Ingénierie dirigée par les modèles
Mini-projet
Groupe L34-02

Auteurs :

Mahmoud LAANAIYA – Mohamed Hamza KADRI

Professeur encadrant : Peter RIVIERE

Département Sciences du Numérique - Deuxième année
2021-2022

Table des matières

Table des figures

1 Introduction

Il est nécessaire de construire un métamodèle juste et précis afin de valider les différents modèles décrivant le projet car ce dernier représente une représentation de ces modeles tandis que les modéles sont des instanciations du métamodele. Et c'est grâce à ce métamodele qu'on pourra construire des langages de modélisations , des relations entre les modèles et leurs représentation, des contraintes générales sur les modèles et enfin des transformations de structure entre deux entités.

Dans ce mini-projet nous utiliserons le modèle suivant :

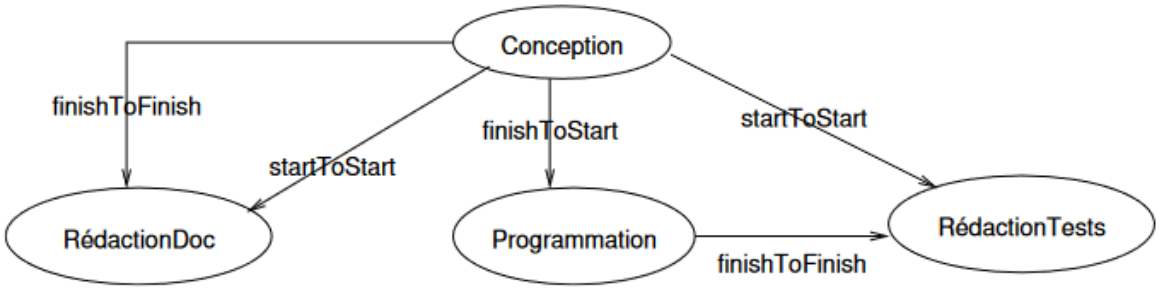

FIGURE 1 – Exemple de modèle de procédé

à noter que à ce modèle s'ajoute des ressources suivant le tableau suivant :

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

Nous allons suivre ainsi plusieurs étapes pour définir des métamodèles (petrinet,simplepdl) afin de représenter au mieux le modèle de procédé suivi :

- Définition des métamodèles avec Ecore.
- Définition des contraintes avec OCL.
- La transformation de ce modèle à un réseau de Pértri avec atl/JAVA.
- La représentation de ce modèle avec Sirius.
- mtl, Nous n'avons pas réussi à le faire (Voir la section des transformations à texte en utilisant Acceleo)

2 Métamodèle SimplePDL

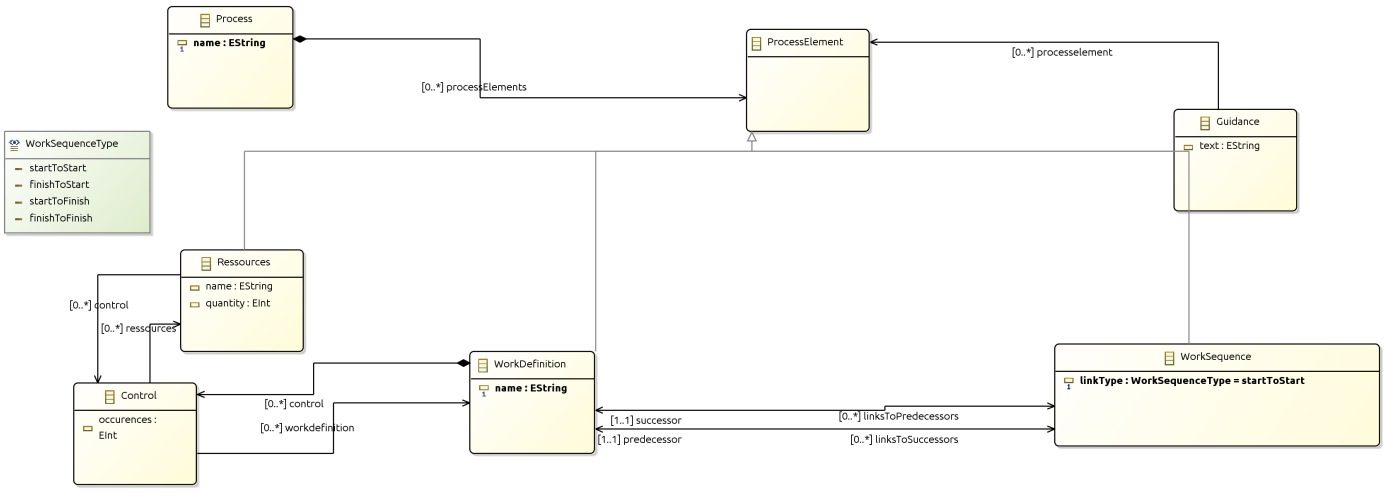

FIGURE 2 – Métamodèle simplePDL

On voit à travers ce métamodèle le principe de processus(Process) qui contient des activités(Work Definition) et ses activités sont reliées avec des relations (Work Sequence) et enfin des ressources (Ressources). Les activités sont ce que le processus doit réaliser, ils sont liées avec des relations de types : startToStart, startTofinish, finishTostart, finishToFinish. Ces relations représente le début et la fin d'une activité selon l'activité qui la précède ou la succède. Par exemple l'activité « Programmation » est reliée à « RédactionsTests » à travers finishtoFinish c'est-à-dire que la rédaction des tests ne peut se terminer que si la programmation se termine.

On remarque l'ajout d'une classe Control qui sert d'intermédiaire entre Ressources et WorkDefinition. La classe Ressources a comme attribut le nom de la ressource et sa quantité. L'intérêt de la classe Control est qu'elle possède un attribut (occurrences : int) qui représente le nombre de ressources demandées par l'activité. Comme son nom l'indique elle permet de contrôler cette allocation de ressources , par exemple s'il y a 5 concepteurs , il se peut que l'activité en demande moins ou plus.

Ce genre de situation nous pousse à définir des contraintes avec OCL. On a opté pour plusieurs contraintes afin d'éviter le plus de situations critiques. Parmi ces contraintes :

— La vérification des noms du processus et de ses composants.
— La vérification que deux entités ne possèdent pas le même nom.
— La non-réfléxivité des relations.
— La disponibilité des ressources demandées.
— La quantité des ressources n'est pas nulle.

Ci dessous un extrait du code OCL simplePDL :

```
/*Le nom d'une activité ne doit être composé que de lettres,chiffres et soulignés,un chiffre ne
 * peut pas être en première position
 */
context WorkDefinition
inv formatName: self.name.matches('[A-Za-z_][A-Za-z0-9_]*')

/*le nombre initial d'une ressource ne doit pas être nul */
context Ressources
inv notEmptyRessources: self.quantity > 0

/*le nombre de ressources demandées est disponible */
context Ressources
inv ressourcesDispobible: self
                -> select(p | p.oclIsKindOf(Control))
                -> collect(p | p.oclAsType(Control))
                -> forAll(occ | self.quantity >= occ.occurences)
endpackage
```

FIGURE 3 – Extrait du code SimplePDL.ocl

3 Métamodèle PetriNet

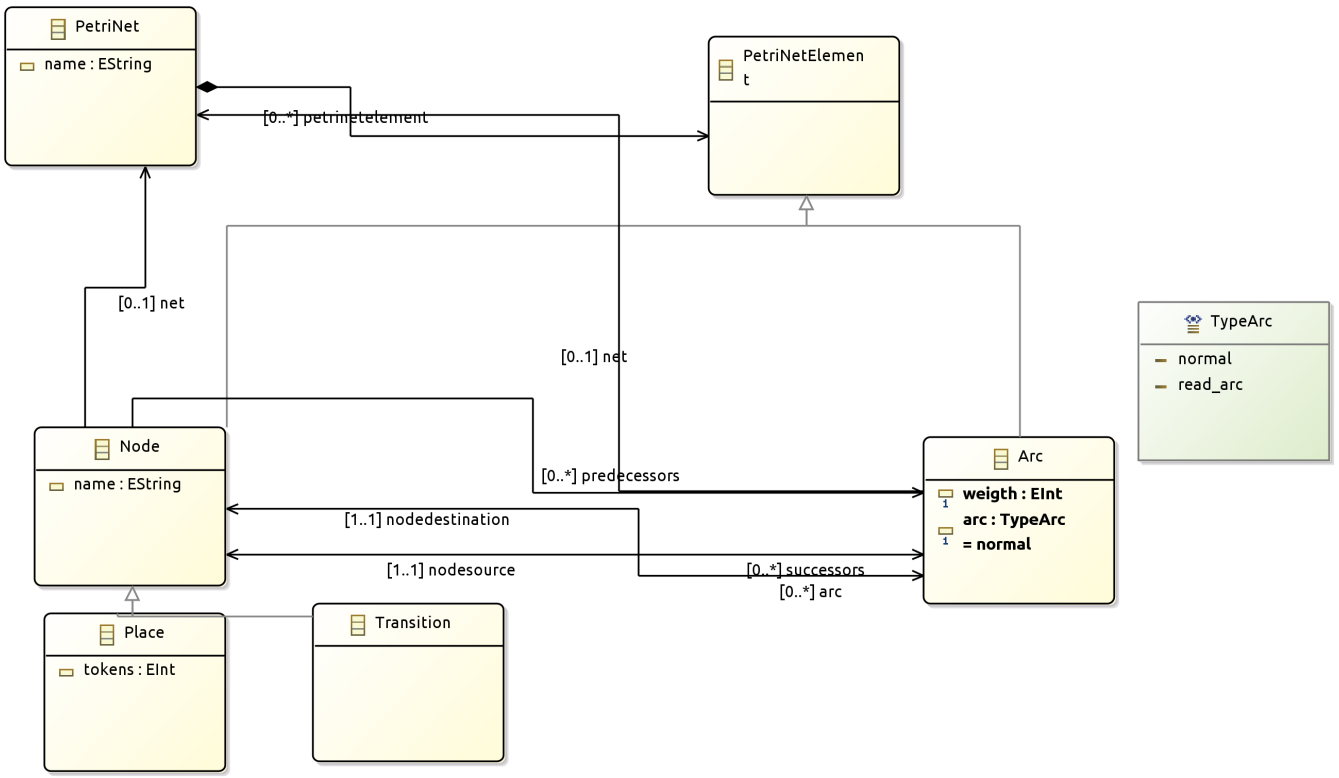

FIGURE 4 – Métamodèle PetriNet

On retrouve le concept de réseau de pétri (PetriNet) qui contient nœuds (Node) qui sont soient des transitions (Transition) soient des places (Places) . Les places ont un nombre initial de jetons d'où l'attribut (tokens : int) . On remarque que Les nœuds sont liés par des arcs (Arc) qui ont deux types(normal(place vers transition ou le contraire) , read_arc(connecte une place d'entrée à une transition)) regroupés dans une énumération (TypeArc). Les arcs ont aussi un attribut weigth qui définit le nombre de jetons qui sera consommé d'une place vers une transition ou l'inverse.

Comme pour le simplepdl il faut définir des contraintes avec OCL pour éviter certains problèmes. On a opté pour les contraintes suivantes :

- Impossibilité de relier deux places et deux transitions avec des arcs
- Un nœud ne peut pas être isolé
- Deux nœuds ne peuvent pas avoir le même nom
- Contraintes sur la consommation de jetons

Ci dessous un extrait du code OCL PetriNet :

```
context Place
inv marquageInitial: self.tokens >= 0

context Arc
inv liaisonPetri: self.nodesource.oclAsType(Place) <> self.nodedestination.oclAsType(Transition)
                or self.nodesource.oclAsType(Transition) <> self.nodedestination.oclAsType(Place)

context Place
inv validNamePlace('Invalid name: ' + self.name):
    self.name.matches('[P0-9_][0-9_]*')

context Transition
inv validNameTransition('Invalid name: ' + self.name):
    self.name.matches('[T0-9_][0-9_]*')

context Node
inv notSameName: self.PetriNet.petrinetelement
```

FIGURE 5 – Extrait du code PetriNet.ocl

4 Éditeur graphique simplepdl avec Sirius

Afin de mieux visualiser les modèles du procesuss il est important d'avoir une syntaxe graphique ou plutôt un éditeur graphique qui permet de créer des instances. Pour réaliser cet éditeur on a utilisé l'outil Sirius dans Eclipse qui permet de créer un éditeur graphique avec un métamodèle (Ecore).

En se basant sur notre Ecore on a pu créer un éditeur graphique aui permet de créer des WorkDefinition, des WorkSequence , des Ressources, des notes (Guidance) et les différentes relations qui les lient.

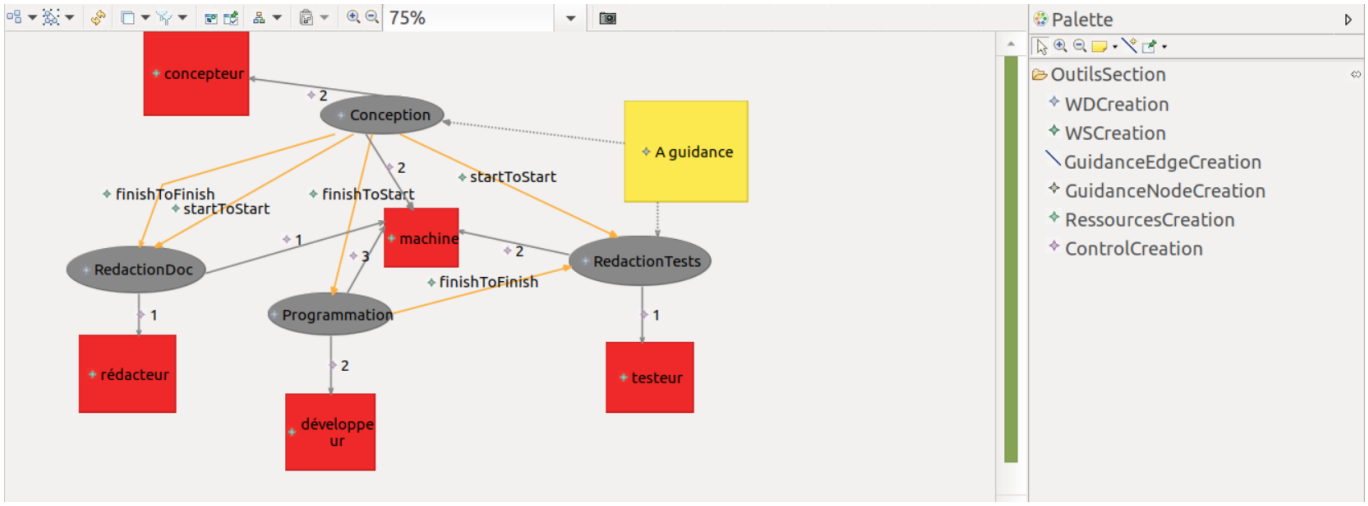

FIGURE 6 – L'éditeur graphique simplePDL

5 La transformation modèle à modèle

5.1 En Java

La transformation modèle à modèle Petri en Java se fait en plusieurs étapes :

— Chargement des packages SimplePDL et PetriNet
— Configuration de l'input(Modèle simplepdl) qu'on fournit au programme Java ainsi que l'output(Modèle PetriNet) 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 modèle PetriNet, en traduisant les WorkDefi- nitions et Resources en Places, puis les WorkSequences en Arcs.

5.1.1 Application

On execute notre code Java sur un modèle simplePDL(input-Figure 7), et on obtient un modèle PetriNet(output-Figure 8)

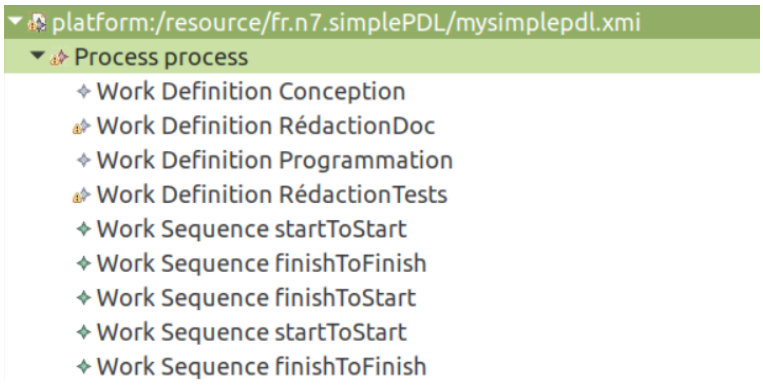

FIGURE 7 – Extrait du modèle simplepdl (input)

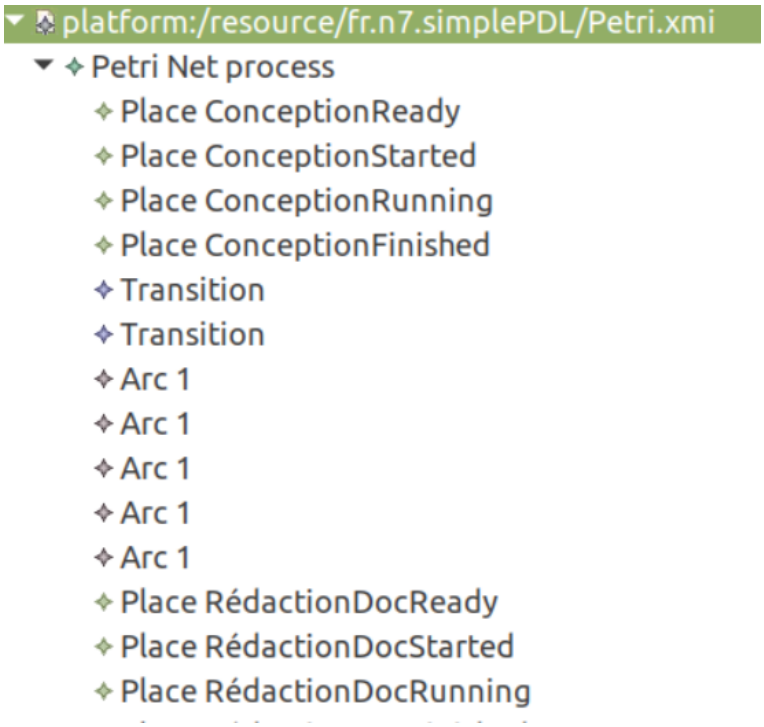

FIGURE 8 – Extrait du modèle petrinet (output)

5.2 En ATL

La transformation modèle à modèle Petri en ATL se fait analogiquement comme la transformation en Java, sauf que la transformation en ATL est plus efficace que celle en Java.

5.2.1 Application

On execute notre code ATL sur un modèle simplePDL(input-Figure 9), et on obtient un modèle PetriNet(output-Figure 10)

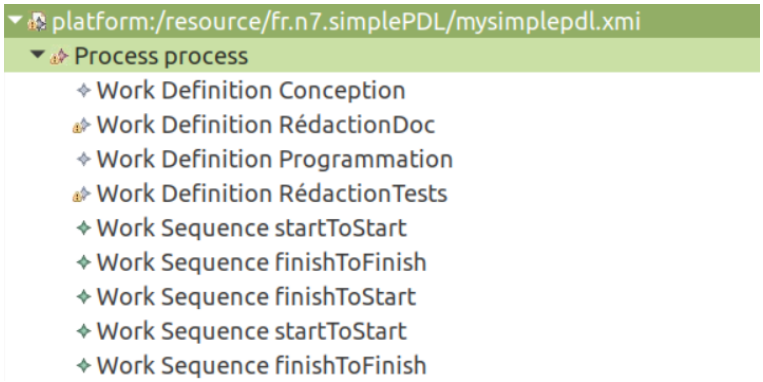

FIGURE 9 – Extrait du modèle simplepdl (input)

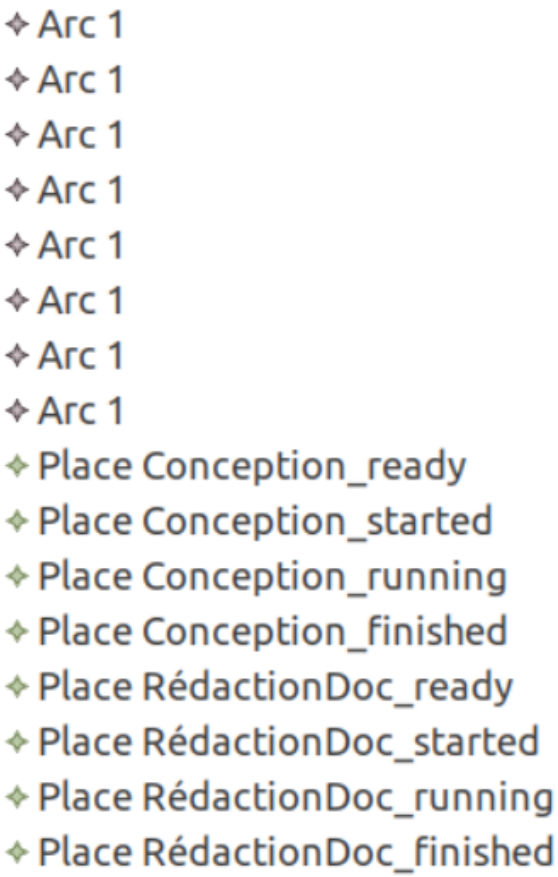

FIGURE 10 – Extrait du modèle petrinet (output)

6 Définir des syntaxes concrètes en utilisant Xtext

Pour faciliter la construction et la modification des modèles, il est souhaitable d'associer à une syntaxe abstraite une ou plusieurs syntaxes concrètes .
Ces syntaxes concrètes peuvent être textuelles ou graphiques. Pour la définition des syntaxes concrètes textuelles, nous utiliserons l'outil Xtext.

Nous avons utilisé SimplePDL.xtext pour définir des syntaxes concrètes où on peut définir des WorkDefinition, WorkSequence... avec des symbôles définis dans le xtext.

```
ProcessElement_Impl returns ProcessElement:
    {ProcessElement}
    'ProcessElement'
    ;

WorkDefinition returns WorkDefinition:
    'WorkDefinition'
    name=EString
    '{'
        ('linksToPredecessors' '(' linksToPredecessors+=[WorkSequence|ES
        ('linksToSuccessors' '(' linksToSuccessors+=[WorkSequence|EStrin
        'control' '{' control+=Control ( "," control+=Control)* '}'
    '}';

WorkSequence returns WorkSequence:
    'WorkSequence'
    '{'
```

FIGURE 11 – Extrait du code SimplePDL.xtext

7 Transformations à texte en utilisant Acceleo

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 Tina, à savoir la syntaxe en extention.

Pour vérifier le bon fonctionnement de cette transformation, on doit l'appliquer sur un modèle petrinet pour avoir un fichier .net qui sera de la forme indiquée dans la Figure 12.

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

FIGURE 12 – Syntaxe textuelle

Malheureusement nous n'avons pas réussi à générer ce .net. En effet, après avoir déployer les gréffons, nous n'avons pas trouver l'entrée Acceleo Model to Text. Aussi nous avons essayer d'executer le code mtl mais sans résultat (sachant qu'il n'y avaient pas d'erreurs sur la console).

```
46 [template public getArcsSource(arcs : OrderedSet(Arc)) post
47     [for (a : Arc | arcs)][a.nodesource.name/][if (a.arc = Ty
48 [/template]
49
50 [template public getArcsDestination(arcs : OrderedSet(Arc)) p
51     [for (a : Arc | arcs)][a.nodedestination.name/][if (a.we
52 [/template]
```

FIGURE 13 – Extrait du code mtl

8 Conclusion

Ce projet nous a permis de comprendre et mettre en pratique les notions vu en cours/TD, ainsi que la maîtrise des processus de développement logiciel en s'appuyant sur des techniques de développement et de gestion de projet.
Nous avons eu quelques difficultés dans ce projet, pariculièrement dans la partie Acceleo/mtl que nous n'avons pas réussi à la tester.