Rapport de Projet : Chaîne de vérification de modèles de processus

Justin Jourdant, Issam Alouane

20 octobre 2022

Contents

1 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.

Le modèle de procédé utilisé est inspiré de SPEM, norme de l'OMG. norme. Dans un premier temps, nous nous intéressons à des processus simples composés seulement d'activités (WorkDefinition) et de dépendances (WorkSequence). Les activités seront représentées par des ellipses. Les arcs entre activités représentent les dépendances. Une étiquette permet de préciser la nature de la dépendance sous la forme " étatToAction " qui précise l'état qui doit être atteint par l'activité source pour réaliser l'action sur l'activité cible.

Pour réaliser une activité, des ressources peuvent être nécessaires. Une ressource peut correspondre à un acteur humain, un outil ou tout autre élément jouant un rôle dans le déroulement de l'activité. Ici, nous nous intéressons simplement aux types de ressources nécessaires et au nombre d'occurrences d'un type de ressource. Par exemple, il peut y avoir deux développeurs, trois machines, un bloc-note, etc. Un type de ressource sera seulement caractérisé par son nomet la quantité d'occurrences de celle-ci.

Pour pouvoir être réalisée, une activité peut nécessiter plusieurs ressources (éventuellement aucune). Les occurrences de ressources nécessaires à la réalisation d'une activité sont prises au moment de son démarrage et rendues à la fin de son exécution. Bien entendu, une même occurrence de ressource ne peut pas être utilisée simultanément par plusieurs activités. Les types de ressource et leur nombre d'occurrences sont définis en même temps que le procédé lui-même.

2 Définition des métamodèles avec Ecore

Deux métamodèles ont été définis avec Ecore le premier pour SimplePDL et le deuxième pour PetriNet.
Concernant le métamodèle SimplePDL (voir fichier SimplePDL.ecore) nous avons repris le modèle proposé en TP et l'avons agrémenté de ressources. Pour faire ceci, nous avons créé 2 classes : la classe Ressources qui modélise les ressources (leur nom et leur quantité) et la classe BesoinRessources qui fait la liaison entre la WorkDefinition et la ressource en indiquant la quantité de ressource que la WorkDefinition a besoin pour commencer

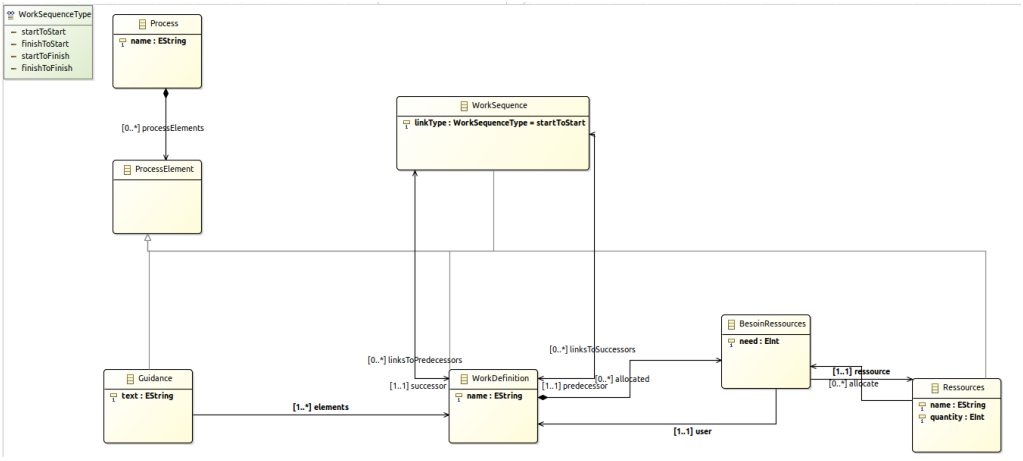

Figure 1: Le Métamodèle simplePDL sous Eclipse

Concernant le métamodèle PetriNet (voir fichier Petri.ecore) nous nous sommes inspiré du métamodèle SimplePDL pour la structure arborescente.

On peut voir sur cette description qu'un modèle PetriNet est tout d'abord un Réseau constitué de PetriElements. Ces PetriElements peuvent aussi bien être des Places que des Transitions. Les arcs ont un comportement assez différéents de ces deux éléments et ont été traités à part (on a aussi introduit une énumération qui indique si l'arc est un arc normal ou un readArc). Le marquage initial des Places a été retiré au cours du travail de modélisation car il s'est vite révélé peu utile : il est inclus lorsque l'on entre le nombre de jetons.

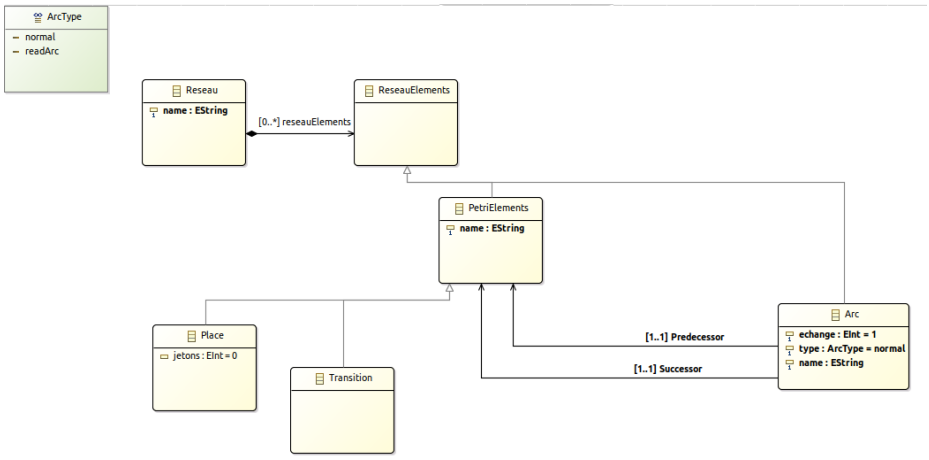

Figure 2: Le Métamodèle PetriNet sous Eclipse

3 Définition de la sémantique statique avec OCL

Les métamodèles précédents suivent un ensemble de règles que l'on a défini au travers de contraintes OCL. Certaines contraintes de bonne définitions des noms, de non-redondances d'éléments etc. ont été définis pour les deux métamodèles. Des règles plus spécifiques ont été définis par la suite :
- Pour SimplePDL (dans le fichier ocl correspondant) : La quantité de ressources est toujours positive ou nulle.
- Pour PetriNet (dans le fichier ocl correspondant) : Une place ne peut pas avoir un nombre de jeton négatif, le nombre de jetons échangés sur un arc ne peut pas être négatif ou nul, un arc ne peut relier deux places ou deux transitions.

4 Définition de transformations modèle à texte (M2T) avec Acceleo

L'objectif de la transformation modèle à texte avec Acceleo était de passer de la syntaxe du métamodèle PetriNet à une syntaxe textuelle utilisée par les outils Tina. Grâce au fichier toTina.mtl, cette transformation est possible en utilisant les outils proposés par Acceleo sous forme de Template et de requête dans un fichier.atl .
Pour ce faire, nous avons d'abord transformer toutes les Places en effectuant une boucle sur les places du modèle de Petri sur lesquels nous avons appliqué une template pour adapter la syntaxe à Tina. Nous avons ensuite fait de même avec les Transitions pour grâce à une boucle, convertir toutes les syntaxes en Tina. Enfin il a fallu tenir compte des arcs et les adapter pour faire apparaitre leur impact à la suite des transitions comme spécifié dans la syntaxe Tina.

5 Définition d'une transformation de modèle à modèle (M2M) avec EMF/Java et avec ATL

L'infrastucture fournie par EMF nous a ensuite permis de manipuler ces modèles. Nous avons réaliser un module permettant de passer de SimplePDL à PetriNet en utlisant la syntaxe propre à Java (voir fichier SimplePDLToPetri.java). Grâce à ce module et celui réalisé sous Acceleo entre PetriNet et Tina nous pouvons convertir un modèle SimplePDL en un modèle sous Tina automatiquement et rapidement.
Avec l'outil ATL nous avons pu réaliser la même opération qu'avec EMF pour passer de SimplePDL à PetriNet (voir fichier SimplePDL2PetriNet.atl). Cette troisième méthode se base quand à elle sur l'utilisation de règle qui seront appliquées à tous les éléments d'un modèle SimplePDL pour les convertir en élément d'un modèle PetriNet.
Ainsi, nous avons basé nos transformations sur la transformation élémentaire du figure 3 qui illustre une WorkDefinition A dans un petriNet, et donc pour les WorkSequence il suffit d'ajouter un arc entre les 2 présentations dépendant de la relation. Si on suppose par exemple qu'une WorkSequence de type startToStart lie deux WorkDefinition A et B alors elle sera traduit par un arc de A_started vers B_start, pour startToFinish ça sera de A_started vers B_finish, pour finishToStart ça sera de A_finished vers B_start, et enfin, pour finishToFinish ça sera de A_finished vers B_finish. Ensuite, pour les ressources, nous avons transformé une ressource en une place du même nom que la ressource et de variable jetons égale à la quantité de ressource. Enfin, pour BesoinRessources, nous avons transformé une BesoinRessources d'une WorkDefinition A et une ressource r en un arc entre la place r et A_start

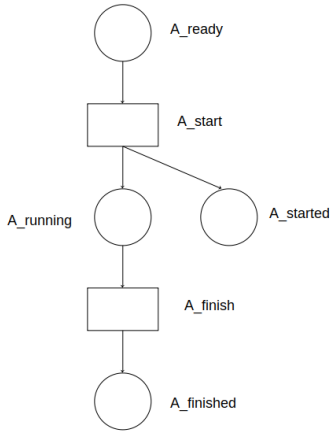

Figure 3: WorkDefinition A en PetriNet

Les 2 figures suivantes montrent l'exemple d'une transformation simplePDL vers PetriNet:

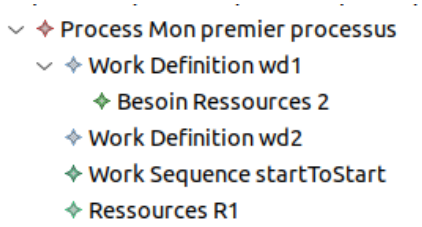

Figure 4: Modèle original en simplePDL

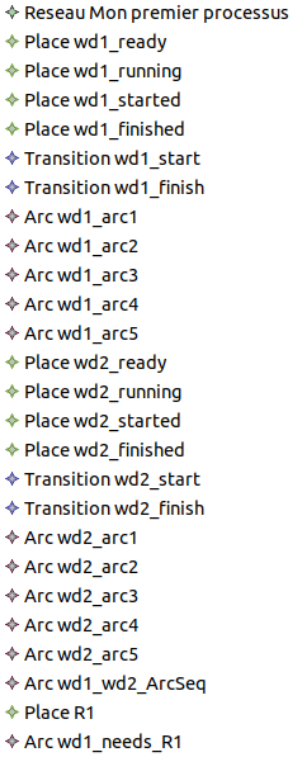

Figure 5: Modèle transformé en PetriNet

6 Définition de syntaxes concrètes textuelles avec Xtext

La syntaxe concrète textuelle définie avec Xtext permet par exemple d'engendrer des métamodèles à partir d'une syntaxe définie dans un fichier Xtext. La grammaire définie est très importante car la manière syntaxique de décrire un modèle SimplePDL engendrera différents métamodèles avec des structures parfois très différentes.
Dans notre cas (voir fichier PDL.xtext), nous sommes basés sur le pdl1 fait au tp7, et avons ajouté dans la règle Guidance le mot-clé "for" et après "for" se trouve les WorkDefinition. Ensuite, nous avons ajouté la règle Ressource qui commence par le mot clé "ressource", après se trouve le nom de la ressource, après la "quantity" suivi d'un entier qui indique la quantité, et enfin "need" suivi des BesoinRessources qui incluent la ressource. Enfin, la règle BesoinRessources qui montrent les WorkDefinition qui ont besoin de la ressource et la quantité nécessaire pour leur démarrage.
La figure suivante montre un exemple de fichier pdl créé avec cette syntaxe :

```
process ex1 {
    wd a
    wd b
    wd c
    ws s2s from a to b
    ws f2f from b to c
    note test for wd a wd b
    ressource rs quantity 5
    need user a needs 2 of rs
    user b needs 2 of rs
}
```

Figure 6: Exemple de Modèle réalisé avec Xtext

7 Définition de syntaxes concrètes graphiques avec Sirius

À l'instar d'une syntaxe concrète textuelle, une syntaxe concrète graphique fournit un moyen de visualiser et éditer plus agréablement et efficacement un modèle. Sirius est un outil permettant justement de représenter les métamodèles graphiquement sous la forme d'un graphe composé de nœuds et d'arcs. Grâce à Sirius, nous avons pu représenter des modèles SimplePDL au format graphique. Dans notre représentation graphique (voir fichier simplepdl.odesign) , les WorkDefinition sont représentés par des losange, les WorkSequence par des flèche avec son type écrit dessus, les guidances par des carrés jaunes et des flèches vers les WorkDefinition concernés, les ressources par des cercles, et les BesoinRessources par des flèche vers les WorkDefinition concernés avec son besoin (la variable entier need) écrit dessus.
La figure suivante montre un exemple de représentation d'un modèle simplePDL:

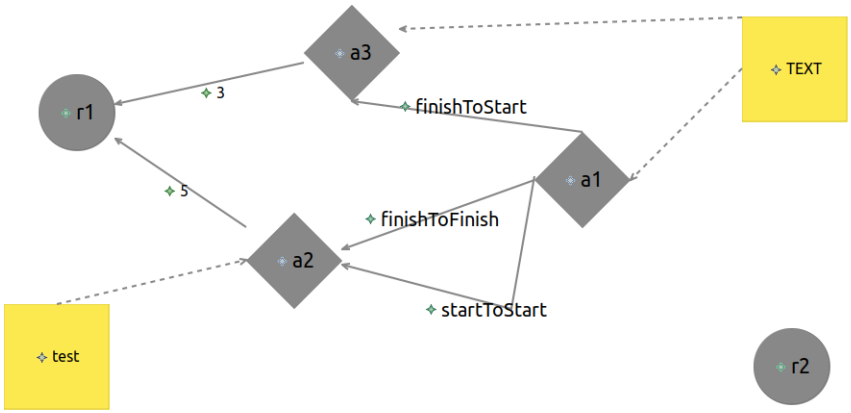

Figure 7: Exemple de Modèle réalisé avec Sirius

En outre, on peut modifier le graphe en ajoutant des éléments (voir palette dans la figure suivante), et les modifications effectué sur le graphe vont être appliqué sur le modèle originale

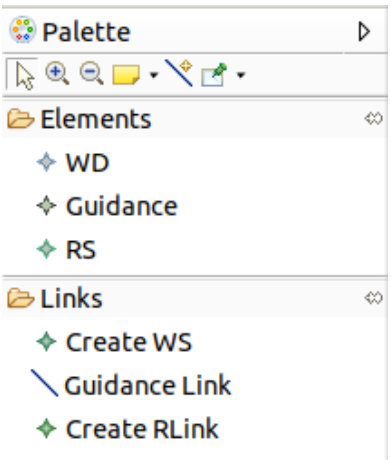

Figure 8: Palette de création des outils

8 Définition des propriétés LTL

La partie la plus difficile du mini-projet car il n'y avait pas assez de ressource dans les tp donc il fallait qu'on investit du temps à rechercher partout les syntaxe ltl
Ainsi, dans notre toLTL.mtl, on avait défini des règles LTL qui prend en considération qu'un processus terminé n'évolue plus, et que chaque processus est soit non commencée, soit en cours, soit terminée

9 Conclusion

Au travers de ce projet, nous avons pu nous familiariser avec la manipulation de méta-modèle sous Eclipse. Après avoir défini les métamodèles avec Ecore, nous avons pu les convertir à l'aide de Java et ATL puis à l'aide d'Acceleo pour les passer au format test et ainsi les exploiter dans d'autres logiciels comme Tina. Les différents modules réalisés permettent une meilleure compréhension d'un modèle SimplePDL en l'affichant (avec Sirius par exemple), en le convertissant vers d'autres modèles (PetriNet ou Tina). Le modèle créé peut par ailleurs être vérifié à l'aide des contraines OCL qui viennent compléter la description du métamodèle sous Ecore.