Rapport Mini-Projet IDM

Jonas Lavaur
Gabriel Rochaix–Yamamoto

Département Sciences du Numérique - Deuxième année
2022-2023

Sommaire

Table de Figures

Fichiers de l'archive

- simplePDL_diagramme.png
- petriNet_diagramme.png
- cuisiner.xmi
- developpement.xmi
- developpementAvecRessources.xmi
- petriNameProblems-ko.xmi
- petriNumberProblems-ko.xmi
- petriOtherProblems-ko.xmi
- processNameProblems-ko.xmi
- processOtherProblems-ko.xmi
- prodTampCons.xmi
- quatreSaisons.xmi
- src/PetriNet.ecore
- src/SimplePDL.ecore
- src/PetriNet.ocl
- src/SimplePDL.ocl
- src/SimplePDLToPetriNet.java
- src/SimplePDL2PetriNet.atl
- src/totina.mtl
- src/generateLTL.mtl

- src/terminaisonAbsolueLTL.mtl
- src/terminaisonInexistenceLTL.mtl
- src/PDL.xtext
- src/cuisiner.pdl
- src/developpementAvecRessources.pdl
- src/simplepdl.odesign
- src/developpementAvecRessources.simplepdl

1 Introduction

Ce projet a pour but de savoir si un processus décrit peut se terminer ou non. Nous avons donc commencé par définir un métamodèle Ecore de processus : SimplePDL. Pour vérifier si un processus peut se terminer ou non, nous utilisons les outils de model-checking définis sur les réseaux de Petri à l'aide de la boîte à outils Tina.

Nous avons donc défini un métamodèle Ecore de réseaux de Pétri : PetriNet, puis à l'aide de EMF/Java et ATL nous avons codé deux programmes qui permettent de transformer un modèle respectant le métamodèle SimplePDL.ecore en un modèle respectant PetriNet.ecore. Pour faciliter la création de ces modèles, nous avons créé une interface graphique qui permet de manipuler les modèles SimplePDL. Pour compléter les metamodèles Ecore avec une sémantique statique, nous avons écrit des contraintes OCL.

Enfin, nous avons codé un programme Acceleo qui permet de traduire un modèle PetriNet en un modèle .net utilisable par la boîte à outils Tina. Pour ajouter des conditions de logiques temporelles linéaires nous avons également codé un programme Acceleo qui prend en entrée un modèle SimplePDL et qui écrit un fichier à text ltl utilisable par la commande selt de Tina.

2 Définition des modèles Ecore SimplePDL avec Ressources et PetriNet (TP2)

2.1 Modèle Ecore SimplePDL avec Ressources

Nous avons modélisé les Ressources par une nouvelle classe `Ressource` de type ProcessElement qui est caractérisée par son nom et sa quantité initiale de la ressource donnée.

Pour modéliser l'utilisation des Ressources par les activités nous avons décidé de définir une nouvelle classe `UsefulRessource` caractérisé par un attribut `EInt usefulQuantity` et une référence `Ressource ressource`. L'arité de cette référence est de 1 car elle doit pouvoir savoir quelle Ressource l'activité utilise et une seule Ressource pour des raisons de Modélisation. Cette classe décrit une utilisation d'une quantité `usefulQuantity` de la `Ressource` référencée.

Il suffit donc ensuite d'ajouter à la classe `WorkDefinition` une référence de composition vers `UsefulRessource`. L'arité de cette référence est `0..*` car une activité peut ne pas utiliser de Ressource et peut utiliser plusieurs Ressouces différentes.

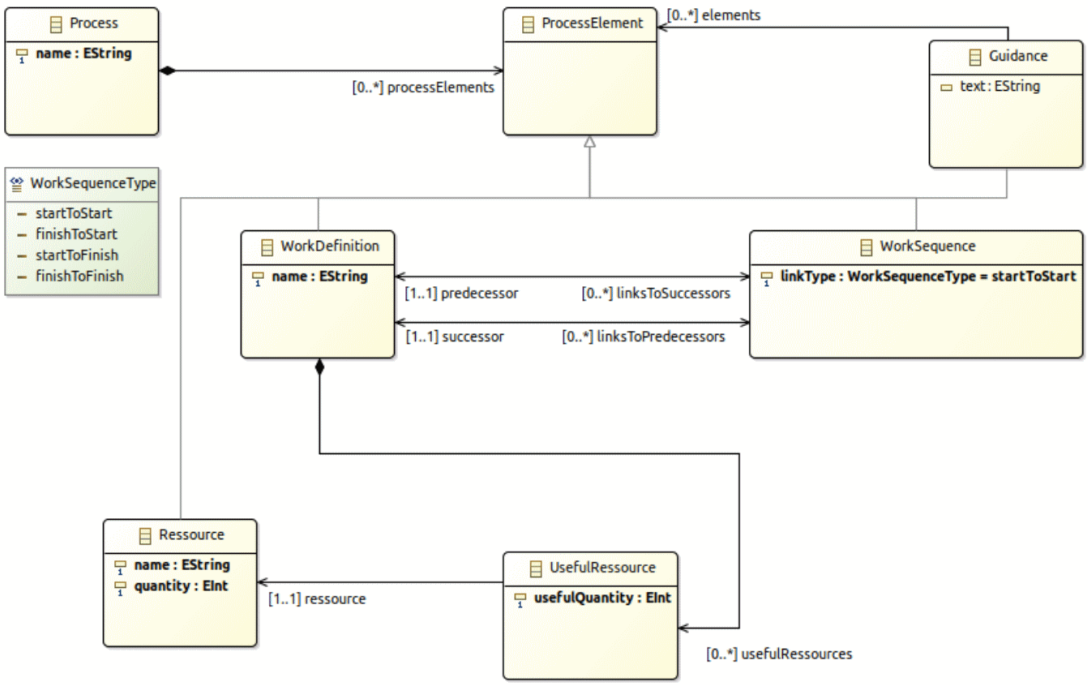

Figure 1: Modèle Ecore SimplePDL v1

Cependant nous avons remarqué qu'il était également judicieux de mettre en EOpposites cette référence des `WorkDefinition` vers les `UsefulRessource`, notamment lors de la conception des conditions OCL. Nous avons donc créé cette deuxième version finale.

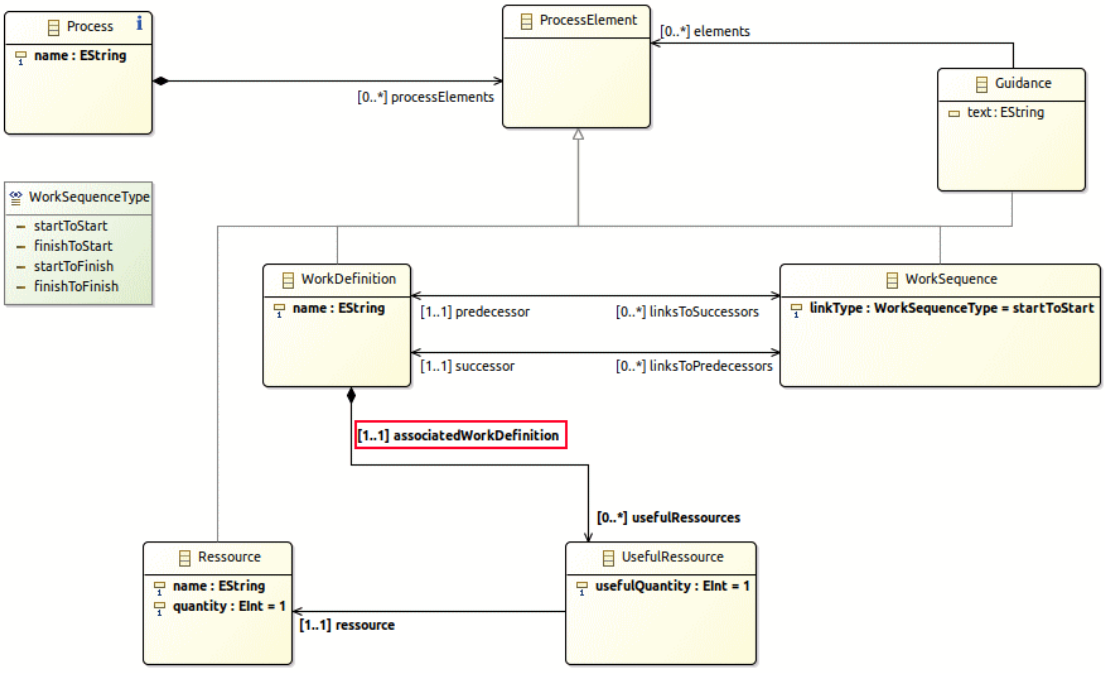

Figure 2: Modèle Ecore SimplePDL v2

2.2 Modèle Ecore PetriNet

De la même manière que simplePDL, nous avons modlisé un réseau de pétri par une classe Petri qui contiendrait tous les éléments du réseau. Tous ces éléments héritent de la classe `PetriElement`. Nous avons classé ces éléments sous deux types d'éléments : les `Arc` et les `Node` (Noeuds). Les `Node` regroupant les `Place` et les `Transition`.

Tous les noeuds ont un nom mais seulement les Places ont un entier caractérisant le nombre de jetons qui sont actuellement sur la place. Les `Arc` sont caractérisés par leurs poids, s'ils sont un readArc ou un normalArc, par leur noeud Source et leur noeud Destination. Nous avons mis ces références en EOpposites pour faciliter le déplacement. Cela a beaucoup aidé lors de la conception des propriétés LTL.

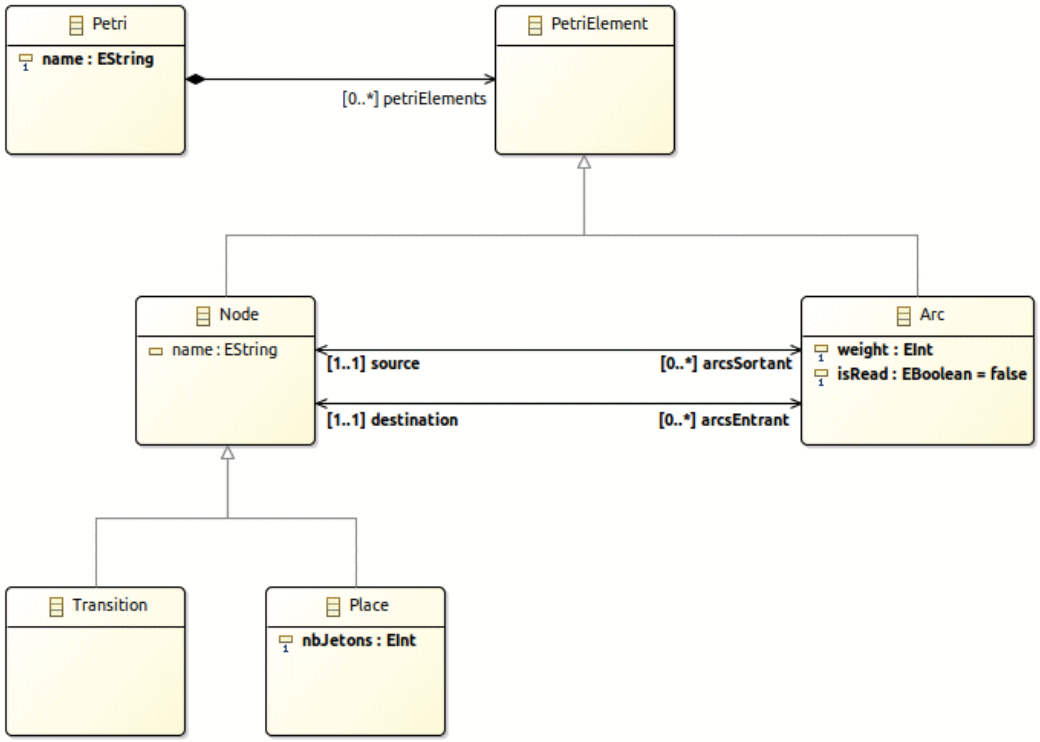

Figure 3: Modèle Ecore PetriNet v1

Nous avions décidé dans la première version de caractériser le type de l'arc par un booléen isRead, mais il semble plus judicieux de créer un type ArcType en énumérant normalArc et readArc. Cela permet plus de flexibilité dans le futur. Par ailleurs afin de pouvoir remonter au réseau de pétri nous avons également mis en EOpposites la relation de composition. Ces changements nous amène à cette deuxième version finale.

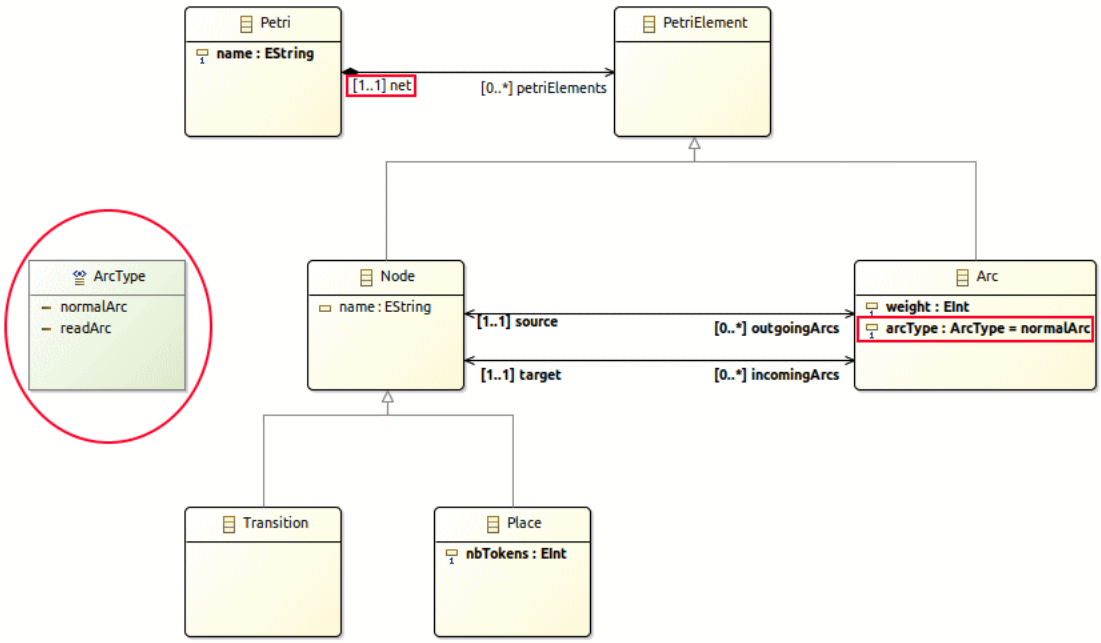

Figure 4: Modèle Ecore PetriNet v2

3 Définition de la Sémantique Statique de SimplepDL et PetriNet avec OCL (TP3)

Cette partie traite de l'élaboration des contraintes OCL afin de capturer les contraintes qui n'ont pu l'être par les métamodèles SimplePDL et PetriNet.

3.1 Contraintes OCL pour SimplePDL

Le fichier définissant les contraintes OCL se nomme : SimplePDL.ocl

Afin de montrer la pertinence de ces différentes contraintes, on a créé ces différents modèles de processus : cuisiner.xmi ; developpement.xmi (exemple sujet) ; developpementAvecRessources.xmi (exemple sujet) ; processNameProblems-ko.xmi ; processOtherProblems-ko.xmi.
Les modèles dont le nom a pour terminaison "-ko" sont des contre-exemples, mettant en évidence certaines contraintes. Tandis que ceux ne disposant pas de ce suffixe sont des exemples de modèles respectant toutes les contraintes OCL définies.

• processNameProblems-ko.xmi met en évidence les contraintes liées aux noms donnés aux éléments d'un processus par l'utilisateur. C'est-à-dire :
- les noms du processus, des activités et des ressources doivent respecter un certain format
- deux activités et deux ressources différentes d'un même processus ne peuvent pas avoir le même nom
- le nom d'une activité doit être composé d'au moins deux caractères

• processOtherProblems-ko.xmi met en évidence le reste des contraintes définies dans SimplePDL.ocl. C'est-à-dire :
- une activité ne peut pas utiliser la même ressource plusieurs fois en utilisant plusieurs liens (car autant augmenter la quantité utile que de créer plusieurs liens avec la même ressource)
- si une activité demande à utiliser une ressource alors la quantité utile de la ressource ne peut pas être négative ou nulle

- une activité ne peut pas utiliser plus de ressources que ce qu'il y a de disponible
- une dépendance ne peut pas être réflexive
- la quantité d'une ressource ne peut pas être strictement négative

Remarque : nous ne sommes pas parvenus à mettre en évidence la contrainte disant qu'un successeur et un prédecesseur doivent être dans le même processus, car nous n'avons pas réussi à créer un modèle contenant deux processus différents.

3.2 Contraintes OCL pour PetriNet

Le fichier définissant les contraintes OCL se nomme : PetriNet.ocl

Pour PetriNet, on a créé ces modèles de réseaux de Petri : quatreSaisons.xmi ; prodTampCons.xmi ; petriNameProblems-ko.xmi ; petriNumberProblems-ko.xmi ; petriOtherProblems-ko.xmi.

• petriNameProblems-ko.xmi met en évidence les contraintes liées aux noms donnés aux éléments d'un réseau de Petri :
- les noms d'un réseau de Petri, des places et des transitions doivent respecter un certain format
- deux places et deux transitions différentes d'un même réseau ne peuvent pas avoir le même nom

• petriNumberProblems-ko.xmi met en évidence les contraintes liées aux nombres dans un réseau de Petri :
- un réseau de Petri doit contenir au moins 1 jeton
- une place ne peut pas avoir un nombre de jetons strictement négatif
- un arc ne peut pas avoir un poids négatif ou nul

• petriOtherProblems-ko.xmi met en évidence le reste des contraintes définies dans PetriNet.ocl :
- plusieurs arcs de même type ne peuvent pas avoir les mêmes sources et cibles
- un arc ne peut pas être réflexif
- un arc ne peut pas partir d'une place (resp. transition) et pointer vers une place (resp. transition)
- un read arc ne peut pas partir d'une transition

Remarque : la même que précédemment mais concernant ici la source et la cible d'un arc qui doivent être dans le même réseau de Petri.

4 Transformation des modèles SimplePDL en PetriNet avec EMF/Java (TP4)

Cette partie traite la conception du fichier SimplePDLToPetriNet.Java

La première difficulté qu'on rencontre lorsqu'on commence à coder cette transformation en Java, est que tous les éléments d'un processus SimplePDL sont obtenus dans une `EList<ProcessElement>` et tous les éléments ne sont pas à priori triés.

On a donc choisi de d'abord parcourir toute la `EList<ProcessElement>` et de trier les éléments. On trie les `WorkDefinition` dans des `ArrayList<WorkDefinition>` et les `WorkSequence` dans des `ArrayList<WorkSequence>`. Pour être plus efficace on a choisi de directement traduire les `Ressource` en une Place et de les garder en mémoire dans une `HashMap<String, Place>` avec comme clé le nom de la Ressource, pour pouvoir les retrouver quand on traduira les `WorkDefinition`. On a également choisi de ne pas traduire les `Guidance` car il n'existe pas d'éléments de réseaux de pétri pour traduire une `Guidance`.

On peut ensuite traduire toutes les activités (`WorkDefinition`) en réseau de pétri en sachant qu'une activité correspond au départ au réseau de pétri de la figure 5 avec en Vert une éventuelle Place

Ressource qui a déjà été créée et en Rouge des arcs qu'on doit créé pendant cette même étape.

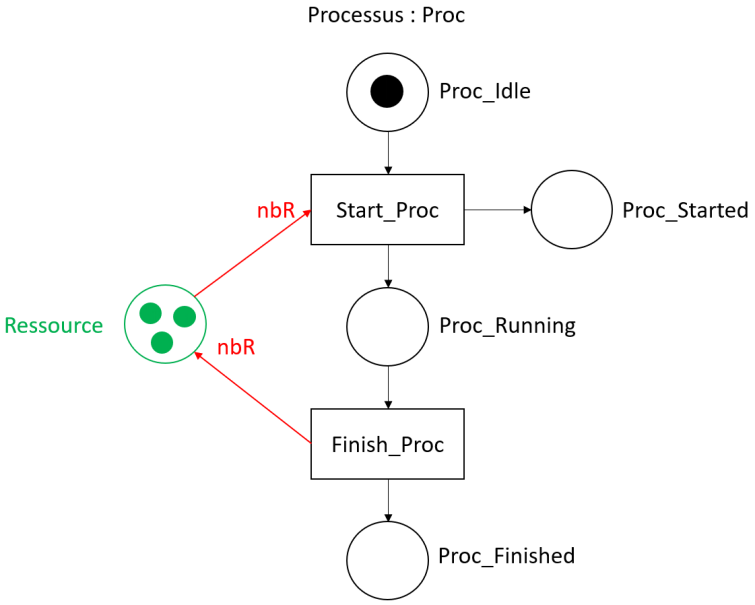

Figure 5: Réseau de Pétri d'une activité "Proc"

Pour ensuite gérer les dépendances entre les activités (`WorkSequence`) il faut pouvoir retrouver les activités correspondantes. En construisant une `HashMap<String, Node[]>` avec comme clé le nom d'une activité et en valeur un tableau trié d'une manière qu'on puisse retrouver les `Node` voulus. On peut ainsi retrouver les activités avec leurs noms et avoir accès aux Places qu'on veut.

Pour créer une dépendance `WorkSequence` entre deux activités A et B, on peut suivre le schéma suivant :

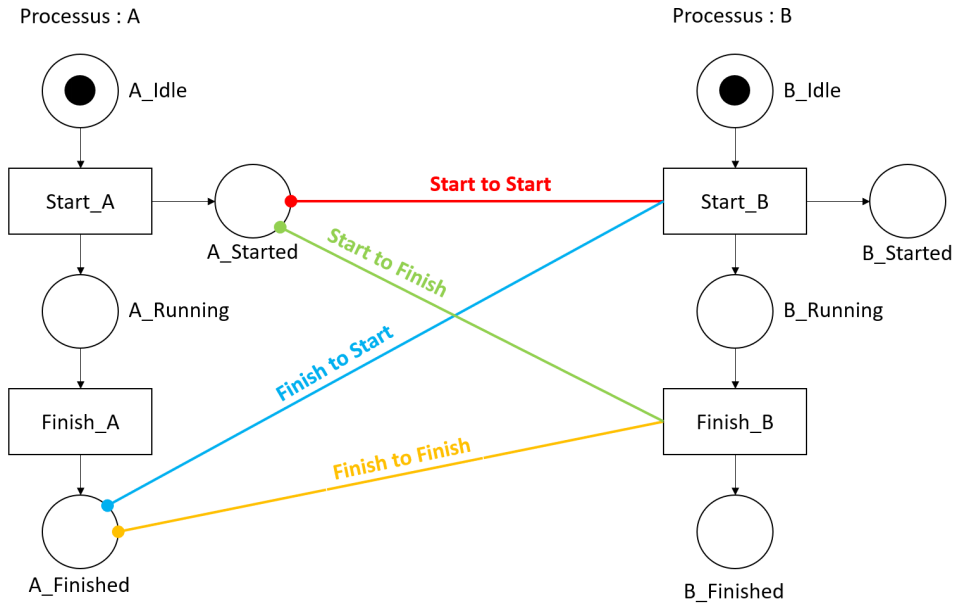

Figure 6: Schéma d'implémentation des WorkSequences

Après avoir implémenté une fonction `createWDPlaces()` qui créé une activité avec ses dépendances aux ressources, et une fonction `createWSConnection()` qui créé les dépendances (que des `readArcs`) entre les activités il suffit de les appeler dans des boucles `for`.

5 Transformation des modèles PetriNet en modèle à texte avec Acceleo (TP5)

Cette partie traite la conception du fichier totina.mtl

Ce fichier prend en entrée un modèle de type PetriNet et doit créer un fichier à texte .ltl

On peut donc pour chaque `Place` créer une place avec le bon nombre de tokens initiaux.

Ensuite pour chaque `Transition` on doit prendre les arcs entrants et les arcs sortants. Puisque le métamodèle PetriNet a été bien défini à l'aide d'EOpposites, cette tâche est facilitée. Pour chaque arc on peut regarder son type et écrire `*` ou `?` selon si c'est un `normalArc` ou un `readArc` et ensuite ajouter son poids. Selon si ce sont des arcs entrants ou sortants on peut les mettre à gauches ou à droites de la flèche `->`.

Nous avons ajouté une condition Si le poids d'un arc est de 0, alors on écrit 1, Sinon on écrit le poids de l'arc. Nous avions un problème avec les modèles PetriNet généré par ATL : lorsque un arc était de poids 1, `[arc.weight/]` renvoyait 0, bien que dans le fichier xmi la valeur affichée est 1. Nous avons donc était contraint de rajouter ce bloc if.

6 Définition d'une syntaxe graphique pour SimplePDL avec Sirius (TP6)

Cette partie traite des fichiers : simplepdl.odesign et developpementAvecRessources.simplepdl

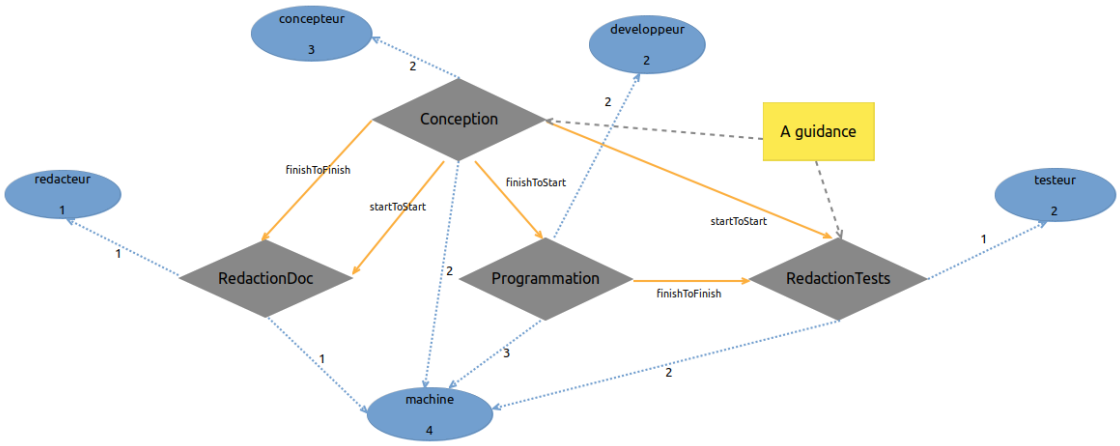

Figure 7: Graphique obtenu pour le modèle developpementAvecRessources.simplepdl

Nous sommes partis du simplepdl.odesign réalisé durant le TP6 auquel nous y avons intégré les ressources.

Au niveau de la définition de la partie graphique de l'éditeur, l'intégration des ressources n'a pas posé de réelles difficultés, mise à part le fait d'afficher dans une même ellipse le nom d'une ressource et sa quantité associée...

Au niveau de la définition de la palette, nous avons rencontré des diffcultés pour créer le RessourceEdgeCreation, c'est-à-dire à créer le lien représentant la ressource utile à l'activité entre l'élément graphique représentant la ressource et celui représentant l'activité. C'est à ce moment là que nous avons dû modifier notre métamodèle SimplePDL.ecore afin d'y intégrer la nouvelle référence opposite : associatedWorkDefinition dans la classe UsefulRessource, afin d'avoir accès à l'activité nécessitant la ressource utile.

De plus, on a créé deux calques : un calque associé aux éléments Guidance et l'autre associé aux éléments concernant les ressources. Ce qui donne les résultats suivants :

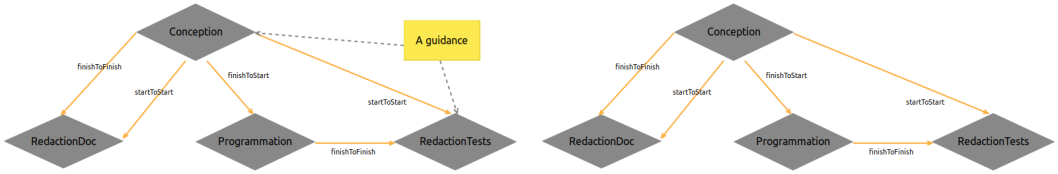

Figure 8: Graphiques obtenus sans le calque Ressource (à gauche) et sans les 2 calques (à droite)

Et enfin, on a organisé les outils de la palette en plusieurs sections comme ceci :

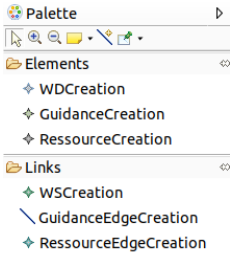

Figure 9: Palette organisée

7 Définition d'une syntaxe concrète textuelle de SimplePDL avec Xtext (TP7)

Cette partie traite des fichiers : PDL.xtext ; cuisiner.pdl ; developpementAvecRessources.pdl

Nous sommes partis du fichier PDL.xtext, généré à partir de notre métamodèle SimplePDL.ecore par Xtext. Nous avons modifié la grammaire afin de reproduire notre syntaxe souhaitée. Cette dernière se rapproche de la syntaxe de PDL1 du TP7 car on la trouve très concise.

Il ne manquait plus qu'à choisir comment représenter les éléments liés aux ressources :

```
wd Programmation {
        needs 2 of developpeur
        needs 3 of machine
}
ressource developpeur 2
ressource machine 4
```

Figure 10: Syntaxe textuelle concernant les éléments liés aux ressources

En faisant ce choix de conception pour les ressources utiles, il fallait faire attention que notre syntaxe accepte bien les activités ne nécessitant pas de ressources, c'est-à-dire les activités qui ne sont pas suivies de "{"...

Les fichiers cuisiner.pdl et developpementAvecRessources.pdl sont des exemples de processus respectant notre syntaxe textuelle.

8 Transformation des modèles SimplePDL en PetriNet avec ATL (TP8)

Cette partie concerne le fichier SimplePDL2PetriNet.atl

Au niveau théorique, la transformation est la même que pour EMF/Java (cf. Figures 5 et 6 de la Partie 4 du rapport). Mais cette fois-ci, on fait la transformation SimplePDL vers PetriNet à l'aide d'ATL. Sans traduire une Guidance en un élément du réseau de Petri.

D'abord, c'est lors de la réalisation de cette transformation que l'on s'est rendu compte de l'utilité de la reference opposite "net", pour obtenir le réseau de Petri associé à un petri element sans passer par un helper.

Au niveau des difficultés, l'utilisation de la fonction thisModule.resolveTemp s'est révélée très pratique afin d'attribuer un élément de SimplePDL à un élément de PetriNet défini précédemment dans le fichier ATL. De plus, la traduction d'une WorkSequence en un motif sur le réseau de Petri s'est révélée plutôt technique, notamment à cause de l'utilisation d'un if dans le 2ème paramètre de la fonction thisModule.resolveTemp (cf. ligne 98 du fichier SimplePDL2PetriNet.atl).

9 Production des Propriétés LTL

Cette partie traite de la conception des fichiers generateLTL.mtl, terminaisonInexistenceLTL.mtl et terminaisonAbsolueLTL.mtl

Afin d'engendrer les propriétés correspondant aux invariants de SimplePDL et permettant de vérifier la terminaison d'un processus, nous avons décider d'utiliser Acceleo car les fichiers ltl qu'on souhaite créer sont des fichiers à texte et qu'Acceleo est adaptée pour cette tâche.

9.1 generateLTL.mtl

Ce fichier génère les propriétés LTL correspondant aux invariants de SimplePDL.

9.1.1 Chaque activité est soit non commencée, soit en cours, soit terminée

On peut d'abord écrire une propriété qui dit que l'activité est toujours soit non commencée, soit en cours, soit terminée.

On peut ensuite écrire 3 propriétés pour faire des exclusions deux à deux des 3 états de l'activités. Par exemple on écrit qu'on n'a jamais en même temps l'activité non commencée et en cours.

9.1.2 Une activité terminée n'évolue plus

On peut écrire qu'une activité terminée implique qu'il y aura toujours l'activité terminée.

Puisque chaque activité est soit non commencée, soit en cours, soit terminée, on a que l'activité n'évolue plus.

9.1.3 Une activité en cours ou finit, a commencé

On peut écrire qu'une activité qui n'est pas "non commencée" implique que l'activité a commencé.

9.1.4 A StartToStart B

On peut écrire que A non "a commencé" implique que B est non commencée.

9.1.5 A FinishToStart B

On peut écrire que A n'est pas terminée implique que B est non commencée.

9.1.6 A StartToFinish B

On peut écrire que A non "a commencé" implique que B n'est pas terminée.

9.1.7 A StartToStart B

On peut écrire que A n'est pas terminé implique que B n'est pas terminée.

9.2 terminaisonInexistenceLTL.mtl

Nous voulons vérifier la terminaison d'un processus qui se traduit par la terminaison de toutes les activités. Pour cela nous avons écrit des propriétés qui vérifie la non existence d'une terminaison, de cette façon selt nous renvoie un exemple d'une terminaison si cela est possible.

Pour cela on doit écrire qu'on a toujours au moins une activité non terminée. Par exemple si la listes des activités du processus est A, B et C, alors on écrit qu'on a toujours A non terminée ou B non terminée ou C non terminée.

9.3 terminaisonAbsolueLTL.mtl

Nous voulions également vérifier si quelque soit l'ordre dans lequel on actionne les transitions, il était toujours possible de terminer le processus. Sion n'a pas de terminaison absolue alors selt nous renverra une situation d'interblocage.

Nous avons donc écrit qu'éventuellement toutes les activités sont terminées.

10 Conclusion

Ce mini-projet avait pour but de mettre en place une chaîne de vérification de modèles de processus SimplePDL dans l'optique de vérifier leur cohérence, notamment afin de savoir si un processus peut finir ou non. Pour répondre à cela, nous avons découvert et manipulé une multitude d'outils, à travers Eclipse.

De plus, ce mini-projet nous a permis de découvrir GitHub car on a décidé de travailler avec un workspace partagé et cela s'est révélé très utile !