Ingénierie Dirigée par les Modèles

Mini-Projet IDM

Groupe L34-02

Élèves :

THEVENET Louis

SABLAYROLLES Guillaume

Octobre 2024

Contents

1. Méta-Modèles

1.1. SimplePDL

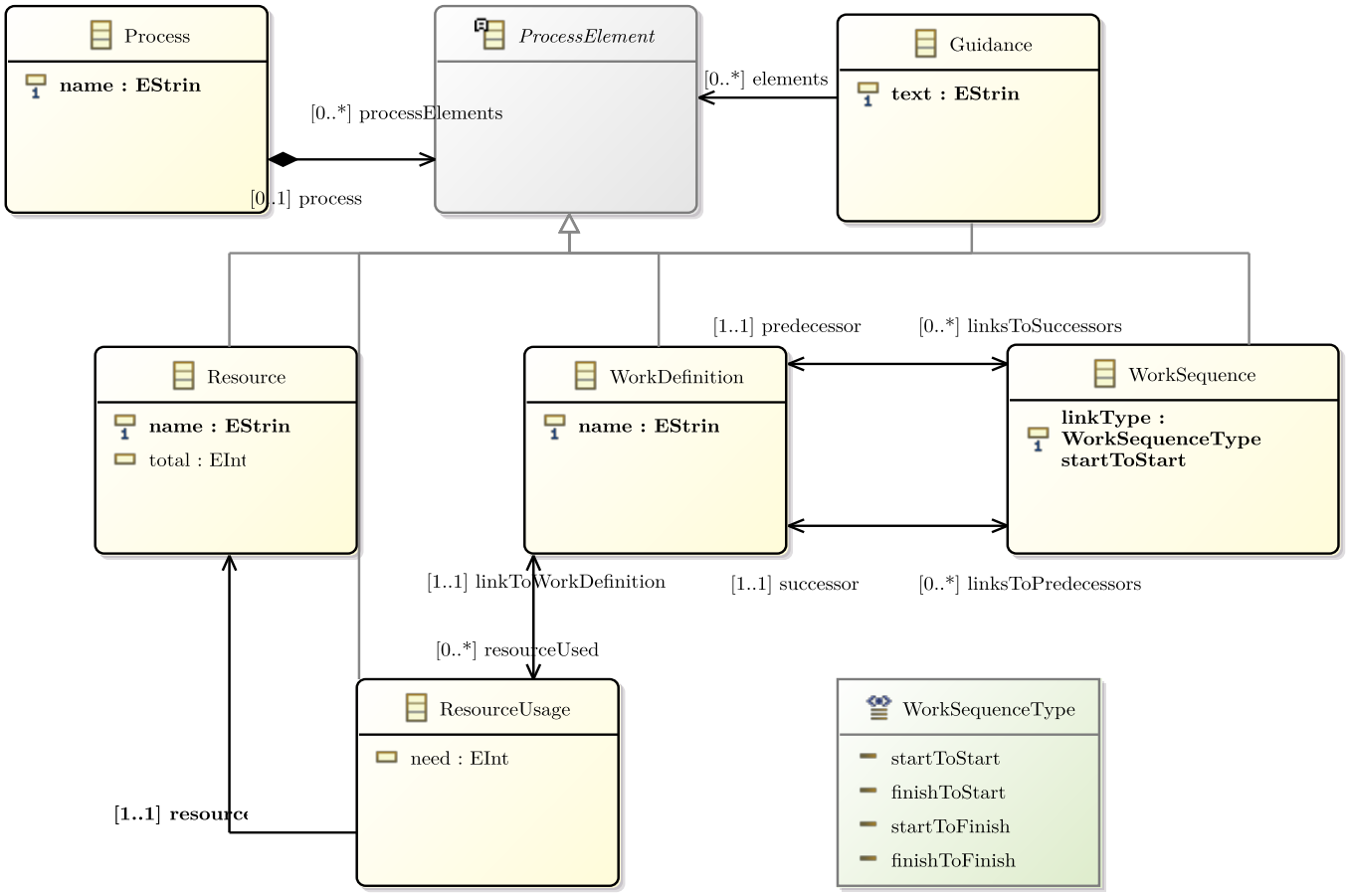

Figure 1: Méta-Modèle de SimplePDL

A partir du méta-modèle fourni, nous avons rajouté l'utilisation des 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 :

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

1.2. PetriNet

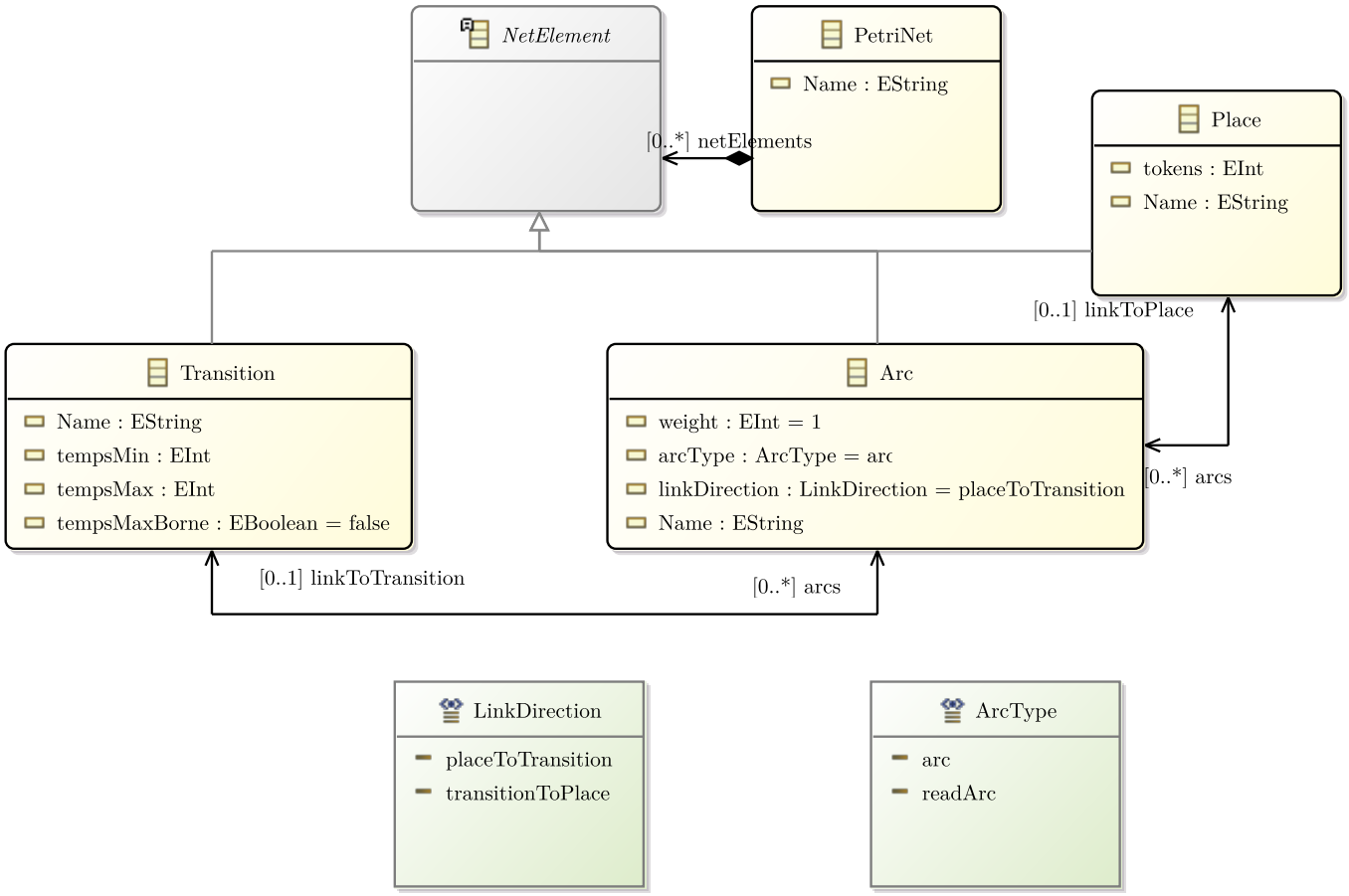

Figure 2: Méta-Modèle de PetriNet

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. Transformation 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.

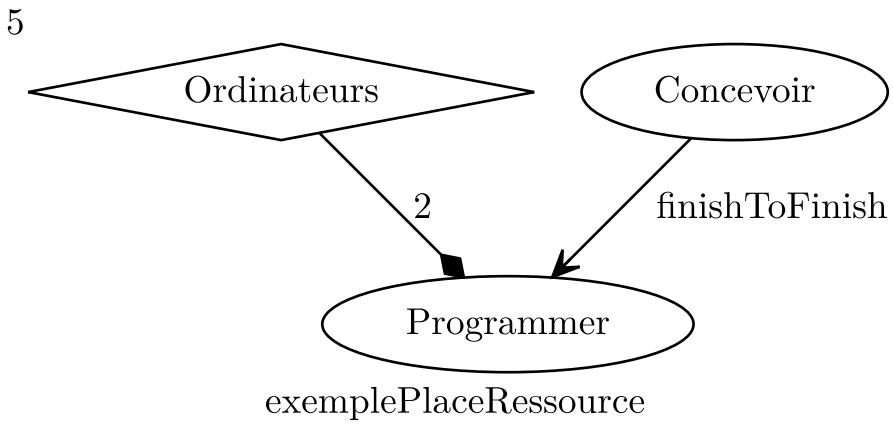

Figure 3: Modèle de processus simple avec une ressource

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)
3. `WorkSequence`

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

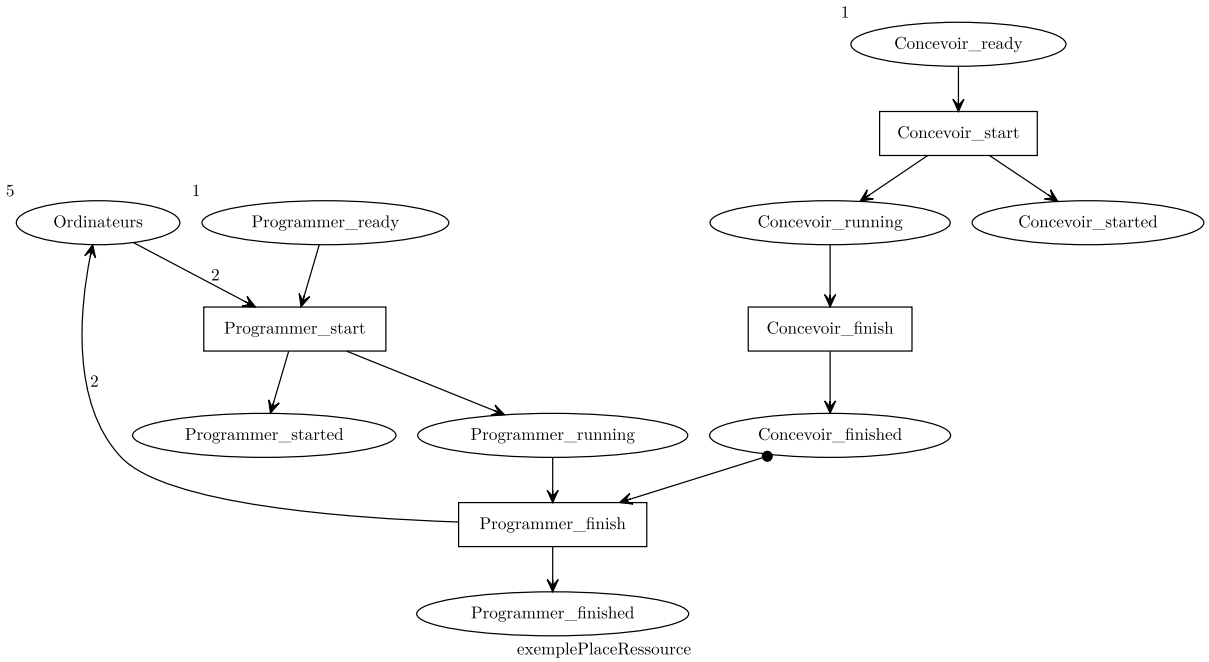

Figure 4: Réseau de Petri résultant de la transformation par Java

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`)

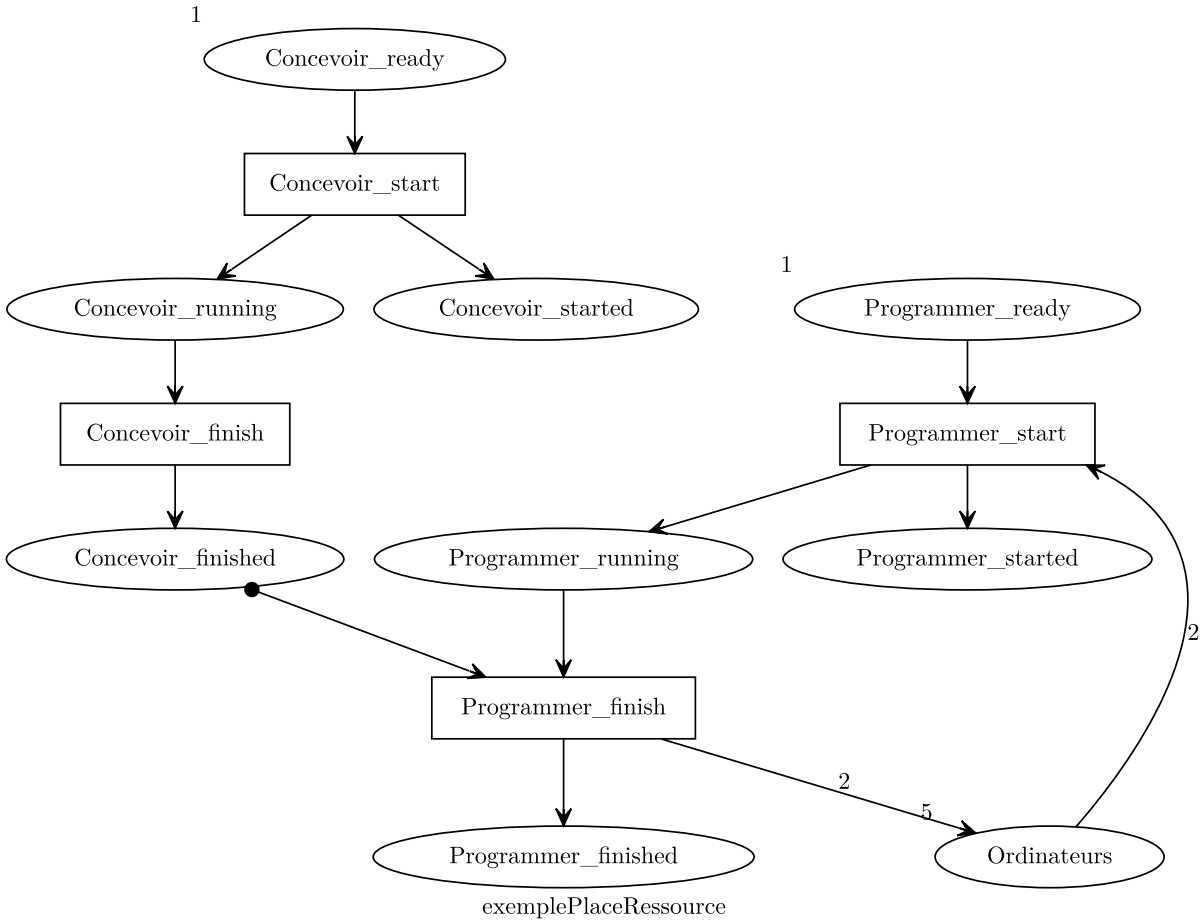

Figure 5: Réseau de Petri résultant de la transformation par ATL

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 Figure 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.

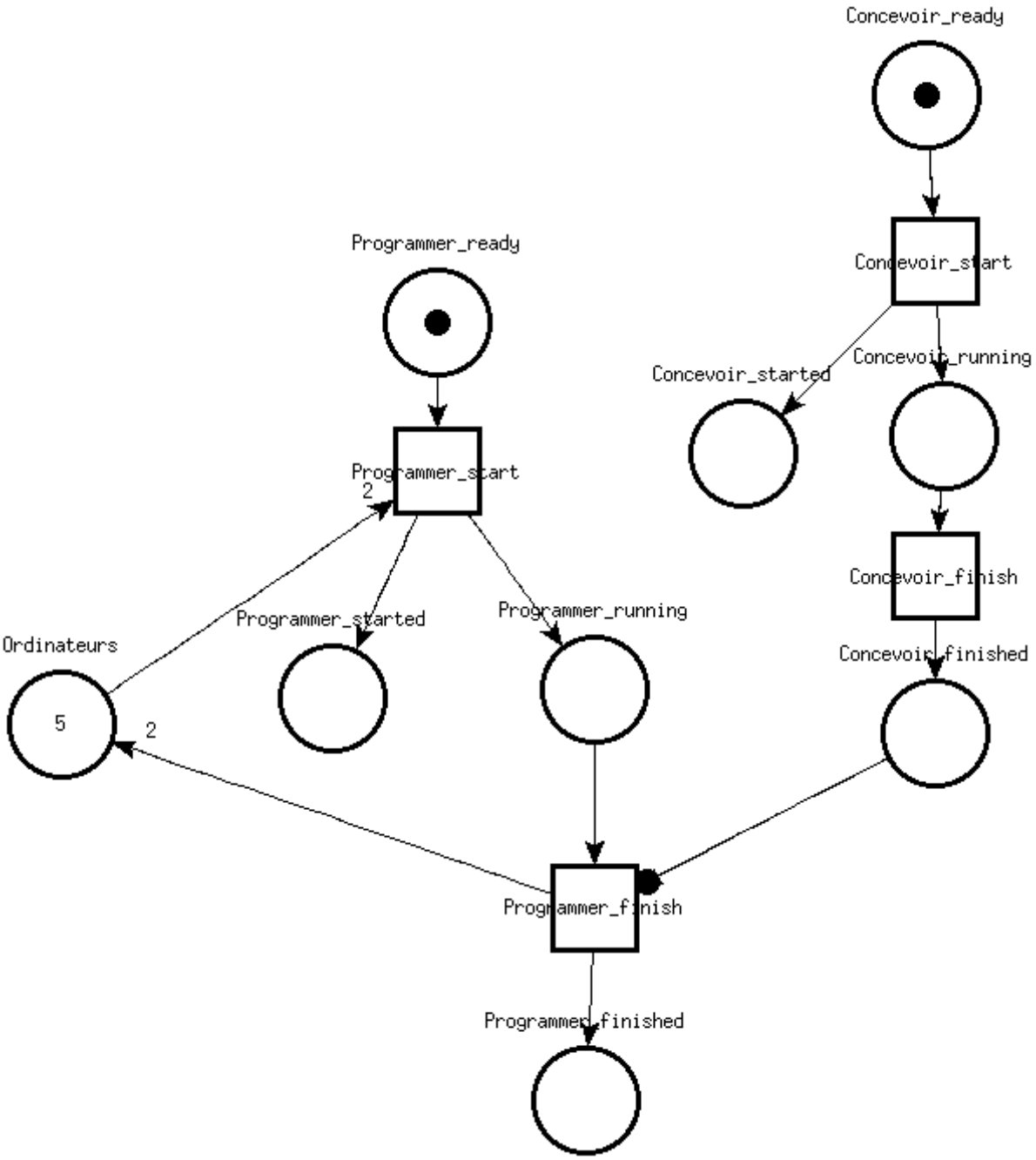

Figure 6: Capture d'écran de Tina affichant le fichier NET résultat

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.

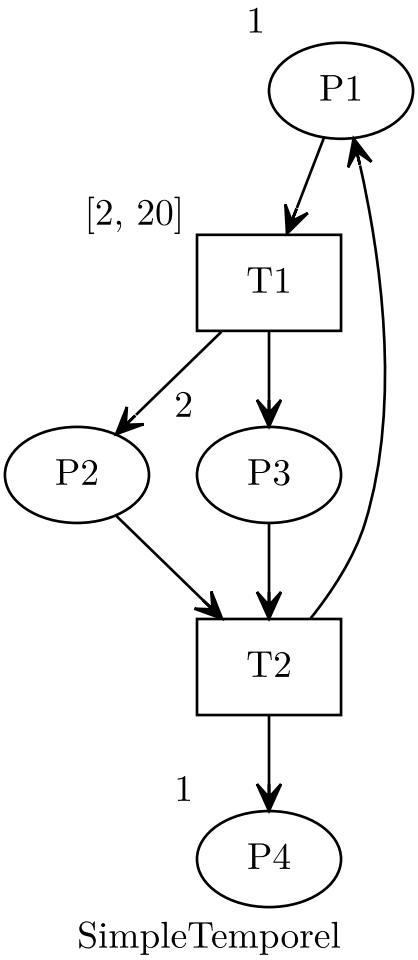

Figure 7: Exemple résultant de la transformation en DOT

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;
```

Listing 1: Exemple de fichier conforme à la grammaire

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

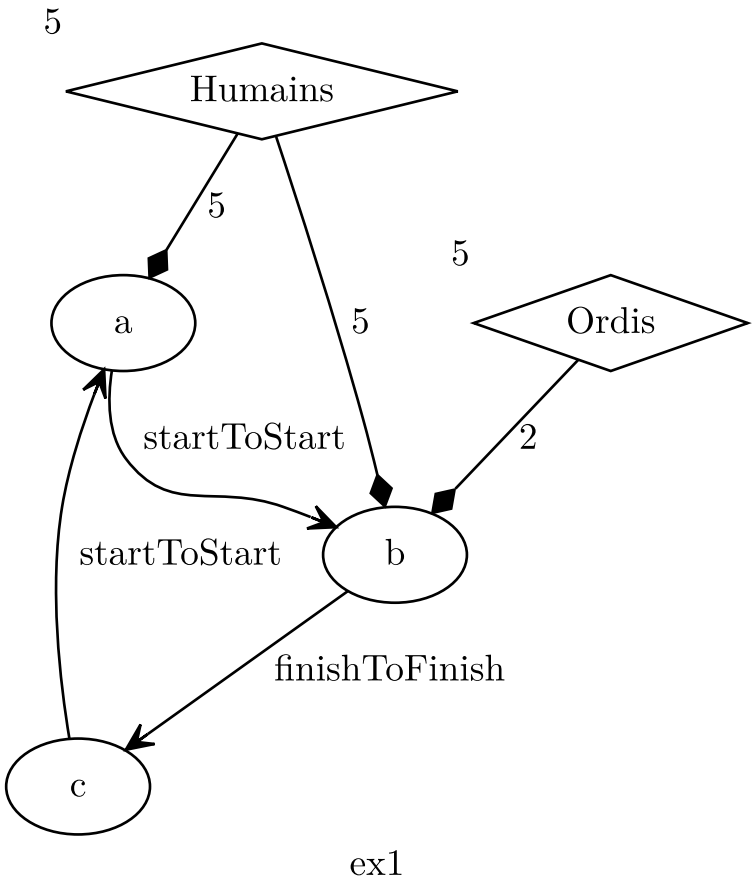

Figure 8: Résultat de `SimplePDL3` → `SimplePDL` → `DOT`

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

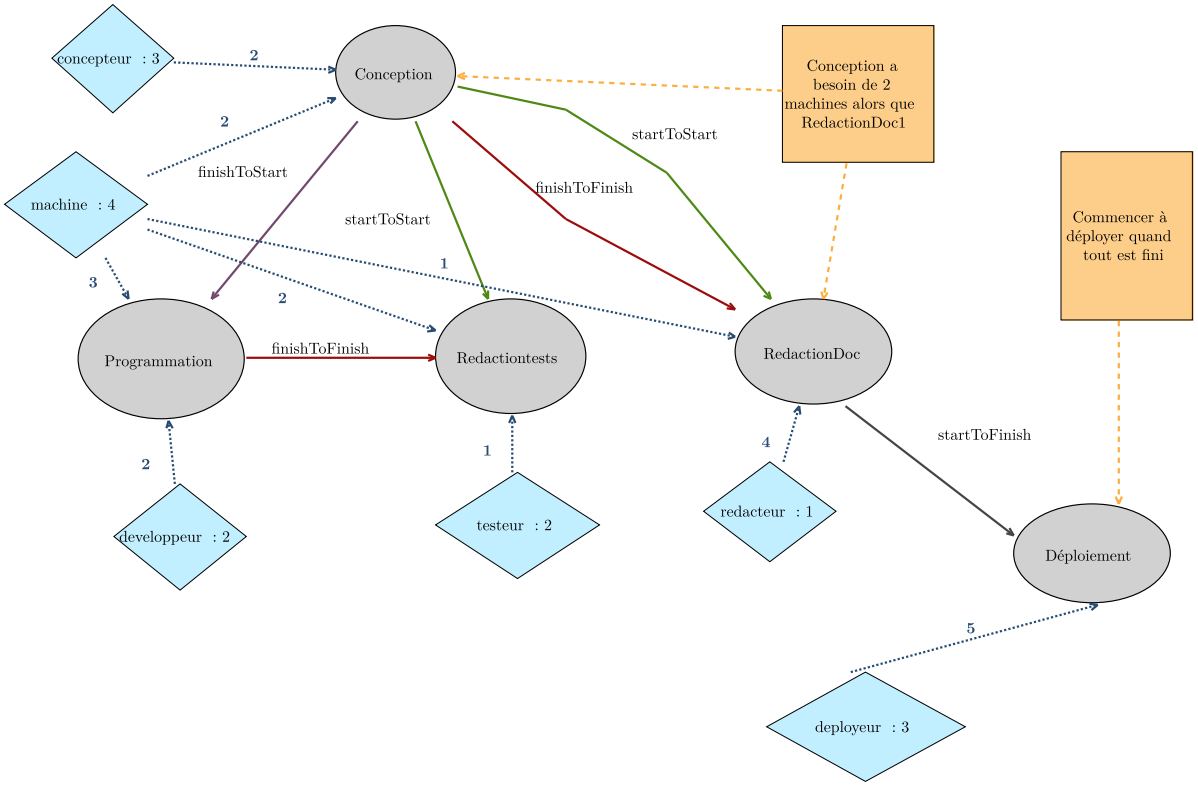

Figure 9: Edition de `pdl-sujet-ressources.xmi` avec Sirius

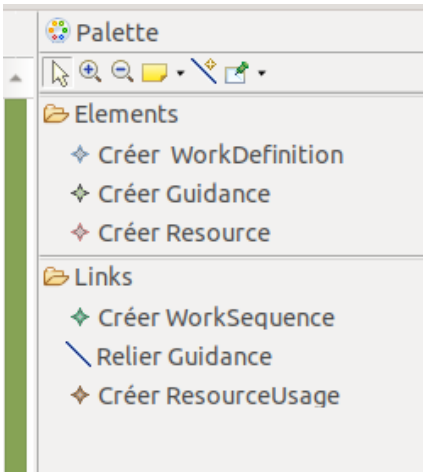

Figure 10: Palette de création dans Sirius

6. Vérification de terminaison et invariants

Pour vérifier si un processus se termine bien et qu'il respecte certaines propriétés, il faut les exprimer en LTL. Ces propriétés seront vérifiées sur le réseau de Péttri associé au processus.

- 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;
```

- Les invariants de processus sont les mêmes que ceux de réseaux de Pétri. Un processus ne peut être en cours et en même temps avoir fini, ses états sont donc exclusifs.

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

- Il aurait été intéréssant de vérifier que dès qu'une place `started` possède un jeton, celui y reste pour toujours. Ou qu'une place modélisant une `Resource` récupérera forcémment les jetons qu'elle donne un jour.

7. Conclusion

Nous avons vu différentes méthodes de transformation : M2M, T2M, M2T et graphique. La combinaison des méthodes nous a donc permis de partir d'une certaine représentation de processus puis de le transformer succintement pour vérifier les propriétés et s'assurer de la terminaison et de la consitance de la représentation.

Pour ce faire nous avons testé l'exemple vu en TD :

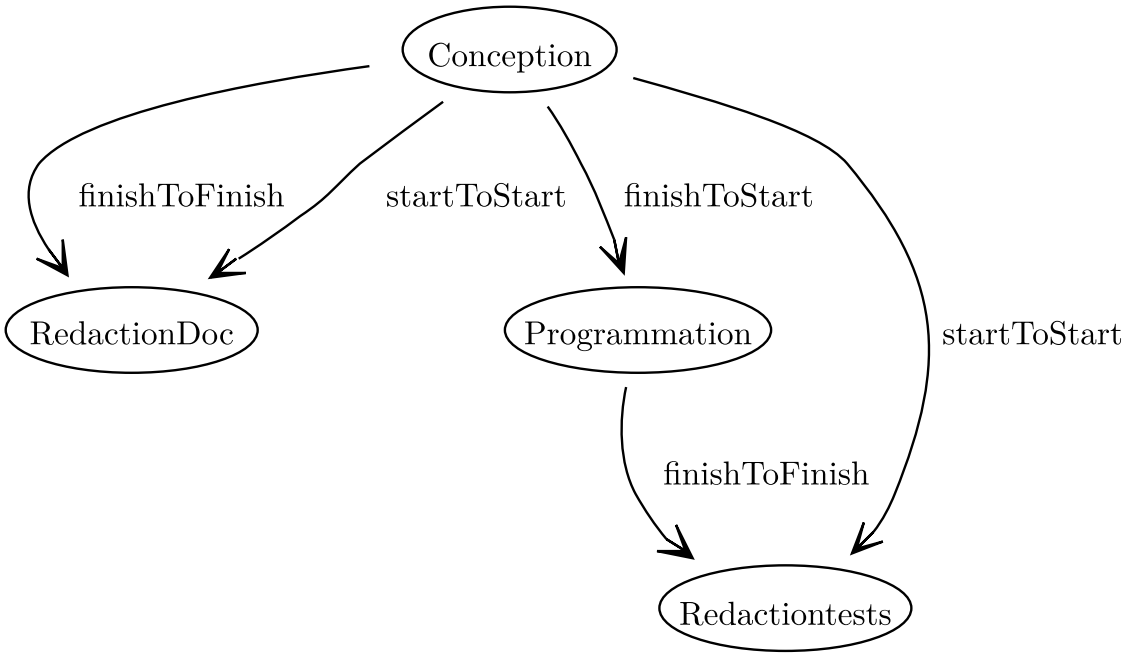

Figure 11: Résultat de `SimplePDL3` → `SimplePDL` → `DOT`

Nous avons commencé par la syntaxe textuelle pour la transformation T2M à l'aide de `Xtext` en SimplePDL (vu sous forme de dot Figure 11)

Ensuite avec une transformation M2M avec `Java` ou `ATL` nous avons pu transformer ce `Process` en un `PetriNet`.

C'est enfin que la transformation M2T nous a permi d'engendrer le réseau de Pétri sous syntaxe `Tina` pour pouvoir obtenir le `PetriNet` correspondant.

Ensuite, nous avons créé nos propriétés LTL à partir de nos deux templates :

```
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);
```

Une fois lancé avec la commande `selt -p Sujet_TD.ktz -prelude Sujet_TD_termine.ltl` nous obtenons le résultat souhaité : à savoir que le réseau de Pétri finira toujours.

Cependant pour les invariants, la sortie nous montre qu'il est possible d'être dans un état dans lequel les invariants ne sont pas respectés. Exemple pour la seconde propriété :

```
FALSE
state 0: Conception_ready Programmation_ready RedactionDoc_ready
Redactiontests_ready
-Conception_start->
state 1: Conception_running Conception_started Programmation_ready
RedactionDoc_ready Redactiontests_ready
-Conception_finish->
state 2: Conception_finished Conception_started Programmation_ready
RedactionDoc_ready Redactiontests_ready
-Programmation_start->
state 3: Conception_started Programmation_running Programmation_started
RedactionDoc_ready Redactiontests_ready
-Programmation_finish->
state 4: Conception_started Programmation_finished Programmation_started
RedactionDoc_ready Redactiontests_ready
-RedactionDoc_start->
state 5: L.dead Programmation_finished Programmation_started RedactionDoc_running
RedactionDoc_started Redactiontests_ready
-L.deadlock->
state 6: L.dead Programmation_finished Programmation_started RedactionDoc_running
RedactionDoc_started Redactiontests_ready
[accepting all]
```

Lors de ce mini-projet nous avons donc réussi à transformer des modèles et vérifier leur terminaison et consistance (ou pas). Nous avons compris qu'il pouvait être long et de se rendre compte de la véracité d'un modèle sans le transformer dans un autre aux propriétés plus simples. De plus, même des exemples simples peuvent ne pas obtenir un résultat vérifiant des propriétés 'simple'.