Projet IDM

Thomas Berbessou
Émile De Vos

Département Sciences du Numérique - Deuxième année - Parcours HPC et Big Data
2023-2024

Table des matières

Table des figures

1 Introduction

Ce projet a pour but de découvrir l'ingénierie des modèles. Pour ce faire, notre projet se porte sur la création de deux modèles, SimplePDL et le réseau de pétri. Lors de ce projet, après avoir écrit les méta-modèles des modèles, nous avons créé des outils pour écrire des modèles selon SimplePDL, pour traduire des modèles de SimplePDL au réseau de pétri et pour vérifier la justesse des modèles.

2 Mise en place des méta-modèles

Avant de travailler sur les modèles, il est essentiel de définir les méta-modèles, ou encore les "règles" de construction des modèles. Lors de ce projet, nous avons deux méta-modèles : SimplePDL et le réseau de Pétri.

2.1 Méta-modèle de SimplePDL

2.1.1 SimplePDL

Ce méta-modèle se place sous la forme d'un processus. Un processus possède plusieurs éléments. Ces éléments pouvant être des tâches (WorkDefinition), des conditions entre les tâches (WorkSequence) ou encore des ressources (Ressource). Chaque tache a besoin de ressources pour s'effectuer et les rend une fois terminé. De plus, pour commencer ou pour finir, une tâche doit valider les conditions données par les WorkSequence. Enfin, le méta-modèle permet de placer des commentaires.

2.1.2 Ajout des ressources

La première tâche de ce miniprojet était de compléter le méta-modèle de SimplePDL pour prendre en compte les ressources. En effet, comme expliquer plus haut, une WorkDefinition aura certainement besoin d'utiliser une ressource pour s'exécuter. On devra alors rajouter deux classes :
- La classe Ressource qui est décrite par son nom (ESTRING) et un entier qui indique combien de ressource de ce type que nous avons à notre disposition.
- La classe QuantiteRessource permet, quant à elle, de faire le lien entre une WorkDefinition et une ressource, en indiquant de combien de ressource la WorkDefinition doit disposer pour fonctionner.

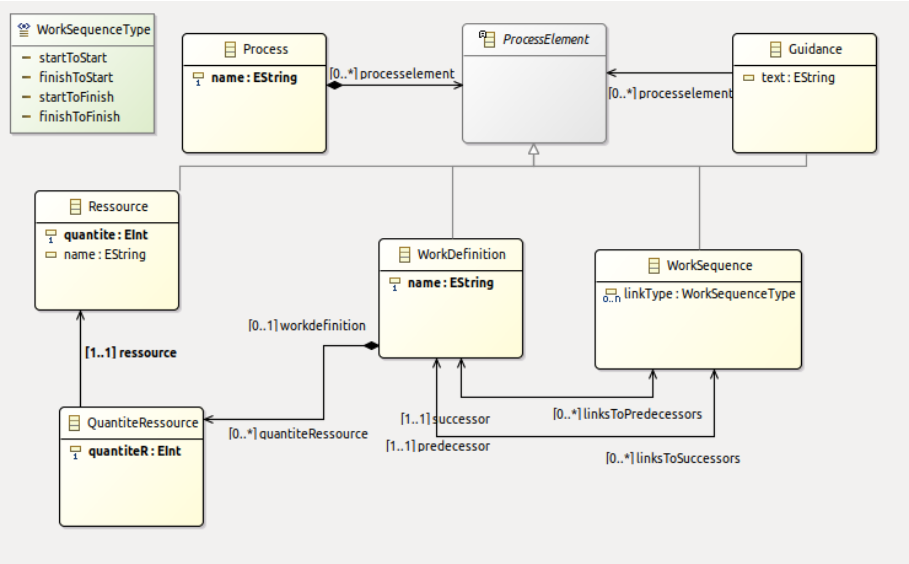

FIGURE 1 – méta-modèle SimplePDL

2.2 Méta-modèle de Pétri

2.2.1 PetriNet

Ce méta-modèle est composé de places, de transitions et d'arcs. Les places possèdent un certain nombre de jetons. les jetons passent de places en places au travers des transitions et des arcs. les arcs possèdent un poids signifiant le nombre de jetons qu'ils prennent ou donnent. Il est défini de façon plus rigoureuse de la façon suivante : Un réseau de pétri (aussi connu comme un réseau de Place/Transition ou réseau de P/T) est un modèle mathématique servant à représenter divers systèmes travaillant sur des variables discrètes.

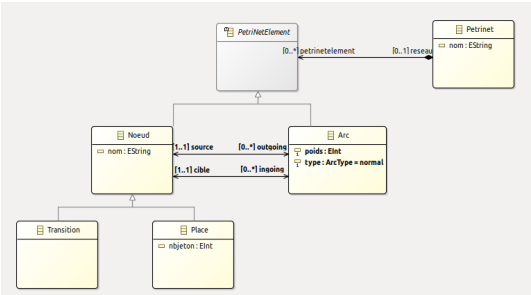

FIGURE 2 – méta-modèle de PetriNet

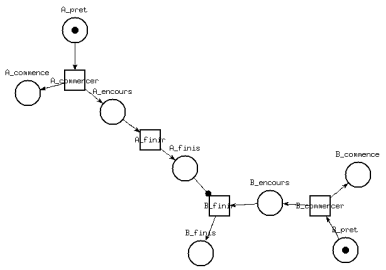

FIGURE 3 – exemple réseau de petri

2.2.2 Prise en compte des ressources

Nous n'avons pas besoin de modifier le méta-modèle de PetriNet pour prendre en compte les ressources. Les ressources disponibles seront modélisées par une place portant leur nom, et le nombre de ressources disponibles est modélisé par le nombre de jetons présent dans les arcs reliant la place ressource et la transition commencer pour permettre à un réseau de pétri pour commencer. On crée également un arc de la transition terminé à la place de la ressource afin de rendre la ressource à la fin.

3 Écriture des modèles

Pour écrire des modèles suivant le méta-modèle SimplePDL, nous devons utiliser des outils. Il existe plusieurs types d'outils pour créer ces modèles. Durant notre projet, nous avons développé deux types d'outils, un outil graphique (Sirius) et un outil textuel (Xtext).

3.1 Outil Sirius

3.1.1 Affichage graphique

La première étape de la mise en place de l'outil Sirius a été la mise en place d'un affichage graphique du modèle. Pour ce faire, nous avons représenté chaque élément par une figure selon certaines règles :
- Une WorkDefinition est représenté par un losange gris portant le nom de la WorkDefinition.
- Une WorkSequence est représenté par une flèche partant de WorkSequence.predecessor et allant à WorkSequence.successor. La flèche porte le type de la WorkSequence comme nom.
- Une Guidance (un commentaire) est représenté par un carré jaune possédant le commentaire comme nom et des flèches reliant la Guidance à ce qu'elle commente.
- Une ressource est représentée par une ellipse portant le nom de la ressource et le nombre de ressources disponibles comme attribut.
- Une QuantiteRessource est représenté par une flèche allant de la ressource demandée à la WorkDefinition qui la demande.

3.1.2 Palette d'outils

Une fois avoir créé les outils nécessaires à l'affichage des modèles, il nous fallait pouvoir modifier ces modèles. Pour ce faire, nous avons créé une palette d'outils permettant d'éditer un modèle.
Voici les règles de production, commençons par les nœuds : - WDCReation permet de créer un nœud qui est une WorkDefinition - GCreation permet de créer un nœud guidance et de remplir un commentaire. - RCreation permet de créer une ressource dans le modèle en lui donnant un nom et un nombre, correspondant à la quantité totale de cette ressource disponible. On a également les arcs reliant les nœuds entre eux : - WSEdge permet de créer une WorkSequence reliant deux WorkDefinition, en leur donnant des contraintes, comme Start2Start. - GEdge permet de relier une guidance avec une WorkDefinition pour lui affecter un commentaire. - QREdgeCreation permet de relier une WorkDefinition avec une ressource en indiquant un entier, qui correspondra au nombre de ressources que la WorkDefinition aura besoin de prendre pour marcher.

3.1.3 Exemple

On peut voir ci-dessous un exemple simple d'un modèle créé et visualisé avec Sirius. Cet exemple nous servira de fil rouge pour illustrer les différents outils et étapes de ce projet.

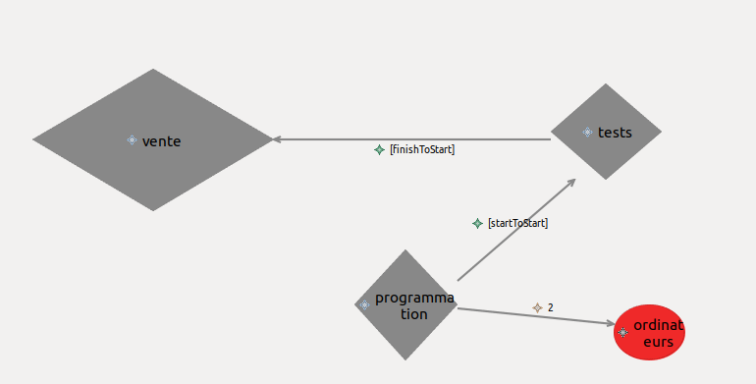

FIGURE 4 – modèle SimplePDL simple avec Sirius

Nous pouvons voir que ce modèle est composé de deux WorkDefinition (A et B) ainsi que d'une WorkSequence de type FinishToFinish de A vers B.

3.2 Outils Xtext

L'outil Xtext est un outil textuel de création de modèle selon SimplePDL. Il permet de traduire du texte en un modèle. Voici les règles de production :
- Le mot "process " permet de créer un process
- Le mot "wd" permet de créer une WorkDefinition. Lorsqu'il est suivi de "", cela crée un QuantiteRessource - Le mot "qr" permet de créer une QuantiteRessource, il est suivi du nombre de ressources demandé, du demandeur de ressource ainsi que du type de ressource demandé.
- Le mot "ws" permet de créer une WorkSequence. il est suivi du type de WorkSequence, du mot "from" puis de l'origine de la WorkSequence, de to et enfin de la cible de la WorkSequence.
- Le mot "res" permet de créer une ressource. Il est suivi du nom de la ressource, du mot "qt" puis de la quantité de la ressource disponible au process.

```
process ex2 {
res ordi qt 4
wd a { qr 4 utiPar a besoinde ordi }
wd b
wd c
ws s2s from a to b
ws f2f from b to c
}
```

FIGURE 5 – exemple de modèle avec Xtext

4 Traduction des modèles

4.1 Introduction

ATL est un langage de programmation permettant la transformation de modèle à modèle. Dans notre cas, nous avons programmé une transformation du méta-modèle SimplePDL vers le méta-modèle de PetriNet. Pour faire cette transformation, il faut saisir comment vont être transformé chaque élément d'un modèle de processus SimplePDL.

4.2 Traduction des WorkDefinition

Une WorkDefinition sera transformée en réseau de pétri composé de quatre places, deux transitions et cinq arcs. L'une de ces places servira à implémenter les relations des WorkDefinition du type startToX.

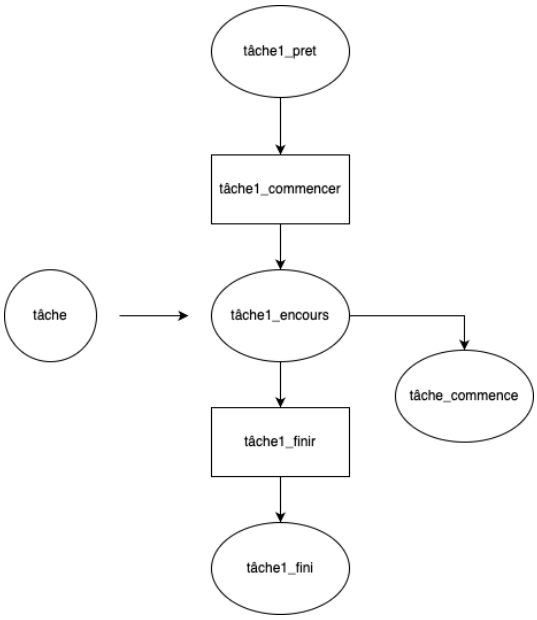

FIGURE 6 – transformation d'une tâche de SimplePDL à PetriNet

4.3 Traduction des WorkSequence

Comme on le voit sur l'image précédente, une place Acommence est présente dans le réseau de pétri. C'est grâce à elle que les WorkSequence seront codés. En effet, dans le cas où deux WorkDefinition seront reliés par une WorkSequence StartToFinish, leur réseau de pétri seront reliés de façon à ce que ceci soit respecté, c'est-à-dire de la place Acommence à la transition termine. Pour les autres types de WorkSequence, l'idée est la même, mais les liens sont faits différemment.

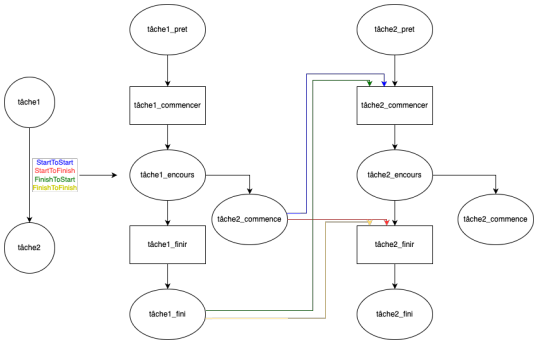

FIGURE 7 – transformation d'une WorkSequence de SimplePDL à PetriNet

4.4 Traduction des Ressources

Enfin, comme nous avons ajouté les ressources dans le méta-modèle SimplePDL, il faut aussi les prendre en compte quand on fait la transformation vers PetriNet. Ainsi, les ressources seront matérialisées par une place, initialisé avec autant de jeton qu'il y a de ressource initialement. Les ressources seront reliées d'un côté avec la transition commencé de la WorkDefinition avec laquelle elle est liée par un arc pondéré par la quantité de ressource nécessaire, afin qu'on ne puisse commencer que si on a assez de ressources. Enfin, lors la transition finir, on crée également un arc vers la place ressource avec la même pondération afin de rendre les ressources utilisées par l'utilisateur.

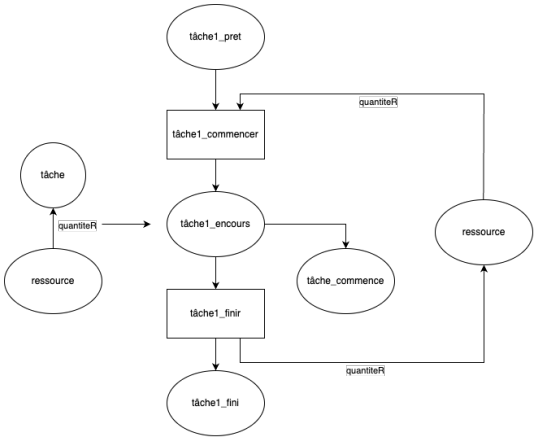

FIGURE 8 – transformation d'une tâche avec implémentation des ressources

voici la traduction du modèle créé avec Sirius en réseau de pétri :

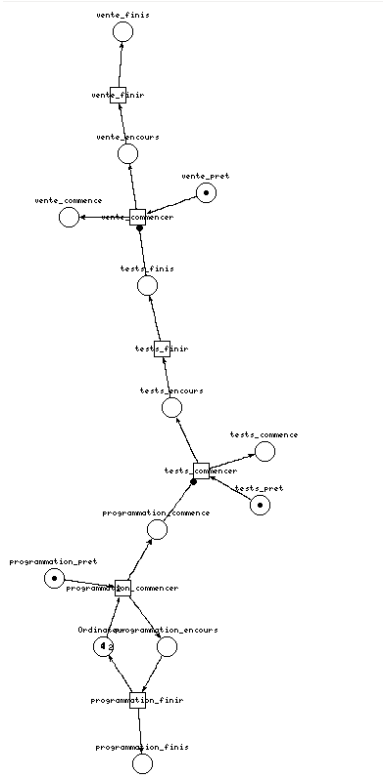

FIGURE 9 – exemple de modèle en réseau de pétri

```
net devRessource
pl vente_pret (1)
pl vente_encours (0)
pl vente_finis (0)
pl vente_commence (0)
pl tests_pret (1)
pl tests_encours (0)
pl tests_finis (0)
pl tests_commence (0)
pl programmation_pret (1)
pl programmation_encours (0)
pl programmation_finis (0)
pl programmation_commence (0)
pl Ordinateur (4)
tr vente_commencer  vente_pret*1 -> vente_commence *1  vente_encours *1
tr vente_finir  vente_encours*1  tests_finis?1 -> vente_finis *1
tr tests_commencer  tests_pret*1 -> tests_commence *1  tests_encours *1
tr tests_finir  tests_encours*1  programmation_finis?1 -> tests_finis *1
tr programmation_commencer  programmation_pret*1  Ordinateur*2 -> programmation_commence *1  programmation_encours
tr programmation_finir  programmation_encours*1 -> programmation_finis *1  Ordinateur *2
```

FIGURE 10 – exemple de modèle en réseau de pétri textuel

Nous pouvons voir deux façons différentes de visualiser le réseau de pétri. L'une est textuelle tandis que l'autre est graphique. Ces deux affichages présentent tous deux des avantages et des inconvénients. L'affichage graphique est idéal pour visualiser l'architecture globale du réseau de pétri alors que l'affichage textuel permet de mieux voir la composition du réseau de pétri (nombre de places, de transitions et de ressources)

5 Vérification des modèles

5.1 Contraintes OCL

Les contraintes OCL permettent de vérifier les propriétés sémantiques relatives au méta-modèle. Ces contraintes permettent de vérifier que les modèles sont bien écrits et respectent le méta-modèle. Ces règles concernent notamment la validité des noms des différents éléments, de la construction des WorkSequences ainsi que des ressources (Le nombre de ressources doit être positif).

5.2 Contraintes LTL

Les contraintes LTL permettent de vérifier des propriétés temporelles relatives au Réseau de Pétri à l'aide de la boite à outils Tina. En effet, la transformation Acceleo toTina permet dans un premier temps de transformer un modèle PetriNet en fichier texte .net qui est reconnu par la boite à outil Tina. Dans notre cas, le modèle de base est un modèle respectant le méta-modèle SimplePDL qui à était transformé en réseau de pétri à l'aide d'une transofrmation modèle à texte. Donc en écrivant les propriétés LTL et en les vérifiant sur ce modèle .net, nous pourrons vérifier des propriétés sur les modèles de simplePDL.

Ainsi, on va s'interesser à la terminaison des modèles SimplePDL. Pour cela, on va donc créer une transformation d'un modèle SImplePDL pour générer automatiquement des propriétés LTL que nous vérifierons avec le fichier .net et l'application selt.

On va donc écrire des propriétés dans ce sens : On écrit d'abord une fonction getWorkDefinition() qui permet de créer un ensemble ordonné de toutes les WorkDefinition.

On va ensuite créer les variables toutFinit, toutEncours et toutPret qui correspondent au fait que toutes les workdefinition sont sur l'état fini ou sur l'état prêt.

```
[comment encoding = UTF-8 /]
[module toLTL('http://simplepdl')]


[template public processToLTL(aProcess : Process)]
[comment @main/]

[file (aProcess.name +'.ltl', false, 'UTF-8')]
[let wd : OrderedSet(WorkDefinition) =aProcess.getWordefinition()]

op toutfinit = [for (workDef : WorkDefinition | wd)] [workDef.name/]_finis /\ [/for]1;
op toutPret = [for (workDef: WorkDefinition | wd)] [workDef.name/]_pret /\ [/for]1;
op toutEncours = [for (workDef : WorkDefinition | wd)] [workDef.name/]_encours /\ [/for]1;
op toutCommencer = [for (workDef : WorkDefinition | wd)] [workDef.name/]_commencer /\ [/for]1;



['['/]['']'/] <> toutfinit;
['['/]['']'/]( toutfinit => dead);
['['/]['']'/](toutfinit => ['['/]['']'/] toutfinit);

[for (workDef : WorkDefinition | wd)] ['['/]['']'/] ([workDef.name/]_finis + [workDef.name/]_encours + [workDef.name/]_pret =1);
[/for]
[/let]

[let res : OrderedSet(Ressource) = aProcess.getR()]
[for (r : Ressource | res)] ['['/]['']'/] (tousfinit => [r.name/] = [r.quantite/]);
[/for]
[/let]




[/file]
[/template]

[query public getWordefinition(p: Process) : OrderedSet(WorkDefinition) =
    p.processelement->select( e | e.oclIsTypeOf(WorkDefinition) )
    ->collect( e | e.oclAsType(WorkDefinition) )
    ->asOrderedSet()
/]
```

FIGURE 11 – code génération LTL

On va ensuite écrire les propriétés qu'on peut retrouver sur la figure ci-dessus. La première permet de s'assurer qu'à tout moment, il existe une séquence qui permet au Processus de finir, car toutes les WorkDefinition seront finies. Ensuite, on s'assure que si tout est fini, les états correspondant à chaque WorkDefinition n'évolueront plus. Enfin, pour chaque WorkDefinition, on vérifie si elle est seulement dans un état en même temps, c'est-à-dire pret, finit ou encours. Enfin, nous vérifions que toutes les ressources sont bien rendues à la fin des processus en comparant les quantités de ressource finale et initiale. Cette étape permet de juger du bon fonctionnement des ressources et de s'assurer qu'il n'y a eu, ni duplication, ni perte de ressource durant le processus. La vérification de la duplication des ressources évite également la possibilité qu'un processus commence si les ressources nécessaires ne sont pas disponibles.

```
1
2 op toutfinit =  vente_finis /\  tests_finis /\  programmation_finis /\ 1;
3 op toutPret =  vente_pret /\  tests_pret /\  programmation_pret /\ 1;
4 op toutEncours =  vente_encours /\  tests_encours /\  programmation_encours /\ 1;
5 op toutCommencer =  vente_commencer /\  tests_commencer /\  programmation_commencer /\ 1;
6
7
8
9 [] <> toutfinit;
10 [](  toutfinit => dead);
11 [](toutfinit => [] toutfinit);
12
13 [] (vente_finis + vente_encours + vente_pret =1);
14 [] (tests_finis + tests_encours + tests_pret =1);
15 [] (programmation_finis + programmation_encours + programmation_pret =1);
16
17 [] (tousfinit => Ordinateur = 4);
18
19
20
21
22
23
```

FIGURE 12 – exemple de contraintes LTL

Le code ci-dessus est le code représentant les contraintes LTL pour le modèle qui nous a servis d'exemple tout au long de ce rapport.

5.3 Conclusion

Lors de ce projet, nous avons tout d'abord créé les méta-modèle de SimplePDL et PetriNet puis, nous avons produit divers outils autours de ces méta-modèles. Nous avons produit des outils pour produire, visualiser et éditer des modèles suivant SimpleDL, des outils pour traduire ces modèles de SImplePDL vers PetriNet et enfin, des outils pour visualiser et vérifier la justesse des modèles en réseau de pétri.