



# Chaîne de vérification de modèles de processus

MANGENOT Guilhem  
SCHEYDER Claire

Département Sciences du Numérique - Deuxième année  
2023-2024

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Les métamodèles</b>	<b>3</b>
2.1	Métamodèle SimplePDL . . . . .	3
2.2	Métamodèle PetriNet . . . . .	4
<b>3</b>	<b>Les contraintes OCL</b>	<b>5</b>
<b>4</b>	<b>Validation des métamodèles et des contraintes</b>	<b>5</b>
4.1	Exemples de modèles SimplePDL . . . . .	5
4.2	Exemples de modèles PetriNet . . . . .	6
<b>5</b>	<b>Transformation SimplePDL vers PetriNet</b>	<b>7</b>
5.1	En utilisant EMF/Java . . . . .	9
5.2	En utilisant ATL . . . . .	9
<b>6</b>	<b>Validation de la transformation</b>	<b>9</b>
<b>7</b>	<b>Transformation modèle à texte</b>	<b>10</b>
<b>8</b>	<b>Développement d'un éditeur graphique pour SimplePDL</b>	<b>10</b>
<b>9</b>	<b>Définition d'une syntaxe textuelle pour SimplePDL</b>	<b>11</b>
<b>10</b>	<b>Vérification de la terminaison d'un processus</b>	<b>12</b>
<b>11</b>	<b>Conclusion</b>	<b>12</b>

## Table des figures

1	Chaine de vérification . . . . .	3
2	Diagramme de classe du métamodèle SimplePDL . . . . .	4
3	Diagramme de classe du métamodèle PetriNet . . . . .	5
4	Modèle développement . . . . .	6
5	Modèle blocage . . . . .	6
6	Modèle concurrence . . . . .	6
7	Modèle saisons . . . . .	7
8	Modèle ReadArc . . . . .	7
9	WorkSequence en réseau de Petri . . . . .	8
10	WorkDefinition en réseau de Petri . . . . .	8
11	Ressource en réseau de Petri . . . . .	8
12	Modèle développement sous forme de réseau de Petri . . . . .	9
13	Modèle développement visualisé avec Sirius . . . . .	11

# 1 Introduction

L'objectif de ce projet est de vérifier la cohérence de modèles de processus SimplePDL. Pour cela, nous avons transformé ces modèles en réseaux de Petri, ce qui nous permet de vérifier certaines propriétés comme la terminaison des processus.

La chaîne de transformation, ainsi que les outils utilisés, sont représentés en figure 1.

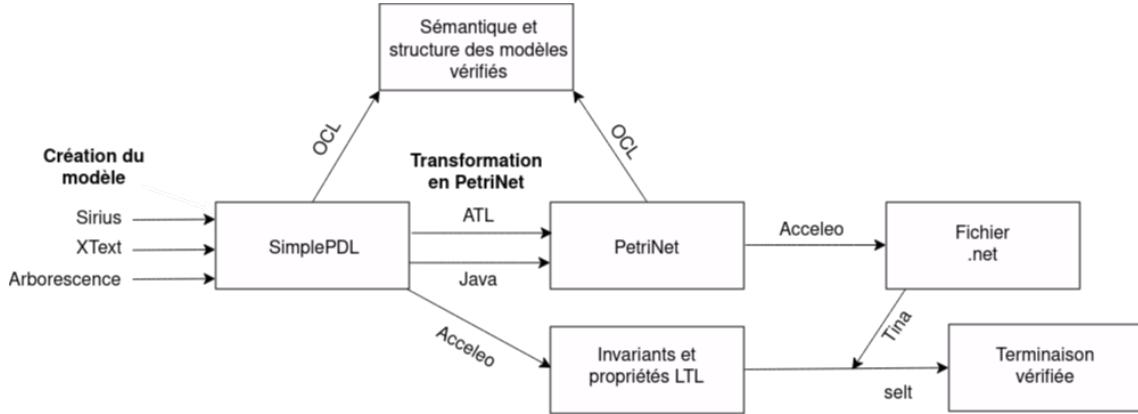


FIGURE 1 – Chaîne de vérification

Ce rapport est accompagné des fichiers source suivants :

- SimplePDL.ecore : métamodèle SimplePDL
- PetriNet.ecore : métamodèle PetriNet
- diagramme\_classe\_\*.jpg : images des métamodèles SimplePDL et PetriNet
- SimplePDL.ocl : contraintes OCL de SimplePDL
- PetriNet.ocl : contraintes OCL de PetriNet
- PetriNetFromSimplePDL.java : code Java de la transformation PetriNet à SimplePDL
- simplepdlToPetrinet.atl : code ATL de la transformation PetriNet à SimplePDL
- toTina.mtl : code Acceleo de la transformation SimplePDL en fichier .net
- toLTL.mtl : engendrement de propriétés LTL pour le réseau de Pétri correspondant à un modèle SimplePDL
- simplepdl.odesign : modèle Sirius décrivant l'éditeur graphique pour SimplePDL
- pdl1ToSimplepdl.atl : transformation PDL1 à SimplePDL
- PDL1.xtext : modèle XText décrivant la syntaxe textuelle de PDL1
- \*.xmi : modèles SimplePDL et PetriNet cités dans ce rapport

## 2 Les métamodèles

Nous avons commencé par définir les métamodèles SimplePDL et PetriNet. Ils correspondent aux fichiers SimplePDL.ecore et PetriNet.ecore.

### 2.1 Métamodèle SimplePDL

Un processus SimplePDL est un processus (Process) composé d'élément de processus (ProcessElement) représentant les activités (WorkDefinition), les dépendances (WorkSequence) et les ressources du processus. Nous y avons ajouté des commentaires (Guidance) qui peuvent annoter n'importe lequel de ces éléments.

Une activité peut avoir besoin de ressources pour s'exécuter. Ainsi, une WorkDefinition peut avoir des RessourceNeeds, qui décrivent pour une ressource donnée la quantité de celle-ci exigée par la WorkDefinition.

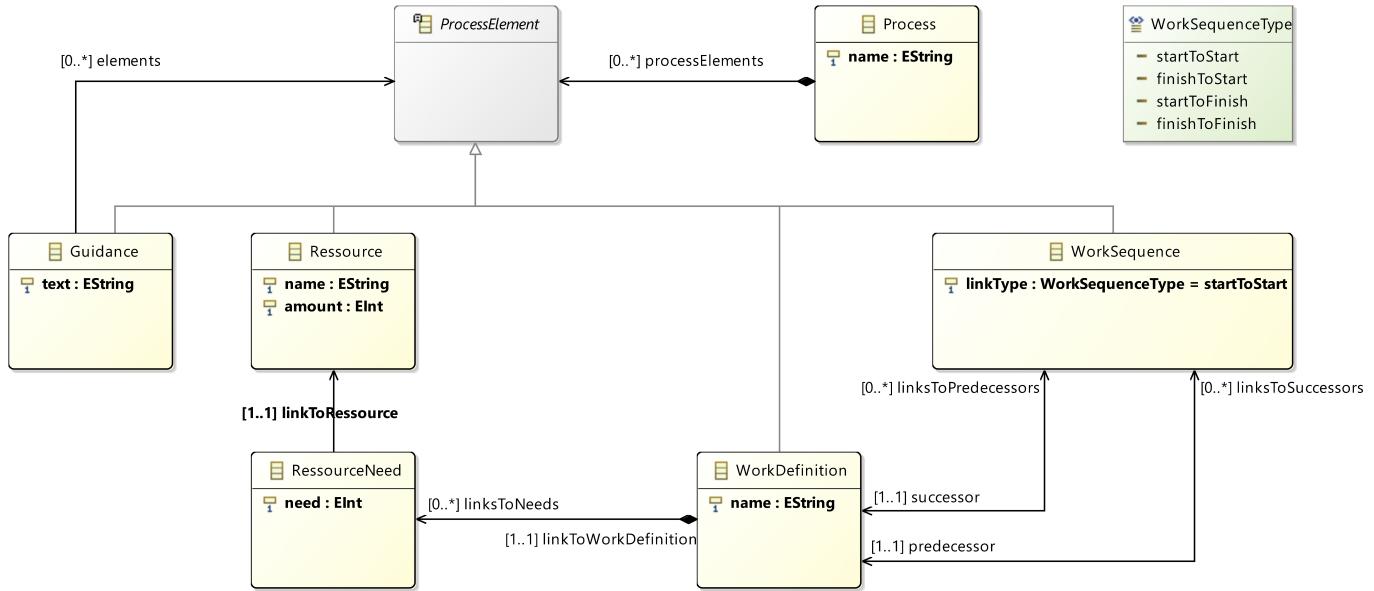


FIGURE 2 – Diagramme de classe du métamodèle SimplePDL

## 2.2 Métamodèle PetriNet

Un réseau de Petri est un Network, composé de NetworkElements. Ces éléments peuvent être des places, des transitions ou des arcs.

Nous avons choisi de faire hériter Place et Transition de CasesElement pour factoriser les éléments communs aux deux classes (le nom et les liens vers les arcs précédents et suivants). Grâce à cela, nous avons également pu construire un seul type d'arc, qui relie deux CasesElement, au lieu de deux différents (un qui permet d'aller d'une place vers une transition et un qui permet d'aller d'une transition vers une place).

Cependant, avec cette modélisation, des problèmes de structure peuvent avoir lieu. Par exemple, il est possible qu'un arc relie deux places ou deux transitions entre elles. Ces problèmes seront résolu à l'aide des contraintes OCL.

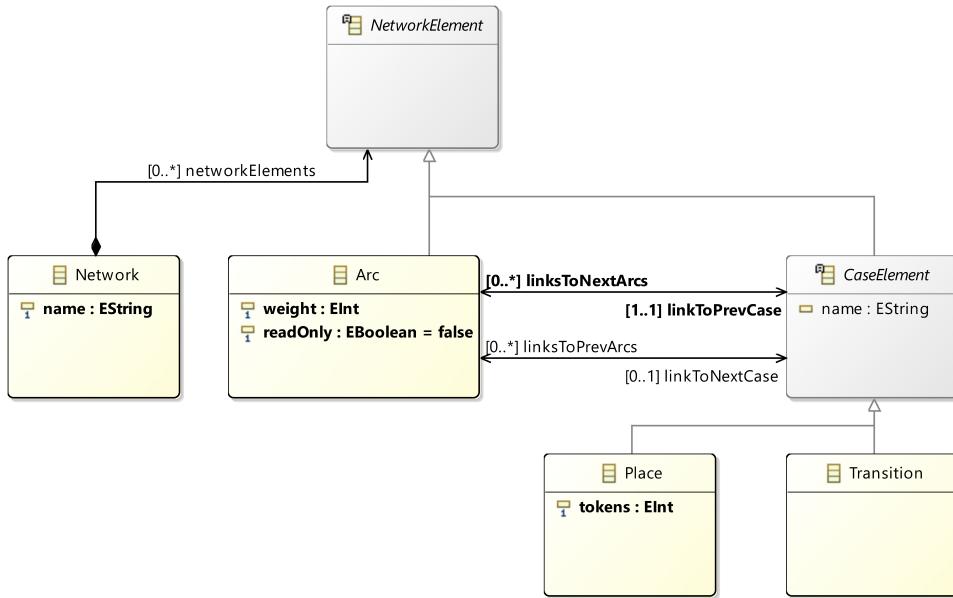


FIGURE 3 – Diagramme de classe du métamodèle PetriNet

### 3 Les contraintes OCL

Pour les contraintes OCL de chacun des métamodèles, nous nous sommes assurés que les noms donnés aux différents objets étaient corrects, c'est-à-dire des noms suffisamment longs, avec des caractères alphanumériques et différents pour chaque objet.

Concernant les ressources du métamodèle SimplePDL, nous avons mis comme contraintes que les ressources disponibles doivent être positives, ainsi que les ressources demandées. Nous avons réfléchi à ajouter que les ressources demandées soient inférieures aux ressources disponibles, mais ce sera vérifié grâce à l'outil Selt, qui nous montrera que le processus ne peut pas se finir. Ces contraintes se trouvent dans le fichier SimplePDL.ocl.

Pour les contraintes liées au réseaux de Petri, nous avons fait en sorte que un arc ne puisse pas relier deux cases ou deux transitions ensemble. Nous avons également fait en sorte qu'un read-arc puisse uniquement relier une place à une transition et non le contraire. Ces contraintes se trouvent dans le fichier PetriNet.ocl.

Certaines de ces contraintes sont essentielles pour exécuter plusieurs étapes de la chaîne de transformation.

### 4 Validation des métamodèles et des contraintes

Pour valider nos métamodèles et tester nos contraintes OCL, nous avons créé des modèles qui suivent ou non les contraintes OCL.

#### 4.1 Exemples de modèles SimplePDL

Nous avons créé le modèle developpement.xmi, représenté en figure 4, qui vérifie les contraintes OCL et est à priori censé se terminer.

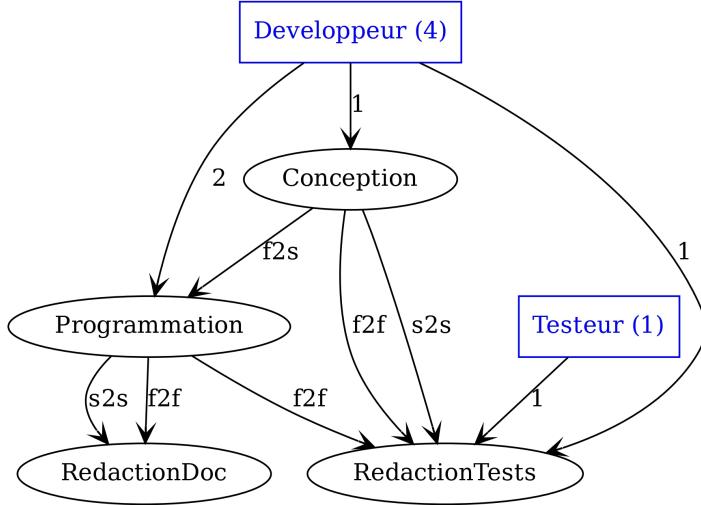


FIGURE 4 – Modèle développement

Nous avons ensuite créé les modèles `blocage.xmi` et `concurrence.xmi`, représentés en figures 5 et 6, qui ne sont pas censés pouvoir finir. Le processus blocage en raison des WorkSequences qui ne permettent à aucune activité de commencer. Le processus concurrence en raison des ressources disponibles qui ne sont pas suffisantes pour les deux processus qui ont lieu en parallèle (au moins pendant un certain temps).

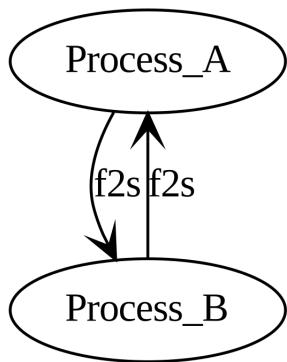


FIGURE 5 – Modèle blocage

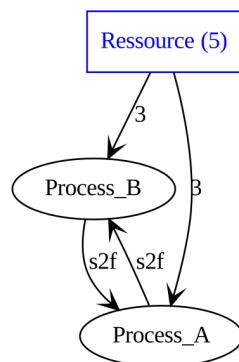


FIGURE 6 – Modèle concurrence

Nous avons créé un exemple de modèle SimplePDL non validé par nos contraintes OCL. Il correspond au fichier `Process-batard.xmi`. Cela permet de vérifier que toutes les contraintes sont bien implémentées.

## 4.2 Exemples de modèles PetriNet

Les figures 7 et 8 représentent des exemples de modèles PetriNet validés par notre métamodèle et nos contraintes OCL. Ils correspondent aux fichiers `Network-saisons.xmi` et `Network-readarc.xmi`.

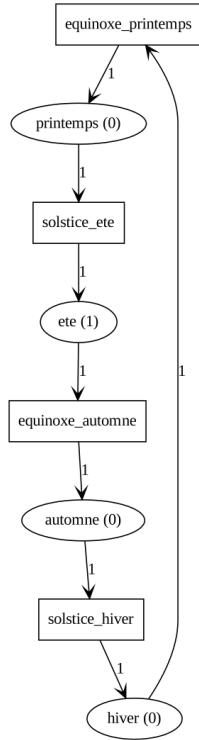


FIGURE 7 – Modèle saisons

Nous avons créé un exemple de modèle PetriNet non validé par nos contraintes OCL. Il correspond au fichier `Network-batard.xmi`.

## 5 Transformation SimplePDL vers PetriNet

Pour utiliser la boîte à outils Tina, nous avons besoin de transformer notre processus en réseau de Petri. Nous avons implémenté deux méthodes différentes de transformation d'un modèle conforme à SimplePDL en un modèle conforme à Petrinet : en utilisant Java et en utilisant ATL. Les deux suivent les principes de transformation suivants.

- Une activité se traduit par un ensemble de places et de transitions permettant de décrire son exécution et son état. Cet ensemble est décrit par la figure 10.
- Une dépendance se traduit par un arc en lecture seule (readarc) reliant la place *started* ou *finished* d'une activité à la transition *start* ou *finish* d'une autre (figure 11).
- Une ressource se traduit par une place dotée d'autant de jetons qu'il y a de ressources disponibles.
- Les besoins d'une activité en une ressource sont traduits par des arcs reliant la ressource aux transitions *start* et *finish* de l'activité et dotés d'un poids égal à la demande de ressource par cette activité (figure 9).
- Les commentaires ne sont pas traduits car ils n'ont pas de rôle dans l'exécution du processus.

Par exemple, notre modèle développement se traduit en réseau de Petri ainsi que montré par la figure 12.

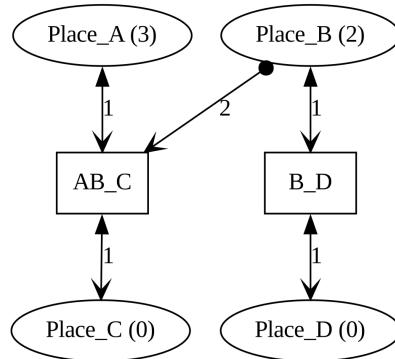


FIGURE 8 – Modèle ReadArc

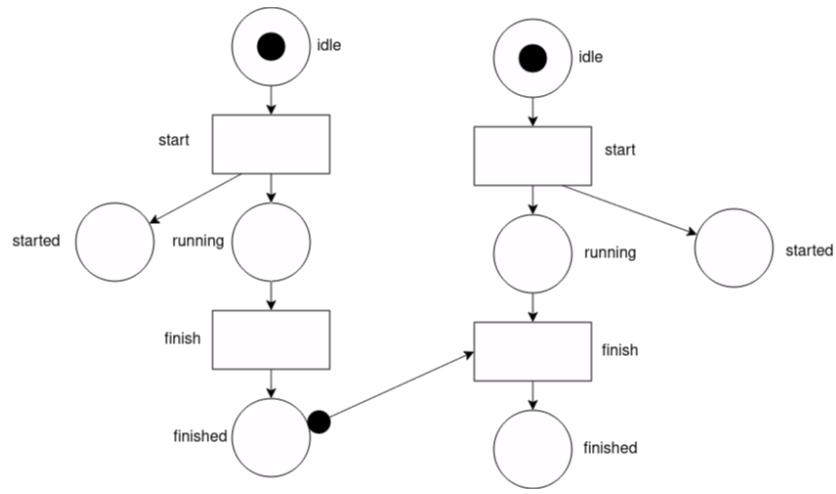


FIGURE 9 – WorkSequence en réseau de Petri

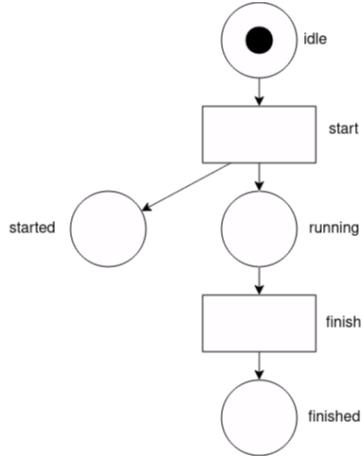


FIGURE 10 – WorkDefinition en réseau de Petri

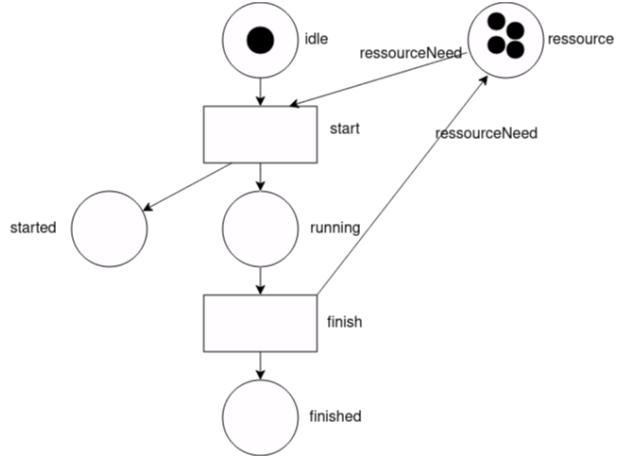


FIGURE 11 – Ressource en réseau de Petri

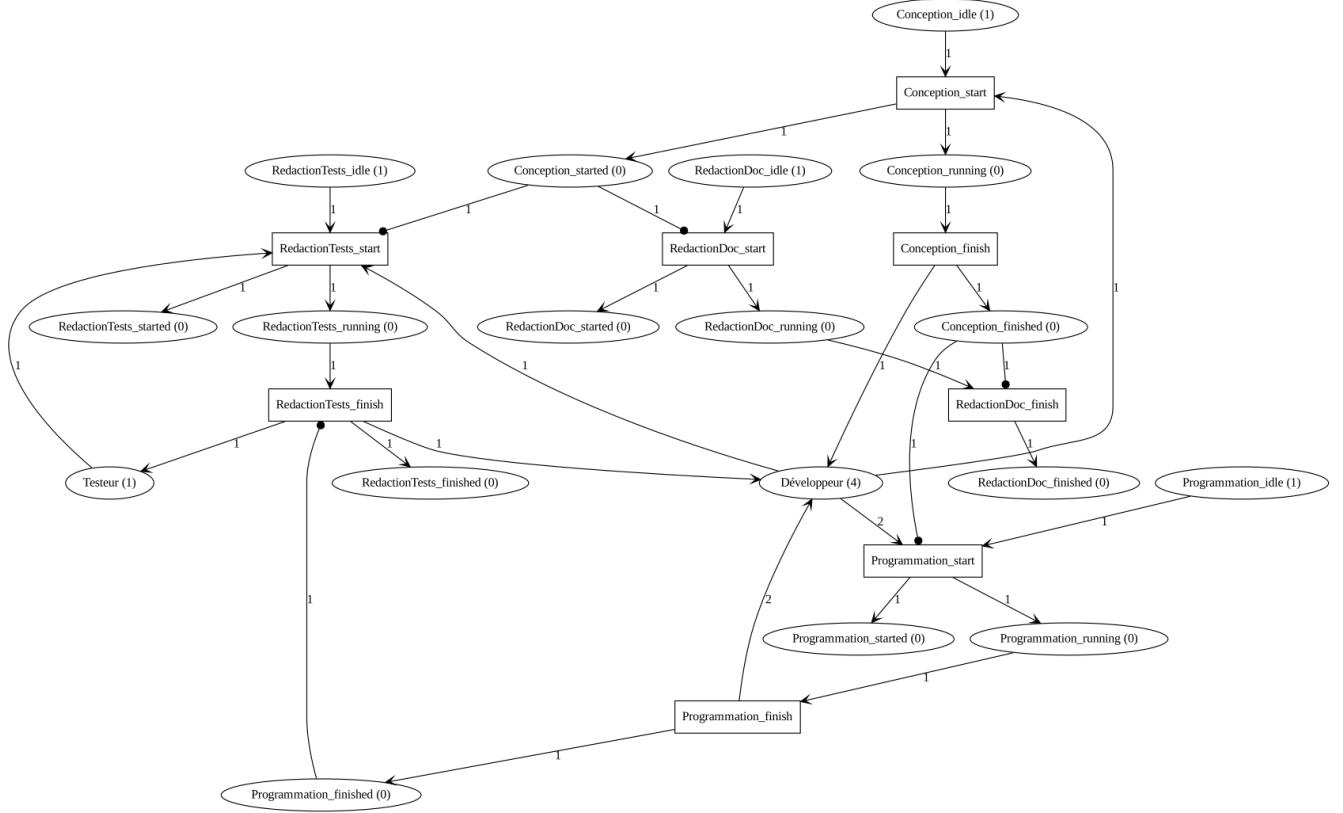


FIGURE 12 – Modèle développement sous forme de réseau de Petri

## 5.1 En utilisant EMF/Java

La première méthode consiste à manipuler des modèles EMF grâce à Java. Le code de cette transformation se trouve dans le fichier `PetriNetFromSimplePDL.java`.

Cette méthode est pénible car elle nécessite de spécifier manuellement le chargement des packages et des ressources nécessaires, puis l'attribution des liens entre chacun des éléments instanciés.

## 5.2 En utilisant ATL

La seconde méthode consiste à utiliser l'outil ATL, permettant de spécifier des règles de production plutôt que les étapes précises de la transformation. Le code de cette transformation se trouve dans le fichier `simplepdlToPetrinet.atl`.

## 6 Validation de la transformation

Pour valider la transformation SimplePDL vers PetriNet, nous avons évalué grâce à la génération d'un fichier Itl les invariants suivants :

- Chaque activité est soit non commencée, soit en cours, soit terminée
- Une activité terminée n'évolue plus
- Une activité en attente ne peut pas être démarrée
- Une activité ne peut pas être terminée ou en cours d'exécution sans être démarrée

Ensuite, à l'aide de l'outil selt, on a pu observer que les invariants étaient vérifiés pour nos modèles SimplePDL.

## 7 Transformation modèle à texte

A l'aide d'Acceleo, nous pouvons transformer les modèles PetriNet en syntaxe Tina. Le code de cette transformation se trouve dans le fichier `toTina.mtl`.

Pour le processus développement, nous avons par exemple obtenu le résultat suivant à partir de sa transformation en réseau de Petri :

```
net developpement
  pl Developpeur (4)
  pl Testeur (1)
  pl Conception_idle (1)
  pl Conception_started (0)
  pl Conception_running (0)
  pl Conception_finished (0)
  pl RedactionDoc_idle (1)
  pl RedactionDoc_started (0)
  pl RedactionDoc_running (0)
  pl RedactionDoc_finished (0)
  pl Programmation_idle (1)
  pl Programmation_started (0)
  pl Programmation_running (0)
  pl Programmation_finished (0)
  pl RedactionTests_idle (1)
  pl RedactionTests_started (0)
  pl RedactionTests_running (0)
  pl RedactionTests_finished (0)
  tr Conception_start Conception_idle Developpeur -> Conception_started Conception_running
  tr Conception_finish Conception_running -> Conception_finished Developpeur
  tr RedactionDoc_start RedactionDoc_idle Programmation_started ?1 -> Redaction-
Doc_started RedactionDoc_running
  tr RedactionDoc_finish RedactionDoc_running Programmation_finished ?1 -> Redaction-
Doc_finished
  tr Programmation_start Programmation_idle Developpeur*2 Conception_finished ?1 -> Pro-
grammation_started Programmation_running
  tr Programmation_finish Programmation_running -> Programmation_finished Developpeur*2
  tr RedactionTests_start RedactionTests_idle Developpeur Testeur Conception_started ?1 ->
RedactionTests_started RedactionTests_running
  tr RedactionTests_finish RedactionTests_running Conception_finished ?1 Programma-
tion_finished ?1 -> RedactionTests_finished Developpeur Testeur
```

## 8 Développement d'un éditeur graphique pour SimplePDL

Le projet EMF du métamodèle SimplePDL nous permet de saisir des métamodèles à l'aide d'un éditeur arborescent. Cependant, cette syntaxe n'est pas toujours très pratique pour saisir un modèle. Pour visualiser et éditer plus simplement nos modèles, nous avons utilisé l'outil Sirius permettant de définir une syntaxe concrète graphique correspondant à une modélisation Ecore.

Nous avons définit les règles de représentation des éléments d'un modèle SimplePDL ainsi qu'une palette d'outils permettant de créer tous les éléments permettant d'en construire un. Elle se trouve dans le fichier `simplepdl.odesign`.

La figure 13 montre le modèle développement visualisé et arrangé avec Sirius.

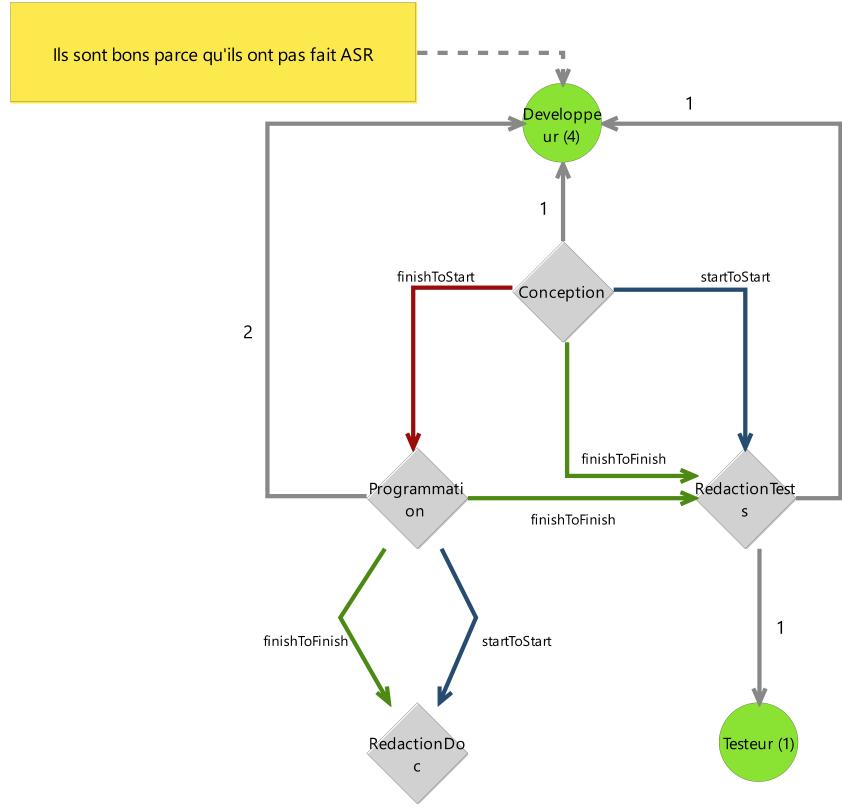


FIGURE 13 – Modèle développement visualisé avec Sirius

## 9 Définition d'une syntaxe textuelle pour SimplePDL

Nous venons de définir une syntaxe concrète graphique avec Sirius pour saisir nos modèles plus facilement qu'avec l'éditeur arborescent. Nous pouvons également définir une syntaxe concrète textuelle à l'aide l'outil XText.

Nous avons engendré un nouveau métamodèle PDL1 à l'aide de XText, dont la structure est proche de celle de SimplePDL. Celle-ci est contenue dans le fichier `PDL1.xtext`. On peut ainsi saisir un modèle conforme à PDL1 avec une syntaxe textuelle, puis transformer ce modèle en un modèle conforme à SimplePDL à l'aide d'une transformation ATL (voir fichier `pdl1ToSimplepdl.atl`).

Nous pouvons par exemple décrire le processus développement ainsi :

```

process Developpement {
    wd Conception [ ressourceNeed 1 from Developpeur ]
    wd RedactionDoc [ ]
    wd Programmation [ ressourceNeed 2 from Developpeur ]
    wd RedactionTests [ ressourceNeed 1 from Developpeur ressourceNeed 1 from Testeur ]
    ws f2f from Conception to RedactionTests
    ws s2s from Conception to RedactionTests
    ws f2s from Conception to Programmation
    ws f2f from Programmation to RedactionTests
    ws s2s from Programmation to RedactionDoc
    ws f2f from Programmation to RedactionDoc
    ressource Developpeur 4
    ressource Testeur 1
    note "Ils sont bons parce qu'ils ont pas fait ASR" [ Developpeur ]
}

```

Notons que nous aurions pu directement définir une syntaxe XText pour SimplePDL plutôt que de passer par un métamodèle intermédiaire. Cependant, utiliser un autre métamodèle permet d'éloigner la syntaxe textuelle de la structure de SimplePDL. Nous aurions pu davantage le mettre en évidence en utilisant par exemple le métamodèle PDL3 proposé en TP.

## 10 Vérification de la terminaison d'un processus

Pour vérifier la terminaison ou la non-terminaison d'un processus, nous avons générée à partir des modèles SimplePDL les propriétés LTL suivantes :

- A chaque instant, il existe un instant ultérieur où le processus est fini

- Le processus ne se finit jamais

Ensuite, à l'aide de l'outil selt, on peut observer quelles propriétés sont vérifiées ou non.

En testant sur le processus développement, on obtient que celui-ci peut toujours finir. On obtient aussi un contre-exemple prouvant qu'il se finit un jour.

## 11 Conclusion

Nous sommes maintenant capables de créer et de modifier un modèle conforme à SimplePDL à l'aide de l'éditeur arborescent, d'une syntaxe graphique à l'aide de Sirius ou d'une syntaxe textuelle à l'aide de XText.

Les contraintes liées au métamodèle ainsi que celles engendrées à l'aide d'OCL permettent de déterminer la validité du modèle.

Enfin, nous pouvons le transformer en réseau de Petri afin de pouvoir vérifier sa terminaison à l'aide des outils de model-checking de Tina et des propriétés LTL que nous avons définies.