IDM Mini-projet rapport

Najmeddine Ayoub
Sijelmassi Idrissi Ilyass
Zougari Belkhayat Hamza

Parcours Systèmes de télécommunications
2021-2022

Table des matières

Table des figures

1 Introduction

L'objectif de ce mini-projet est la simulation d'un projet sous forme de métamodèle, capable de valider les modèles conceptuels. Plusieurs étapes seront alors abordées pour la production d'une chaîne de vérification de modèles de processus SimplePDL en général. C'est ainsi qu'on pourra vérifier si le processus est susceptible de se terminer ou pas.

2 Les métamodèles SimplePDL et PetriNet

2.1 Le métamodèle Simple PDL

Il s'agit d'un language de métamodèlisation très utile pour la description des modèles de processus, respectant plusieurs critères par définition tel la validité des noms du processus et de ses composant, l'unicité du nom des ressources, etc ... L'ajout des ressources dans le métamodèle SimplePDL fût un premier pas dans notre mini-ptojet, d'où la figure qui suit :

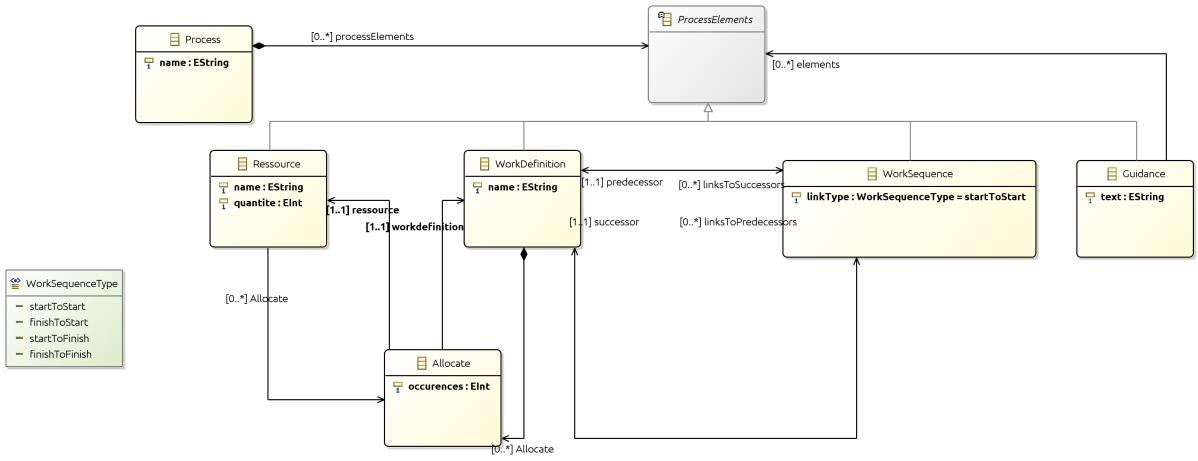


FIGURE 1 – Métamodèle SimplePDL

Notons aussi l'efficacité de l'outil ECore dans la réalisation graphique des métamodèles.

2.2 Le métamodèle PetriNet

Ci-dessous la représentation de notre réseau de Petri. Celui-ci permet une description dynamique du comportement des systèmes par rapport à des unités discrètes. On y trouve les principales composantes d'un métamodèle PetriNet :

— **arcs :** flèches permettant la liaison des **noeuds**.

— **noeud :** entité pouvant être soit une **place**, soit une **transition**.

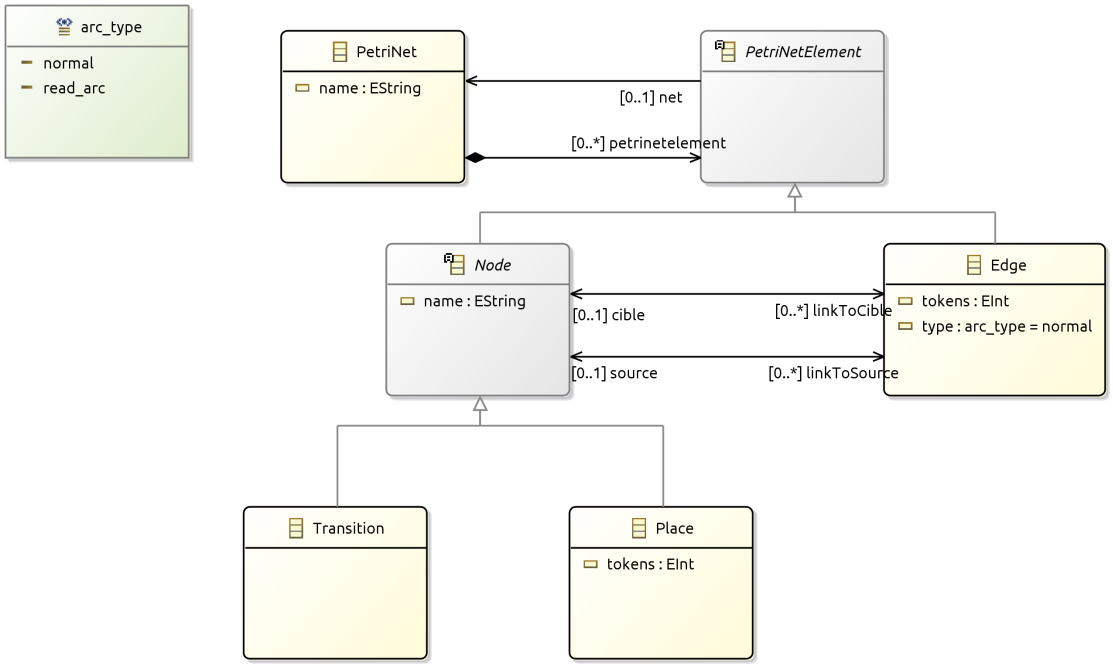


FIGURE 2 – Métamodèle PetriNet

3 Contraintes OCL

Malgré la précision des métamodèles précédents, il n'est pas possible de définir ou d'exprimer la totalité des contraintes à respecter par les modèles de processus. La description structurelle, sémantique et celle des ressources du métamodèle en question est alors réalisée en OCL.

3.1 Pour les ressources : contraintes ajoutées au SimplePDL

— Unicité du nom (identifiant) pour deux activités différentes d'un même processus.

— Un processus doit avoir un nom bien défini, à taille d'au moins 1, composé que de lettres, de chiffres ou de soulignés. Un chiffre ne peut pas être en première position.

— Deux activités identiques ne peuvent pas être liées.

— La quantité maximale demandée à une ressource doit être inférieur au nombre d'occurence pouvant être offerte par celle-ci.

3.2 Pour la sémantique statique : contraintes ajoutées au PetriNet

— Les jetons de **places** doivent être des entier naturel.

— Les jetons transportés par les **arcs** doivent être des entiers naturels.

— Deux **places** ou **transitions** ne peuvent pas être liées par un même **arc**.

```
9
10 context ProcessElements
11 def: process(): Process =
12     Process.allInstances()
13         ->select(p | p.processElements->includes(self))
14         ->asSequence()->first()
15
16 context WorkSequence
17 inv successorAndPredecessorInSameProcess('Activities not in the same process : '
18         + self.predecessor.name + ' in ' + self.predecessor.process().name+ ' and '
19         + self.successor.name + ' in ' + self.successor.process().name
20 ):
21     self.process() = self.successor.process()
22     and self.process() = self.predecessor.process()
23
24 context WorkDefinition
25 inv uniqNames: self.Process.processElements
26                 ->select(pe | pe.oclIsKindOf(WorkDefinition))
27                 ->collect(pe | pe.oclAsType(WorkDefinition))
28                 ->forAll(w | self = w or self.name <> w.name)
29
30 context WorkSequence
31 inv notReflexive: self.predecessor <> self.successor
32
33 context Process
34 inv nameIsDefined: if self.name.oclIsUndefined() then false
35                    else self.name.size() > 1
36                    endif
37
38 context Allocate
39 inv ressourceSuffisante:
40     self.occurences <= self.ressource.quantite
41
42 endpackage
```

FIGURE 3 – Contraintes OCL de SimplePDL

```
1  import 'PetriNet.ecore'
2
3  package petrinet
4
5
6  context PetriNet
7  inv validName('Invalid name: ' + self.name):
8      self.name.matches('[A-Za-z][A-Za-z0-9]*')
9
10
11 context Place
12 inv Initialize: self.tokens >= 0
13
14 context Edge
15 inv tokensMoving: self.tokens >= 1
16
17
18 context Edge
19 inv arcCoherence: self.cible.oclIsTypeOf(Place) <> self.source.oclIsTypeOf(Place)
20
21 context Place
22 inv nameIsDefined: if self.name.oclIsUndefined() then false
23                    else self.name <> ''
24                    endif
25
26
27 endpackage
```

FIGURE 4 – Contraintes OCL de PetriNet

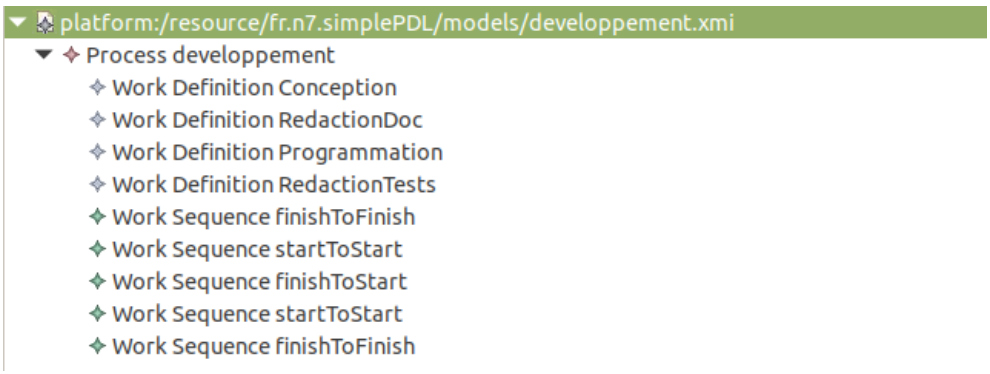
4 Transformations

4.1 Transformation modèle à modèle (M2M) avec EMF/Java

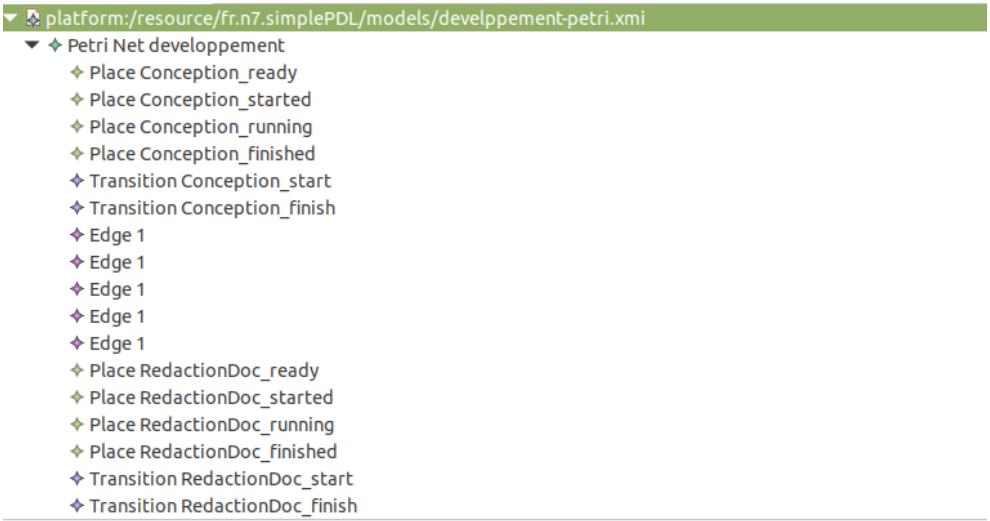
Il s'agit d'une transformation modèle de processus SimplePDL en un modèle de réseaux de Pétri en Java. Celle-ci passe par quelques étapes clés :

1. chargement des packages SimplePDL et PetriNet grâce aux fichiers ECore dont nous disposons.
2. rédaction du code Java assurant la transformation.
3. visualisation et validation du résultat.

Voici donc l'éditeur arborescent **avant la transformation :**



puis **après la transformation :**



4.2 Transformation modèle à texte (M2T)

Il s'agit d'une transformation modèle de réseaux de Petri dans la syntaxe concrète de Tina. Voici ci-dessous l'illustration du résultat obtenu :

```
net developpement
pl Conception_ready (1)
pl Conception_started (0)
pl Conception_running (0)
pl Conception_finished (0)
pl RedactionDoc_ready (1)
pl RedactionDoc_started (0)
pl RedactionDoc_running (0)
pl RedactionDoc_finished (0)
pl Programmation_ready (1)
pl Programmation_started (0)
pl Programmation_running (0)
pl Programmation_finished (0)
pl RedactionTests_ready (1)
pl RedactionTests_started (0)
pl RedactionTests_running (0)
pl RedactionTests_finished (0)
tr Conception_start Conception_ready -> Conception_started Conception_running
tr Conception_finish Conception_running -> Conception_finished
tr RedactionDoc_start RedactionDoc_ready Conception_started?1 -> RedactionDoc_started RedactionDoc_running
tr RedactionDoc_finish RedactionDoc_running Conception_finished?1 -> RedactionDoc_finished
tr Programmation_start Programmation_ready Conception_finished?1 -> Programmation_started Programmation_running
tr Programmation_finish Programmation_running -> Programmation_finished
tr RedactionTests_start RedactionTests_ready Conception_started?1 -> RedactionTests_started RedactionTests_running
tr RedactionTests_finish RedactionTests_running Programmation_finished?1 -> RedactionTests_finished
```

FIGURE 5 – Résultat de la transformation appliquée sur l'exemple du sujet

4.3 Syntaxes concrètes textuelles avec Xtext

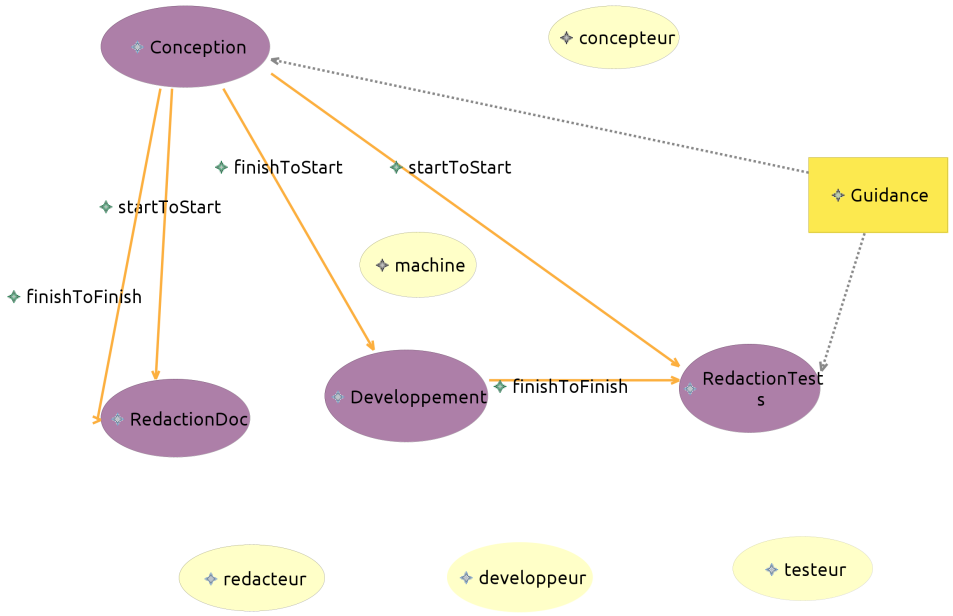
Les classes en Java n'étant pas toujours efficaces dans l'étude des modèles, il est des fois nécessaire de se référer à une syntaxe contrainte, réalisable avec l'outil Xtext à titre d'exemple. Voici ci-dessous un aperçu de modèle généré par cet outil dans le cadre de notre mini-projet :

```
// automatically generated by Xtext
grammar fr.n7.simplepdl.txt.PDL with org.eclipse.xtext.common.Terminals

import "http://simplepdl"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Process returns Process:
    {Process}
    'Process'
    name=EString
    '{'
        ('processElements' '{' processElements+=ProcessElements ( "," processElements+=ProcessElements)* '}' )?
    '}';

ProcessElements returns ProcessElements:
    WorkDefinition | WorkSequence | Guidance | Ressource;




EString returns ecore::EString:
    STRING | ID;

WorkDefinition returns WorkDefinition:
    {WorkDefinition}
    'WorkDefinition'
    name=EString
    '{'
        ('linksToPredecessors' '(' linksToPredecessors+=[WorkSequence|EString] ( "," linksToPredecessors+=[WorkSequence|EString])* ')' )?
        ('linksToSuccessors' '(' linksToSuccessors+=[WorkSequence|EString] ( "," linksToSuccessors+=[WorkSequence|EString])* ')' )?
        ('Allocate' '{' Allocate+=Allocate ( "," Allocate+=Allocate)* '}' )?
    '}';

WorkSequence returns WorkSequence:
    'WorkSequence'
    '{'
        'linkType' linkType=WorkSequenceType
        'predecessor' predecessor=[WorkDefinition|EString]
        'successor' successor=[WorkDefinition|EString]
    '}';
```

4.4 Syntaxes concrètes graphiques avec Sirius

Abordée pour les mêmes raisons de l'outil Xtext, il s'agit d'une version graphique de syntaxe concrète :

NB : nous avons trouvé une diffculté à ajouter les *liens allocate* entre les *WorkDefinition* et les *ressources*. Il s'agit éventuellement d'une panne technique dans l'application Eclipse.



4.5 Transformation LTL pour terminaison de processus

La logique temporelle LTL permet de décrire formellement le concept de terminaison des processus. Ceci est décrit par ce qu'on appelle une *transformation LTL pour terminaison de processus*. Nous allons pour cela créer un fichier LTL, engendré à partir d'un modèle SimplePDL. Voici alors le résulat de l'exécution :

```
1 op finished = a1_finished/\a2_finished/\a3_finished;
2 op running = a1_running/\a2_running/\a3_running;
3 op started = a1_started/\a2_started/\a3_started;
4 op ready = a1_ready/\a2_ready/\a3_ready;
5
6 [] (finished => dead);
7 [] <> dead;
8 [] (dead => finished);
9 - <> finished;
10 [] (finished => - <> ready /\ - <> running /\  started);
11 [] (running => ready /\ - <> finished /\ started);
12 [] (- <> started => - <> ready /\ - <> running /\ - <> finished);
13 [] (dead => finished);
```

5 Conclusion

Ce mini-projet nous a permis de déduire l'importance de la modélisation des projets et la possibilité de vérifier la cohérence de leur structure grâce aux notions vues en TP/TD. Nous avons donc déduit l'importance de telles représentations dans le monde du logiciel.