Mini-Projet d'Ingénierie Dirigée par les Modèles : Chaîne de vérification de modèles de processus

Youssef EL ALAMI
Tom AUDARD
Thomas BOCANDE

Département Sciences du Numérique - Deuxième année
2023-2024

Table des matières

Table des figures

1 Introduction

L'objectif principal de ce projet est de développer une chaîne de vérification de modèles solide pour garantir la qualité des processus SimplePDL, en vérifiant leur cohérence, en particulier leur capacité à se terminer. Pour ce faire, nous avons utilisé des outils de model-checking basés sur les réseaux de Petri, intégrés dans la boîte à outils Tina.
Le projet a été réalisé en plusieurs étapes : définition des métamodèles et de la sémantique statique, manipulation des modèles à l'aide de l'infrastructure EMF (Eclipse Modeling Framework), création des transformations modèle-à-texte et modèle-à-modèle, et enfin définition des syntaxes textuelles et graphiques concrètes.

2 Définition des métamodèles avec Ecore

Nous avons tout d'abord commencé par compléter le métamodèle SimplePDL en y ajoutant 2 classes pour prendre en compte les ressources (voir figure 1 ou fichier **SimplePDL.ecore**) : Ressource et RessourceUsed. Ainsi la classe Ressource est un sous-type de ProcessElement. Elle possède en attribut son nom et sa quantité. La classe RessourceUsed quant-à elle fait le lien entre la classe Ressource et la classe WorkDefinition en possédant un attribut d'occurrence.

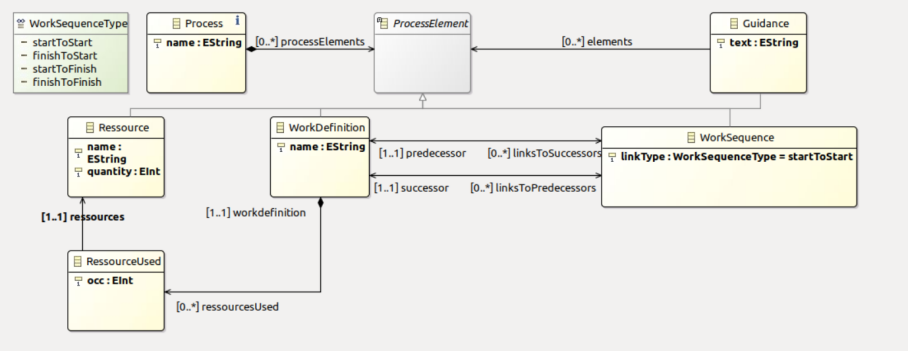

FIGURE 1 – Diagramme de classe de SimplePDL

Nous avons ensuite défini le métamodèle PetriNet (voir figure 2 ou fichier **PetriNet.ecore**). Il modélise un réseau de pétri à partir de ses noeuds (places et transitions) et de ces arcs. Les arcs peuvent être de type normal ou read (ArcKind) et un arc peut aller d'une place à une transition ou d'une transition à une place (placeToTransition ou transitionToPlace de ArcDirection).

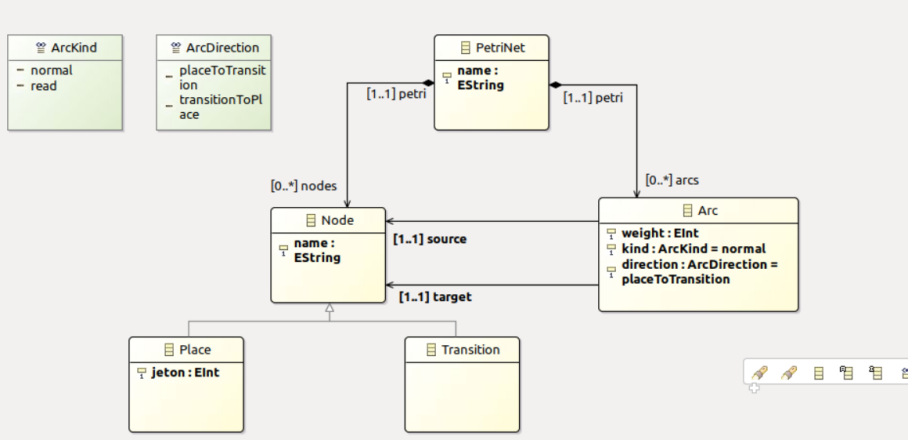

FIGURE 2 – Diagramme de classe de PetriNet

3 Définition de la sémantique statique avec OCL

Ensuite, nous avons défini des contraintes OCL sur ces deux métamodèles afin de capturer les contraintes qui n'ont pas pu l'être.
Pour le métamodèle Simple PDL, nous avons défini les contraintes suivantes (fichier **SimplePDL.ocl**) :

- Pour les **Processus** : Leurs noms ne peuvent pas avoir moins de deux caractères et ne peuvent pas commencer par un chiffre.
- Pour les **WorkDefinitions** : Leurs noms doivent être composés d'au moins un caractère et commencent par une lettre. Deux WorkDefinitions ne peuvent pas avoir le même nom.
- Pour les **WorkSequences** : Elles ne peuvent pas être reflexive.
- Pour les **Ressources** : Leurs quantités ne peuvent pas être négatives ou nulles, leurs noms doivent être composés d'au moins deux caractères et commencent par une lettre. En ce qui concerne les ressources utilisés par les WorkDefinitions, leurs occurrences positives respectent la borne maximale qui est la quantité de la ressource correspondante.

Puis nous avons créé un exemple de modèle s'appuyant sur SimplePDL respectant les contraintes (voir fichier **process_exemple.xmi**) et un contre-exemple (voir fichier **process1-ko.xmi**).

Pour PetriNet nous avons défini les contraintes suivantes (voir fichier **PetriNet.ocl**) :

- Pour le **PetriNet** : Doit avoir un nom.
- Pour les **Places** et les **Transitions** : Leurs noms ne peuvent pas commencer par un chiffre et ne peuvent pas avoir moins de deux caractères.
- Pour les **Arcs** : Ne doivent pas reliés deux places ni deux transitions. Leurs poids ne doivent pas être nuls.

Puis nous avons créé un exemple de modèle s'appuyant sur PetriNet qui respecte les contraintes (voir fichiers **petrinet_ok.xmi** et **petrinet_ok_tp5.xmi**) ainsi qu'un contre-exemple (voir fichier **petrinet_ko.xmi**).

4 Manipulation des modèles avec EMF

Nous avons défini une transformation SimplePDL vers PetriNet en utilisant EMF/Java (voir fichier **SimplePDL2PetriNet.java**). Pour réaliser cela, nous avons défini une méthode de classe pour chaque classe dans SimplePDL. On trouve ainsi les méthodes suivantes :

— Créer les quatre places, deux transitions et cinq arcs nécessaires pour représenter une WorkDefinition.
— Créer un read-arc pour représenter une WorkSequence entre deux WorkDefinition suivant son type. Nous avons d'ailleurs utiliser un attribut de classe de type Map pour retrouver les places qu'il fallait relier.
— Créer une place pour représenter une Ressource avec un nombre de jeton qui vaut la quantité de la ressource.
— Créer deux arcs, un depuis la ressource jusqu'à la transition start de la WorkDefinition et un depuis la transition finish de la WorkDefinition jusqu'à la ressource, afin de représenter les RessourceUsed. Ces deux arcs ont un poids qui vaut l'occurrence de la ressource utilisée.

Ainsi le programme Java lit le fichier correspondant au SimplePDL et crée le fichier équivalent qui respecte PetriNet.

5 Définition de Transformations Modèle à Texte avec Acceleo

Pour cette partie, nous avons défini une transformation d'un modèle respectant le métamodèle PetriNet (un réseau de pétri) vers le format texte compris par l'outil Tina (.net) (fichier **toTina.mtl**), comme on peut le voir sur la figure 3. Nous avons aussi défini un transformation d'un modèle respectant SimplePDL vers un fichier LTL comprenant les propriétés suivantes (fichier **toLTL.mtl**) :

— Chaque activité est soit non commencée, soit en cours, soit terminée.
— Le processus peut se finir.
— Une activité finie n'évolue plus.

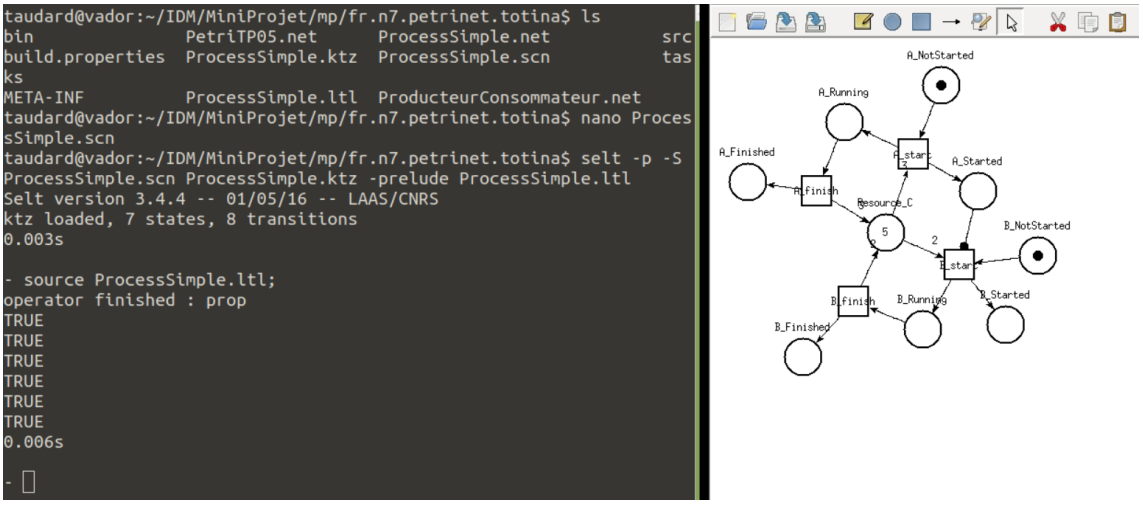

FIGURE 3 – Vérification des propriétés LTL et visualisation du réseau de pétri grâce à Tina

6 Définition de Transformations Modèle à Modèle avec ATL

De la même manière qu'avec Java, nous avons défini une transformation d'un modèle conforme à SimplePDL vers un modèle équivalent conforme à PetriNet (**simplepdl2petrinet.atl**). Pour cela, nous avons défini une règle ATL pour chaque classe de SimplePDL :

— La règle WD2PetriNet : Génère un réseau de pétri pour chaque WorkDefinition (les 4 places, 2 transitions et 5 arcs).
— La règle WS2ReadArc : Crée les read arc qui vont relier les réseaux de pétri des WorkDefinitions liées par les WorkSequences.
— La règle Ressource2Place : Crée la place qui contient autant de jetons que la quantité de la ressource.
— La règle RessourceUsed2Arc : Crée les deux arcs pour chaque ressource utilisée par une WorkDefinition. Un arc de la ressource jusqu'à la transition start et un de la transition finish à la ressource (pour "rendre" la ressource une fois le processus terminé). Le poids de ces arcs est l'occurrence de la ressource utilisée.

7 Définition de Syntaxes Concrètes Textuelles avec Xtext

Pour la définition des syntaxes concrètes textuelles, nous avons utilisé l'outil Xtext. Donc pour le métamodèle Simple PDL, nous avons repris la syntaxe PDL1 fournie dans le TP8 et l'avons ajustée pour prendre en compte les ressources (voir fichier **PDL1.xtext**) :

— Pour définir une **Ressource** : il faut écrire le mot clé 'ressource', suivi du nom de la ressource, le mot 'quantity' et la quantité des ressources.

— Pour définir les **Ressources Utilisées** par une activité : nous avons modifiés la définition du WorkDefinition, en ajoutant, après le nom de l'activité, le mot clé 'uses' suivi de la définition d'une ou plusieures ressources utilisées en précisant leurs occurences et leurs noms. Cette definition est optionelle, ce qui signifie qu'une WorkDefinition peut, ou non, utiliser des ressources.

Pour vérifier le bon fonctionnement de l'outil Xtext, nous avons élaboré l'exemple dans le fichier **processExample.pdl1**.

8 Définition de Syntaxes Concrètes Graphiques avec Sirius

Pour prendre en compte les ressources, nous avons rajouté un noeud pour la classe Ressource et un lien pour la classe RessourceUsed avec les outils de création correspondants dans la palette (voir fichier **simplepdl.odesign**). On a alors décidé pour la ressource utilisée que la flèche irait de la WorkDefinition vers la ressource pour dire "wd utilise n ressources". On obtient alors le résultat de la figure 4 (fichier **developpement.simplepdl**).

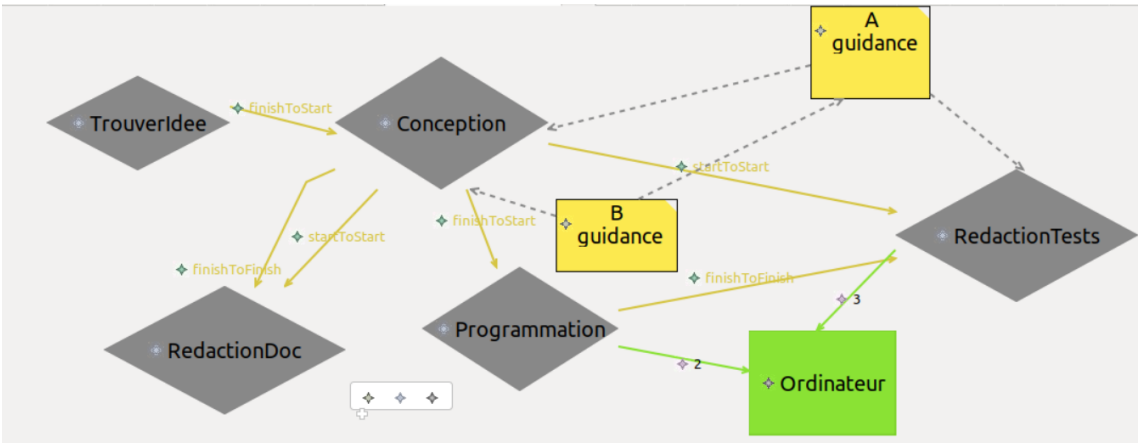

FIGURE 4 – Visualisation du processus grâce à Sirius

9 Conclusion

Pour conclure, nous avons définit des métamodèles de SimplePDL et PetriNet. On a ensuite définit des contraintes OCL sur ces deux métamodèles. Nous avons utilisé Java pour définir une transformation d'un modèle conforme à SimplePDL vers un modèle équivalent conforme à PetriNet. Nous avons défini une transformation d'un modèle conforme à PetriNet vers un fichier texte au format .net pour le visualiser sur Tina avec Acceleo. De la même manière qu'avec Java, nous avons défini la transformation modèle à modèle avec ATL. Enfin nous avons défini une syntaxe concrète textuelle avec Xtext et une syntaxe concrète graphique avec Sirius. Nous avons pris en compte les ressources dans les transformations.