Ingénierie Dirigée par les Modèles

Mini-Projet IDM

Groupe L34-02

Élèves :

THEVENET Louis

SABLAYROLLES Guillaume

Octobre 2024

Table des matières

1. Méta-Modèles

1.1. SimplePDL

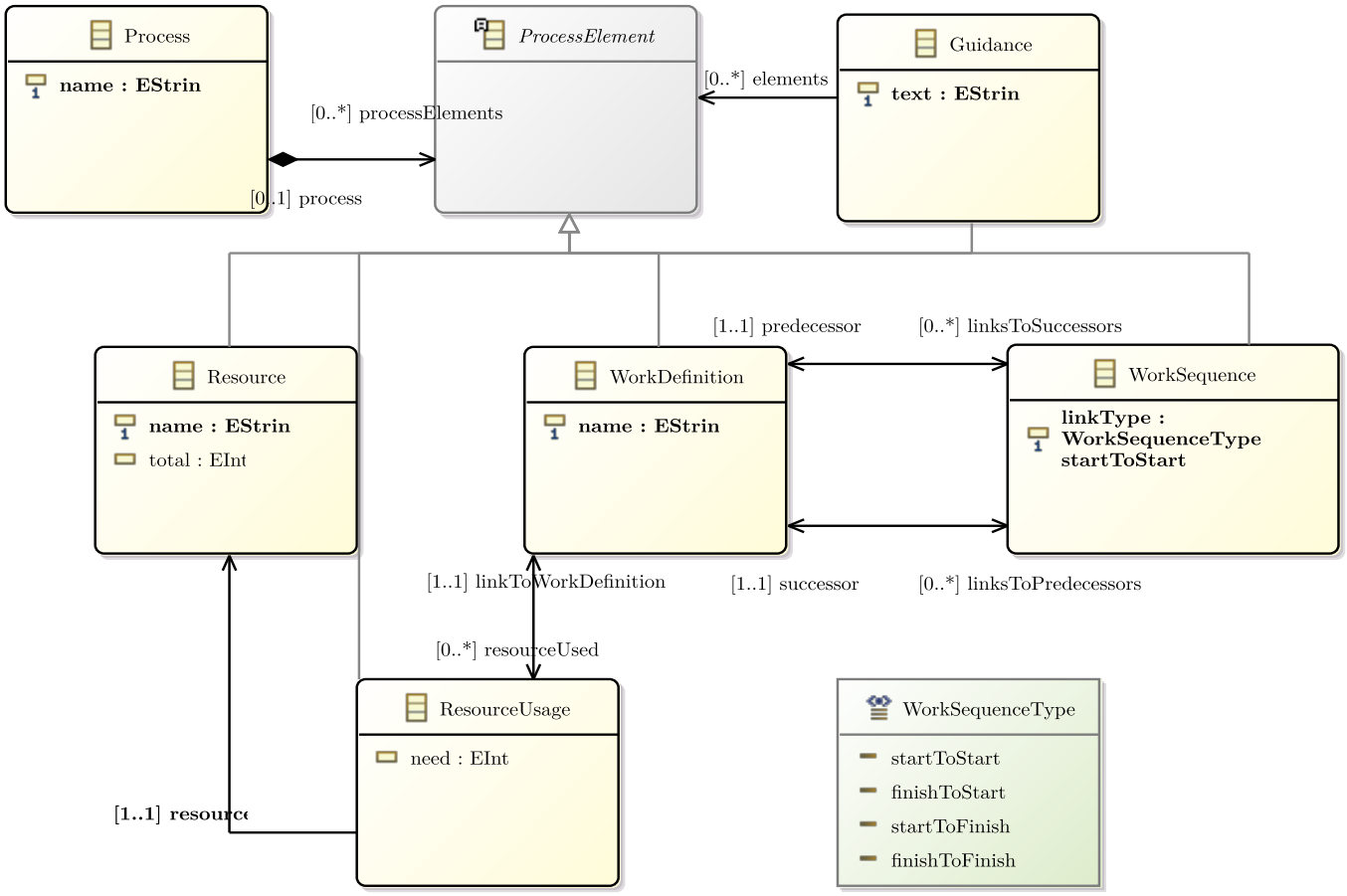

Fig. 1. – Méta-Modèle SimplePDL

Nous sommes parti du méta-modèle fourni et avons rajouté les ressources. Une ressource est défini comme une EClass `Resource` ayant :

- un nom
- et un nombre total d'éléments disponible (`total`)

Une `WorkDefinition` utilise une `Resource` en ajoutant une référence à une `ResourceUsage` qui contient :

- une référence à la `Resource` en question
- la quantité demandée (`need`)

1.2. PetriNet

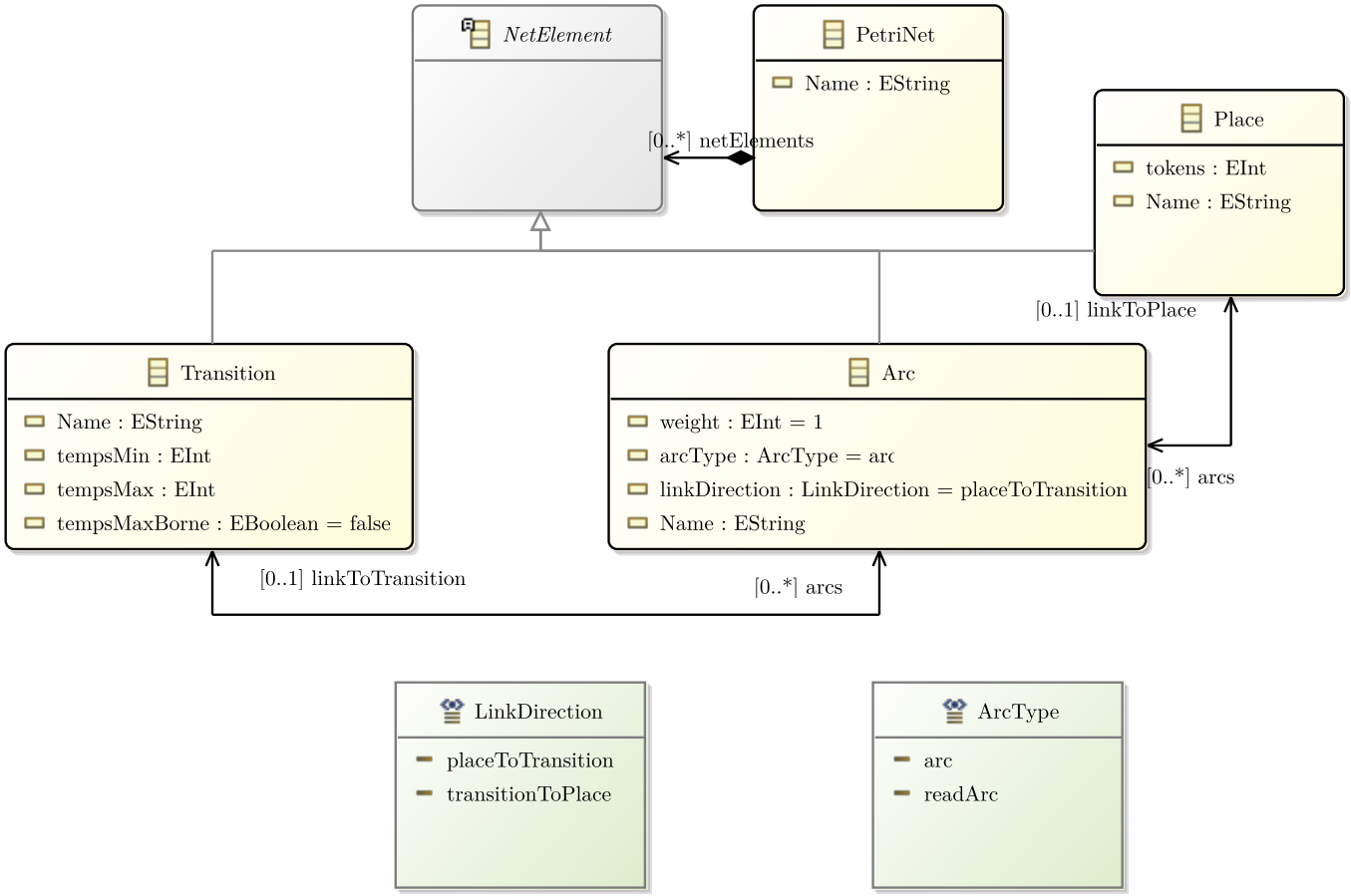

Fig. 2. – Méta-Modèle PetriNet

Un `PetriNet` est constitué de `NetElement`. Ces éléments sont les `EClass` `Place`, `Transition` et `Arc`.

Une `Place` est définie par un nom et un nombre de jetons (`tokens`).

Une `Transition` est définie par un nom.

Un `Arc` contient :

- un nom
- un poids
- une référence vers une `Place`
- une référence vers une `Transition`
- Une `LinkDirection` (soit `placeToTransition`, soit `transitionToPlace`)
- Un `arcType` (soit `arc`, soit `readArc`).

Nous avons également ajouté des attributs `tempsMin`, `tempsMax` et `tempsMaxBorne` pour ajouter la notion de temps aux `Transition`.

Ce méta-modèle permet de s'assurer que les `Arc` ne relient jamais deux `Transition` ou deux `Place`.

2. Transformations Modèle à Modèle

2.1. SimplePDL vers PetriNet

Principe de la transformation d'un modèle de processus en réseau de Petri:

- Un élément `Process` devient un élément `PetriNet`
- Une `WorkDefinition` devient 4 places `ready` (avec 1 jeton), `started`, `running` et `finished` et deux transitions `start` et `finish` reliées par des arcs
- Une `WorkSequence` devient un read-arc entre une place de l'activité précédente (`started` ou `finished`) et une transition de l'activité cible (`start` ou `finish`)

Transformation des ressources :

- Une `Resource` devient une place dont le nombre de jetons est égal au nombre de ressources initialement disponibles
- Une `ResourceUsage` devient deux arcs avec pour poids le nombre de ressources demandé :
  - De la place représentant la `Resource` utilisée à la transition `start` de la `WorkDefinition`
  - De la transition `finish` de la `WorkDefinition` à la place représentant la `Resource` utilisée

2.1.1. Fichier d'entrée

Pour illustrer les transformations, nous utiliserons l'exemple de modèle de processus suivant.

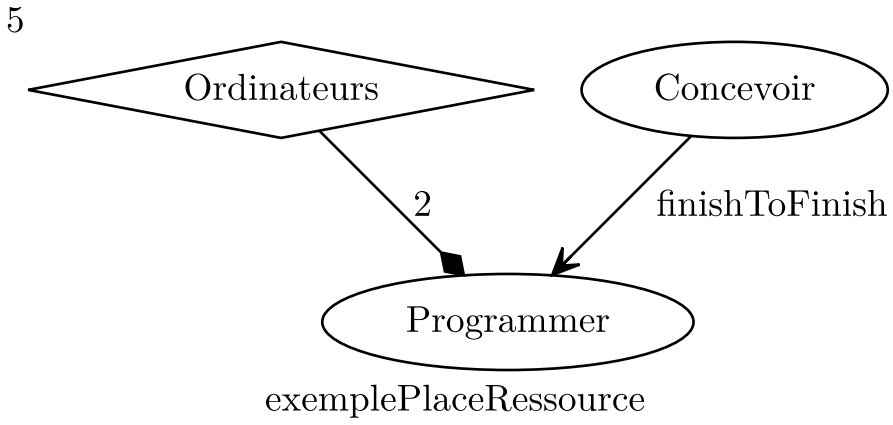

Fig. 3. – Modèle de processus simple avec une ressource

2.1.2. Java/EMF

On réalise un premier programme de transformation en Java (voir `SimplePDL2PetriNet.java`)

Lors de cette transformation, on traite les `ProcessElement` dans cet ordre :

1. `Resource`
2. `WorkDefinition`
   - `ResourceUsage` (on traite les `ResourceUsage` attachés à la `WorkDefinition` courante)
1. `WorkSequence`

La Fig. 4 représente le réseau de Petri en sortie du programme.

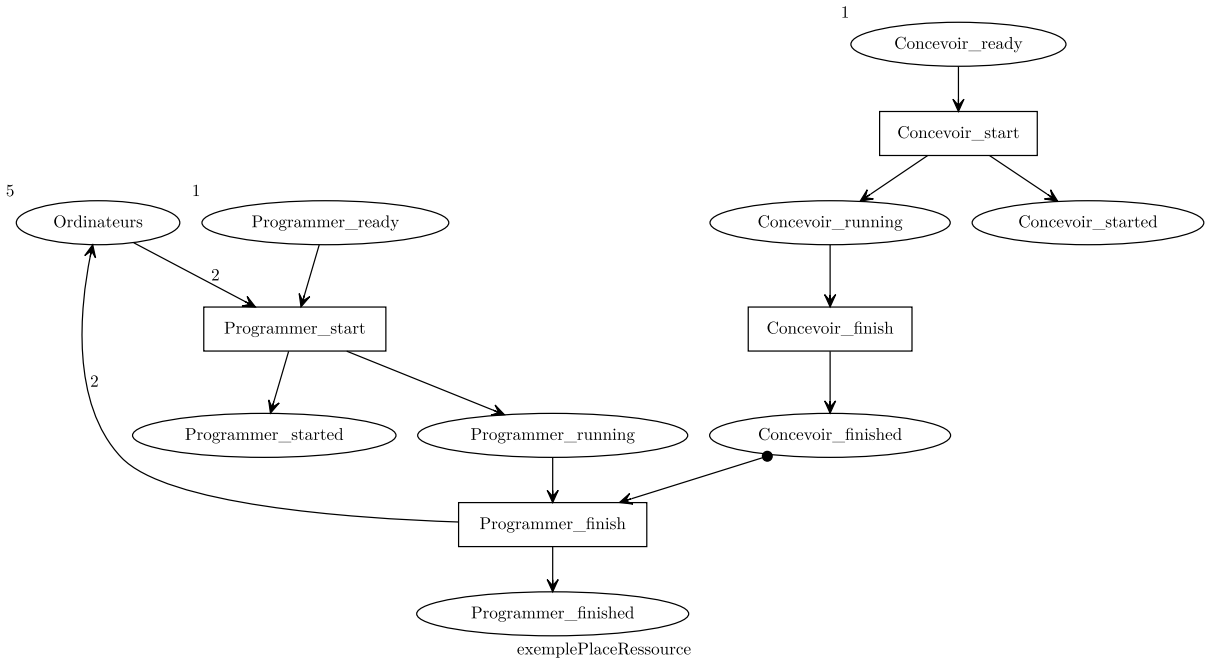

Fig. 4. – Réseau de Petri résultant de la transformation par Java

On distingue aisément les différents sous-réseaux de Petri associés aux `WorkDefinition` ainsi que la `Ressource` et les arcs qui la relient au sous-réseau associé à *Programmer*

2.1.3. ATL

On réalise également la même transformation à l'aide d'ATL (voir `SimplePDL2PetriNet.atl`)

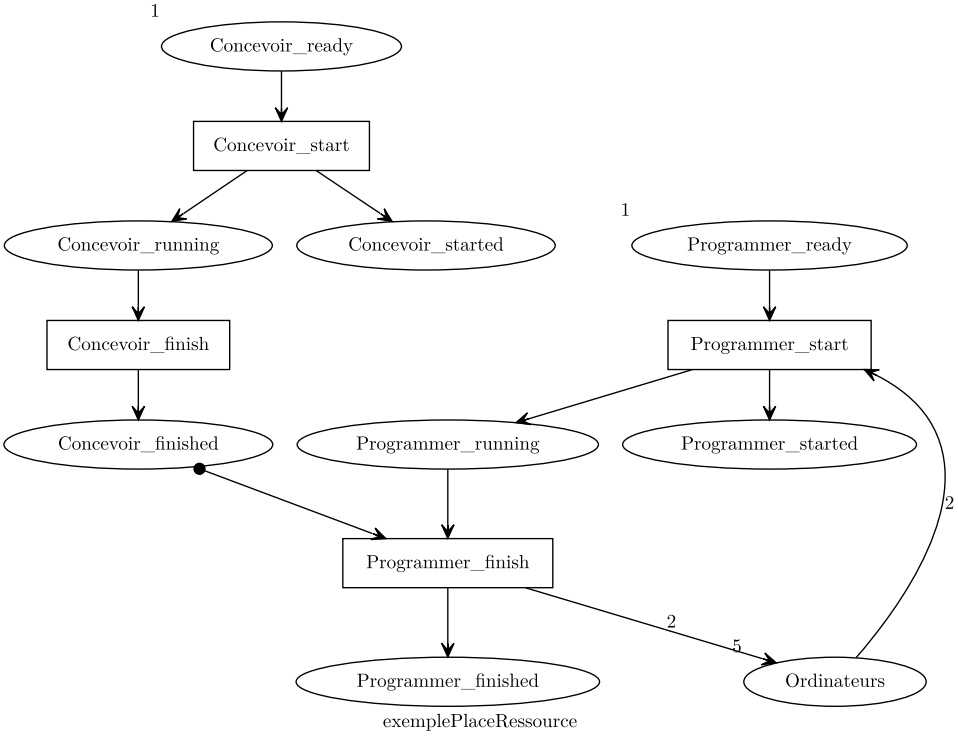

Fig. 5. – Réseau de Petri résultant de la transformation par ATL

L'emplacement des noeuds n'est plus exactement le même mais les graphes sont bien identiques.

3. Transformation Modèle à Texte

Nous avons réalisé plusieurs transformations modèle à texte :

- `SimplePDL2Dot`
- `PetriNet2Tina`
- `PetriNet2Dot`

Les images précédentes ont été réalisées à partir des transformation vers le format DOT.

3.1. `SimplePDL` vers Dot

Pour chaque `Resource`, on déclare un `node` avec la forme `diamond`, le même nom et le nombre total de ressources.

Pour chaque `WorkSequence` on déclare un arc entre le prédecesseur et le successeur (les `node` associés aux `WorkDefinition` seront générés automatiquement)

Pour chaque `ResourceUsage` on déclare un arc entre la ressource et la `WorkDefinition` et une tête avec la forme `diamond`.

Voir Fig. 3 pour un exemple.

3.2. `PetriNet` vers Tina

Le format NET est une traduction presque directe du méta-modèle `PetriNet`, ce qui rend la transformation très simple. (voir `PetriNet2Tina.mtl`)

On parcourt d'abord les `Place` pour les déclarer, puis on déclare chaque transition en ajoutant si besoin les contraintes temporelles et en traitant le cas des read-arcs.

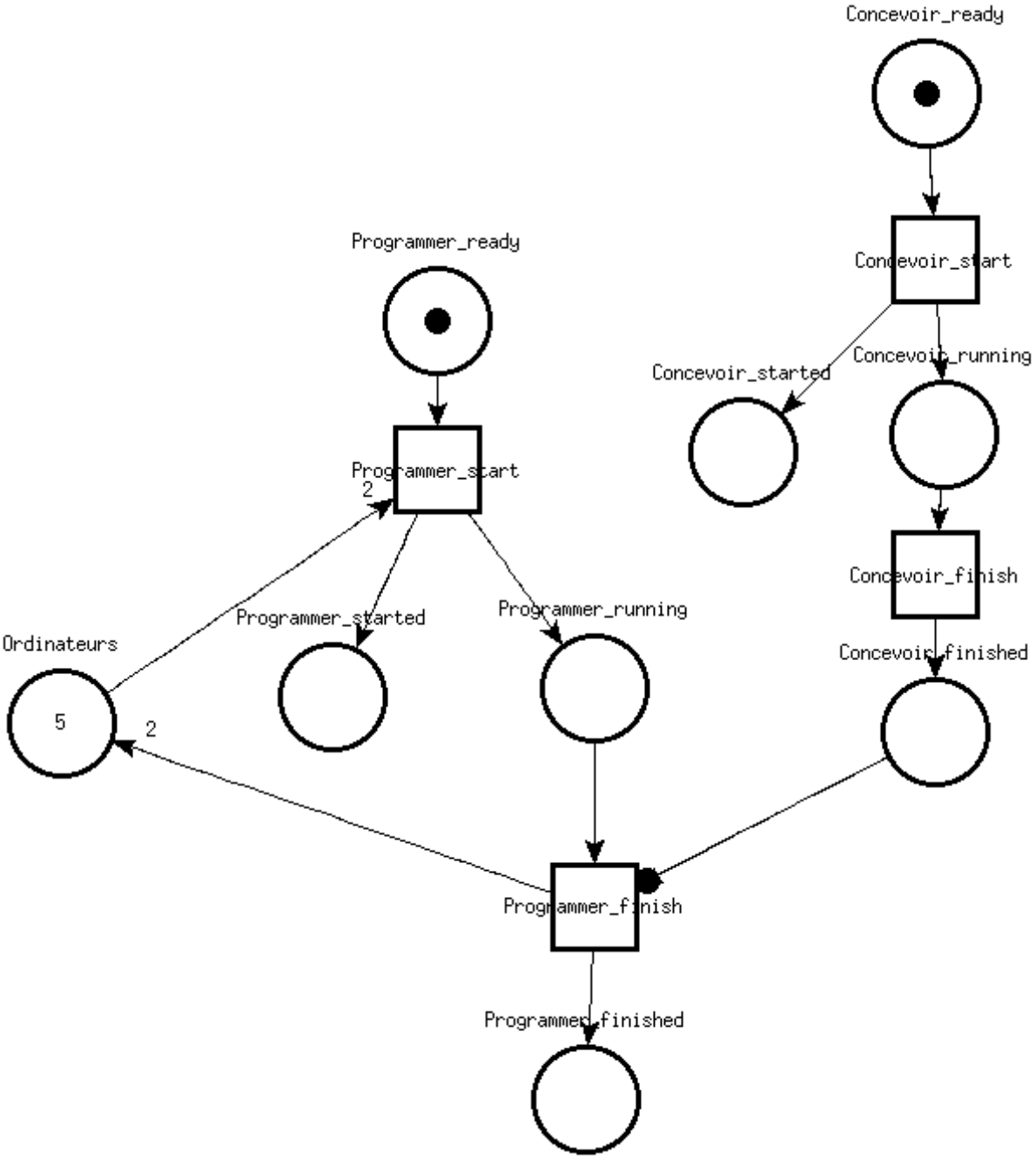

Fig. 6. – Capture d'écran de Tina affichant le fichier NET résultat

3.3. `PetriNet` vers Dot

Pour chaque `Place`, on déclare un `node` avec le même nom et le nombre de jetons associés.

Pour chaque `Transition`, on déclare un `node` avec le même nom et les éventuelles contraintes temporelles.

On déclare ensuite les arcs en traitant les read-arcs.

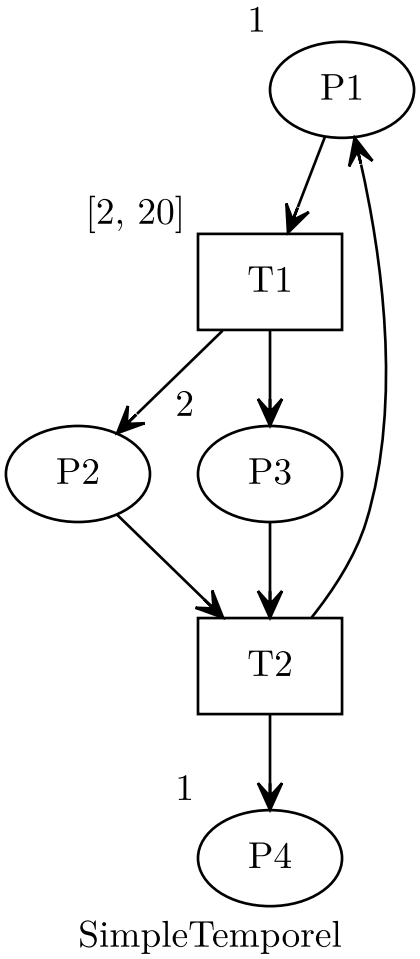

Fig. 7. – Exemple résultant de la transformation en DOT

4. Transformation Texte à Modèle de `SimplePDL`

Nous avons décidé de partir de la troisième grammaire du sujet de TP et d'y ajouter le support pour les ressources.

```
process : ex1

resources: Humains:5; Ordis:5;
workdefinitions : a; b; c;
resourceusages: a:Humains=5; b:Humains=5; b:Ordis=2;
worksequences : a s2s b; b f2f c; c s2s a;
```

Liste 1. – Exemple de fichier conforme à la grammaire

On réalise ensuite la transformation du méta-modèle `SimplePDL3` issu de cette grammaire en `SimplePDL`.

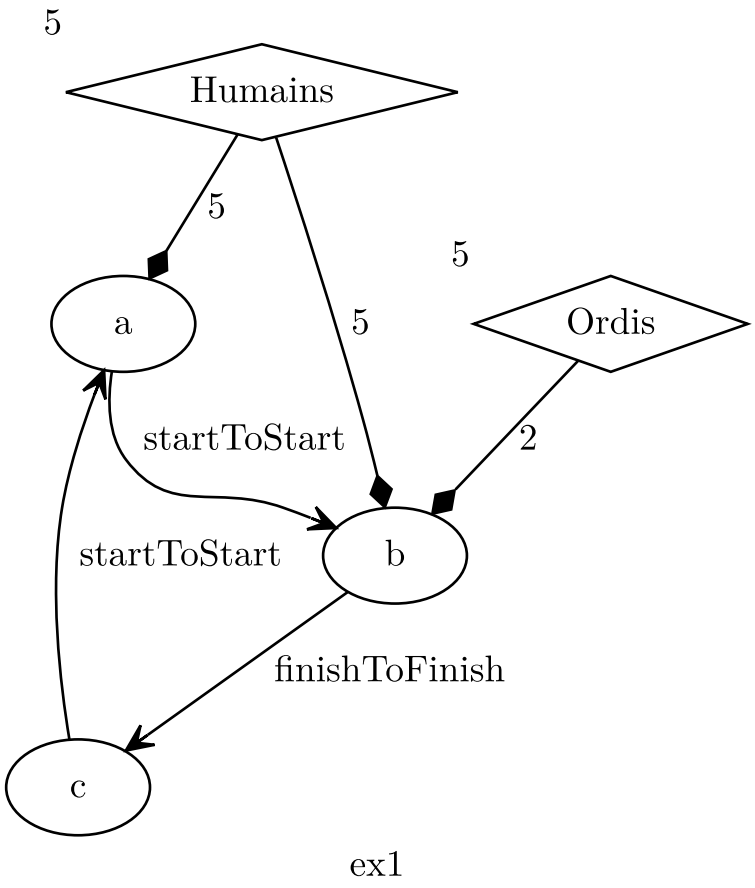

Fig. 8. – Résultat de `SimplePDL3` → `SimplePDL` → `DOT`

5. Edition graphique

Pour obtenir une visualisation et une édition plus agréable des modèles, nous avons développé une syntaxe graphique à l'aide de Sirius.

L'éditeur ainsi obtenu nous permet de modifier des processus (`SimplePDL`) par édition graphique et ainsi rendre l'expérience utilisateur plus agréable.

- Les `WorkDefinition` sont représentées par des ovales gris
- Les `WorkSequence` sont représentées par flèches de couleurs différentes selon `WorkSequenceType`
- Les `Resource` sont représentées par des losanges bleus
- Les `ResourceUsage` sont représentées par des flèches bleues
- Les `Guidance` et leurs liens sont représentées par des rectangles et flèches

oranges

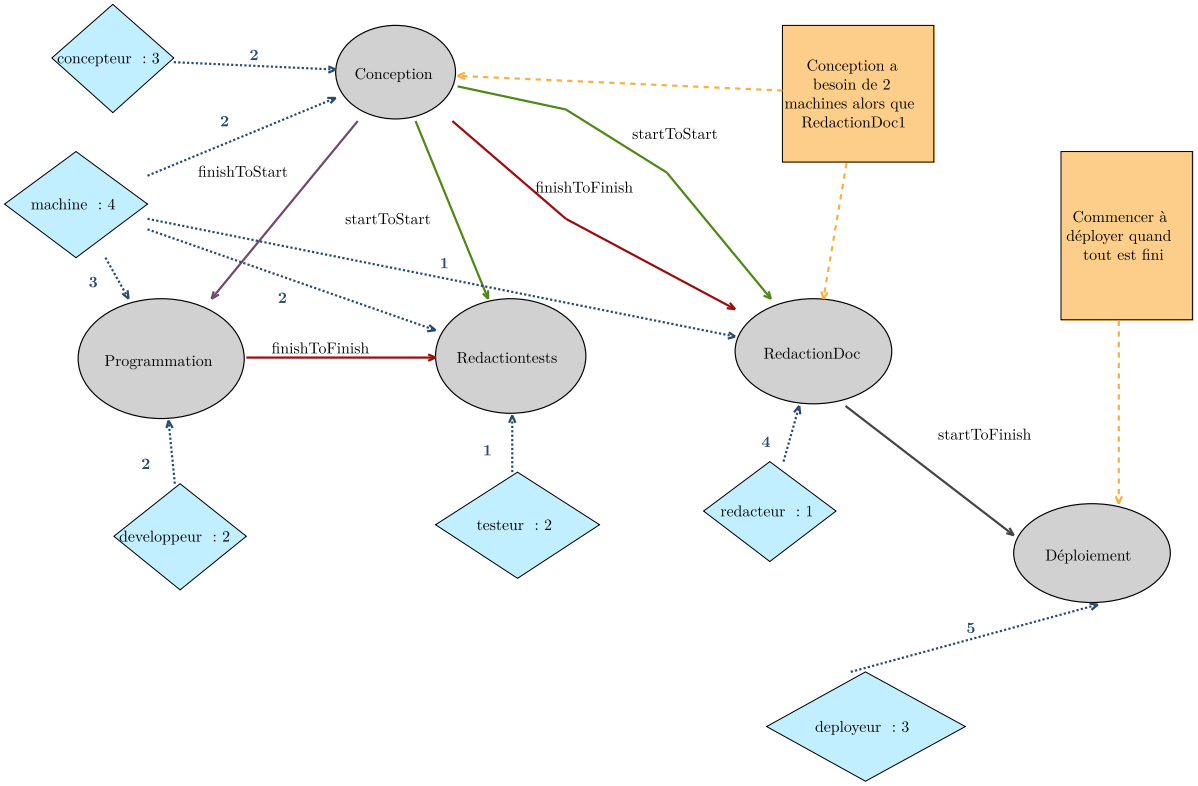

Fig. 9. – Edition de `pdl-sujet-ressources.xmi` avec Sirius

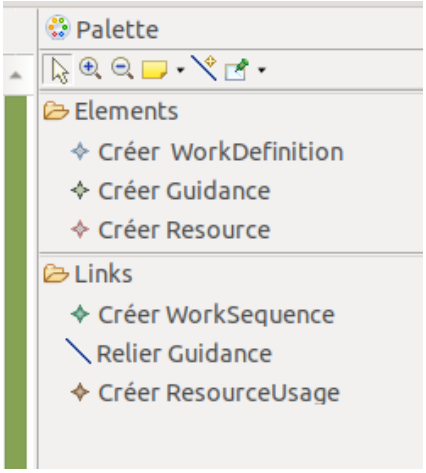

Fig. 10. – Palette de création dans Sirius

6. Vérification de terminaison et d'invariants

On souhaite vérifier certaines propriétés sur les modèles de processus. On réalise pour celà deux transformations vers le format LTL.

On réutilisera l'exemple présenté en Fig. 3.

6.1. Terminaison

Un processus se termine si toutes ses activités se terminent, c'est-à-dire qu'il y a un jeton dans chaque place `finished` associées aux `WorkDefinition`

```
op finished = (Programmer_finished  /\ Concevoir_finished  /\  T);
[] (finished => dead);
[] <> dead;
[] dead => finished;
- <> finished;
```

Liste 2. – Fichier LTL en sortie de transformation

6.2. Invariants

Les invariants de processus sont les mêmes que ceux de réseaux de Pétri. Une activité ne peut être en cours et en même temps terminée, ses états sont exclusifs.

```
[] (Programmer_finished + Programmer_running + Programmer_ready = 1);
[] (Concevoir_finished + Concevoir_running + Concevoir_ready = 1);
```

Liste 3. – Fichier LTL en sortie de transformation

On aurait également pu vérifier que dès qu'une place `started` possède un jeton, celui-ci y reste pour toujours, ou qu'une place modélisant une `Resource` récupérera forcément son nombre initial de ressources.

7. Application et conclusion

Nous avons réalisé différentes transformation : M2M, T2M, M2T et graphique. Nous allons appliquer ces transformations à l'exemple donné en Fig. 11.

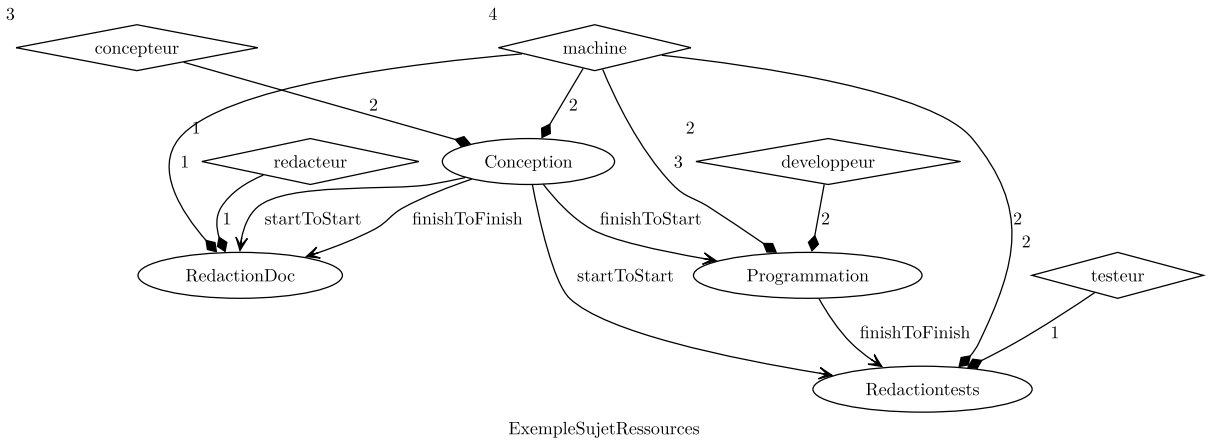

Fig. 11. – Représentation d'un modèle de processus avec ressources

Ce modèle de processus conforme au méta-modèle `SimplePDL` a été transformé en modèle conforme à `PetriNet`. Nous avons ensuite produit un fichier NET par la transformation `Petrinet` vers Tina.

Nous avons généré les propriétés LTL décrivant la terminaison et les invariants que nous voulons vérifier sur ce modèle de processus.

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

```
[] (Redactiontests_finished + Redactiontests_running + Redactiontests_ready = 1);
[] (Conception_finished + Conception_running + Conception_ready = 1);
[] (Programmation_finished + Programmation_running + Programmation_ready = 1);
[] (RedactionDoc_finished + RedactionDoc_running + RedactionDoc_ready = 1);
```

Les invariants sont bien vérifiés, ce qui confirme le bon fonctionnement de notre réseau de Petri et de la transformation sur cet exemple.

Quant à la terminaison, `selt` exhibe un contre-exemple rapporté au Liste 4.

```
1  TRUE
2  TRUE
3  TRUE
4  FALSE
5  state 0: Conception_ready Programmation_ready RedactionDoc_ready
   Redactiontests_ready concepteur*3 developpeur*2 machine*4 redacteur testeur*2
6  -Conception_start->
   state 1: Conception_running Conception_started Programmation_ready
7  RedactionDoc_ready Redactiontests_ready concepteur developpeur*2 machine*2
   redacteur testeur*2
8  -Conception_finish->
   state 2: Conception_finished Conception_started Programmation_ready
9  RedactionDoc_ready Redactiontests_ready concepteur*3 developpeur*2 machine*4
   redacteur testeur*2
10 -RedactionDoc_start->
   state 26: Conception_finished Conception_started Programmation_ready
11 RedactionDoc_running RedactionDoc_started Redactiontests_ready concepteur*3
   developpeur*2 machine*3 testeur*2
12 -RedactionDoc_finish->
   state 27: Conception_finished Conception_started Programmation_ready
13 RedactionDoc_finished RedactionDoc_started Redactiontests_ready concepteur*3
   developpeur*2 machine*4 redacteur testeur*2
14 -Redactiontests_start->
   state 28: L.dead Conception_finished Conception_started Programmation_ready
15 RedactionDoc_finished RedactionDoc_started Redactiontests_running
   Redactiontests_started concepteur*3 developpeur*2 machine*2 redacteur testeur
16 -L.deadlock->
   * [accepting] state 29: L.dead Conception_finished
17 Conception_started Programmation_ready RedactionDoc_finished RedactionDoc_started
   Redactiontests_running Redactiontests_started concepteur*3 developpeur*2 machine*2
   redacteur testeur
18 -L.deadlock->
   state 29: L.dead Conception_finished Conception_started Programmation_ready
19 RedactionDoc_finished RedactionDoc_started Redactiontests_running
   Redactiontests_started concepteur*3 developpeur*2 machine*2 redacteur testeur
```

Liste 4. – Résultat de `selt` sur la terminaison du modèle de processus

Cette sortie n'est pas très lisible, dans ce contre-exemple, seules les activités `Conception` et `RedactionDocs` ont terminé, `RedactionTests` est en cours et `Programmer` n'a jamais démarré.

`Programmer` nécessite 3 ressources `Programmeurs`, or, seules 2 sont disponibles. Le réseau est bloqué dans cet état.