Ingénierie Dirigée par les Modèles

Compte-Rendu de Mini-Projet

Chaîne de vérification de modèles de processus

Antoine Girard et Priscilia Gonthier
Groupe L2

Département Sciences du Numérique
2022-2023

Table des matières

Table des figures

Introduction

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.

Liste des fichiers utilisés dans le projet

— Métamodèle SimplePDL : SimplePDL.ecore

— Image du métamodèle SimplePDL : SimplePDL.png

— Métamodèle PetriNet : PetriNet.ecore

— Image du métamodèle PetriNet : PetriNet.png

— Contraintes OCL associées à SimplePDL : SimplePDL.ocl

— Exemples associées aux contraintes de SimplePDL : developpement.xmi, simplepdl_ko1.xmi et simplepdl_ko2.xmi

— Contraintes OCL associées à PetriNet : PetriNet.ocl

— Exemples associées aux contraintes de PetriNet : developpement_petrinet.xmi et petrinet_ko.xmi

— Code Java de la transformation SimplePDL en PetriNet : SimplePDL2petrinet.java

— Code ATL de la transformation SimplePDL en PetriNet : SimplePDL2PetriNet.atl

— Code Acceleo des transformations modèle à texte : toTina.mtl et toltl.mtl

— Modèle Sirius décrivant l'éditeur graphique pour simplePDL : simplepdl.odesign

— Modèle Xtext décrivant la syntaxe textuelle de SimplePDL : PDL.xtext

— Exemple de modèle de processus : developpement.xmi

1 Définition des métamodèles

1.1 SimplePDL

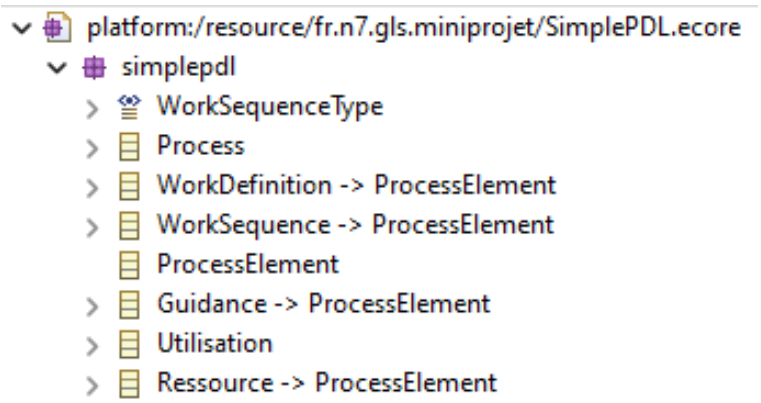

FIGURE 1 – Métamodèle de simplePDL

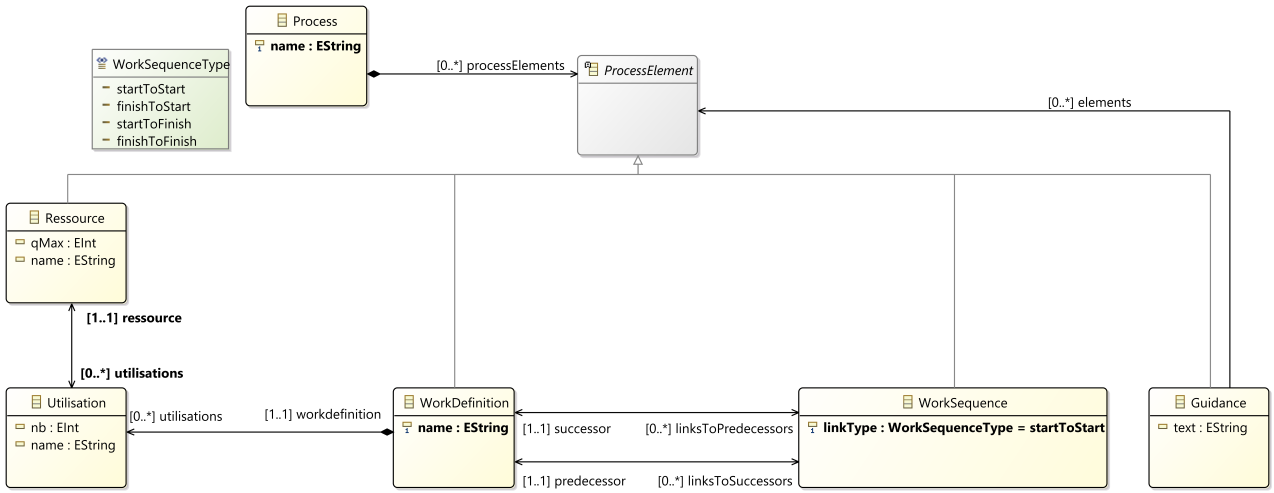

FIGURE 2 – Métamodèle de simplePDL

Le métamodèle de SimplePDL est stocké dans le fichier SimplePDL.ecore. Nous sommes partis d'un métamodèle de simplePDL basique qui comportait des Process et des ProcessElements qui sont soit des WorkDefinitions, soit des WorkSequences, soit des Guidances. La première étape a été d'ajouter des ressources qui sont nécessaires aux activités (WorkDefinitions) de notre métamodèle.

Pour cela nous avons créé une classe Ressource qui hérite de ProcessElement car la ressource est un élément du Process. Cette classe a comme attribut un nom et la quantité maximale de cette ressource.

Afin de relier ces ressources aux WorkDefinitions, nous avons créé une classe Utilisation. Il y a une relation de composition avec la WorkDefinition, car l'utilisation est unique à chaque Workdefinition et une relation d'association avec les ressources car une ressource peut être utilisée par plusieurs WorkDefinitions. La classe Utilisation a en attribut un nom et le nombre de ressource utilisé.

1.2 PetriNet

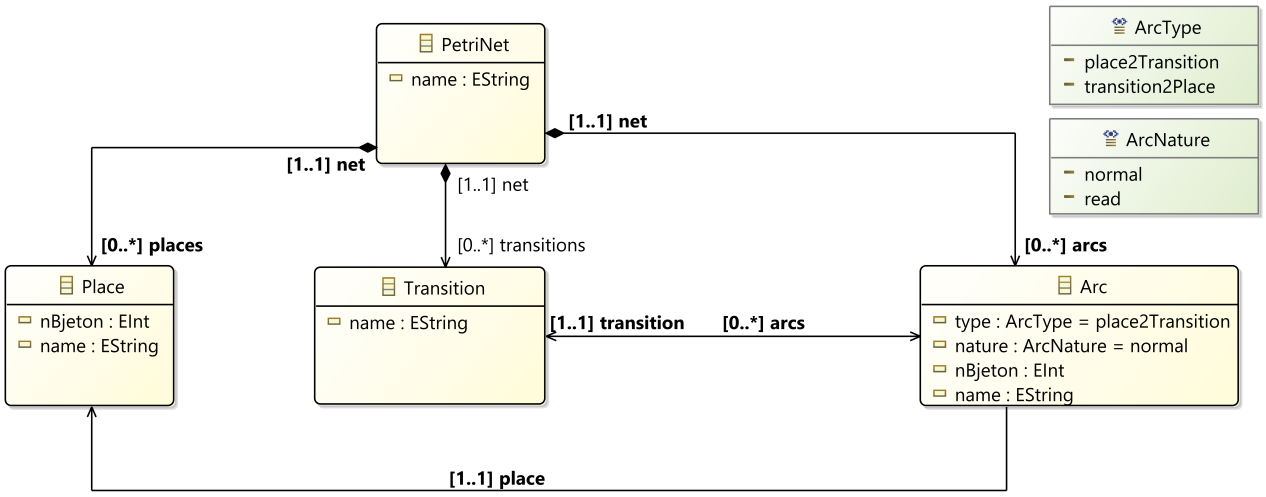

FIGURE 3 – Métamodèle de petriNet

Le métamodèle de PetriNet est stocké dans le fichier PetriNet.ecore. Le métamodèle du PetriNet est composé d'une classe PetriNet qui est définie par des places, des transitions et des arcs pour relier les places et les transitions entre elles. L'objectif premier était de définir un réseau de pétri basique pour pouvoir dans un second temps ajouter la notion de nombre de jetons nécessaires pour chaque transition.

Pour cela, nous avons trois classes qui représentent : les places, les transitions et les arcs. Ce sont toutes les trois des compositions de la classe PetriNet.

Nous avons dû définir aussi 2 énumérations pour pouvoir différencier les types d'arc ("place to transition" ou "transition to place") et la nature des arcs (arc "normal" avec consommation de jeton et arc "read" pour un arc qui ne consomme pas de jeton).

Nous devions ensuite définir la notion d'utilisation de jetons pour notre réseau. Nous avons donc ajouté un attribut nBjeton aux classes Place et Arc. Pour la classe Place cet attribut correspond au nombre de jetons disponibles à cette place et pour l'arc il s'agit du nombre de jetons minimum nécessaire qu'il est censé y avoir sur la place source pour que ce soit possible d'avancer dans le réseau.

2 Définition de la sémantique statique associée aux modèles

2.1 Introduction

La sémantique statique de nos métamodèles est définie grâce aux fichiers SimplePDL.ocl et PetriNet.ocl. Dans ces derniers, nous avons la possibilité de définir des règles sur nos métamodèles qui vont permettre leur bon fonctionnement.

2.2 SimplePDL (cf : SimplePDL.ocl)

Pour la sémantique statique associée au modèle SimplePDL, nous avons besoin de définir les règles suivantes :

1. Toutes les activités doivent être dans le même Process.
2. Les noms :
   (a) Les noms des WorkDefinition, Ressource et Utilisation commencent par une lettre (majuscule ou minuscule) suivie de lettres (majuscules ou minuscules) ou de chiffres.
   (b) Deux WorkDefinitions différentes ne peuvent pas avoir le même nom.
   (c) Deux Ressources différentes ne peuvent pas avoir le même nom.
3. Une activité ne peut pas être reflexive
4. Les ressources :
   (a) Il faut qu'il y ait assez de ressources pour lancer une activité.
   (b) Il faut que le nombre de ressources soit un entier positif ou nul.

Les contraintes peuvent être testés sur developpement.xmi qui est un modèle correct, sur simplepdl_ko1.xmi où les contraintes de nom de WorkDefinition et de réflexivité ne sont pas respectées, ainsi que sur simplepdl_ko2.xmi où les contraintes de nom de Ressource, de positivité sur le nombre de jeton et d'utilisation des ressources ne sont pas respectées.

2.3 PetriNet (cf : PetriNet.ocl)

Pour la sémantique statique associée au modèle PetriNet, nous avons besoin de définir les règles suivantes :

1. Les noms :
   (a) Les noms des PetriNet, Transition et Place commencent par une lettre (majuscule ou minuscule) suivie de lettres (majuscules ou minuscules) ou de chiffres.
   (b) Deux places différentes ne peuvent pas avoir le même nom.
   (c) Deux transitions différentes ne peuvent pas avoir le même nom.
2. Les ressources :

(a) Les Places doivent avoir un nombre de jetons positif ou nul.

(b) Les Arcs doivent avoir un nombre de jetons strictement positif.

Les contraintes peuvent être testés sur developpement_petrinet.xmi qui est un modèle correct ainsi que sur petrinet_ko.xmi où les contraintes de noms et de positivité sur le nombre de jetons ne sont pas respectées.

3 Les transformations modèle à texte

3.1 PetriNet to Tina via Acceleo (cf : toTina.mtl)

L'objectif de cette partie est de transformer un modèle PetriNet en texte en syntaxe Tina à l'aide de l'outil Acceleo.

Notre fichier nous permet de prendre les éléments un à un de notre modèle de pétri et les transforme en texte en se servant de leurs paramètres.

Par exemple, la ligne : pl [p.name/] ([p.nBjeton/]) permet de transformer une place en une ligne de texte qui contient son nom et le nombre de jetons qu'elle possède.

3.2 PetriNet to LTL

L'objectif de cette partie est d'ajouter des contraintes sur le réseau de pétri pour s'assurer qu'il est bien construit et fonctionnel.

Dans le fichier toltl.mtl, on définit principalement deux choses :
- Les invariants
- Les conditions de terminaison

Parmis les invariants, on retrouve les conditions suivantes :

1. Il ne peut y avoir plus d'un jeton dans l'ensemble des places {p_ready, p_started} d'une même WorkDefinition.
2. Il ne peut y avoir plus d'un jeton dans l'ensemble des places {p_ready, p_running, p_finished} d'une même WorkDefinition.
3. Si une WD est finie, alors elle a commencé (p_finished => p_started)
4. Si une WS est de type start_2_start, il faut que le prédécesseur ait commencé avant le successeur
5. Même chose pour les autres types de WS

Pour ce qui est des conditions de terminaison, on peut définir la suivante :
Il faut que toutes les WD soient finies. Dans ce cas, on arrive dans un état terminal.

Si on est dans un état terminal, il faut forcement que toutes les WD soient finies.

Il y a forcément un état terminal à la fin d'un réseau de pétri correct. Si ce n'est pas le cas, alors le réseau de pétri n'est pas fonctionnel. Dans le fichier ltl, la propriété "- <> dead" qui signifie qu'il n'y aura pas d'état terminal est donc toujours fausse et nous renvoie un contre-exemple, où il y a une terminaison.

4 Les transformations modèle à modèle

4.1 SimplePDL to PetriNet via EMF/Java

4.1.1 Introduction

L'objectif de cette partie était de réaliser le modèle PetriNet à partir du modèle SimplePDL en utilisant une transformation via EMF/Java. Cette transformation est définie dans le fichier SimplePDL2petrinet.java

4.1.2 Explications

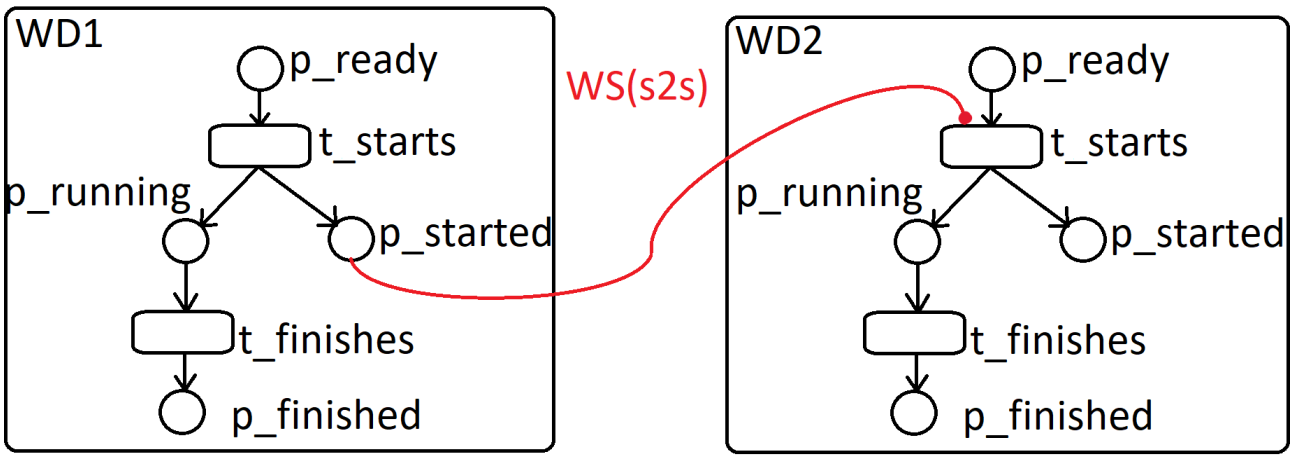

FIGURE 4 – WD en petriNet

Pour réaliser cela nous avons dû convertir une à une les WorkDefinitions (WD), WorkSequences (WS) et Ressources en Places, Arcs et Transitions.

1. Comme montré dans le schéma ci-dessus, une WD est composé de 4 places (p_ready, p_started, p_running et p_finished) et de 2 transitions (t_starts et t_finishes). Pour ceci nous procédons de la manière suivante :

```
// Places d'une WorkDefinition
Place p_ready = myFactory.createPlace();
p_ready.setName(wd.getName() + "_ready");
p_ready.setNBjeton(1);
p_ready.setNet(petriNet);
petriNet.getPlaces().add(p_ready);
```

FIGURE 5 – Création d'une place via EMF

On peut voir qu'on définit les éléments qui composent une place un à un en les liant au paramètre de la WD (nom, nbJeton)
On procède ensuite de la même manière pour la création des autres places, arcs et transitions.

2. Les WS sont définies en fonction de leur type (START_2_START, START_2_FINISH, FINISH_2_START et FINISH_2_FINISH), on a donc fait des cas et en fonction de ces derniers, on définit la source et la destination de l'arc représentant la WS.
3. Enfin, pour les ressources, nous avons créé une place Ressource dans laquelle sont stockées les ressources disponibles. Nous avons aussi créé deux arcs par WD (si besoin) représentant l'utilisation des ressources :
   (a) Provenant de la place Ressource et à destination de la transition t_starts (consommer des ressources pour commencer l'activité).
   (b) Provenant de la transition t_finishes et à destination de la place Ressource (libérer les ressources pour finir l'activité).

4.2 SimplePDL to PetriNet via ATL

4.2.1 Introduction

L'objectif de cette partie était de réaliser le modèle PetriNet à partir du modèle SimplePDL en utilisant une transformation via ATL. Cette transformation est définie dans le fichier SimplePDL2PetriNet.atl

4.2.2 Explications

Il s'agit de la même manière de procéder que pour la transformation via EMF/Java. C'est-à-dire traduction des WD, WS et Ressources une à une.

La seule différence notable provient du fait que la définition des utilisations est réalisée dans une règle différente de celle des Ressources.

5 Définition d'une syntaxe concrète textuelle

5.1 Xtext

Le principe de Xtext est de décrire la syntaxe d'un langage qui sera reconnu dans un éclipse de développement. Dans ce dernier, on va pouvoir y définir des fonctions pour nous permettre de générer un modèle avec des WD, WS, Ressources et Guidance beaucoup plus facilement.

6 Définition d'une syntaxe concrète graphique

6.1 Sirius

L'objectif de cette partie est de créer un greffon qui nous permettra de construire et visualiser par un graphique clair un modèle ecore.

Pour cela nous avons défini une syntaxe graphique en y ajoutant les WD, WS, Ressources et Guidances.

Ainsi, dans notre eclipse de développement, nous pouvons à présent créer un process et y ajouter des ProcessElements (WD, WS, Ressources et Guidance)

Nous avons défini le graphique engendré par notre greffon sur deux calques. Ceci nous permet de cacher à volonté les Guidances pour rendre le graphique moins chargé.

7 Conclusion

En conclusion, nous avons grâce à ce projet pu comprendre et utiliser les outils d'Eclipse pour les modèles de réseau de Pétri.

Nous avons réussi à implanter les transformations modèle à modèle et modèle à texte et nous avons réussi à définir nos syntaxes concrètes textuelles et graphiques.

La plupart des difficultés que l'on a rencontrées proviennent de l'utilisation d'eclipse et de tous les packages dont on s'est servi.