

# **Rapport mini-projet**

## **Ingénierie Dirigée par les**

### **Modèles**



Diego Rodriguez  
Thomas Nadal Elizalde  
SN2APP

<b>Introduction</b>	<b>3</b>
<b>Point d'entrée des documents</b>	<b>4</b>
<b>I - Les métamodèles : SimplePDL et PetriNet</b>	<b>5</b>
SimplePDL	5
PetriNet	6
<b>II - Les contraintes OCL</b>	<b>7</b>
SimplePDL	7
PetriNet	8
<b>III - JAVA EMF - Transformation modèle à modèle</b>	<b>9</b>
<b>IV - ATL - Transformation modèle à modèle</b>	<b>12</b>
<b>V - ACCELEO - Transformation de modèle à texte</b>	<b>13</b>
ToTina	13
ToLTL	13
<b>VI - SIRIUS - Définition d'une syntaxe graphique</b>	<b>15</b>
<b>VII - Xtext - Définition d'une syntaxe textuelle</b>	<b>17</b>
<b>Conclusion</b>	<b>18</b>

# Introduction

L'objectif du mini-projet d'IDM est d'appliquer les méthodes apprises en TP afin de l'appliquer sur un métamodèle SimplePDL amélioré dans lequel nous avons ajouté les ressources. L'idée est ensuite de créer plusieurs modèles issues de ce modèle afin d'y appliquer des transformations avec les outils que nous avons utilisés en TP comme Acceleo, Xtext, ATL et d'autres encore que nous verrons dans la suite du rapport et que vous pouvez retrouver listés dans le sommaire. Nous pouvons aussi manipuler graphiquement le modèle grâce à un outil comme Sirius. Nous avons utilisé la boîte à outils Tina et les validations de model checking afin de vérifier la cohérence des méta-modèles et modèles utilisés. C'est pour cela qu'au cours du projet, nous avons eu l'occasion de traduire SimplePDL en Tina par exemple afin de pouvoir effectuer ces vérifications.

Ce rapport a pour objectif de présenter le travail que nous avons réalisé (Diego et Thomas) afin d'avoir un point d'entrée sur le code des documents fournis pour mieux comprendre ce que nous avons fait. Nous ne partageons pas l'archive entière du projet car il ne sera pas nécessaire de l'utiliser. En effet, nous avons eu l'occasion de faire une séance de démonstration des générations automatiques des différents outils. Cette séance a permis d'éviter les problèmes de compilation si nous transférons notre archive sur une autre machine. Nous allons donc seulement nous concentrer sur les documents que nous allons rendre qui sont associés aux tâches que nous avons à réaliser et les expliquer.

Tout au long du rapport, l'ajout des ressources et expliqué brièvement puis une redirection vers le fichier intéressant (fourni dans l'archive de ce rapport) est donnée. Tous les fichiers sont triés d'une manière logique avec des noms de dossiers (D1, D2...) équivalent à ce qui est indiqué dans le sujet du mini-projet. C'est dans ce fichier que nous avons mis les commentaires sur les ressources afin d'expliquer la logique établie. Nous avons essayé de mettre les captures d'écran qui nous paraissaient cohérente à détailler mais nous n'avons pas été exhaustif car lors de la démonstration, nous avons déjà prouvé ce qui fonctionnait et nous pensons qu'il n'est pas la peine de re-faire la démonstration dans le rapport. Étant donné que nous n'avons pas détaillé l'implémentation durant la démonstration, c'est dans ce rapport que nous le ferons.

## Point d'entrée des documents

Voici un tableau récapitulatif de la correspondance des fichiers et captures d'écrans qui sont fournis dans l'archive. Dans les parties correspondantes, on vous indiquera quel fichier regarder par rapport aux tâches effectuées. N'hésitez pas à vous référer à ce tableau pour savoir quel document regarder par rapport aux explications du rapport. Il y a d'autres documents que ceux listés dans le tableau. Vous les trouverez dans l'archive et vous pourrez aller les voir si vous avez besoin de plus de détails. Nous avons veillé à laisser des noms très parlants pour que vous n'ayez pas à chercher.

Nom du fichier	Correspondance
MetaModeleSimplePDL.jpg	Image du méta-modèle SimplePDL (D1)
MetaModelePetriNet.jpg	Image du méta-modèle PetriNet (D1)
SimplePDL.ocl	Contraintes OCL associées au méta-modèle SimplePDL (D2)
ProcessInvalidName.xmi, PetriNetContreExemple.xmi, SimplePDLContreExemple.xmi	Modèle qui ne valide pas une contrainte sur les ressources (D2)
SimplePDLToPetriNet.java	Classe java de transformation SimplePDL to PetriNet (D3)
SimplePDL2PetriNet.atl	Code ATL de transformation d'un modèle SimplePDL vers un modèle PetriNet (D4)
totina.mtl	Fichier de transformation d'un modèle PetriNet vers du texte Tina (D5)
toLTL.mtl	Fichier de transformation d'un modèle PetriNet vers de la syntaxe LTL (D5)
simplepdl.odesign	Description de la syntaxe graphique pour SimplePDL (D6)
PDL.xtext	Définition de la syntaxe textuelle pour SimplePDL (D7)

# I - Les métamodèles : SimplePDL et PetriNet

## SimplePDL

Un Process est principalement composé de **WorkDéfinitions** et de **WorkSequences**. Ici ces deux éléments héritent du concept abstrait “ProcessElement” pour mieux définir leur place dans le Process. Ils ont des liens de successeurs/prédécesseurs car comme vu en cours, un process est un enchaînement de WorkDefinition/WorkSequences.

A partir de cette notion vue en classe et du méta-modèle déjà créé, nous ajoutons la notion de **ressources**. Une WorkDefinition peut avoir besoin de ressources, on a donc un ensemble de Need par type de ressources (humain, machine, ...) à qui on associe un nombre de ressources. C'est la classe Ressource qui s'occupera de gérer le type et le nombre de ressources.

Par exemple, il peut exister 3 humains et 2 machines, on aura deux instances de Ressource: une nommée “humains” avec une quantity à 3, l'autre “machines” avec quantity à 2. Les WorkDefinitions expriment des besoins en Ressources humaines, machines, etc... en renseignant le nombre de ressources de tel ou tel type dont elles ont besoin. On a une association entre les Ressources et les Needs, ce qui veut dire que les ressources devront être partagées entre les WorkDefinitions.

Voici la version améliorée du méta-modèle SimplePDL qui prend en compte la notion de ressources. On peut y voir deux nouvelles classes. Une classe **Need** qui permet d'exprimer un besoin sur le nombre de ressources nécessaires pour accomplir une tâche et une classe **Resource** qui représente l'ensemble des ressources disponibles.

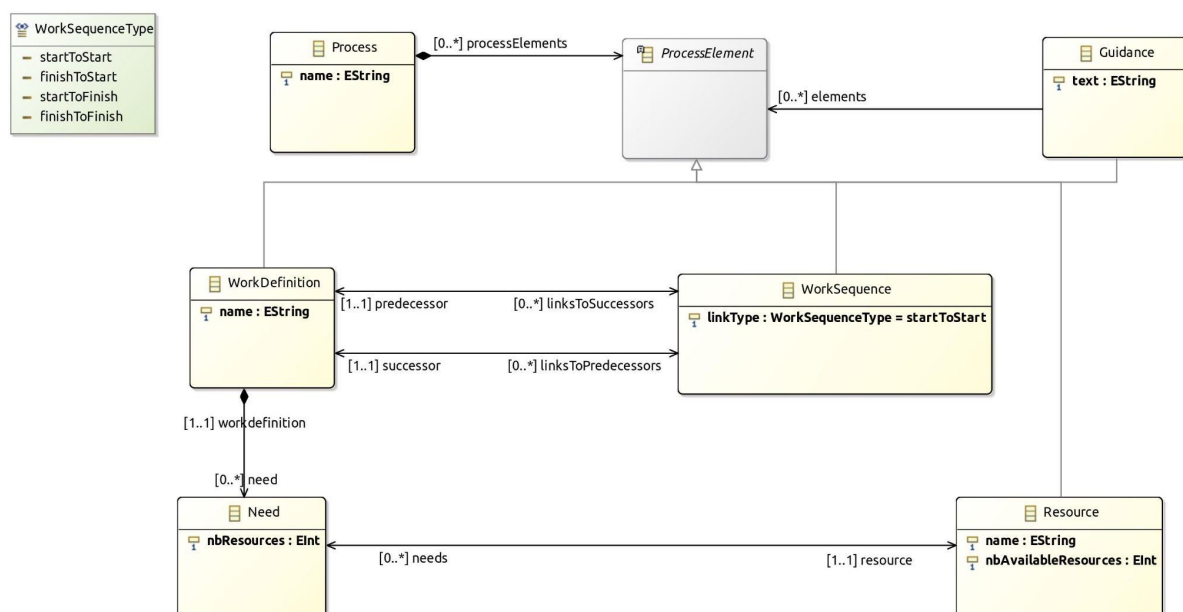


FIGURE 1 - Méta-modèle SimplePDL prenant en compte l'ajout de ressources

# PetriNet

Un réseau de pétri est composé **d'arcs**, de **transitions** et de **places**. Les transitions et les places sont considérées comme des nœuds, les classes héritent donc du concept de Node. Un arc va d'un Node source à un Node target, et une place a un nombre de jetons. Un arc peut consommer des ressources à travers l'attribut weight. S'il n'en consomme pas, l'arc est en lecture seule avec l'attribut **isReadArc** à true. Nous ne détaillons pas plus ce méta-modèle car nous l'avons défini en TP.

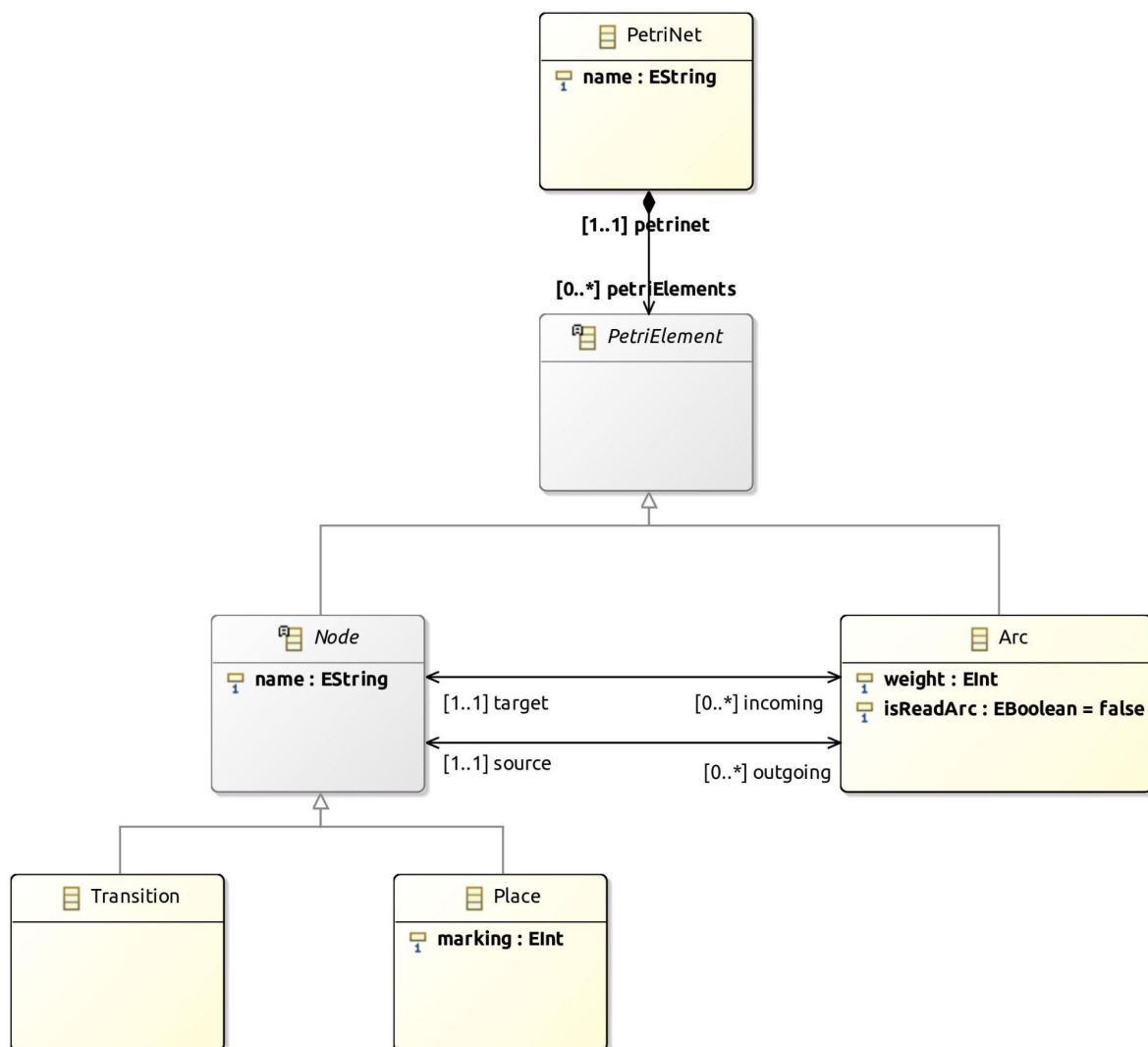


FIGURE 2 - Méta-modèle Petrinet

## II - Les contraintes OCL

OCL est un langage de requête qui permet de vérifier des contraintes sûr un modèle que l'on est pas capable de valider avec seulement le méta-modèle. On définit des propriétés d'invariants pour vérifier que les classes et méthodes les respectent.

### SimplePDL

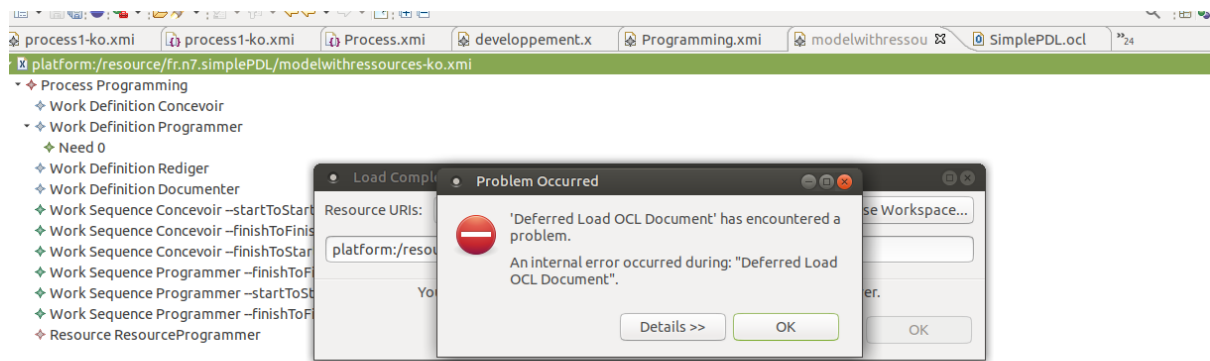
On avait déjà défini des contraintes OCL en classe pour vérifier la validité du nom d'un process, et d'autres éléments logiques comme par exemple s'assurer que les deux extrémités d'une WorkSequence soient deux WorkDefinition distinctes.

Pour le mini-projet, nous avons ajouté des contraintes sur les **besoins** et les **ressources**. Ils représentent les points suivants :

- Une ressource possède un nom correct
- La quantité de ressource est supérieure ou égale à 1
- Le besoin en ressources est entre 1 et le nombre de ressources maximum
- Il n'existe pas deux besoins rattachés à la même ressource et à la même WorkDefinition

Nous avons réfléchi à ces contraintes dans le but de vérifier des détails qui ne sont pas vérifiables avec le seul méta-modèle. Vous pouvez retrouver toutes les contraintes dans le fichier **SimplePDL.ocl**.

Afin de vérifier ces contraintes, il suffit tout simplement de créer un modèle SimplePDL qui ne respecte pas ces contraintes et de vérifier que lorsque l'on charge le fichier OCL sur le modèle correspondant, un warning est levé. On peut voir un exemple avec les fichiers \*.xmi fournis en annexe. Ce sont des fichiers qui contiennent des erreurs OCL. Par exemple, le fichier **modelwithressources-ko.xmi** qui contient une WorkDefinition ayant un besoin et l'associant à 0 ressources. C'est une chose que l'on a interdit dans nos contraintes. On aurait pu créer de nombreux autres modèles pour tester chaque contrainte OCL, mais une seule nous a suffi pour s'assurer que le fichier OCL et les contraintes sur les ressources sont bien prises en compte. Malheureusement, lors de la vérification des contraintes, il n'était pas possible de charger les fichiers .ocl à cause de cette erreur. Nous en avons discuté lors de la démonstration avec votre étudiant en thèse et il a dit que ce problème survient pour tout le monde.



*FIGURE 3 - Erreur lors du chargement des contraintes OCL*

## PetriNet

De même qu'avec SimplePDL, nous avons déjà défini des contraintes OCL sur le méta-modèle PetriNet. Ces contraintes sont par exemple vérifier que le nom d'un PetriNet est valide, que le poids d'un arc est strictement positif et d'autres encore. Nous ne les détaillons pas car cela a été fait en TP et non durant le mini-projet.



## III - JAVA EMF - Transformation modèle à modèle

L'idée ici est de faire une transformation d'un modèle vers un autre modèle avec **Java EMF**. Dans le cadre de ce mini-projet, ce sera une transformation d'un modèle SimplePDL vers un modèle PetriNet. Pour chaque élément du méta-modèle SimplePDL, nous créons un élément du méta-modèle PetriNet. Par exemple, si nous avons un élément WorkDefinition, nous créons plusieurs Places afin de faire correspondre cette notion en Petrinet avec la notion de **\_start**, de **\_finish**, de place ready, running, started, finished. Les WorkSequence seront aussi transformées en Arcs.

Nous avons dû ajouter la gestion des ressources. Pour cela, nous créons une place pour un objet ressource. De plus, pour un objet need, nous créons plusieurs arcs associées pour **\_start** et **\_finish**. Vous pouvez trouver l'implémentation et les explications commentées dans le fichier SimplePDLToPetriNet.java.

En exécutant le fichier java et en partant du modèle SimplePDL suivant :



FIGURE 4 - Modèle source SimplePDL

Nous avons obtenu le modèle PetriNet suivant :

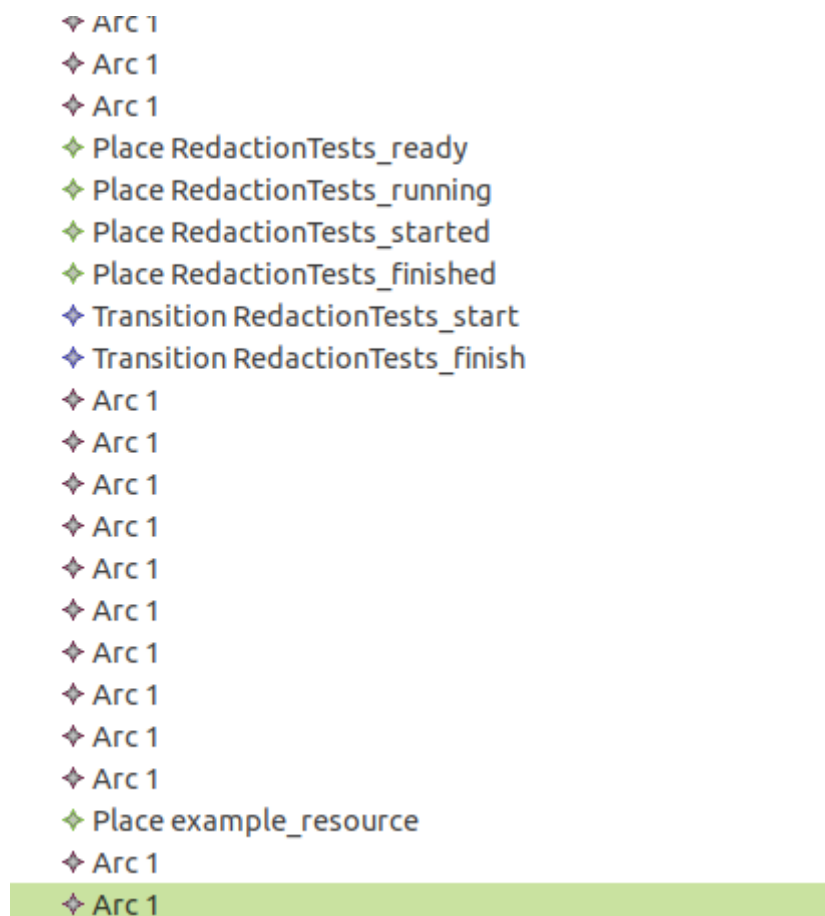
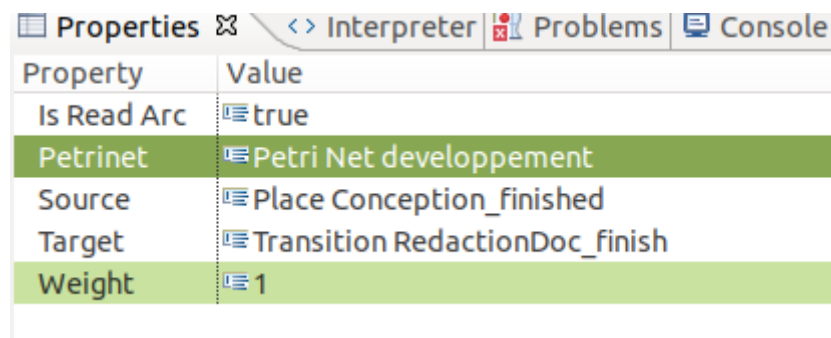


FIGURE 5 - Modèle de sortie PetriNet

Le résultat est trop grand pour tout prendre mais la capture d'écran indique au moins la ressource. On peut aussi voir que les transformations en un **arcs** start2start, start2finish (et autres...) a également été prise en compte pour les arcs.



Property	Value
Is Read Arc	true
Petrinet	Petri Net developpement
Source	Place Conception_finished
Target	Transition RedactionDoc_finish
Weight	1

*FIGURE 6 - Arc start2start*

On peut ainsi remarquer que la transformation fonctionne bien et que la prise en compte de ressources à fonctionnée dans la transformation.

Nous retenons de cette partie qu'il est possible, à partir d'un modèle, de générer le code Java des classes des éléments du modèle (et leurs tests), et qu'il est possible de les manipuler. Il est intéressant de savoir qu'il est possible de faire des transformations de modèle à modèle en Java, mais dans la partie suivante, nous utiliserons ATL qui est plus adapté pour faire cette transformation car il est plus précis et permet de plus de vérifier la cohérence du modèle.

## IV - ATL - Transformation modèle à modèle

Une deuxième manière de transformer un modèle en un autre modèle est d'utiliser atl. Dans notre cas, nous faisons encore une transformation de SimplePDL vers PetriNet, nous avons déjà fait comme tous les points précédent une transformation en TP. L'idée maintenant est d'ajouter la prise en compte des ressources, de générer la transformation et de vérifier que notre nouveau fichier **ATL** fonctionne et prend en compte les ressources.

Avec ATL, on décrit plus précisément les transformations de chaque propriété du modèle simplePDL vers le modèle PetriNet. L'idée pour les ressources est d'ajouter de nouvelles règles pour les ressources. Vous pouvez trouver le code de la transformation ATL dans le fichier **SimplePDL2PetriNet.atl**. Sur la capture d'écran ci-dessous, vous pouvez voir le résultat de la transformation qui prend bien en compte les ressources.

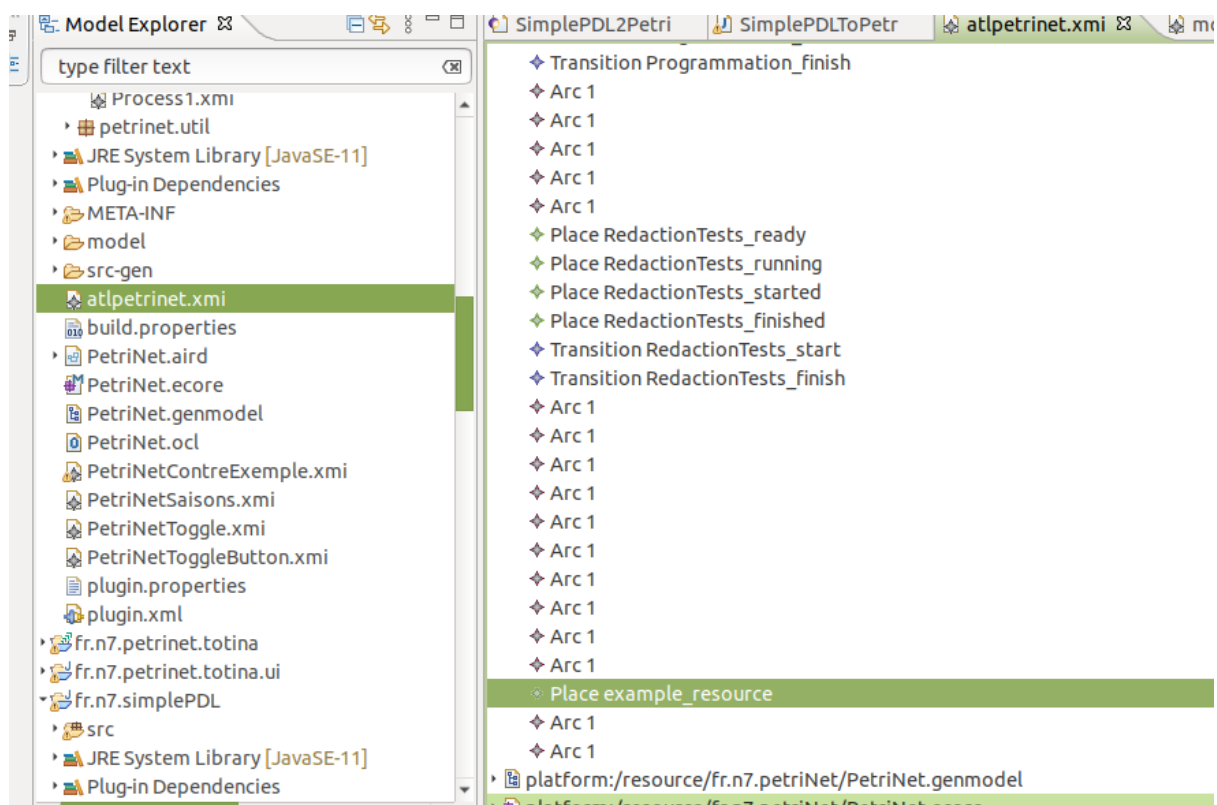


FIGURE 7 - Modèle PetriNet résultant de la transformation ATL

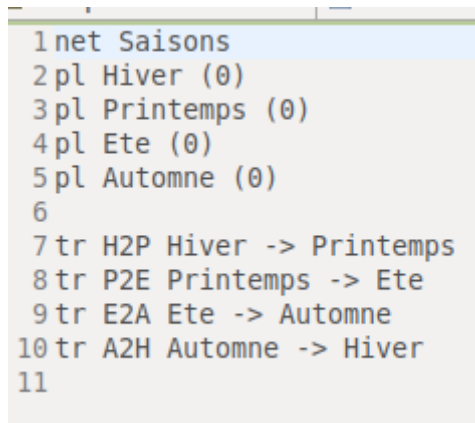
Nous pouvons conclure que la transformation modèle vers modèle en ATL est plus **simple, lisible et maintenable** qu'en EMF Java.

## V - ACCELEO - Transformation de modèle à texte

Il existe un outil qui permet de transformer un modèle en texte. Cet outil se nomme Acceleo. En TP, nous avons créé des transformations **simplePDL to HTML** et **simplePDL to dot**. L'objectif au sein de ce mini-projet a été de réaliser une application aux réseaux de Pétri. Pour cela, il fallait faire une transformation d'un modèle PetriNet **to Tina** pour que l'outil Tina puisse le lire. Ensuite, nous avons fait une transformation PetriNet to dot. Finalement, une transformation SimplePDL **to LTL** a été effectuée.

### ToTina

Vous pouvez voir dans le fichier **totina.mtl** toute la logique pour transformer un fichier PetriNet en syntaxe textuelle Tina. En utilisant cette transformation sur un modèle PetriNet, on obtient le résultat suivant:



```
1 net Saisons
2 pl Hiver (0)
3 pl Printemps (0)
4 pl Ete (0)
5 pl Automne (0)
6
7 tr H2P Hiver -> Printemps
8 tr P2E Printemps -> Ete
9 tr E2A Ete -> Automne
10 tr A2H Automne -> Hiver
11
```

*FIGURE 8 - Résultats de la transformation PetriNet to Tina*

## ToLTL

De même, il a été demandé de définir une transformation d'un modèle vers une syntaxe textuelle de logique LTL. Vous pouvez aussi voir le résultat ci-dessous et le fichier avec les explications dans le fichier **toLTL.mtl**.

```
1 <> (Conception_finished /\ RedactionDoc_finished /\ Developpement_finished /\ RedactionTests_finished);
2
3 [] (Conception_ready + Conception_running + Conception_finished = 1);
4 [] (RedactionDoc_ready + RedactionDoc_running + RedactionDoc_finished = 1);
5 [] (Developpement_ready + Developpement_running + Developpement_finished = 1);
6 [] (RedactionTests_ready + RedactionTests_running + RedactionTests_finished = 1);
7
8 [] (Conception_started => [] (Conception_started));
9 [] (RedactionDoc_started => [] (RedactionDoc_started));
10 [] (Developpement_started => [] (Developpement_started));
11 [] (RedactionTests_started => [] (RedactionTests_started));
12
```

*FIGURE 9 - Règles LTL engendrées depuis simplePDL to LTL*

## VI - SIRIUS - Définition d'une syntaxe graphique

Nous avons en TP créé notre propre **syntaxe graphique** avec l'outil **Sirius** pour pouvoir éditer des modèles. Dans le cadre du mini-projet, nous avons ajouté des calques et des sections relatifs aux nouvelles **ressources** créées afin de pouvoir les ajouter graphiquement. Pour les créer, nous avons d'abord défini comment afficher les ressources dans l'éditeur puis comment le créer. Nous avons donc défini des consignes de représentation graphique. Nous avons fait un **carré gris** pour les ressources avec une **icône rouge**. Nous avons ensuite défini le lien qui permet de relier une WorkDefinition à une ressource. Vous pouvez voir ci-dessous la vue Sirius avec l'inclusion des ressources:

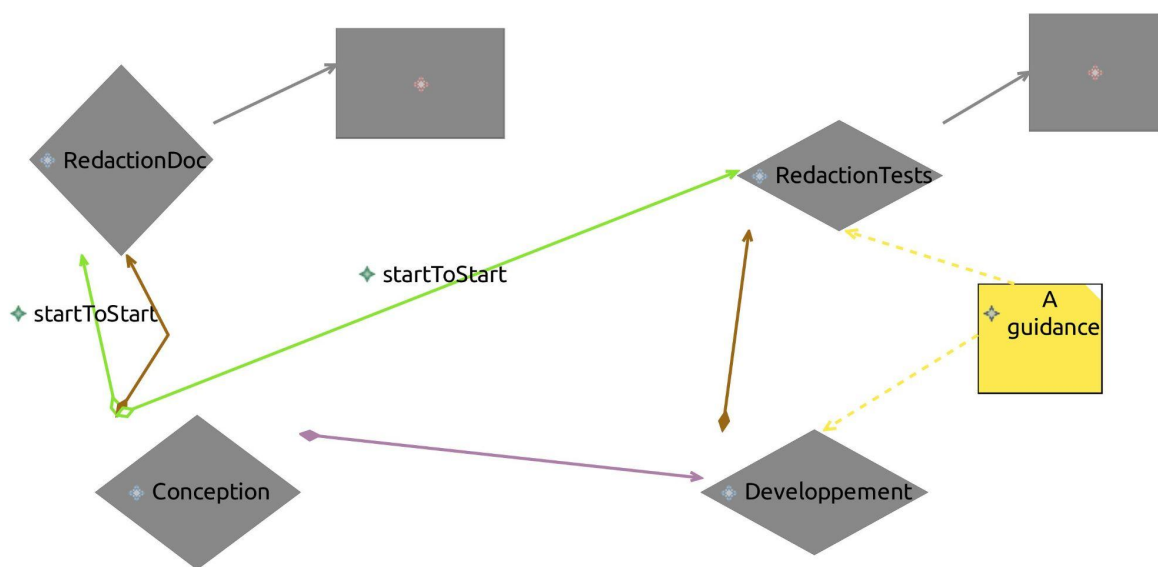


FIGURE 10 - Affichage graphique d'un modèle avec l'éditeur Sirius

Dans notre cas, nous n'avons pas mis de nom aux ressources (carré avec logo rouge au centre), mais nous aurions très bien pu le faire. Vous pouvez voir ci-dessous à quoi ressemble l'arborescence de la création de la syntaxe graphique. On peut y voir les ressources et les liens correspondants.



FIGURE 11 - Arborescence de la création de l'éditeur graphique Sirius

Nous retenons de cette partie qu'il est possible de créer des représentations graphiques des modèles en définissant sa propre syntaxe graphique, pour pouvoir manipuler/éditer des modèles.



## VII - Xtext - Définition d'une syntaxe textuelle

Vous pouvez retrouver le fichier définissant la **syntaxe textuelle** de SimplePDL. On obtient donc une syntaxe textuelle décrivant un processus SimplePDL dans un fichier PDL.xtext. Vous pourrez y voir que l'on a ajouté la notion de **Need** et de **Ressource** sous la forme par exemple : *"need 4 of Human"* ou encore *"create 3 of Machine"*. Need permet de spécifier un besoin de ressource, et create permet de la créer. Vous pouvez voir ci-dessous un exemple d'affichage textuel et on voit bien que les ressources sont prises en compte dans la syntaxe textuelle.

```
1 process Developpement {
2     create 4 of Developpeur
3     create 3 of Serveur
4
5     wd Coder {
6         need 3 of Developpeur
7     }
8
9     wd Deployer {
10        need 2 of Serveur
11    }
12
13    ws finishToStart from Coder to Deployer
14
15    note "Toujours deployer sur aws" for Deployer
16    note "https://stackoverflow.com/" for Coder,Deployer
17 }
```

FIGURE 12 - Exemple d'affichage de la syntaxe textuelle d'un processus SimplePDL de développement

# Conclusion

Nous avons pu voir au travers des TPs et de ce mini-projet que notre travail d'ingénieur, et en particulier ici la modélisation, peut être facilité grâce à de nombreux outils. Il est possible de modéliser des modèles et de créer soi-même des outils (plugins, syntaxes, ...) pour manipuler ces modèles. Parmi ces manipulations, la transformation de modèles/texte vers un autre modèle ou du texte permet d'utiliser les modèles que nous avons pu concevoir: dans le cas de ce projet, il est intéressant de passer du métamodèle du début du cours au réseau de pétri, dynamique et utilisable via Tina. Tout cela a été fait en ajoutant tout au long du mini-projet la notion de ressources. Nous avons généré des modèles graphiquement et textuellement et dans différents types de méta-modèles (SimplePDL, PetriNet...). Il a été très intéressant de passer d'un modèle à l'autre pour pouvoir le visualiser sur Tina.

Nous avons bien saisi le gain de temps que peut nous apporter ces outils pour créer des modèles et outils d'automatisation. Ce sont toutes ces possibilités que nous retiendrons de ce cours d'Ingénierie Dirigée par les Modèles.