Rapport: Mini projet IDM

AIT ALLA Yahia MOUKAN Yassin BAHOU Ayman

Groupe B1

Table de matière:

1- Introduction

Ce projet consiste à développer une chaîne de vérification pour les modèles de processus SimplePDL, afin d'évaluer leur cohérence, notamment en vérifiant si le processus peut être mené à terme. Pour aborder ce problème, nous utiliserons des techniques de vérification basées sur les réseaux de Petri à l'aide de l'outil Tina. L'objectif sera donc de transformer un modèle de processus en réseau de Petri en appliquant deux méthodes.

2- Les métamodèles

2.1- SimplePDL

SimplePDL, ou "Simple Process Description Language", est un langage de métamodélisation utilisé en ingénierie logicielle pour modéliser des processus de développement. Son métamodèle de base repose sur deux concepts principaux :

- **WorkDefinition** : représente les différentes activités.
- **WorkSequence** : décrit les dépendances entre ces activités.

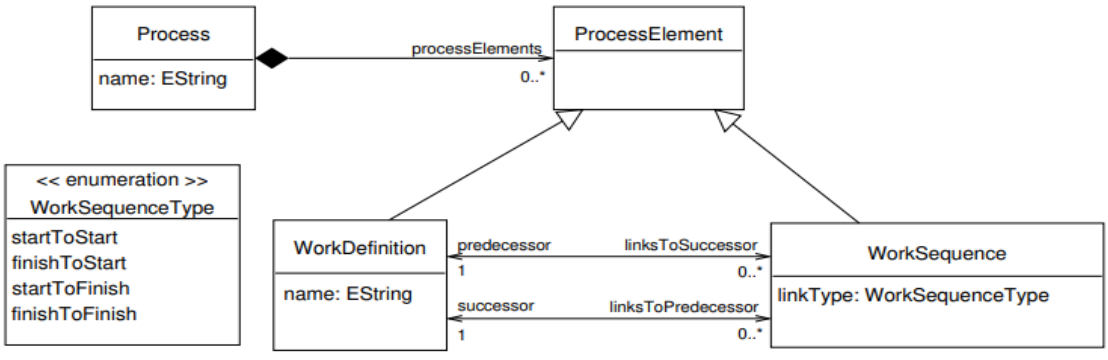


Figure 1 : métamodèle simplepdl conforme à Ecore

Dans notre projet, nous avons enrichi ce métamodèle en y ajoutant de nouveaux concepts pour mieux capturer la complexité des processus de développement. Ces ajouts incluent deux classes essentielles :

- **Ressource** : désigne les entités nécessaires à l'exécution des activités (humaines, matérielles, etc.), caractérisées par leur nom et le nombre total disponible.
- **Allocation** : représente l'allocation d'une ressource à une activité spécifique, en indiquant combien d'occurrences de cette ressource sont utilisées exclusivement par cette activité durant son déroulement.

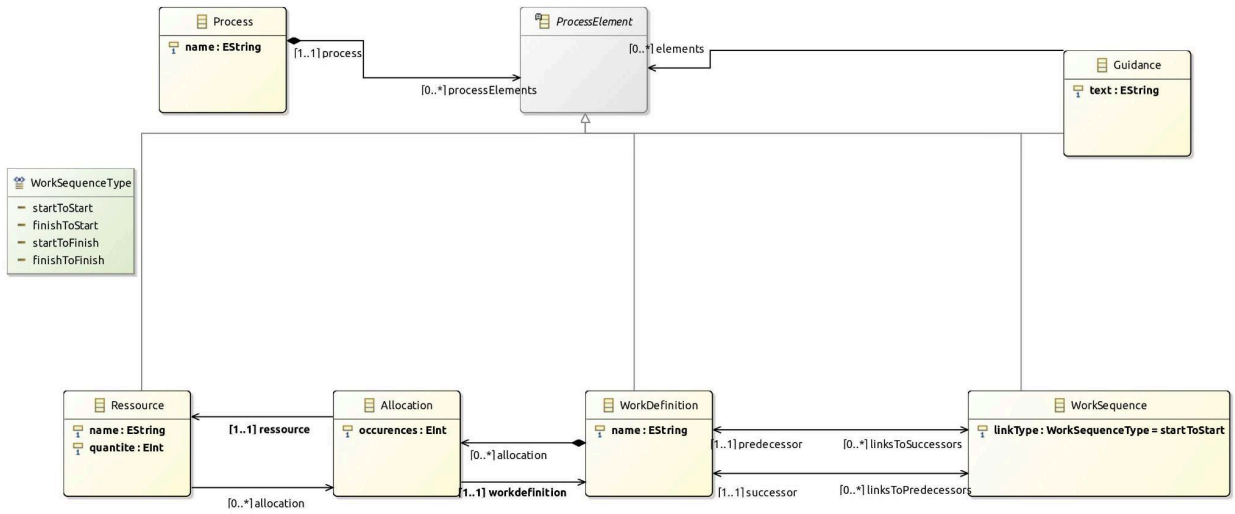


Figure : métamodèle simplepdl avec Ressources

2.2- PetriNet

On a essayé de créer un métamodèle décrivant un réseau de Petri. En se basant sur la conceptualisation

 du réseau de Petri, nous identifions les composants clés tels que les arcs, les places et les transitions.

Classe Noeud: La classe Node, caractérisée par son attribut principal (name:EString), est soit une Place ou une Transition. Chaque Node est également équipée d'une liste arcs de sortie (successors) et d'une liste d'arcs d'entrée (predecessors).

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

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

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

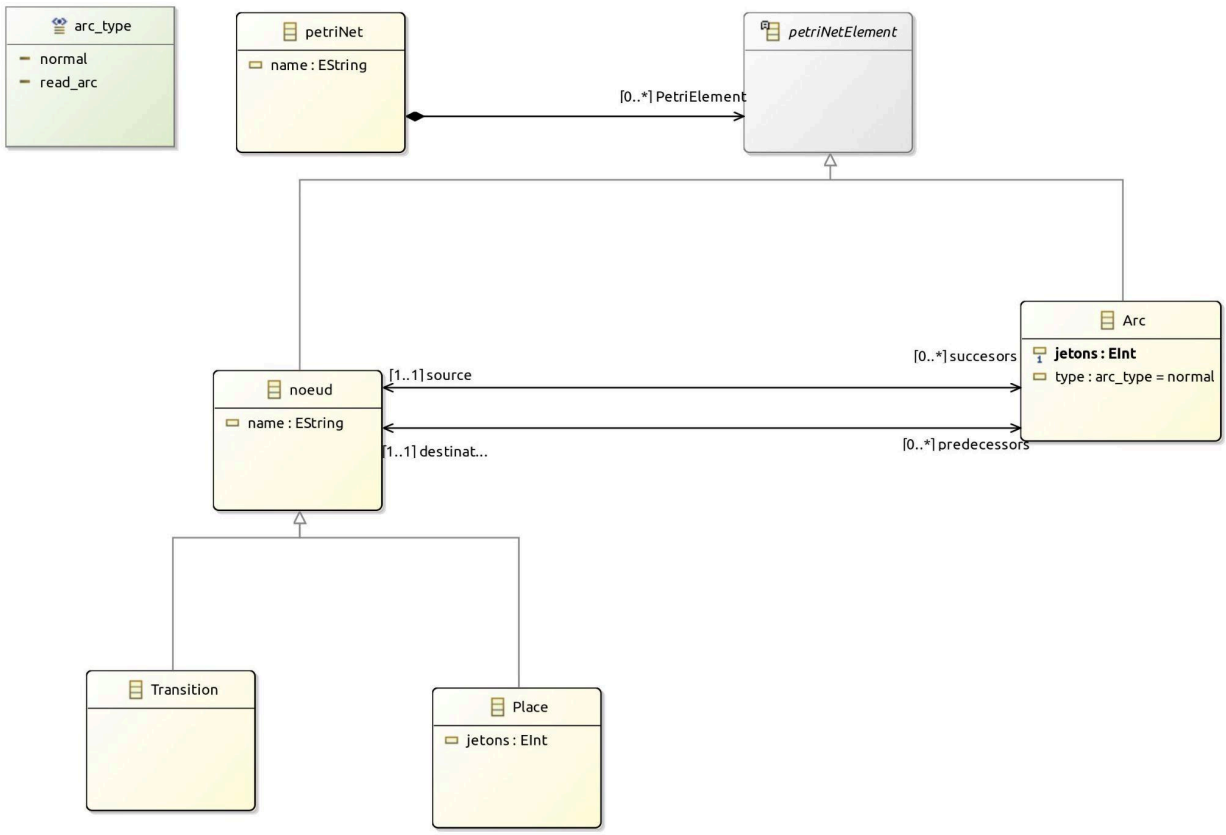


Figure : métamodèle Petrinet

3- Les contraintes OCL

3.1- Contraintes ajoutées au SimplePDL

```java
@Override
public Boolean caseProcess(simplepdl.Process object) {
    // Contraintes sur process
    this.result.recordIfFailed(
            object.getName() != null && object.getName().matches(IDENT_REGEX),
            object,
            "Le nom du process ne respecte pas les conventions Java");
    // Visite
    for (ProcessElement pe : object.getProcessElements()) {
        this.doSwitch(pe);
    }
    for (ProcessElement pe : object.getProcessElements()) {
        if (pe.eClass().getClassifierID() == SimplepdlPackage.WORK_DEFINITION) {
            for (Allocation e : ((WorkDefinition) pe).getAllocation()) {
                this.doSwitch(e);
            }
        }
    }
    return null;
}
/**
 * MÃ©thode appelÃ©e lorsque l'objet visitÃ© est un ProcessElement (ou un sous type).
 * @param object Ã©lÃ©ment visitÃ©
 * @return rÃ©sultat de validation (null ici, ce qui permet de poursuivre la visite
 * vers les classes parentes, le cas Ã©chÃ©ant)
 */
@Override
public Boolean caseProcessElement(ProcessElement object) {
    return null;
}
/**
 * MÃ©thode appelÃ©e lorsque l'objet visitÃ© est une WorkDefinition.
 * @param object Ã©lÃ©ment visitÃ©
 * @return rÃ©sultat de validation (null ici, ce qui permet de poursuivre la visite
 * vers les classes parentes, le cas Ã©chÃ©ant)
 */
@Override
public Boolean caseWorkDefinition(WorkDefinition object) {
    // Contraintes sur WD
    this.result.recordIfFailed(
            object.getName() != null && object.getName().matches(IDENT_REGEX),
            object,
            "Le nom de l'activitÃ© ne respecte pas les conventions Java");
    this.result.recordIfFailed(
            object.getProcess().getProcessElements().stream()
                .filter(p -> p.eClass().getClassifierID() == SimplepdlPackage.WORK_DEFINITION)
                .allMatch(pe -> (pe.equals(object) || !((WorkDefinition) pe).getName().contains(object.getName()))),
            object,
            "Le nom de l'activitÃ© (" + object.getName() + ") n'est pas unique");
    return null;
}
```

Figure: contraintes OCL (a),(b) et (c)

```java
@Override
public Boolean caseAllocation(Allocation object) {
    this.result.recordIfFailed(
            object.getRessource().getQuantite()>=object.getOccurences(),
            object,
            "la quantité d'une ressource doit être sup ou égale au nbr d'occurences alloué par une activité");
    return null;
}
```

Figure: Contraintes OCL (d)

a. Contrainte sur les conventions de nommage (Lignes 69-72)

- Condition : Le nom d’un processus ou d’une activité doit respecter une convention de nommage définie par une expression régulière (IDENT_REGEX).
- Message en cas de violation : "Le nom du processus ne respecte pas les conventions Java".
- Objectif : Cette contrainte garantit que chaque processus respecte les conventions de nommage standards, similaires aux conventions Java.

b. Contrainte sur les éléments du processus (Lignes 74-80 dans la première image)

- Condition : Chaque élément de processus (ProcessElement) doit être visité, et s'il appartient à la classe WorkDefinition, toutes les allocations associées doivent aussi être visitées(on a ajouté les lignes 77-80 au parcours des ProcessElements car dans notre choix de métamodèle la classe Allocation n’hérite pas de ProcessElement)
- Objectif : S'assurer que tous les éléments du processus,et aussi les éléments de type Allocation sont parcourus et validés.

c. Contrainte sur les noms des activités (Lignes 105-114)

- Condition : Le nom d'une activité (WorkDefinition) doit respecter les conventions de nommage (IDENT_REGEX), et chaque activité dans le même processus doit avoir un nom unique.
- Messages en cas de violation : "Le nom de l’activité ne respecte pas les conventions java" et "Le nom de l’activité n’est pas unique".
- Objectif : En plus de la contrainte de nommage, cette vérification garantit que toutes les activités d'un processus ont des noms uniques, ce qui évite des conflits de nommage entre activités.

d. Contrainte sur l’allocation des ressources (Lignes 176-180 dans la deuxième image)

- Condition : La quantité de ressources disponibles pour une activité doit être supérieure ou égale au nombre d'occurrences allouées à cette activité.
- Message en cas de violation : "La quantité d'une ressource doit être supérieure ou égale au nombre d'occurrences alloué par une activité".
- Objectif : Assurer que chaque activité dispose des ressources nécessaires pour être exécutée. Si une activité nécessite plus de ressources qu'il n'y en a de disponibles, cela serait considéré comme une incohérence dans le modèle.

3.2- Contraintes Réseau de Pétri

```java
package petrinet.validation;

import org.eclipse.emf.ecore.EObject;


public class petrinetValidator extends PetrinetSwitch<Boolean> {
    /**
     * Expression rÃ©guliÃ¨re qui correspond Ã  un identifiant bien formÃ©.
     */
    private static final String IDENT_REGEX = "^[A-Za-z_][A-Za-z0-9_]*$";

    /**
     * RÃ©sultat de la validation (Ã©tat interne rÃ©initialisÃ© Ã  chaque nouvelle validation).
     */
    private ValidationResultpetri result = null;

    /**
     * Construire un validateur
     */
    public petrinetValidator() {}
    public ValidationResultpetri validate(Resource resource) {
        this.result = new ValidationResultpetri();
        for (EObject object : resource.getContents()) {
            this.doSwitch(object);
        }
        return this.result;
    }
    @Override
    public Boolean casepetriNet(petrinet.petriNet object) {
        // Contraintes sur process
        this.result.recordIfFailed(
                object.getName() != null && object.getName().matches(IDENT_REGEX),
                object,
                "Le nom du process ne respecte pas les conventions Java");
        for (petriNetElement pe : object.getPetriElement()) {
            this.doSwitch(pe);
        }
        return null;
    }
    @Override
    public Boolean caseTransition(petrinet.Transition object) {
        // Contraintes sur process
        this.result.recordIfFailed(
                object.getName() != null && object.getName().matches(IDENT_REGEX),
                object,
                "Le nom du process ne respecte pas les conventions Java");

        return null;
    }
```

```java
    @Override
    public Boolean casePlace(petrinet.Place object) {
        // Contraintes sur process
        this.result.recordIfFailed(
                object.getName() != null && object.getName().matches(IDENT_REGEX),
                object,
                "Le nom du process ne respecte pas les conventions Java");
        this.result.recordIfFailed(
                object.getJetons() >= 0,
                object,
                "le nombre de jetons doi etre positif");
        return null;
    }
    @Override
    public Boolean caseArc(petrinet.Arc object) {
        // Contraintes sur process
        this.result.recordIfFailed(
                object.getJetons()>=1,
                object,
                "le nombre de jetons doi etre superieur ou égale à 1");
        this.result.recordIfFailed(
                ((object.getSource() instanceof Place)&&(object.getDestination() instanceof Transition)
                        ||((object.getSource() instanceof Transition) && (object.getDestination() instanceof Place))),
                object,
                "un arc ne relie pas deux noueds de même type");
        return null;
    }
    @Override
    public Boolean defaultCase(EObject object) {
        return null;
    }
```

a. Contrainte sur les conventions de nommage (Lignes 42,54,64)

- Condition **:** Les noms des éléments (processus, transitions, places) doivent respecter une expression régulière (IDENT_REGEX).
- Objectif **:** Assurer que les noms suivent les conventions de nommage Java pour maintenir la cohérence.

b. Contrainte sur les jetons dans une place(Lignes 67-70)

- Condition **:** Le nombre de jetons dans une place doit être supérieur ou égal à zéro.
- Objectif **:** Empêcher qu'une place ait un nombre négatif de jetons, garantissant ainsi la validité du modèle.

c. Contrainte sur le nombre minimum de jetons dans un arc(Lignes 76-79)

- Condition **:** Chaque arc doit transporter au moins un jeton.
- Objectif **:** Garantir que les arcs véhiculent des jetons et ne soient pas inutiles.

d. Contrainte sur la connexion des arcs (Lignes 80-84)

- Condition **:** Un arc doit relier une place à une transition (ou vice-versa), mais pas deux nœuds du même type.
- Objectif **:** Assurer la structure correcte du réseau, en évitant que des arcs connectent deux places ou deux transitions, ce qui serait incohérent.

4- La transformation modèle à modèle

Dans cette section, nous allons explorer différentes manières de réaliser une transformation de SimplePDL vers PetriNet en utilisant EMF/Java, ATL . Nous utiliserons un modèle de processus composé de 4 activités WorkDefinition (Conception, RedactionDoc, Programmation, et RedactionTests) et de 5 dépendances WorkSequence (finishToFinish, startToStart, finishToStart, startToStart et finishToFinish) avec une ressource(r)et une guidance(AGuidance).

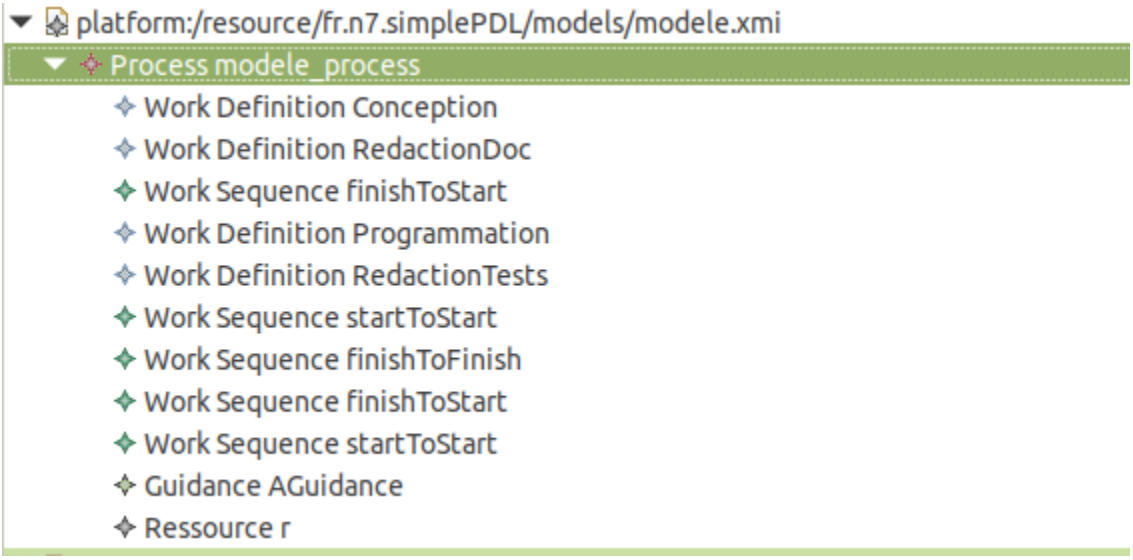


Figure : diagramme arborescent de notre modèle simplepdl de départ

4.1- Transformation de SimplePDL vers PetriNet avec EMF/Java

Nous avons commencé à définir une transformation de SimplePDL vers un réseau de Petri en utilisant EMF/Java. Cette transformation modèle à modèle suit les étapes suivantes :

— **Chargement** des packages SimplePDL et PetriNet et enregistrement dans le registre d'Eclipse.

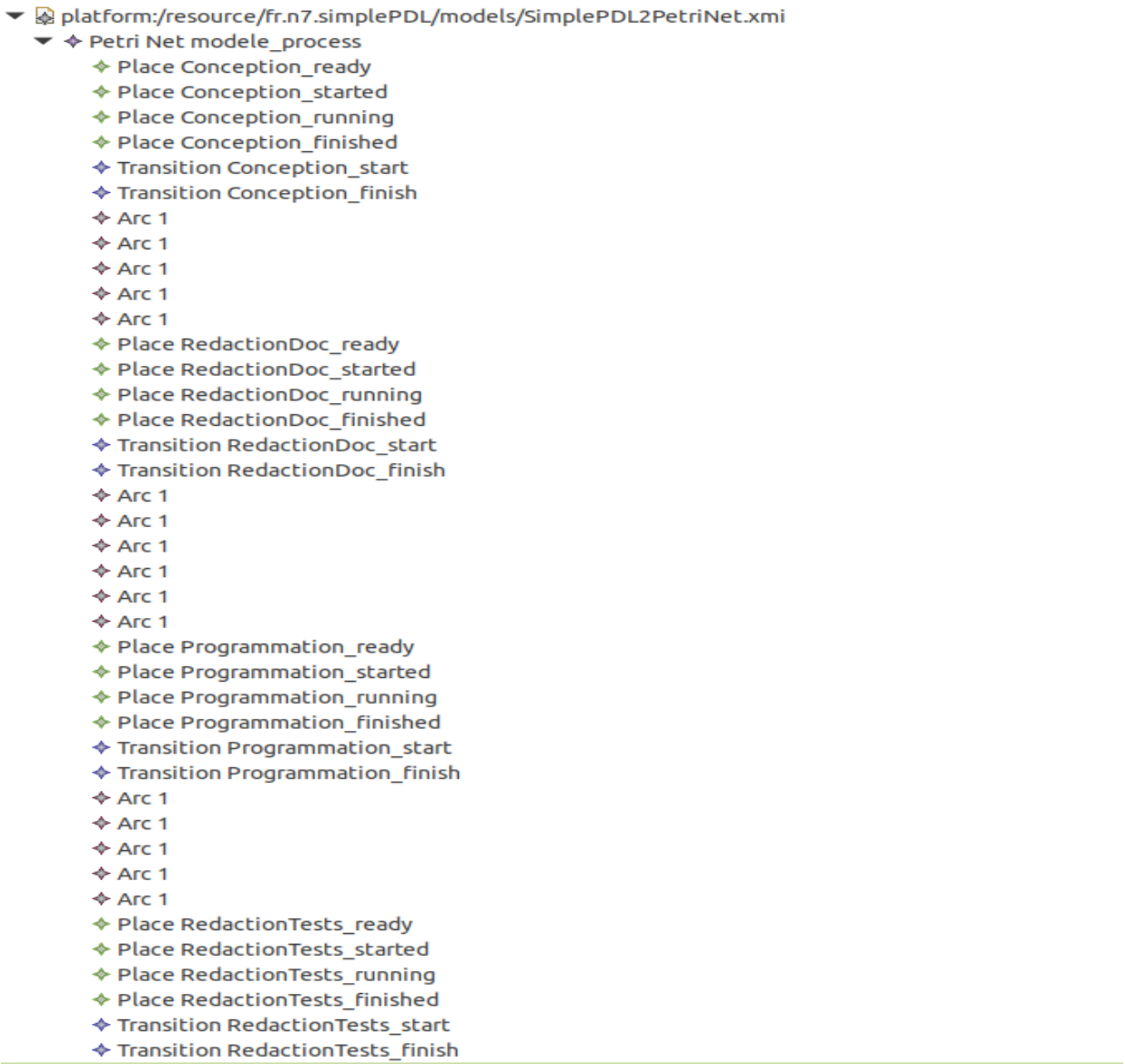
— **Configuration** des entrées (input) fournies au programme Java et des sorties (output) attendues.

— **Récupération** du premier élément du modèle de processus (élément racine) et instanciation de la fabrique.

— **Conversion** des WorkDefinitions et Resources en Places, et des WorkSequences en Arcs pour construire le réseau de Pétri.

Sur la base de ces étapes, un code simplePDL2net.java est implémenté pour effectuer la transformation d'un modèle SimplePDL en un modèle PetriNet. Ce code est fourni dans les livrables.

L'éxecution de la classe Java **SimplePDL2PetriNet.java** sur **modele.xmi** donne en sortie un modèle en extension .xmi qui modèlise la transformation du modèle de procédé en modèle de PetriNet. le résultat, modèle de sortie est le suivant:



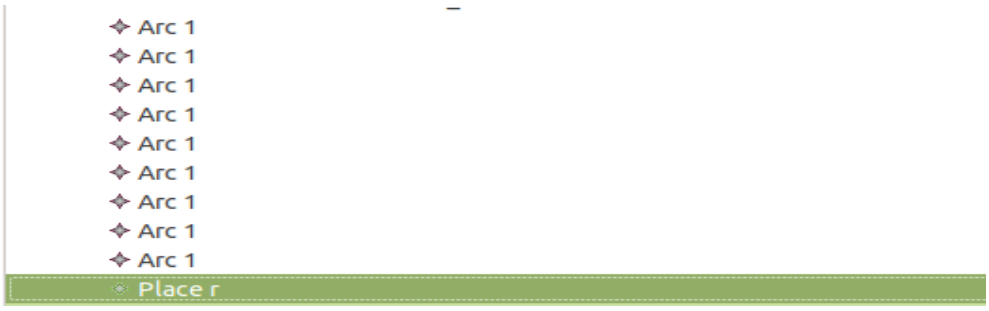


Figure : résultat de la transformation M2M du modèle initiale en utilisant java

4.2- Transformation de SimplePDL vers PetriNet avec ATL

ATL se distingue par son efficacité accrue par rapport à la transformation Java. La transformation modèle à modèle en ATL consiste à transformer un modèle de processus en un modèle de réseau de Pétri. Cette transformation passe par les mêmes étapes que la transformation Java:

1. **Traduction d'un Process en un PetriNet** : Chaque élément de type Process est transformé en un PetriNet portant le même nom.
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 entre eux et ajoutés au réseau de Pétri en cours de construction.
3. **Conversion des WorkSequences** : Chaque WorkSequence est transformée en un arc relié aux WorkDefinitions correspondantes. Cet arc est ensuite intégré dans le réseau de Pétri en cours de construction.
4. **Conversion des Ressources** : Chaque ressource est convertie en une place et connectée d'une manière appropriée au reste du réseau. Tous les éléments créés lors de cette conversion sont ajoutés au réseau global.

L'éxecution de la classe Java ***simplepdl2petri.atl*** donne en sortie un modèle en extension .xmi qui modèlise la transformation du modèle de procédé en modèle de PetriNet. le résultat de sortie est le suivant:

Figure : résultat de la transformation M2M sur le modèle initiale(modele.xmi) par ATL

5- Transformation modèle à texte (M2T) avec Acceleo

En utilisant la syntaxe textuelle employée par les outils de Acceleo, à savoir la syntaxe avec l'extension .net ,nous allons nous concentrer sur les transformations d'un modèle en texte.

Il s'agit donc d'une transformation modèle vers texte (M2T), où nous utiliserons l'outil Acceleo pour effectuer cette conversion.
Afin de valider le bon fonctionnement de cette transformation, nous continuerons à utiliser le ***modele.xmi***(figure…) qui a été transformé en réseau de petri à l'aide de ATL.
L'exécution de la classe ***Java toTina.mtl*** sur ***transfATLexemple.xmi*** donne en sortie un modèle en extension .**net** qui modélise la transformation du modèle de PetriNet en texte, le résultat de sortie est le suivant:

```
net modele_process
pl Conception_ready (1)
pl Conception_started (0)
pl Conception_running (0)
pl Conception_finished (0)
pl RedactionDoc_ready (1)
pl RedactionDoc_started (0)
pl RedactionDoc_running (0)
pl RedactionDoc_finished (0)
pl Programmation_ready (1)
pl Programmation_started (0)
pl Programmation_running (0)
pl Programmation_finished (0)
pl RedactionTests_ready (1)
pl RedactionTests_started (0)
pl RedactionTests_running (0)
pl RedactionTests_finished (0)
pl r (2)
tr Conception_start Conception_ready -> Conception_started Conception_running
tr Conception_finish Conception_running -> Conception_finished
tr RedactionDoc_start RedactionDoc_ready Conception_finished?1Conception_started?1 -> RedactionDoc_started RedactionDoc_running
tr RedactionDoc_finish RedactionDoc_running Conception_finished?1 -> RedactionDoc_finished
tr Programmation_start Programmation_ready Conception_finished?1 -> Programmation_started Programmation_running
tr Programmation_finish Programmation_running -> Programmation_finished
tr RedactionTests_start RedactionTests_ready Conception_started?1 -> RedactionTests_started RedactionTests_running
tr RedactionTests_finish RedactionTests_running -> RedactionTests_finished


```

Figure : résultat de la transformation M2T du modèle intiale(modele.xmi) avec acceleo

6- Vérification de la chaine de transformation(avec LTL):

On a écrit deux transformations modèle à texte(acceleo) la première qui traduit des propriétés vérifiant la terminaison ,et la deuxième qui traduit les invariants de notre modèle, chaque transformation génère à partir du modèle simplepdl un fichier .ltl qui décrie les propriétés. L'outil selt permettra alors de vérifier si elles sont effectivement satisfaites sur le modèle de réseau de Petri.

```
op finished = Conception_finished /\ RedactionDoc_finished /\ Programmation_finished /\ RedactionTests_finished;
[] (finished => dead);
[] <> dead ;
[] (dead => finished);
- <> finished;

```

```
ymn8673@sunfire:~/eclipse-modelling-workspace$ tina fr.n7.petriNet.exemples/modele_process.net modele_process.ktz
# net modele_process, 17 places, 8 transitions                      #
# bounded, not live, not reversible                                 #
# abstraction       count     props     psets      dead      live #
#     states          31        17        31         1         1 #
# transitions         60         8         8         0         0 #
ymn8673@sunfire:~/eclipse-modelling-workspace$ selt -p -S modele_process.scn modele_process.ktz -prelude fr.n7.LTL/modele_process.ltl
Selt version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 31 states, 60 transitions
0.003s

- source fr.n7.LTL/modele_process.ltl;
operator finished : prop
TRUE
TRUE
TRUE
FALSE
  state 0: Conception_ready Programmation_ready RedactionDoc_ready RedactionTests_ready r*2
  -Conception_start->
  state 1: Conception_running Conception_started Programmation_ready RedactionDoc_ready RedactionTests_ready r*2
  -Conception_finish->
  state 2: Conception_finished Conception_started Programmation_ready RedactionDoc_ready RedactionTests_ready r*2
  -Programmation_start->
  state 3: Conception_finished Conception_started Programmation_running Programmation_started RedactionDoc_ready RedactionTests_ready r*2
  -Programmation_finish->
  state 4: Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_ready RedactionTests_ready r*2
  -RedactionDoc_start->
  state 5: Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_running RedactionDoc_started RedactionTests_ready r*2
  -RedactionDoc_finish->
  state 6: Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_finished RedactionDoc_started RedactionTests_ready r*2
  -RedactionTests_start->
  state 7: Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_finished RedactionDoc_started RedactionTests_running RedactionTests_started r*2
  -RedactionTests_finish->
  state 8: L.dead Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_finished RedactionDoc_started RedactionTests_finished RedactionTests_started r*2
  -L.deadlock->
  state 9: L.dead Conception_finished Conception_started Programmation_finished Programmation_started RedactionDoc_finished RedactionDoc_started RedactionTests_finished RedactionTests_started r*2
  [accepting all]
0.005s
```

Figure : fichier .LTL généré pour la vérification de la terminaison et résultats de la vérification par tina et l'outil selt.

Interprétation:

L’outil selt vérifie les propriétés présentes dans le fichier .ltl et retourne True s’ils sont vérifiées sinon il retourne False avec un contre exemple.

State 0 : Le modèle commence avec tous les activités dans un état de "ready" (prêt).

State 1 : La conception a commencé, mais rien d'autre n'a démarré.

State 2 - 7 : À chaque état, on observe l'évolution des activités (Conception, Programmation, RedactionDoc, etc.) avec des transitions comme Conception_finish, Programmation_start. Ces états montrent l'enchaînement des étapes, parfois avec des activités qui terminent ou démarrent.

State 8 - 9 : Ce sont les états marqués par "dead" où le processus est dans un état de mort (fin d'exécution), ce qui montre qu’on peut arriver à un état ou toutes les activités sont dans l’état finished .

7- L’éditeur graphique de SimplePdl (Sirius):

Les syntaxes graphiques sont très importantes pour la visualisations des modèles d’une manière plus lisible et plus agréable, c’est pour cela qu’on a utilisé l’outil Sirius d’Eclipse basé sur les technologies Eclipse Modeling (EMF) permettant d’engendre un éditeur graphique à partire d’un modèle Ecore.

Dans notre cas, notre point de départ était la syntaxe abstraite du DSML définie par le modèle ecore déja présenté, on est arrivé à définir une syntaxe graphique pour ce métamodèle, dans laquelle il est possible de définir graphiquement pour un Process, les WorkDefinitions, les WorkSequences ainsi que les Ressources

sauf qu'on peut pas Allouer les ressources à partir de la barre à outils mais à travers l'éditeur arborescent.Comme le montre la figure ci-dessous.

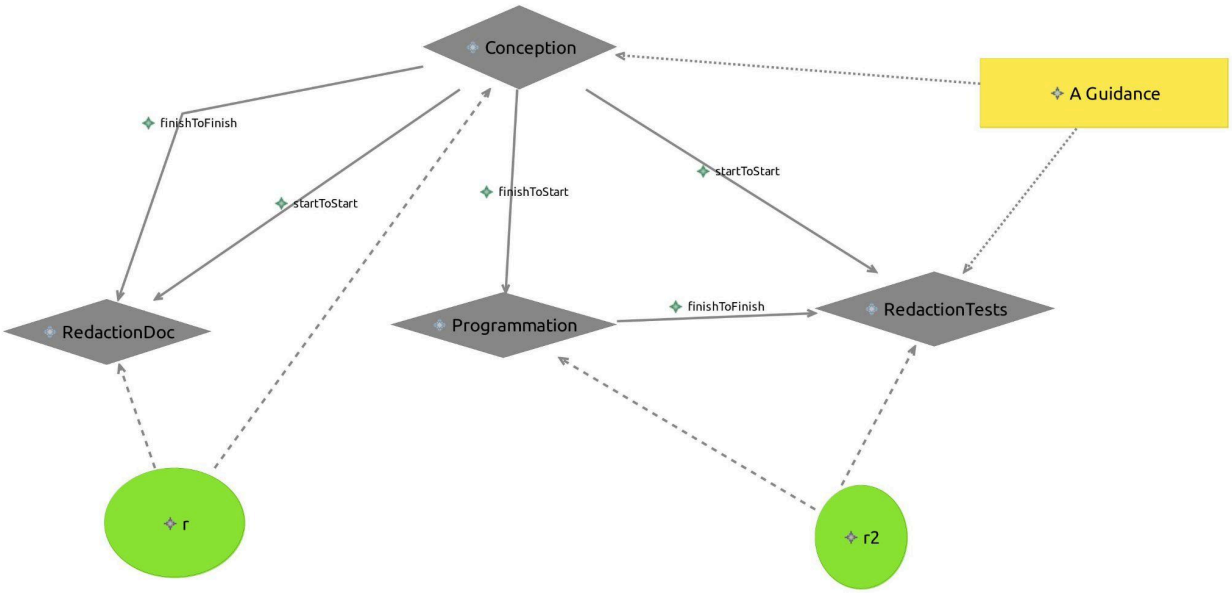


Figure : l'éditeur graphique simplepdl

8 - Définition d'une syntaxe concrète textuelle avec Xtext

La manipulation des modèles conforment aux métamodèles avec l'éditeur arborescant d'EMF ou les classes java n'est pas pratique, d'où l'intêret d'une syntaxe abstraite permettant la construction ou la modification facile et accessible pour ces modèles. On a vu d'abord une version graphique de cette syntaxe abstraite avec l'outil Sirius mais il existe aussi une version textuelle avec l'outil Xtext.

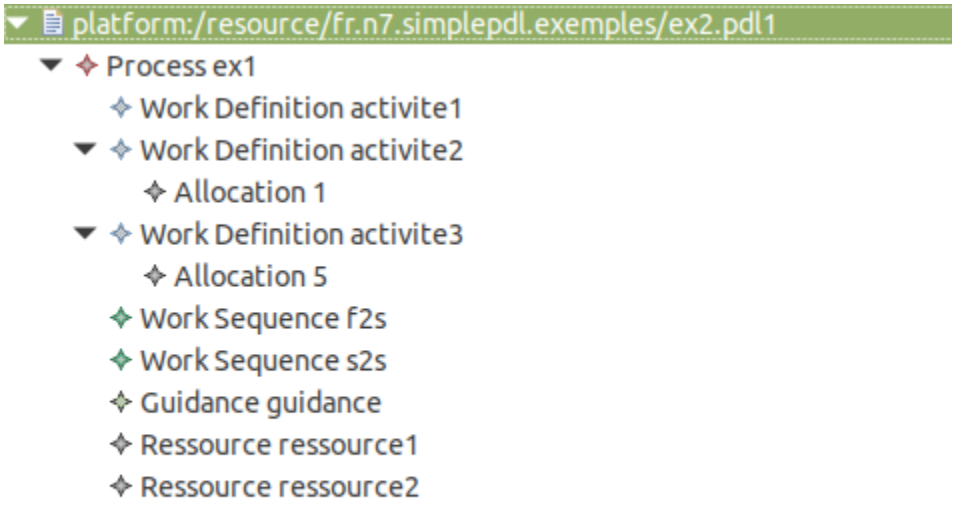
Xtext appartient au TMF (Textual Modeling Framework) et permet d'avoir un éditeur syntaxique avec beaucoup d'options: coloration, complétion, détection des erreurs... . Le fichier PDL.xtext contient la description XText pour la syntaxe associée à SimplePDL, qui engendre la syntaxe suivante pour un exemple donné:

```
process ex1 {
wd activite1
wd activite2 {
    alloc 1 of ressource2
}
wd activite3 {
    alloc 5 of ressource1
}
ws f2s from activite1 to activite2
ws s2s from activite2 to activite3
note "guidance"
res ressource1 qte 5
res ressource2 qte 2
}
```

On remarque bien l'existence d'une coloration par exemple pour : process, wd , ws, f2s, from to , note(guidance), res(ressource) qte(quantité de la ressource) et alloc of.

Pour l'allocation de ressource on avait choisi d'inclure ou non dans chaque activité la ressource qu'elle alloue en indiquant le nombre d'occurence(l'entier après le alloc), après pour la définition de la ressource on utilise le mot clé res puis on met son nom après le mot clé qte puis on met la quantité de cette ressource.

Le code dans l'image ci dessus permet de générer la structure arborescente suivante:



9- Bilan sur notre Projet (ce qui marche et ce qui ne marche pas et changements apportés après L'oral)

Dans notre projet **tous fonctionne bien sauf** comme déjà mentionné dans la partie de l'éditeur graphique **Sirius**, le fait **d'allouer des ressources à travers la barre des outils**, pourtant on l'avait fait en utilisant l'éditeur arborescent comme le montre la figure de l'éditeur graphique.

Modification apportées après oral : on a pu ajouter **l'allocation des ressources** dans la partie **Xtext**, et du coup cette partie marche bien maintenant.

10- Conclusion:

Ce mini-projet nous a donné l'occasion de mettre en pratique les notions vues en cours et en TP, en travaillant sur tous les éléments de la chaîne de transformation de SimplePDL vers PetriNet. Cela nous a aidé à comprendre l'importance d'une bonne conception des modèles et des possibilités de vérification offertes par l'outil Tina. Ce mini-projet nous a également aidés à mieux comprendre et interpréter les erreurs fréquentes rencontrées dans Eclipse.