



Chaîne de vérification de modèles de processus

Youssef Sraidi, Philippe Wu

Génie du Logiciel et des Systèmes

Mini-Projet

Departement Sciences du Numerique - Deuxième année
2022-2023

Contents

1	Introduction	4
2	Métamodèles SimplePDL et PetriNet	5
2.1	SimplePDL	5
2.2	Réseau de Petri	7
3	Les contraintes OCL	8
3.1	SimplePDL	8
3.2	PetriNet	9
3.3	Test des contraintes OCL	9
4	Syntaxe concrète textuelle de SimplePDL avec Xtext	10
5	Syntaxe concrète graphique de SimplePDL avec Sirius	11
5.1	Définition de la partie graphique de l'éditeur	11
5.2	Définition de la palette	12
6	Transformation SimplePDL2PetriNet avec EMF/Java	13
7	La transformation SIMPLEPDL2PETRINET avec ATL	14
8	La transformation PETRINET2TINA avec Acceleo	16
9	La transformation SIMPLEPDL2LTL avec Acceleo	17
10	Conclusion	19
11	Livrables	20

List of Figures

1	Schéma général des transformations	4
2	Exemple de modèle de procédé	4
3	Méta-modèle de SimplePDL	5
4	SimplePDL.ecore - Méta-modèle de SimplePDL avec ressources	6
5	PetriNet.ecore - Méta-modèle des réseaux de Petri	7
6	Editeur graphique pour SimplePDL (representations.aird)	11
7	Palette de l'éditeur graphique pour SimplePDL	12
8	Schéma de traduction de SimplePDL en PetriNet	13
9	Modèle en entrée : SimplePDL2PetriNet-ATL-IN.xmi	14
10	Modèle en sortie : SimplePDL2PetriNet-ATL-OUT.xmi	15
11	Modèle en sortie : SimplePDL2PetriNet-ATL-OUT.xmi	16
12	TinaDeveloppementRes.ltl	17
13	Tableau des résultats	18
14	Exécution qui termine dans le cas du modèle avec les ressources	18
15	Fichiers contenus l'archive livrable.zip	20

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 allons utiliser les outils de model-checking définis sur les réseaux de Petri au travers de la boîte à outils Tina. Il nous faudra donc réaliser d'une part une transformation du modèle de processus en un réseau de Petri pour ensuite le convertir au format tina .net, et d'autre part engendrer les propriétés LTL à partir du modèle de processus.

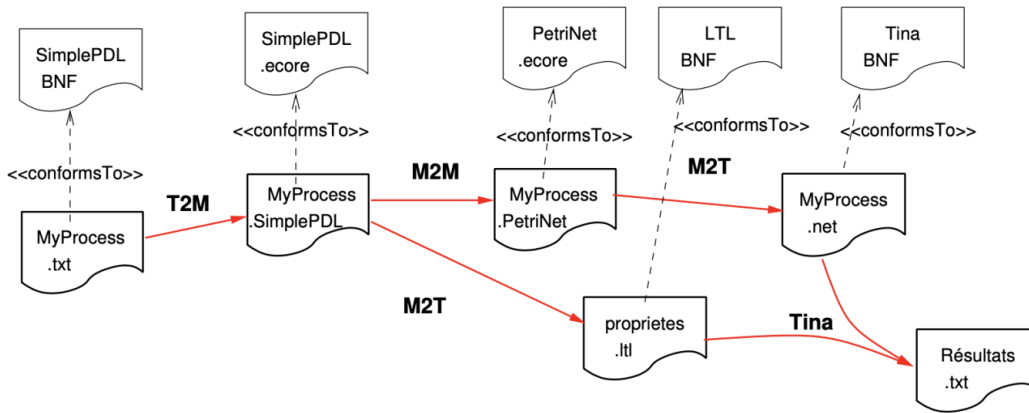


Figure 1: Schéma général des transformations

Il est très important de détecter les erreurs éventuelles dans un projet le plus tôt possible avant d'entamer ses phases les plus profondes, c'est pour cela qu'on peut considérer la méta modélisation comme une étape importante pour la validation des modèles décrivant le projet.

Dans le cadre du mini-projet, nous nous fixons comme objectif de tester notre chaîne de vérification avec le modèle de procédé (avec et sans ressources) suivant :

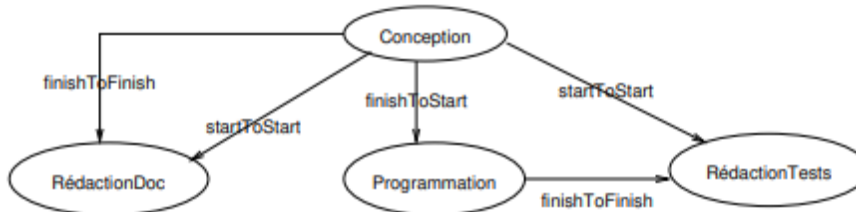


Figure 2: Exemple de modèle de procédé

2 Métamodèles SimplePDL et PetriNet

2.1 SimplePDL

SimplePDL est un langage simplifié de modélisation de processus de développement. Son métamodèle Ecore est présenté à la Figure 3 :

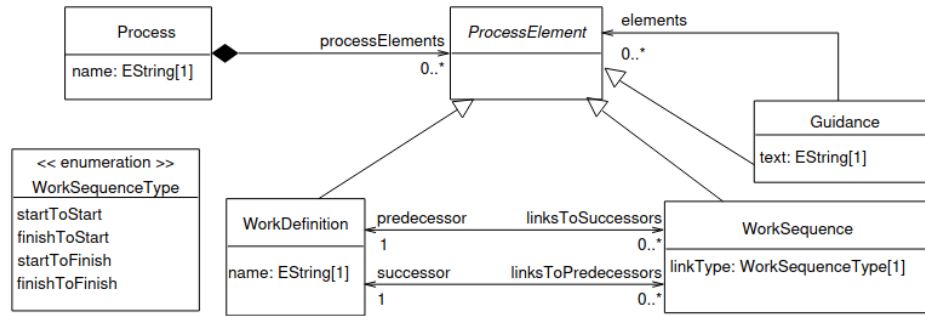


Figure 3: Méta-modèle de SimplePDL

Dans le cadre du projet, nous nous sommes permis de le compléter en ajoutant d'une part une EClass **ProcessElement** comme généralisation de **WorkDefinition** (activité) et **WorkSequence** (dépendance) et d'autre part la notion des ressources et leur utilisation par les activités.

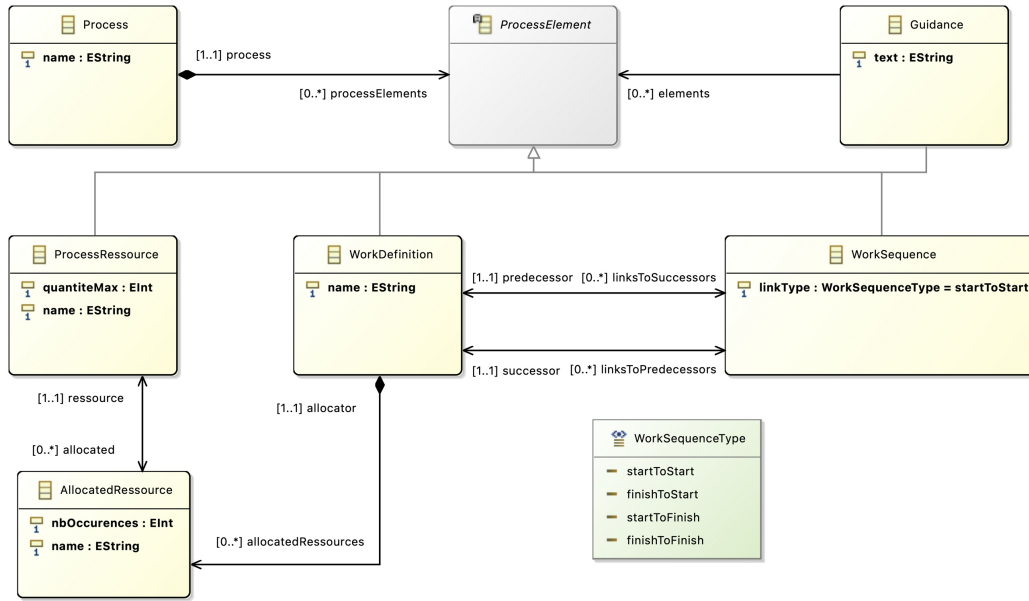


Figure 4: SimplePDL.ecore - Méta-modèle de SimplePDL avec ressources

On retrouve donc la EClass **Process** dont ses **ProcessElements** sont composés soit d'activités (**WorkDefinition**), soit de dépendances (**WorkSequence**) ou de ressources (**ProcessResource**).

Un **ProcessResource** est caractérisé par un nom (l'attribut **name**) et sa quantité maximale disponible (l'attribut **QuantiteMax**). La réalisation d'une activité peut nécessiter plusieurs ou aucune ressources. Cette dépendance est matérialisée par une EReference **allocatedResources** vers la EClass **AllocatedResource** comme le montre la figure 4. En effet cette classe représente la ressource qui est demandée par une activité avec un nombre d'occurrence (attribut **nbOccurrences**). La relation entre une activité et la ressource allouée est donc Opposite. La ressource allouée (la partie) n'a pas de sens sans l'activité (le tout) qui la demande, d'où la relation de composition.

De la même manière, la ressource (**ProcessResource**) et la ressource allouée (**AllocatedResource**) sont en relation opposée. Une ressource allouée pointe vers une unique ressource (EReference **resource**) tandis qu'une ressource peut être allouée à plusieurs activités (EReference **allocated**).

2.2 Réseau de Petri

Le réseau de Petri est un modèle mathématique qui permet de décrire le comportement dynamiques des systèmes aux éléments discrets. Il se représente par un graphé orienté composé d'arcs reliant des places et des transitions. Son métamodèle PetriNet est donné ci-dessous :

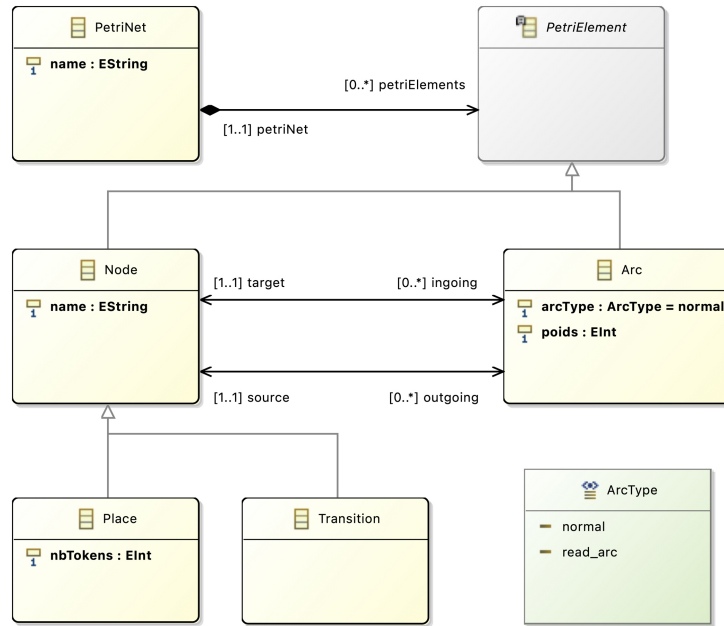


Figure 5: PetriNet.ecore - Méta-modèle des réseaux de Petri

Notre métamodèle **PetriNet** se compose d'arcs (**Arc**) reliant des noeuds (**Node**) qui sont soit des places (**Place**) ou des transitions (**Transition**).

Les places contiennent un nombre initial de jetons (attribut `nbToken`). les arcs peuvent être de type normal (qui consomme/produit les jetons) ou `read_arc` (qui lit seulement les jetons). Les arcs possèdent aussi un poids en jetons (l'attribut `poids`) représentant leur consommation (ou production) de jetons lors de l'exécution d'une transition.

Un noeud peut posséder un ou plusieurs arc entrants (*EReference ingoing*) et sortants (*EReference outgoing*) tandis qu'un arc se caractérise par une seule source (*EReference source*) et une seule cible (*EReference target*).

3 Les contraintes OCL

Nous avons utilisé Ecore pour définir un méta-modèle pour les processus et les réseaux de Petri. Cependant, le langage de méta-modélisation Ecore ne permet pas d'exprimer toutes les contraintes que doivent respecter nos modèles. Ainsi, on complète la sémantique statique du méta-modèle réalisée en Ecore par des contraintes exprimées en OCL.

Au niveau de l'implémentation des contraintes, nous avons eu le choix d'utiliser OCL ou OCLinEcore. L'approche OCL définit les contraintes à côté du méta-modèle, ce qui impose de devoir charger le fichier .ocl sur le modèle. A l'opposé, l'approche OCLinEcore permet de mettre directement les contraintes OCL sur le méta-modèle.

Dans notre cas, nous avons décidé d'opter pour OCLinEcore pour deux raisons : sur l'IDE Eclipse, OCL présente des dysfonctionnements lors du chargement du fichier .ocl et l'option de 'Live Validation' d'Eclipse avec OCLinEcore est beaucoup plus pratique.

3.1 SimplePDL

Plusieurs règles peuvent être définies sur le métamodèle SimplePDL (cf. SimplePDL.oclincore). Par exemple nous avons défini des contraintes sur la validité des noms du processus et de ses composants, la non-reflexivité des dépendances, la disponibilité des ressources demandées par les activités, etc.

- **Process** : Le nom d'un Process doit être non nul, composé de chiffres/lettres, commencé par une lettre ou `_` et la taille doit être supérieure strictement à 1. (valable aussi pour ProcessElement)

- **ProcessElement** : Deux activités différentes d'un même processus ne peuvent pas avoir le même nom

- **WorkSequence** : Une dépendance ne peut pas être réflexive.

- **ProcessResource** : Une ressource doit avoir une quantité positive.

- **AllocatedResource** : Une activité ne peut pas demander un nombre d'occurrences d'une ressource supérieur à sa quantité max.

3.2 PetriNet

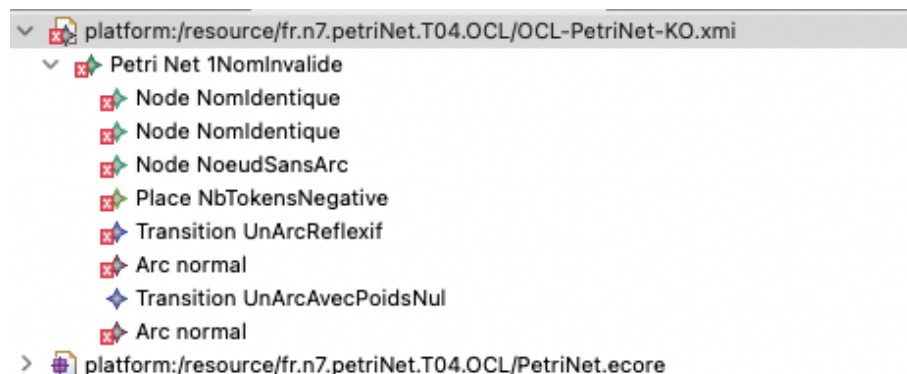
Dans le cas du métamodèle PetriNet, nous avons défini des contraintes : syntaxiques pour les éléments du réseau, algébriques sur les valeurs des poids/jetons et relationnelles entre les noeuds et les arcs. (cf. PetriNet.oclincore)

- **PetriNet** : Le nom d'un PetriNet doit être non nul, composé de chiffres/lettres, commencé par une lettre ou `_` et la taille doit être supérieure strictement à 1. (valable aussi pour Node).
- **Node** : Deux noeuds ne peuvent pas avoir le même nom. Un Noeud ne peut pas être isolé (contenir au moins un arc)
- **Place** : Le nombre de jetons d'une place doit être positive.
- **Arc** : Un arc ne doit pas être réflexif, avoir un poids strictement positif. La source et la cible d'un arc ne doit pas être du même type Transition ou Place.

3.3 Test des contraintes OCL

Pour tester nos contraintes OCL, nous avons créé pour chaque métamodèle un modèle qui respecte les contraintes OCL (fichiers terminant par OK) et un autre qui ne les respecte pas (fichiers terminant par KO) : OCL-PetriNet-KO.xmi, OCL-SimplePDL-KO.xmi, OCL-PetriNet-OK.xmi et OCL-SimplePDL-OK.xmi.

On observe dans les modèles terminant par KO que les contraintes enfreintes sont signalés au niveau de l'éditeur



4 Syntaxe concrète textuelle de SimplePDL avec Xtext

La syntaxe abstraite d'un DSML (exprimée en Ecore ou un autre langage de métamodélisation) ne peut pas être manipulée directement. Le projet EMF d'Eclipse permet d'engendrer un éditeur arborescent et les classes Java pour manipuler un modèle conforme à un métamodèle.

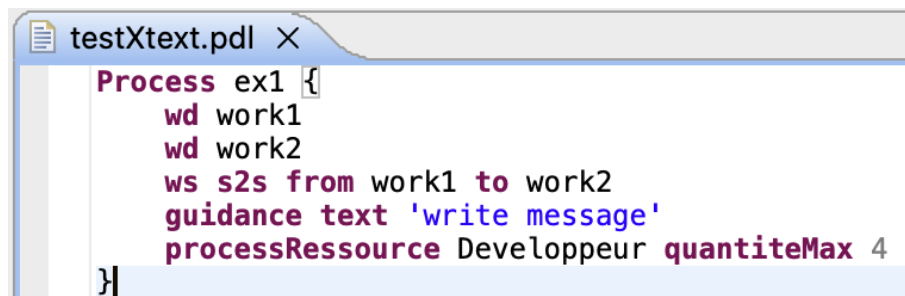
Cependant, ce ne sont pas des outils très pratiques lorsque l'on veut saisir un modèle, d'où l'intérêt d'une syntaxe concrète permettant la construction ou la modification facile et accessible pour les modèles.

Pour la syntaxe concrète textuelle nous avons utilisé l'outil Xtext qui permet de disposer, au travers d'Eclipse, d'un environnement de développement pour ce langage, avec en particulier un éditeur syntaxique (coloration, détection et visualisation des erreurs, etc).

Dans le cadre du mini-projet, nous avons défini une syntaxe textuelle (cf. [PDL.xtext](#)) dans laquelle un processus est déclaré avec le mot-clé `Process` suivi de son nom. Les éléments du processus sont décrits au sein d'un bloc entre accolades sous la forme d'une pseudo liste.

Chaque élément du processus est identifié par un mot-clé qui lui est propre :

- `wd` pour une `WorkDefinition`, suivi de son nom
- `ws` pour une `WorkSequence`, suivi de son `linkType` en abrégé puis son prédécesseur et son successeur
- `guidance text` pour une `Guidance`, suivi du commentaire en `String`
- `processRessource` pour une ressource, suivi de son nom, puis le mot-clé `quantiteMax` suivi de sa valeur



```
Process ex1 {  
    wd work1  
    wd work2  
    ws s2s from work1 to work2  
    guidance text 'write message'  
    processRessource Developpeur quantiteMax 4  
}
```

On remarque une coloration pour les mots-clés que l'on a définis dans notre grammaire (cf. [TestXtext.pdl](#)) ce qui signifie qu'elle est bien reconnue. On a également aucune erreur syntaxique signalée, ce qui signifie que notre syntaxe est bien formée.

5 Syntaxe concrète graphique de SimplePDL avec Sirius

À l'instar d'une syntaxe concrète textuelle, une syntaxe concrète graphique fournit un moyen de visualiser et/ou éditer plus agréablement et efficacement un modèle. Nous allons utiliser l'outil Eclipse Sirius qui permet de définir une syntaxe graphique pour un langage de modélisation décrit en Ecore et d'engendrer un éditeur graphique intégré à Eclipse.

5.1 Définition de la partie graphique de l'éditeur

Afin d'afficher sur l'éditeur les différents éléments de nos modèles, nous devons définir leur représentation graphique. Nous avons opté pour le choix suivant (cf. [simplepdl.odesign](#)) :

- EClass WorkDefinition représentée par un rectangle à fond gris foncé et bordure grasse
- EClass ProcessRessource représenté par une ellipse à fond gris clair et bordure grasse
- EClass Guidance représentée par un rectangle à fond jaune et bordure légère
- EClass AllocatedRessource représentée par une flèche à trait espacé et affiche le nombre d'occurrences demandé
- EClass WorkSequence représentée par une flèche colorée à trait plein en fonction de son type et l'affiche.
- EReference elements de Guidance par une flèche pointillée

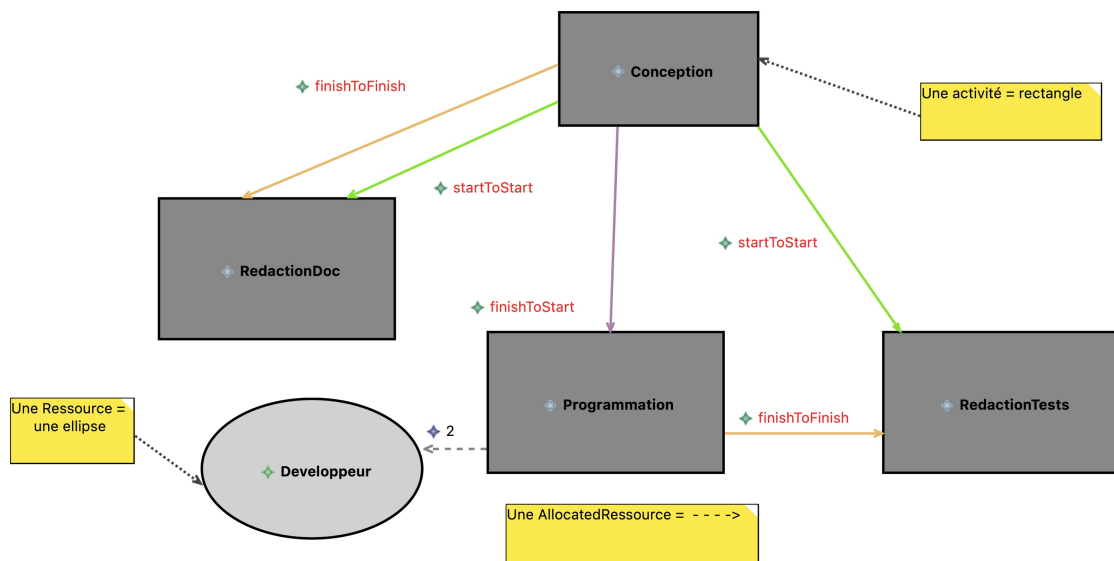


Figure 6: Editeur graphique pour SimplePDL (representations.aird)

5.2 Définition de la palette

Tous les éléments graphiques que nous avons décrits précédemment peuvent être édités grâce à la palette (Figure 7).

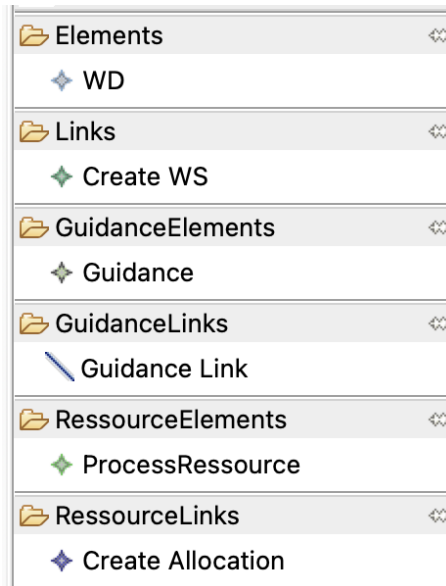


Figure 7: Palette de l'éditeur graphique pour SimplePDL

- WD pour créer une WorkDefinition
- Create WS pour créer une WorkSequence
- Guidance pour créer une Guidance
- Guidance Link pour créer un lien entre une Guidance et un processElement
- ProcessRessource pour créer un ProcessRessource
- Create Allocation pour créer une AllocatedRessource entre une WorkDefinition et un ProcessRessource

6 Transformation SimplePDL2PetriNet avec EMF/Java

EMF permet la génération du code java pour SimplePDL et PetriNet une fois qu'on dispose des fichiers ecore grâce au genmodel. Ensuite, en se basant sur les packages engendrés, on peut écrire un code Java qui transforme un modèle de processus en un modèle de réseau de Pétri. La transformation suit la modélisation suivante : (cf. [SimplePDL2PetriNet.java](#))

- **WorkDefinition** traduite en :
 - 4 places : PlaceReady, PlaceStarted, PlaceRunning et PlaceFinished
 - 2 transitions : TransitionStart, TransitionFinish
 - 5 arcs: ArcReadyToStart, ArcStartToStarted, ArcStartToRunning, ArcRunningToFinish et ArcFinishToFinished
- **WorkSequence** modélisée par un read_arc reliant les places PlaceStarted ou PlaceFinished avec les transitions TransitionStart ou TransitionFinish selon le linkType
- **ProcessRessource** correspond à une place avec comme marquage initial le quantité max disponible
- **AllocatedRessource** représentée par un arc sortant du ProcessRessource (allocation) et un arc entrant vers le ProcessRessource (récupération de la ressource allouée)

La figure ci-dessous illustre la transformation pour une WorkDefinition et une WorkSequence.

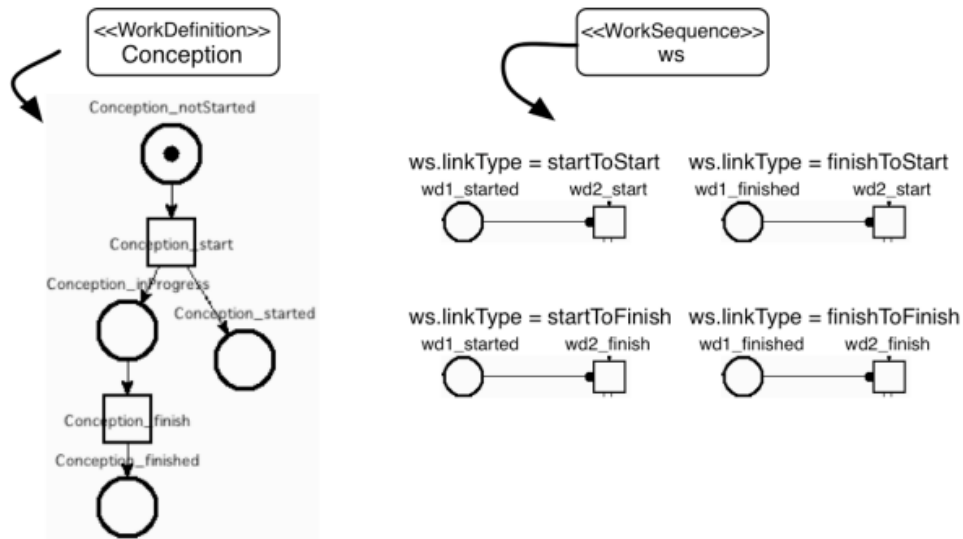


Figure 8: Schéma de traduction de SimplePDL en PetriNet

7 La transformation SIMPLEPDL2PETRINET avec ATL

Notre transformation modèle à modèle en ATL consiste comme pour la transformation Java à transformer un modèle de processus en un modèle de réseau de Pétri (cf. [SimplePDL2PetriNet.atl](#)). Cette transformation passe par les mêmes étapes que la transformation Java.

Pour tester notre transformation ATL (et JAVA), nous le testons avec un modèle simple (Figure 9) pour vérifier que chaque élément d'un process est bien transformé en son élément équivalent au sein du réseau de Petri.

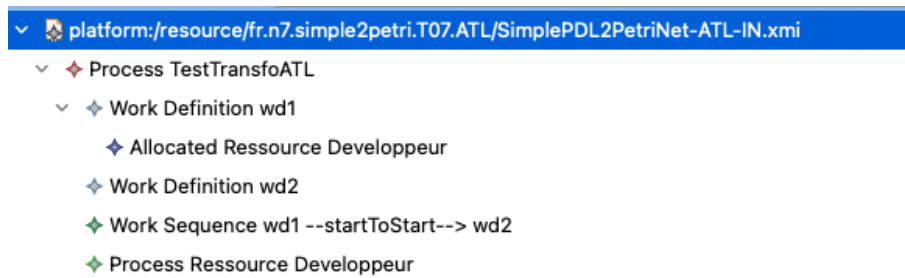


Figure 9: Modèle en entrée : SimplePDL2PetriNet-ATL-IN.xmi

Comme on peut le voir sur la Figure 10, la transformation s'est effectuée correctement selon la modélisation expliquée dans la partie précédente. On obtient le même résultat avec la transformation en Java. Ceci nous permet de valider nos transformations ATL et JAVA.



Figure 10: Modèle en sortie : SimplePDL2PetriNet-ATL-OUT.xmi

8 La transformation PETRINET2TINA avec Acceleo

Afin de pouvoir manipuler le réseau de Petri dans l'outil TINA, nous devons réaliser la transformation du réseau de Petri en la syntaxe concrète textuelle de TINA (cf. [petri2tina.mtl](#))

Comme pour les transformations précédentes, nous avons réalisé un modèle simple de test pour vérifier que la transformation est valide, comme on peut le voir sur la Figure 11

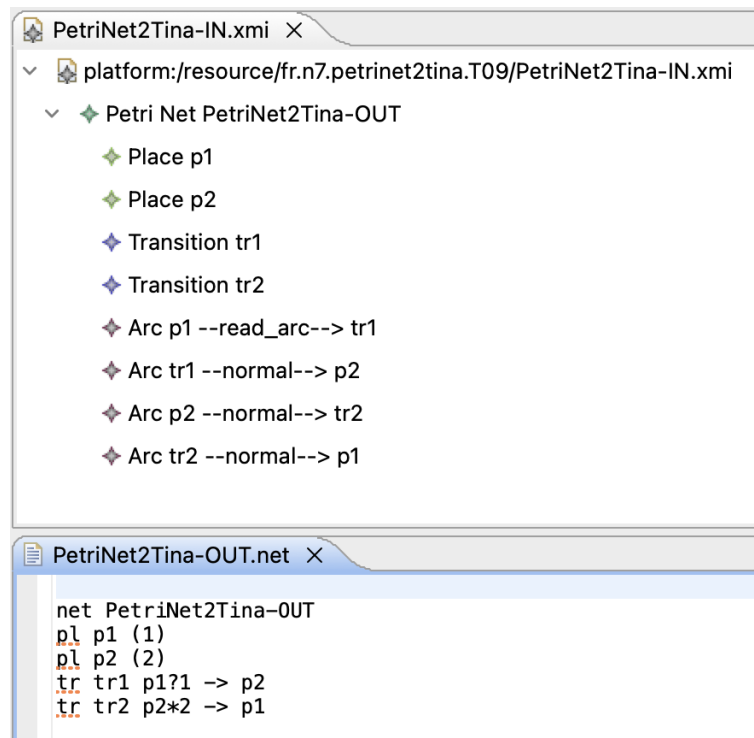


Figure 11: Modèle en sortie : SimplePDL2PetriNet-ATL-OUT.xmi

9 La transformation SIMPLEPDL2LTL avec Acceleo

Après avoir réalisé la transformation du modèle de processus en la syntaxe concrète textuelle de TINA, il nous reste plus qu'à engendrer les propriétés LTL à partir du modèle de processus, avec Acceleo (cf. `simple2LTL.mtl`). Un des objectifs du mini-projet est de savoir si un processus peut se terminer ou non. Pour cela, nous allons générer un fichier LTL pour tester le modèle de processus sur TINA avec l'outil selt.

Voici les propriétés que l'on a vérifiées sur notre modèle de procédé :

(1) $\square (\text{wd1_started} \vee \text{wd1_ready}) \wedge (\text{wd2_started} \vee \text{wd2_ready})$ (Invariant)

$\text{op finished} = \text{wd1_finished} \wedge \text{wd2_finished}$

(2) $\square (\text{dead} \Rightarrow \text{finished})$: Tout procédé terminé est dans son état final.

(3) $\square (\text{finished} \Rightarrow \text{dead})$: Tout procédé, qui est dans son état final, est terminé.

(4) $\square \neg \text{dead}$: Toute exécution se termine.

(5) $\neg \neg \text{finished}$: Aucune exécution ne termine

La propriété (5) permet de s'assurer de l'existence d'au moins une exécution qui termine. Avoir en retour **False** permet grâce à l'outil LTL d'avoir un chemin à suivre pour avoir une terminaison.

L'invariant est défini par la propriété (1). En effet, toutes les activités sont soit commencées soit non commencées.

```
# Propriété 1 - Invariant : chaque activité est soit non commencée soit en cours
[] (Conception_ready \vee Conception_started) /\ (RedactionDoc_ready \vee RedactionDoc_started) /\ (Programmation_ready \vee Programmation_started) /\
(RedactionTests_ready \vee RedactionTests_started);

# Vérification de la terminaison
op finished = Conception_finished /\ RedactionDoc_finished /\ Programmation_finished /\ RedactionTests_finished;
[] (dead => finished); # Propriété 2 : Si le reseau est dead alors toutes les activités sont en état finished
[] (finished => dead); # Propriété 3 : Si toutes les activités sont en état finished alors le reseau est dead
[] <=> dead; # Propriété 4 : Pour toute exécution, le reseau finit par être dead.
- <=> finished; # Propriété 5 : Il n'existe pas d'exécution pour lequel toutes les activités sont en état finished
```

Figure 12: TinaDeveloppementRes.ltl

Pour vérifier du bon fonctionnement de nos propriétés, nous les avons testées avec deux modèles simples où l'un termine et l'autre ne termine pas (cf. TinaDeadFinishedPetri.xmi et TinaDeadNonFinishedPetri.xmi)

Après avoir validé les propriétés LTL que nous avons engendrées, nous nous sommes intéressés à un modèle de procédé complet, celui donné au début du sujet Figure 2. Nous avons testé la terminaison dans le cas où le modèle comporte ou non des ressources.

Ensuite nous avons réalisé un tableau comparatif des résultats des propriétés LTL (Figure 13. On peut observer que dans les deux cas, il existe une exécution qui se termine. Cependant on se rend compte que dans le cas du modèle avec les ressources, il existe une exécution pour laquelle l'exécution se termine sans que le procédé se retrouve dans son état final. Les résultats TINA ont été stockés dans le fichier TinaResults.txt.

	TinaDeadFinished	TinaDeadNonFinished	TinaDeveloppementSansRes	TinaDeveloppementRes
Prop 1 : Invariant	VRAI	VRAI	VRAI	VRAI
Prop 2 : \square (dead \Rightarrow finished)	VRAI	FAUX	VRAI	FAUX
Prop 3 : \square (finished \Rightarrow dead)	VRAI	VRAI	VRAI	VRAI
Prop 4 : $\square \Leftrightarrow$ dead	VRAI	VRAI	VRAI	VRAI
Prop 5 : $\neg \Leftrightarrow$ finished	FAUX	VRAI	FAUX	FAUX

Figure 13: Tableau des résultats

```

TRUE
FALSE
state 0: Conception_ready Programmation_ready RedactionDoc_ready RedactionTests_ready concepteur*3 developpeur*2 machine*4 redacteur testeur*2
-Conception_start->
state 1: Conception_running Conception_started Programmation_ready RedactionDoc_ready RedactionTests_ready concepteur developpeur*2 machine*2
redacteur testeur*2
-Conception_finish->
state 2: Conception_finished Conception_started Programmation_ready RedactionDoc_ready RedactionTests_ready concepteur*3 developpeur*2
machine*4 redacteur testeur*2
-Programmation_start->
state 3: Conception_finished Conception_started Programmation_running Programmation_started RedactionDoc_ready RedactionTests_ready
concepteur*3 machine redacteur testeur*2
-Programmation_finish->
state 4: Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_ready RedactionTests_ready
concepteur*3 developpeur*2 machine*4 redacteur testeur*2
-RedactionDoc_start->
state 5: Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_running RedactionDoc_started
RedactionTests_ready concepteur*3 developpeur*2 machine*3 testeur*2
-RedactionDoc_finish->
state 6: Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_finished RedactionDoc_started
RedactionTests_ready concepteur*3 developpeur*2 machine*4 redacteur testeur*2
-RedactionTests_start->
state 7: Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_finished RedactionDoc_started
RedactionTests_running RedactionTests_started concepteur*3 developpeur*2 machine*2 redacteur testeur
-RedactionTests_finish->
state 8: L.dead Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_finished RedactionDoc_started
RedactionTests_finished RedactionTests_started concepteur*3 developpeur*2 machine*4 redacteur testeur*2
-L.deadlock->
state 9: L.dead Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_finished RedactionDoc_started
RedactionTests finished RedactionTests started concepteur*3 developpeur*2 machine*4 redacteur testeur*2

```

Figure 14: Exécution qui termine dans le cas du modèle avec les ressources

10 Conclusion

Ce projet nous a permis de mettre en pratique les notions vu en cours/TP dans le cadre de la réalisation d'une chaîne de vérification de modèles de processus. En effet, nous avons pu traduire nos modèles SimplePDL en réseaux de Petri, générer les propriétés LTL pour ensuite utiliser TINA pour étudier la terminaison des modèles de processus.

Ceci nous a montré l'importance de la modélisation dans la résolution de problèmes et la maîtrise du processus de développement logiciel basé sur des méthodologies de modélisation, de conception et de développement.

En guise de fin, nous regrettons malheureusement le nombre conséquent de bug rencontré lors de l'utilisation d'Eclipse, qui nous a fait perdre beaucoup de temps, temps auquel nous aurions aimé consacrer davantage à l'approfondissement du projet et à rendre un travail encore plus qualitatif.

11 Livrables

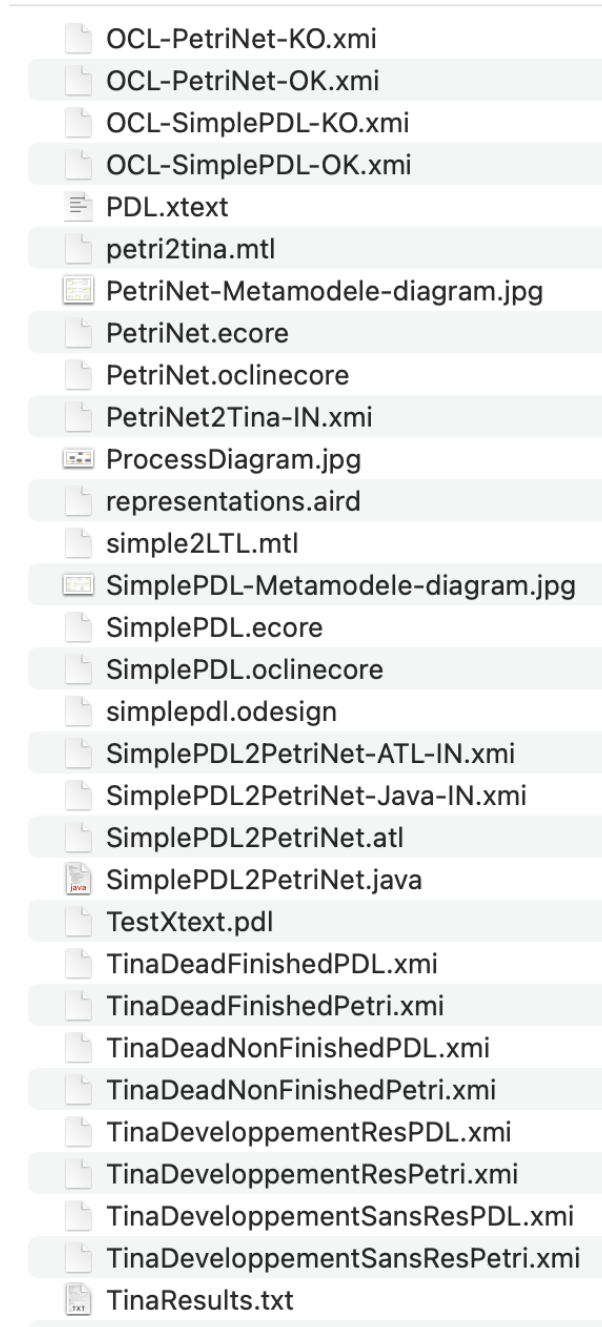


Figure 15: Fichiers contenus l'archive livrable.zip