Ingénierie dirigée par les Modèles
Mini-Projet

Thierry Xu
Tom Bonetto

Département Sciences du Numérique - Deuxième année
2022-2023

Contents

List of Figures

1 Introduction

1.1 Chaîne de vérification de modèles de processus

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. Le mini-projet correspond à ce qui a été fait dans les TP avec une petite extension : les ressources.

1.2 Objectif du mini-projet

1. Définition des métamodèles avec Ecore.
2. Définition de la sémantique statique avec OCL (Complete OCL).
3. Utilisation de l'infrastructure fournie par EMF pour manipuler les modèles.
4. Définition de transformations modèle à texte (M2T) avec Acceleo, par exemple pour engendrer la syntaxe attendue par Tina à partir d'un modèle de réseau de Petri ou engendrer les propriétés LTL à partir d'un modèle de processus.
5. Définition d'une transformation de modèle à modèle (M2M) avec EMF/Java et avec ATL
6. Définition de syntaxes concrètes textuelles avec Xtext.
7. Définition de syntaxes concrètes graphiques avec Sirius.

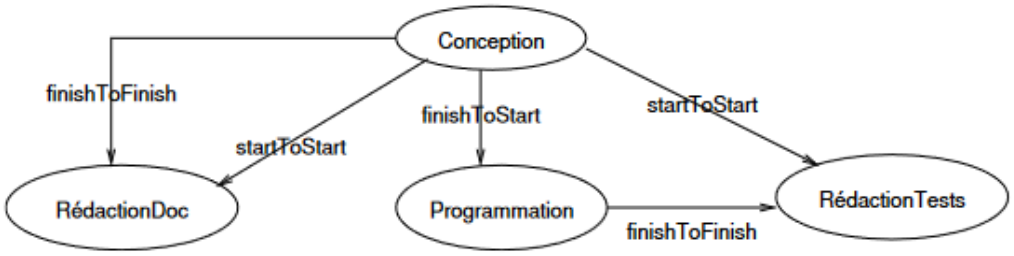

Figure 1: Exemple de modèle de procédé

On utilisera ce modèle de procédé, auquel on ajoutera des ressources pour la suite du projet.

1.3 Les métamodèles

1.3.1 SimplePDL

Le SimplePDL est un langage de métamodélisation qui permet de 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), la notion de Guidance est également ajoutée qui est l'équivalent d'une annotation UML : elle permet d'associer un texte à plusieurs éléments d'un processus.

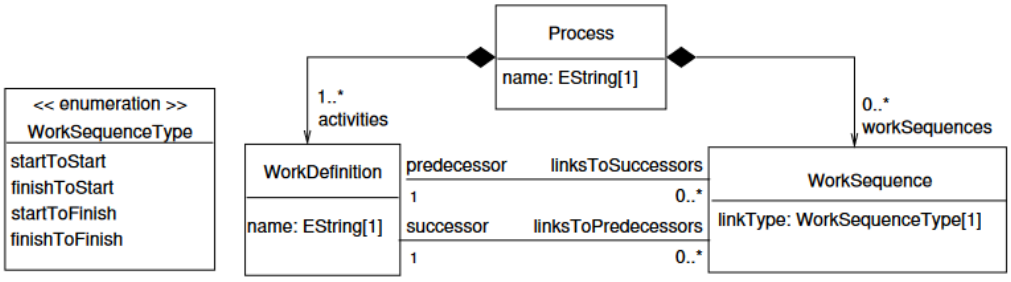

Figure 2: Métamodèle initial de SimplePDL

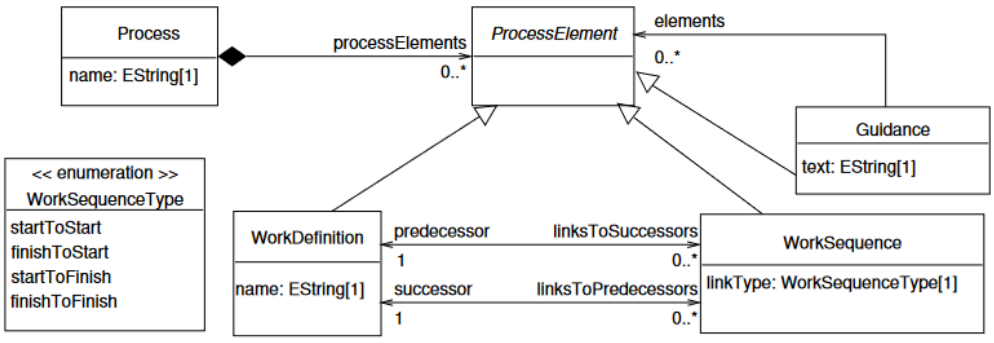

Figure 3: Nouveau métamodèle de SimplePDL

1.3.2 Réseaux de Petri

Un réseau de Petri est un modèle mathématique servant à représenter divers systèmes travaillant avec des variables discrètes. 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.

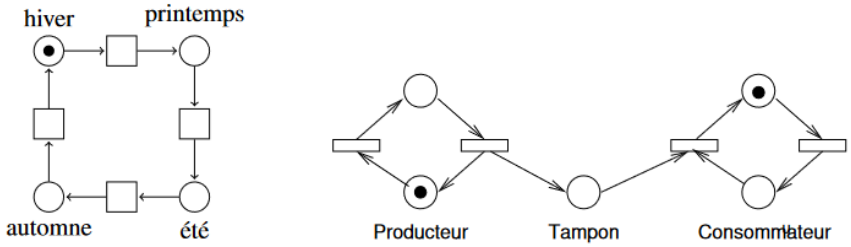

Figure 4: Exemples de réseaux de Petri

Un réseau de Petri évolue lorsqu'on exécute une transition : des jetons sont pris dans les places en entrée de cette transition et envoyés dans les places en sortie de cette transition. Le nombre de jetons pris dans les places d'entrée et mis dans les places de sortie est celui indiqué par l'arc correspondant. Une transition ne peut être exécutée que si elle est exécutable 3, c'est-à-dire qu'il y a dans chaque place d'entrée un nombre de jetons au moins égal au poids de l'arc. L'exécution d'une transition est une opération indivisible qui est déterminée par la présence de jetons sur les places d'entrée.

2 Métamodèles

Cette partie correspond aux tâches 1 et 2 qui ont pour but de compléter le métamodèle SimplePDL pour prendre en compte les ressources et de définir le métamodèle PetriNet.

Cf fichier :

- Métamodèle SimplePDL : SimplePDL.ecore
- Image du métamodèle SimplePDL : SimplePDL.png
- Métamodèle PetriNet : PetriNet.ecore
- Image du métamodèle PetriNet : PetriNet.png

2.1 SimplePDL

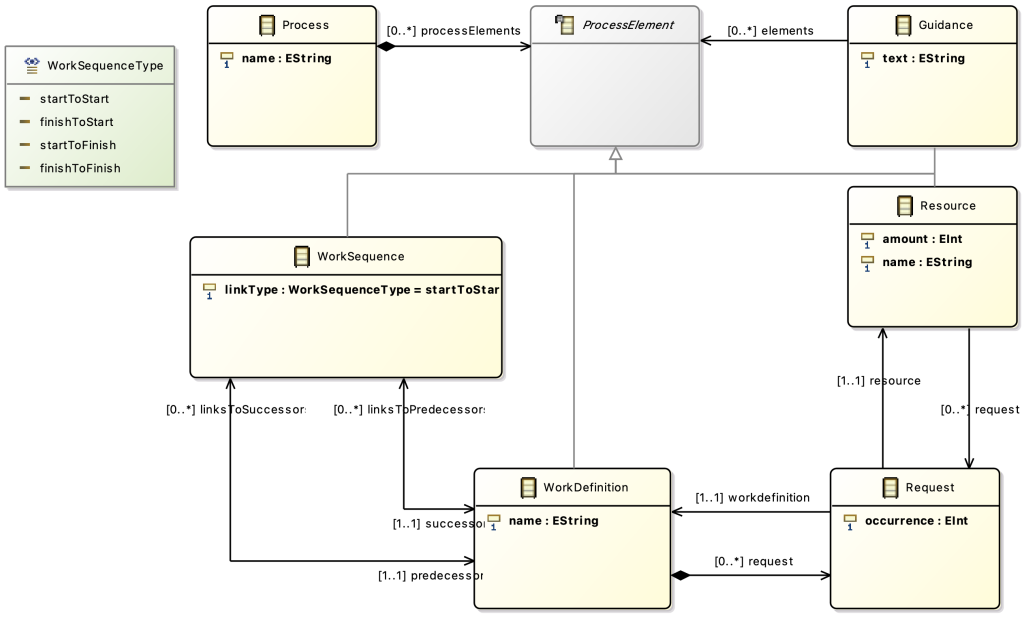

Figure 5: Métamodèle Ecore de SimplePDL

Après le rajout des ressources, on retrouve donc le concept de processus (**Process**) qui se compose des activités (**WorkDefinition**), des dépendances (**WorkSequence**) et des ressources (**Resource**). Les activités sont les tâches que le processus doit réaliser, les dépendances relient les activités entre eux avec des relations d'ordonnancement représentées par l'énumération (**WorkSequenceType**) qui caractérise le démarrage et la fin d'une certaine activité et possédant quatre valeurs (**startToStart, finishToStart, startToFinisih, finishToFinish**). Dans notre cas concret voire figure 1, l'activité "Conception" est par exemple reliée à l'activité "RédactionTests" avec la relation startToStart: la rédaction des tests ne peut commencer que dans le cas où la conception a commencé.

La réalisation d'une activité peut nécessiter plusieurs ou aucune ressource. Ces ressources sont caractérisées par leurs types (l'attribut name) et leurs quantités disponibles (l'attribut amount) . Par exemple (voire figure 6), la réalisation de l'activité programmation peut nécessiter trois développeurs (acteur humain) et quatre machines (outil machine). Cependant, une activité doit demander une quantité disponible des ressources d'où le recours à la classe (**Request**) qui contrôle ceci (l'attribut occurrence).

	Quantité	Conception	RédactionDoc	Programmation	RédactionTest
concepteur	3	2			
développeur	2			2	
machine	4	2	1	3	2
rédacteur	1		1		
testeur	2				1

Figure 6: Affection de ressources pour le processus décrit figure 1

2.2 PetriNet

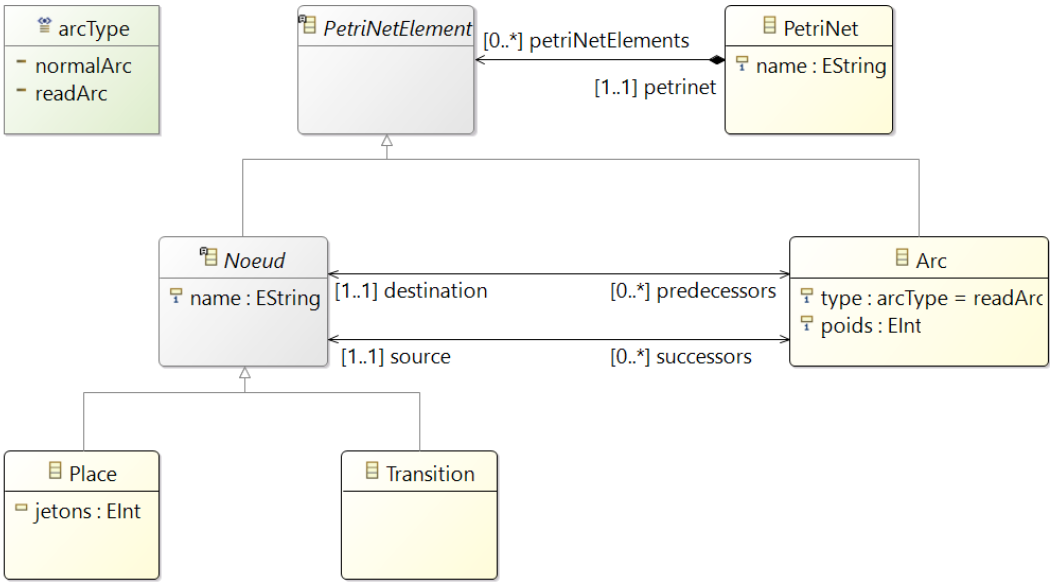

Figure 7: Métamodèle Ecore de PetriNet

On retrouve dans notre réseau de Petri le concept principal (**PetriNet**) qui se compose de plusieurs éléments (**PetriNetElement**) : les arcs (**Arc**) reliant des noeuds (**Noeud**) qui sont soit des places (**Place**) soit des transitions (**Transition**). Les places contiennent un nombre initial de jetons (l'attribut jetons), les arcs possèdent deux types possibles regroupés dans l'énumération (**arcType**): normalArc (qui connecte une place et une transition) et readArc (qui connecte une place d'entrée et une transition). Les arcs possèdent aussi un nombre de jetons (l'attribut poids) qui sera consommé d'une place vers une transition et inversement d'une transition vers une place.

3 Éditeur graphique SimplePDL

L'objectif de cette partie qui correspond à la tâche 3 est de définir une syntaxe graphique permettant de saisir des modèles conformes à SimplePDL. Nous utilisons l'outil Sirius pour la réalisation de cette syntaxe.

Cf fichier :

- Modèles Sirius décrivant l'éditeur graphique pour SimplePDL : simplepdl.osdesign

À l'instar d'une syntaxe concrète textuelle, une syntaxe concrète graphique fournit un moyen de visualiser et/ou éditer agréablement et efficacement un modèle. Nous allons utiliser l'outil Eclipse Sirius, il permet de définir une syntaxe graphique pour un langage de modélisation décrit en Ecore et d'engendrer un éditeur graphique intégré à Eclipse.

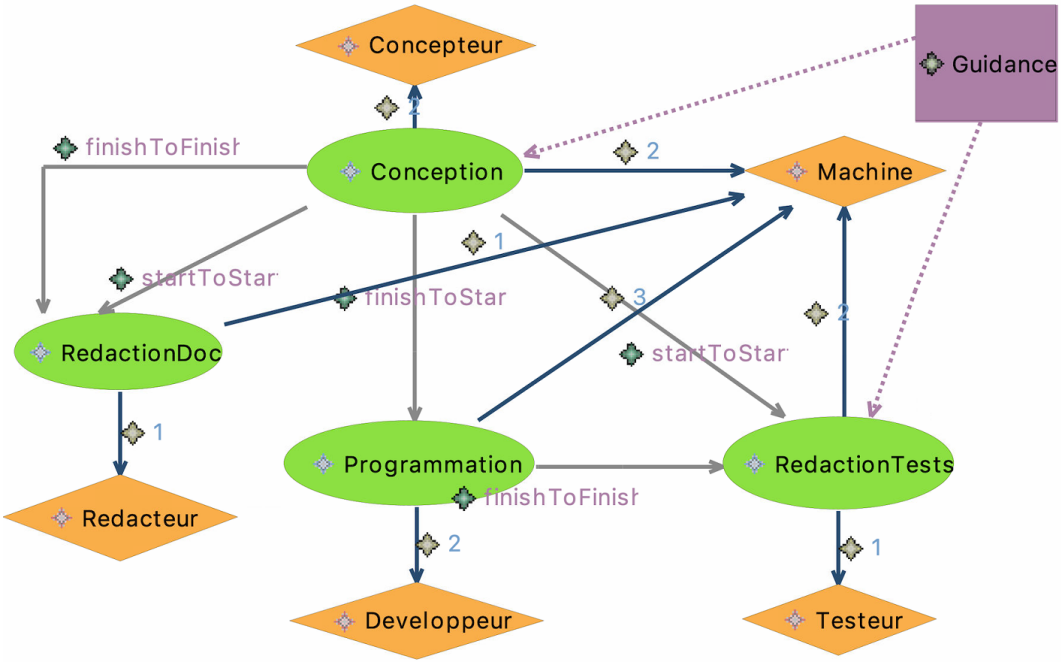

Figure 8: Représentation graphique Sirius du modèle figure 1 avec ressources

On a donc défini avec Sirius une syntaxe graphique pour SimplePDL, ce qui nous permet à la fois d'avoir un affichage graphique des modèles mais également de pouvoir saisir graphiquement de nouveaux modèles grâce aux outils graphiques développés dans la palette Sirius.

4 Contraintes OCL

L'objectif de cette partie qui correspond à la tâche 4 est de définir les contraintes qui ne sont pas prises en charge lors de la conception des métamodèles.

Cf fichier :

- Contraintes OCL SimplePDL : simplepdl.ocl
- Exemple de procdédé avec Request non valide : SimplePDL_requete_invalide.xmi
- Contraintes OCL PetriNet : PetriNet.ocl
- Exemple de Petri avec Arc non valide : PetriNet_Invalid_Arc.xmi

Nous avons utilisé Ecore pour définir un méta-modèle pour les processus. 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 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.

4.1 Contraintes SimplePDL

Voici les différentes contraintes que nous avons définies pour SimplePDL dans le fichier .ocl :

- Nom valide Process
- Non réflexivité des dépendances
- Successeur et prédécesseur différents pour une même WorkSequence
- Nom différents pour des sous-activités différentes
- Nom valide d'acitivité (WorkDefinition)
- Une unique requête par ressource pour une activité
- La demande d'une ressource ne doit pas être supérieur à la quantité disponible et doit être non nulle
- Deux ressources différentes ont des noms différents
- Quantité d'une ressource positive strictement
- Nom valide ressource

4.2 Contraintes PetriNet

Voici les différentes contraintes que nous avons définies pour PetriNet dans le fichier .ocl :

- Nom valide PetriNet
- Non réflexivité des dépendances
- Marquage des places positives
- Nom valide Noeud
- Nom différents pour des noeuds différents
- Poids d'un arc supérieur à 0
- Arc relie une Place et une Transition

5 Syntaxe concrète textuelle Xtext

L'objectif de cette partie qui correspond à la tâche 5 est de définir une syntaxe concrète textuelle permettant de décrire un processus SimplePDL dans un fichier .pdl grâce au greffon XText.

- Modèle Xtext décrivant la syntaxe concrète : PDL.xtext
- Modèle Xtext généré à partir du modèle Ecore : PDL1.xtext

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 syntaxe concrète 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. 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). Voici le résultat sur un exemple de procédé :

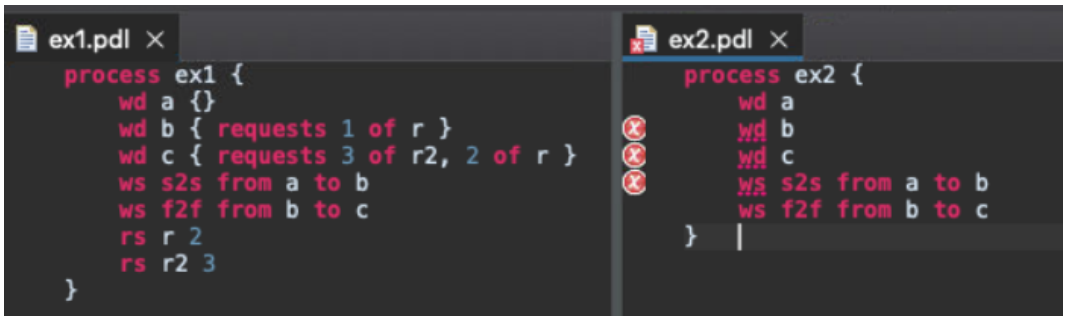

Figure 9: Syntaxe textuelle pour un processus

On remarque la présence de coloration pour les différentes instances de SimplePDL (process, wd, s2s, etc...), on remarque également pour ex2.pdl la détection d'erreurs suite à une mauvaise déclaration des WorkDefinition b et c.

6 Transformation SimplePDL vers PetriNet

L'objectif de cette partie qui correspond aux tâches 6,7 et 8 est de définir une transformation SimplePDL vers PetriNet en utilisant EMF/Java et ATL et de valider ces transformations en faisant des tests.

EMF permet de passer d'un modèle à un autre en utilisant du code Java. À partir des deux métamodèles que nous avons écrits, du code Java a été généré permettant d'interagir avec eux : des classes représentant les différents éléments des métamodèles et des méthodes pour les créer. En parcourant les éléments du modèle SimplePDL on applique les transformations dans le but de créer le modèle PetriNet à l'aide des outils générés précédemment. Il est aussi possible de faire la transformation SimplePDL vers PetriNet en utilisant ATL, un langage qui permet la traduction de modèle d'un métamodèle vers un modèle d'un autre métamodèle. Le principe d'ATL est de définir un ensemble de règles de transformations permettant de passer d'un élément d'un métamodèle à un élément d'un autre métamodèle.

Le principe de la transformation est semblable pour les 2 méthodes, on transforme le Process en un PetriNet qui possède le même nom. Les WorkDefinition sont transformés en 4 places, 2 transitions et 5 arcs suivant le schéma figure 10, les WorkSequence sont transformés en 4 arcs qui relient les places et les transitions des 2 WorkDefinition en lien. Enfin chaque Resource est transformé en 1 place avec un nombre de jetons égal à la quantité de la ressource, et ce sont les Request transformés en arc qui lie les ressources aux activités : un arc pour permettre aux activités d'emprunter une quantité de ressources et un arc pour libérer cette quantité et la rendre.

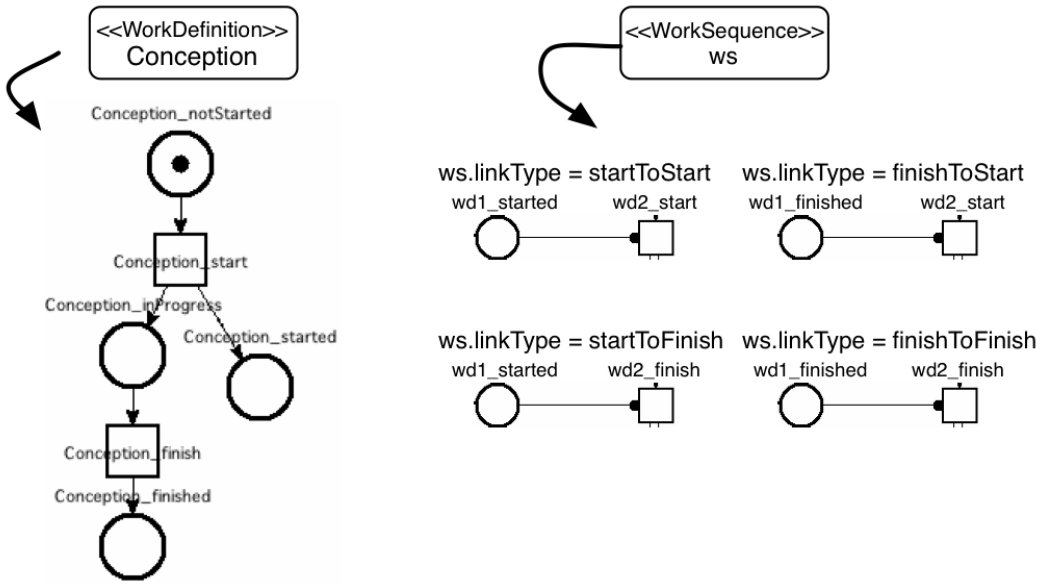

Figure 10: Exemple de transformation pour l'activité Conception

6.1 Transformation ATL

Cf fichier :

- Métamodèle SimplePDL : SimplePDL2PetriNet.atl

On a effectué cette transformation sur l'exemple du sujet que l'on a pu décrire grâce au Ecore de SimplePDL (fichier sujetPDL.xmi modélisant ce processus voire figure 11).

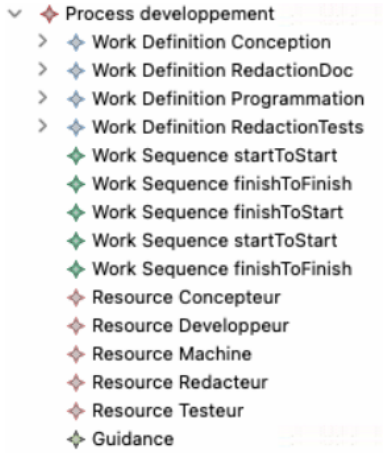

Figure 11: Modèle de procédé utilisé pour transformation ATL

Figure 12: Modèle de PetriNet résultat de la transformation ATL

6.2 Transformation EMF/Java

Cf fichier :

- Métamodèle SimplePDL : SimplePDL2PetriNetEMF.java

La transformation EMF/Java donne le même résultat que la transformation ATL.

7 Transformation PetriNet vers Tina Acceleo

L'objectif de cette partie qui correspond à la tâche 9 est de définir une transformation PetriNet vers Tina en utilisant Acceleo.

Cf fichier :

- Template Acceleo toTina: toTina.mtl
- Template Acceleo toDot: toDot.mtl

Le templace Acceleo toTina.mtl transforme un modèle de réseau de Petri en un fichier .net qui correspond à une représentation textuelle du réseau. Ce fichier est visible sur la figure 13.

```
net developpement
pl Conception_ready (1)
pl Conception_inProgress (0)
pl Conception_started (0)
pl Conception_finished (0)
pl RedactionDoc_ready (1)
pl RedactionDoc_inProgress (0)
pl RedactionDoc_started (0)
pl RedactionDoc_finished (0)
pl Programmation_ready (1)
pl Programmation_inProgress (0)
pl Programmation_started (0)
pl Programmation_finished (0)
pl RedactionTests_ready (1)
pl RedactionTests_inProgress (0)
pl RedactionTests_started (0)
pl RedactionTests_finished (0)
pl Concepteur_resource (3)
pl Developpeur_resource (2)
pl Machine_resource (4)
pl Redacteur_resource (1)
pl Testeur_resource (2)

tr Conception_start  Conception_ready Concepteur_resource*2 Machine_resource*2 ->  Conception_inProgress Conception_started
tr Conception_finish  Conception_inProgress ->  Conception_finished Concepteur_resource*2 Machine_resource*2
tr RedactionDoc_start  RedactionDoc_ready Conception_started?1 Machine_resource Redacteur_resource ->  RedactionDoc_inProgress RedactionDoc_started
tr RedactionDoc_finish  RedactionDoc_inProgress Conception_finished?1 ->  RedactionDoc_finished Machine_resource Redacteur_resource
tr Programmation_start  Programmation_ready Conception_finished?1 Developpeur_resource*2 Machine_resource*3 ->  Programmation_inProgress Programmation_started
tr Programmation_finish  Programmation_inProgress ->  Programmation_finished Developpeur_resource*2 Machine_resource*3
tr RedactionTests_start  RedactionTests_ready Conception_started?1 Machine_resource*2 Testeur_resource ->  RedactionTests_inProgress RedactionTests_started
tr RedactionTests_finish  RedactionTests_inProgress Programmation_finished?1 ->  RedactionTests_finished Machine_resource*2 Testeur_resource
```

Figure 13: Fichier généré par le template toTina

Quant au template Acceleo toDot.mtl, celui-ci permet de transformer un modèle de réseau de Petri en un fichier .dot qui permettra de visualiser graphiquement le réseau. Sur les figures 14 et 15 ci-dessous, on observe le réseau de Petri généré à partir de l'exemple de procédé du sujet.

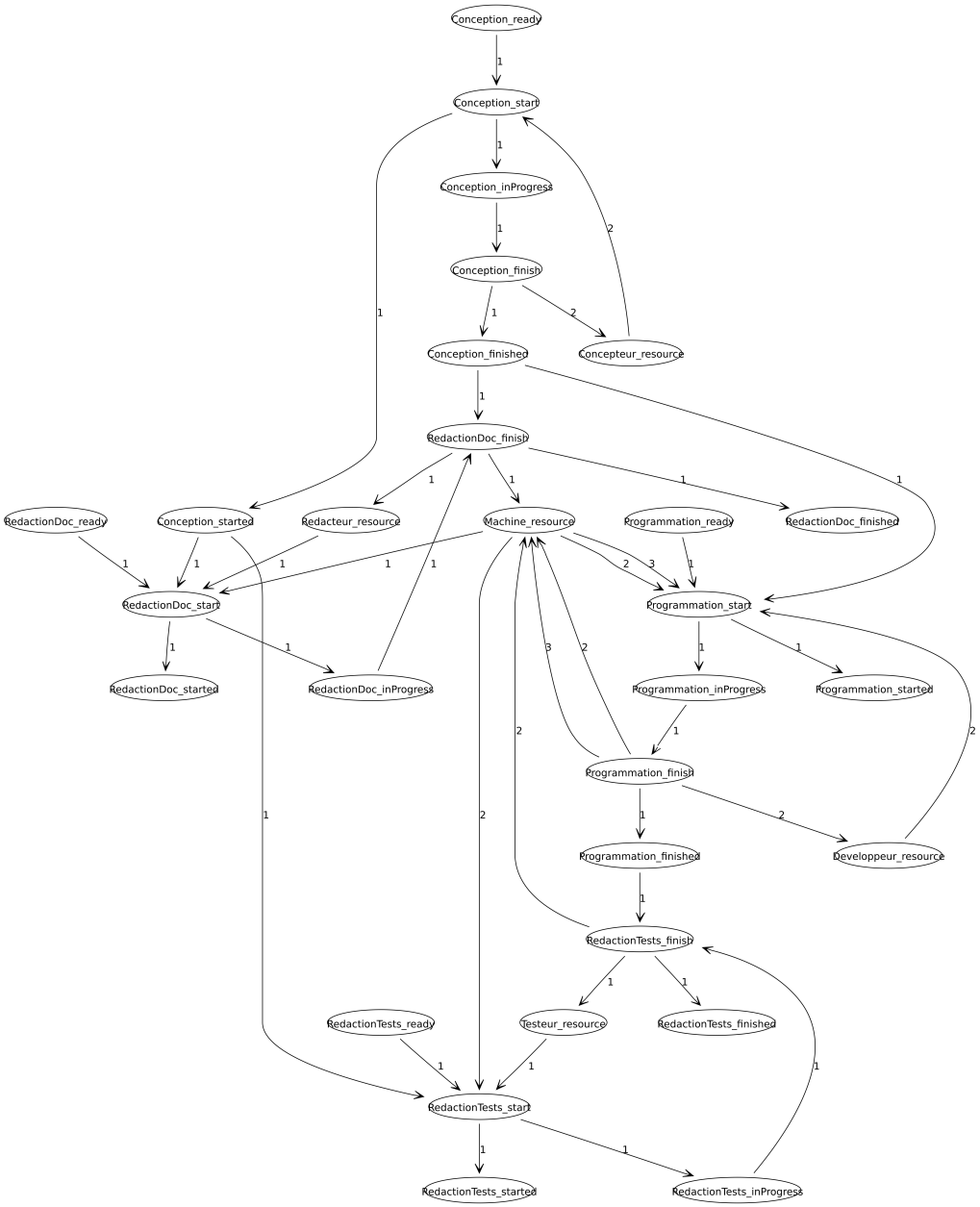

Figure 14: Représentation graphique du PetriNet résultant de la transformation Java

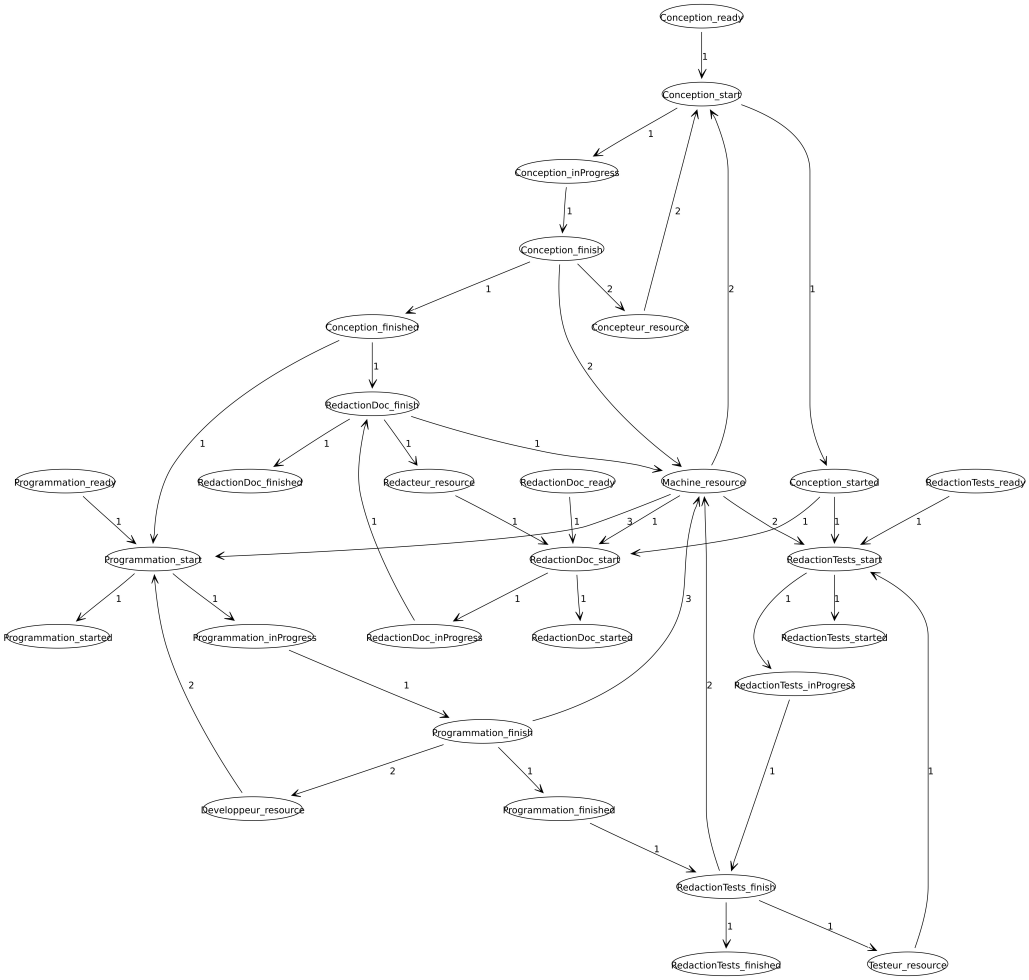

Figure 15: Représentation graphique du PetriNet résultant de la transformation ATL

8 Propriétés LTL

L'objectif de cette partie qui correspond aux tâches 10 et 11 est d'engendrer les propriétés LTL permettant de vérifier la terminaison d'un processus et les appliquer sur différents modèles de processus. Et d'engendrer les propriétés LTL correspondant aux invariants de SimplePDL pour valider la transformation écrite.

Cf fichier :

- Template Acceleo toLTL: toLTL.mtl

Le fichier .ltl (voire figure 16) généré à l'aide du template contient les propriétés citées ci-dessus.

```
1 <> (Conception_finished /\ RedactionDoc_finished /\ Programmation_finished /\ RedactionTests_finished);
2
3 [](Conception_ready + Conception_inProgress + Conception_finished = 1);
4 [](RedactionDoc_ready + RedactionDoc_inProgress + RedactionDoc_finished = 1);
5 [](Programmation_ready + Programmation_inProgress + Programmation_finished = 1);
6 [](RedactionTests_ready + RedactionTests_inProgress + RedactionTests_finished = 1);
7
8 [](Conception_finished => [] (Conception_finished));
9 [](RedactionDoc_finished => [] (RedactionDoc_finished));
10 [](Programmation_finished => [] (Programmation_finished));
11 [](RedactionTests_finished => [] (RedactionTests_finished));
12
13 - [] (- dead)
14
```

Figure 16: Fichier généré par le template toLTL

À la ligne 13 a été ajouté une propriété veillant à vérifier la présence d'interblocages. Sur la figure 17, nous remarquons que la propriété de terminaison d'un processus n'est pas vérifiée. Cela peut être effectivement expliqué par la présence d'interblocages dans le procédé.

```
Selt version 3.7.0 -- 05/19/22 -- LAAS/CNRS
ktz loaded, 22 states, 33 transitions
0.001s

- source eclipse-workspace/fr.n7.toLTL/result/developpement.ltl;
FALSE
  state 0: L.scc*21 Concepteur_resource*3 Conception_ready Developpeur_resource*2 Machine_resource*4 Programmation_ready Redacteur_resource RedactionDoc_ready Red
actionTests_ready Testeur_resource*2
  -Conception_start->
  state 1: L.scc*20 Concepteur_resource Conception_inProgress Conception_started Developpeur_resource*2 Machine_resource*2 Programmation_ready Redacteur_resource
RedactionDoc_ready RedactionTests_ready Testeur_resource*2
  -Conception_finish->
  state 2: L.scc*17 Concepteur_resource*3 Conception_finished Conception_started Developpeur_resource*2 Machine_resource*4 Programmation_ready Redacteur_resource
RedactionDoc_ready RedactionTests_ready Testeur_resource*2
  -RedactionDoc_start->
  state 15: L.scc*15 Concepteur_resource*3 Conception_finished Conception_started Developpeur_resource*2 Machine_resource*3 Programmation_ready RedactionDoc_inPro
gress RedactionDoc_started RedactionTests_ready Testeur_resource*2
  -RedactionDoc_finish->
  state 16: L.scc*13 Concepteur_resource*3 Conception_finished Conception_started Developpeur_resource*2 Machine_resource*4 Programmation_ready Redacteur_resource
 RedactionDoc_finished RedactionDoc_started RedactionTests_ready Testeur_resource*2
  -RedactionTests_start->
* [accepting] state 17: L.scc*12 L.dead Concepteur_resource*3 Conception_finished Conception_started Developpeur_resource*2 Machine_resource*2 Programmation_ready
 Redacteur_resource RedactionDoc_finished RedactionDoc_started RedactionTests_inProgress RedactionTests_started Testeur_resource
  -L.deadlock->
  state 17: L.scc*12 L.dead Concepteur_resource*3 Conception_finished Conception_started Developpeur_resource*2 Machine_resource*2 Programmation_ready Redacteur_r
esource RedactionDoc_finished RedactionDoc_started RedactionTests_inProgress RedactionTests_started Testeur_resource
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
0.016s
```

Figure 17: Vérification des propriétés LTL