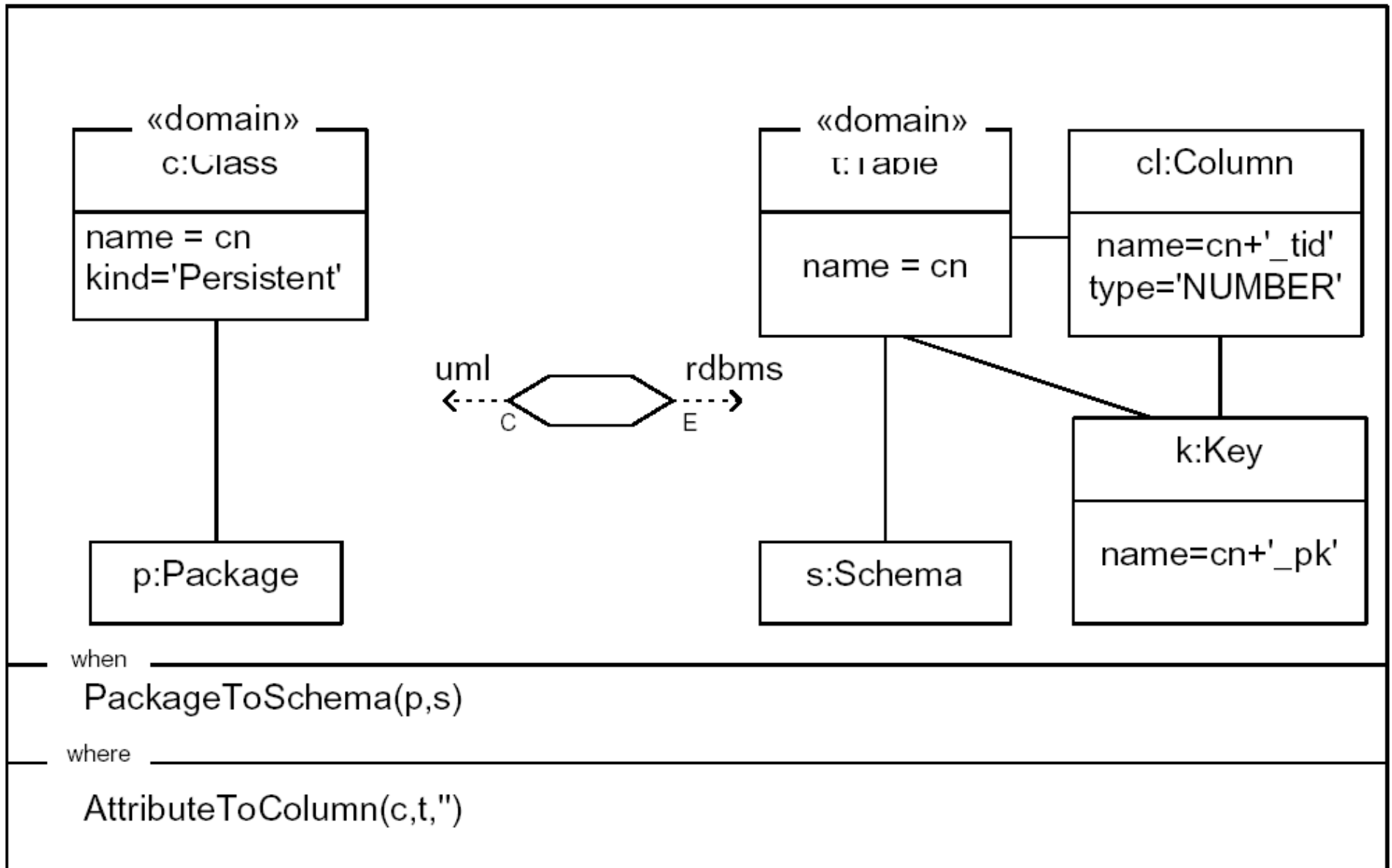
# Exemple : UML vers SGBDR

```
top relation ClassToTable {
  domain uml c:Class {
    namespace = p:Package {},
    kind='Persistent',
    name=cn
  }
  domain rdbms t:Table {
    schema = s:Schema {},
    name=cn,
    column = cl:Column {
      name=cn+'_tid',
      type='NUMBER'},
    primaryKey = k:PrimaryKey {
      name=cn+'_pk',
      column=cl}
  }
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t);
  }
}
```

- ◆ Pour chaque classe persistante
- ◆ On a une table avec
  - ◆ Le même nom
  - ◆ Une colonne pour l'identifiant avec nom formé à partir du nom de la classe
  - ◆ Une clé primaire avec nom formé à partir du nom de la classe
- ◆ Dépendances avec autres relations
  - ◆ « ClasseToTable » est appliquée quand on applique « PackageToSchema »
  - ◆ Et il faut appliquer aussi « AttributeToColumn » pour la classe et la table

# Exemple : UML vers SGBDR

- ◆ Relation « ClassToTable », syntaxe graphique



# *Types de relations entre modèles*

- ◆ Transformations définies par des relations
  - ◆ Correspondances/dépendances entre 2 modèles dans un sens comme dans l'autre
    - ◆ Spécification est bi-directionnelle par défaut
  - ◆ A l'exécution, on choisit une direction de transformation
    - ◆ Exécution est mono-directionnelle
  - ◆ Modèle cible peut exister ou pas à l'exécution
    - ◆ Sera alors complété, modifié ou créé selon les cas
- ◆ Possibilité de spécialiser une transformation/relation
  - ◆ Juste vérifier si le modèle cible est cohérent rapport au modèle source (checkonly)
  - ◆ Imposer que le modèle cible soit cohérent rapport au modèle source (enforced)

# *Types de relations entre modèles*

## ◆ Exemple avec transformation UML/SGDBR

```
◆ relation PackageToSchema {  
    checkonly domain uml p:Package {name=pn}  
    enforce domain rdbms s:Schema {name=pn}  
}
```

## ◆ Exécution dans le sens UML vers SGBDR

◆ Source = uml, cible = rdbms

◆ Le modèle cible comportera strictement un schéma pour chaque package du modèle source

◆ Création des schémas manquants

◆ Suppression des schémas existants mais ne correspondant pas

## ◆ Exécution dans le sens SGBDR vers UML

◆ Source = rdbms, cible = uml

◆ Vérifie seulement, en précisant les erreurs le cas échéant, que chaque schéma du modèle source correspond à un package du modèle cible (aucune modification du modèle cible)

# *Exécution/spécification*

- ◆ Problématiques d'exécution de transformations sont fondamentales
- ◆ Mais doit aussi être capable de spécifier des transformations
- ◆ Trois buts principaux
  - ◆ Spécification et documentation
    - ◆ Préciser ce que fait la transformation, dans quelles conditions on peut l'utiliser
  - ◆ Vérification, validation et test
    - ◆ S'assurer qu'un modèle peut bien être transformé ou bien est le résultat valide d'une transformation
  - ◆ Validation de l'enchaînement de transformation
    - ◆ Enchaînement de transformations est à la base de tout processus de développement basé sur le MDE

# Spécification de transformation

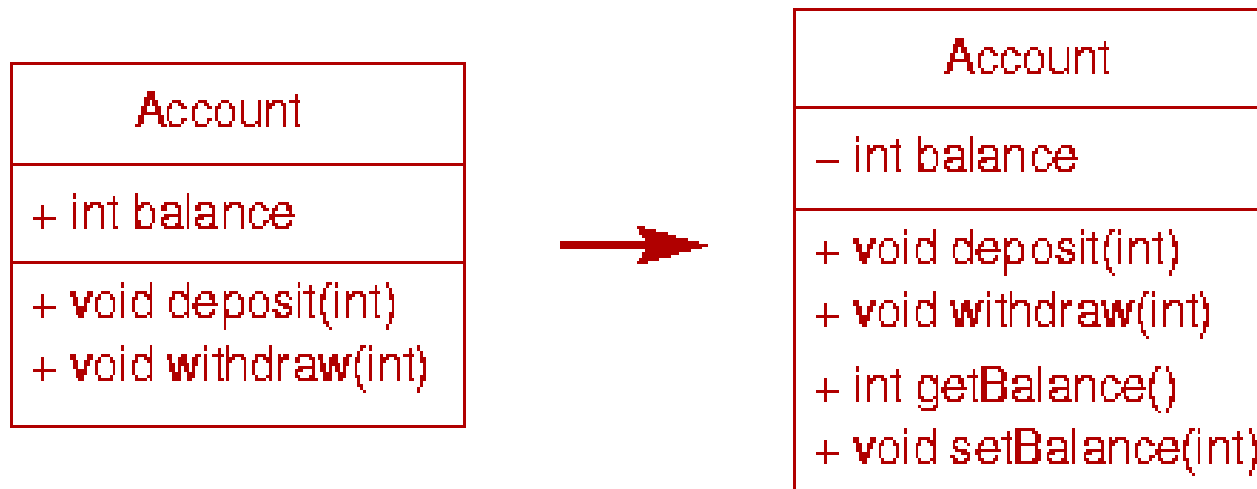
- ◆ Mes travaux de recherche
  - ◆ Spécification de transformations via des contrats de transformations
    - ◆ Contrats : ensemble de contraintes sur un élément logiciel que s'engage à respecter l'élément (et l'utilisateur de l'élément)
    - ◆ Spécifie ce que ce fait l'élément sans détailler comment il fait (ce qui correspond au code)
    - ◆ Exemple du compte bancaire du cours sur OCL
      - ◆ **context** Compte : débiter(somme : int)  
**pre:** somme > 0  
**post:** solde = solde@pre - somme
      - ◆ L'opération débiter s'engage à respecter la post-condition si l'élément appelant l'opération respecte la pré-condition
  - ◆ Utilisation du langage OCL pour définir ces contrats



# Exemple basique de transformation

## ◆ Privatisation des attributs

- ◆ Chaque attribut d'une classe devient privé et se voit associer une paire d'accesseurs



## ◆ Contrat de la transformation

- ◆ Chaque attribut du modèle cible possède un « getter » et un « setter »
- ◆ Les attributs ont tous été conservés pendant la transfo : pas d'ajout, de suppression ou modification

# *Contrats de transformations*

- ◆ Contrat de transformations de modèles = 3 ensembles de contraintes
  - ◆ Contraintes que doit respecter un modèle source
    - ◆ De manière générale, indépendamment du contenu du modèle
  - ◆ Contraintes que doit respecter un modèle cible
    - ◆ De manière générale, indépendamment du contenu du modèle
  - ◆ Contraintes sur les relations des éléments du modèle source vers le modèle cible
    - ◆ Contraintes de passage (des éléments) du modèle source au cible
- ◆ Deux premiers ensembles de contraintes
  - ◆ Contraintes sur les modèles source et cible
    - ◆ S'expriment au niveau de leur méta-modèle
    - ◆ Ensemble de contraintes OCL standard

# *Contrats de transformations*

- ◆ Contraintes sur relations entre source et cible
  - ◆ Idée intuitive : spécifier en OCL via des pré et post-conditions l'opération de transformation définie au niveau du méta-modèle
    - ◆ Pré-condition = état avant la transfo : modèle source
    - ◆ Post-condition = état après la transfo : modèle cible
    - ◆ Dans post-condition : construction @pre permet de référencer les éléments avant la transformation
      - ◆ Peut donc référencer à la fois des éléments du source et du cible et définir des contraintes entre ces éléments
  - ◆ Problème de cette approche
    - ◆ Ne peut vérifier les contraintes que lors de l'exécution de la transformation
    - ◆ Doit avoir un outil capable de faire les 2 en même temps
    - ◆ Impossible à appliquer avec des modèles obtenus de manière quelconque ou modifiés/transformés à la main

# Contrats de transformations

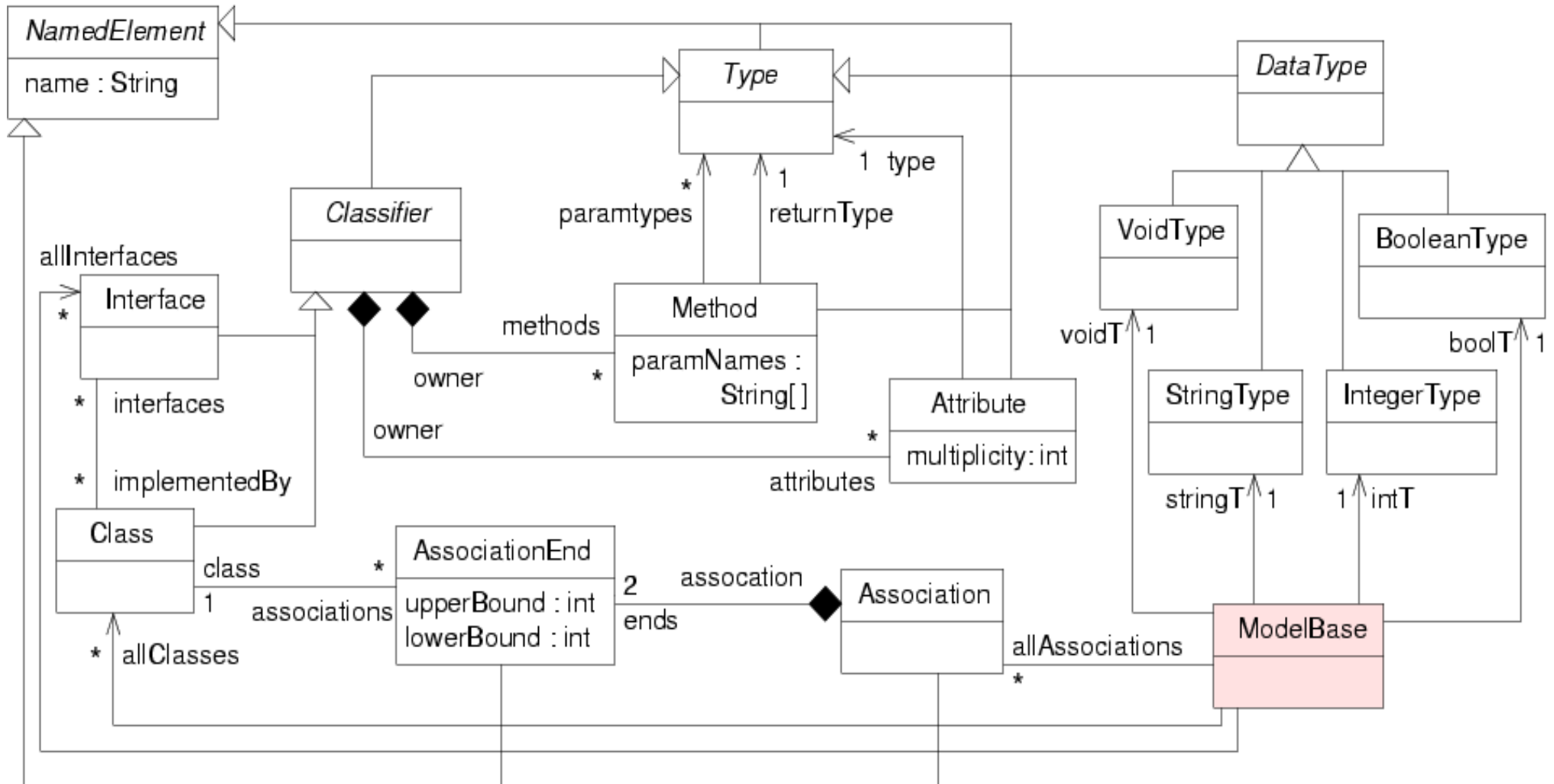
- ◆ Contraintes sur relations entre source et cible
  - ◆ Autre technique : utiliser un ensemble d'invariants s'appliquant sur les 2 modèles à la fois
  - ◆ Limite d'OCL ici : le contexte simple d'expression des contraintes
    - ◆ Ne permet pas de manipuler 2 modèles en même temps
  - ◆ Solution
    - ◆ Concaténer les 2 modèles en un seul et définir les invariants sur le modèle concaténé
  - ◆ Mise en oeuvre pratique (contexte endogène uniquement ici)
    - ◆ Le méta-modèle est modifié pour y ajouter un élément nommé *ModelReference* et contenant une chaîne *modelName*
    - ◆ Tous les éléments du méta-modèle héritent de cet élément
    - ◆ On concatène les 2 modèles en « taggant » chaque élément avec « source » ou « target » selon le modèle auquel il appartient
    - ◆ On peut ensuite écrire l'ensemble des contraintes en pouvant référencer facilement les éléments des modèles source et cible

# *Exemple de privatisation d'attributs*

# *Privatisation d'attributs*

- ◆ Pour un diagramme de classe, passer les attributs d'une classe en privé et leur associer un « getter » et un « setter »
- ◆ On va traiter en entier cet exemple
  - ◆ Transformation en Kermeta, style impératif
  - ◆ Transformation en ATL, style déclaratif
  - ◆ Contrat de transformation associé
- ◆ Utilisation d'un méta-modèle simplifié de diagramme de classe
  - ◆ Méta-modèle UML trop complexe pour cet exemple
- ◆ Sources disponibles en ligne
  - ◆ <http://web.univ-pau.fr/~ecariou/contracts/>

# MM simplifié de diagramme de classe



- ◆ Note: les visibilités ne sont pas définies, on ne les gèrera pas pendant la transformation

# MM simplifié de diagramme de classe

## ◆ Contraintes OCL pour compléter le méta-modèle

### ◆ Unicité des noms de type

```
context Type inv uniqueTypeNames:  
Type.allInstances() -> forall (t1, t2 |  
    t1 <> t2 implies t1.name <> t2.name)
```

### ◆ Une interface n'a pas d'attributs

```
context Interface inv noAttributesInInterface:  
attributes -> isEmpty()
```

### ◆ Une méthode à une liste de noms de paramètres et une liste de types de paramètres de même taille

```
context Method inv sameSizeParamsAndTypes:  
paramNames -> size() = paramTypes -> size()
```

### ◆ ...



# Règles générales de la transformation

- ◆ Pour un attribut *att* de type *type*, la forme des accesseurs est
  - ◆ Getter : *type getAtt()*
  - ◆ Setter : *void setAtt(type xxx)*
    - ◆ Le nom de l'attribut est quelconque
- ◆ Règles de transformations
  - ◆ Pour chaque attribut de chaque classe
    - ◆ On ajoute, s'ils n'existaient pas déjà, un setter et un getter dans la classe qui possède l'attribut
  - ◆ Doit donc prévoir des fonctions de vérification de la présence d'un getter ou d'un setter

# Transformation en Kermeta

## ◆ Vérification de la présence d'un getter

```
◆ operation hasGetter(att : Attribute) : Boolean is do  
  result := att.owner.methods.exists { m |  
    if (m.name.size <=3)  
    then  
      false  
    else  
      m.name.substring(0,3).equals("get") and  
      m.name.substring(3,4).equals(att.name.substring(0,1).toUpperCase) and  
      m.name.substring(4,m.name.size).equals(  
        att.name.substring(1,att.name.size)) and  
  
      m.returnType == att.type and  
      m.paramNames.empty() and  
      m.paramTypes.empty()  
    end  
  }  
end
```

## ◆ Pour un attribut, on récupère les méthodes de sa classe (att.owner.methods) et on vérifie qu'il existe un getter

- ◆ Nom « getAtt » (vérifie taille chaîne > 3 sinon les substring plantent)
- ◆ Même type de retour que l'attribut
- ◆ Liste de paramètres vide

# Transformation en Kermeta

## ◆ Vérification de la présence d'un setter

```
◆ operation hasSetter(att : Attribute) : Boolean is do  
  result := att.owner.methods.exists { m |  
    if (m.name.size <=3)  
    then  
      false  
    else  
      m.name.substring(0,3).equals("set") and  
      m.name.substring(3,4).equals(att.name.substring(0,1).toUpperCase) and  
      m.name.substring(4,m.name.size).equals(  
        att.name.substring(1,att.name.size)) and  
      m.returnType.name.equals("void") and  
      m.paramNames.size() == 1 and  
      m.paramTypes.includes(att.type)  
    end  
  }  
end
```

## ◆ Vérification d'un setter

- ◆ Nom « setAtt »
- ◆ Un seul paramètre, du même type que l'attribut
- ◆ Type de retour est « void »

# Transformation en Kermeta

```
◆ operation addAccessors(base : ModelBase) : ModelBase is do
  base.allClasses.each { cl |
    cl.attributes.each { att |
      var met : Method

      if not(hasGetter(att)) then
        met := Method.new
        met.name := "get"+ att.name.substring(0,1).toUpperCase +
                               att.name.substring(1,att.name.size)

        met.returnType := att.type
        met.owner := att.owner
        cl.methods.add(met)
      end

      if not(hasSetter(att)) then
        met := Method.new
        met.name := "set" + att.name.substring(0,1).toUpperCase +
                               att.name.substring(1,att.name.size)

        met.returnType := base.voidT
        met.paramNames.add("value")
        met.paramTypes.add(att.type)
        met.owner := att.owner
        cl.methods.add(met)
      end
    }
  }
  result := base
end
```

# *Transformation en Kermeta*

- ◆ Réalisation de la transformation
  - ◆ A partir de la base du modèle, on parcourt l'ensemble des classes et pour chacun de leur attribut, s'il ne possède pas un getter ou un setter
    - ◆ Crée la méthode en instanciant le méta-élément Method
    - ◆ Positionne son nom en « setAtt » ou « getAtt »
    - ◆ Positionne son type de retour
      - ◆ Référence sur le type void (voidT) ou le type de l'attribut
    - ◆ Positionne les listes des paramètres pour un getter à partir du type de l'attribut et le nom « value »
      - ◆ Sinon reste vide par défaut (cas d'un setter)
    - ◆ Positionne les associations entre la méthode créée et la classe qui possède l'attribut
      - ◆ Cette classe doit aussi posséder la méthode créée
  - ◆ Retourne ensuite la base du modèle modifiée

# Transformation en ATL

- ◆ Pour vérification des présences des getter et setter, utilise des helpers écrits en OCL (légère diff de syntaxe)

- ◆ **helper context** CDSOURCE!Attribute **def:** hasGetter() : Boolean =  
self.owner.methods -> exists ( m |  
 m.name = 'get' + self.name.firstToUpper() **and**  
 m.paramNames -> isEmpty() **and**  
 m.paramTypes -> isEmpty() **and**  
 m.returnType = self.type  
);

- ◆ **helper context** CDSOURCE!Attribute **def:** hasSetter() : Boolean =  
self.owner.methods -> exists ( m |  
 m.name = 'set' + self.name.firstToUpper() **and**  
 m.paramNames -> size() = 1 **and**  
 m.paramTypes -> includes( self.type ) **and**  
 m.returnType = thisModule.voidType  
);

- ◆ Fonction pour gérer le premier caractère d'une chaîne en majuscule

```
helper context String def: firstToUpper() : String =  
self.substring(1, 1).toUpperCase() + self.substring(2, self.size());
```

- ◆ Référence sur le type void

```
helper def: voidType : CDSOURCE!VoidType =  
CDSOURCE!VoidType.allInstances() -> asSequence() -> first();
```

# *Transformation en ATL*

- ◆ Transformation ATL en mode raffinement
  - ◆ Duplication d'éléments référencés, directement ou indirectement, dans le modèle cible sans modification
    - ◆ Et sans règles explicites pour les créer coté cible
  - ◆ Cinq règles de transformation
    - ◆ Création d'une base de modèle identique
      - ◆ Le raffinement fait que toutes les classes, interfaces et associations de la base seront alors automatiquement dupliquées
    - ◆ Pour chaque attribut, on a des règles qui créent l'attribut coté cible et les éventuels méthodes accesseurs manquantes
    - ◆ Selon qu'il possède déjà un getter ou un setter, 4 cas différents
      - ◆ Possède un gettter et un setter (règle « hasAll »)
      - ◆ Possède un setter mais pas un getter (règle « hasSetter »)
      - ◆ Possède un getter mais pas un setter (règle « hasGetter »)
      - ◆ Ne possède ni l'un ni l'autre (règle « hasNothing »)

# Transformation en ATL

```
◆ module AddAccessorRefining;  
create cible : CDTarget refining source : CDSource;  
  
... liste des helpers ...  
  
rule duplicateModelBase {  
from  
  sourceBase : CDSource!ModelBase  
to  
  cibleBase : CDTarget!ModelBase (  
    allClasses <- sourceBase.allClasses,  
    allInterfaces <- sourceBase.allInterfaces,  
    allAssociations <- sourceBase.allAssociations,  
    voidT <- sourceBase.voidT,  
    intT <- sourceBase.intT,  
    stringT <- sourceBase.stringT,  
    boolT <- sourceBase.boolT )  
}  
  
rule attributeHasAll {  
from  
  attSource : CDSource!Attribute (  
    attSource.hasSetter() and attSource.hasGetter())  
to  
  attTarget : CDTarget!Attribute (  
    name <- attSource.name,  
    owner <- attSource.owner,  
    type <- attSource.type,  
    multiplicity <- attSource.multiplicity )  
}
```



# Transformation en ATL

```
◆ rule attributeHasSetter {
  from
    attSource : CDSource!Attribute (
      attSource.hasSetter() and not(attSource.hasGetter())
    )
  to
    attTarget : CDTarget!Attribute (
      name <- attSource.name,
      owner <- attSource.owner,
      type <- attSource.type,
      multiplicity <- attSource.multiplicity
    ),
    getter : CDTarget!Method (
      name <- 'get' + attSource.name.firstToUpper(),
      returnType <- attTarget.type,
      owner <- attTarget.owner
    )
  do {
    attTarget.owner.methods <- attTarget.owner.methods -> including(getter);
  }
}
```

◆ Pour un attribut du source, 2 éléments sont créés coté cible

◆ L'attribut équivalent

◆ La méthode getter associée, qui est ajoutée dans la liste des méthodes de la classe de l'attribut via la section impérative « do »

# Transformation en ATL

```
◆ rule attributeHasGetter {
  from
    attSource : CDSource!Attribute (
      not(attSource.hasSetter()) and attSource.hasGetter()
    )
  to
    attTarget : CDTarget!Attribute (
      name <- attSource.name,
      owner <- attSource.owner,
      type <- attSource.type,
      multiplicity <- attSource.multiplicity
    ),
    setter : CDTarget!Method (
      name <- 'set' + attSource.name.firstToUpper(),
      returnType <- thisModule.voidType,
      owner <- attTarget.owner,
      paramNames <- Set { 'value' },
      paramTypes <- Set { attTarget.type }
    )
  do {
    attTarget.owner.methods <-attTarget.owner.methods -> including(setter);
  }
}
```

# Transformation en ATL

```
◆ rule attributeHasNothing {
  from
    attSource : CDSource!Attribute (
      not(attSource.hasSetter()) and not(attSource.hasGetter())
    )
  to
    attTarget : CDTarget!Attribute (
      name <- attSource.name,
      owner <- attSource.owner,
      type <- attSource.type,
      multiplicity <- attSource.multiplicity
    ),
    setter : CDTarget!Method (
      name <- 'set' + attSource.name.firstToUpper(),
      returnType <- thisModule.voidType,
      owner <- attTarget.owner,
      paramNames <- Set { 'value' },
      paramTypes <- Set { attTarget.type }
    ),
    getter : CDTarget!Method (
      name <- 'get' + attSource.name.firstToUpper(),
      returnType <- attTarget.type,
      owner <- attTarget.owner
    )
  do {
    attTarget.owner.methods <- attTarget.owner.methods -> including(getter);
    attTarget.owner.methods <- attTarget.owner.methods -> including(setter);
  }
}
```

# *Contrat de transformation*

- ◆ Trois ensembles de contraintes pour la privatisation des attributs
  - ◆ Sur modèles source : aucune
  - ◆ Sur modèles cible : chaque attribut possède un getter et un setter
  - ◆ Sur les relations entre les modèles source et cible : la liste des attributs ne change pas
    - ◆ Pas de création, de modification (nom, type, ...) ou de suppression d'attributs entre le source et le cible
  - ◆ Peut écrire un ensemble unique de contraintes qui vérifient les 2 derniers ensembles sur le modèle concaténé
    - ◆ Rappel : chaque élément est taggé par « source » ou « target » selon qu'il appartient au modèle source ou cible

# *Contrat de transformation*

- ◆ Pour vérifier la non modification d'un attribut
  - ◆ Doit commencer par récupérer l'attribut équivalent dans l'autre modèle
  - ◆ Il a le même nom, le même type et le même propriétaire
    - ◆ Même propriétaire = appartient à la même classe dans l'autre modèle
      - ◆ La classe du même nom puisque les noms de types sont uniques
    - ◆ Même type = même nom de type
  - ◆ Puis on vérifie que les 2 attributs sont identiques
    - ◆ Qu'ils ont le même nom, le même type et le même propriétaire
    - ◆ Il suffit donc de trouver l'attribut équivalent : s'il existe c'est bon
  - ◆ On définit pour cela une fonction de correspondance d'attribut
    - ◆ Générée automatiquement par un outil

# *Contrat de transformation*

- ◆ Non modification d'un attribut
  - ◆ Pour tout attribut, on vérifie qu'il lui existe un attribut équivalent dans l'autre modèle
    - ◆ Si attribut du source, cherche dans le cible
    - ◆ Si attribut du cible, cherche dans le source
    - ◆ La vérification dans les 2 sens assure qu'on ne rajoute ni ne supprime aucun attribut
- ◆ Vérification de la présence d'un getter ou d'un setter
  - ◆ A faire pour un attribut du coté cible
  - ◆ Réutilise directement les fonctions définies pour la transformation ATL

# Contrat de transformation

```
◆ context Attribute def:  
    MappingAttribute (attribute : Attribute) : Boolean =  
    self.name = attribute.name and  
    self.multiplicity = attribute.multiplicity and  
    self.owner.name = attribute.owner.name and  
    self.type.name = attribute.type.name
```

```
context Attribute def:  
    existsMappingAttribute (modelKind : String) : Boolean =  
    Attribute.allInstances () -> exists (att |  
        att.modelName = modelKind and  
        att.MappingAttribute (self)  
    )
```

```
context Attribute inv accessorContract:  
    if (self.modelName = 'source')  
        self.existsMappingAttribute ('target')  
    else  
        self.existsMappingAttribute ('source') and  
        self.hasGetter () and  
        self.hasSetter ()  
    endif
```