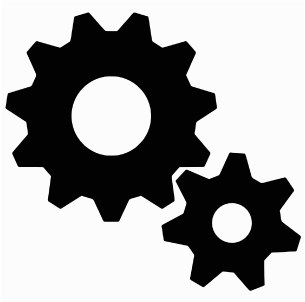
ENSEEIHT : Génie du Logiciel et des Systèmes (GLS)

Rapport de projet

GLS : Rapport de projet



Matthieu Pizenberg

20 décembre 2013

Table des matières

Chapitre 1

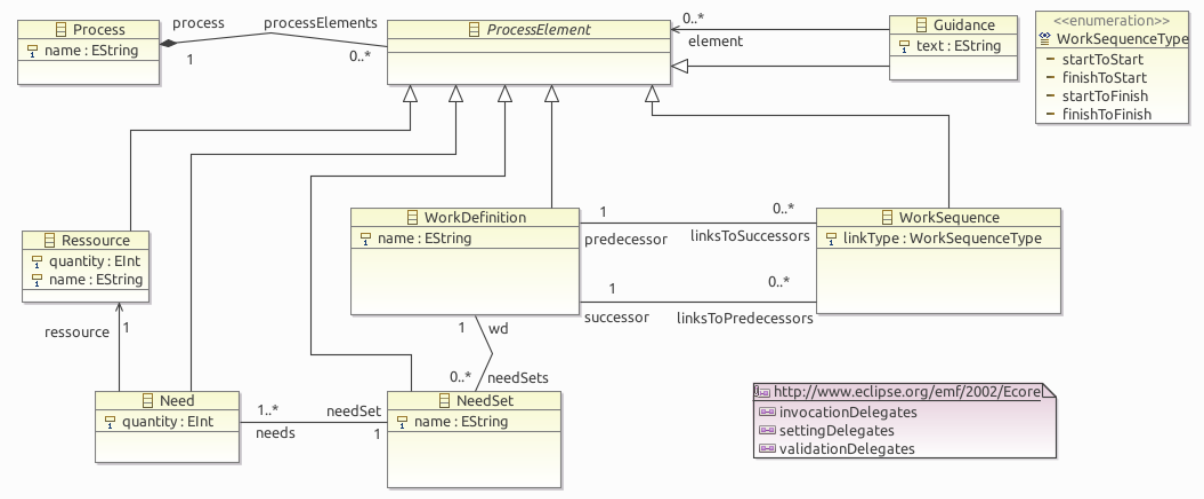
Les métamodèles SimplePDL et PetriNet

Pour l'ensemble de la chaine de transformation on aura besoin des métamodèles SimplePDL et PetriNet, on définit donc ces deux métamodèles ainsi que des exemples de modèles qui serviront à faire les tests.

1.1 SimplePDL

On a ajouté au modèle SimplePDL la gestion des ressources. Pour celà, on a introduit :
- Ressource : une ressource ayant un nom et étant présente en une certaine quantité
- Need : un besoin représentant un nombre de ressources d'un certain type
- NeedSet : un ensemble de besoins nécessaires à la réalisation d'une WD

Voilà un diagramme représentant le métamodèle SimplePDL que j'ai utilisé :

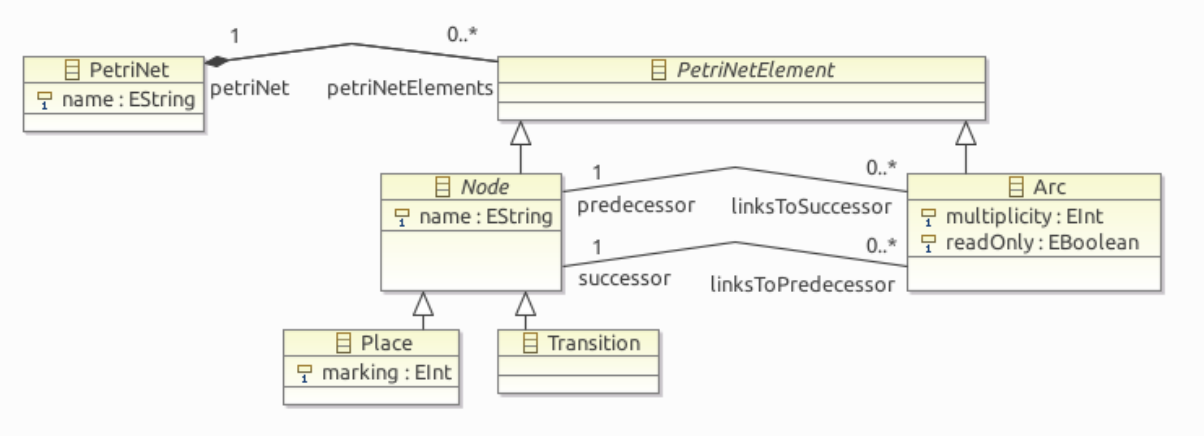


Ce modèle n'est pas suffisant pour décrire entièrement l'ensemble des contraintes. Pour celà on a donc ajouté des contraintes OCL avec OCLinEcore.

On a ajouté les règles suivantes :
- class Process
  - sameWDName : les WD doivent avoir des noms différents
  - sameRessourcesName : les ressources doivent avoir des noms différents
  - nameForbidden : un processus ne peut pas s'appeler : "Process"
  - sameNeedSetsName :
- class WorkDefinition
  - voidName : nom vide interdit
- class WorkSequence
  - previousWDinSameProcess : la WD précédente doit être dans le même processus
  - reflexivity : le prédécesseur et le successeur oivent être différents
  - nextWDinSameProcess : la WD suivante doit être dans le même processus
- class Ressource
  - voidName : nom vide interdit
  - positiveQuantity : la quantité de ressources doit être $>= 0$
- class Need
  - positiveQuantity : la quantité de ressources d'un besoin doit être > 0
- class NeedSet
  - voidName : nom vide interdit

1.2 PetriNet

Le modèle PetriNet que j'ai utilisé est représenté par le diagramme suivant :

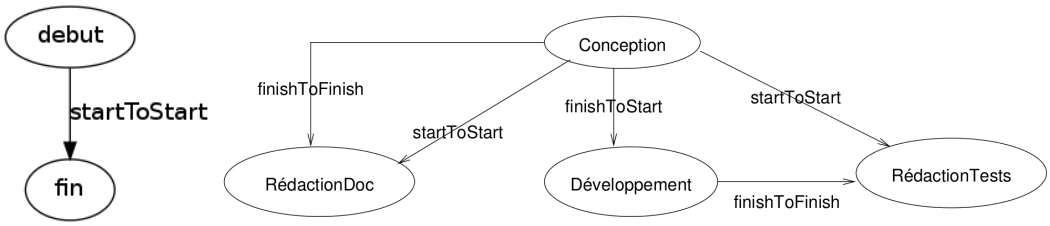


Ce modèle n'est pas suffisant pour décrire entièrement l'ensemble des contraintes. Pour celà on a donc ajouté des contraintes OCL avec OCLinEcore.

On a ajouté les règles suivantes :
- class PetriNet
  - voidPetriName : nom vide interdit
  - sameNodeName : les noms des noeuds doivent être différents
- abstract class Node
  - voidNodeName : nom vide interdit
- class Arc
  - positiveMultiplicity : la multiplicité doit être > 0
  - sameTypeOfPredecessorAndSuccessor : Prédecesseur et successeur doivent être différents
  - nextNodeNotInSamePetriNet : les noeuds reliés sont dans le même réseau de pétri
  - previousNodeNotInSamePetriNet :les noeuds reliés sont dans le même réseau de pétri
- class Place
  - positiveMarking : le marquage d'une place est $>= 0$

1.3 Exemples de modèles

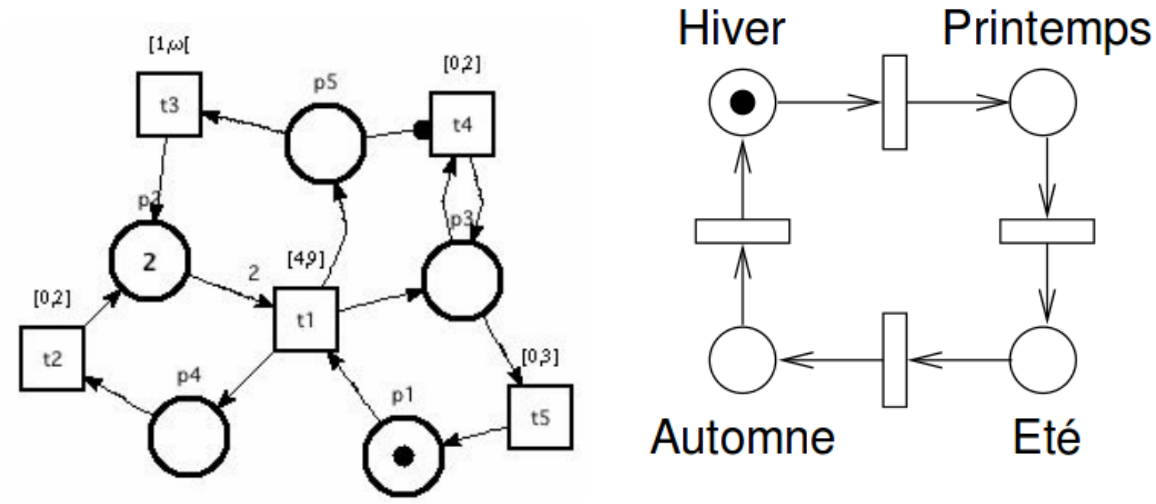
Voici quelques exemples de modèles SimplePDL (sans ressources) :



(a) Processus simple (b) Processus

FIGURE 1.1: Exemples de modèles de processus SimplePDL

Et quelques exemples de modèles PetriNet :



(a) Exemple (b) 4 saisons

FIGURE 1.2: Exemples de modèles de réseaux de pétri

Chapitre 2

Transformations modèle à modèle (M2M)

Pour analyser les modèles de processus en SimplePDL, on va avoir besoin de les transformer en réseaux de pétri. Pour çà, on doit effectuer des transformations ModelToModel (M2M) avec comme origine un modèle SimplePDL et comme arrivée un modèle PetriNet.

On utilise l'outil ATL (plugin pour Eclipse) afin d'effectuer ces transformations.

2.1 L'outil ATL

On montre ici brièvement la syntaxe d'un fichier ATL avec quelques exemples :

```
-- Traduire un Process en un PetriNet de même nom
rule Process2PetriNet {
   from p: simplepdl!Process
   to pn: petrinet!PetriNet (name <- p.name)
}
```

Les règles de transformation sont exprimées avec le mot-clé "rule". Les mots "from" et "to" permettent d'identifier les éléments de la transformation.

```
-- Traduire une WorkSequence en un motif sur le réseau de Petri
rule WorkSequence2PetriNet {
   from ws: simplepdl!WorkSequence
   to
      a_ws: petrinet!Arc(
         petriNet <- ws.getProcess(),
         readOnly <- true,
         multiplicity <- 1,
         predecessor <- thisModule.resolveTemp(ws.predecessor,
            if((ws.linkType = #finishToStart) or (ws.linkType = #finishToFinish))
               then 'p_finished'
               else 'p_started'
            endif),
         successor <- thisModule.resolveTemp(ws.successor,
            if((ws.linkType = #finishToStart) or (ws.linkType = #startToStart))
               then 't_start'
               else 't_finish'
            endif)
         )
}
```

La commande "resolveTemp" permet d'identifier un élément parmi ceux présents dans les règles de transformations.

2.2 De SimplePDL à PetriNet

Le code de transformation complet étant assez long je vais ici expliquer à l'aide de graphiques les opérations pour transformer les éléments SimplePDL en éléments de réseau de pétri :

2.2.1 WorkDefinition WD

J'ai voulu modifier au minimum la structure des WD et des WS dont on avait les transformations avant l'ajout des ressources aux modèles. Ainsi on assure une évolution simple des WD et WS.

On commence par une WorkDefinition WD. Avec les ressources en plus, une WD a besoin de savoir si des ressources sont disponibles pour pouvoir démarrer une activité.

- On ajoute donc une place ressources_available reliée à la transition start d'une WD.
- On ajoute également une place ressources_wait (marquée à 1) pour indiquer que la WD est en attente de ressources
- Enfin on ajoute une place ressources_returned reliée en sortie de la transition finish qui inque que la WD a fini d'utiliser les ressources

On a donc ajouté 3 places et 2 arcs à notre WD. Voici un graphique récapitulatif :

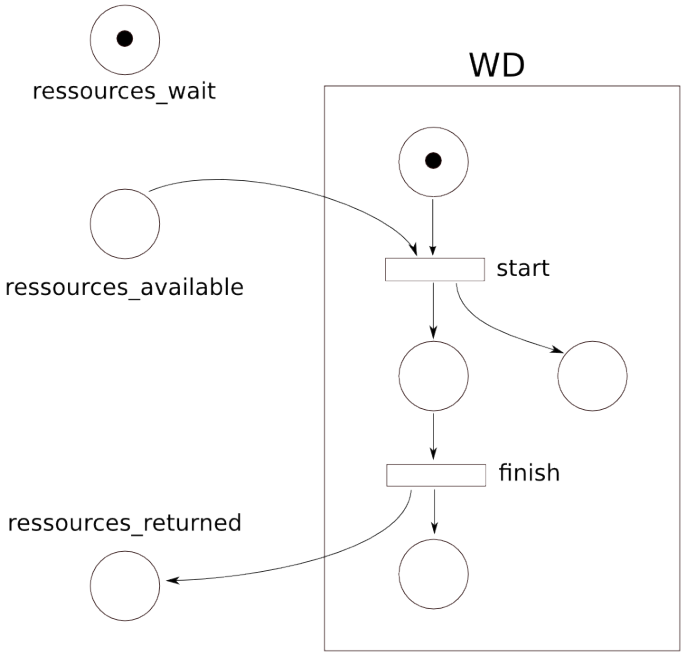


FIGURE 2.1: Transformation d'une WorkDefinition WD en réseau de pétri

2.2.2 NeedSet

Je ne décris pas le besoins (Need) car il s'agit d'une simple place avec un marquage correspondant à "quantity".

Chaque ensemble de besoins (NeedSet) correspond à un ensemble constitué de :
- une place "set" pour indiquer de quel NeedSet il s'agit
- 3 transition :
  - "take" qui est reliée à chaque ressource nécessaire pour le NeedSet
  - "makeAvailable" entre la place "set" du NeedSet et la place "ressources_available" de la WD
  - "return" reliée aux ressources qui vont être rendues
- enfin les arcs qui vont bien (voir schéma)

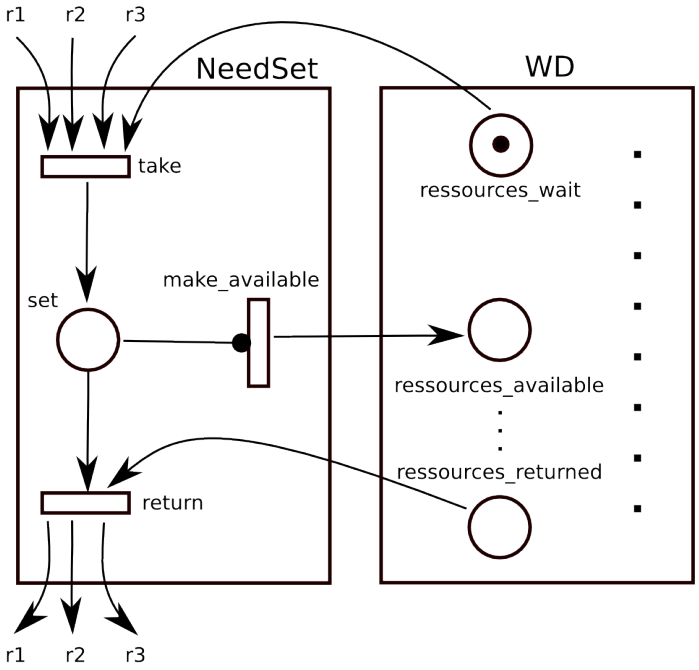


FIGURE 2.2: Transformation d'un ensemble de besoins NeedSet en réseau de pétri

Chapitre 3

Transformations modèles à textes (M2T)

Les transformations M2T (ModelToText) permettent d'obtenir des fichiers avec uns syntaxe respectant certaines contraintes. Un unique modèle peut être représenté par différents fichiers ayant des syntaxes différentes, facilitant ainsi l'usage dans un environnement donné.

Pour effectuer ces transformations nous avons utilisé Acceleo, lui ausi intégré à Eclipse sous la forme d'un plugin.

3.1 L'outil Acceleo

Le langage utilisé par Acceleo (d'extension .mtl) est un langage de template. On construit bout par bout notre fichier texte à l'aide de boucles, de conditions et de requêtes.

Voilà un exemple de boucle avec condition :

```
[for (pl : Place | petriNet.petriNetElements->getPlaces())]
[if (pl.marking > 0)]pl [pl.name/] ([pl.marking/])[/if]
[/for]
```

Et voilà un exemple de requête pour récupérer les places d'un réseau de pétri :

```
[query public getPlaces(elements : OrderedSet(PetriNetElement)) : OrderedSet(Place) =
   elements->select( e | e.oclIsTypeOf(Place) )
      ->collect( e | e.oclAsType(Place) )
      ->select( e | e.marking > 0 )
      ->asOrderedSet()
/]
```

3.2 PetriNet

Sans entrer dans les détails du code disponible dans les sources je précise quelques spécificités de la transformation. Le template principal fait appel à d'autres templates pour rendre la lecture du .mtl plus facile et plus modulaire. Le template différencie bien les read-arc mais il n'est pas possible de préciser un marquage maximal autorisé.

Voilà ce que ça donne avec nos deux fichiers pdl convertis en petrinet :

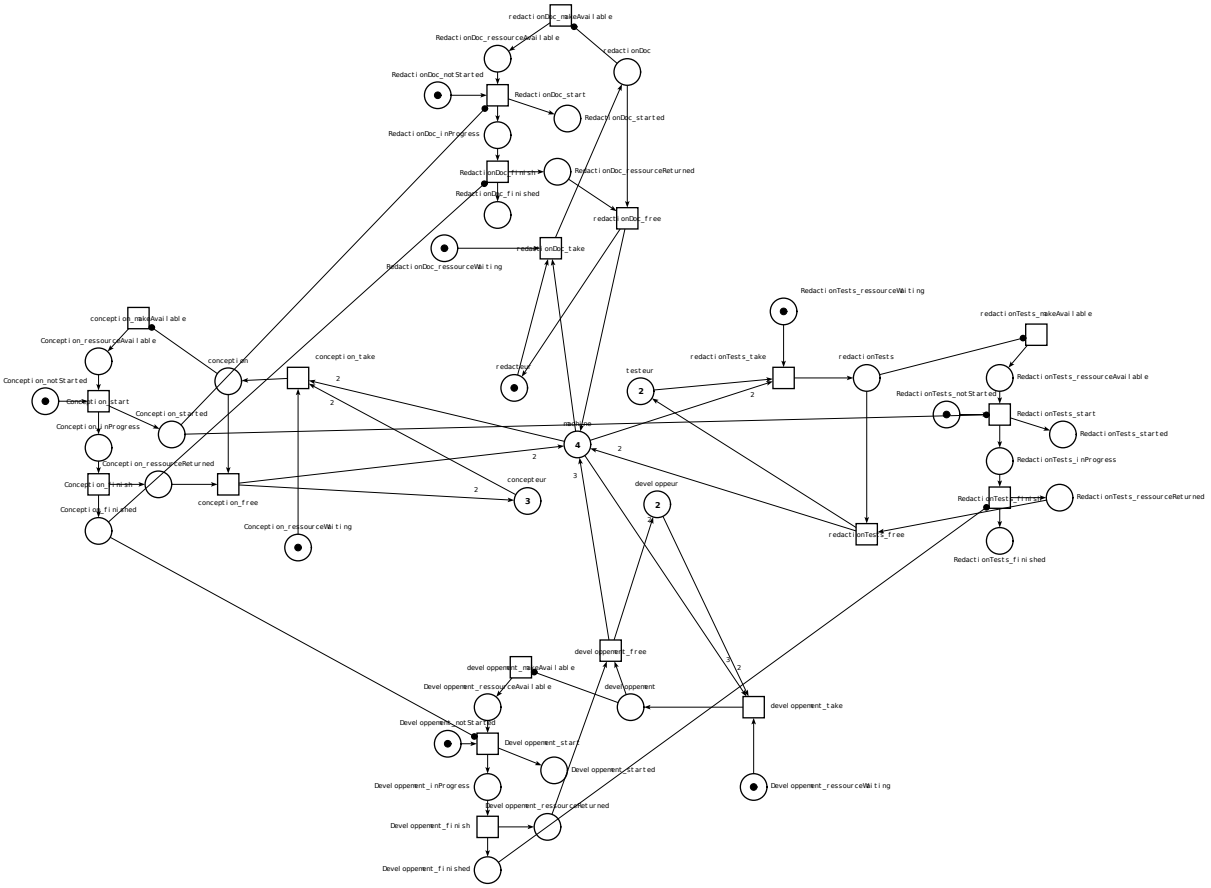


FIGURE 3.1: Processus tranformé en pétrinet

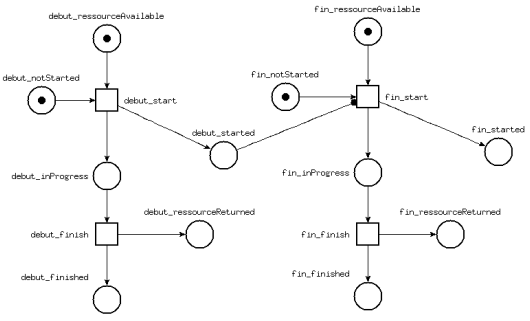


FIGURE 3.2: Processus simple tranformé en pétrinet

Chapitre 4

Transformations textes à modèles (T2M)

Enfin nous avons besoin de pouvoir saisir rapidement et simplement des modèles de réseau de pétri ou SimplePDL. Pour celà, on utilise la transformation T2M (TextToModel) qui permet d'établir une syntaxe pour un modèle. L'outil que l'on utilise est également intégré à Eclipse sous la forme d'un plugin, il s'agit d'Xtext.

4.1 L'outil Xtext pour SimplePDL et PetriNet

Xtext permet de spécifier la syntaxe d'un fichier décrivant un modèle. Cette spécification permet également la construction automatique du métamodèle correspondant à la syntaxe concrète décrite.

En fait, il apparait que le métamodèle construit est légèrement différents du métamodèle initial. Ceci est d'autant plus vrai que la syntaxe est simplifiée et s'éloigne de la structure du modèle. Ce n'est pas si grave, puisque l'on connait maintenant les outils permettant de faire des transformations M2M.

Voilà à quoi ressemble la syntaxe d'un fichier xtext, ici le contenu du xtext pour petrinet :

```
PetriNet :
    'petrinet' name=ID '{'
        petriNetElements+=PetriNetElement*
    '}'
    ;

PetriNetElement :
    Node
    | Arc
    ;

Node :
    Place
    | Transition
    ;

Place :
    'place' name=ID ('('marking=INT')')?
    ;

Transition :
    'transition' name=ID
    ;

Arc :
    'arc' ('('multiplicity=INT')')? (readOnly ?= 'r')?
    'from' predecessor=[Node]
    'to' successor=[Node]
```

```
    ;
```

Le fichier xtext pour notre syntaxe pdlx (pdl extended for ressources) est donnée dans les sources.

On donne ici un exemple en .pdlx :

```
process test {
   wd conception
   wd redactionDoc
   wd developpement
   wd redactionTests

   ws s2s from conception to redactionDoc
   ws f2f from conception to redactionDoc
   ws f2s from conception to developpement
   ws s2s from conception to redactionTests
   ws f2f from developpement to redactionTests

   r concepteur (3)
   r developpeur (2)
   r machine (4)
   r redacteur (1)
   r testeur (2)

   set set_conception wd conception
      n concepteur(2) set set_conception
      n machine(2) set set_conception
   set set_redactionDoc wd redactionDoc
      n machine(1) set set_redactionDoc
      n redacteur(1) set set_redactionDoc
   set set_developpement wd developpement
      n developpeur(2) set set_developpement
      n machine(3) set set_developpement
   set set_redactionTests wd redactionTests
      n testeur(2) set set_redactionTests
      n machine(2) set set_redactionTests
}
```

Et pour un fichier en .petri :

```
petrinet petrinet1 {
   place place1
   place place2 (2)
   transition transition1
   arc r from place1 to transition1
   arc (2) from transition1 to place2
}
```

Chapitre 5

Conclusion

On a presque complété toute la chaine de transformation des modèles. Par manque de temps, je n'ai pas fait la partie concernant la vérification de modèle.

Ce projet m'a beaucoup intéressé, mais il semble qu'il manque un élément dans la chaine de transformation : La génération d'un modèle en .xmi à partir d'un modèle respectant la syntaxe établie avec Xtext. Ceci permettrait de rendre vraiment efficace la génération de modèle.