Transformation de Modèles

Principes, Standards et Exemples

Benoît Combemale[†], Xavier Crégut[‡], Marc Pantel[‡]

[†] IRISA CNRS Laboratory, *University of Rennes 1*
benoit.combemale@irisa.fr

[‡] IRIT CNRS Laboratory, *University of Toulouse*
{xavier.cregut, marc.pantel}@enseeiht.fr

dernière mise à jour le 19 octobre 2009

Support disponible à l'adresse (*teaching part*) :
http://perso.univ-rennes1.fr/benoit.combemale/

Sommaire

1. Contexte / Motivation
2. La transformation de modèle
3. Le standard QVT
4. Exemples de transformations endogènes
5. Transformation de modèle avec Kermeta
6. Conclusion

Sommaire

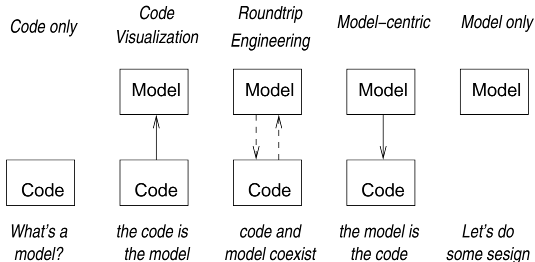
1. Contexte / Motivation
   - Ingénierie dirigée par les modèles
   - Langages dédiés (DSL)
   - Exemples de transformation
2. La transformation de modèle
3. Le standard QVT
4. Exemples de transformations endogènes
5. Transformation de modèle avec Kermeta
6. Conclusion

Importance des modèles

- On utilise des modèles pour mieux comprendre un système.

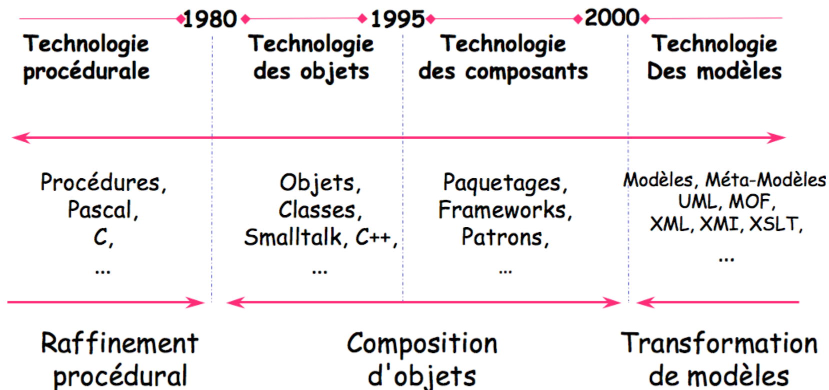
  Pour un observateur A, M est un modèle de l'objet O, si M aide A à répondre aux questions qu'il se pose sur O. (Minsky)

- Un modèle est une simplification, une abstraction du système.
- Exemple : une carte routière, une partition de musique, un diagramme UML



Évolution des technologies

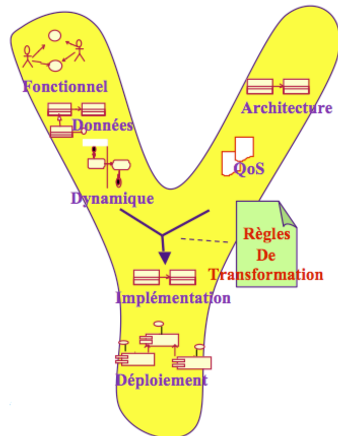
L'Ingénierie Dirigée par les Modèles : de l'OMA vers le MDA



Proposition de l'OMG

Un processus dirigé par les modèles

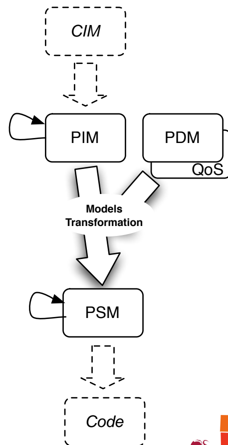
« OMG is in the ideal position to provide the model-based standards that are necessary to extend integration beyond the middleware approach ? Now is the time to put this plan into effect. Now is the time for the Model Driven Architecture. »
Richard Soley and the OMG staff,
MDA Whitepaper Draft 3.2
November 27, 2000



Le modèle au centre du développement

Objectif : Tenter une interopérabilité par les modèles

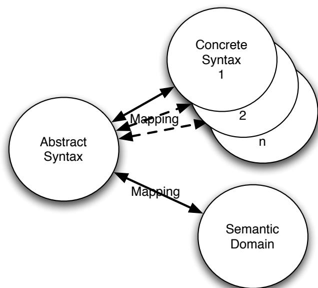
- Partir de CIM (Computation Independant Model) :
  - aucune considération informatique n'apparaît
- Faire des modèles indépendants des plateformes (PIM)
  - rattaché à un paradigme informatique ;
  - indépendant d'une plateforme de réalisation précise
- Spécifier des règles de passage (transformation) vers ...
- ... les modèles dépendants des plateformes (PSM)
  - version modélisée du code ;
  - souvent plus facile à lire.
- Automatiser au mieux la production vers le code
  PIM → PSM → Code

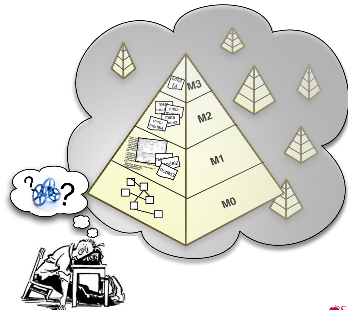


Définition d'un DSL

Un DSL c'est :

- une syntaxe abstraite (concepts et relations)
- des syntaxes concrètes (graphiques et textuelles)
- des domaines sémantiques





Exemples de transformation

- PIM ⟶ PIM :
  - privatiser les attributs (principe de protection en écriture et principe de l'accès uniforme)
  - refactoring : réorganiser le code :
    - utilisation d'un patron de conception (état, composite, visiteur, observateur...)
  - génération semi-automatique grâce à des marqueurs :
    - classe marquée active ⇒ hérite de Thread...
    - persistance
- PIM ⟶ PSM
  - motif de passage d'une classe UML à une classe Java
  - prise en compte de l'hériage multiple (C++, Eiffel , Java)
  - prise en compte des technologies disponibles
- PSM ⟶ PIM :
  - rétroconception, abstraction, analyse statique...

CONCLUSION : L'Ingénierie Dirigée par les Modèles repose sur la méta-modélisation ET sur les transformations.

Sommaire

1 Contexte / Motivation

2 La transformation de modèle
- Taxonomie des transformations
- Génération 1 : Transformation de structures séquentielles d'enregistrements
- Génération 2 : Transformation d'arbres
- Génération 3 : Transformation de graphes
- Types de transformation

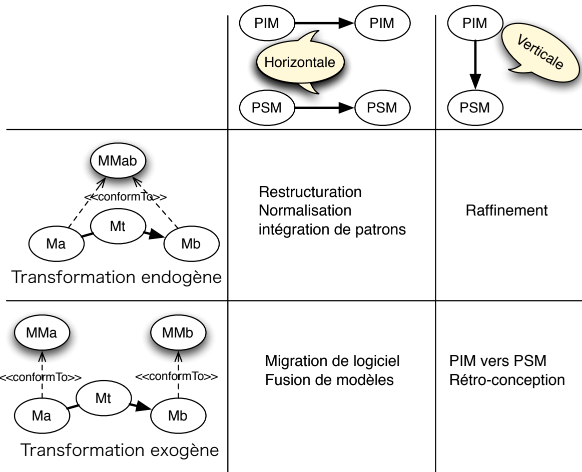
3 Le standard QVT

4 Exemples de transformations endogènes

5 Transformation de modèle avec Kermeta

6 Conclusion

Classes de transformation



Propriétés d'une transformation

- Transformations endogènes, exogènes
- Transformations unidirectionnelles ou bidirectionnelles
- Réversibilité
- Traçabilité
- Incrémentalité
- Réutilisabilité
- ...

Génération 1 : Transformation de structures séquentielles d'enregistrements

Le système Unix :

- une commande peut être constituée d'autres commandes
- les commandes peuvent être chaînées (pipe)
- structures de contrôle possibles sur l'exécution des commandes
- possibilité de tester le résultat d'une commande

Exemple : AWK, un générateur de rapports par Aho, Kernighan etWeinberg

- recherche de motifs
- exécution d'actions

⇒ style déclaratif

```
BEGIN {
  profondeur = 0;
  nb_blocs = 0;
  max_p = 0;
}
/^6.*/ {
  profondeur++;
  nb_blocs++;
  if (profondeur > max_p)
    max_p = profondeur;
}
/^-6.*/ {
  if (profondeur == 0)
    print "ERREUR";
  else
    profondeur--;
}
END {
  print "nb_blocs = " nb_blocs;
  print "max_p = " max_p;
}
```

Génération 2 : Transformations d'arbres

Principe : Exploiter des dommaines structurés en arbre. Le parcours est guidé par la structure d'arbre.

Exemples :

- XSLT ou XQuery
- Compilation et grammaires attribuées

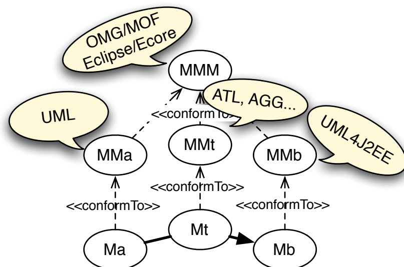
Génération 3 : Transformations de graphes

Principe : Un modèle en entrée (un graphe orienté étiqueté) est transformé en un modèle de sortie. La transformation est spécifiée par un autre modèle.

Transformation :
Mb* ← **f** (MMa*, MMb*, Mt, Ma*)

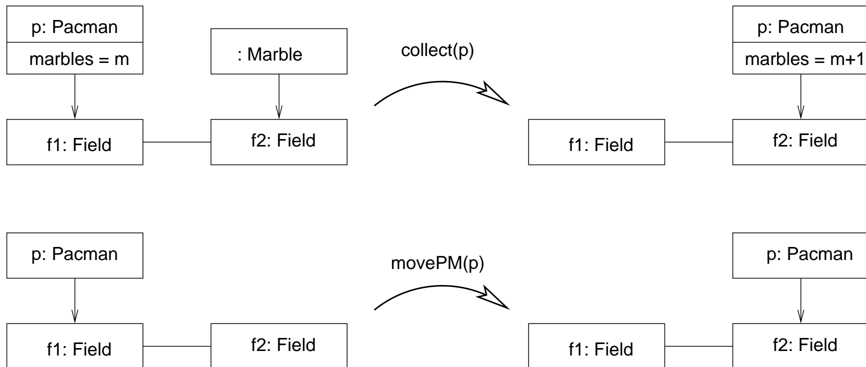
Remarque : Il pourrait y avoir plusieurs Ma et Mb.

Exemples : AGG, MDA/QVT, ATL...



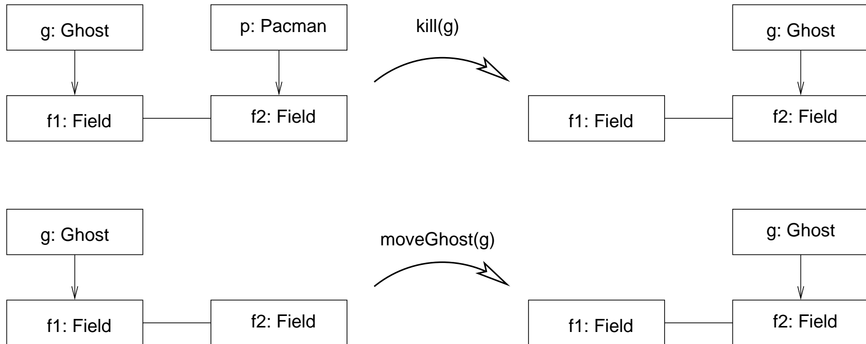
Génération 3 : Transformations de graphes

Exemple avec AGG et le pacman : règles du pacman



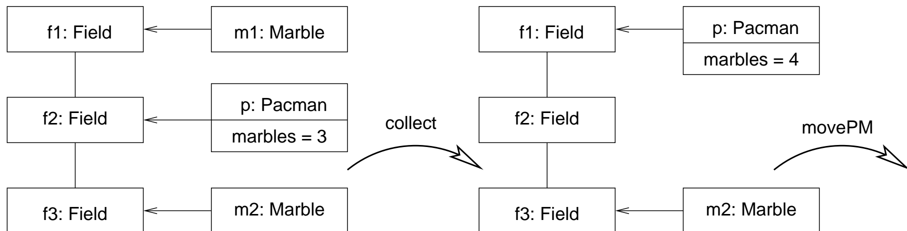
Génération 3 : Transformations de graphes

Exemple avec AGG et le pacman : règles du fantôme

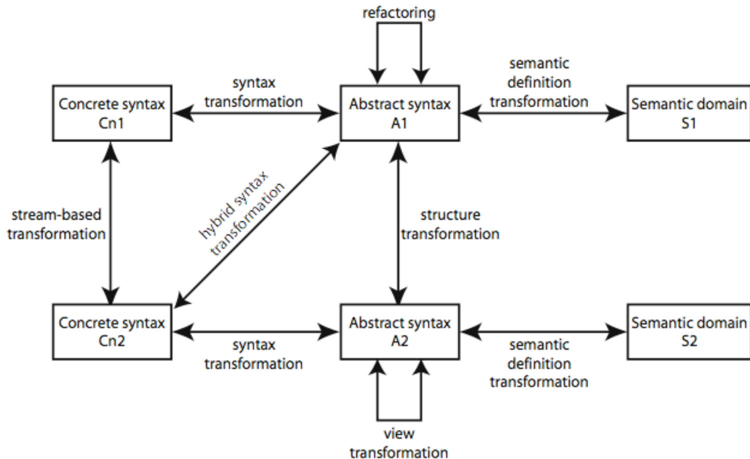


Génération 3 : Transformations de graphes

Exemple avec AGG et le pacman : exemple



Types de transformation



Sommaire

1 Contexte / Motivation

2 La transformation de modèle

3 Le standard QVT
- Appel à proposition QVT
- Le standard QVT
- Le langage Relations
- Operational mappings

4 Exemples de transformations endogènes

5 Transformation de modèle avec Kermeta

6 Conclusion

L'appel à propositions pour QVT

Description du problème

QVT = Query / Views / Transformations

Description du problème au quel doivent répondre les propositions de l'OMG QVT Request for Proposals (QVT RFP, ad/02-04-10) de 2002 :

- normaliser un moyen d'exprimer des correspondances (transformations) entre langages définis avec MOF ;
- exprimer des requêtes (**Query**) pour filtrer et sélectionner des éléments d'un modèle (y compris sélectionner les éléments source d'une transformation.)
- proposer un mécanisme pour créer des vues (**Views**).
  Vue = modèle déduit d'un autre pour en révéler des aspects spécifiques.
- formaliser une manière de décrire des transformations (**Transformations**) qui sont actuellement décrites en langage naturel, BNF, etc.

L'appel à propositions pour QVT

Exigences obligatoires

- définir un **langage de requêtes** sur des modèles ;
- définir un **langage de transformation** entre un méta-modèle source S et un méta-modèle cible T qui à partir d'un modèle source conforme à S engendre un modèle cible conforme à T.
- la **syntaxe abstraite** des langages de transformation, requête et vues, doit être définie comme un **méta-modèle MOF 2.0**.
- le langage de transformation doit permettre d'exprimer toute information nécessaire pour **engendrer automatiquement** le modèle cible depuis le source.
- le langage de transformation doit permettre de **créer une vue** d'un méta-modèle.
- le langage de transformation doit être **déclaratif**.
  Intérêt : pouvoir répercuter les changements sur les modèles.
- tous les mécanismes spécifiés dans la réponse doivent **opérer sur des modèles conformes aux méta-modèles définis avec MOF 2.0**.

L'appel à propositions pour QVT

Exigences optionnelles

- Les définitions de transformations peuvent être **bidirectionnelles**.
- **Traçabilité** entre les éléments source et destination lors de l'exécution d'une transformation.
- Mécanismes pour **réutiliser les définitions de transformation** (templates, pattern, redéfinition, surcharge).
- Utilisation de données supplémentaires à celle des modèles source et destination (**données intermédiaires**).
- Possibilité d'avoir **même modèle source et destination** (mise à jour d'un modèle, exemple : information dérivée).

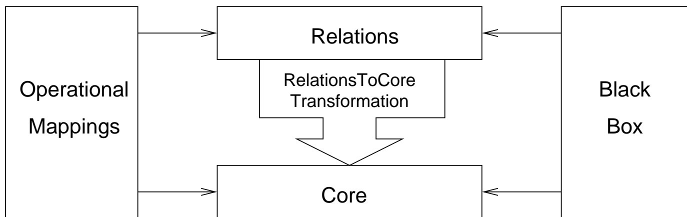
Le standard QVT

Informations générales

- **Standard OMG :** *MOF QVT final adopted specification* (ptc/05-11-01).
- Syntaxe abstraite en MOF 2.0 (et syntaxe concrète textuelle et graphique)
- Possibilité de plusieurs modèles (conformes à des méta-modèles issus de MOF)
- Langage de requête s'appuyant sur OCL
- Gestion automatique des liens de traçabilité
- Plusieurs scénarios d'exécution :
  - check-only : vérifier si les modèles donnés respectent une relation
  - transformations unidirectionnelles
  - transformations multi-directionnelles
  - répercutions sur les modèles cibles des modifications progressives sur les modèles sources (transformation incrémentale)

Le standard QVT

Architecture



- **Style déclaratif :**
  - *Relations :* spécification déclarative de haut niveau entre modèles MOF
  - *Core :* expressivité équivalente à *Relations* mais simplifié ($\equiv$ JVM)
- **Style impératif :**
  - *Operational Mappings Language* : étend *Relations* avec des constructions impératives (extension d'OCL)
  - *Black Box :* mécanisme pour appeler un programme externe

$\Rightarrow$ 3 langages qui ensemble constituent un « langage hybride ».

Le standard QVT
Conformité

	Syntax Executable	XMI Executable	Syntax Exportable	XMI Exportable
Core				
Relations				
Operational				

- **Axe horizontal :** intéropérabilité (4 niveaux)
- **Axe vertical :** langages (3 niveaux)
- 12 points de conformité possibles (pour les outils)
- **Notation :** QVT-<language level>-<interoperability level>
- *Exemple :* QVT-Relations-XMIExecutable

Le standard QVT

Transformations

transformation umlRdbms (uml: SimpleUML, rdbms: SimpleRDBMS) {
...

- Une transformation a un **nom** : umlRdbms
- Elle porte sur des **modèles candidats**. Ici deux modèles candidats uml et rdbms
- Chaque modèle candidat est **typé** par un **méta-modèle** : SimpleUML et SimpleRDBMS
  $\Longrightarrow$ Le modèle candidat doit être conforme à son méta-modèle.
- Une transformation contient des **relations** qui doivent être vraies sur les modèles candidats pour que la transformation soit réussie.
- Une **direction de transformation** peut être précisée (en choisissant un modèle qui devient cible). Si la transformation est réussie, le modèle cible sera modifié en conséquence : ajout, suppression ou modification d'éléments.

Le standard QVT

Relations

```
relation PackageToSchema { /* map each package to a schema */
    domain uml   p:Package { name = pn }
    domain rdbms s:Schema  { name = pn }
}
```

- Une **relation** spécifie des contraintes qui doivent être satisfaites sur les éléments de modèles candidats d'une transformation.
- **Domaine** = motif typé qui correspond à une partie d'un modèle candidat.
  - un paquetage p de uml qui a pour nom pn
  - un schéma s de rdbms de nom pn
  - p, s et pn sont des variables libres
- Les différents domaines (2 ici) doivent pouvoir être mis en correspondance.
  - Pour chaque paquetage, il doit y avoir un schéma de même nom
  - Pour chaque schéma, il doit y avoir un paquetage de même nom

Le standard QVT

Les clauses **when** et **where**

```
relation ClassToTable {
/* map each persistent class to a table */
  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
    }
  }
  domain uml c:Class {
    namespace = p:Package {},
    kind = ' Persistent ',
    name = cn
  }
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t);
  }
}
```

- Clause **when** : condition sous laquelle la relation doit être vérifiée.
- Clause **where** : condition que doivent vérifier tous les éléments de modèle participant à la relation. Permet de contraindre les variables libres de la relation et ses domaines.

Le standard QVT

Top−level relations

```
transformation umlRdbms (uml: SimpleUML, rdbms: SimpleRDBMS) {
  top relation PackageToSchema { ... }
  top relation ClassToTable { ... }
  relation AttributeToColumn { ... }
}
```

- une transformation contient deux types de relation : **top** et non-**top**
- l'exécution d'une transformation nécessite que toutes les **top** soient satisfaites. Les non-**top** ne doivent l'être que si elles sont invoquées directement ou transitivement dans une clause **where**.

Le standard QVT

Les modes check et **enforce**

```
relation PackageToSchema { /* map each package to a schema */
    checkonly domain uml   p:Package { name = pn }
    enforce   domain rdbms s:Schema  { name = pn }
}
```

- **checkonly** : indique que le domaine ne peut pas être modifié
- **enforce** : indique que le domaine peut être modifié (s'il est la cible de la transformation).
- Conséquences :
  - Si le modèle cible est marqué **checkonly**, alors la tranformation ne pourra qu'indiquer si elle est vérifiée ou non.
  - Si le modèle cible est marqué **enforce**, alors la tranformation peut le modifier pour faire que la relation soit satisfaite.

Le standard QVT

Pattern matching : définition

- Un motif (pattern) apparaît dans un domaine et permet de sélectionner une partie du modèle candidat (*pattern matching*).
- Le motif suivant est associé au domaine SimpleUML et possède des variables libres (potentiellement : c, p, cn).

```
domain uml c:Class {
  namespace = p:Package {},
  kind = ' Persistent ',
  name = cn
}
```

- Certaines variables peuvent déjà être contraintes (par une clause **when** ou un autre pattern).
  - p est contrainte par la clause **when** PackageToSchema(p, s)
- Le résultat d'un « pattern matching » est une correspondance sur des éléments du modèle candidat (qui donne une valeur aux variables libres) : tuple

Le standard QVT

Pattern matching : évaluation

Sur l'exemple précédent, la mise en correspondance (*matching*) suit les étapes :

- Filtrer tous les objets de type Class dans le modèle « uml »
- Éliminer les classes qui n'ont pas les propriétés imposées dans le motif par des valeurs littérales.
  - Ici, suppression des classes dont la propriété kind n'est pas ' Persistent '
- Deux possibilités pour les propriétés liées par une variable (ex : name = cn) :
  - si cn est déjà liée, suppression des éléments qui n'ont pas la même valeur.
  - si cn est libre, elle est liée à la valeur de la propriété les classes non supprimées par un autre filtrage.
    La valeur de cn pourra alors être utilisée dans d'autres motifs ou être plus tard contrainte (exemple : **where**)
- Continuer avec les motifs imbriqués.
  Exemple : namespace = p:Package {}
  Dans notre cas, elle est liée par la clause **when**
- Les motifs peuvent être arbitrairement imbriqués.
- Les tuples résultats sont utilisés suivant la direction de la transformation

Le standard QVT

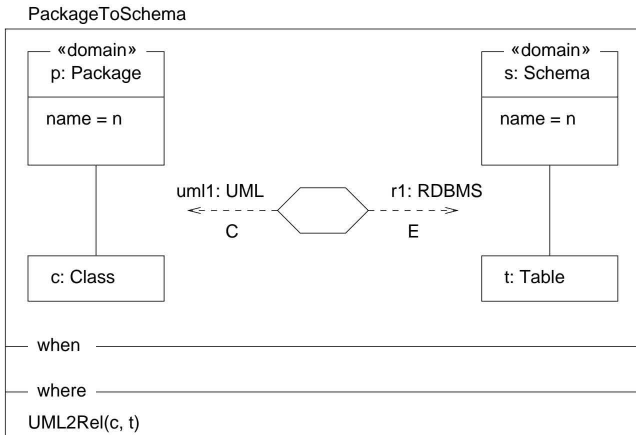
Pattern matching : exploitation

- La résultat d'un « pattern matching » donne un ensemble de tuples (la variables libres et leur valeur)
- Si la transformation est exécutée dans la direction de rdbms (modèle cible) :
  - pour chaque tuple valide du domaine uml, il doit exister au moins un tuple valide du domaine rdbms qui satisfait la clause **where**
  - si pour un tuple valide de uml, il n'existe pas de tuple dans rdbms, de nouveaux éléments sont créés avec les bonnes propriétés (rdbms est marqué **enforce**).
  - inversement, pour chaque tuple valide de rdbms, il doit exister au moins un tuple valide de uml qui satifait la clause **where**
  - si pour un tuple valide de rdbms, il n'existe pas de tuple valide dans uml, les éléments correspondants de rdbms sont supprimés.
- **Remarque :** Pour pouvoir identifier les objets et savoir s'il faut en créer un nouveau ou modifier un objet existant, QVT propose de définir des clés (ensemble de propriétés sur l'objet)

  key Table { schema, name};

Le standard QVT

Représentation graphique



Le standard QVT
Operational transformations

```
transformation Uml2Rdbms(in uml: UML, out rdbms: RDBMS) {
   // the entry point for the execution of the transformation
   main() {
         uml. objectsOfType(Package)->map packageToSchema();
   }
  ....
}
```

- Une *operational transformation* est la définition d'une transformation unidirectionnelle exprimée dans un style impératif
- Elle définit une signature avec les modèles impliqués et leur direction (**in**, **out**, **inout**)
- Elle définit un point d'entrée (**main**) qui ici applique sur chaque élément du modèle uml de type Package le *mapping* packageToSchema().
- Comme une classe, une telle transformation est instanciable et peut avoir des propriétés et opérations.
- Notion d'héritage (**extends**) et d'utilisation (**access**)

Le standard QVT

Mapping operation

```
mapping Package::packageToSchema() : Schema
    when { self.name.startingWith() <> "_"} // self is a Package
{
    name := self.name; // name of the result Schema
    table := self.ownedElement->map class2table();
}
```

- Une « mapping operation » fait correspondre à un ou plusieurs éléments source, un ou plusieurs éléments cibles
- Elle définit une signature (modèle impliqués et direction), une garde (**when**), un corps et une post-condition (**where**).
- est toujours unidirectionnelle
- sélectionne les éléments source sur leur type (Package) et une garde (**when**)
- exécute les opérations de son corps pour créer les éléments cibles
- peut invoquer d'autres « mapping operations »
- peuvent être reliées par héritage

Le standard QVT

Mapping operation : autres concepts

```
mapping Package::packageToSchema() : result:Schema
   when {  self.name.startingWith() <> "_"}
{
    init  {}
   // implicit  creation of objects not  initialised  or created in  init
   population {
     object  result :Schema {
       name :=  self.name;
       table  :=  self .ownedElement−>map class2table();
     }
   }
   end {}
}
```

- Cette nouvelle formulation est équivalente à la précédente.
- **init** : permet d'initialiser les éléments résultat (sinon création implicite)
- **population** : code qui définit la valeur de l'objet résultat et des paramètres en **out** et **inout** (action par défaut).
- **end** : exécuté avant de quitter l'opération

Le standard QVT

Helpers

```
query Class :: isPersistent () : Boolean = self.kind='persistent';
query Association :: isPersistent () : Boolean =
      ( self.source.kind='persistent' and self.destination.kind='persistent');
query Class :: checkConsistency(typename:String) : Boolean {
   if (not typename) return false;
   if (cl := self.namespace.lookForClass(typename) ) return false;
   return self.compareTypes(cl);
}
helper Package::computeCandidates(inout list : List) : List {
   if ( self.nothingToAdd()) return list;
   list += self.retrieveCandidates ();
   return list;
}
```

- **helper :** opération sur des objets source et qui fournit un résultat.
- Un helper est associé à un type, primitif ou de modèle.
- Il a un code séquentiel (avec :=, **while**, **forEach**...).
- **Intérêt :** faciliter l'écriture de navigation complexe.
- **Query (requête) =** helper sans effet de bord

Le standard QVT

Mise à jour d'objet déjà créés

```
mapping Association::asso2table() : Table
when { self.isPersistent ()}
{ -- result is the default name for the output parameter of the rule
  init { result := self.destination.resolveone( isKindOf(Table)); }
  foreignKey := self.map asso2ForeignKey();
  column := result.foreignKey.column;
}
```

- **Justification :** Les transformations peuvent être faites en plusieurs passes : $\Longrightarrow$ nécessité de retrouver les objets déjà construits pour les mettre à jour
- **Moyen :** opérations de résolution qui s'appuie sur les informations de trace
- **resolveone**(condition) : ici, inspecte les données de trace pour y trouver les objets qui vérifient la condition (être une instance de Table).
- Autres variantes :
  - **invresolve**(cond) : l'objet responsable de la création de l'objet *self*
  - **resolveIn**(aMapping, cond) : les objets cible créés par un même mapping

Le standard QVT

Mise à jour d'objet déjà créés : **resolveIn**

```
transformation JClass2JPackage(inout javamodel:JAVA);
    main () {
        javamodel-> objectsOfType(JClass)->jclass2jpackage();
    }
    mapping Class:: jclass2jpackage () : JPackage () {
        init {
            result := resolveIn( jclass2jpackage , true)
                    // all JPackage objet created using jclass2jpackage
                ->select(p| self.package=p.name)->first();
                    // select those named p.name and take the first one
            if result then return;
                // avoid creating two packages with the same name
        }
        name := self.package;
            // set the name of the new created JPackage object
        }
}
```

Sommaire

1. Contexte / Motivation

2. La transformation de modèle

3. Le standard QVT

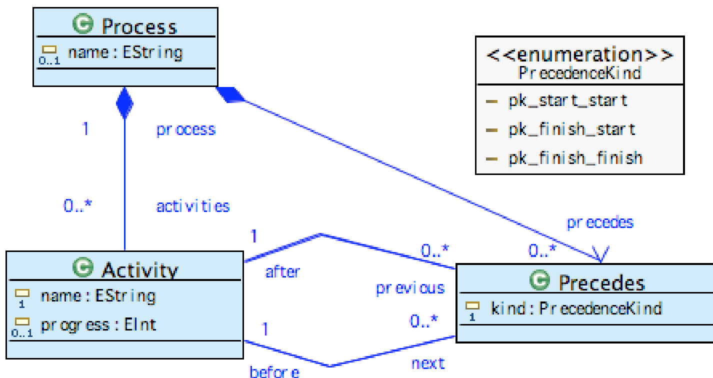
4. Exemples de transformations endogènes
   - Problème posé
   - Approche méta-programmation : Kermeta
   - Approche transformation : ATL

5. Transformation de modèle avec Kermeta

6. Conclusion

Exécution (simulation) d'un modèle SimplePDL

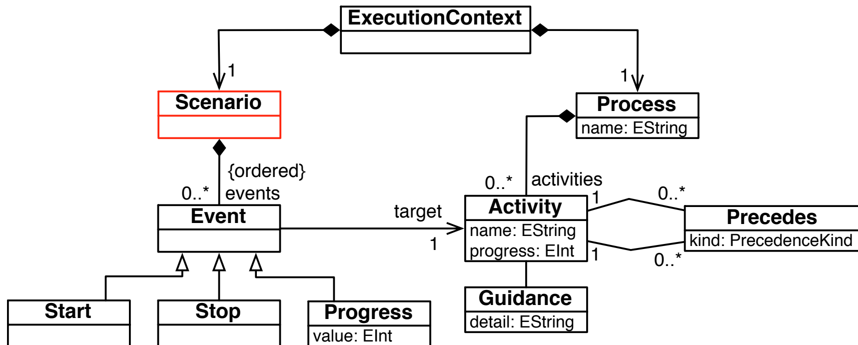
Le méta-modèle



Remarque : l'attribut progress d' Activity a été ajouté pour conserver l'état d'un procédé en cours de simulation.

Exécution (simulation) d'un modèle SimplePDL

Les événements utilisateur

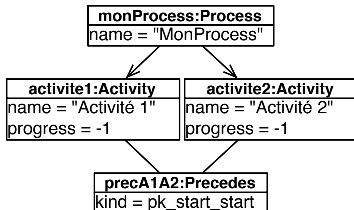


Exécution (simulation) d'un modèle SimplePDL

Exemple de procédé

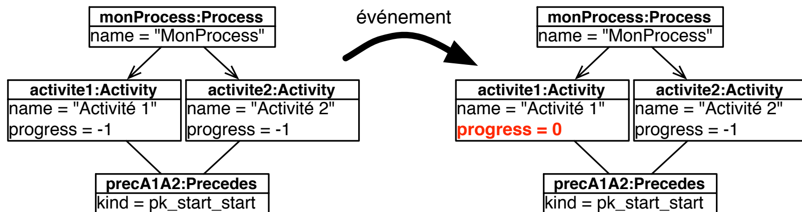
Exécution (simulation) d'un modèle SimplePDL

Évolution du procédé



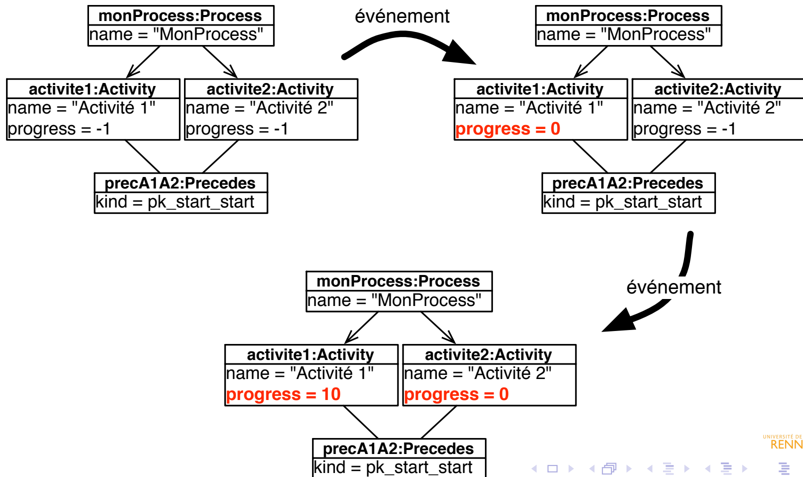
Exécution (simulation) d'un modèle SimplePDL

Évolution du procédé



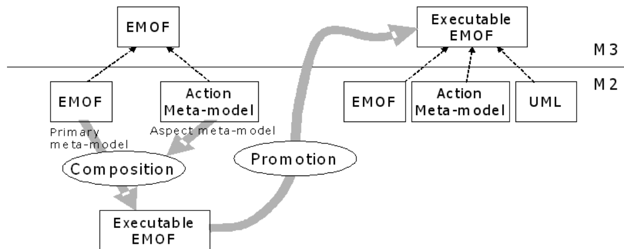
Exécution (simulation) d'un modèle SimplePDL

Évolution du procédé



Approche méta-programmation : Kermeta
Présentation générale

- Développé par l'équipe Triskell à l'IRISA (Rennes)
- Défini comme un « langage de méta-programmation orienté objet »



Approche méta-programmation : Kermeta

Méta-modèle augmenté

Approche méta-programmation : Kermeta

Le code de startable

```
operation startable () : Boolean is do
    var start_ok : kermeta::standard::Boolean
    var previousActivities : seq Activity [0..*]
    var prevPrecedes : seq Precedes [0..*]
    if progress==-1 then
        // Getting the activities which have to be started
        prevPrecedes := previous.select{p | p.kind == PrecedenceKind.pk_start_start }
        previousActivities := prevPrecedes.collect{p | p.before}
        start_ok := previousActivities .forAll{a | a.progress >= 0}
        // Getting the activities which have to be finished
        prevPrecedes := previous.select{p | p.kind == PrecedenceKind.pk_finish_start }
        previousActivities := prevPrecedes.collect{p | p.before}
        start_ok := start_ok and ( previousActivities .forAll{a | a.progress==100})
        result := start_ok or (previous.size() == 0)
    else
        result := false
    end
end
operation progress() is do
    if self.progress >= 0 and self.progress < 90 then
        progress := progress + 10;
    endif
end
```

Approche transformation : ATL

ATL : ATLAS Transformation Language

- Développé par l'équipe de recherche ATLAS INRIA ET LINA (Jean Bézivin),
- Langage de transformation **hybride**
  - Propose des structures **déclarative** et **impérative**.
- Une transformation ATL est composée de **règles (rules) déclaratives**.
- Une **règle** définit une transformation d'un **élément du modèle source** vers **un élément du modèle cible**
  - Possibilité d'appeler des **Helpers** (eq. méthodes) pour faciliter le traitement dans les règles.

Approche transformation : ATL

SimplePDL2SimplePDL : *helpers* startable , finishable et newProgree

```
helper context simplepdl!Activity
    def : startable () : Boolean = (
        self.progress < 0          -- not started
        and self.previous->select(p | p.kind = #pk_start_start)
                ->collect(p | p.before)          -- precedence kind start/start
                ->forAll(a | a.progress >= 0)    -- started
        and self.previous->select(p | p.kind = #pk_finish_start)
                ->collect(p | p.before)          -- precedence kind finish/start
                ->forAll(a | a.progress = 100)   -- stoped
);
helper context simplepdl!Activity
    def : finishable () : Boolean = (
        self.progress >= 0 and self.progress < 100
        and self.previous->select(p | p.kind = #pk_finish_finish)
                ->collect(p | p.before)                  -- previous finish/finish
                ->forAll(a | a.progress = 100)           -- are finished
    );
helper context simplepdl!Activity
    def : newProgress () : Interger = self.progress + 10;
```

Approche transformation : ATL

SimplePDL2SimplePDL : *règle* sur Activity

```
rule progressActivity {
  from
      a_in : simplepdl!Activity
  to
      a_out : simplepdl!Activity (
          -- compute new progress value
          progress <-
              if a_in.startable()
              then 0
              else
                  if a_in.progress >= 0 and a_in.progress < 90
                  then a_in.newProgress()
                  else if a_in.finishable()
                      then 100
                      else a_in.progress
                      endif
                  endif
              endif
          -- copy other attributes
          name <- a_in.name,
          process <- a_in.process,
          previous <- a_in.previous,
          next <- a_in.next,
     )
}
```

Sommaire

1. Contexte / Motivation
2. La transformation de modèle
3. Le standard QVT
4. Exemples de transformations endogènes
5. Transformation de modèle avec Kermeta
6. Conclusion

Problématique générale

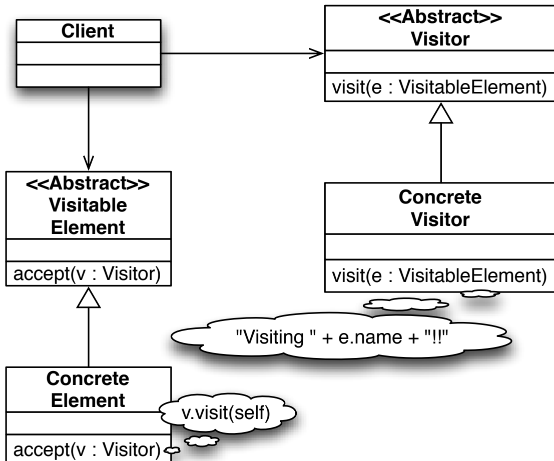
Problème

- La transformation prend en entrée un modèle *m1*
- Et doit produire en sortie un autre modèle *m2*
- Approche structurelle :
  - Les métamodèles définissent les structures
  - Mais aucun support à la transformation !

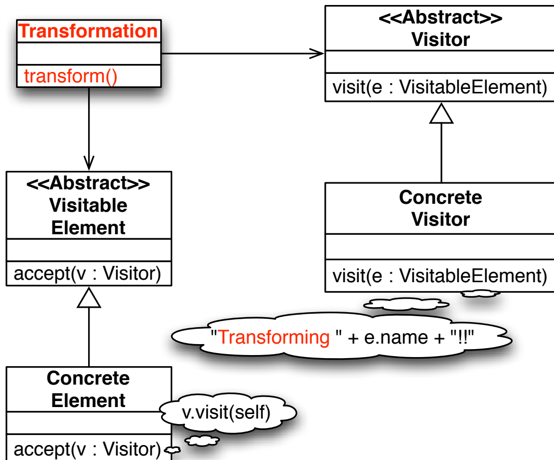
Solution

- Schéma de conception visiteur
- Visite des entités composant le modèle *m1*
- Approche structurelle :
  1. Construction des entités de *m2* correspondantes
  2. Liaison des nouvelles entités entre elles

Implanter la transformation sous la forme d'un visiteur



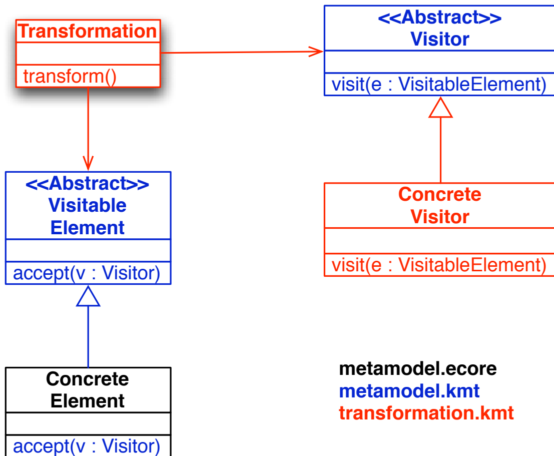
Implanter la transformation sous la forme d'un visiteur



Utilisation des aspects pour implanter la transformation

- Le métamodèle originel n'implante pas un schéma Visiteur
  $\Rightarrow$ Il suffit de l'enrichir (tissage d'aspect)
- Quelle que soit la transformation :
  - On requiert le métamodèle voulu
  - On y tisse :
    - La méthode ACCEPT sur toutes ses entités
    - Une classe abstraite décrivant les visiteurs de ce modèle
    - Une classe abstraite représentant la transformation
- Pour une transformation donnée :
  - On requiert le métamodèle précédemment enrichi
  - On y ajoute :
    - Un (ou plusieurs) visiteurs
    - Une classe de transformation

Utilisation des aspects pour implanter la transformation



Sommaire

1. Contexte / Motivation
2. La transformation de modèle
3. Le standard QVT
4. Exemples de transformations endogènes
5. Transformation de modèle avec Kermeta
6. Conclusion

Conclusion

Les transformations : un complément indispensable pour l'IDM

Gains espérés :

- mise en œuvre des processus en Y
- automatiser (au moins partiellement) le passage entre modèles et/ou espaces technologiques
- favoriser l'interopérabilité entre outils
- capitaliser le savoir faire
- plus haut niveau d'abstraction → meilleure maîtrise, vérification, validation

État actuel :

- manque de maturité des outils disponibles ;
- comment tester une transformation ?
- nombreuses contraintes techniques ;
- écrire une transformation en Java a encore un sens. Mais vérifier son programme Java est difficile !

Un peu de recul...

Une autre vision (grossière) du MDE ! en un slide...

MDE (*Model Driven Engineering*)

= **DSL** (*Domain Specific Language*) : séparation des préoccupations au travers de langages offrant des constructions capitalisant l'expérience d'un domaine particulier (p.-ex., le MDA qui préconise une séparation du métier et de la plate-forme)

+ **Generative approach** approche s'appuyant sur le langage de manière à raisonner sur l'ensemble des modèles (mis en oeuvre généralement par des transformations)

Code

FIG.: Exemple du MDA

Un peu de recul...

Une autre vision (grossière) du MDE ! en un slide...

MDE (*Model Driven Engineering*)

= **DSL** (*Domain Specific Language*) : séparation des préoccupations au travers de langages offrant des constructions capitalisant l'expérience d'un domaine particulier (p.-ex., le MDA qui préconise une séparation du métier et de la plate-forme)

\+ **Generative approach** approche s'appuyant sur le langage de manière à raisonner sur l'ensemble des modèles (mis en oeuvre généralement par des transformations)

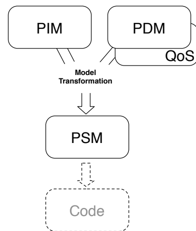


FIG.: Exemple du MDA

Un peu de recul...

Une autre vision (grossière) du MDE ! en un slide...

MDE (*Model Driven Engineering*)

= **DSL** (*Domain Specific Language*) : séparation des préoccupations au travers de langages offrant des constructions capitalisant l'expérience d'un domaine particulier (p.-ex., le MDA qui préconise une séparation du métier et de la plate-forme)

+ **Generative approach** approche s'appuyant sur le langage de manière à raisonner sur l'ensemble des modèles (mis en oeuvre généralement par des transformations)

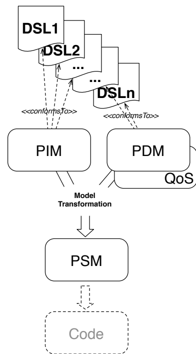


FIG.: Exemple du MDA

Un peu de recul...

Une autre vision (grossière) du MDE ! en un slide...

MDE (*Model Driven Engineering*)

= **DSL** (*Domain Specific Language*) : séparation des préoccupations au travers de langages offrant des constructions capitalisant l'expérience d'un domaine particulier (p.-ex., le MDA qui préconise une séparation du métier et de la plate-forme)

\+ **Generative approach** approche s'appuyant sur le langage de manière à raisonner sur l'ensemble des modèles (mis en oeuvre généralement par des transformations)

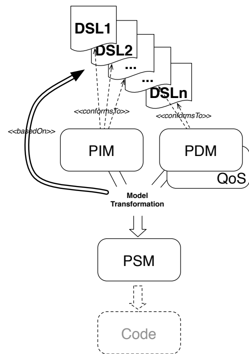


Fig.: Exemple du MDA

Un peu de recul...

L'informaticien 2.0

Expert Métier

- Capitalise son expérience au sein de langages métiers
- Utilise des langages dédiés aux domaines adressés par les systèmes complexes qu'il doit définir

Expert Plate-forme

- Fournit des approches génératives prenant en compte les spécificités d'une plate-forme d'exécution particulière

Expert Langage

- Fournit des *méta-* approches génératives facilitant l'outillage d'un nouveau DSL