

Rapport du mini projet d'IDM

Ayoub Bouchama

Ayoub Diouri

Oussama ElGuerraoui

Contents

1	Introduction	2
2	Les métamodèles	2
2.1	SimplePDL	2
2.2	PetriNet	2
3	Les contraintes OCL associés aux métamodèles	3
3.1	SimplePDL	4
3.2	PetriNet	6
4	La transformation modèle à modèle	6
4.1	Transformation SimplePDL vers PetriNet en utilisant EMF/JAVA	7
4.2	Transformation SimplePDL vers PetriNet en utilisant ATL	7
5	Transformation modèle à texte (M2T) avec Acceleo	9
5.1	Transformation PetriNet vers Tina	9
6	Modèle xtext	13
7	Conclusion	14

1 Introduction

Ce petit projet implique le développement d'une chaîne de vérification pour les modèles de processus SimplePDL, visant à examiner leur cohérence, notamment en déterminant si le processus décrit peut se conclure. Pour résoudre cette problématique, nous exploiterons les techniques de vérification de modèles définies sur les réseaux de Petri via l'outil Tina. Ainsi, la tâche consistera à traduire un modèle de processus en un réseau de Petri avec différentes méthodes.

2 Les métamodèles

2.1 SimplePDL

Le langage SimplePDL, abréviation de "Simple Process Description Language", est un langage de métamodélisation utilisé dans le domaine de l'ingénierie logicielle pour décrire des modèles de processus de développement. Le métamodèle SimplePDL se compose initialement de deux concepts fondamentaux :

- **WorkDefinition** : représentant les activités.
- **WorkSequence** : décrivant les dépendances entre ces activités.

Dans le cadre de notre projet, nous avons étendu le métamodèle SimplePDL pour inclure de nouveaux concepts qui enrichissent la représentation des processus de développement. Ces ajouts comprennent deux classes clés :

- **Ressource** (*Ressource*) : Une ressource est caractérisée par son nom, qui décrit son type, et par le nombre d'occurrences associées. Ces ressources représentent les entités (humaines, matérielles, etc.) nécessaires à l'exécution des activités.
- **RessourceAllocation** (*Allocation*) : La classe *RessourceAllocation* représente le nombre d'occurrences d'une ressource allouées à une activité. Cette allocation spécifie combien d'occurrences d'une ressource particulière sont utilisées exclusivement par une activité pendant sa réalisation.

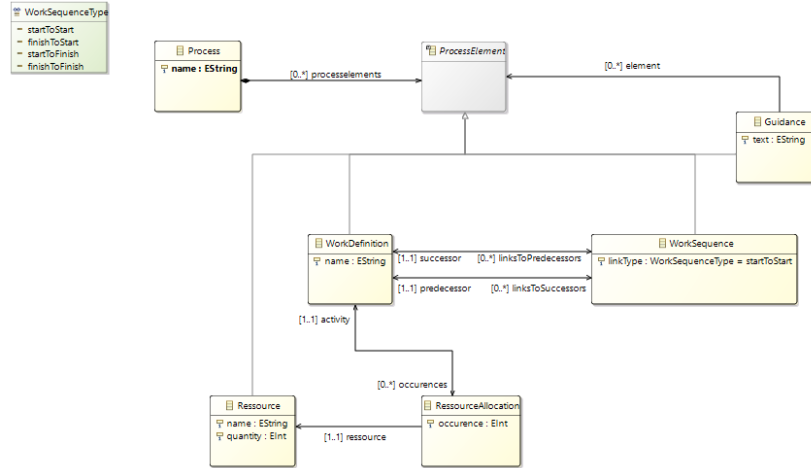


Figure 1: Métamodèle SimplePDL

2.2 PetriNet

Nous débutons la création de notre structure en définissant la classe **PetritNet**, qui représente un réseau de Petri. En se basant sur la conceptualisation du réseau de Petri, nous identifions les com-

posants clés tels que les arcs, les places et les transitions.

1. Classe Node

La classe, **Node**, caractérisée par son attribut principal (**name:EString**), est étendue par **Place** et **Transition**, en raison de leurs propriétés communes. Chaque **Node** est également équipé d'une liste arcs de sortie (**linksToTarget:Arc**) et d'une liste d'arcs d'entrée (**linksToSource:Arc**).

2. Classe Place

La classe **Place** est caractérisée par son attribut principal (**Token:EInt**) qui constitue le nombre de jetons que possède la chaque place et qu'elle peut transférer à travers les arcs.

3. Classe Transition

La classe **Transition** représente les transitions de réseau de petri et ne contient qu'un attribut unique **name:EString** hérité de la classe **Node**.

4. Classe Arc

Concernant les arcs, deux catégories sont définies : les arcs classiques et les arcs de lecture (**read**). Pour différencier ces deux types, nous utilisons une énumération **ArcKind**. Chaque **Arc** est caractérisé par un poids (**weight:EInt**), établissant ainsi une liaison entre deux nœuds via les attributs **source** et **target**.

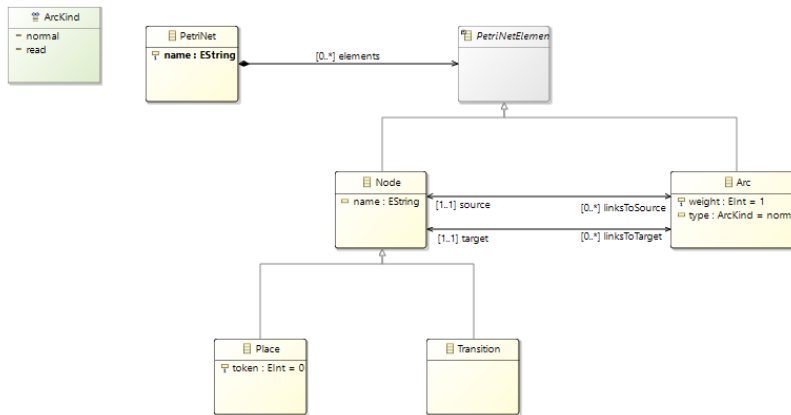


Figure 2: Métamodèle PetriNet

3 Les contraintes OCL associés aux métamodèles

Nous avons utilisé Ecore pour définir un méta-modèle de processus, mais Ecore présente des limitations dans l'expression de toutes les contraintes nécessaires. Pour résoudre cela, nous avons ajouté des contraintes en OCL à la description structurelle du méta-modèle. Cette combinaison Ecore-OCL assure une définition complète de la syntaxe abstraite du langage de modélisation des processus.

3.1 SimplePDL

Les contraintes pour SimplePDL sont les suivantes :

- **processValidName** : Le nom d'un processus doit être constitué uniquement de lettres, de chiffres ou de soulignés, et un chiffre ne peut pas être en première position.
- **uniqueActivitiesNames** : Deux activités différentes d'un même processus ne peuvent pas partager le même nom.
- **nameLength** : Le nom d'une WorkDefinition doit comporter au moins deux caractères.
- **resourceUniqueType** : Une WorkDefinition ne peut avoir qu'une seule allocation par ressource.
- **nameCharacters** : Le nom d'une activité doit être composé uniquement de lettres, de chiffres ou de soulignés, et un chiffre ne peut pas être en première position.
- **noReflexiveDependency** : Une dépendance ne peut pas être réflexive (une WorkSequence ne peut pas lier deux activités identiques).
- **successorAndPredecessorInSameProcess** : Le prédécesseur et le successeur d'une WorkSequence doivent appartenir au même processus.
- **sufficientRessourceQuantity** : Pour qu'une ressource soit définie, son nombre doit être strictement positif.
- **validName** : Le nom d'une ressource doit être constitué uniquement de lettres, de chiffres ou de soulignés, et un chiffre ne peut pas être en première position.
- **validOccurrence** : Le nombre d'occurrences dans Allocate doit être supérieur ou égal à zéro.
- **lowerAllocation** : Pour Allocate, le nombre d'occurrences doit être inférieur à la quantité de la ressource associée.

```

1 import 'SimplePDL.ecore'
2
3 package simplepdl
4
5 -- Vérifie que les noms des processus sont valides.
6 context Process
7 inv validName('Invalid name: ' + self.name):
8     self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
9
10 -- Renvoie le processus parent d'un élément de processus.
11 context ProcessElement
12 def: process(): Process =
13     Process.allInstances()
14     ->select(p | p.processElements->includes(self))
15     ->asSequence()->first()
16
17 -- Vérifie que les activités précédentes et suivantes sont dans le même processus.
18 context WorkSequence
19 inv successorAndPredecessorInSameProcess('Activities not in the same process : '
20     + self.predecessor.name + ' in ' + self.predecessor.process().name+ ' and '
21     + self.successor.name + ' in ' + self.successor.process().name
22 ):
23     self.process() = self.successor.process()
24     and self.process() = self.predecessor.process()
25
26 -- Vérifie si le nom d'une activité est unique dans le processus.
27 context Process
28 inv uniqueActivitiesNames('Duplicate activity name : ' + self.name):
29     self.processElements
30     ->select(element | element.oclIsTypeOf(WorkDefinition))
31     ->collect(element | element.oclAsType(WorkDefinition))
32     ->forAll(w1, w2 | w1 <> w2 implies w1.name <> w2.name)
33
34 -- Vérifie qu'une dépendance n'est pas réflexive
35 context WorkSequence
36 inv noReflexiveDependency('Reflexive dependency between ' + self.predecessor.name + ' and ' + self
37     self.predecessor <> self.successor
38
39 -- Vérifie que le nom d'une activités se composent d'au moins 2 caractères.
40 context WorkDefinition
41 inv nameLength('Activity name must be at least 2 characters long : ' + self.name):
42     self.name.size() >= 2
43
44 -- Vérifie que le nom d'une activité est correct.
45 context WorkDefinition
46 inv nameCharacters('Activity name must be composed of letters, numbers or underscores, a number ca
47     self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
48
49 -- Contraintes sur les ressources.
50 context Ressource
51 inv validName('Invalid name: ' + self.name): -- Vérifie que les noms des ressources sont valides.
52     self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
53 inv sufficientRessourceQuantity('Not enough ressources : ' + self.name): -- Vérifie que les ressou
54     self.quantity > 0
55
56 -- Contraintes sur l'allocation des ressources.
57 context RessourceAllocation
58 inv validOccurrence('Invalid occurrence : ' + self.occurence.toString()): -- Vérifie que les occur
59     self.occurence >= 0
60 inv lowerAllocation : -- l'allocation est inférieure à la quantité de la ressource.
61     self.occurence <= self.ressource.quantity
62
63 endpackage
64

```

Figure 3: – Les contraintes OCL du métamodèle SimplePDL

3.2 PetriNet

Les contraintes pour PetriNet sont les suivantes :

- **validName** : Le nom d'un réseau de Petri ne doit être constitué que de lettres, de chiffres, ou de soulignés, et un chiffre ne peut pas être en première position.
- **positifToken** : Chaque place doit avoir un nombre entier naturel de jetons.
- **placeUniqNames** : Deux places différentes d'un même réseau de Petri ne peuvent pas partager le même nom.
- **transitionUniqNames** : Deux transitions différentes d'un même réseau de Petri ne peuvent pas avoir le même nom.
- **positifWeight** : Le poids d'un arc doit être strictement positif.
- **transitionToPlace et placeToTransition** : Un arc doit relier une transition et une place.

```
1 import 'PetriNet.ecore'
2
3 package petrinet
4
5 -- Vérifie que les noms des processus sont valides
6 context PetriNet
7 inv validName('Invalid name: ' + self.name):
8   self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
9
10 -- Contrainte de validité des transitions
11 context Transition
12 inv validName('Invalid name: ' + self.name): -- Vérifie que le nom est valide
13   self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
14
15 -- Contrainte de validité des places
16 context Place
17 inv validName('Invalid name: ' + self.name): -- Vérifie que le nom est valide
18   self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
19 inv positifToken('Invalid number of tokens: ' + self.token.toString()): -- Vérifie que le nombre c
20   self.token >= 0
21
22 -- Contrainte de validité des arcs
23 context Arc
24 inv positifWeight('Invalid weight: ' + self.weight.toString()): -- Le poids doit être positif
25   self.weight > 0
26 inv transitionToPlace('Invalid arc'): -- Un arc doit relier une transition à une place
27   self.source.oclIsTypeOf(Transition) and self.target.oclIsTypeOf(Place)
28 inv placeToTransition('Invalid arc'): -- Un arc doit relier une place à une transition
29   self.source.oclIsTypeOf(Place) and self.target.oclIsTypeOf(Transition)
30
31 endpackage
```

Figure 4: – Les contraintes OCL du métamodèle PetriNet

4 La transformation modèle à modèle

Dans cette section, on va exploiter quelques manières de faire une transformation SimplePDL vers PetriNet en utilisant EMF/Java, ATL et Xtext.

Pour tester les différentes transformations, on va utiliser un modèle de procédé composé de 4 activités **WorkDefinition** (Conception, Rédaction de la documentation, Programmation, et Rédaction des tests) et de 5 dépendances **WorkSequence** (finishToFinish, startToStart, finishToStart, startToStart et finishToFinish).

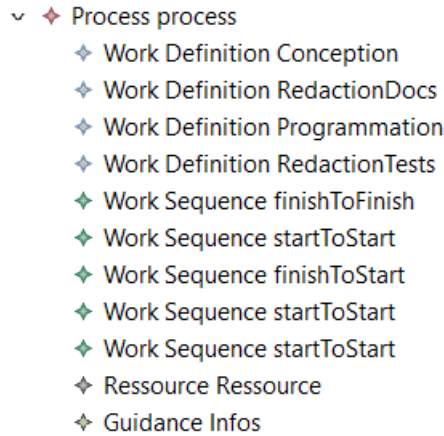


Figure 5: –Modèle CollegeApplication en SimplePDL

4.1 Transformation SimplePDL vers PetriNet en utilisant EMF/JAVA

Nous avons entrepris la définition d’une transformation du langage SimplePDL vers le réseau de Pétri en utilisant EMF/Java. Cette démarche de transformation modèle à modèle implique les étapes suivantes :

- **Chargement des Packages :** Les packages SimplePDL et PetriNet sont chargés et enregistrés dans le registre d’Eclipse.
- **Configuration de l’Input et de l’Output :** Les extensions “.xmi” sont enregistrées pour le traitement par le programme Java. Un objet `ResourceSet` est créé pour gérer les ressources EMF, définissant les fichiers d’entrée et de sortie.
- **Récupération de l’Élément Racine et Instanciation de la Fabrique :** Le premier élément du modèle `process` (élément à la racine) est extrait de la ressource SimplePDL. La fabrique correspondante est instanciée pour créer les éléments du modèle PetriNet.
- **Logique de Construction du Réseau de Petri :** Chaque `WorkDefinition` donne lieu à la création de quatre places dans le réseau (`ready`, `started`, `running`, `finished`), représentant les états d’évolution de l’activité, ainsi que deux transitions (`start`, `finish`). Des jetons sont placés dans la place “ready”, et des arcs connectent ces nœuds (`Place` et `Transition`) selon la logique du processus.

Chaque `Ressource` se traduit en une `Place` dont le nombre de jetons est initialisé en fonction de la quantité de cette ressource. Une association est établie entre cette `Place` et la transition “start” de chaque `WorkDefinition` qui utilise cette ressource. Le poids de ces arcs est déterminé par l’attribut “occurrence” de la classe `Allocate`, correspondant au nombre d’occurrences de la ressource nécessaires au `WorkDefinition`.

Chaque `WorkSequence` est représentée par un arc qui relie, en fonction du type de la `WorkSequence`, les places “started/finished” et les transitions “start/finish” entre deux activités distinctes. Ces arcs sont spécifiés comme des `readArc`, ce qui signifie qu’ils ne consomment pas le jeton pris depuis la place source.

4.2 Transformation SimplePDL vers PetriNet en utilisant ATL

La transformation modèle à modèle en ATL vise à convertir un modèle de processus en un modèle de réseau de Pétri, suivant un processus similaire à la transformation en Java. Cependant, l’approche

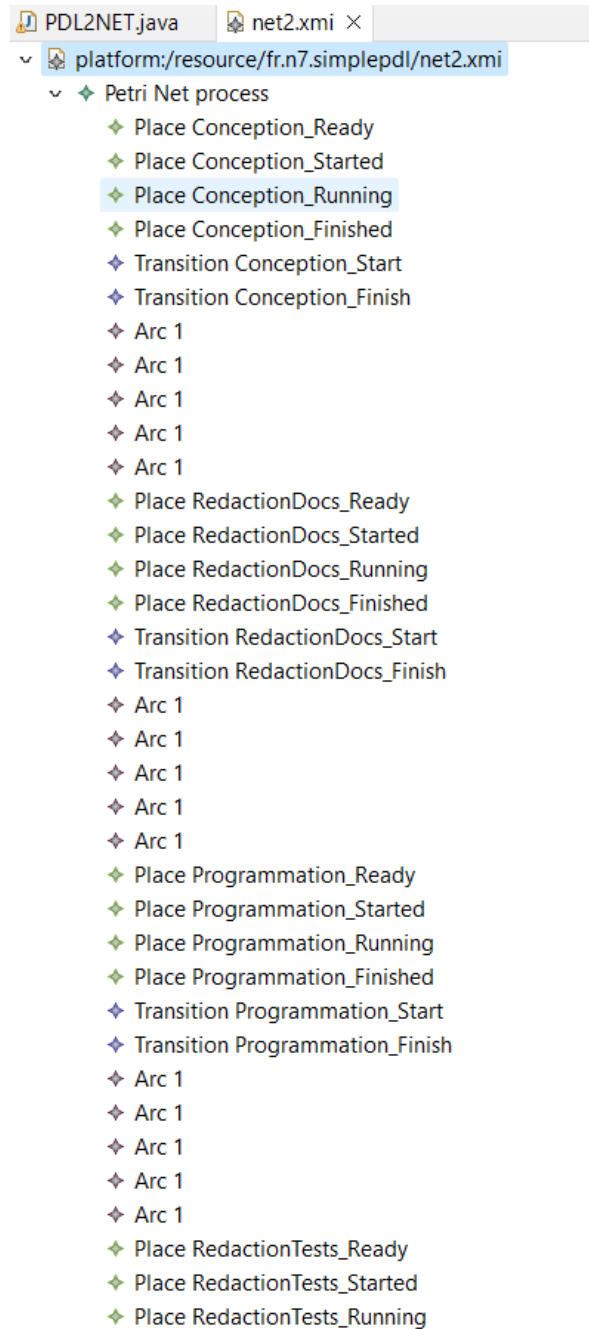


Figure 6: Modèle CollegeApplication transformé en PetriNet en utilisant EMF/JAVA

ATL se distingue par son efficacité accrue par rapport à la transformation Java.

Le processus de transformation en ATL suit les étapes suivantes, illustrées dans la Figure 17, à travers lesquelles un modèle de processus est converti en un réseau de Pétri :

1. **Traduction d'un Process en un PetriNet** : Chaque élément de type Process est traduit en un PetriNet portant le même nom.

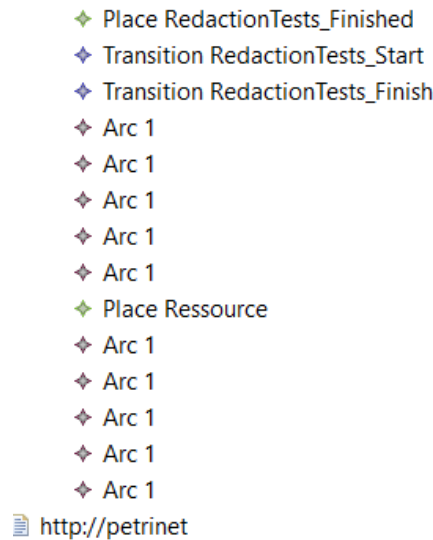


Figure 7: Modèle CollegeApplication transformé en PetriNet en utilisant EMF/JAVA

2. **Conversion des WorkDefinitions** : Chaque WorkDefinition est convertie en un réseau de Pétri composé de 5 arcs, 2 transitions, et 4 places. Ces éléments sont correctement reliés et ajoutés au réseau de Pétri en cours de construction.
3. **Conversion des WorkSequences** : Chaque WorkSequence est transformée en un arc, soigneusement relié aux WorkDefinitions appropriées. L'arc résultant est intégré dans le réseau de Pétri en construction.
4. **Conversion des Ressources** : Chaque ressource est convertie en une place et connectée de manière appropriée au reste du réseau. Tous les éléments créés au cours de cette conversion sont ajoutés au réseau global.

5 Transformation modèle à texte (M2T) avec Acceleo

5.1 Transformation PetriNet vers Tina

Nous avons mis en place une transformation de modèle à texte pour convertir un modèle de réseau de Pétri dans la syntaxe textuelle utilisée par les outils de Tina, spécifiquement avec l'extension .net. En nous appuyant sur cette syntaxe et en nous inspirant des fichiers toHTML.mtl et toDOT.mtl que nous avons précédemment élaborés lors des travaux pratiques, nous avons enrichi le template Acceleo. Cette adaptation nous a permis d'accomplir avec succès la transformation d'un réseau de Pétri dans le format attendu par Tina.

Tina.

- platform:/resource/fr.n7.simplepdl/outatl2.xml
 - Petri Net process
 - Place Conception_Ready
 - Place Conception_Started
 - Place Conception_Finished
 - Place Conception_Running
 - Transition Conception_Start
 - Transition Conception_Finish
 - Arc 1
 - Arc 1
 - Arc 1
 - Arc 1
 - Arc 1
 - Place RedactionDocs_Ready
 - Place RedactionDocs_Started
 - Place RedactionDocs_Finished
 - Place RedactionDocs_Running
 - Transition RedactionDocs_Start
 - Transition RedactionDocs_Finish
 - Arc 1
 - Arc 1
 - Arc 1
 - Arc 1
 - Arc 1
 - Place Programmation_Ready
 - Place Programmation_Started
 - Place Programmation_Finished
 - Place Programmation_Running
 - Transition Programmation_Start
 - Transition Programmation_Finish
 - Arc 1
 - Arc 1
 - Arc 1
 - Arc 1
 - Arc 1
 - Place RedactionTests_Ready
 - Place RedactionTests_Started
 - Place RedactionTests_Finished

Figure 8: Modèle CollegeApplication transformé en PetriNet en utilisant ATL

- ◆ Place RedactionTests_Running
- ◆ Transition RedactionTests_Start
- ◆ Transition RedactionTests_Finish
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Arc 1
- ◆ Place Ressource_Ressource


 <http://petrinet>

Figure 9: Modèle CollegeApplication transformé en PetriNet en utilisant ATL

- platform:/resource/fr.n7.petrinet/jours.xml
 - ◆ Petri Net semaine
 - ◆ Place Lundi
 - ◆ Place Mardi
 - ◆ Place Mercredi
 - ◆ Place Jeudi
 - ◆ Place Vendredi
 - ◆ Place Samedi
 - ◆ Place Dimanche
 - ◆ Transition LM
 - ◆ Transition MM
 - ◆ Transition MJ
 - ◆ Transition JV
 - ◆ Transition VS
 - ◆ Transition SD
 - ◆ Transition DL
 - ◆ Arc 1
 - ◆ Arc 1
 - ◆ Arc 1
 - ◆ Arc 1
 - ◆ Arc 1
 - ◆ Arc 1
 - ◆ Arc 1
 - ◆ Arc 1
 - ◆ Arc 1
 - ◆ Arc 1
 - ◆ Arc 1
 - ◆ Arc 1
 - ◆ Arc 1

Figure 10: Modèle du semaine en PetriNet

```

net semaine
pl Lundi (1)
pl Mardi (0)
pl Mercredi (0)
pl Jeudi (0)
pl Vendredi (0)
pl Samedi (0)
pl Dimanche (0)
tr LM Lundi -> Mardi
tr MM Mardi -> Mercredi
tr MJ Mercredi -> Jeudi
tr JV Jeudi -> Vendredi
tr VS Vendredi -> Samedi
tr SD Samedi -> Dimanche
tr DL Dimanche -> Lundi

```

Figure 11: Transformation du modèle du semaine en syntaxe textuelle de Tina

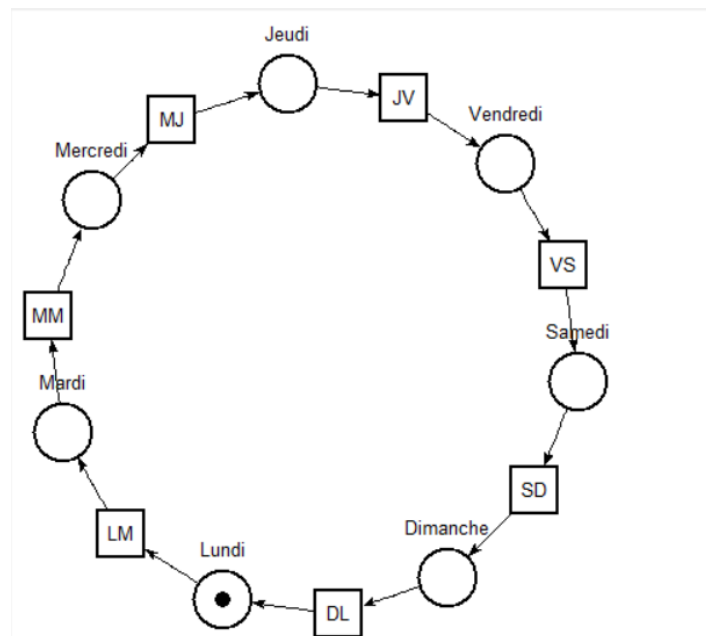


Figure 12: Visualisation graphique d'évolution du semaine par l'outil Net Draw de Tina

6 Modèle xtext

En modélisation de domaines spécifiques (DSML), la syntaxe abstraite, souvent exprimée en Ecore ou un autre langage de métamodélisation, ne se prête pas à une manipulation directe. Bien que le projet EMF d'Eclipse offre la possibilité de générer un éditeur arborescent et des classes Java pour manipuler un modèle conforme à un métamodèle, ces outils présentent des limitations pratiques lorsqu'il s'agit de créer ou de modifier des modèles. Ainsi, il est avantageux d'associer à une syntaxe abstraite une ou plusieurs syntaxes concrètes, qu'elles soient textuelles ou graphiques. Dans le contexte des syntaxes textuelles, l'outil Xtext, faisant partie du projet TMF, se distingue en permettant la définition d'une syntaxe textuelle pour un DSL tout en offrant un environnement de développement Eclipse complet, incluant un éditeur syntaxique avec des fonctionnalités telles que la coloration, la complétion, l'outline, la détection et la visualisation des erreurs, etc. Xtext repose sur un générateur d'analyseurs descendants rékursifs (LL(k)).

```
grammar fr.n7.Simplepdl with org.eclipse.xtext.common.Terminals

generate simplepdl "http://www.n7.fr/Simplepdl"

@Process :
    'process' name=ID '{'
        processelements+=ProcessElement*
    '}' ;

@ProcessElement :
    WorkDefinition | WorkSequence | Guidance | Ressource ;

@WorkDefinition :
    'wd' name=ID '{'
        '}'
        'from' LinkTopredecessors=[WorkSequence]
        'to' LinkTosuccessor=[WorkSequence]
        occurences=[RessourceAllocation];

@WorkSequence :
    'ws' linkType=WorkSequenceType
        'from' successor=[WorkDefinition]
        'to' predecessor=[WorkDefinition];

@RessourceAllocation :
    'ra' occurence=INT
    'ressource' ressource=[Ressource]
    activity=[WorkDefinition];

@Guidance :
    'infos' texte=STRING
    element=[ProcessElement];

@Ressource :
    'ressource' name=ID 'quantity' quantity=INT;

@enum WorkSequenceType :
    start2start = 's2s'
    | finish2start = 'f2s'
    | start2finish = 's2f'
    | finish2finish = 'f2f';
```

Figure 13: Le code xtext du métamodèle simplePDL

Ce fichier xtext qu'on vient de créer nous permet de créer un fichier SimplePDL.ecore qui contient notre métamodèle.

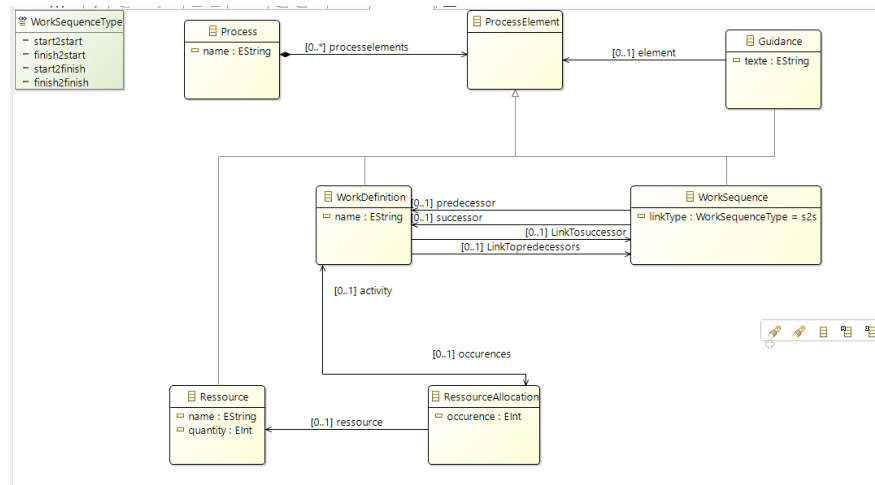


Figure 14: Diagramme SimplePDL généré à l'aide de xtext

7 Conclusion

Ce projet a offert une opportunité précieuse d'explorer et de comprendre divers processus de développement logiciel. Nous avons pu plonger plus en profondeur dans les étapes et les méthodologies impliquées dans la création et le développement de logiciels, acquérant ainsi une vision détaillée des pratiques de ces processus.