

Rapport de Projet

Ingénierie Dirigée par les Modèles



Sommaire

Introduction	3
Point d'entrée des documents	4
Conception des méta-modèles	6
SimplePDL	6
PetriNet	8
Contraintes OCL	9
SimplePDL	9
PetriNet	12
Transformations modèle à modèle	14
EMF	15
ATL	16
Transformations modèle à texte	18
ToTina	18
ToLTL	20
Définition d'une syntaxe graphique	21
Définition d'une syntaxe textuelle	23
Chaîne de vérification de modèle de processus	24
Conclusion	26

Introduction

Ce mini-projet d'IDM avait pour objectif de nous faire produire une chaîne de vérification de modèles de processus SimplePDL. Dans le but de vérifier leur cohérence, la boîte à outils Tina et les outils de *model-checking* qu'elle propose ont été utilisés. Il s'agissait donc de traduire un modèle de processus SimplePDL en un réseau de Petri (afin d'utiliser Tina).

De nombreux outils de modélisation présentés en TP tels que Xtext, Sirius, Acceleo, ATL, OCL, Ecore et EMF ont été utilisés. Tous ces derniers étaient inclus dans une version d'Eclipse qui nous a été fournie et avec laquelle nous avons travaillé.

À travers ce rapport, nous allons présenter notre travail (réalisé en binôme), que nous estimons remplir toutes les exigences initiales (bon, de toute façon, vous avez lu le sommaire 😊).

Point d'entrée des documents

L'entièreté de notre workspace Eclipse vous a été fournie. Il convient donc d'expliquer où retrouver dans notre arborescence tel ou tel document. Si vous souhaitez suivre ce rapport en consultant les fichiers en parallèle, voici où les retrouver :

Nom du projet Eclipse	Chemin d'accès du fichier	Signification
fr.n7.simplePDL	SimplePDL.ecore	Méta-modèle SimplePDL
	SimplePDL.png	Image du méta-modèle SimplePDL
fr.n7.simplePDL.exemples	developpement.xmi	Modèle d'exemple SimplePDL
fr.n7.petriNet	PetriNet.ecore	Méta-modèle PetriNet
	PetriNet.png	Image du méta-modèle PetriNet
fr.n7.simplePDL	SimplePDL.ocl	Fichier de contrainte OCL pour SimplePDL
fr.n7.simplePDL.exemples	developpement_KO_ressources.xmi	Modèle SimplePDL levant l'invariant sur la quantité de ressources
	developpement_KO_reflexivite.xmi	Modèle SimplePDL levant l'invariant sur la réflexivité
fr.n7.petriNet	PetriNet.ocl	Fichier de contrainte OCL pour PetriNet
fr.n7.petriNet.exemples	petrinet_KO_arc_position.xmi	Modèle PetriNet levant l'invariant sur la position des arcs
fr.n7.simplepdl2petrinetEMF	src/simplepdl2petrinetEMF/SimplePDL2PetriNetEMF.java	Code Java de la transformation SimplePDL vers PetriNet
fr.n7.petriNet.exemples	developpementEMF.xmi	Modèle PetriNet transformé à partir de developpement.xmi via EMF
fr.n7.simplepdl2petrinetATL	SimplePDL2PetriNetATL.atl	Code ATL de la transformation SimplePDL vers PetriNet
fr.n7.petriNet.exemples	developpementATL.xmi	Modèle PetriNet transformé à partir de developpement.xmi via ATL
fr.n7.petriNet.toTina	src/fr/n7/petriNet/toTina/main/toTina.mtl	Code Acceleo de la transformation PetriNet vers Tina
fr.n7.petriNet.exemples	developpement.net	Fichier Tina généré à partir de developpementATL.xmi via toTina

fr.n7.simplePDL.toLTL	src/fr/n7/simplePDL/toLTL/main/toLTL.mtl	Code Acceleo de la transformation SimplePDL vers règles LTL
fr.n7.petriNet.exemples	developpement.ltl	Fichier LTL généré à partir de developpement.xmi via toLTL
fr.n7.simplePDL.design	description/simplePDL.odesign	Fichier Sirius de syntaxe graphique SimplePDL
fr.n7.simplePDL.exemples	representations.aird	Fichier de visualisation graphique du modèle developpement.xmi
fr.n7.simplePDL.txt.pdl	src/fr/n7/simplePDL/txt/PDL.xtext	Fichier Xtext de syntaxe textuelle SimplePDL
fr.n7.simplePDL.exemples	developpement.pdl	Exemple de modèle SimplePDL défini avec l'éditeur textuel
	repas.pdl	

Conception des méta-modèles

Il s'agissait dans cette partie de faire de la conception de méta-modèles. Dans le cas de SimplePDL, il s'agissait juste de compléter celui qui nous était fourni afin de prendre en compte l'utilisation de ressources. En ce qui concerne PetriNet, nous l'avons entièrement conçu, à partir des observations faites lors du TP1 (manipulation de Tina).

SimplePDL

Nous ne pensons pas nécessaire de rappeler en détail ce qu'est SimplePDL, retenons juste qu'il permet de modéliser des processus, en définissant des tâches (WorkDefinition) et on les associant les unes avec les autres à l'aide de différents types de liaisons (startToStart, finishToStart, etc...).

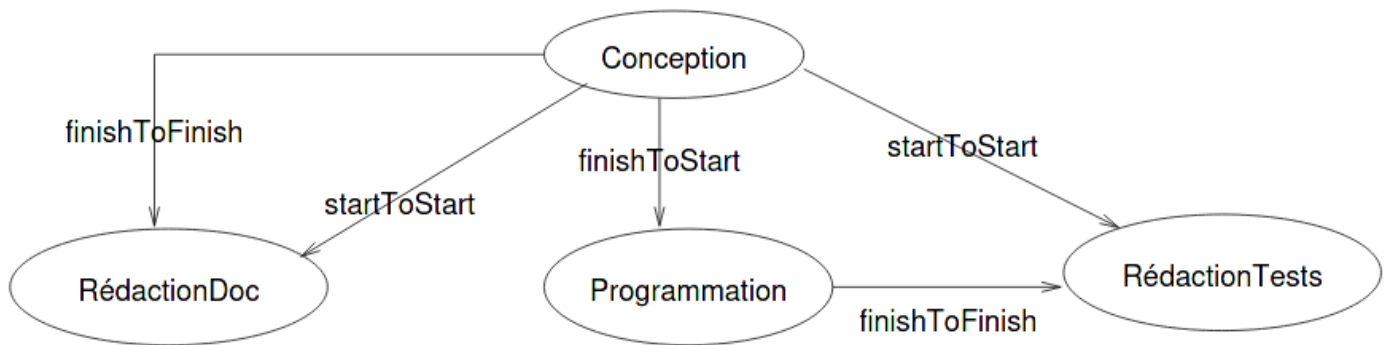


Figure 1 : Exemple de modèle de procédé

Il s'agissait ici de compléter le méta-modèle qui nous a été fourni :

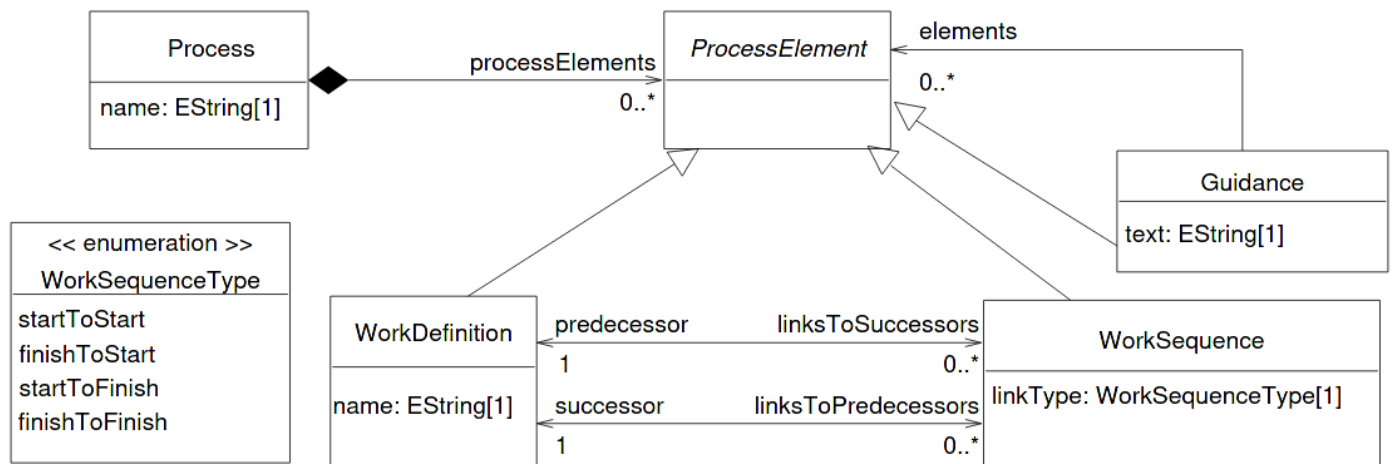


Figure 2 : Méta-modèle SimplePDL du sujet

afin de prendre en charge l'utilisation de ressources par les différentes tâches. Voici le méta-modèle sur lequel nous avons abouti :

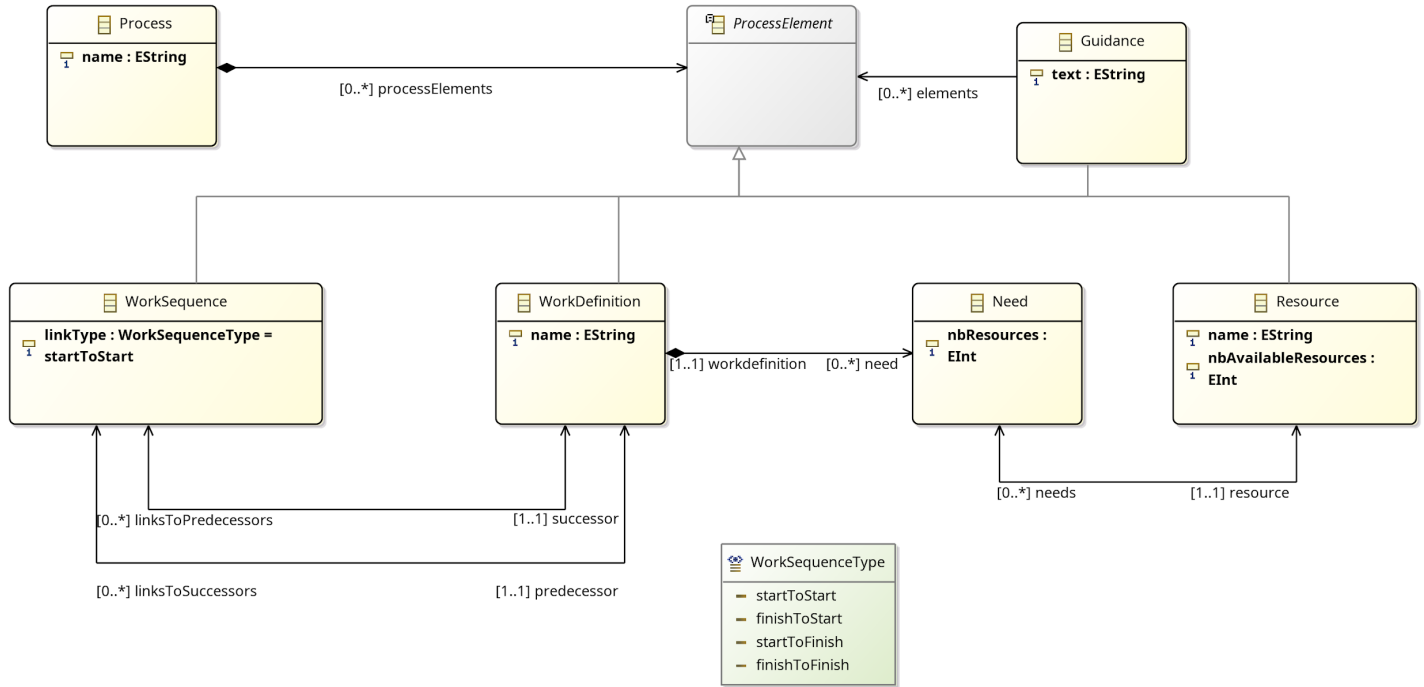


Figure 3 : Méta-modèle SimplePDL avec prise en compte des ressources

On remarque ici l'ajout de deux EClass (Requirement et Resource). Un Requirement représente le fait pour une WorkDefinition d'utiliser une certaine quantité de ressource. Une Resource a un nom suffisamment explicite. Chaque type de ressource est différencié par un nom et a une certaine quantité disponible.

Il semblait cohérent de faire hériter Resource de ProcessElement, en effet, les Resource sont nécessairement rattachées à un processus. En ce qui concerne les Requirement, nous n'avons pas jugé que cela était très pertinent (à priori, un besoin n'a pas besoin d'être rattaché à un processus). Cette décision bien que logique a mené à une certaine difficulté lors de la transformation de modèle à modèle (voir plus bas).

Les liaisons entre WorkDefinition et Requirement ainsi qu'entre Requirement et Resource sont doublement navigables, c'était nécessaire pour les transformations EMF et ATL.

PetriNet

Dans cette partie, nous devons concevoir nous-même le méta-modèle de PetriNet à partir de notre compréhension des réseaux de Petri dans le TP1. Voici ce sur quoi nous avons abouti :

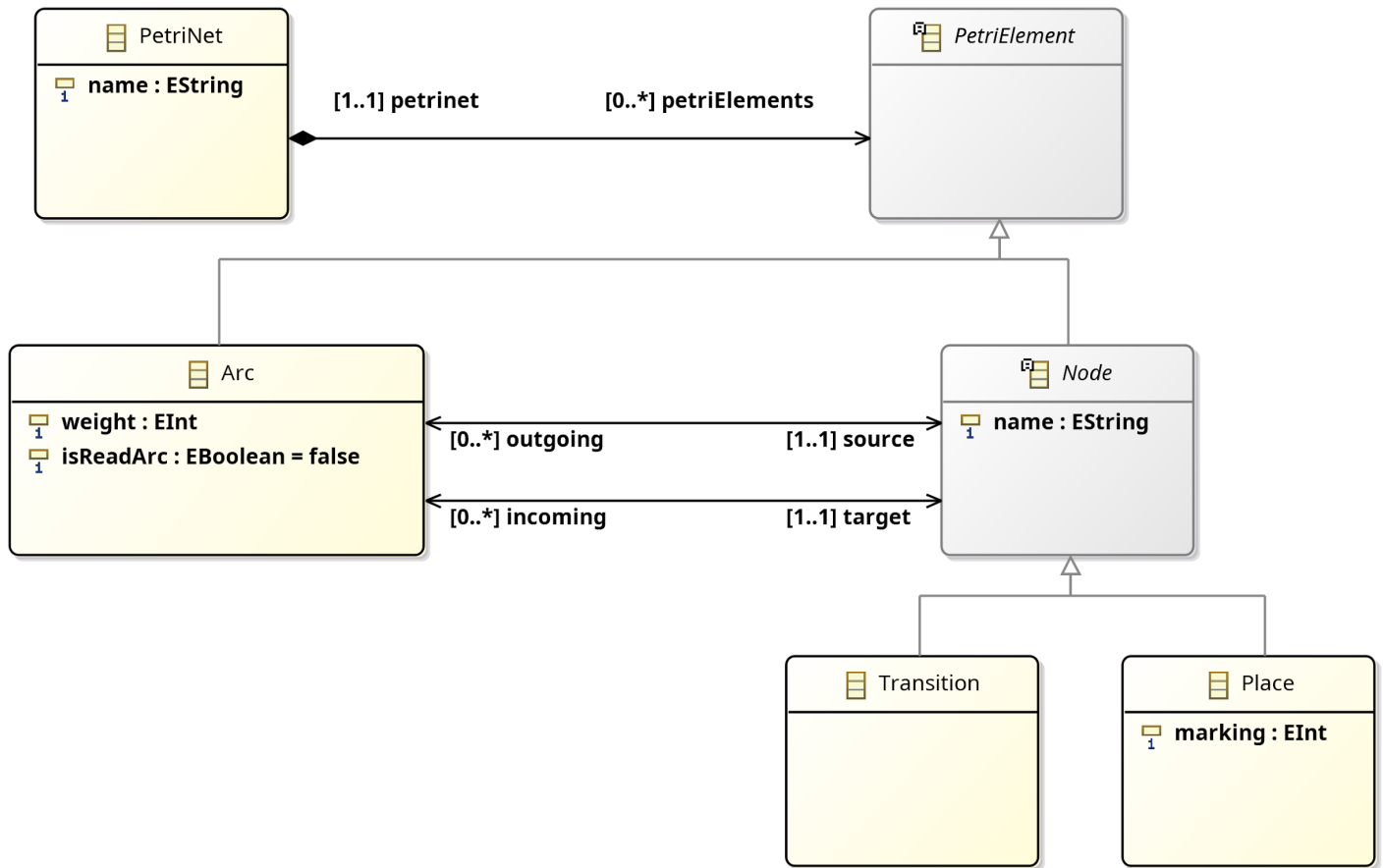


Figure 4 : Méta-modèle PetriNet

Rien de très compliqué ici, on a créé une classe `PetriElement` qui est l'équivalent de `ProcessElement` dans le méta-modèle `SimplePDL`. S'en suit alors deux types : les arcs et les nœuds. Les nœuds peuvent être de deux natures : transition (les éléments rouges sur lesquels on peut cliquer pour faire évoluer le réseau dans Tina) et les places (nœuds stationnaires). Un arc fait nécessairement le lien entre deux nœuds. Un arc peut avoir un poids (coût du passage par cet arc) ou peut ne rien coûter (dans ce cas là, nous avons positionné un booléen `isReadArc` qui, quand il est à vrai, fait en sorte qu'un arc n'ait pas de coût). Les places peuvent avoir un marquage qui représente leur quantité initiale de ressource. Dans les réseaux de Petri, les ressources n'ont pas vraiment de natures différentes (contrairement à `SimplePDL`, nous allons le voir), mais sont plutôt identifiées par les places dans lesquelles elles sont utilisées.

Contraintes OCL

Dans cette partie, des contraintes OCL ont été définies afin de spécifier des invariants qui n'étaient ou ne pouvaient être pris en charge lors de la conception des méta-modèles. OCL est un outil puissant qui permet également d'écrire des requêtes à utiliser sur des modèles, ici nous l'avons seulement utilisé pour définir des invariants. Nous avons néanmoins eu besoin d'une requête permettant de récupérer le Process / PetriNet attaché à un élément.

Bizarrement, il nous était impossible de valider ou d'invalidier nos modèles avec OCL une fois que nos greffons étaient déployés... nous avons donc décidé de fonctionner en parallèle avec un Eclipse possédant les greffons, et un sans. Ce problème semble courant mais nous tenions à le souligner.

Pour éviter de trop allonger cette partie, nous allons donner tous les invariants OCL que nous avons développés, et en présenter quelques-uns via des exemples de modèles qui ne les passent pas.

SimplePDL

Pour SimplePDL, les invariants développés sont les suivants :

- Le nom du Process est valide (au moins deux caractères alphanumériques, commence par une lettre)
- Un prédécesseur et un successeur d'une WorkSequence se situent dans le même Process
- Une WorkSequence n'est pas réflexive (ie. le prédécesseur et le successeur sont distincts)
- Le nom des WorkDefinition est unique (ie. il n'existe pas dans le même Process des WorkDefinition avec le même nom)
- Le quantité disponible de ressources est supérieure ou égale à 1
- Le nom des Resource est unique
- Le besoin en Resource se situe dans le bon interval (ie. quand une WorkDefinition a besoin d'une Resource, elle en demande nécessairement 1 ou plus et maximum la quantité initialement disponible)
- La Resource et la WorkDefinition rattachées à un besoin sont dans le même Process
- Le nom des WorkDefinition est valide (même règle que pour Process)
- Le nom des Resource est valide (même règle que pour Process)
- Il n'existe pas de besoins redondants (ie. il n'existe pas deux Need rattachés à la même Resource et à la même WorkDefinition)

Passons à la présentation de quelques modèles en erreur.

Voici le code de la règle OCL qui vérifie que le nombre disponible de ressources est strictement positif :

```
context Resource
inv hasOneOrMoreNbResources('Resource property "nbAvailableResources" should be greater or equal to 1: ' + self.name):
self.nbAvailableResources >= 1
```

Figure 5 : Invariant OCL portant sur le nombre disponible de ressources

Et voici un modèle ne respectant pas cet invariant :

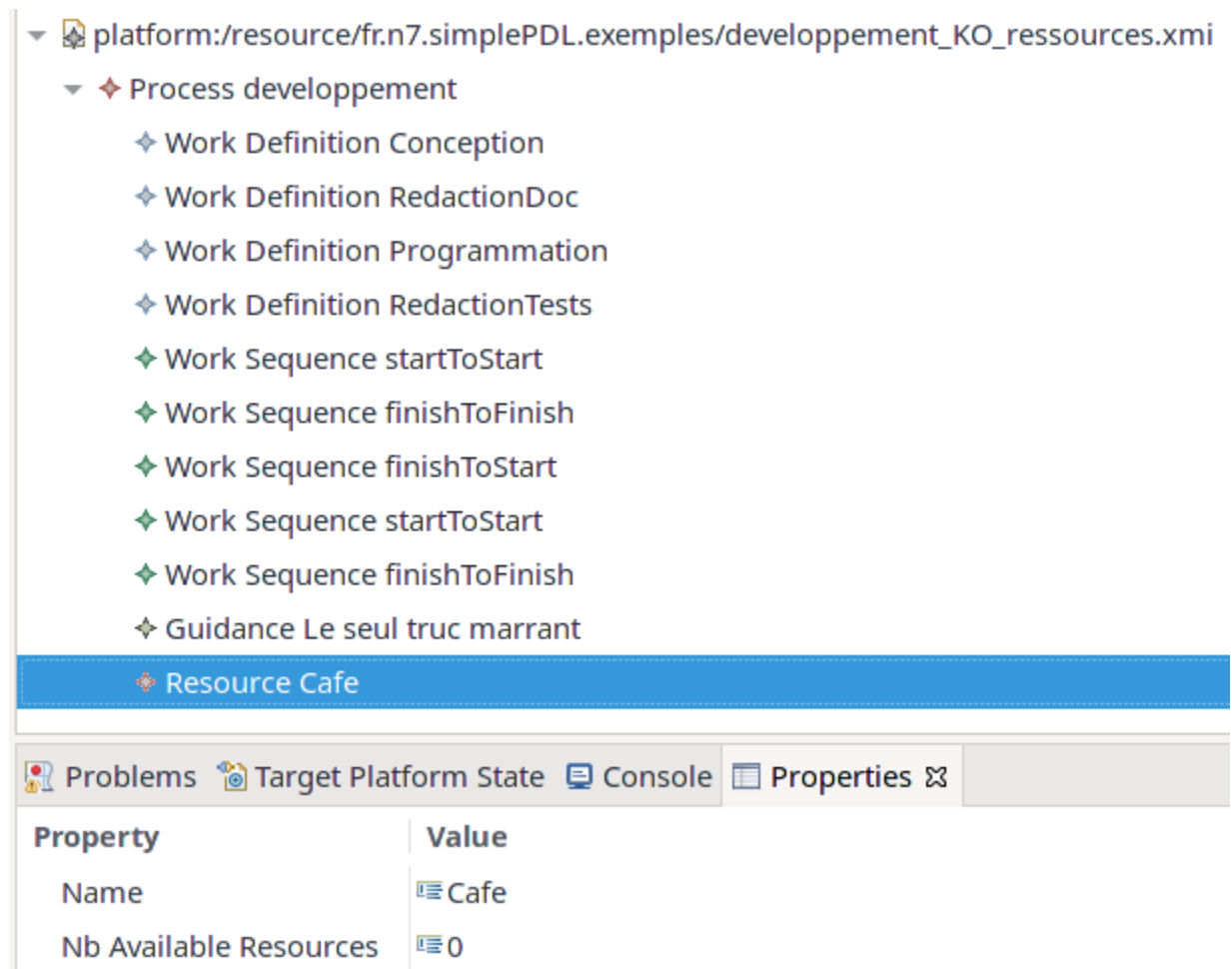


Figure 6 : Exemple d'un modèle SimplePDL ne respectant pas la contrainte sur les ressources

Si on décide de vérifier avec notre fichier OCL que le modèle respecte les contraintes, on obtient bien l'erreur suivante :

⚠ Resource property "nbAvailableResources" should be greater or equal to 1: Cafe

Figure 7 : Message d'erreur de l'invariant OCL portant sur le nombre disponible de ressources

Voici le code OCL qui vérifie qu'une WorkSequence n'est pas réflexive :

```
context WorkSequence
inv notReflexiveSuccessor(
  'Successor (' + self.successor.name + ') and predecessor (' + self.predecessor.name + ') are the same'
):
  self.predecessor <> self.successor
```

Figure 8 : Invariant OCL portant sur la réflexivité des WorkSequence

Voici un modèle ne respectant pas cet invariant :

The screenshot shows a project structure on the left with the following items:

- platform:/resource/fr.n7.simplePDL.exemples/developpement_KO_reflexivite.xml
- Process developpement
 - Work Definition Conception
 - Work Definition RedactionDoc
 - Work Definition Programmation
 - Work Definition RedactionTests
 - Work Sequence startToStart
 - Work Sequence finishToFinish
 - Work Sequence finishToStart
 - Work Sequence startToStart
 - Work Sequence finishToFinish
 - Guidance Le seul truc marrant
 - Resource Cafe

The 'Work Sequence finishToFinish' item is selected, and the Properties window on the right shows the following details:

Property	Value
Link Type	finishToFinish
Predecessor	Work Definition Programmation
Successor	Work Definition Programmation

Figure 9 : Exemple d'un modèle SimplePDL ne respectant pas la contrainte sur la réflexivité des WorkSequence

En validant avec notre fichier OCL, on obtient bien le message d'erreur suivant :

⚠ Successor (Programmation) and predecessor (Programmation) are the same

Figure 10 : Message d'erreur de l'invariant OCL portant sur la réflexivité des WorkSequence

Ici, on pourrait présenter de nombreuses autres contraintes OCL et des modèles en erreur associés, mais on aimerait bien prendre notre week-end 😊.

PetriNet

Pour PetriNet, les invariants développés sont les suivants :

- Le nom d'un PetriNet est valide (toujours les mêmes règles que précédemment)
- La position d'un Arc est valide (ie. il est situé entre une Place et une Transition, qu'importe le sens)
- Le poids d'un Arc est strictement positive
- La source et la destination d'un Arc sont dans le même PetriNet
- Un Arc partant d'une Transition ne peut pas être un ReadArc
- Une Place a un marquage supérieur ou égal à 0
- Le nom d'un Node est valide
- Les Node sont uniques (ie. il n'existe pas de Node avec le même nom dans le même PetriNet)
- Il n'existe pas d'Arc redondant (ie. la même source et la même target)

Passons à la présentation d'un modèle en erreur (on a vraiment la flemme de faire plus 😭).

Voici le code OCL qui vérifie qu'un Arc se situe bien entre un Node et une Transition :

```
context Arc
inv validArcPosition('All arcs should be between Place and Transition'):
  (self.source.ocIsKindOf(Place)
   and self.target.ocIsKindOf(Transition))
  or (self.target.ocIsKindOf(Place)
   and self.source.ocIsKindOf(Transition))
```

Figure 11 : Invariant OCL portant sur la position des Arc

Voici un modèle levant cet invariant :

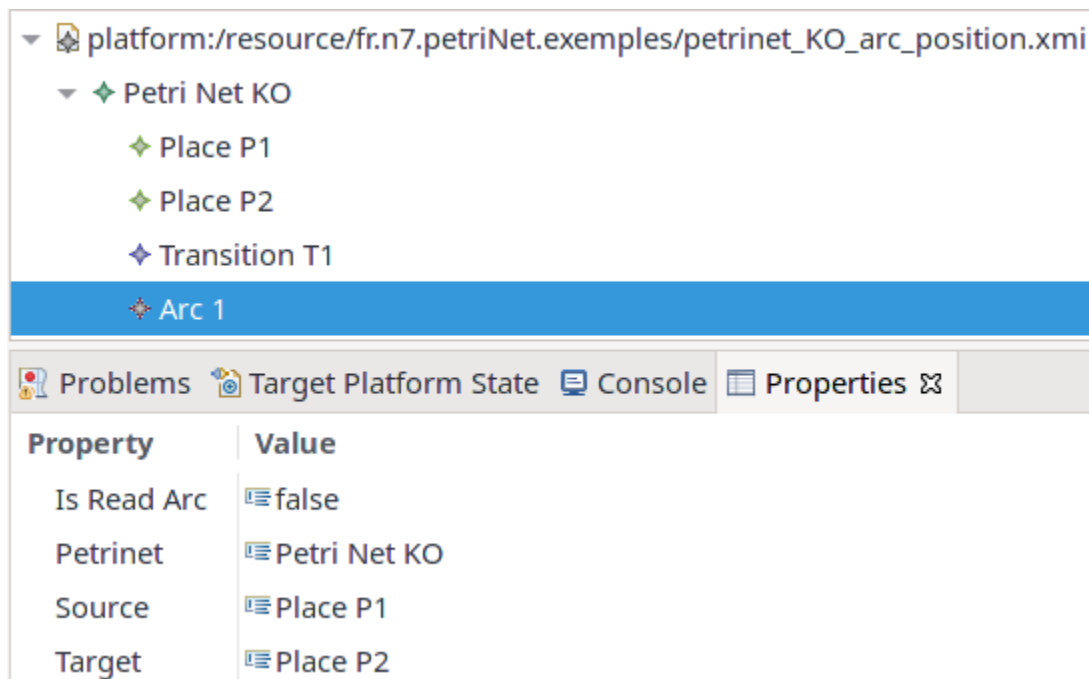


Figure 12 : Exemple d'un modèle PetriNet ne respectant pas la contrainte sur la position des Arc

En validant avec notre fichier OCL, on obtient bien le message d'erreur suivant :

⚠ All arcs should be between Place and Transition

Figure 13 : Message d'erreur de l'invariant OCL portant sur la position des Arc

Transformations modèle à modèle

Dans les deux premières parties, nous avons mis en place deux méta-modèles permettant de représenter un modèle de processus et un réseau de Petri. Nous allons maintenant voir comment passer d'un modèle à l'autre de manière automatisée. Ici nous allons vous présenter deux méthodes permettant de passer d'un modèle SimplePDL à un modèle PetriNet.

Ci-dessous, vous pouvez observer les transformations que nous avons définie pour passer d'un modèle SimplePDL à un modèle d'un réseau PetriNet :

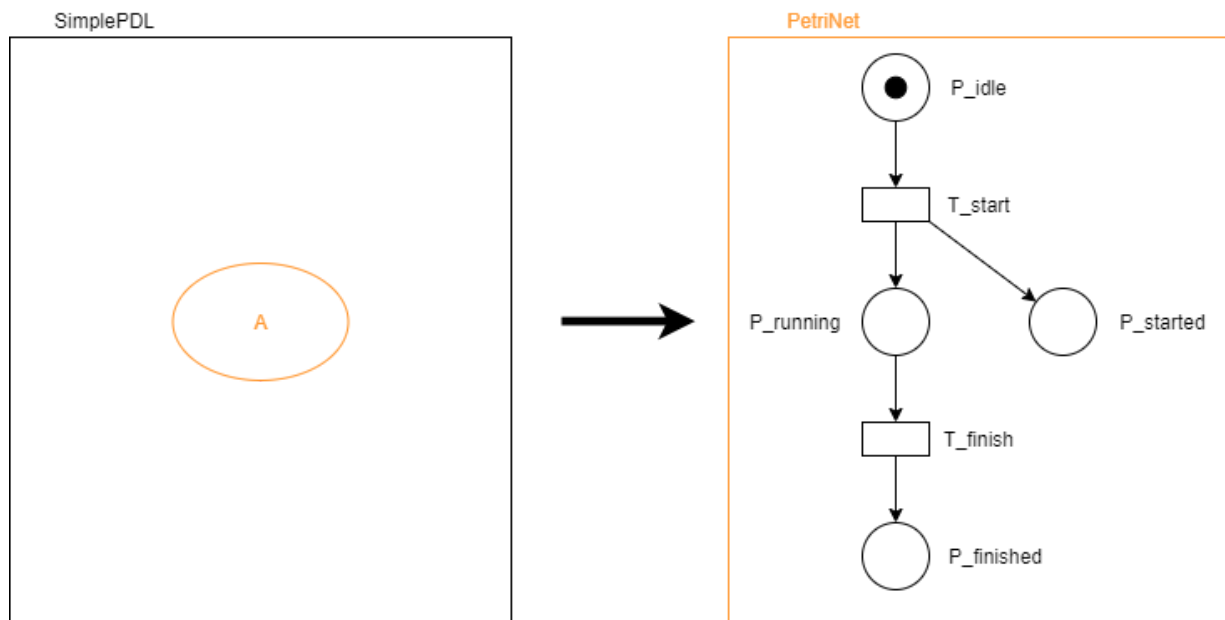


Figure 14 : Transformation d'un Processus SimplePDL vers un réseau PetriNet

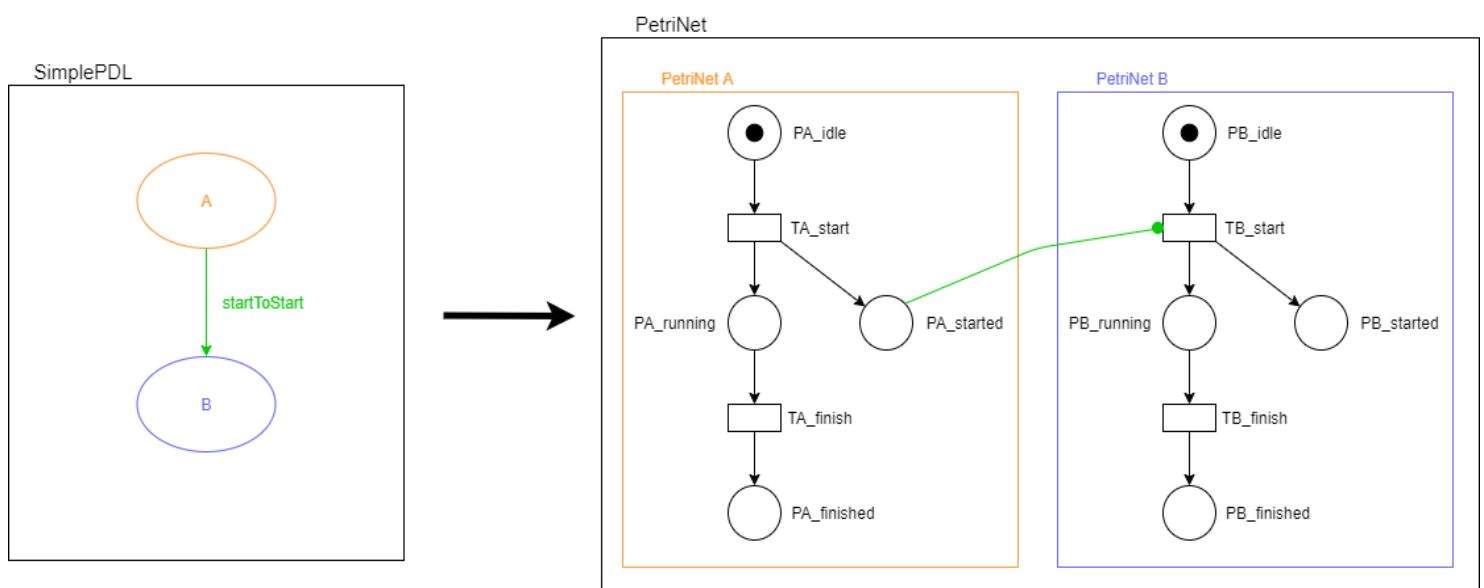


Figure 15 : Transformation d'un processus SimplePDL composé de plusieurs tâches vers un réseau PetriNet

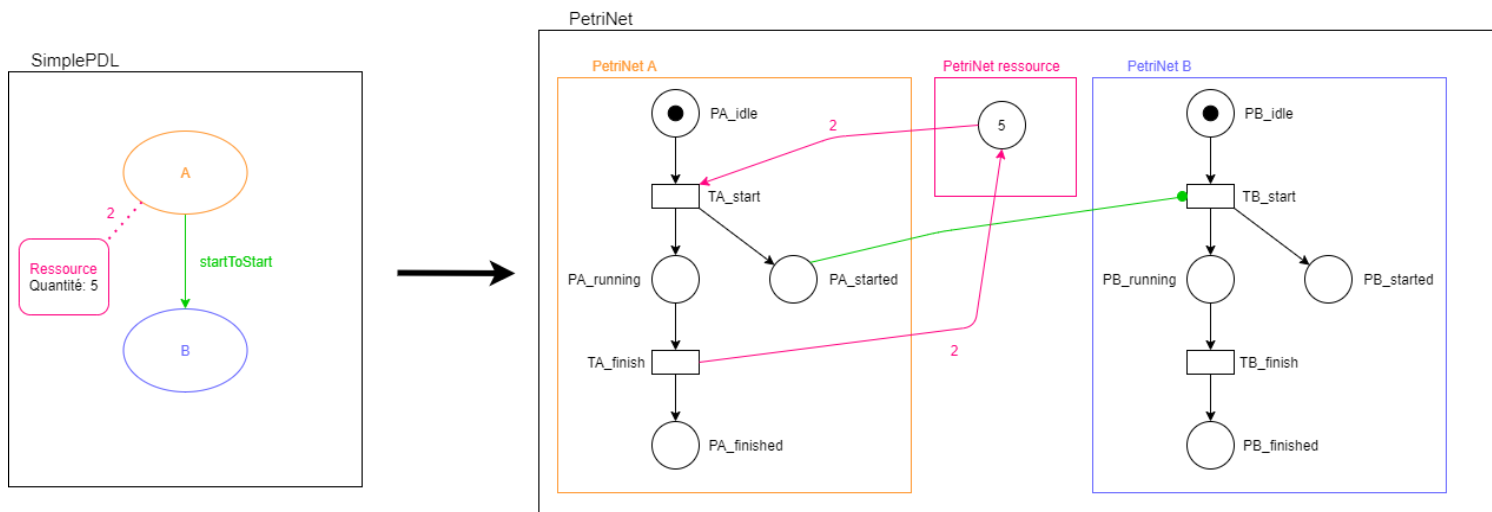


Figure 16 : Transformation d'une chaîne de processus SimplePDL avec ressources vers un réseau PetriNet

EMF

La première méthode est EMF pour Eclipse Modeling Framework. Il va s'agir d'exploiter les possibilités offertes d'office par le framework Eclipse. Cette méthode permet de passer d'un modèle à un autre en utilisant du code Java. À partir des deux méta-modèles que nous avons écrits, du code Java a été généré permettant d'interagir avec eux : des classes (avec getter/setter) représentant les différents éléments des méta-modèles et une factory pour les créer.

Nous avons alors écrit un programme parcourant les éléments du modèle SimplePDL et appliquant les transformations afin de créer le modèle PetriNet (images précédentes) à l'aide des outils générés (classes, factories...). Nous nous sommes inspirés de ce qui a été vu en TP.

Afin de ne pas avoir à modifier le code Java à chaque fois que nous souhaitons changer le modèle d'entrée, nous avons fait en sorte qu'il prenne deux paramètres. Le premier est le chemin d'accès du modèle SimplePDL à transformer, le deuxième est le chemin d'accès du modèle PetriNet généré.

Difficultés rencontrées :

- La gestion des références objets : chacun des éléments des modèles ont des références vers d'autres éléments.
- La complexité du code : il faut parcourir tous les éléments du SimplePDL, puis déduire quel est le type de cet élément (donc cast obligatoire), puis le traduire en élément PetriNet, stocker sa référence car un autre élément peut en avoir besoin... etc

Axes d'améliorations :

- Nous avons fait le choix de ne pas stocker les références des objets mais de les récupérer à chaque fois que l'on en a besoin : grande complexité car à chaque fois que l'on a besoin d'une référence on parcourt tous les objets du réseau de Petri créé pour trouver la référence de l'objet souhaité. La solution serait de stocker les références dans une structure de données appropriée (ex: HashMap) afin de faciliter la recherche et de réduire la complexité.
- Utiliser ATL !

ATL

La seconde méthode que nous avons utilisée se base sur ATL. Le langage ATL permet la traduction de modèle d'un méta-modèle A vers un modèle d'un méta-modèle B différent. Le principe d'ATL est de définir un ensemble de règles (transformations) permettant de passer d'un élément du méta-modèle A vers un/plusieurs élément/s du méta-modèle B.

En prenant dans notre cas le méta-modèle A SimplePDL et le méta-modèle B PetriNet, une règle est représentée de cette manière :

```
rule NomTransformation {
  from nom_element_SimplePDL: simplepdl!Element_SimplePDL
  to
    nom_element1_PetriNet: petrinet!Element_Petrinet (attributs..)
    nom_element2_PetriNet: petrinet!Element_Petrinet (attributs..)
    .
    .
    .
}
```

Figure 17 : Règle ATL pour la transformation d'un élément SimplePDL vers PetriNet

On peut observer une amélioration apparaître au niveau de la complexité du code écrit. Il va nous suffire d'écrire une règle par élément SimplePDL que l'on veut transformer en PetriNet. Voici par exemple la règle que nous avons écrite pour la transformation d'une WorkDefinition vers PetriNet (correspond à la transformation représentée dans la Figure 14) :

```
-- Traduire une WorkDefinition en un motif sur le réseau de Petri
rule WorkDefinition2PetriNet {
  from wd: simplepdl!WorkDefinition
  to
    -- PLACES d'une WorkDefinition
    p_ready: petrinet!Place(...),

    p_running: petrinet!Place(...),

    p_started: petrinet!Place(...),

    p_finished: petrinet!Place(...),

    -- TRANSITIONS d'une WorkDefinition
    t_start: petrinet!Transition(...),

    t_finish: petrinet!Transition(...),

    -- ARCS d'une WorkDefinition
    a_ready2start: petrinet!Arc(...),

    a_start2running: petrinet!Arc(...),

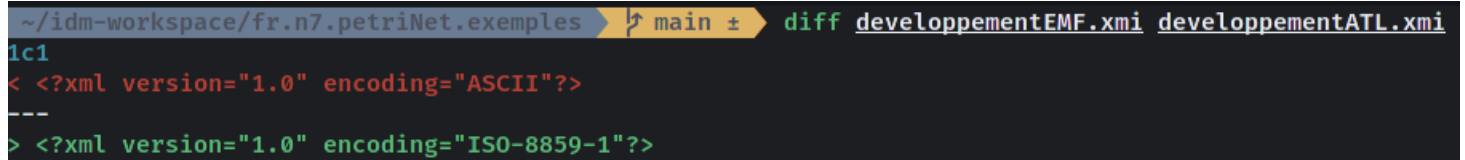
    a_start2started: petrinet!Arc(...),

    a_running2finish: petrinet!Arc(...),

    a_finish2finished: petrinet!Arc(...)
}
```

Figure 18 : Règle ATL pour la transformation d'une WorkDefinition vers PetriNet

La grande classe dans cette partie est que lorsque l'on fait un diff entre deux modèles PetriNet transformés à partir du même modèle SimplePDL, l'un avec EMF et l'autre avec ATL, on obtient aucune ligne de différence si ce n'est l'encodage du fichier, ce signifie que notre code Java est on ne peut plus juste :



```
~/idm-workspace/fr.n7.petriNet.exemples > main ± diff developpementEMF.xmi developpementATL.xmi
1c1
< <?xml version="1.0" encoding="ASCII"?>
--
> <?xml version="1.0" encoding="ISO-8859-1"?>
```

Figure 19 : Aucune différence entre les transformations EMF et ATL

Transformations modèle à texte

Les deux sous-parties qui vont suivre portent sur la transformation de modèle à texte. Dans la première, il s'agissait de savoir transformer un modèle conforme au méta-modèle PetriNet en un fichier .net utilisable par Tina. Dans la deuxième, il s'agissait de générer les propriétés LTL relatives à un réseau de Petri. Nous avons utilisé l'outil Acceleo qui nous a été présenté en TP.

Un petit problème technique a été que Acceleo ne trouvait pas le code Java correspondant à nos méta-modèles, bien que nos greffons soient déployés. La solution a été de modifier la méthode "registerPackages" des fichiers Java générés par Acceleo.

ToTina

Pour la transformation de PetriNet à fichier Tina, nous avons commencé par développer deux fonctions qui nous renvoient l'ensemble des Places et l'ensemble des Transitions du modèle :

```
[query public getPlaces(p: PetriNet) : OrderedSet(Place) =
    p.petriElements->select(e | e.ocIsTypeOf(Place))
    ->collect(e | e.ocAsType(Place))
    ->asOrderedSet()
/]

[query public getTransitions(p: PetriNet) : OrderedSet(Transition) =
    p.petriElements->select(e | e.ocIsTypeOf(Transition))
    ->collect(e | e.ocAsType(Transition))
    ->asOrderedSet()
/]
```

Figure 20 : Fonctions Acceleo de récupération des places et des transitions

Nous avons ensuite développé une fonction ayant pour but de transformer une transition en texte :

```
[template public toTina(tr : Transition) post (trim()) ]
tr [tr.name /] [for (arc: Arc | tr.incoming)
    before ( ' ' ) separator ( ' ' ) after ( ' ' )
][arc.source.name /][if (arc.isReadArc)]?[arc.weight/][elseif (arc.weight > 1)]*[arc.weight/][if][for]->[for (arc: Arc | tr.outgoing)
    before ( ' ' ) separator ( ' ' ) after ( ' ' )
][arc.target.name /][if (arc.isReadArc)]?[arc.weight/][elseif (arc.weight > 1)]*[arc.weight/][if][for]
[/template]
```

Figure 21 : Fonction de transformation d'une transition en texte

On affiche ici les arcs entrants (incoming) et sortants (outgoing) d'une transition en prenant en compte les éventuels coûts en ressource (dans la conditionnelle).

Le reste du programme est composé de deux simples boucles :

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

[template public petrinetToTina(aPetriNet : PetriNet)]
[comment @main/]
[file (aPetriNet.name + '.net', false, 'UTF-8')]
[let places : OrderedSet(Place) = aPetriNet.getPlaces() ]
[for (place : Place | places)]
pl [place.name/] ([place.marking/])
[/for]
[/let]

[let transitions: OrderedSet(Transition) = aPetriNet.getTransitions() ]
[for (transition : Transition | transitions)]
[transition.toTina()/]
[/for]
[/let]
[/file]
[/template]
```

Figure 22 : Boucles principales de la transformation Acceleio de PetriNet vers Tina

La première parcourt l'ensemble des places, puis les affiche sous forme la forme “pl <nom_de_la_place> (<marquage_de_la_place>)”. Nous n'avons pas jugé nécessaire de développer une fonction pour cela.

La deuxième parcourt l'ensemble des transitions, et appelle la fonction présentée précédemment pour les transformer en texte dessus.

Cela donne des fichiers de la même forme que ceux présentés dans le TP1 (saisons.net). Notons que Tina semble capable “d'inférer” les places à partir des transitions, mais nous ne savions pas sous quelles modalités il en était capable, c'est la raison pour laquelle nous transformons systématiquement les places en une ligne commençant par “pl” en début de fichier.

ToLTL

Dans cette partie, il aurait pu sembler logique de développer une transformation de modèle PetriNet à propriétés LTL, puisque c'est sur des réseaux de Petri que nous voudrions vérifier ces propriétés. Cependant, il était bien plus simple de développer une transformation à partir de modèle de processus SimplePDL en propriétés LTL (le fait par exemple qu'une WorkDefinition donne lieu à 4 Places avec les Transitions et les Arcs qui vont avec donne tout de suite beaucoup plus de complexité). Nous avons donc fait le choix de nous baser sur des modèles SimplePDL.

La seule règle LTL qui nous a été spécifiée était de vérifier que les réseaux de Petri terminent, il fallait donc vérifier que toutes les activités arrivent dans l'état finished. Nous devons ensuite laisser libre court à notre imagination pour trouver d'autres règles LTL, nous avons choisi de vérifier la cohérence des états : une activité ne peut être que dans un état à la fois (ready, running ou finished). Enfin, nous vérifions qu'une activité démarrée reste toujours démarrée.

Pour rendre les fichiers générés plus lisibles, nous faisons trois parcours des WorkDefinition. Il serait plus optimisé de tout faire en une seule itération, néanmoins, Acceleo nous a posé beaucoup de problèmes au niveau du formatage des espaces et des retours de ligne dans les fichiers de sortie. Quoi qu'il en soit, voici ces trois boucles :

```
[comment Toutes les activités finissent (T10) /]  
[for (wd: WorkDefinition | wds)  
before ('<> (') separator (' /\ ' ') after(');')  
][wd.name /]_finished[/for]
```

Figure 23 : Génération de la propriété de terminaison des réseaux de Petri

```
[comment Cohérence des états, un seul état par activité (T11) /]  
[for (wd : WorkDefinition | wds)]  
['[]' /]([wd.name /]_ready + [wd.name /]_running + [wd.name /]_finished = 1);  
[/for]
```

Figure 24 : Génération de la propriété de cohérence des états des réseaux de Petri

```
[comment Une activité démarrée reste démarrée pour toujours (T11) /]  
[for (wd: WorkDefinition | wds)  
]['[]' /]([wd.name /]_started => ['[]' /] ([wd.name /]_started));  
[/for]
```

Figure 25 : Génération de la propriété portant sur le démarrage des états des réseaux de Petri

Définition d'une syntaxe graphique

Dans cette partie, il s'agissait de définir une syntaxe graphique permettant de saisir des modèles conformes à SimplePDL. Il s'agit d'une alternative à la partie qui suit, dans laquelle nous définissons une syntaxe textuelle. Ici, l'outil Sirius a été utilisé.

Le TP présentant cet outil nous ayant amené à faire quasiment tout, nous avons juste eu à rajouter la prise en charge des ressources et des besoins, ainsi que quelques améliorations graphiques telles que la différenciation des WorkSequence en fonction de leur nature.

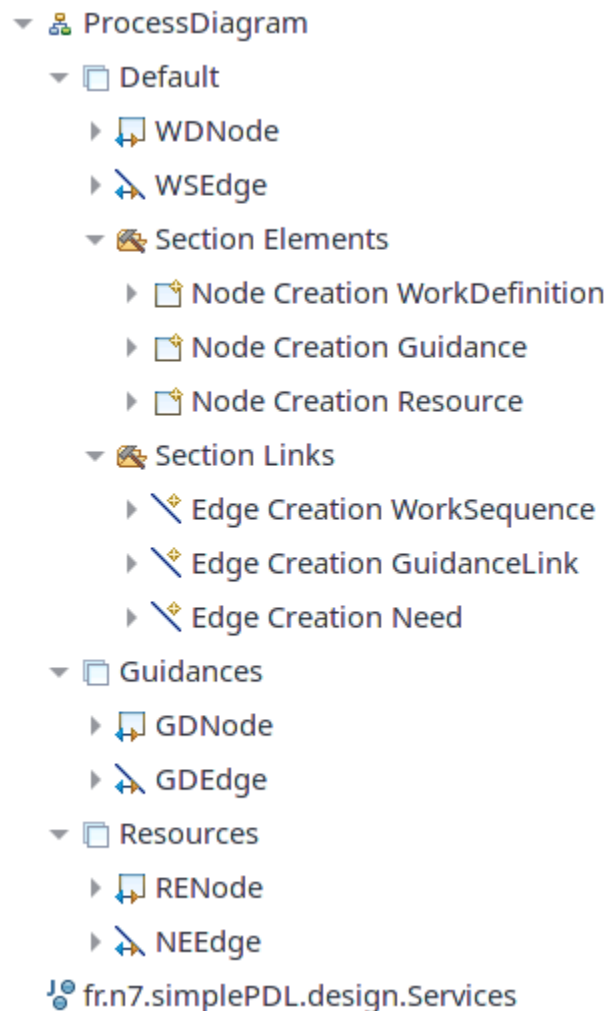


Figure 26 : Arborescence de la syntaxe graphique

Nous avons, comme ça l'était suggéré en TP, séparé notre boîte à outil en deux sections : les éléments (nœuds) et les liens ; ainsi qu'en calques : un par défaut qui continent les WorkSequence et les WorkDefinition, un qui contient les Guidance et les liens les reliant aux éléments, et un qui contient les Resource et les Need.

Nous avons utilisé le langage AQL (Acceleo Query Language) pour définir des syntaxes conditionnelles sur les WorkSequence.

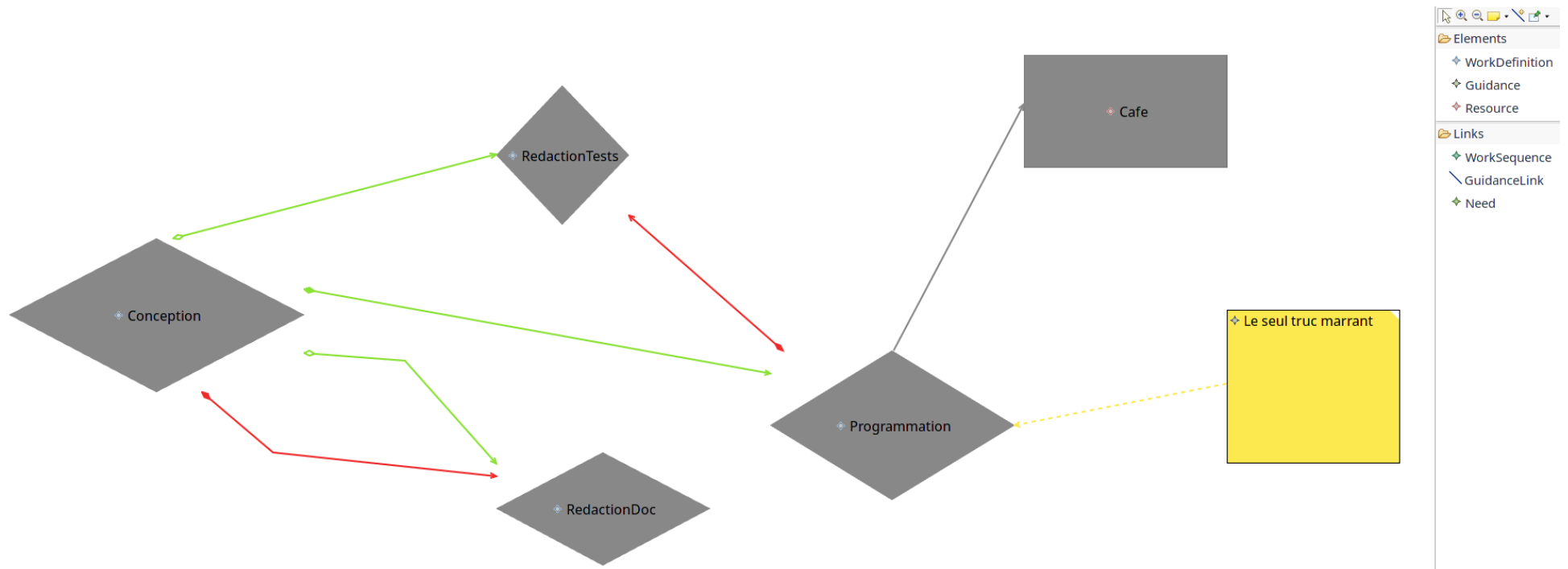


Figure 27 : Exemple de modèle SimplePDL et syntaxe graphique Sirius

La difficulté ici a essentiellement été de gérer les différents types de lien. Il en existe deux : les Element Base Edge et les Relation Base Edge. Les uns sont représentés par une classe du modèle (comme les WorkSequence, par exemple) tandis que les autres permettent d'associer un élément du modèle à un attribut d'un autre élément du modèle (c'est par exemple le cas du lien qui relie les Guidance avec des ProcessElement).

Il y a deux étapes majeures dans la définition de syntaxes graphiques avec Sirius, la première est de définir comment les éléments vont s'afficher dans l'éditeur, la deuxième va être de définir comment créer un élément. C'est la deuxième qui a été la plus compliquée. Pour simplifier ce rapport déjà beaucoup trop long, nous n'allons pas détailler ces dernières ici. Le fichier .odesign est fourni dans les sources.

Définition d'une syntaxe textuelle

Après avoir défini une syntaxe graphique (*cf partie précédente*), nous avons défini une syntaxe textuelle permettant de décrire un processus SimplePDL dans un fichier .pdl grâce au greffon XText.

Documentation de notre syntaxe PDL :

- **Création d'un processus :** `process` <nom_du_processus> {...}
 - **Création d'une ressource :** `create` <quantité_de_la_ressource> `of` <nom_de_la_ressource>
 - **Création d'une tâche (Workdefinition) :** `task` <nom_de_la_tâche> {...}
 - **Spécifier un besoin de ressource :**
`need` <quantité_requise_de_la_ressource> `of` <nom_de_la_ressource>
 - **Créer une dépendance entre deux tâches :**
`dep` <startToStart|startToFinish|finishToStart|finishToFinish> `from` <nom_tâche> `to` <nom_tâche>
 - **Créer une note pour aucun/un/plusieurs éléments du processus :**
`note` <text_de_la_note> [`for` <nom_element>, <nom_element>, <nom_element>, ...]¹

Voici un exemple de code pdl illustrant les différentes notions de la syntaxe :

```
process Développement {
  create 4 of Developpeur
  create 3 of Serveur

  task Coder {
    need 3 of Developpeur
  }

  task Deployer {
    need 2 of Serveur
  }

  dep finishToStart from Coder to Deployer

  note "Pas sur OVH (lol)" for Deployer
  note "Ne pas utiliser trop d'argent" for Coder, Deployer
}
```

Figure 28 : Fichier .pdl décrivant un processus SimplePDL de développement

Xtext génère automatiquement un IDE pour la syntaxe (avec linter, auto complétion, colorations...).

Apports : Si la syntaxe est bien définie et respecte le métamodèle, alors on peut garantir que le modèle créé à partir de cette syntaxe le respectera également. Une syntaxe textuelle pourrait également à l'avenir servir pour générer des fichiers de modèle automatiquement (plus simple de générer un code concis comme notre pdl plutôt que du xmi). Et pour finir cela permet de donner une alternative aux personnes allergiques aux interfaces graphiques. #Ctrl+Alt+F[1-6]

¹ Les [] signifient que le code à l'intérieur est optionnel.

Dans le cas d'une note : on peut créer une note mais ne la lier à aucun élément : `note` <text_de_la_note>

Chaîne de vérification de modèle de processus

Comme vous l'avez peut-être remarqué au fil de ce rapport, l'exemple de modèle de processus sur lequel nous avons appliqué notre chaîne de vérification / transformation est celui de la *Figure 1* (et du sujet du mini-projet). Nous y avons ajouté la ressource café qui a une quantité initiale de 10 et un besoin de 5 cafés pour la WorkDefinition Programmation (*Figure 27*).

Les diverses transformations (SimplePDL vers PetriNet, PetriNet vers Tina, SimplePDL vers LTL) ayant été présentées, nous allons ici montrer l'objectif du projet : à savoir, la génération d'un réseau de Petri à partir d'un modèle de processus SimplePDL valide.

Toujours dans le même exemple de modèle, voici le réseau de Petri généré par notre chaîne de transformation :

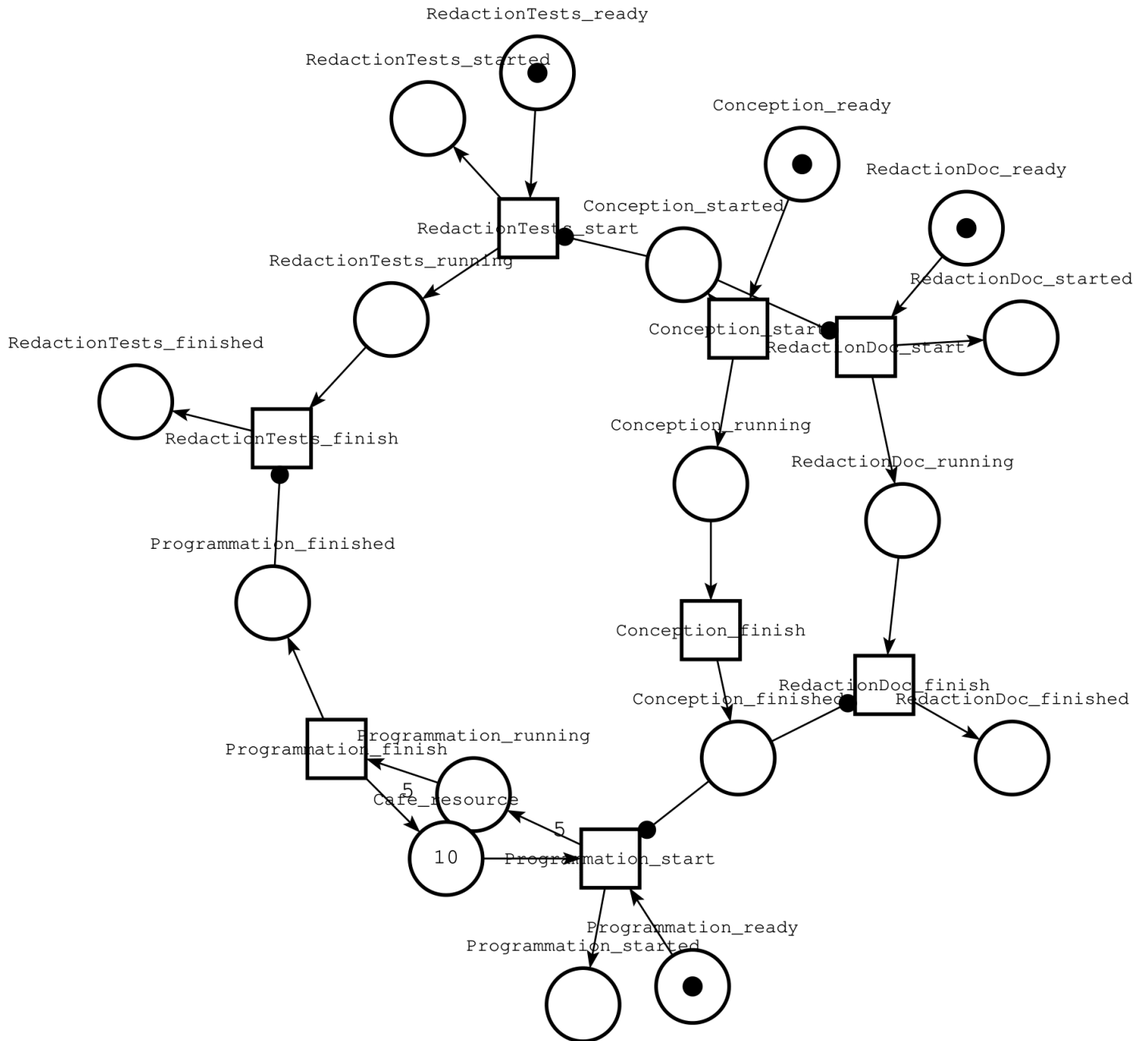


Figure 29 : Réseau de Petri correspondant au modèle de processus Développement

Voici le fichier de règles LTL générées par notre chaîne de vérification :

```
<> (Conception_finished /\ RedactionDoc_finished /\ Programmation_finished /\ RedactionTests_finished);

[] (Conception_ready + Conception_running + Conception_finished = 1);
[] (RedactionDoc_ready + RedactionDoc_running + RedactionDoc_finished = 1);
[] (Programmation_ready + Programmation_running + Programmation_finished = 1);
[] (RedactionTests_ready + RedactionTests_running + RedactionTests_finished = 1);

[] (Conception_started => [] (Conception_started));
[] (RedactionDoc_started => [] (RedactionDoc_started));
[] (Programmation_started => [] (Programmation_started));
[] (RedactionTests_started => [] (RedactionTests_started));
```

Figure 30 : Règles LTL du réseau de la Figure 29

On peut bien voir que les règles sont validées avec selt :

```
~/tina bin/selt -p -S MP/developpement.scn MP/developpement.ktz -prelude MP/developpement.ltl
Selt version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 26 states, 47 transitions
0.002s

- source MP/developpement.ltl;
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
0.004s

- |
```

Figure 31 : Sortie de la commande selt lors de la vérification des règles de la Figure 30 sur le réseau de la Figure 29

Conclusion

En conclusion, nous sommes parvenus à bout de ce mini-projet d'IDM. Bien que pénible au début car nous ne comprenions pas la multitude des outils présentés et avons eu quelques soucis avec ces derniers, nous avons pris du plaisir à mesure que nous comprenions ce dans quoi nous avancions. Le fait de parvenir à créer des modèles conformes à SimplePDL graphiquement ou textuellement avec des syntaxes que nous avons défini est déjà plutôt intéressant. Mais le fait de pouvoir les transformer en réseaux de Petri puis les visualiser dans Tina était vraiment gratifiant ! À mesure que nous avancions, nous avons pris de plus en plus de plaisir à faire ce projet.

Le cours nous a paru pertinent bien que compliqué à comprendre au début. Nous arrivons facilement à voir le gain de temps que peut être le travail d'ingénieur en utilisant des modèles et outils d'automatisation 😊.