Rapport : Mini projet IDM

HUANG Julien
AFKER Samy

Département Sciences du Numérique - Deuxième année
2023-2024

Table des matières

Table des figures

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.

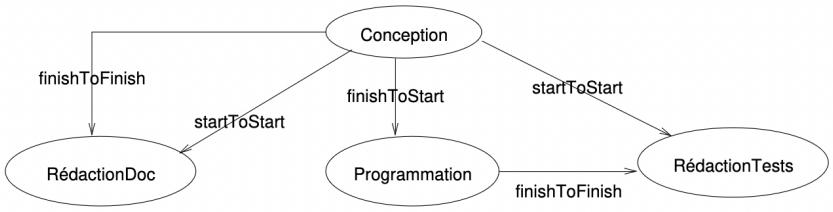


FIGURE 1 – Exemple de modèle de procédé

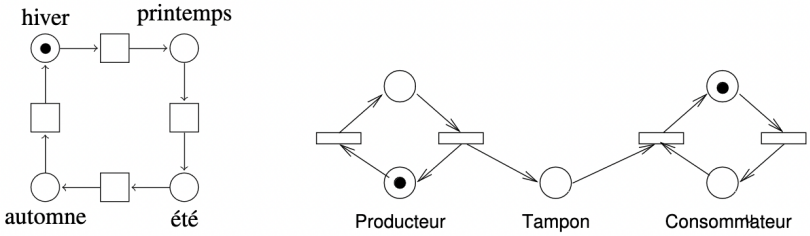


FIGURE 2 – Exemples de réseaux de Petri

2 Métamodèle de SimplePDL et des réseaux de Petri

2.1 SimplePDL

La figure 3 donne le métamodèle du langage SimplePDL, un langage très simplifié de description des procédés de développement. Ce métamodèle est conforme à EMOF/Ecore. Il a été dessiné en utilisant les conventions traditionnellement utilisées qui sont empruntées au diagramme de classe UML.

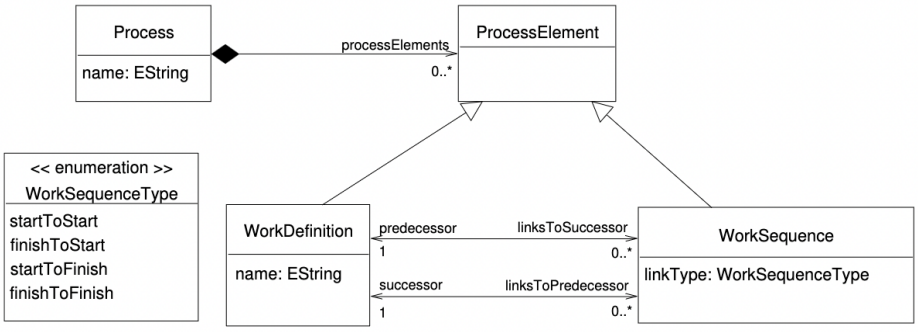


FIGURE 3 – Métamodèle initial de SimplePDL

— La notion de ProcessElement a été ajoutée comme généralisation de WorkDefinition et WorkSequence. Un processus est donc un ensemble d'éléments de processus qui sont soit des activités, soit des dépendances. Ce qui permet de factoriser des éléments et peut simplifier des choses si l'on decider de rajouter de nouveaux ProcessELements.

— la notion de Guidance a été également ajoutée. C'est l'équivalent d'une annotation UML : elle permet d'associer un texte à plusieurs éléments d'un processus.

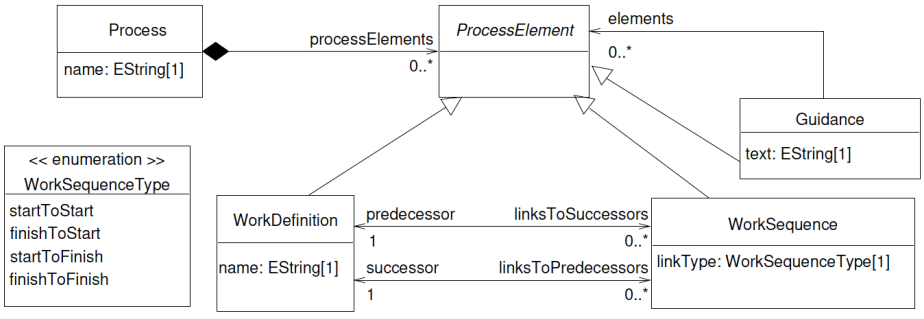


FIGURE 4 – Métamodèle de SimplePDL sans les ressources

Pour réaliser une activité, des ressources peuvent être nécessaires. Une ressource peut correspondre à un acteur humain, un outil ou tout autre élément jouant un rôle dans le déroulement de l'activité.

Ici, nous nous intéressons simplement aux types de ressources nécessaires et au nombre d'occurrences d'un type de ressource. Par exemple, il peut y avoir deux développeurs, trois machines, un bloc-note, etc. Un type de ressource sera seulement caractérisé par son nom et la quantité d'occurrences de celle-ci.

Pour pouvoir être réalisée, une activité peut nécessiter plusieurs ressources (éventuellement aucune).

Pour se faire il faut rajouter de nouvelles classes :

— Ressource : une ressource est définit par son nom (Estring) et sa quantité (Eint).

— Gestionnaire de ressource (GestRessource) : Représenté par la quantité qui est nécessaire (Eint) pour réaliser une activité (WorkDefinition).

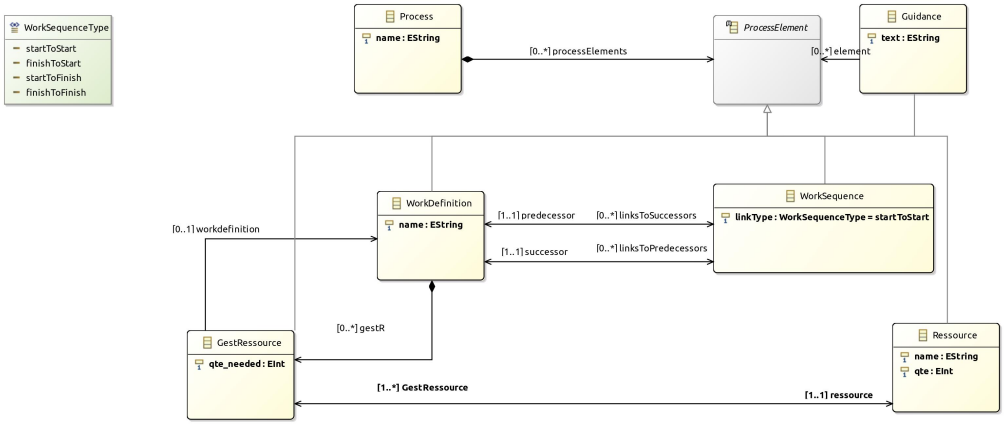


FIGURE 5 – Métamodèle de SimplePDL

2.2 Réseaux de Pétri

Un réseau de Petri se représente par un graphe orienté composé d'arcs reliant des places et des transitions.

Deux places ne peuvent pas être reliées entre elles, ni deux transitions. Les places peuvent contenir des jetons. La distribution des jetons dans les places est appelée le marquage du réseau de Petri.

Les entrées d'une transition sont les places desquelles part une flèche pointant vers cette transition, et les sorties d'une transition sont les places pointées par une flèche ayant pour origine cette transition.

La figure 2 propose quelques exemples de réseaux de Petri.

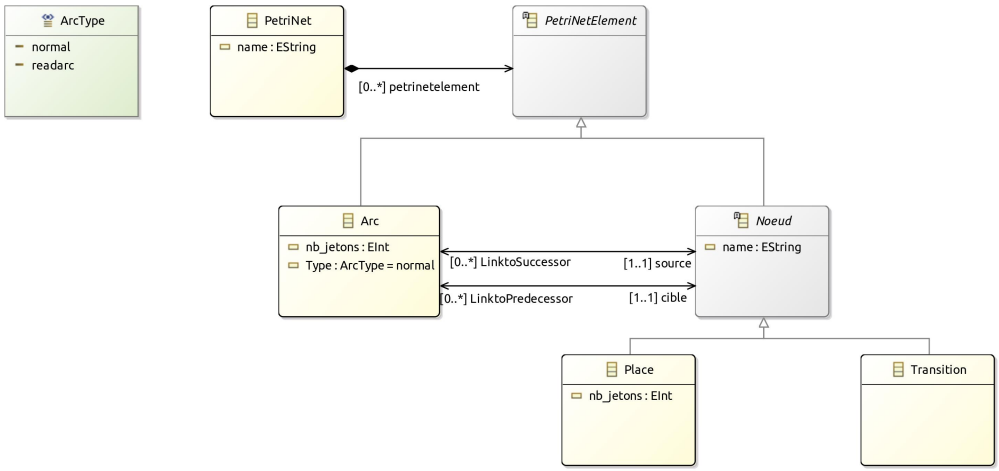


FIGURE 6 – Métamodèle de PetriNet

un **PetriNet** est composé de **PetriElement** qui est soit un **Arc** ou un **Noeud** (qui est lui même soit soit une **Place** soit une **Transitionn**)

— PetriNet : le réseau de Pétri est définit par son titre (Estring)

— Arc : Est définit par le nombre de jeton (Eint) nécessaire pour effectuer une transition, et son type (par défaut c'est un arc "normale", sinon elle peut être un read ard)

— Noeud : Est soit une place ou une transition, et définit par son nom (Estring)

— Place : définit par son nombre de jeton (Eint) (0 par défaut)

Le choix de crée une classe intermédiaire Noeud permet de factoriser de liaisons avec la classe Arc. Cela évite de faire deux fois la même chose pour les relations : Place et Arc et Transition et Arc.

3 Les contraintes statiques : OCL

Certaines contraintes sur les modèles de processus ont pu être exprimées. C'est par exemple le cas de celles qui concernent les multiplicités.

Cependant, le langage de méta-modélisation (Ecore ou EMOF) ne permet pas d'exprimer toutes les contraintes que doivent respecter les modèles de processus. Aussi, on complète la description structurelle du méta-modèle 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 Contraintes sur un modèle de processus

```
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  inv errorSeverity: null
9
10 inv validName('Invalid name: ' + self.name):
11     self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
12
13 context ProcessElement
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.oclIsKindOf(WorkDefinition))
30                 ->collect(pe | pe.oclAsType(WorkDefinition))
31                 ->forAll(w | self = w or self.name <> w.name)
32
33 context WorkSequence
34 inv notReflexive: self.predecessor <> self.successor
35
36 context Process
37 inv nameIsDefined: if self.name.oclIsUndefined() then false
38                    else self.name.size() > 1
39                    endif
40
41 endpackage
```

FIGURE 7 – Contraintes OCL sur le modèle des processus

— **Contexte :** Process (*lignes 1-11/ 36-39*)

  — **Contrainte 'warningSeverity' :** La sévérité d'avertissement est désactivée.
  — **Contrainte 'withMessage' :** Message explicite dans le processus avec désactivation.
  — **Contrainte 'errorSeverity' :** La sévérité d'erreur est nulle.
  — **Contrainte 'validName' :** le nom d'une activité ne doit être composé que de lettres, chiffres ou soulignés, un chiffre ne peut pas être première position.
  — **Contrainte 'nameIsDefined' :** le nom d'une activité doit être composé d'au moins deux caractères.

— **Contexte :** ProcessElement (*lignes 13-17*)

  — **Définition 'process' :** Fonction renvoyant le processus auquel l'élément appartient.

— **Contexte :** WorkSequence (*lignes 19-25*)

  — **Contrainte ‘successorAndPredecessorInSameProcess‘ :** Les activités prédécesseur et successeur doivent être dans le même processus.

— **Contexte :** WorkDefinition (*lignes 27-31*)

  — **Contrainte ‘uniqNames‘ :** Deux activités différentes d’un même processus ne peuvent pas avoir le même nom.

— **Contexte :** WorkSequence (*lignes 33-34*)

  — **Contrainte ‘notReflexive‘ :** Un séquence de travail ne peut pas être réflexive.

3.2 Contraintes sur un modèle de réseau de Pétri

```
1  import 'PetriNet.ecore'
2
3  package petrinet
4
5  context PetriNet
6
7  inv validNamePetriNet('Invalid name: ' + self.name):
8      self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
9
10 inv uniqName('Noms en double'):
11     self.petrinetelement
12     ->select(pe|pe.oclIsKindOf(Noeud))
13     ->collect(pe|pe.oclAsType(Noeud))
14     ->forAll(w1,w2 | (w1 = w2) or (w1.name <> w2.name))
15
16 context Noeud
17 inv validNameNoeud('Invalid name: ' + self.name):
18     self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
19
20 context Arc
21 inv nbJetonPositifArc('Nombre de jetons invalide (<1): ' + self.nb_jetons.toString()):
22     self.nb_jetons >= 1
23
24 inv ValidLink('Lien invalide :' + self.source.oclType().name + '->' + self.cible.oclType().name):
25     (self.source.oclIsTypeOf(Place) and self.cible.oclIsTypeOf(Transition)) or
26     (self.source.oclIsTypeOf(Transition) and self.cible.oclIsTypeOf(Place))
27
28 context Place
29 inv nbJetonPositifNoeud('Nombre de jetons invalide (<0): ' + self.nb_jetons.toString()):
30     self.nb_jetons >= 0
31
32 endpackage
```

FIGURE 8 – Contraintes OCL sur le réseau de Pétri

— **Contexte :** PetriNet (*lignes 5-14*)

  — **Contrainte ‘validNamePetriNet‘ :** le nom d’une activité ne doit être composé que de lettres, chiffres ou soulignés, un chiffre ne peut pas être première position.

  — **Contrainte ‘uniqName‘ :** deux activités différentes d’un même processus ne peuvent pas avoir le même nom.

— **Contexte :** Noeud (*lignes 16-18*)

  — **Contrainte ‘validNameNoeud‘ :** Le nom du nœud doit être valide.

— **Contexte :** Arc (*lignes 20-26*)

  — **Contrainte ‘nbJetonPositifArc‘ :** Le nombre de jetons dans l’arc doit être supérieur ou égal à 1.

    - **Contrainte ‘ValidLink‘ :** Le lien entre la source et la cible de l’arc doit être valide.

- **Contexte :** Place (*lignes 28-30*)
    - **Contrainte ‘nbJetonPositifNoeud‘ :** Le nombre de jetons dans la place doit être supérieur ou égal à 0.

Remarque : Ici on ne prend on compte les contraintes que de manière statique. Si on le faisait de manière dynamique le problème devient nettement plus compliqué.

4 Transformation d'un modèle de processus en un modèle de réseau de Pétri : JAVA, ATL

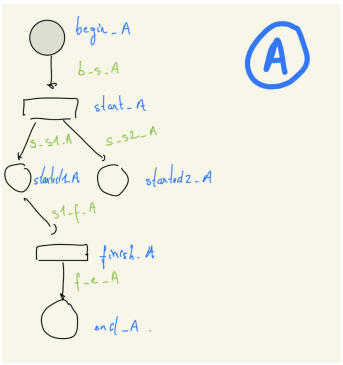


FIGURE 9 – PDL2Petri : WorkDefinition

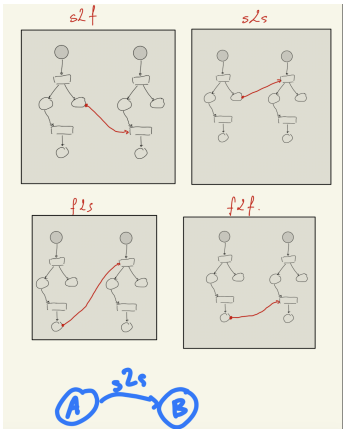


FIGURE 10 – PDL2Petri : WorkSequence

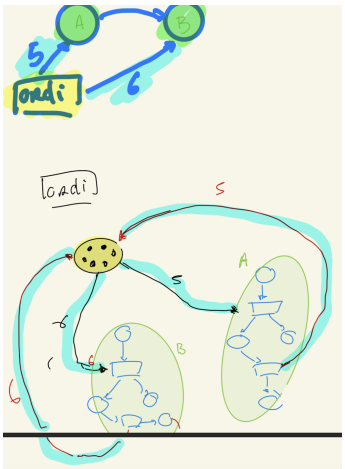


FIGURE 11 – PDL2Petri : Ressources

FIGURE 12 – Transformation SimplePDL en PetriNet

4.1 en JAVA

Avant de procéder à la transformation, nous devons charger les packages SimplePDL et PetriNet dans le registre EMF d'Eclipse. Cela est réalisé en utilisant les classes générées par EMF pour les packages respectifs, comme illustré dans le code Java suivant :

```
// Charger le package SimplePDL et PetriNet afin de l'enregistrer dans le registre d'Eclipse.
SimplepdlPackage packageInstancePDL = SimplepdlPackage.eINSTANCE;
PetrinetPackage packageInstancePetri = PetrinetPackage.eINSTANCE;
```

4.1.1 Configuration des Ressources EMF

Nous configurons ensuite la gestion des ressources EMF pour pouvoir charger et sauvegarder nos modèles. Nous enregistrons l'extension ".xmi" pour être ouverte à l'aide d'un objet "XMIResourceFactoryImpl", comme indiqué ci-dessous :

```
Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
Map<String, Object> m = reg.getExtensionToFactoryMap();
m.put("xmi", new XMIResourceFactoryImpl());
```

4.1.2 Chargement des Modèles SimplePDL et Création du Modèle PetriNet

Nous créons un objet 'ResourceSet' pour chaque modèle (SimplePDL et PetriNet) et définissons les URIs pour les localiser. Nous chargeons le modèle SimplePDL et récupérons l'élément 'Process'. Ensuite, nous créons un nouveau modèle PetriNet et le configurons avec le nom du processus.

```
ResourceSet resSetPDL = new ResourceSetImpl();
ResourceSet resSetPetri = new ResourceSetImpl();

URI modelURIPetri = URI.createURI("models/PetriNet_developpement.xmi");
```

```
Resource resourcePetri = resSetPetri.createResource(modelURIPetri);

URI modelURIPDL = URI.createURI("developpement.xmi");
Resource resourcePDL = resSetPDL.getResource(modelURIPDL, true);

Process process = (Process) resourcePDL.getContents().get(0);
Petrinet petrinet = pnFactory.createPetriNet();
petrinet.setName(process.getName());
```

4.1.3 Transformation des Éléments

Nous parcourons ensuite tous les éléments du processus SimplePDL et effectuons la transformation en éléments PetriNet correspondants.

```
for (Object o : process.getProcessElements()) {
    if (o instanceof WorkDefinition) {
        ConvertWD((WorkDefinition) o);
    } else if (o instanceof WorkSequence) {
        ConvertWS((WorkSequence) o);
    } else if (o instanceof Ressource) {
        ConvertResource((Ressource) o);
    }
}
```

4.1.4 Sauvegarde de la Ressource PetriNet

Enfin, nous sauvegardons le modèle PetriNet nouvellement créé.

```
try {
    resourcePetri.save(Collections.EMPTY_MAP);
} catch (IOException e) {
    e.printStackTrace();
}
```

Cette transformation permet de convertir un modèle SimplePDL en un modèle PetriNet, respectant les règles définies pour chaque type d'élément du processus. Les détails de la transformation de chaque type d'élément (WorkDefinition, WorkSequence, Ressource) sont encapsulés dans des fonctions spécifiques (ConvertWD, ConvertWS, ConvertResource) pour maintenir la lisibilité et la modularité du code.

4.1.5 Après l'oral

On a ajouté la ligne "magique" ce qui permet de compiler petri2tina.

4.2 en ATL

Le processus de transformation implique plusieurs étapes pour obtenir un modèle de réseau de Petri à partir d'un modèle de processus, ce sont les mêmes étape qu'en JAVA :

4.2.1 Traduction d'un Process en un PetriNet

La première étape consiste à traduire le modèle de processus (Process) en un modèle de réseau de Petri (PetriNet) portant le même nom.

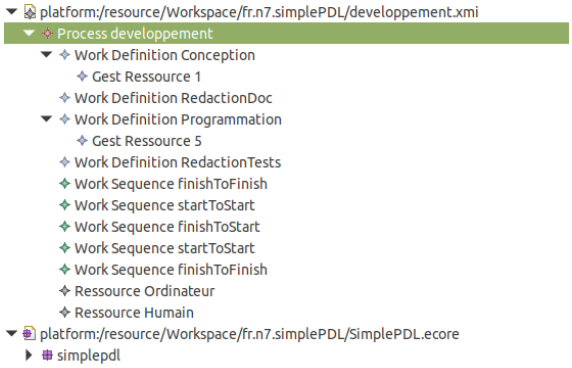


FIGURE 13 – developpement_pdl.xmi

FIGURE 14 – developpement_petri.xmi

FIGURE 15 – Transformation SimplePDL en PetriNet

4.2.2 Conversion des WorkDefinitions en réseaux de Pétri

Composés de 5 arcs, 2 transitions et 4 places. Les éléments sont ensuite reliés correctement au réseau global de Petri.

4.2.3 Conversion des WorkSequences en arcs

Les WorkSequences, définissant les dépendances entre les WorkDefinitions, sont transformées en arcs. Les nouveaux arcs créés sont ensuite ajoutés au réseau global de Petri.

4.2.4 Conversion des ressources et gestionnaires en places

Les ressources du modèle de processus sont converties en places dans le réseau de Petri. Les connexions entre ces places et le reste du réseau sont établies.

5 Transformation de modèle à texte : Acceleo

5.1 Engendrer un fichier HTML à partir d'un modèle SimplePDL

Avec Acceleo, nous générons un fichier HTML à partir d'un modèle SimplePDL. Cette approche automatise la création de documents en utilisant des templates Acceleo pour formater les informations du modèle dans le fichier HTML. Une solution concise et efficace pour la génération automatisée de documents.

```
[comment encoding = UTF-8 /]
[module toHTML('http://simplepdl')]

[template public processToHTML(aProcess : Process)]
[comment @main/]
[file (aProcess.name + '.html', false, 'UTF-8')]
<head><title>[aProcess.name/]</title></head>
<body>
<h1>Process "[aProcess.name/]"</h1>
<h2>Work definitions</h2>
[let wds : OrderedSet(WorkDefinition) = aProcess.getWDs() ]
    [if (wds->size() > 0)]
  <ul>
        [for (wd : WorkDefinition | wds)]
    <li>[wd.toHTML()/]</li>
        [/for]
  </ul>
    [else]
  <b>None.</b>
    [/if]
[/let]
</body>
[/file]
[/template]

[query public getWDs(p: Process) : OrderedSet(WorkDefinition) =
    p.processElements->select( e | e.oclIsTypeOf(WorkDefinition) )
        ->collect( e | e.oclAsType(WorkDefinition) )
        ->asOrderedSet()
/]

[query public toState(link: WorkSequenceType) : String =
    if link = WorkSequenceType::startToStart or link = WorkSequenceType::startToFinish then
        'started'
    else
        'finished'
    endif
/]

[template public toHTML(wd : WorkDefinition) post (trim()) ]
[wd.name /] [for (ws: WorkSequence | wd.linksToPredecessors)
        before (' requires ') separator (', ') after('.')
][ws.predecessor.name /] to be [ws.linkType.toState()
/][/for]
[/template]
```

FIGURE 16 – Template SimplePDL to HTML

```
<head><title>developpement</title></head>
<body>
<h1>Process "developpement"</h1>
<h2>Work definitions</h2>
  <ul>
    <li>Conception</li>
    <li>RedactionDoc requires Conception to be finished, Conception to be started.</li>
    <li>Programmation requires Conception to be finished.</li>
    <li>RedactionTests requires Conception to be started, Programmation to be finished.</li>
  </ul>
</body>

```

FIGURE 17 – Exemple : Developpement

5.2 Traduire un modèle de procédé en une syntaxe dot

Le modèle de procédé peut être représenté graphiquement en utilisant la syntaxe DOT. Le code DOT suivant illustre cette transformation :

```
digraph ModelGraph {
  // Définition des nœuds
  Task1 [label="Tâche 1"];
  Task2 [label="Tâche 2"];
  Task3 [label="Tâche 3"];

  // Définition des dépendances
  Task1 -> Task2;
  Task2 -> Task3;
}
```

```
[comment encoding = UTF-8 /]
[module toDot('http://simplepdl')]

[template public processToDot(aProcess : Process)]
[comment @main/]
[file (aProcess.name + '.dot', false, 'UTF-8')]
digraph [aProcess.name/] {
[let ws : OrderedSet(WorkSequence) = aProcess.getWs() ]
    [for (w : WorkSequence | ws)]
        [w.predecessor.name/] -> [w.successor.name/] ['['/]arrowhead=vee label=[w.linkType.toState()/]['']'/]
    [/for]
[/let]
}
[/file]
[/template]

[query public getWs(p: Process) : OrderedSet(WorkSequence) =
    p.processElements->select( e | e.oclIsTypeOf(WorkSequence) )
        ->collect( e | e.oclAsType(WorkSequence) )
        ->asOrderedSet()
/]

[template public toState(link: WorkSequenceType) post (trim()) ]
    [if (link = WorkSequenceType::startToStart)] s2s
    [elseif (link = WorkSequenceType::finishToFinish)] f2f
    [elseif (link = WorkSequenceType::finishToStart)] f2s
    [else] s2f
    [/if]
[/template]
```

FIGURE 18 – Template SimplePDL to DOT

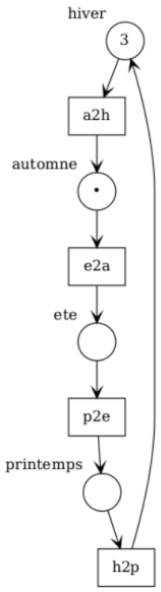


FIGURE 19 – 4Saisons

FIGURE 20 – Transformation PetriNet en Dot

```
digraph developpement {
        Conception -> RedactionDoc [arrowhead=vee label=f2f]
        Conception -> RedactionDoc [arrowhead=vee label=s2s]
        Conception -> Programmation [arrowhead=vee label=f2s]
        Conception -> RedactionTests [arrowhead=vee label=s2s]
        Programmation -> RedactionTests [arrowhead=vee label=f2f]
}
```

FIGURE 21 – Exemple SimplePDL2Dot : Developpement

Voir figure 2 pour l'exemple 22

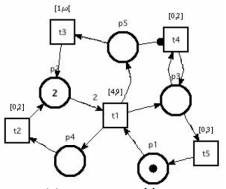
```
digraph PetriNetTest {
    subgraph Place {
        node [shape=circle]
        hiver [label="3", xlabel="hiver"]; automne [label="•", xlabel="automne"]; printemps [label="", xlabel="printemps"]; ete [label="", xlabel="ete"];
    }
    subgraph Transition {
        node [shape=rect]
        h2p; p2e; e2a; a2h;
    }
    h2p -> hiver [arrowhead=vee];
    p2e -> printemps [arrowhead=vee];
    e2a -> ete [arrowhead=vee];
    a2h -> automne [arrowhead=vee];
    hiver -> a2h [arrowhead=vee];
    automne -> e2a [arrowhead=vee];
    printemps -> h2p [arrowhead=vee];
    ete -> p2e [arrowhead=vee];
}

```

FIGURE 22 – Exemple Petri2Dot : 4Saisons

5.3 Transformation d'un réseau de Petri en Tina

Nous allons maintenant définir une transformation modèle à texte pour les réseaux de Petri. L'objectif est d'engendrer la syntaxe textuelle utilisée par les outils Tina, en particulier nd (NetDraw). La figure 23 illustre les principaux concepts présents des réseaux de Petri temporels que nous considérons. Le fichier textuel de ce réseau est donné au format Tina.

Dans ce code, nous utilisons des templates Acceleo pour générer le fichier Tina correspondant au modèle PetriNet.



(a) syntaxe graphique

```
net ifip
pl p1 (1)
pl p2 (2)
pl p3 (0)
pl p4 (0)
pl p5 (0)
tr t1 [4,9] p1 p2*2 -> p3 p4 p5
tr t2 [0,2] p4 -> p2
tr t3 [1,w[ p5 -> p2
tr t4 [0,2] p3 p5?1 -> p3
tr t5 [0,3] p3 -> p1
```

(b) syntaxe textuelle

FIGURE 23 – Exemple PetriNet2Tina

Voir figure 2 pour l'exemple 25

```
[comment encoding = UTF-8 /]
[module Petri2Tina('http://petrinet')]

[template public Petri2Tina(aPetriNet : PetriNet)]
[comment @main/]
[file (aPetriNet.name + '.net', false, 'UTF-8')]
net [aPetriNet.name/]
[for (pl : Place | aPetriNet.petrinetelement->getPlaces())]
pl [pl.name/] ([pl.nb_jetons/])
[/for]
[for (tr : Transition | aPetriNet.petrinetelement->getTransistions())]
tr [tr.name/] [tr.getPredecessors()/] -> [tr.getSuccessors()/]
[/for][/file][/template]

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

[query public getTransistions(n: OrderedSet(PetriNetElement) ): OrderedSet(Transition) =
    n->select( e | e.oclIsTypeOf(Transition) )
        ->collect( e | e.oclAsType(Transition) )
        ->asOrderedSet()
/]

[template public getPredecessors(tr : Transition)]
[for (a: Arc | tr.LinktoPredecessor)][a.source.name/] [/for]
[/template]

[template public getSuccessors(tr : Transition)]
[for (a: Arc | tr.LinktoSuccessor)][a.cible.name/] [/for]
[/template]


```

FIGURE 24 – Template PetriNet to Tina

```
net PetriNetTest
pl hiver (3)
pl automne (0)
pl printemps (0)
pl ete (0)
tr h2p hiver  -> printemps
tr p2e printemps  -> ete
tr e2a ete  -> automne
tr a2h automne  -> hiver

```

FIGURE 25 – Exemple PetriNet2Tina : 4Saisons

6 Définir une syntaxe graphique : Sirius

Sirius permet de représenter des modèles graphiquement sous la forme d'un graphe composé de nœuds et d'arcs. Il est possible de choisir la forme d'un nœud (rectangle, ellipse, etc.) et la forme d'un arc (décoration à l'extrémité, tracé du trait, etc.)

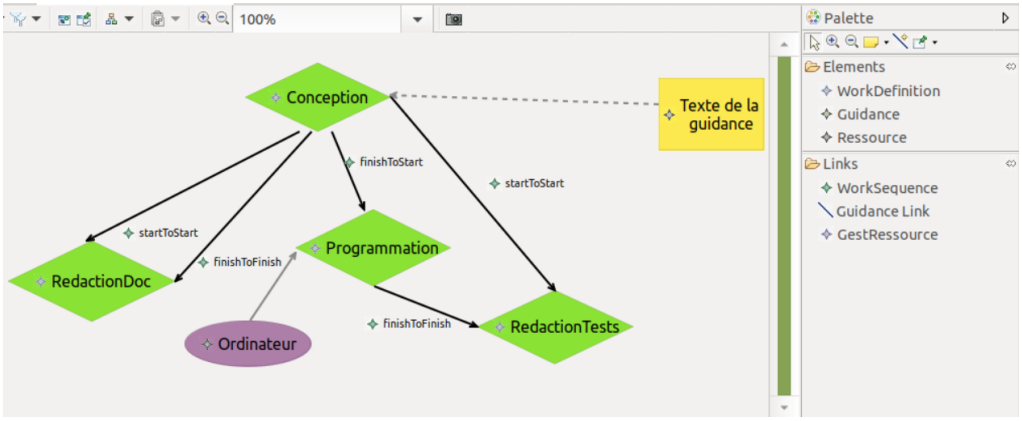


FIGURE 26 – Editeur graphique

7 Définir des syntaxes concrètes textuelles pour SimplePDL : Xtext

Pour la définition des syntaxes concrètes textuelles, nous utiliserons l'outil Xtext 1 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)). L'idée est de décrire à la fois la syntaxe concrète du langage et comment le modèle EMF est construit en mémoire au fur et à mesure de l'analyse syntaxique.

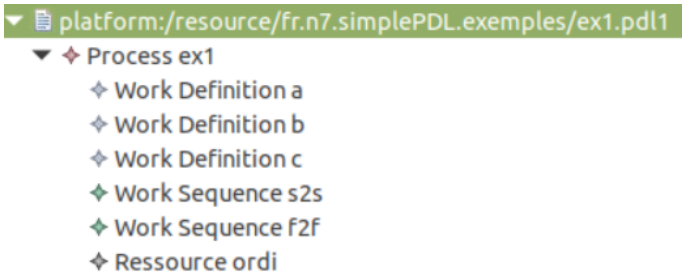


FIGURE 27 – PDL1

```
process ex1 {
wd a
wd b
wd c
ws s2s from a to b
ws f2f from b to c
res ordi qte 4
}
```

FIGURE 28 – Xtext

8 Après l'oral

Apres l'ajout de la ligne magique permet de trouver les bons packages

8.1 Compilation de la commande tina

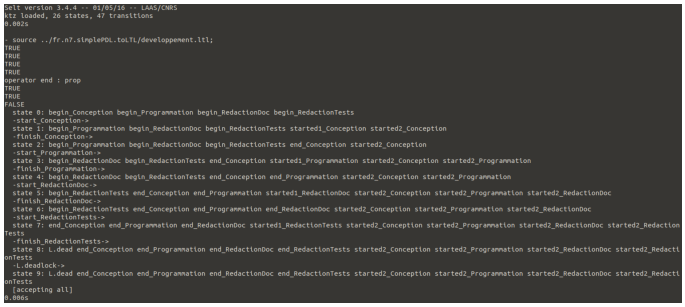


FIGURE 29 – cmd tina

avec les contraintes LTL suivante :

— chaque activité est soit non commencée, soit en cours, soit terminée ;

— une activité terminée n'évolue plus.

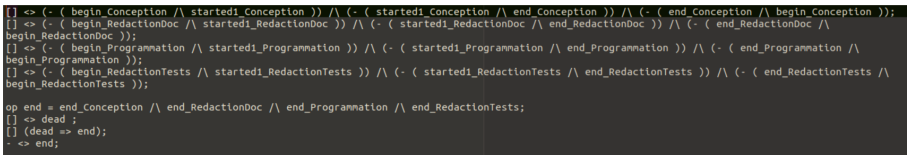


FIGURE 30 – Contraintes LTL sur le processus Developpement

9 Conclusion

En résumé, ce projet a exploré la modélisation graphique avec Sirius, la transformation de modèles avec ATL, JAVA et la génération de code avec Acceleo. Les outils choisis ont démontré leur efficacité dans la représentation visuelle des modèles, la transformation entre différentes

représentations, et la production de code, et surtout la transformation d'un modèle en un autre modèle (petri) nous permet de connaitre la terminaison ou la non terminaison d'un processus qu'on défnit avec les contraintes LTL (contrainte dynamique par rapport à aux contraintes OCL qui sont statiques).

Ce parcours a été instructif, offrant des enseignements sur les technologies de modélisation et de transformation. Les défis rencontrés ont été des opportunités d'apprentissage, contribuant à notre développement professionnel.