

## Chaîne de vérification de modèles de processus



Hamza Mouddene  
Faical Toubali Hadaoui

14 novembre 2020



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Les métamodèles SimplePDL et PetriNet</b>	<b>1</b>
2.1	SimplePDL . . . . .	1
2.2	Réseaux de Pétri . . . . .	4
2.3	Définition . . . . .	4
2.4	Schématisation . . . . .	4
2.5	Explication de l'exécution : . . . . .	4
2.6	Exemple de réseau de Petri : . . . . .	5
<b>3</b>	<b>Les contraintes OCL associés à ces métamodèles</b>	<b>8</b>
3.1	Modèle de processus . . . . .	8
3.1.1	Explication des contraintes ajoutées au SimplePDL : . . .	8
3.2	Réseau de Pétri . . . . .	9
3.2.1	Explication des contraintes principales ajoutées au réseau de Petri : . . . . .	9
<b>4</b>	<b>La transformation modèle à modèle</b>	<b>10</b>
4.1	En Java . . . . .	10
4.2	Application sur un modèle de procédé : . . . . .	10
4.3	En ATL . . . . .	13
4.3.1	Introduction . . . . .	13
4.4	Implantation . . . . .	13
<b>5</b>	<b>Les transformations modèle à texte en utilisant Acceleo</b>	<b>14</b>
5.1	La transformation d'un modèle de <i>Process</i> en <i>HTML</i> . . . . .	14
5.2	Template Acceleo de la transformation d'un modèle de <i>Process</i> en <i>ToDo</i> . . . . .	15
5.3	La transformation d'un réseau de Pétri en <i>Tina</i> . . . . .	15
5.4	Exemple de réseau de Petri : . . . . .	16
<b>6</b>	<b>Sirius</b>	<b>19</b>
6.1	Introduction . . . . .	19
6.2	Définition de la syntaxe graphique avec Sirius . . . . .	19
<b>7</b>	<b>Le modèle Xtext</b>	<b>20</b>
<b>8</b>	<b>Conclusion</b>	<b>21</b>
<b>9</b>	<b>Annexe</b>	<b>22</b>
9.1	Table des figures . . . . .	22

# 1 Introduction

Ce mini-projet consiste à produire une chaîne de vérification de modèles de processus SimplePDL dans le but de vérifier leur cohérence, en particulier pour savoir si le processus décrit peut se terminer ou non. Pour répondre à cette question, nous utilisons les outils de model-checking définis sur les réseaux de Petri au travers de la boîte à outils Tina. Il nous faudra donc traduire un modèle de processus en un réseau de Petri.

## 2 Les métamodèles SimplePDL et PetriNet

### 2.1 SimplePDL

Le SimplePDL est un langage de métamodélisation qui sert pour décrire des modèles de processus. Il existe deux formes de langages pour le SimplePDL : un langage simplifié de description des procédés de développement et un langage plus avancé résultant de l'ajout de la notion de ProcessElement comme généralisation de WorkDefinition (activité) et WorkSequence (dépendance).

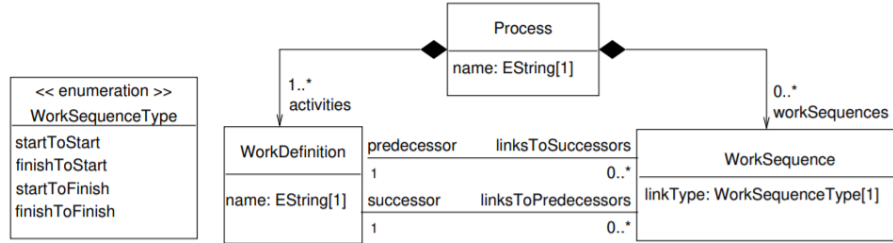


FIGURE 1 – Métamodèle simplifié de SimplePDL

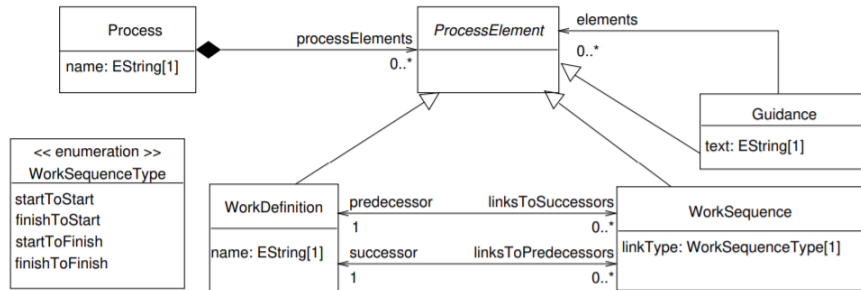


FIGURE 2 – Métamodèle avancé de SimplePDL

Dans le cadre d'Eclipse, Eclipse Modeling Project offre des outils pour manipuler des métamodèles, en particulier le métamodèle de SimplePDL qui en est conforme. Les principaux outils utilisés sont :

- Eclipse Modeling Framework (EMF) qui est une framework de modélisation, il fournit une infrastructure pour la manipulation des modèles, cette infrastructure permet aussi la génération de code et des applications à partir des modèles.
- ECore qui est aussi un métamodèle disposant d'un éditeur graphique pour manipuler des métamodèles.

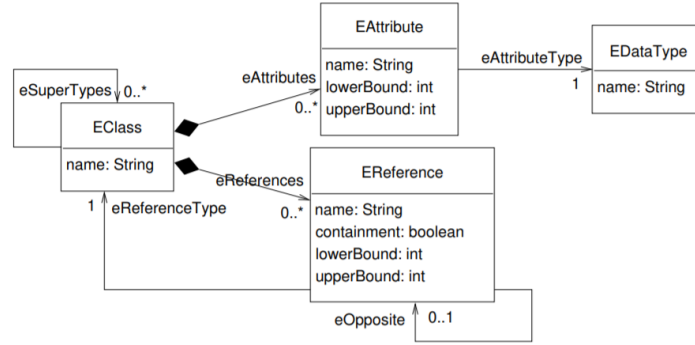


FIGURE 3 – Métamodèle simplifié d'ECore (Eclipse/EMF)

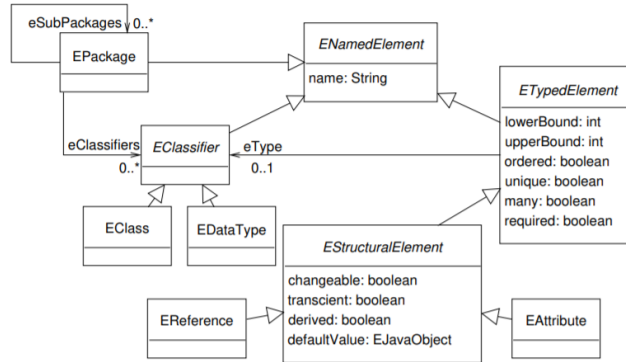


FIGURE 4 – Métamodèle avancé d'ECore (Eclipse/EMF)

[T1] : Un métamodèle de SimplePDL définit de différentes activités ( WorkDefinition ) et dépendances ( WorkSequence ). Cependant, afin qu'une activité soit réalisée, elle aura probablement disposer d'une ou plusieurs ressources pour assurer sa finalité. Une ressource contient des occurrences, une activité loue des occurrences d'une ressources au début de son execution, ces occurrences sont alors utilisés exclusivement par cette activité jusqu'à la fin de sa réalisation.

Afin de réaliser cette tâche, on aura besoin d'ajouter deux classes au métamodèle de SimplePDL :

- Ressource : une ressource est définie par son nom (EString) qui décrit son type et ses occurrences (Eint).
- Allocate : cette classe représente le nombre d'occurrences prises par une activité parmi les occurrences d'une ressource pour effectuer sa réalisation.

A noter qu'une activité (WorkDefinition) aura besoin éventuellement de plusieurs ressources, ainsi elle sera composer de plusieurs classes Allocate.

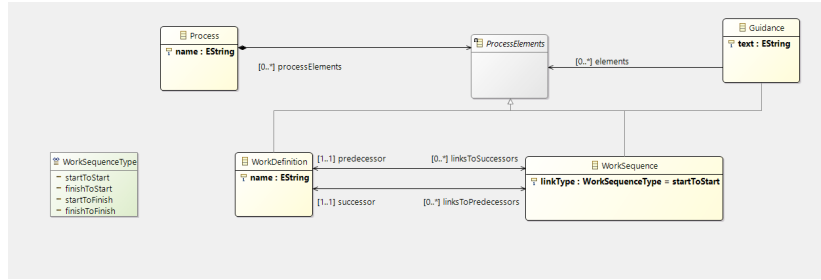


FIGURE 5 – Le métamodèle SimplePDL sans ressources ajoutées

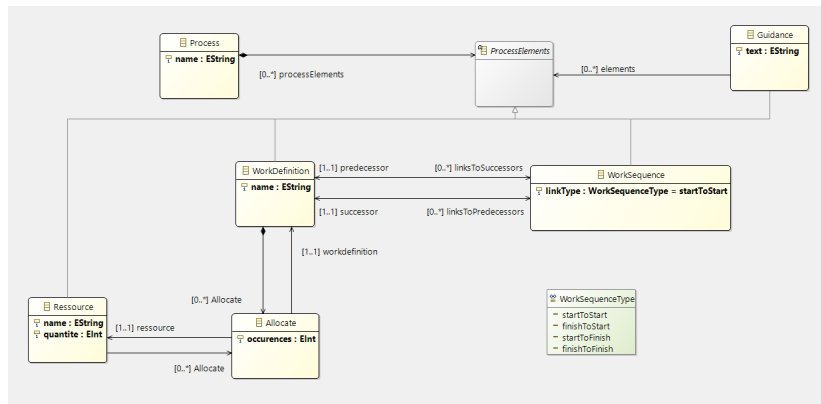


FIGURE 6 – Le métamodèle SimplePDL avec ressources ajoutées

## 2.2 Réseaux de Pétri

Le réseau de Petri est un métamodèle qui permet de décrire le comportement dynamiques des systèmes aux éléments discrets, à la base c'est un outil graphique mathématique qui s'inscrit dans le domaine large de la théorie des réseaux.

## 2.3 Definition

Le réseau de Petri est un tuple  $(S, T, F, M0, W)$  de composant de :

- $S$  définit une ou plusieurs places.
- $T$  définit une ou plusieurs transitions.
- $F$  définit un ou plusieurs arcs (flèches). Un arc ne peut pas être connecté entre 2 places ou 2 transitions ; plus formellement :

$$F \subseteq (S \times T) \cup (T \times S)$$

- $M0 : S \rightarrow \mathbb{N}$  appelé place initiale, où, pour chaque place  $s \in S$ , il y a  $n \in \mathbb{N}$  jetons.
- $W : F \rightarrow \mathbb{N}$  appelé ensemble d'arcs primaires, assignés à chaque arc  $f \in F$  un entier positif  $n \in \mathbb{N}$  qui indique combien de jetons sont consommés depuis une place vers une transition, ou sinon, combien de jetons sont produits par une transition et arrivent pour chaque place.

## 2.4 Schématisation

Graphiquement, un réseau de Petri est représenté par un graphe composé des places et transitions liés les uns aux autres grâce à des arcs. Les places contiennent des jetons ( Tokens ), ces jetons se transportent par le biais des transitions vers une ou plusieurs places, dans ce cas la transition est franchissable.

## 2.5 Explication de l'exécution :

L'évolution du réseau de Petri débute par l'exécution d'une transition, les jetons se déplacent en nombre de la place en entrée de la transition à la place en sa sortie. Ceci n'est valable que si la transition est franchissable, c'est-à-dire la place en entrée dispose d'un nombre de jetons supérieur ou égal aux nombres de jetons indiqués sur l'arc.

## 2.6 Exemple de réseau de Petri :

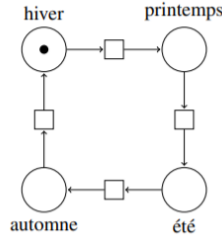


FIGURE 7 – Exemple de réseau de Petri : Evolution des saisons

[T2] : En se basant sur la définition du réseau de Petri, ses composants principaux et l'évolution de son execution, on peut élaborer un métamodèle du réseau de Petri grace aux outils ECore et EMF, pour ce faire, on peut se baser sur l'un des éléments suivants :

- Avec un simple éditeur texte d'ECore : Syntaxe purement textuelle
- Avec l'éditeur arborescent Ecore d'EMF : Forme Arborescente
- Avec l'éditeur graphique Ecore de EMF : En se basant sur la Palette contenant les outils de création des éléments ECore.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <ecore:EPackage xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="petrinet" nsURI="http://petrinet/" nsPrefix="petrinet">
4   <ecore:Classifiers xsi:type="ecore:EClass" name="PetriNet">
5     <ecore:StructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
6     <ecore:StructuralFeatures xsi:type="ecore:EReference" name="petrinetelement" upperBound="-1"
7       eType="#//PetriNetElement" containment="true"/>
8   </ecore:Classifiers>
9   <ecore:Classifiers xsi:type="ecore:EClass" name="Edge" eSuperTypes="#//PetriNetElement">
10     <ecore:StructuralFeatures xsi:type="ecore:EReference" name="cible" eType="#//Node"
11       eOpposite="#//Node/linkToCible"/>
12     <ecore:StructuralFeatures xsi:type="ecore:EReference" name="source" eType="#//Node"
13       eOpposite="#//Node/linkToSource"/>
14     <ecore:StructuralFeatures xsi:type="ecore:EAttribute" name="tokens" eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
15     <ecore:StructuralFeatures xsi:type="ecore:EAttribute" name="type" eType="#//arc_type"/>
16   </ecore:Classifiers>
17   <ecore:Classifiers xsi:type="ecore:EClass" name="Node" abstract="true" eSuperTypes="#//PetriNetElement">
18     <ecore:StructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EString"/>
19     <ecore:StructuralFeatures xsi:type="ecore:EReference" name="linkToSource" upperBound="-1"
20       eType="#//Edge" eOpposite="#//Edge/source"/>
21     <ecore:StructuralFeatures xsi:type="ecore:EReference" name="linkToCible" upperBound="-1"
22       eType="#//Edge" eOpposite="#//Edge/cible"/>
23   </ecore:Classifiers>
24   <ecore:Classifiers xsi:type="ecore:EClass" name="Place" eSuperTypes="#//Node">
25     <ecore:StructuralFeatures xsi:type="ecore:EAttribute" name="tokens" eType="ecore:EDatatype http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
26   </ecore:Classifiers>
27   <ecore:Classifiers xsi:type="ecore:EClass" name="Transition" eSuperTypes="#//Node">
28     <ecore:Classifiers xsi:type="ecore:EEnum" name="arc_type">
29       <ecore:Literal name="normal"/>
30       <ecore:Literal name="read_arc" value="1"/>
31     </ecore:Classifiers>
32   </ecore:Classifiers>
33   <ecore:Classifiers xsi:type="ecore:EClass" name="PetriNetElement" abstract="true">
34     <ecore:StructuralFeatures xsi:type="ecore:EReference" name="net" eType="#//PetriNet"/>
35   </ecore:Classifiers>
36 </ecore:EPackage>

```

FIGURE 8 – Editeur Textuelle ECore : Réseau de PetriNet



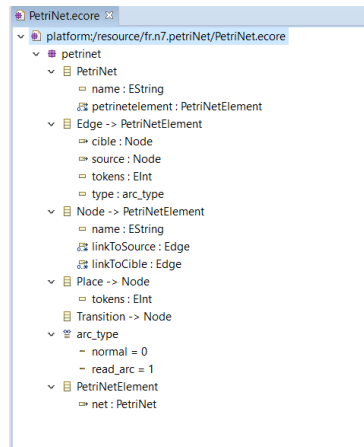


FIGURE 9 – Editeur Arborescent ECore : Réseau de PetriNet

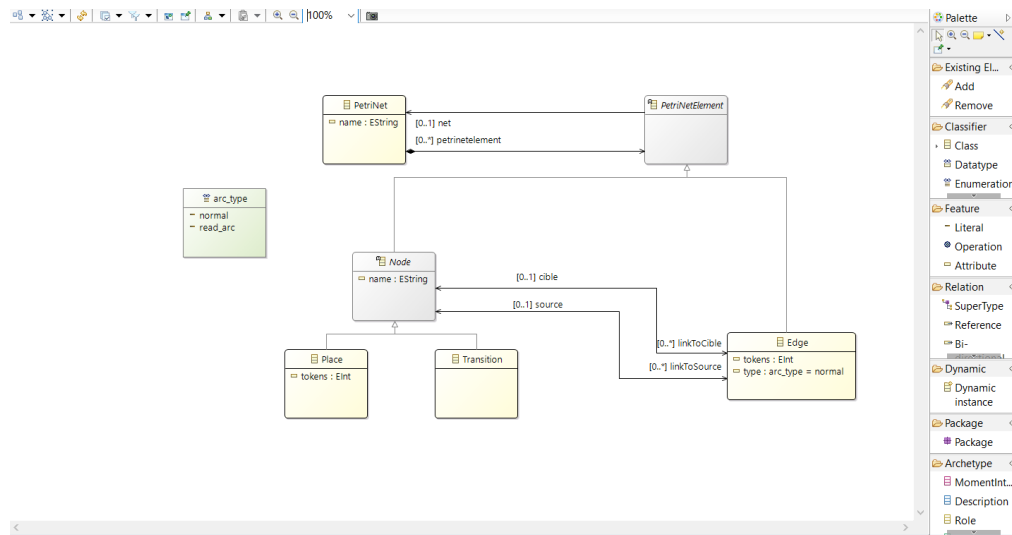


FIGURE 10 – Editeur Graphique ECore : Réseau de PetriNet

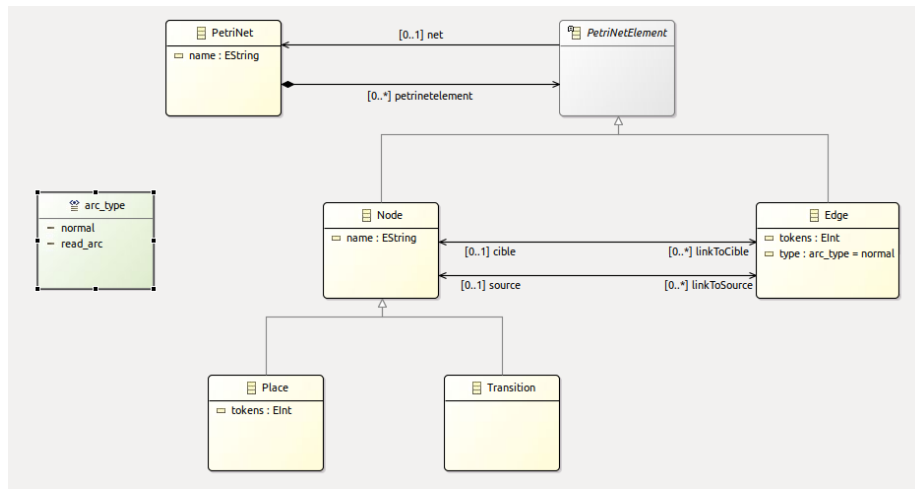


FIGURE 11 – Le métamodèle PetriNet

### 3 Les contraintes OCL associés à ces métamodèles

Nous avons utilisé Ecore pour définir un méta-modèle pour les processus. Cependant, le langage de méta-modélisation Ecore ne permet pas d'exprimer toutes les contraintes que doivent respecter les modèles de processus. Aussi, on complète la description structurelle, sémantique statique du méta-modèle réalisée en Ecore par des contraintes exprimées en OCL. Le méta-modèle Ecore et les contraintes OCL définissent la syntaxe abstraite du langage de modélisation considéré.

#### 3.1 Modèle de processus

```
@ SimplePDLocl 12
1  import 'SimplePDL.ecore'
2
3  package simplepdl
4
5  context Process
6  inv warningSeverity: false
7  inv withMessage('Explicit message in process ' + self.name + ' (withMessage)'): false
8
9  context Process
10 inv validName('Invalid name: ' + self.name);
11    self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
12
13 context ProcessElements
14 def: process(): Process =
15    Process.allInstances()
16    ->select(p | p.processElements->includes(self))
17    ->asSequence()->first()
18
19 context WorkSequence
20 inv successorAndPredecessorInSameProcess('Activities not in the same process : '
21    + self.predecessor.name + ' in ' + self.predecessor.process().name+ ' and '
22    + self.successor.name + ' in ' + self.successor.process().name
23 );
24 self.process() = self.successor.process()
25 and self.process() = self.predecessor.process()
26
27 context WorkDefinition
28 inv uniqNames: self.Process.processElements
29    ->select(pe | pe.ocIsKindOf(WorkDefinition))
30    ->collect(pe | pe.ocIsType(WorkDefinition))
31    ->forall(w | self = w or self.name <> w.name)
32
33
34 context WorkSequence
35 inv notReflexive: self.predecessor <> self.successor
36
37 context Process
38 inv nameIsDefined: if self.name.ocIsUndefined() then false
39    else self.name.size() > 1
40    endif
41
42 context Allocate
43 inv ressourceSuffisante:
44    self.occurrences <= self.ressource.quantite
45
46 endpackage
```

FIGURE 12 – Les contraintes OCL du métamodèle SimplePDL

##### 3.1.1 Explication des contraintes ajoutées au SimplePDL :

- *from line 27 to line 31* : Cette contrainte stipule que deux activités différentes d'un même processus ne peuvent pas avoir le même nom : Pour chaque activité *WorkDefinition* On fait le parcours de tous les *ProcessElements*, on sélectionne ceux du type *WorkDefinition* et on réalise le test manifestant que toute *WorkDefinition* autre que la présente possède un identifiant différent.
- *from line 34 to line 35* : Une dépendance ne peut pas être réflexive : Une *WorkSequence* ne peut pas lier deux activités identiques.
- *from line 37 to line 40* : Un *Process* doit avoir un nom bien définie : à savoir non nul, et dont la taille est supérieure strictement à 1.

- *from line 42 to line 44* : [T3] Contrainte sur l'élément *Allocate* : Une activité ne peut pas demander un nombre d'occurrences d'une ressource supérieur aux occurrences que peut offrir cette ressource à la base.

## 3.2 Réseau de Pétri

Comme pour le métamodèle SimplePDL, le métamodèle réseau de Petri crée ne représente que la syntaxe statique, on doit en ajouter alors des contraintes OCL, on définit ainsi la syntaxe abstraite du réseau de Petri.

```
PetriNet.ocl
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
28
```

FIGURE 13 – Les contraintes OCL du métamodèle PetriNet

### 3.2.1 Explication des contraintes principales ajoutées au réseau de Petri :

- *from line 11 to line 12* Toute *Place* doit avoir un nombre entier naturel de *Jetons*, en fait, ceci découle de la définition du réseau de Petri.
- *from line 14 to line 15* Le nombre de *Jetons* que transporte un arc (*Edge*) doit être un entier naturel.
- *from line 18 to line 19* Un arc ne peut pas lier deux *Places*, en fait, il lie une *Place* avec une *Transition*.

## 4 La transformation modèle à modèle

### 4.1 En Java

[T6] : La transformation modèle à modèle en Java qui consiste à transformer un modèle de processus en un modèle de réseaux de Pétri passe par plusieurs étapes :

- Chargement des packages SimplePDL et PetriNet afin de l'enregistrer dans le registre d'Eclipse.
- Configuration de l'input qu'on fournit au programme Java ainsi que l'output qu'il doit nous rendre.
- On récupère le premier élément du modèle process(élément à la racine), puis instancier la fabrique.
- On commence par construire le PetriNet, en convertissant les *workDefinitions* et *Resources* en *Places*, puis *workSequences* en *Arcs*.

A partir de ces étapes, on implémente un code SimplePDL2PetriNet.java qui réalise la transformation d'un modèle en SimplePDL en un modèle PetriNet. Ce code est fournie dans les livrables.

### 4.2 Application sur un modèle de procédé :

Un modèle de procédé est un modèle tiré de SPEM, norme de l'OMG. Il se compose de quatre activités *workDefinitions* qui sont :Conception, Programmation,RédactionDoc et RédactionTests et de quatre dépendances *workSequences* qui sont finishToFinish, finishToStart, startToStart et startToFinish. Ces dépendances *workSequences* permettent de préciser la nature de la dépendance sous la forme « étatToAction » qui précise l'état qui doit être atteint par l'activité source pour réaliser l'action sur l'activité cible.

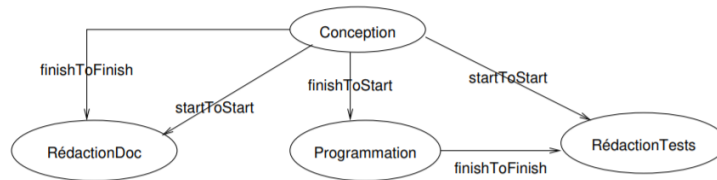


FIGURE 14 – Modèle de Procédé

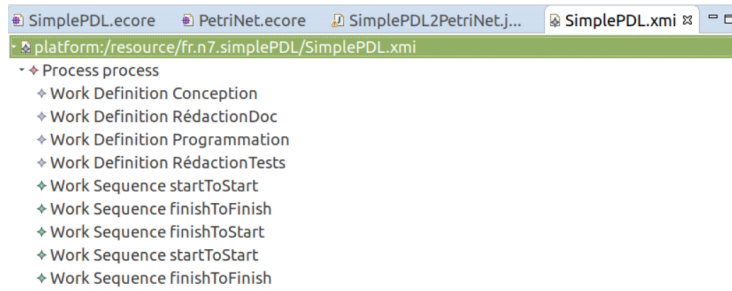


FIGURE 15 – Modèle de Procédé crée à partir du SimplePDL

[T8] : Test sur le modèle de procédé : L'exécution de la classe Java SimplePDL2PetriNet.java donne en sortie un modèle en extension .xml qui modélise la transformation du modèle de procédé en modèle de PetriNet. le résultat, modèle de sortie est le suivant :

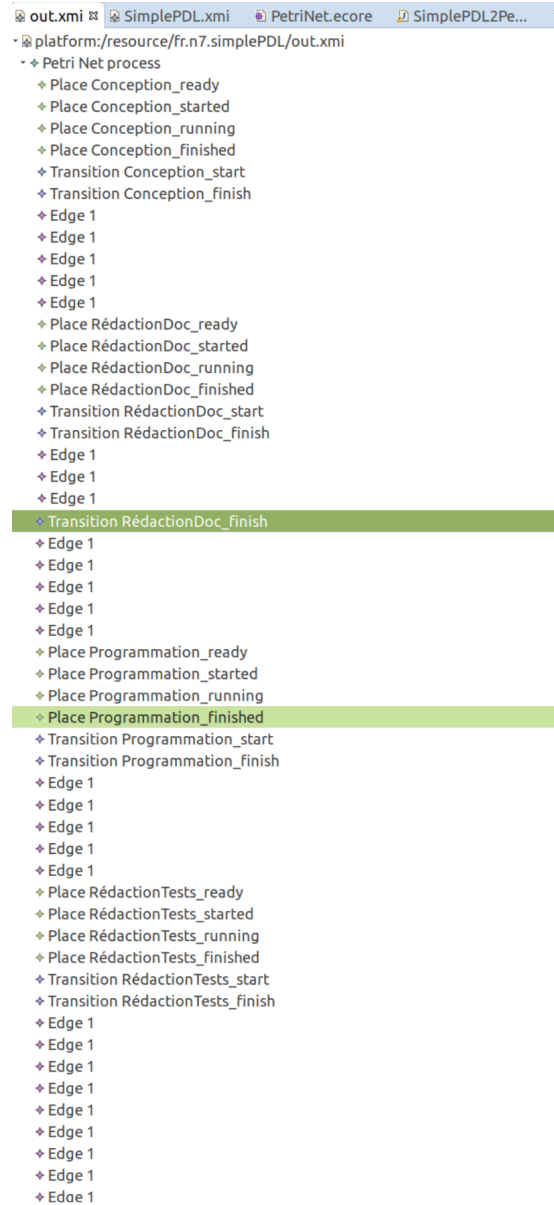


FIGURE 16 – Modèle de Procédé transformé en modèle de PetriNet

## 4.3 En ATL

### 4.3.1 Introduction

ATLAS Transformation Language (*ATL*) est un langage de transformation de modèles plusou moins inspiré par le standard QVT de l'OMG et développé au LINA à Nantes par l'équipe de Jean Bézivin.. Il est disponible en tant que plugin dans le projet Eclipse. ATL est un prototypeacadémique de composant de transformation de modèles du projet Eclipse Modeling. ATL se compose :

- D'un langage de transformation (déclaratif et impératif) ;
- D'un compilateur et d'une machine virtuelle ;
- D'un IDE s'appuyant sur Eclipse ;

Nous allons utiliser ATL pour convertir un SimplePDL en PetriNet de façon beaucoup plus efficace. J'utiliserai exclusivement ce langage dans la suite du projet et de ce rapport.

## 4.4 Implantation

La transformation modèle à modèle en *ATL* consiste à transformer un modèle de processus en un modèle de réseau de Pétri. Cette transformation passe par les mêmes étapes que la transformation *Java*.

La transformation en *ATL* est plus efficace que la transformation en *Java*.

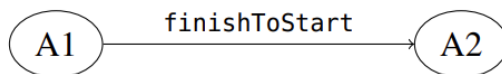


FIGURE 17 – Exemple de processus

Ce processus passe par plusieurs étapes qui sont les suivantes :

- Traduire un Process en un PetriNet de même nom.
- Convertir toutes les *workDefinitions* en réseaux de pétri en 5 arcs, 2 transitions et 4 places. Relier le tout correctement et l'ajouter au réseau de Pétri.
- Convertir toutes les *WorkSequences* en un arc qu'on relie judicieusement aux *workDefinitions* appropriées. Ajouter l'arc au réseau.
- Convertir toutes les ressources en places et les relier convenablement au reste du réseau. Ajouter tous les éléments créés au réseau.

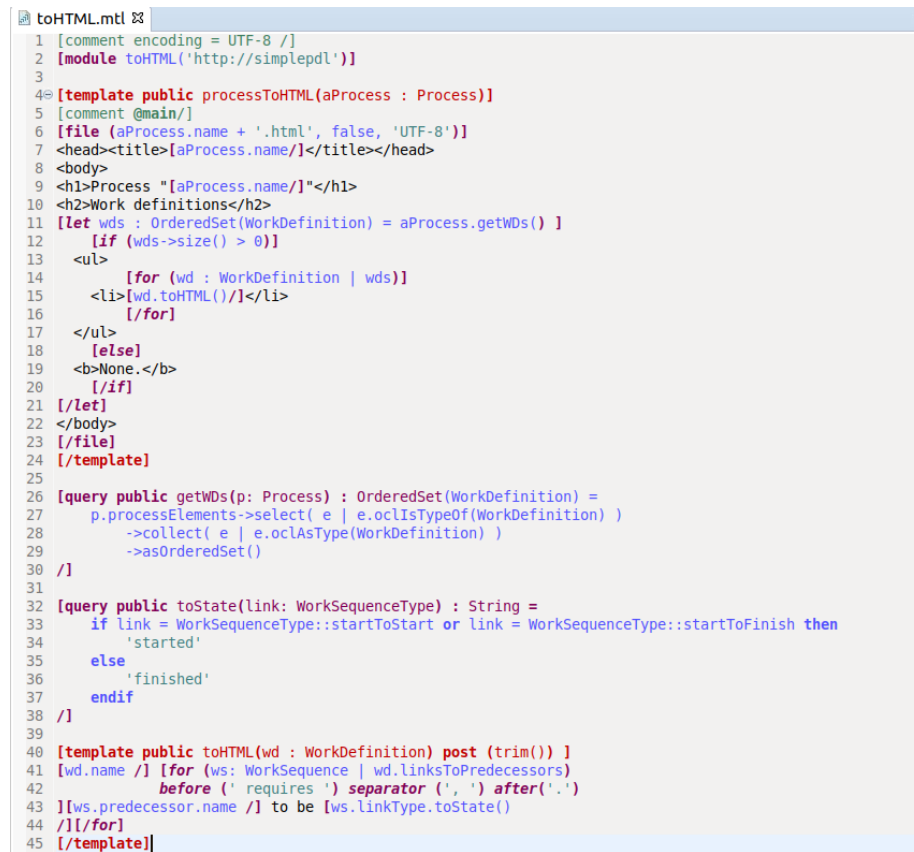


## 5 Les transformations modèle à texte en utilisant Aceleo

Dans cette partie, on va s'intéresser aux transformations d'un modèle en un texte. On parle de transformation modèle vers texte (M2T) où Nous allons utiliser l'outil Aceleo.

### 5.1 La transformation d'un modèle de *Process* en *HTML*

Dans un premier temps, on va réaliser une transformation d'un modèle de *Process* en *HTML*.



```
1 [comment encoding = UTF-8 /]
2 [module toHTML('http://simplepdl')]
3
4 [template public processToHTML(aProcess : Process)]
5 [comment @main/]
6 [file (aProcess.name + '.html', false, 'UTF-8')]
7 <head><title>[aProcess.name/]</title></head>
8 <body>
9 <h1>Process "[aProcess.name/]"</h1>
10 <h2>Work definitions</h2>
11 [let wds : OrderedSet(WorkDefinition) = aProcess.getWds() ]
12 [if (wds->size() > 0)]
13 <ul>
14 [for (wd : WorkDefinition | wds)]
15 <li>[wd.toHTML()/]</li>
16 [/for]
17 </ul>
18 [else]
19 <b>None.</b>
20 [/if]
21 [/let]
22 </body>
23 [/file]
24 [/template]
25
26 [query public getWds(p: Process) : OrderedSet(WorkDefinition) =
27 p.processElements->select( e | e.ocliIsTypeOf(WorkDefinition) )
28 ->collect( e | e.ocliAsType(WorkDefinition) )
29 ->asOrderedSet()
30 /]
31
32 [query public toState(link: WorkSequenceType) : String =
33 if link = WorkSequenceType::startToStart or link = WorkSequenceType::startToFinish then
34 'started'
35 else
36 'finished'
37 endif
38 /]
39
40 [template public toHTML(wd : WorkDefinition) post (trim()) ]
41 [wd.name /] [for (ws: WorkSequence | wd.linksToPredecessors)
42 before (' requires ' separator (', ' ) after(' ' )
43 ][ws.predecessor.name /] to be [ws.linkType.toState()
44 /][/for]
45 [/template]
```

FIGURE 18 – Template Aceleo de la transformation d'un modèle en HTML : toHTML.mtl

Ce fichier contient dans la première *Template* le texte qui sera généré en *HTML*, avec plusieurs variables. On obtient ces variables en récupérant la racine

du modèle *Process* où on l'exploite dans les *query* pour obtenir les *workdDefinitions*, *workSequence*, etc...

Dans la dernière *template*, on convertit le résultat des *query* en *HTML*.

## 5.2 Template Acceleo de la transformation d'un modèle de *Process* en *ToDot*

Maintenant, on va s'intéresser plus à la transformation d'un modèle de *Process* en *.Dot*.

```

1 [comment encoding = UTF-8 /]
2 [module toDot('http://simplepdl')]
3
4 [template public processToDot(aProcess : Process)]
5 [comment @main/]
6 [file (aProcess.name + '.dot', false, 'UTF-8')]
7 digraph [aProcess.name/] {
8   [let ws : OrderedSet(WorkSequence) = aProcess.getWs() ]
9   [for (w : WorkSequence | ws)]
10    [w.predecessor.name/] -> [w.successor.name/] [ '/' arrowhead=vee label=[w.linkType.toState()/] ]
11  [/for]
12 [/let]
13 }
14 [/file]
15 [/template]
16
17 [query public getWs(p: Process) : OrderedSet(WorkSequence) =
18   p.processElements->select( e | e.ocIsTypeOf(WorkSequence) )
19   ->collect( e | e.ocAsType(WorkSequence) )
20   ->asOrderedSet()
21 /]
22
23 [template public toState(link: WorkSequenceType) post (trim()) ]
24 [if (link = WorkSequenceType::startToStart)] s2s
25 [elseif (link = WorkSequenceType::finishToFinish)] f2f
26 [elseif (link = WorkSequenceType::finishToStart)] f2s
27 [else] s2f
28 [/if]
29 [/template]

```

FIGURE 19 – Transformation d'un modèle en *.dot* : *toDot.mtl*

## 5.3 La transformation d'un réseau de Pétri en *Tina*

[T9] : Nous allons s'intéresser maintenant à la transformation modèle à texte d'un modèle de réseau de Petri, la syntaxe textuelle voulue est celle utilisé par les outils de *Tina*, à savoir la syntaxe en extension *.net* . Ainsi, La boîte à outils *Tina* va nous permettre ensuite de visualiser graphiquement le modèle et de le simuler avec l'outil *nd* (*Net Draw*). La template Acceleo de la transformation d'un réseau de Pétri en *tina* est fournie ci-dessous.

Pour valider le bon fonctionnement de cette template on utilisera un modèle schématisant l'évolution des saisons fournie dans la figure suivante.

## 5.4 Exemple de réseau de Petri :

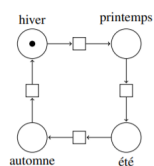


FIGURE 20 – Exemple de réseau de Petri : Evolution des saisons

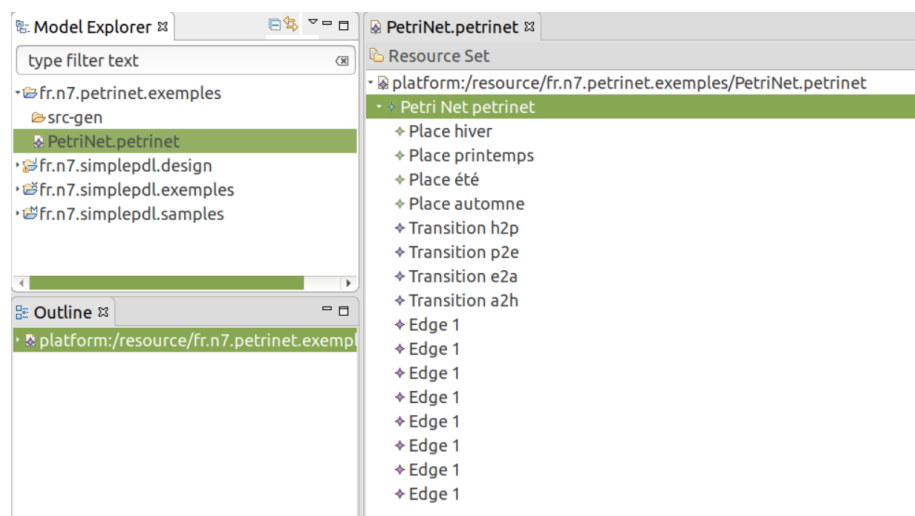


FIGURE 21 – Modèle de saisons en PetriNet

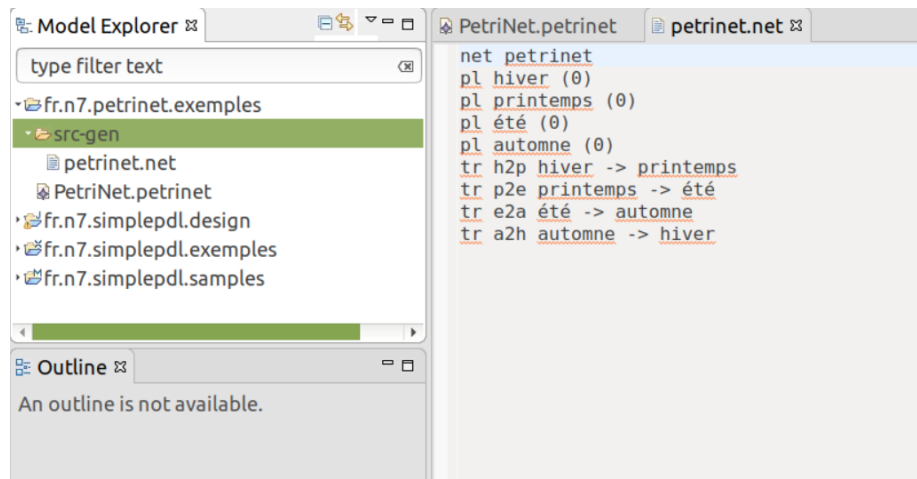


FIGURE 22 – Transformation du modèle de saisons en syntaxe textuelle de Tina

```

1  [comment encoding = UTF-8 /]
2  [module toTina('http://petrinet/')]
3
4
5  [template public petrinetToTina(aPetriNet : PetriNet)]
6  [comment @main/]
7  [file (aPetriNet.name + '.net', false, 'UTF-8')]
8  net [aPetriNet.name/]
9  [let places : OrderedSet(Place) = aPetriNet.getPlaces() ]
10     [for (p : Place | places)]
11     pl [p.name/] ([p.tokens/])
12     [/for]
13 [/let]
14 [let Trs : OrderedSet(Transition) = aPetriNet.getTransitions() ]
15     [for (t : Transition | Trs)]
16     tr [t.name/] [lesArcsSource(t.getSources()/)] -> [lesArcsDestination(t.getDestinatins()/)]
17     [/for]
18 [/let]
19 [/file]
20 [/template]
21
22 [query public getPlaces(p: PetriNet) : OrderedSet(Place) =
23     p.petrinetelement->select( e | e.ocIsTypeOf(Place) )
24     ->collect( e | e.ocAsType(Place) )
25     ->asOrderedSet()
26 /]
27
28 [query public getTransitions(p: PetriNet) : OrderedSet(Transition) =
29     p.petrinetelement->select( e | e.ocIsTypeOf(Transition) )
30     ->collect( e | e.ocAsType(Transition) )
31     ->asOrderedSet()
32 /]
33
34 [query public getSources(t: Transition) : OrderedSet(Edge) =
35     t.linkToCible->select( e | e.ocIsTypeOf(Edge) )
36     ->collect( e | e.ocAsType(Edge) )
37     ->asOrderedSet()
38 /]
39
40 [query public getDestinatins(t: Transition) : OrderedSet(Edge) =
41     t.linkToSource->select( e | e.ocIsTypeOf(Edge) )
42     ->collect( e | e.ocAsType(Edge) )
43     ->asOrderedSet()
44 /]
45
46 [template public lesArcsSource(arcs : OrderedSet(Edge)) post (trim()) ]
47     [for (a : Edge | arcs)][a.source.name/][if (a.type = arc_type::read_arc)][a.tokens/][elseif (a.tokens > 1)][a.tokens
48 [/template]

```

FIGURE 23 – Template Acceleo de la transformation d'un réseau de Pétri en tina : toTina.mtl

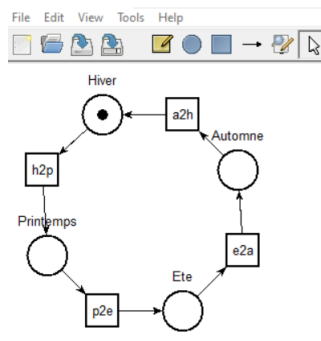


FIGURE 24 – Visualisation graphique de la syntaxe générée par toTina.mtl par l'outil Net Draw de Tina

## 6 Sirius

### 6.1 Introduction

À l'instar d'une syntaxe concrète textuelle, une syntaxe concrète graphique fournit un moyen de visualiser et/ou éditer plus agréablement et efficacement un modèle. Nous allons utiliser l'outil Sirius développé par les sociétés Obeo et Thales, et basé sur les technologies Eclipse Modeling comme EMF et GMF. Il permet de définir une syntaxe graphique pour un langage de modélisation décrit en Ecore et d'engendrer un éditeur graphique intégré à Eclipse.

### 6.2 Définition de la syntaxe graphique avec Sirius

Nous avons créé un projet de nature Sirius qui contient un modèle conforme à SimplePDL qui servira à tester la syntaxe graphique.

Nous avons mis en place le modèle de description de la syntaxe graphique, qui consiste à initier le modèle de description de la syntaxe graphique souhaitée et l'utiliser avec le modèle de test, ainsi que définir la définition de la partie graphique de l'éditeur, afin d'afficher sur l'éditeur les différents éléments de nos modèles.

Finalement, nous avons défini la palette pour manipuler le modèle au travers des objets graphiques de cette vue. Ces outils sont regroupés en sections.

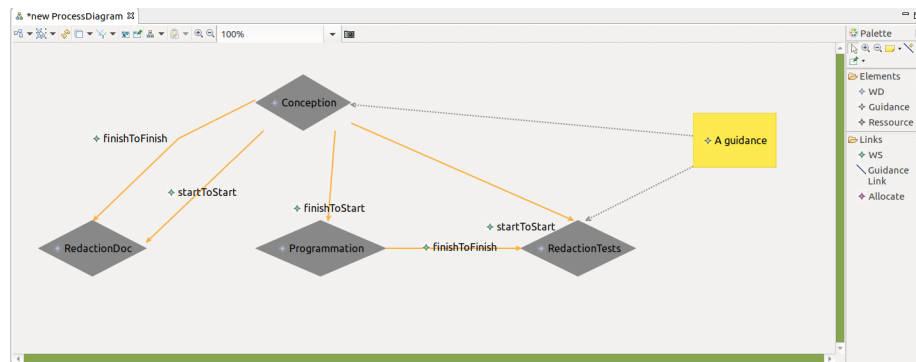
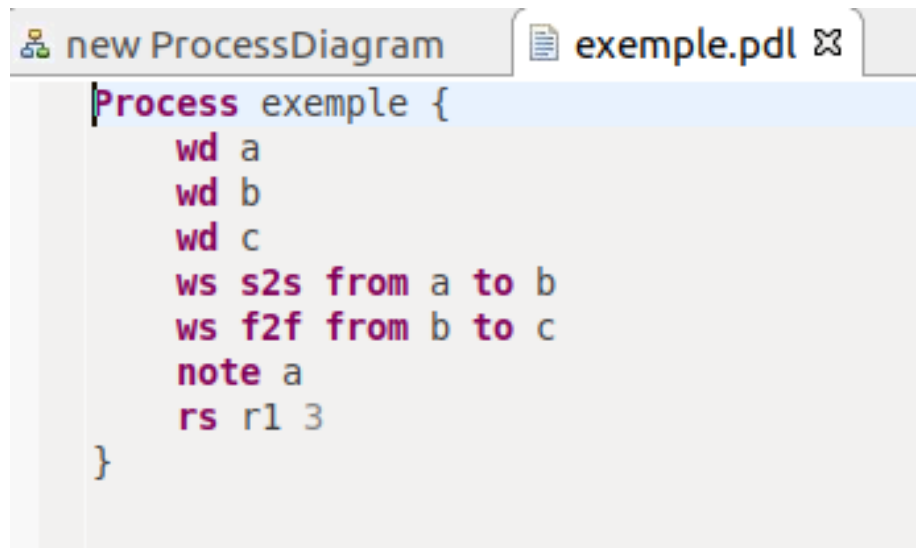


FIGURE 25 – Le modèle généré par l'éditeur graphique

## 7 Le modèle Xtext

La syntaxe abstraite d'un DSML (exprimée en Ecore ou un autre langage de métamodélisation) ne peut pas être manipulée directement. Le projet EMF d'Eclipse permet d'engendrer un éditeur arborescent et les classes Java pour manipuler un modèle conforme à un métamodèle. Cependant, ce ne sont pas des outils très pratiques lorsque l'on veut saisir un modèle. Aussi, il est souhaitable d'associer à une syntaxe abstraite une ou plusieurs syntaxes concrètes pour faciliter la construction et la modification des modèles. Ces syntaxes concrètes peuvent être textuelles ou graphiques. Pour la définition des syntaxes concrètes textuelles, nous utiliserons l'outil Xtext de openArchitectureWare (oAW). Xtext fait partie du projet TMF (Textual Modeling Framework). Il permet non seulement de définir une syntaxe textuelle pour un DSL mais aussi de disposer, au travers d'Eclipse, d'un environnement de développement pour ce langage, avec en particulier un éditeur syntaxique (coloration, complétion, outline, détection et visualisation des erreurs, etc). Xtext s'appuie sur un générateur d'analyseurs descendants récursifs (LL(k)).



```
new ProcessDiagram | exemple.pdl
Process exemple {
    wd a
    wd b
    wd c
    ws s2s from a to b
    ws f2f from b to c
    note a
    rs r1 3
}
```

FIGURE 26 – Le code PDL

## 8 Conclusion

Ce projet nous a dévoilé l'importance de la modélisation dans la résolution des problèmes ainsi que la maîtrise des processus de développement logiciel en s'appuyant sur des techniques de modélisation, de conception, de développement et de gestion de projet. Ces techniques doivent permettre la conception d'applications modernes, nécessitant l'intégration de composants hétérogènes.



## 9 Annexe

### 9.1 Table des figures

1	Métamodèle simplifié de SimplePDL . . . . .	1
2	Métamodèle avancé de SimplePDL . . . . .	1
3	Métamodèle simplifié d'ECore (Eclipse/EMF) . . . . .	2
4	Métamodèle avancé d'ECore (Eclipse/EMF) . . . . .	2
5	Le métamodèle SimplePDL sans ressources ajoutées . . . . .	3
6	Le métamodèle SimplePDL avec ressources ajoutées . . . . .	3
7	Exemple de réseau de Petri : Evolution des saisons . . . . .	5
8	Editeur Textuelle ECore : Réseau de PetriNet . . . . .	5
9	Editeur Arborescent ECore : Réseau de PetriNet . . . . .	6
10	Editeur Graphique ECore : Réseau de PetriNet . . . . .	6
11	Le métamodèle PetriNet . . . . .	7
12	Les contraintes OCL du métamodèle SimplePDL . . . . .	8
13	Les contraintes OCL du métamodèle PetriNet . . . . .	9
14	Modèle de Procédé . . . . .	10
15	Modèle de Procédé crée à partir du SimplePDL . . . . .	11
16	Modèle de Procédé transformé en modèle de PetriNet . . . . .	12
17	Exemple de processus . . . . .	13
18	Template Acceleo de la transformation d'un modèle en HTML : toHTML.mtl . . . . .	14
19	Transformation d'un modèle en .dot : toDot.mtl . . . . .	15
20	Exemple de réseau de Petri : Evolution des saisons . . . . .	16
21	Modèle de saisons en PetriNet . . . . .	16
22	Transformation du modèle de saisons en syntaxe textuelle de Tina	17
23	Template Acceleo de la transformation d'un réseau de Pétri en tina : toTina.mtl . . . . .	18
24	Visualisation graphique de la syntaxe générée par toTina.mtl par l'outil Net Draw de Tina . . . . .	18
25	Le modèle généré par l'éditeur graphique . . . . .	19
26	Le code PDL . . . . .	20