



# Mini-projet IDM

Auteurs :

FARHAT Otman  
ABOUMEJD Wissal  
AKKAR Khadija

25 novembre 2023

Département Sciences du Numérique - Deuxième année - Logiciel  
2023-2024

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Les métamodèles SimplePDL et PetriNet</b>	<b>4</b>
2.1	SimplePDL . . . . .	4
2.2	Exemple de modèle simplePDL . . . . .	5
2.3	PetriNet . . . . .	6
2.4	Exemple de réseau de Petri . . . . .	8
<b>3</b>	<b>Les contraintes OCL associés</b>	<b>8</b>
3.1	Les contraintes ajoutées au SimplePDL . . . . .	8
3.2	Les contraintes principales ajoutées au réseau de Petri . . . . .	9
<b>4</b>	<b>L'éditeur graphique Sirius</b>	<b>10</b>
4.1	Introduction . . . . .	10
4.2	Définition de la syntaxe graphique avec Sirius . . . . .	10
<b>5</b>	<b>Le modèle Xtext</b>	<b>12</b>
<b>6</b>	<b>La transformation modèle à modèle SimplePDL2PetriNet</b>	<b>12</b>
6.1	Principe . . . . .	12
6.2	Transformation en java . . . . .	12
6.3	Transformation en ATL . . . . .	15
<b>7</b>	<b>La transformation d'un réseau de Pétri en Tina</b>	<b>15</b>
<b>8</b>	<b>Les propriétés LTL</b>	<b>16</b>
<b>9</b>	<b>Conclusion</b>	<b>17</b>

## Table des figures

1	Métamodèle initial de SimplePDL . . . . .	4
2	Métamodèle avancé de SimplePDL . . . . .	4
3	Exemple de modèle de procédé . . . . .	5
4	Editeur Arborescent ECore : SimplePDL . . . . .	6
5	Le métamodèle SimplePDL avec ressources . . . . .	6
6	Editeur Arborescent ECore : Réseau de PetriNet . . . . .	7
7	Le métamodèle PetriNet . . . . .	7
8	Exemple de réseau de Petri . . . . .	8
9	Les contraintes OCL du métamodèle PetriNet . . . . .	10
10	Syntaxe graphique du métamodel . . . . .	11
11	exemple Xtext . . . . .	12
12	Arborescence developpement.xmi . . . . .	13
13	Transformation en PetriNet partie 1 . . . . .	14
14	Transformation en PetriNet partie 2 . . . . .	15
15	developpement.net visualisé en tina . . . . .	16
16	Vérification des propriétés avec selt . . . . .	16

# 1 Introduction

Détecter les possibles anomalies dès les débuts d'un projet est crucial avant d'engager ses phases avancées. Pour cette raison, l'utilisation de méta-modèles et de simulations est primordiale afin de confirmer la validité des schémas décrivant le projet.

Dans cette optique, notre but est d'établir un processus de validation des modèles de flux de travail simples, visant à déterminer la viabilité d'un processus dans son ensemble. Pour ce faire, plusieurs étapes clés seront entreprises :

1. Création de représentations métaphoriques à travers une approche Ecore.
2. Définition de règles spécifiques que ces modèles doivent respecter en utilisant OCL.
3. Conversion du modèle de flux de travail en un schéma de réseau adaptatif à l'aide de Java/ATL.
4. La vérification du modèle à l'aide de l'outil Tina

## 2 Les métamodèles SimplePDL et PetriNet

### 2.1 SimplePDL

Le SimplePDL est un langage de métamodélisation qui trouve son utilité dans la création de modèles de processus. Deux variantes de langages sont disponibles pour le SimplePDL : une version simplifiée dédiée à la description des procédés de développement, et une version plus avancée qui intègre la notion de ProcessElement en tant que généralisation à la fois de WorkDefinition (activité) et de WorkSequence (dépendance).

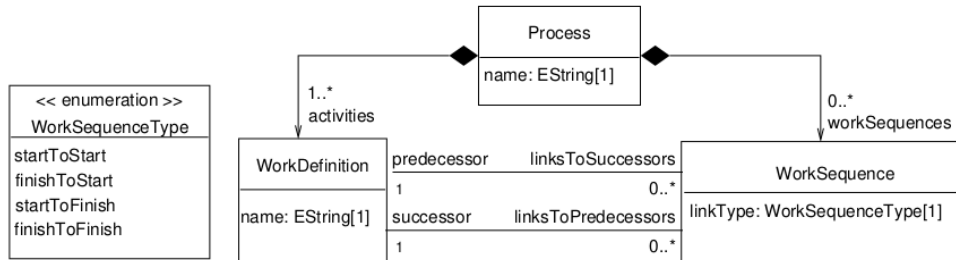


FIGURE. 1 – Métamodèle initial de SimplePDL

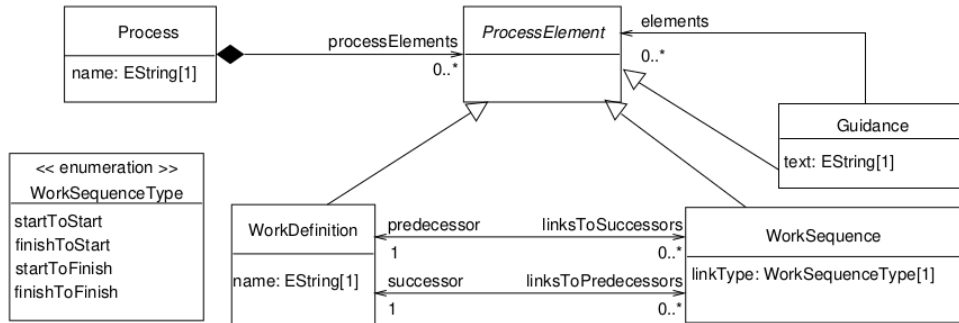


FIGURE. 2 – Métamodèle avancé de SimplePDL

Ainsi, le concept de processus (Process) se compose d'un ensemble d'activités (WorkDefinition) qui représentent les diverses tâches à accomplir au cours du développement. Une activité peut être

liée à une autre (WorkSequence), et une contrainte d'ordonnancement sur le début ou la fin de la seconde activité est spécifiée à l'aide de l'énumération WorkSequenceType, définissant l'attribut linkType.

## 2.2 Exemple de modèle simplePDL

Parmi ces modèles, celui que nous privilégierons principalement dans le cadre de notre mini-projet est le modèle de procédé suivant :

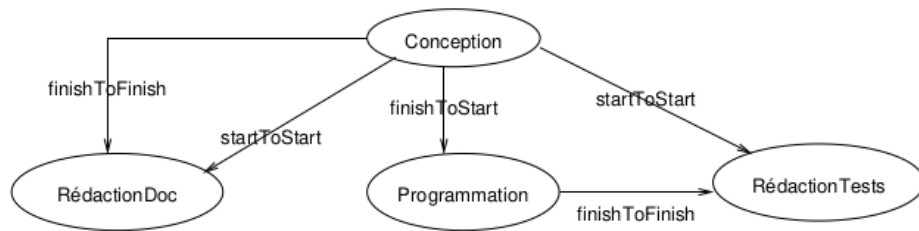


FIGURE. 3 – Exemple de modèle de procédé

Afin d'accomplir une activité, il est probable qu'elle nécessite une ou plusieurs ressources pour atteindre ses objectifs. Une ressource est définie par un nom (EString) décrivant son type et une quantité d'occurrences (Eint). Lorsqu'une activité débute son exécution, elle réserve des occurrences d'une ressource, et ces occurrences sont utilisées exclusivement par cette activité jusqu'à l'achèvement de sa réalisation.

Pour intégrer cette fonctionnalité au métamodèle de SimplePDL, deux nouvelles classes doivent être ajoutées :

- **Ressource** : caractérisée par son nom (EString) décrivant son type et le nombre d'occurrences (Eint).;
- **Amount** : représentant le nombre d'occurrences prises par une activité parmi celles d'une ressource pour mener à bien son exécution.

Il est important de noter que chaque activité (WorkDefinition) peut potentiellement nécessiter plusieurs ressources, ce qui se traduit par la composition de plusieurs classes Allocate.

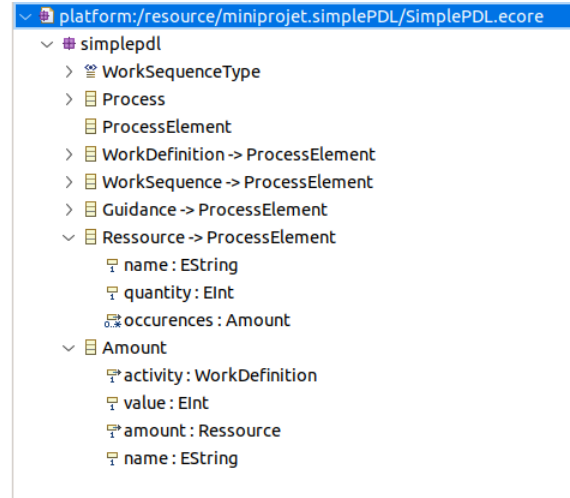


FIGURE. 4 – Editeur Arborescent ECore : SimplePDL

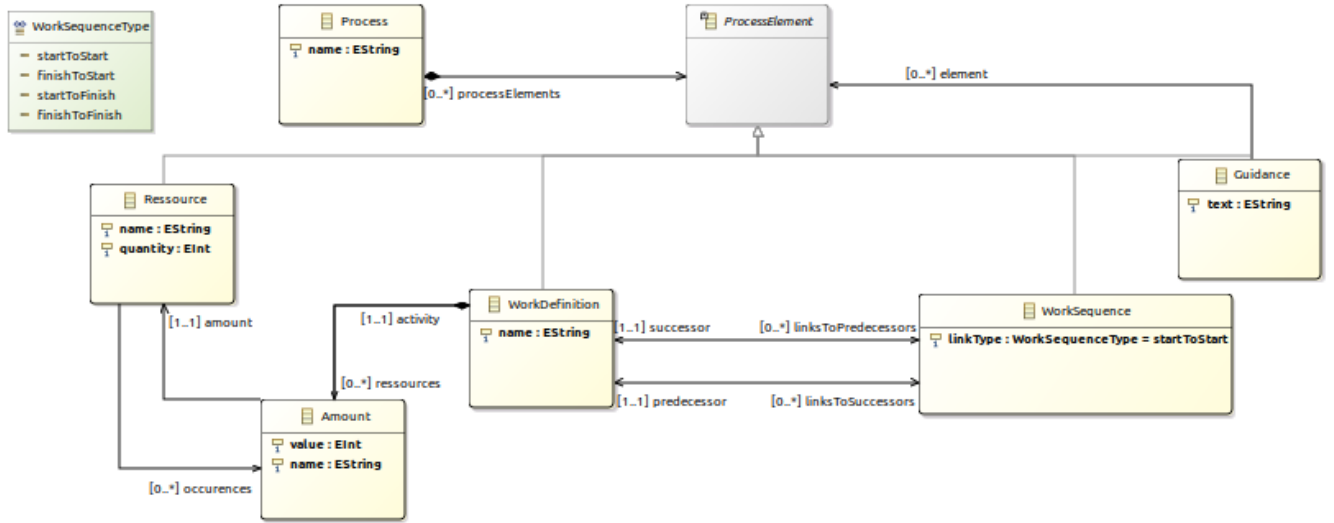


FIGURE. 5 – Le métamodèle SimplePDL avec ressources

## 2.3 PetriNet

Un réseau de Petri (PetriNet) se compose de nœuds (Node), qui peuvent être des places (Place) ou des transitions (Transition). Ces nœuds sont interconnectés par des arcs (Arc), classifiés en arcs normaux ou readArcs (ArcKind). Le poids d'un arc (jetons) indique le nombre de jetons consommés dans la place source ou ajoutés à la place de destination. Cependant, dans le cas d'un readArc, le poids représente simplement le test de la présence d'un nombre correspondant de jetons dans la place source. L'état actuel du réseau de Petri est représenté par l'attribut jetons d'une place.

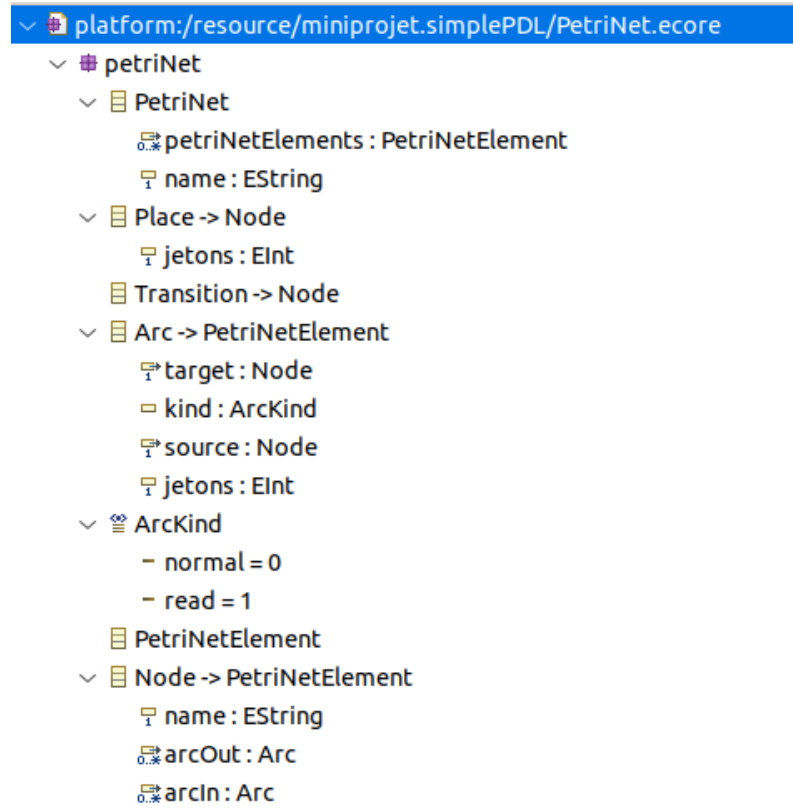


FIGURE. 6 – Editeur Arborescent ECore : Réseau de PetriNet

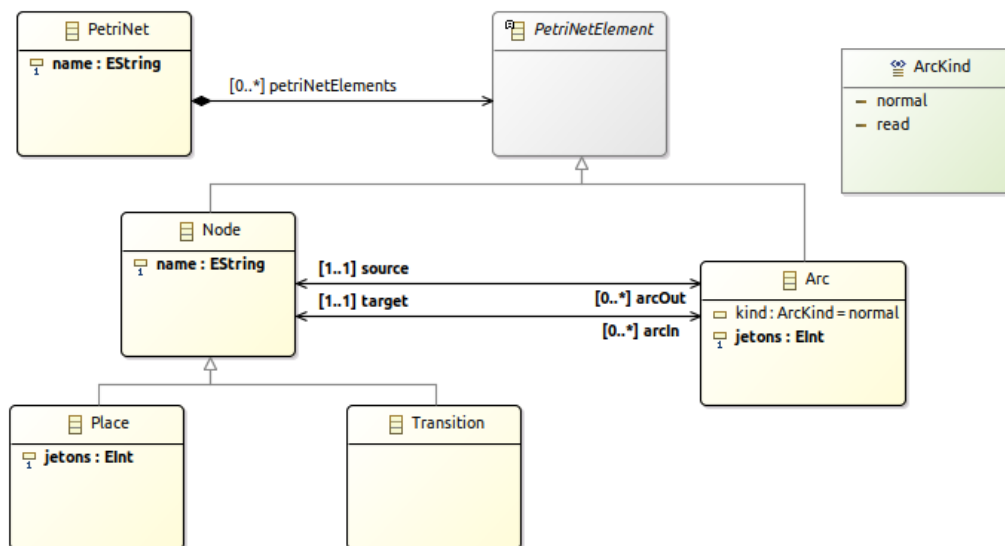


FIGURE. 7 – Le métamodèle PetriNet

## 2.4 Exemple de réseau de Petri

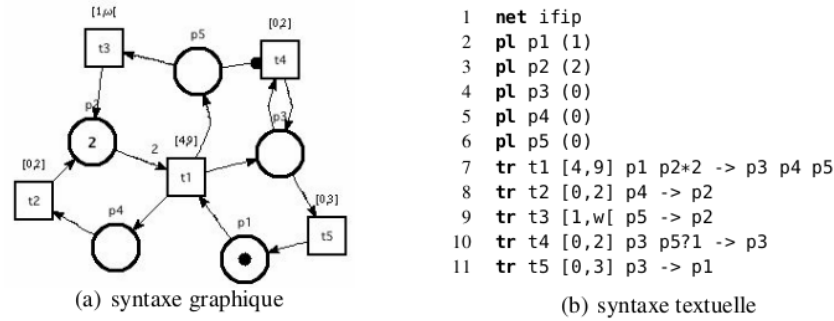


FIGURE. 8 – Exemple de réseau de Petri

## 3 Les contraintes OCL associés

Nous avons opté pour l'utilisation d'Ecore afin de conceptualiser un méta-modèle dédié aux processus. Néanmoins, le langage de méta-modélisation Ecore présente certaines limitations pour exprimer l'ensemble des contraintes requises par les modèles de processus.

Pour remédier à cette lacune, nous enrichissons la description structurelle et sémantique statique du méta-modèle élaboré en Ecore en intégrant des contraintes formulées à l'aide du langage OCL.

Ainsi, le méta-modèle Ecore, conjointement avec les contraintes OCL, détaille la syntaxe abstraite du langage de modélisation en question.

### 3.1 Les contraintes ajoutées au SimplePDL

— **contrainte 1 :**

```
context WorkDefinition
inv uniqueWD : self.Process.processElements
->select(p | p.ocIsKindOf(WorkDefinition))
->collect(p | p.ocIsType(WorkDefinition))
->forAll (a | self = a or self.name <> a.name)
```

Cette contrainte, définie dans le contexte d'une WorkDefinition, garantit l'unicité des noms au sein des éléments de processus de la WorkDefinition, assurant ainsi qu'aucune autre WorkDefinition n'a le même nom que celle-ci.

— **contrainte 2 :**

```
context WorkSequence
inv nonReflexive :
self.predecessor <> self.successor
```

Cette contrainte, spécifiée dans le contexte d'une WorkSequence, stipule que la dépendance entre une tâche antérieure (predecessor) et une tâche succédente (successor) ne peut pas être réflexive, c'est-à-dire qu'une tâche ne peut pas dépendre d'elle-même.



— **contrainte 3 :**

```
context WorkDefinition
inv twoCharacters : if self.name.ocIsUndefined() then false
                  else self.name.size() >= 2
                  endif
```

Cette contrainte, définie dans le contexte d'une WorkDefinition, impose que le nom d'une activité doit être composé d'au moins deux caractères. La contrainte vérifie si le nom est défini et, dans ce cas, s'assure que sa longueur est supérieure ou égale à deux.

— **contrainte 4 :**

```
context Ressource
inv uniqueRessource : self.Process.processElements
->select(p | p.ocIsKindOf(Ressource))
->collect(p | p.ocIsType(Ressource))
->forAll (a | self = a or self.name <> a.name)
```

Cette contrainte, définie dans le contexte d'une Ressource, établit qu'au sein d'un même processus, deux ressources ne peuvent pas partager le même nom. Elle s'assure que, pour chaque élément de processus étant une ressource, la ressource en question ne peut pas avoir le même nom qu'une autre ressource dans le même processus.

— **contrainte 5 :**

```
context Ressource
inv validQty : self.quantity > 0
```

Cette contrainte, formulée dans le contexte d'une Ressource, énonce que la quantité initiale d'une ressource doit être strictement positive. En d'autres termes, la quantité d'une ressource doit être supérieure à zéro pour être valide.

— **contrainte 6 :**

```
context Amount
inv occMax : self.value <= self.amount.quantity
```

Cette contrainte, déclarée dans le contexte d'une Amount, spécifie que le nombre d'occurrences ne peut pas dépasser la quantité totale de la ressource associée. En d'autres termes, la valeur de l'occurrence doit être inférieure ou égale à la quantité totale de la ressource à laquelle elle est liée.

## 3.2 Les contraintes principales ajoutées au réseau de Petri

Tout comme pour le métamodèle SimplePDL, le métamodèle de réseau de Petri que nous avons créé ne couvre que la syntaxe statique. Pour compléter cette représentation, nous introduisons des contraintes OCL qui définissent la syntaxe abstraite du réseau de Petri.

Les contraintes OCL associées à ce métamodèle sont illustrées dans la figure ci-dessous.

```

import 'PetriNet.ecore'

package petriNet

context PetriNet

inv NonNull_petriNetElements('The \'petriNetElements\' property of "' + self.toString() + '" is null'):
    petriNetElements <> null

--le nom d'une petrinet doit etre valide
context PetriNet
inv validName:
    self.name.matches('[A-Za-z_][A-Za-z0-9_]*')

--un arc peut lier qu'une transition et une place
context Arc
inv nodeLinks : if self.source.oclIsKindOf(Transition) then self.target.oclIsKindOf(Place)
                else self.target.oclIsKindOf(Transition)
                endif

/* Le marquage initial des places */
context Place
inv nbInitialJetons: self.jetons >= 0

/* Les jetons consommés d'une noeud vers un autre est non nul */
context Arc
inv jetonsConsom : self.jetons >= 1

endpackage

```

FIGURE. 9 – Les contraintes OCL du métamodèle PetriNet

## 4 L'éditeur graphique Sirius

### 4.1 Introduction

L'importance des syntaxes graphiques se manifeste pleinement dans la visualisation des modèles, conférant ainsi une lisibilité du modèle. Dans cette optique, nous avons utilisé l'outil Sirius d'Eclipse, fondé sur les technologies Eclipse Modeling (EMF). Cet outil permet la création d'un éditeur graphique à partir d'un modèle Ecore.

### 4.2 Définition de la syntaxe graphique avec Sirius

Nous avons débuté avec la syntaxe abstraite du DSML telle qu'elle avait été établie par le modèle Ecore précédemment exposé. Nous avons élaboré une syntaxe graphique pour ce métamodèle, offrant ainsi la possibilité de définir visuellement, pour un Process, les WorkDefinitions, les WorkSequences, les Ressources ainsi que la Guidance. Cette configuration est illustrée de manière détaillée dans la figure ci-dessous.

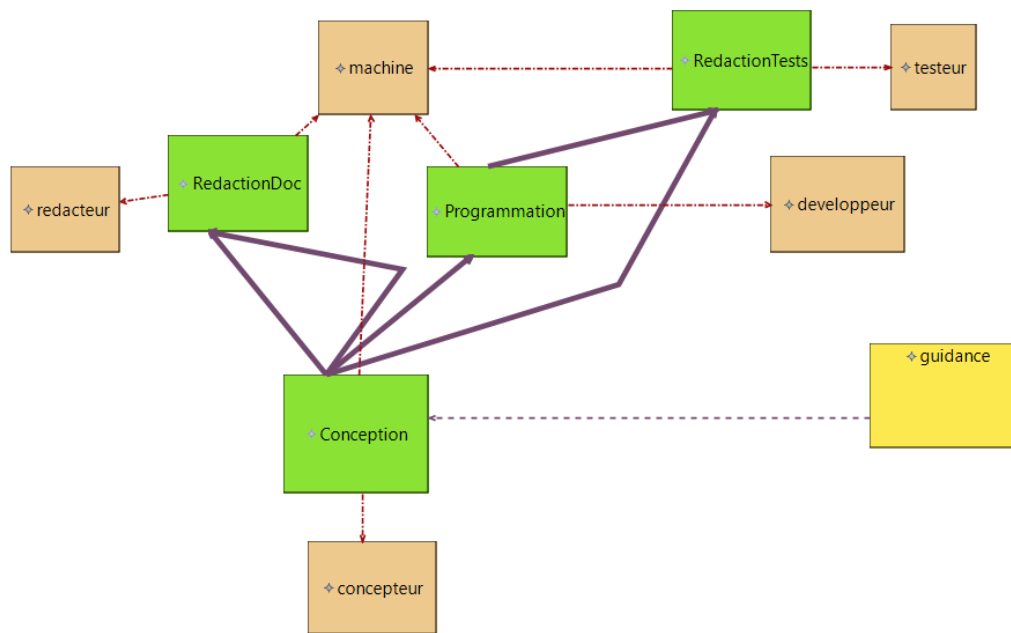


FIGURE. 10 – Syntaxe graphique du métamodèle

## 5 Le modèle Xtext

Une autre méthode de la visualisation des modèles d'une manière plus lisible est la méthode textuelle, il s'agit de l'utilisation de l'outil Xtext qui propose un rédacteur syntaxique doté de plusieurs options. Le fichier PDL1.xtext contient la description XText décrivant la syntaxe associée à SimplePDL. Cette description engendre la syntaxe spécifique à un exemple donné.

```
1 process Process {
2   ressource machine 4
3   ressource testeur 2
4   ressource redacteur 1
5   ressource developpeur 2
6   ressource concepteur 3
7   wd conception amountofressources ( value 2 of "concepteur" , value 2 of "machine",)
8   wd redactionDoc amountofressources ( value 1 of "redacteur" , value 1 of "machine",)
9   wd programmation amountofressources ( value 2 of "developpeur" , value 3 of "machine",)
10  wd redactionTest amountofressources ( value 1 of "machine", value 1 of "testeur" ,)
11  ws s2s from conception to redactionTest
12  ws s2s from conception to redactionDoc
13  ws f2f from conception to redactionDoc
14  ws f2f from programmation to redactionTest
15  ws f2s from conception to programmation
16  note of developpeur : "that's me, Otman Farhat"
17 }
```

FIGURE. 11 – exemple Xtext

## 6 La transformation modèle à modèle SimplePDL2PetriNet

### 6.1 Principe

### 6.2 Transformation en java

La transformation de SimplePDL vers PetriNet se base sur l'EMF, ce dernier nous permet de générer le code java à l'aide du modèle genmodel qui nous génère à son tour 3 projets : simplePDL, simplePDL.edit et simplePDL.editor.

Dans cette transformation, on a représenté une Workdéfinition par 4 places, afin de représenter les 4 états de chaque activité : Ready, Started, Running et Finished. Une Worksequence par un arc reliant les places PlaceStarted ou PlaceFinished avec les transitions TransitionStart ou TransitionFinish (cela dépend de la nature de la liaison : StartToStart, StartToFinish, etc... ).

Quant à nos ressources, on les a transformées en places aussi, et on a représenté leur nombre de jetons par la quantité initiale disponible de chaque ressource.

La demande "Amount" qui représente les occurrences de ressources nécessaires à la réalisation d'une activité est transformé en arc. Chaque ressource est relié avec deux transitions de la même Workdefinition; l'un vers la transition start, l'autre vers la transition finish, afin de rendre les jetons disponible après la fin de l'activité et de vérifier la condition qui dit "les occurrences de ressources [...] sont prises au moment de son démarrage et rendues à la fin de son exécution "

Dans cette transformation EMF, on applique le code java SimplepdlToPetriNet sur un exemple simplePDL représenté par l'arborescence suivante :

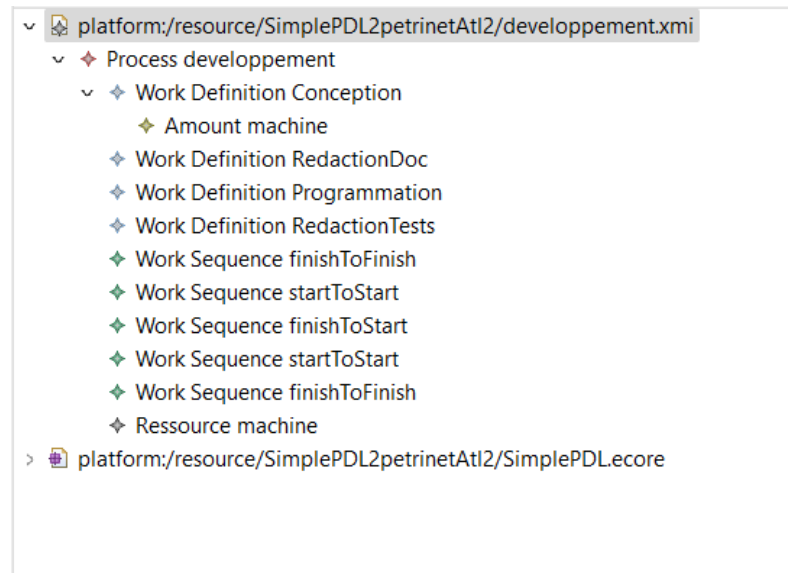


FIGURE. 12 – Arborescence developpement.xml

afin de générer son equivalent en petriNet sous la forme ci-dessous :



FIGURE. 13 – Transformation en PetriNet partie 1

- ◆ Place Programmation\_running
- ◆ Place Programmation\_ready
- ◆ Place Programmation\_started
- ◆ Place Programmation\_finished
- ◆ Transition Programmation\_finish
- ◆ Transition Programmation\_start
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Place RedactionTests\_running
- ◆ Place RedactionTests\_ready
- ◆ Place RedactionTests\_started
- ◆ Place RedactionTests\_finished
- ◆ Transition RedactionTests\_finish
- ◆ Transition RedactionTests\_start
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 2
- ◆ Arc 2
- ◆ Place machine


>  <http://petrinet>

FIGURE. 14 – Transformation en PetriNet partie 2

### 6.3 Transformation en ATL

Le but est de transformer le modèle SimplePDL en un modèle petrinet, à l'aide d'ATL. Le principe est le même que la transformation Java , la seule différence est le langage de programmation

## 7 La transformation d'un réseau de Pétri en Tina

Nous allons maintenant nous concentrer sur la conversion modèle en texte des modèles de réseaux de Petri, la syntaxe de texte requise est la syntaxe utilisée par l'outil Tina, la syntaxe d'extension .net.

Ainsi, la boîte à outils Tina nous permettra de visualiser graphiquement le modèle et de le simuler à l'aide de l'outil nd.

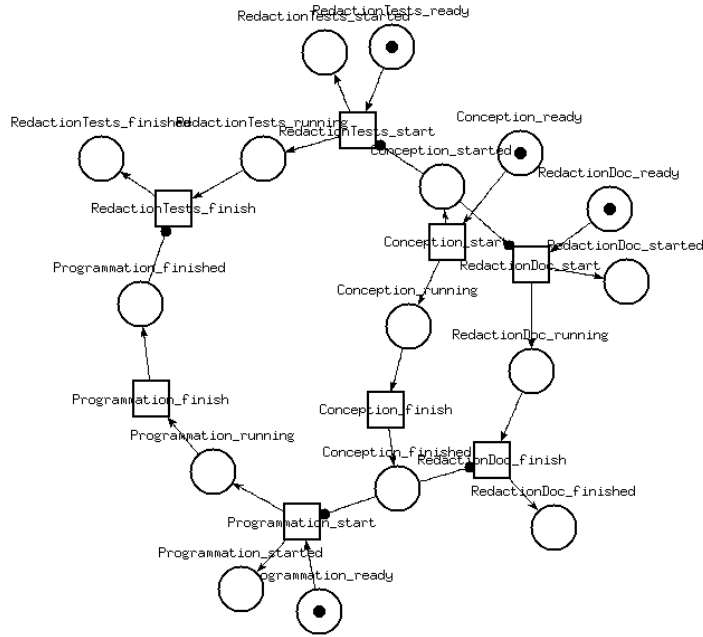


FIGURE. 15 – developpement.net visualisé en tina

## 8 Les propriétés LTL

Les propriétés LTL permettent de vérifier une transformation d'un modèle à un modèle et la boîte à outils Tina vérifie les propriétés en ce qui concerne le modèle à étudier.

Nous utilisons LTL pour vérifier la terminaison d'un processus et les invariants de SimplePDL pour valider la validation de notre projet.

```

developpement@developpement:~/developpement$ ./tina -p -S developpement.scn developpement.ktz -prelude developpement.ltl
selt version 3.7.0 -- 05/19/22 -- LAAS/ONRS
ktz loaded, 26 states, 47 transitions
0.003s

- source developpement.ltl;
operator ready : prop
operator started : prop
operator running : prop
operator finished : prop
TRUE
TRUE
FALSE
state 0: L.scc*25 Conception_ready Programming_ready RedactionDoc_ready RedactionTests_ready
-Conception_start->
state 1: L.scc*24 Conception_running Conception_started Programming_ready RedactionDoc_ready RedactionTests_ready
-Conception_finish->
state 2: L.scc*20 Conception_finished Conception_started Programming_ready RedactionDoc_ready RedactionTests_ready
-Programming_start->
state 3: L.scc*14 Conception_finished Conception_started Programming_running Programming_started RedactionDoc_ready RedactionTests_ready
-Programming_finish->
state 4: L.scc*8 Conception_finished Conception_started Programming_finished Programming_started RedactionDoc_ready RedactionTests_ready
-RedactionDoc_start->
state 5: L.scc*5 Conception_finished Conception_started Programming_finished Programming_started RedactionDoc_running RedactionDoc_started RedactionTests_ready
-RedactionDoc_finish->
state 6: L.scc*2 Conception_finished Conception_started Programming_finished Programming_started RedactionDoc_finished RedactionDoc_started RedactionTests_ready
-RedactionTests_start->
state 7: L.scc Conception_finished Conception_started Programming_finished Programming_started RedactionDoc_finished RedactionDoc_started RedactionTests_running RedactionTests_started
-RedactionTests_finish->
state 8: L.dead Conception_finished Conception_started Programming_finished Programming_started RedactionDoc_finished RedactionDoc_started RedactionTests_finished RedactionTests_started
-L.deadlock->
state 9: L.dead Conception_finished Conception_started Programming_finished Programming_started RedactionDoc_finished RedactionDoc_started RedactionTests_finished RedactionTests_started
[accepting all]
TRUE
TRUE
TRUE
0.011s

```

FIGURE. 16 – Vérification des propriétés avec selt

On remarque que la troisième propriété  $\square$  (dead  $\Rightarrow$  finish) n'est pas vérifiée dans notre exemple.



Il s'agit d'une impasse causée par un manque de ressources. En fait, en suivant le contre-exemple donné, nous nous sommes rendu compte qu'à un moment donné nous ne pouvions pas aller plus loin car RédactionTests-finished attendait la fin du développement, et le développement ne pouvait même pas démarrer car nous n'avions pas un nombre de machines suffisant. Nous sommes donc coincés.

## 9 Conclusion

Ce projet pratique nous a permis d'appliquer les concepts enseignés en cours et travaux pratiques pour agir sur l'ensemble des étapes de la transformation SimplePDL -> PetriNet.

Cette expérience a souligné l'importance d'une conception rigoureuse des modèles dans un projet, mettant en lumière les possibilités de model-checking à travers l'utilisation de l'outil Tina. En outre, il a révélé l'importance cruciale de la modélisation dans la résolution des problèmes, ainsi que la nécessité de maîtriser les processus de développement logiciel en exploitant des techniques de modélisation, de conception, de développement et de gestion de projet.

Ces techniques sont essentielles pour concevoir des applications modernes, exigeant l'intégration harmonieuse de composants hétérogènes.