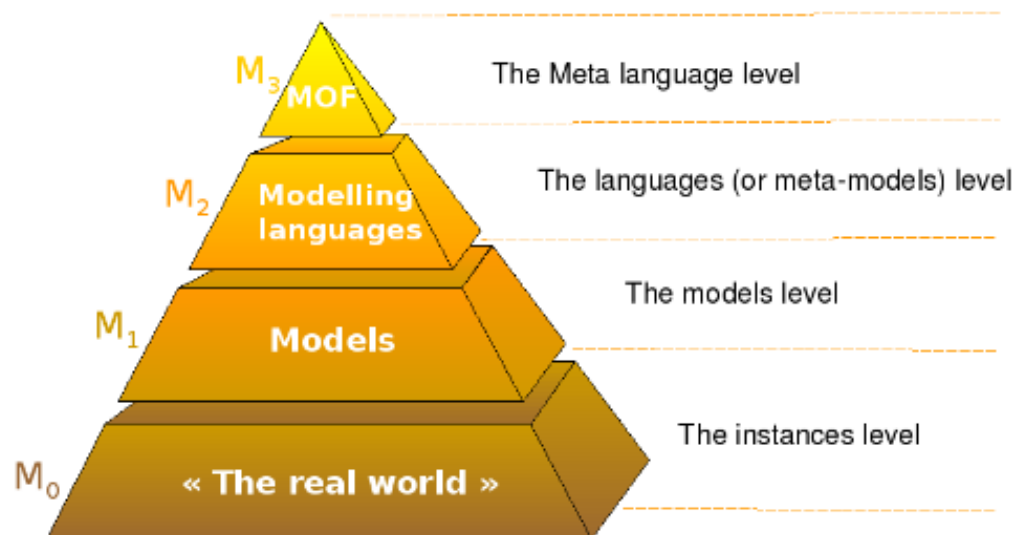


Chaîne de vérification de modèles de processus

Mini-projet IDM

Chaimaa Louahabi - Ahmed Ghanim

November 14, 2020



Contents

1	Objectifs du travail réalisé	3
2	SimplePDL :	3
2.1	Définition du métamodèle SimplePDL avec Ecore.	3
2.2	Définition de la sémantique statique avec OCL.	3
2.3	Des exemples de modèles simplePDL	3
2.3.1	le modèle de procédé de programmation/développement	3
2.3.2	le modèle de procédé de fonctionnement d'un magasin	4
3	PetriNet	4
3.1	Définition du métamodèle PetriNet avec Ecore.	4
3.2	Définition de la sémantique statique avec OCL.	5
3.3	Des exemples de modèles PetriNet	6
3.3.1	Modèle des saisons	6
3.3.2	Modèle de transformations chimiques de H_2O	6
4	Définition de transformations modèle à texte (M2T) avec Acceleo	6
4.1	Engendrer la syntaxe attendue par Tina à partir d'un modèle de réseau de Petri :	6
4.2	Engendrer les propriétés LTL à partir d'un modèle de processus :	7
5	Définition d'une transformation de modèle à modèle (M2M)	7
5.1	Avec EMF/Java : SimplePDL2PetriNet.java	7
5.2	Avec ATL : SimplePDL2PetriNet.atl	8
6	Définition de syntaxes concrètes textuelles avec Xtext	8
7	Définition de syntaxes concrètes graphiques avec Sirius	9
8	Application: Chaîne de vérification du modèle de processus 'magasin'	10
9	Conclusion	11
10	Références	11

List of Figures

1	Le métamodèle simplePDL avec ressources	3
2	Exemple de modèle de procédé de développement	4
3	Le modèle de fonctionnement d'un magasin	4
4	Le métamodèle PetriNet	5
5	Message d'erreur provenant du non-respect d'une contrainte OCL	5
6	un modèle de petriNet représentant les saisons de l'année	6
7	un modèle de petriNet représentant la transformation chimique de H_2O	6
8	Le résultat de NetDraw de saisons.net par TINA	7
9	Le résultat de NetDraw de h2o.net par TINA	7
10	Exemples de l'utilisation de la syntaxe textuelle définie précédemment	8
11	L'Outline lors de la rédaction des règles de grammaire de Figure 10a et 10b	9
12	Palette des outils pour les modèles SimplePDL	9
13	Exemples de visualisation graphiques des modèles de SimplePDL	10
14	Visualisation sur TINA de magasin.net	10
15	Résultat de model-checking de magasin.ltl avec selt	10

1 Objectifs du travail réalisé

Ce mini-projet consiste à produire une chaîne de vérification de modèles de processus Simple-PDL 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.

2 SimplePDL :

2.1 Définition du métamodèle SimplePDL avec Ecore.

Pour pouvoir introduire les ressources, il faut gérer ses allocations et ses libérations. Ainsi, on a introduit la classe 'allocation' qui comporte un attribut 'count' représentant le nombre d'entités allouées d'une ressource donnée.

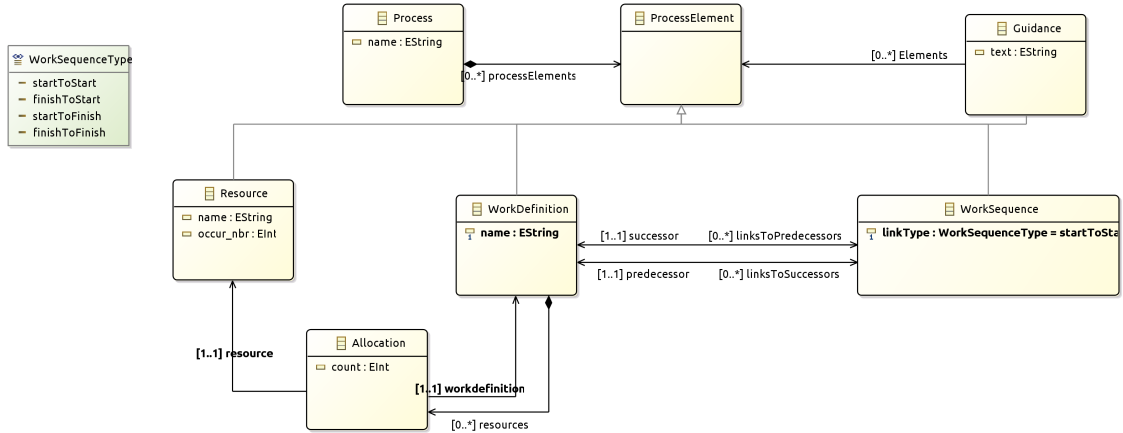


Figure 1: Le métamodèle simplePDL avec ressources

2.2 Définition de la sémantique statique avec OCL.

OCL est un moyen de préciser la sémantique du métamodèle en limitant les modèles conformes. Appliqué au niveau du métamodèle, il permet d'ajouter des propriétés qui n'ont pas pu être capturées par les concepts fournis par le métamodèle.

On l'a utilisé pour imposer quelques invariants sur les modèles de SimplePDL, comme :

- UniqueResourceNames: l'unicité des noms des ressources dans un processus.
- uniqueWorkDefinitionNames: l'unicité des noms des activités dans un processus.
- AllocationIsLessThanResourceOccurence : l'homogénéité du nombre d'allocations d'une ressource avec son occurrence c'est-à-dire le nombre d'occurrences allouées par une activité ne peut pas dépasser la quantité totale de la ressource.
- validName: les noms des ressources et des activités doivent respecter cette expression régulière '[A - Za - z_][A - Za - z0 - 9_]*'
- successorAndPredecessorAreDifferent: une workSequence ne peut pas lier une workDefinition avec soi-même.

2.3 Des exemples de modèles simplePDL

2.3.1 le modèle de procédé de programmation/développement

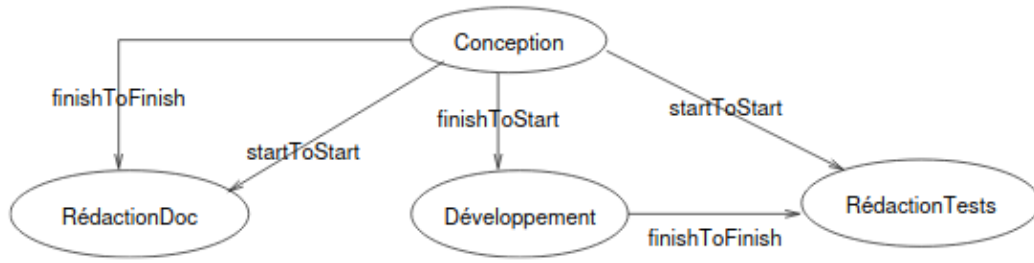


Figure 2: Exemple de modèle de procédé de développement

2.3.2 le modèle de procédé de fonctionnement d'un magasin

Le schéma suivant correspond à une représentation du modèle pddl-magasin.xmi qui décrit le fonctionnement d'un magasin qui reçoit les commandes des clients, les prépare puis les livre.

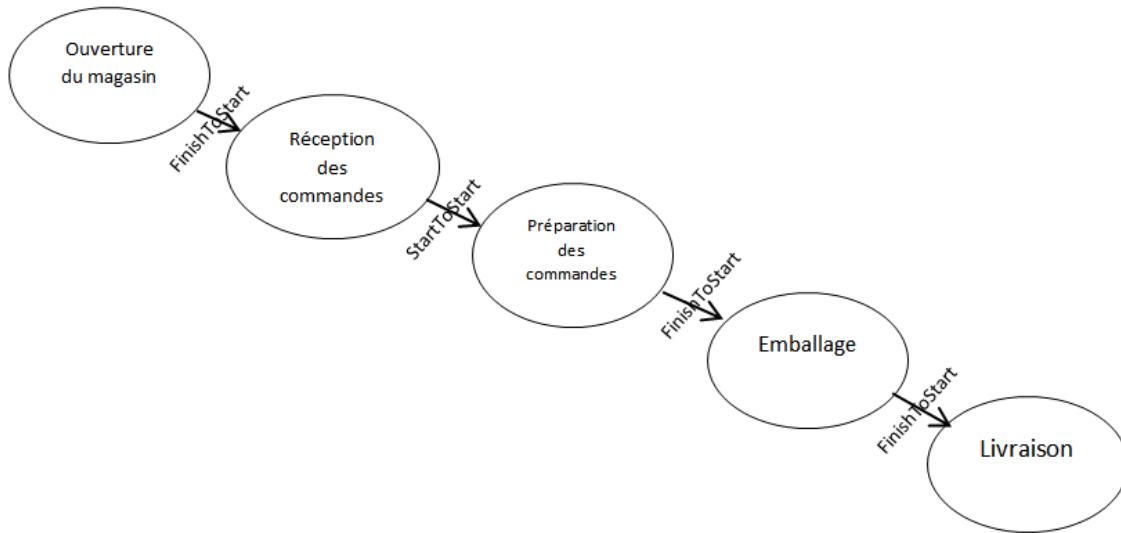


Figure 3: Le modèle de fonctionnement d'un magasin

Voici une affectation possible de ressources pour le processus décrit à la figure 3 . Une ligne correspond à un type de ressource. Elle indique la quantité totale de la ressource ainsi que le nombre d'occurrences de cette ressource nécessaire à la réalisation d'une activité.

	Quantité	Ouverture	Réception	Préparation	Emballage	Livraison
Réceptionniste	1	1	1	0	0	0
Préparateur de commandes	2	0	0	1	1	0
Emballages	100	0	0	0	2	0
livreur	3	0	0	0	0	1

3 PetriNet

3.1 Définition du métamodèle PetriNet avec Ecore.

Un réseau de petri est principalement composé de places, transitions et arcs liant ces deux derniers entre eux. Son métamodèle est présenté à Figure 4 .

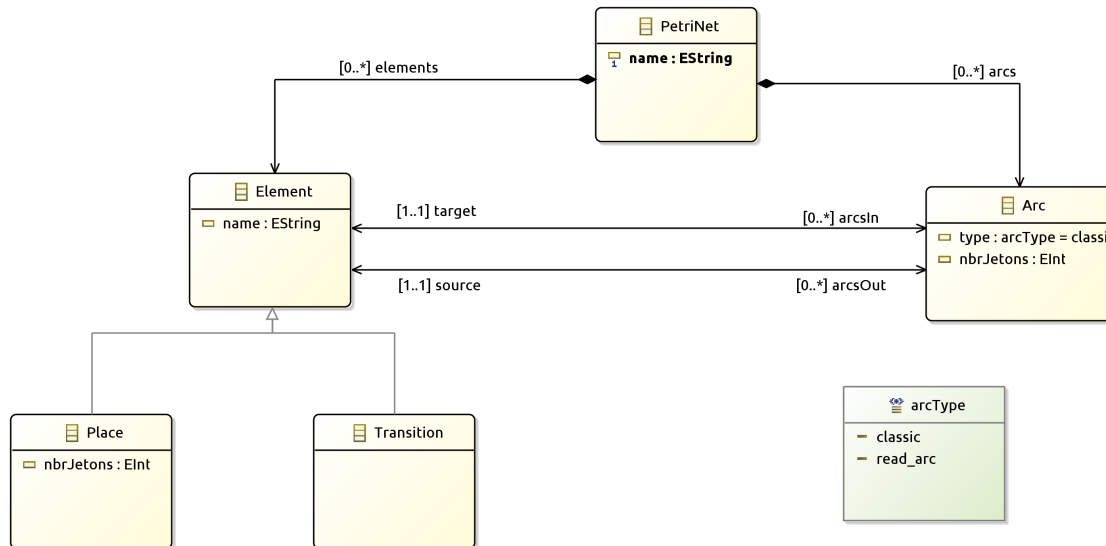


Figure 4: Le métamodèle PetriNet

3.2 Définition de la sémantique statique avec OCL.

Le métamodèle de PetriNet réalisé avec ecore n'as pas exprimé toutes les contraintes fondamentales d'un réseau de Petri, donc on a eu recours à OCL.

- **validName**: les noms des places et des transitioins doivent respecter cette expression régulière $'[A - Z a - z _][A - Z a - z 0 - 9 _] *'$
- **UniquePlaceNames/ UniqueTransitionNames**: l'unicité des noms des places/transitions dans un reseau de Petri.
- **PoidsDArcSuperieurEgalA1**: le poids d'un arc quelconque dois toujours être supérieur ou égal à un .
- **PasDArcsEntrePlacesNiEntreTransitions**: un arc ne peut pas relier deux places ou deux transitions entre elles.
Les modèles ko-net-arcEntePlaces.xmi et ko-net-arcEntreTransitions.xmi nous ont servi pour tester cette contrainte. La figure 5 est le résultat de la validation du modèle.
- **ReadArcDepuisPlaceVersTransitionSeulement**: un arc de type *read_arc* va depuis une place vers une transition il ne peut pas avoir une transition comme source.

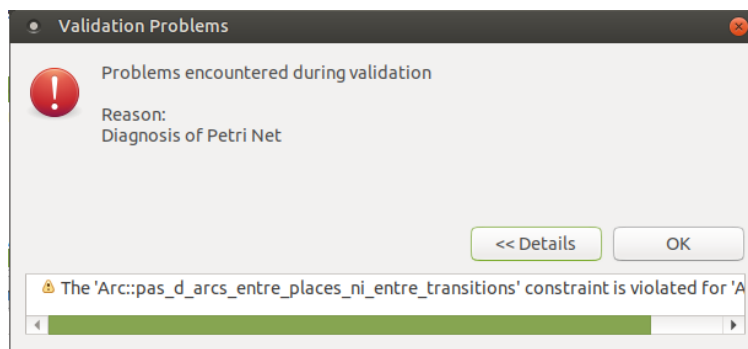


Figure 5: Message d'erreur provenant du non-respect d'une contrainte OCL

3.3 Des exemples de modèles PetriNet

3.3.1 Modèle des saisons

C'est la représentation graphique du fichier net-saisons.xmi qui represente le changement des saisons de l'année:

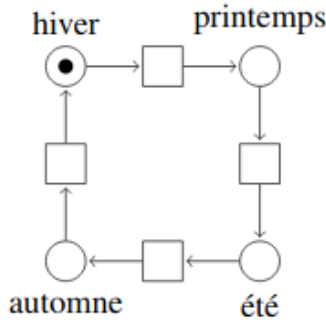


Figure 6: un modèle de petriNet représentant les saisons de l'année

3.3.2 Modèle de transformations chimiques de H_2O

c'est un exemple de modèle simplifié représentant la formation et la dissociation des molécules de H_2O [3] (le fichier net-h2o.xmi) .

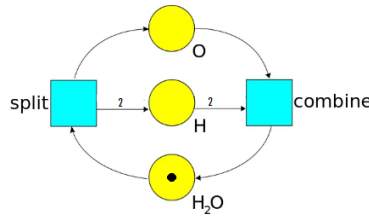


Figure 7: un modèle de petriNet représentant la transformation chimique de H_2O

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

4.1 Engendrer la syntaxe attendue par Tina à partir d'un modèle de réseau de Petri :

Pour pouvoir profiter de la boîte à outils TINA(Time Petri Net Analyzer) qui fait l'analyse de réseaux de Petri, on a besoin de générer un fichier .net à partir de notre modèle PetriNet .

En utilisant Accelele , le code toTina.mtl traduit un modèle de PetriNet en des règles de type "*pl*" et "*tr*".

Nous avons engendré les fichiers h2o.net et saisons.net correspondants respectivement aux modèles net-h2o.xmi (Figure 7) et net-saisons.xmi (Figure 6) afin de le visualiser via NetDraw.

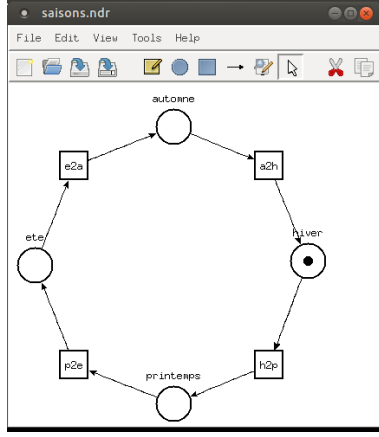


Figure 8: Le résultat de NetDraw de saisons.net par TINA

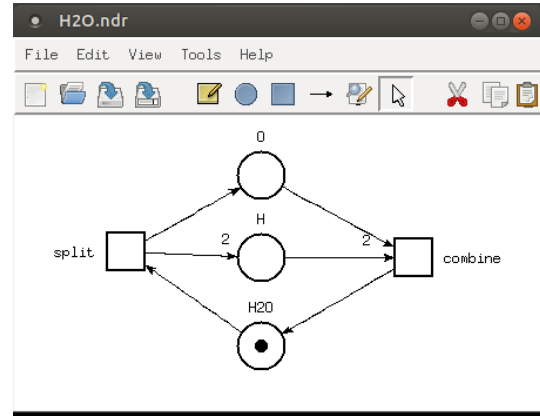


Figure 9: Le résultat de NetDraw de h2o.net par TINA

4.2 Engendrer les propriétés LTL à partir d'un modèle de processus :

La question de savoir si un procédé est réalisable peut se ramener à une étude de la terminaison de l'exécution du procédé. On considère que le procédé est terminé si chacune des activités prévues dans le procédé ont effectivement été exécutés.

Le code de "toLTL.mtl" est une transformation modèle à texte qui prend en entrée un modèle de SimplePdl et traduit les propriétés de sûreté suivantes qui seront vérifiées par le model checker *selt*.

- $\square (\text{dead} \Rightarrow \text{finished})$: " tout procédé terminé est dans son état final "
- $\square \langle \rangle \text{dead}$: "toute exécution se termine"
- $\neg \langle \rangle \text{finished}$: Si le processus peut finir, cette propriété sera fausse et le model checker exhibera un contre-exemple qui correspond à un scénario où le processus se termine.

Afin de valider la transformation SimplePDL vers PetriNet, on a vérifié que les invariants sur le modèle de processus sont préservés sur le modèle de réseau de Petri correspondant. Comme par exemple le faite que chaque état 'ready' sera nécessairement suivi par 'started' puis 'running', ensuite 'finished' et finalement toute l'exécution se termine 'dead'. Prenant l'exemple de l'activité RedactionDocs; Cette propriété de sureté s'écrit comme suit :

- `op RedactionDocs_state = (RedactionDocs_ready =><> RedactionDocs_started =><> RedactionDocs_running =><> RedactionDocs_finished =><> dead);`

5 Définition d'une transformation de modèle à modèle (M2M)

Pour utiliser les outils de simulation et de vérification proposés par la boîte à outils TINA sur un modèle de procédé, il faut le traduire en réseaux de Petri. Cette tranformation peut-être réalisée en utilisant EMF/Java ou ATL.

Chaque *activité* est traduite en quatre places caractérisant son état (ready,Started,Running,Finished) et deux transitions (Start et Finish).

Chaque *WorkSequence* est traduit en un read-arc entre une place de l'activité source et une transition de l'activité cible, la place et la transition dépendant de la valeur de l'attribut ArcType.

Chaque *Resource* est traduite en une place dont le nombre de tokens est égal à la quantité totale de cette ressource.

Chaque *Allocation* est traduite en deux arcs, un depuis la transition start vers la ressource, et l'autre depuis la ressource vers la transition finish.

5.1 Avec EMF/Java : SimplePDL2PetriNet.java

Le code Java correspondant comprend trois méthodes: convertResource, convertWorkSeq et convertWorkDef. Cinq HashMaps sont utilisés pour enregistrer les places et les transitions, ça nous permettra de retrouver les éléments créés dans une autre fonction.

5.2 Avec ATL : SimplePDL2PetriNet.atl

Le code ATL correspondant comprend cinq règles déclaratives: WorkDefinition2PetriNet, WorkSequence2PetriNet, Resource2PetriNet, Allocation2PetriNet et Process2PetriNet qui traduit Process en un PetriNet regroupant les nœuds et arcs construits par les autres règles.

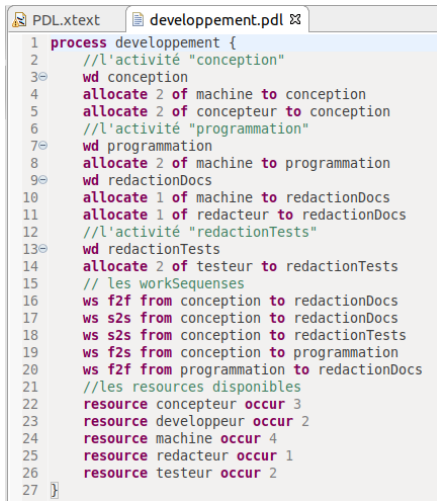
L'opération resolveTemp d'ATL, qui permet de retrouver un élément particulier du modèle cible créé dans une autre règle, est nécessaire car plusieurs places et transitions sont créées pour une même WorkDefinition.

6 Définition de syntaxes concrètes textuelles avec Xtext

Xtext permet d'associer à une syntaxe abstraite une syntaxes concrète simple et assimilable par tous sans prérequis informatique.

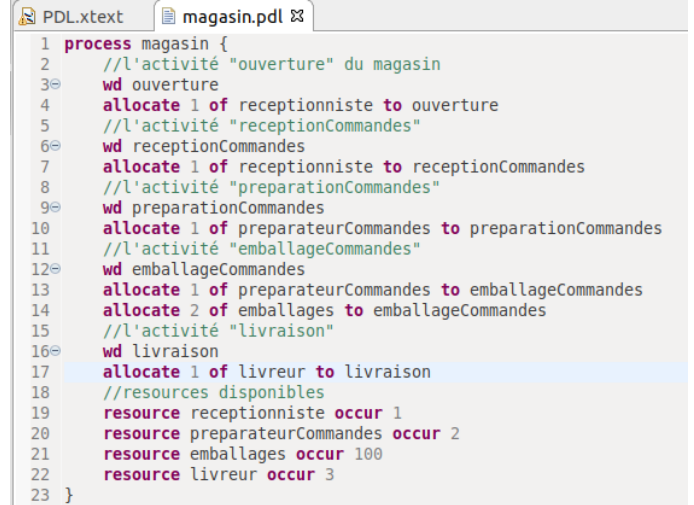
Le fichier PDL.xtext définit une grammaire pour créer des modèles de SimplePDL avec les règles de construction simples suivantes :

- "process ..." : définir un processus.
- "wd nomWD" : créer une workDefinition.
- "allocate nombre of nomRessource to nomWD" : allouer un nombre de ressource à une activité.
- "ws linkType from wd1 to wd2" : lier wd1 à wd2 par une relation de type linkType.
- "resource nomRes occur number" : déclarer la ressource et sa quantité totale.



```
1 process developpement {
2   //l'activité "conception"
3   wd conception
4   allocate 2 of machine to conception
5   allocate 2 of concepteur to conception
6   //l'activité "programmation"
7   wd programmation
8   allocate 2 of machine to programmation
9   wd redactionDocs
10  allocate 1 of machine to redactionDocs
11  allocate 1 of redacteur to redactionDocs
12  //l'activité "redactionTests"
13  wd redactionTests
14  allocate 2 of testeur to redactionTests
15  // les workSequences
16  ws f2f from conception to redactionDocs
17  ws s2s from conception to redactionTests
18  ws f2s from conception to programmation
19  ws f2f from programmation to redactionDocs
20  //les ressources disponibles
21  resource concepteur occur 3
22  resource developpeur occur 2
23  resource machine occur 4
24  resource redacteur occur 1
25  resource testeur occur 2
26 }
27 }
```

(a) Le modèle "developpement"



```
1 process magasin {
2   //l'activité "ouverture" du magasin
3   wd ouverture
4   allocate 1 of receptionniste to ouverture
5   //l'activité "receptionCommandes"
6   wd receptionCommandes
7   allocate 1 of receptionniste to receptionCommandes
8   //l'activité "preparationCommandes"
9   wd preparationCommandes
10  allocate 1 of prepareurCommandes to preparationCommandes
11  //l'activité "emballageCommandes"
12  wd emballageCommandes
13  allocate 1 of prepareurCommandes to emballageCommandes
14  allocate 2 of emballages to emballageCommandes
15  //l'activité "livraison"
16  wd livraison
17  allocate 1 of livreur to livraison
18  //ressources disponibles
19  resource receptionniste occur 1
20  resource prepareurCommandes occur 2
21  resource emballages occur 100
22  resource livreur occur 3
23 }
```

(b) Le modèle "magasin"

Figure 10: Exemples de l'utilisation de la syntaxe textuelle définie précédemment

Sur Eclipse, on peut visualiser sur l'Outline comment le modèle EMF est construit en mémoire au fur et à mesure de l'analyse syntaxique des règles de grammaire saisies, comme montré sur les figures suivantes :

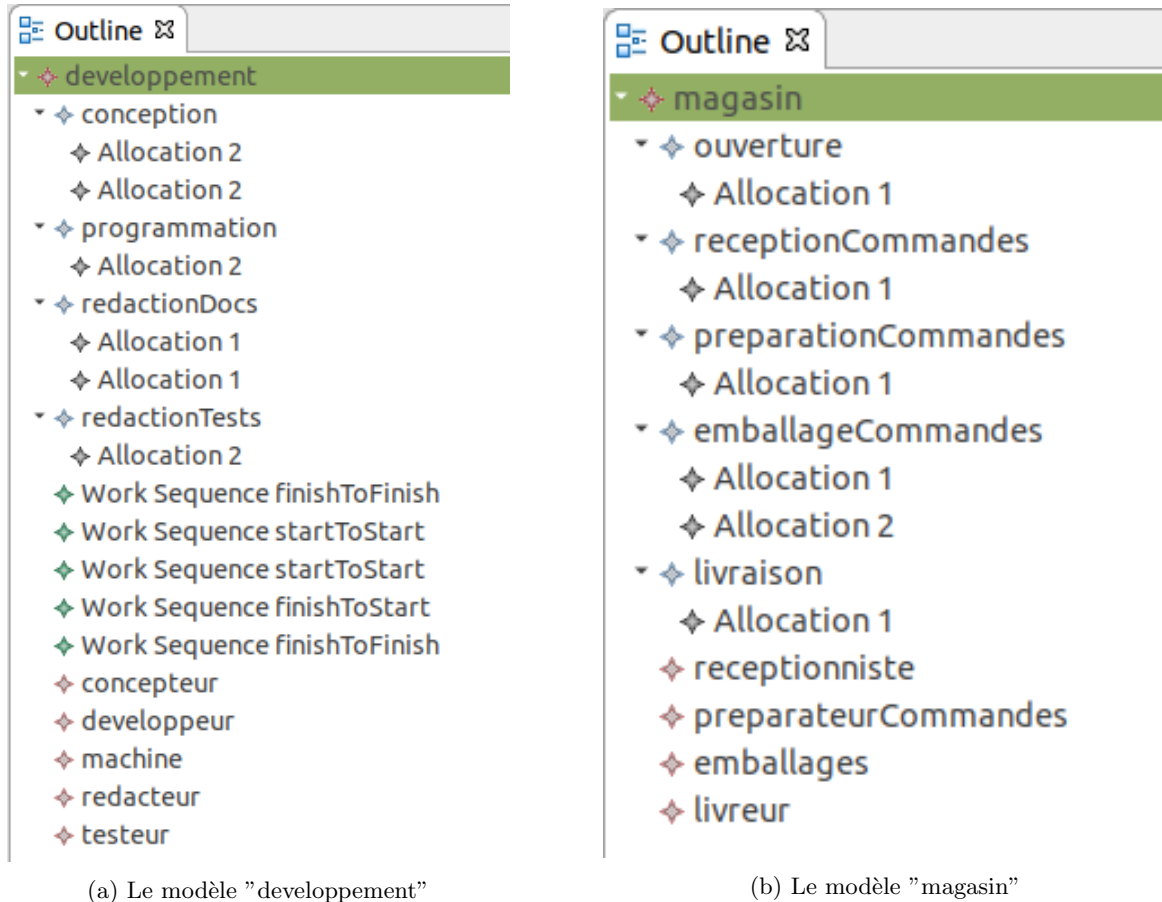


Figure 11: L'Outline lors de la rédaction des règles de grammaire de Figure 10a et 10b

7 Définition de syntaxes concrètes graphiques avec Sirius

Grâce à SIRIUS, nous avons défini une syntaxe graphique pour le langage de modélisation SimplePDL décrit en Ecore, ce qui permet d'engendrer un éditeur graphique intégré à Eclipse.

Voici la palette des outils disponibles pour la création d'un modèle graphique de SimplePDL.

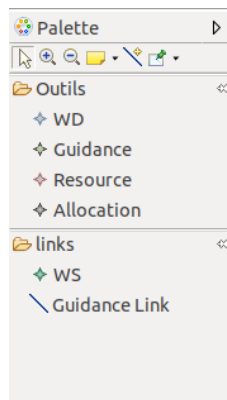


Figure 12: Palette des outils pour les modèles SimplePDL

A partir de cet éditeur graphique, on peut non seulement créer des modèles de A à Z, mais aussi de visualiser des modèles déjà définis. La figure 13 montre la visualisation des modèles 'magasin' et 'developpement'.

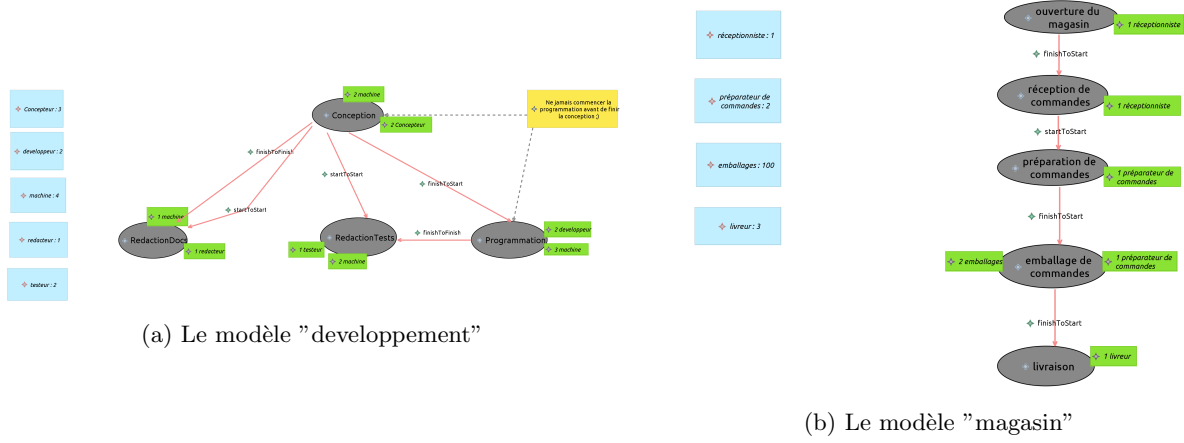


Figure 13: Exemples de visualisation graphique des modèles de SimplePDL

8 Application: Chaîne de vérification du modèle de processus 'magasin'

Partant du modèle de procédé 'pdl-magasin.xmi', on veut vérifier si ce procédé est réalisable ou pas. Ceci est faisable en vérifiant si le processus se termine ou pas en utilisant le logiciel Tina. Par contre, ce dernier a besoin d'un fichier .net d'un réseau de Petri en entrée.

Ainsi, en utilisant la transformation M2M (cf Section 5, livrables: SimplePDLToPetriNet.java ou SimplePDLToPetriNet.atl), on transforme le modèle de procédé 'pdl-magasin.xmi' en modèle de PetriNet 'net-magasin.xmi'.

Puis, on fait la transformation M2T (cf section 4, livrables : toTina.mtl et toLTL.mtl) pour engendrer les fichiers 'magasin.net' à partir de net-magasin.xmi et 'magasin.ltl' à partir de pdl-magasin.xmi.

En utilisant NetDraw de TINA, on visualise le réseau de Petri correspondant au procédé magasin(Figure 14).

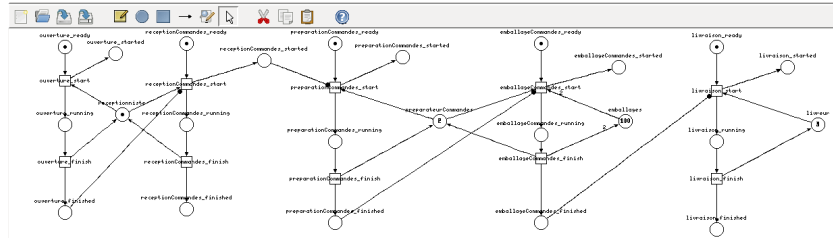


Figure 14: Visualisation sur TINA de magasin.net

Et finalement, on peut vérifier la propriété de terminaison en utilisant le model-checking avec selt qui prend en entrée le fichier magasin.ltl. On voit bien que la propriété 'finished' est true.

```
clouahab@mystique:~/26/IDM/L10/testss$ selt -p -s magasin.sch magasin.ktz -prelude magasin.ltl
Selt version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 17 states, 22 transitions
0.003s

- source magasin.ltl;
operator finished : prop
operator ouverture_state : prop
operator receptionCommandes_state : prop
operator preparationCommandes_state : prop
operator emballageCommandes_state : prop
operator livraison_state : prop
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
FALSE
```

Figure 15: Résultat de model-checking de magasin.ltl avec selt

9 Conclusion

Ce projet nous a permis d'intervenir sur l'ensemble de la chaîne de vérification des modèles de processus SimplePdl en passant par les modèles PetriNet. A savoir, la définition des deux métamodèles avec Ecore, la complétion de leurs propriétés structurelles par des contraintes (invariants, pré-, post-conditions) spécifiées à l'aide du langage d'expression de propriétés OCL, la transformation M2M de deux manières différentes (ATL et Java) et finalement la transformation M2T pour pouvoir profiter de la boîte à outils TINA qui permet de savoir si le processus décrit est réalisable ou pas.

Nous avons aussi pu définir des syntaxes concrètes textuelles (XText) et graphiques (Sirius) pour les modèles Simplepdl. Ce qui montre que l'IDM favorise bien la définition de langages dédiés, ou DSL (Domain Specific Language), qui ont l'avantage de permettre aux utilisateurs de se concentrer sur leur métier en manipulant un formalisme spécifique à leur activité.

10 Références

- [1] Benoît Combemale-Xavier Crégut-Bernard Berthomieu-François Vernadat. SIMPLEPDL2TINA: Mise en oeuvre d'une Validation de Modèles de Processus.
- [2] Benoît Combemale. Approche de métamodélisation pour la simulation et la vérification de modèle—Application à l'ingénierie des procédés. Génie logiciel [*cs.SE*]. Institut National Polytechnique de Toulouse - INPT, 2008. Français. tel-00321863
- [3]- <https://johncarloshaez.wordpress.com/2012/12/20/petri-net-programming-part-2/>