Rapport B.E.

Ingénierie Dirigée par les Modèles

2A SN App 2020

THIERRY Loïc & TIXIER Romain

INP TOULOUSE ENSEEIHT

Sommaire :

I) Définition des métamodèles :

1) SimplePDL

> Fichier disponible sous *fr.n7.simplepdl/models/simplepdl.ecore*.

Nous avons ici fait la modification du métamodèle SimplePDL pour prendre la compte la gestion de ressources nécessaires pour effectuer des activités. Pour cela, nous avons rajouté 2 classes nommées *Ressource* et *RessourceNeeded*. La classe *Ressource* définit les différentes ressources disponibles pour l'ensemble du processus, tandis que la classe *RessourceNeeded* représente la quantité de ressource nécessaire pour réaliser une action.

Voici le diagramme UML correspondant au nouveau méta-modèle de SimplePDL :

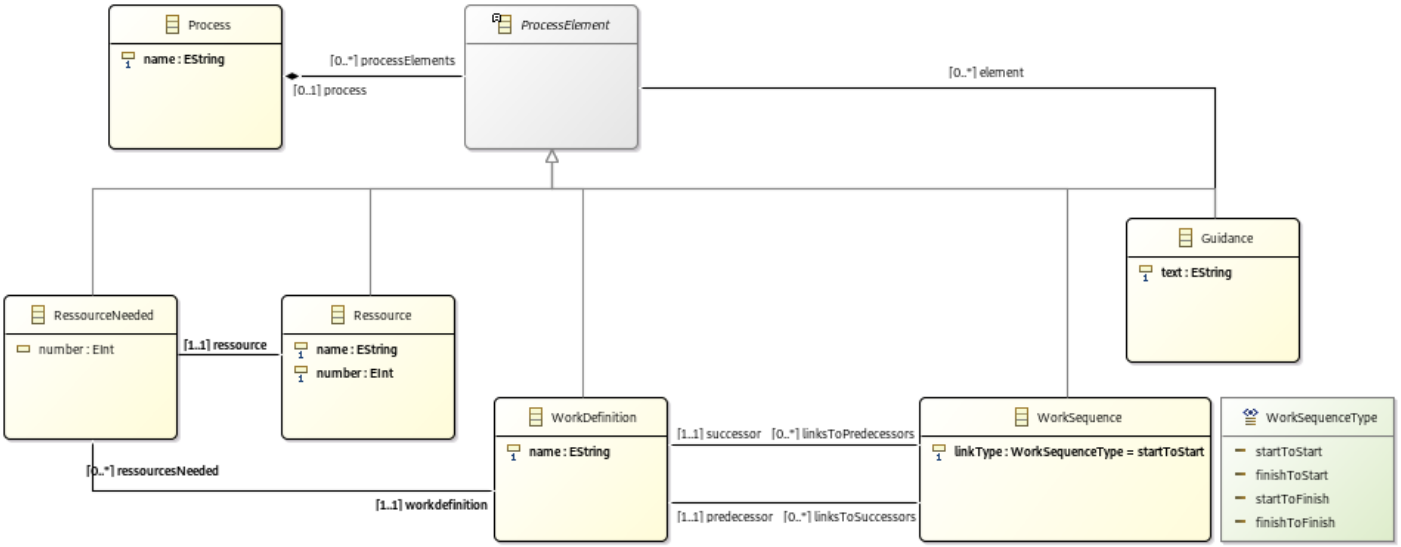

Figure 1 : Méta-modèle de SimplePDL

2) PetriNet

> Fichier disponible sous *fr.n7.petrinet/models/petri.ecore*.

Pour la réalisation du méta-modèle du réseau de Petri, nous sommes sommes basé sur ce que l'on avait déjà réalisé durant les séances de TP. L'architecture se présente sous la forme suivante : nous avons une classe *Net* qui va symboliser notre réseau de Petri, et qui est donc composé de plusieurs éléments regroupés dans une classe *PetriElement*. Un *PetriElement* peut soit être un *Arc* (normal ou read-arc), soit un *Node* (des Places ou des Transitions).

- Un *Arc* va posséder un poid, déterminant le nombre de jetons qui seront déplacés d'un *Node* vers un autre (d'une *Place* vers une *Transition*, ou inversement d'une *Transition* vers une *Place*).
- Un *Node* va potentiellement avoir un ou plusieurs *Arc* en entrée ainsi qu'en sortie. De plus, il va posséder un attribut permettant d'indiquer son marquage initial.

Voici le diagramme UML correspondant au métamodèle de notre PetriNet :

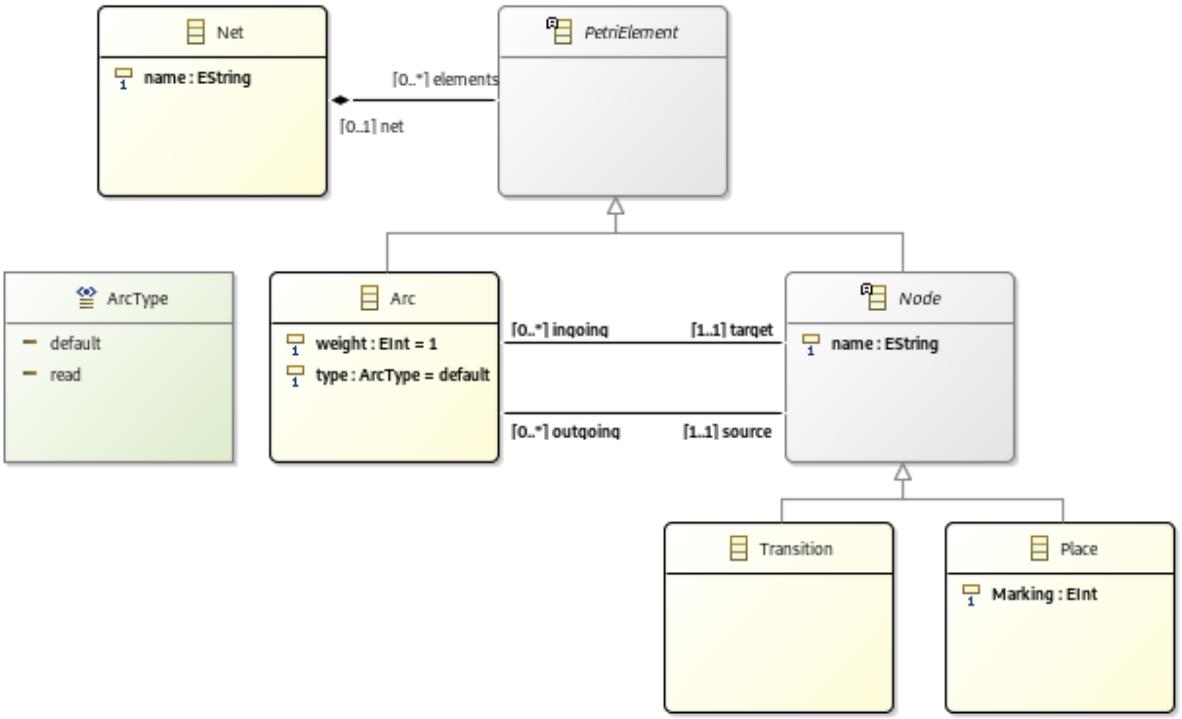

Figure 2 : Méta-modèle du Réseau de Petri

II) Définition des contraintes OCL :

1) SimplePDL

> Fichier disponible sous *fr.n7.simplepdl/models/simplepdl.ocl*.

Pour tester les modèles *simplePDL* avec OCL nous avons fixé ces différentes règles :

- Chaque activité (*WorkDefinition*) et chaque ressource doivent posséder un nom unique.
- Le processus, les activités et les ressources doivent posséder un nom valide. Pour nous, un nom valide ne doit être composé que de lettres, chiffres ou d'underscore, sachant qu'un nom ne peut pas commencer par un chiffre.
- Chaque ressource utilisée pour une activité doit être unique. On ne peut effectivement pas utiliser deux fois la même ressources en même temps pour la même activité.
- Les relations (*WorkSequence*) et les deux activités qu'elles relient doivent se situer dans le même processus.
- Les relations ne doivent pas être réflectives, c'est à dire que les activités qu'elles relient ne doivent pas être les mêmes.
- La quantité de ressources disponibles pour une ressource doit être supérieure à 1.
- Pour chaque ressource nécessaire à une activité (*RessourceNeeded*), il faut que la quantité nécessaire de cette ressource soit supérieure à 1 tout en étant inférieur à sa quantité disponible.
- Enfin, pour chaque *RessourceNeeded*, la ressource demandée et l'activité correspondante doivent se trouver dans le même processus que celle-ci.

Pour vérifier que nos contraintes OCL soient bien toutes prises en comptes, nous avons généré un contre-exemple de modèle qui est volontairement incorrect. Voici son arborescence :

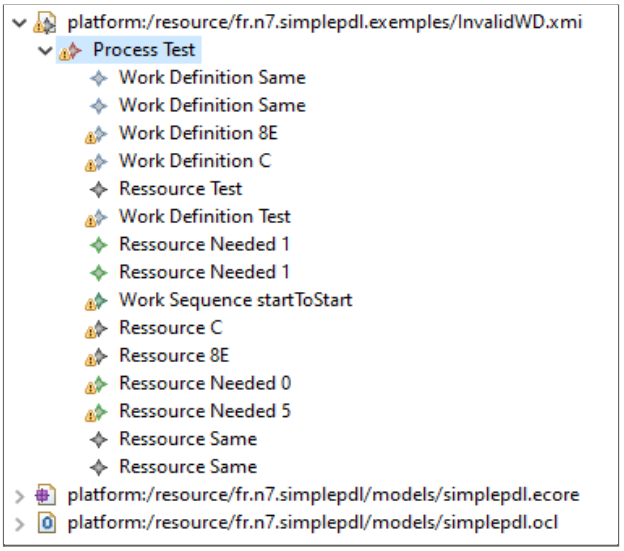

Figure 3 : Exemple d'un modèle SimplePDL incorrect.

Dans ce contre-exemple, on peut facilement voir que nous avons ici des activités possédants des noms identiques et d'autres qui ont des noms que nous considérons non valides (commençant par des chiffres ou bien n'ayant qu'une lettre). Nous avons également fait de même avec quelques ressources, en ajoutant en plus le fait que leur quantité soit nulle. Enfin, nous avons ajouté des WorkSequences ayant comme activité cible la même activité que celle d'où elle vient, et ajouté des ressources nécessaires pour une activité qui dépassent la quantité disponible de cette ressource.

En chargeant le fichier OCL sur ce modèle invalide, on pourra bel et bien observer que celui-ci viole nos contraintes :

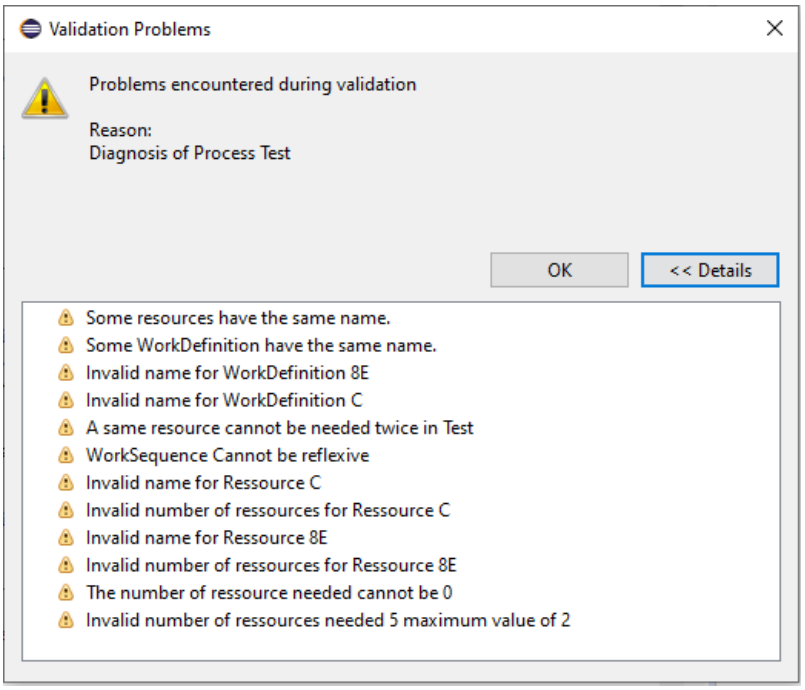

Figure 4 : Erreurs de validation du modèle SimplePDL incorrect.

2) PetriNet

> Fichier disponible sous *fr.n7.petrinet/models/petri.ocl*.

Pour tester les modèles *petri* avec OCL nous avons cette fois-ci défini les règles suivantes :

- La racine *Net* du réseau de Petri, ainsi que les différents noeuds (*Place* et *Transitions)*, doivent posséder des noms valides.
- Chacun des éléments du réseau de Petri doivent de plus posséder un nom unique.
- Pour chaque noeud, tous les arcs entrants et sortants de celui-ci doivent être unique.
- Pour chaque *Arc*, si la source de celui-ci est un noeud de type *Place*, alors sa destination doit être une *Transition*. Inversement, si le noeud source est une *Transition*, sa destination doit être une *Place*.
- Enfin, les noeuds source et de destination d'un *Arc* doivent se situer dans le même réseau de Petri que ce dernier.

Là aussi, pour valider nos contraintes OCL, nous avons créé un exemple assez simple d'un réseau de Petri non-conforme. Nous n'avons pas forcément trouvé très pertinent de retester des erreurs assez triviale comme la vérification des noms par exemple que nous avons déjà montré sur SimplePDL, c'est pourquoi nous nous ici plus penchés sur la validation des nouvelles contraintes.

Par exemple dans le modèle suivant, nous avons deux places Source et Target, avec une transition Middle entre les deux. Nous avons ici deux arcs invalides qui vont tous deux partir de la transition Middle pour aller dans la même place Target, et un autre arc invalide qui va directement de la place Source à la place Target :

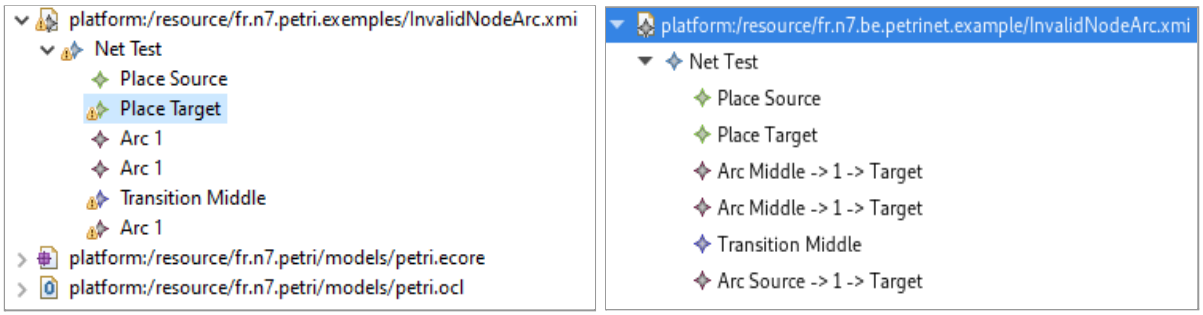

Figure 5 : Exemple d'un modèle PetriNet incorrect.

<u>Note</u> : À droite, nous avons présenté le modèle en question en ayant amélioré l'affichage via le package.edit, ce qui permet de mieux visualiser les sources et destinations des arcs. Sur la capture de gauche cela n'est pas présent car nous n'avons pas généré le code Java de PetriNet sur une de nos deux machines. En effet, à partir du moment où nous franchissions cette étape, il n'était plus possible de valider nos modèles avec OCL.

En tentant de valider le modèle avec nos contraintes OCL, on peut voir ici aussi que ce dernier viole les contraintes que nous avons visé :

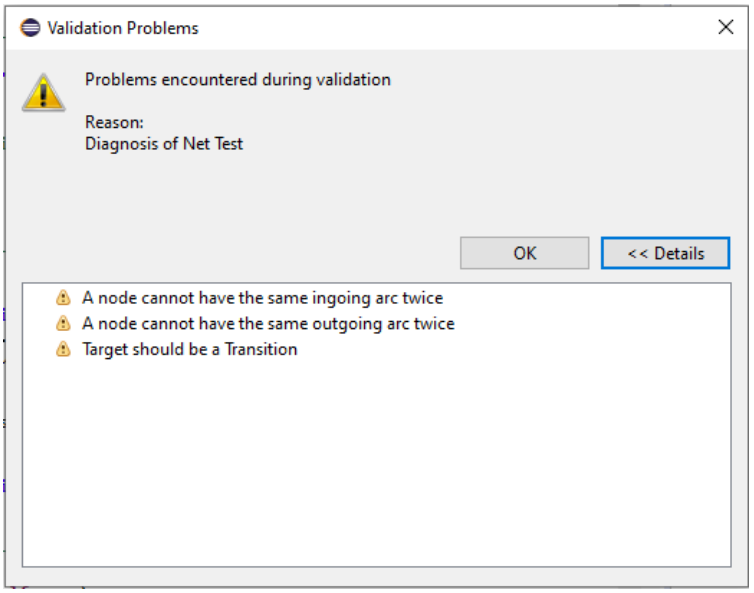

Figure 6 : Erreurs de validation du modèle PetriNet incorrect.

III) Transformation SimplePDL vers PetriNet avec ATL :

> Fichier ATL disponible sous *fr.n7.simplepdl2petri/simplepdl2petri.atl*.

Pour la transformation de SimplePDL vers PetriNet, nous avons réalisé la transformation des WorkDefinition en trois places pour définir les différents états d'une activité (*ready*, *running* et *finish*), ainsi qu'avec deux transitions (*start* et *end*) qui permettent de relier ces places.

À cela, nous avons également rajouté deux places complémentaires que nous avons nommées *hasStarted* et *hasFinished*, qui nous permettent ici de garder en "mémoire" l'information qu'une action est déjà passée. La place *hasFinished* n'est pas forcément nécessaire dans tous les cas mais nous avons préféré la préserver pour avoir une distinction entre les états de l'action et les états passés qui seront utilisés pour modéliser les arcs.

Pour modéliser les arcs nous avons ensuite lié les places (*hasStarted* ou *hasFinished*) aux transitions (*start* ou *finish*), ce qui permet *in fine* de modéliser tous les arcs possibles.

Enfin, les ressources seront définies par une place dédiée, dans laquelle nous allons placer autant de jeton qu'il y a de quantité disponible pour cette même ressource. Pour finir, on tirera des arcs partant de cette ressource vers les transitions *start* des activités qui en ont besoin, et une fois ces activités finies on va rendre la ressource en tirant des arcs des transitions *end* de ces dernières vers la place de la ressource.

Pour illustrer nos propos, voici un exemple de ce que nous donnerait notre code ATL pour une activité A2 qui a besoin de 2 machines pour s'exécuter, et où il est nécessaire qu'une autre activité A1 ait commencée pour qu'elle puisse se finir :

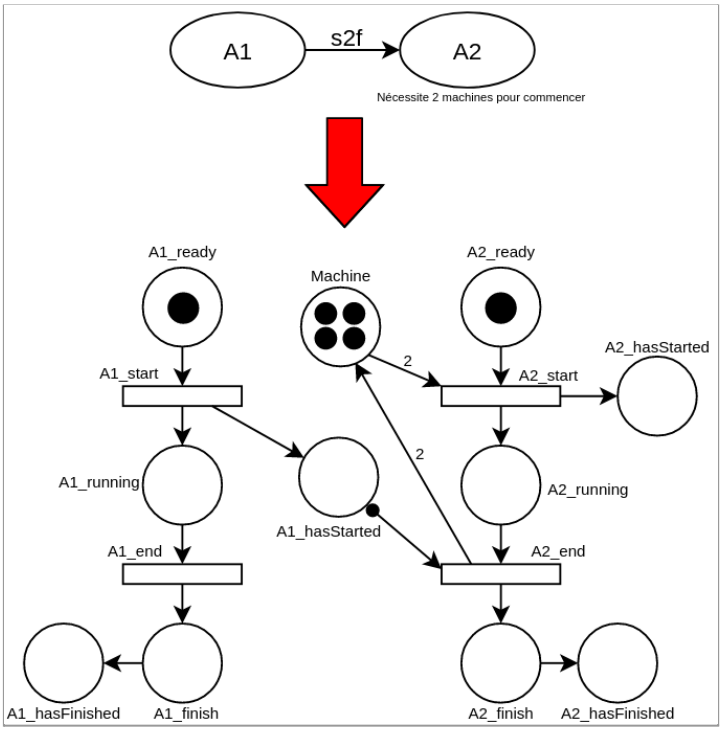

Figure 7 : Schéma d'un exemple de SimplePDL vers Petri

IV) Transformation PetriNet vers Tina avec Acceleo :

> Fichier disponible sous *fr.n7.petrinet.totina/src/fr/petrinet/totina/main/toTina.mtl*.

Code TINA généré avec Acceleo, dans un fichier '.net' :

```
net be
pl Conception_ready (1)
pl Developpement_ready (1)
pl RedactionTests_ready (1)
pl RedactionDoc_ready (1)
pl concepteur (3)
pl developpeur (2)
pl machine (4)
pl testeur (2)
pl redacteur (3)
pl Conception_running (0)
pl Conception_finish (0)
pl Conception_hasStarted (0)
pl Conception_hasFinish (0)
pl Developpement_running (0)
pl Developpement_finish (0)
pl Developpement_hasStarted (0)
pl Developpement_hasFinish (0)
pl RedactionTests_running (0)
pl RedactionTests_finish (0)
pl RedactionTests_hasStarted (0)
pl RedactionTests_hasFinish (0)
pl RedactionDoc_running (0)
pl RedactionDoc_finish (0)
pl RedactionDoc_hasStarted (0)
pl RedactionDoc_hasFinish (0)

tr Conception_start Conception_ready concepteur*2 machine*2
    -> Conception_hasStarted Conception_running

tr Conception_end Conception_running
    -> Conception_finish Conception_hasFinish concepteur*2 machine*2

tr Developpement_start Developpement_ready Conception_hasFinish?1 developpeur*2 machine*3
    -> Developpement_hasStarted Developpement_running

tr Developpement_end Developpement_running
    -> Developpement_finish Developpement_hasFinish developpeur*2 machine*3

tr RedactionTests_start RedactionTests_ready Conception_hasStarted?1 machine*2 testeur
    -> RedactionTests_hasStarted RedactionTests_running

tr RedactionTests_end RedactionTests_running Developpement_hasFinish?1
    -> RedactionTests_finish RedactionTests_hasFinish machine*2 testeur

tr RedactionDoc_start RedactionDoc_ready Conception_hasStarted?1 redacteur machine
    -> RedactionDoc_hasStarted RedactionDoc_running

tr RedactionDoc_end RedactionDoc_running Conception_hasFinish?1
    -> RedactionDoc_finish RedactionDoc_hasFinish redacteur machine
```

Une fois ce code généré, voici le graph que nous pouvons obtenir dans l'outil Tina :

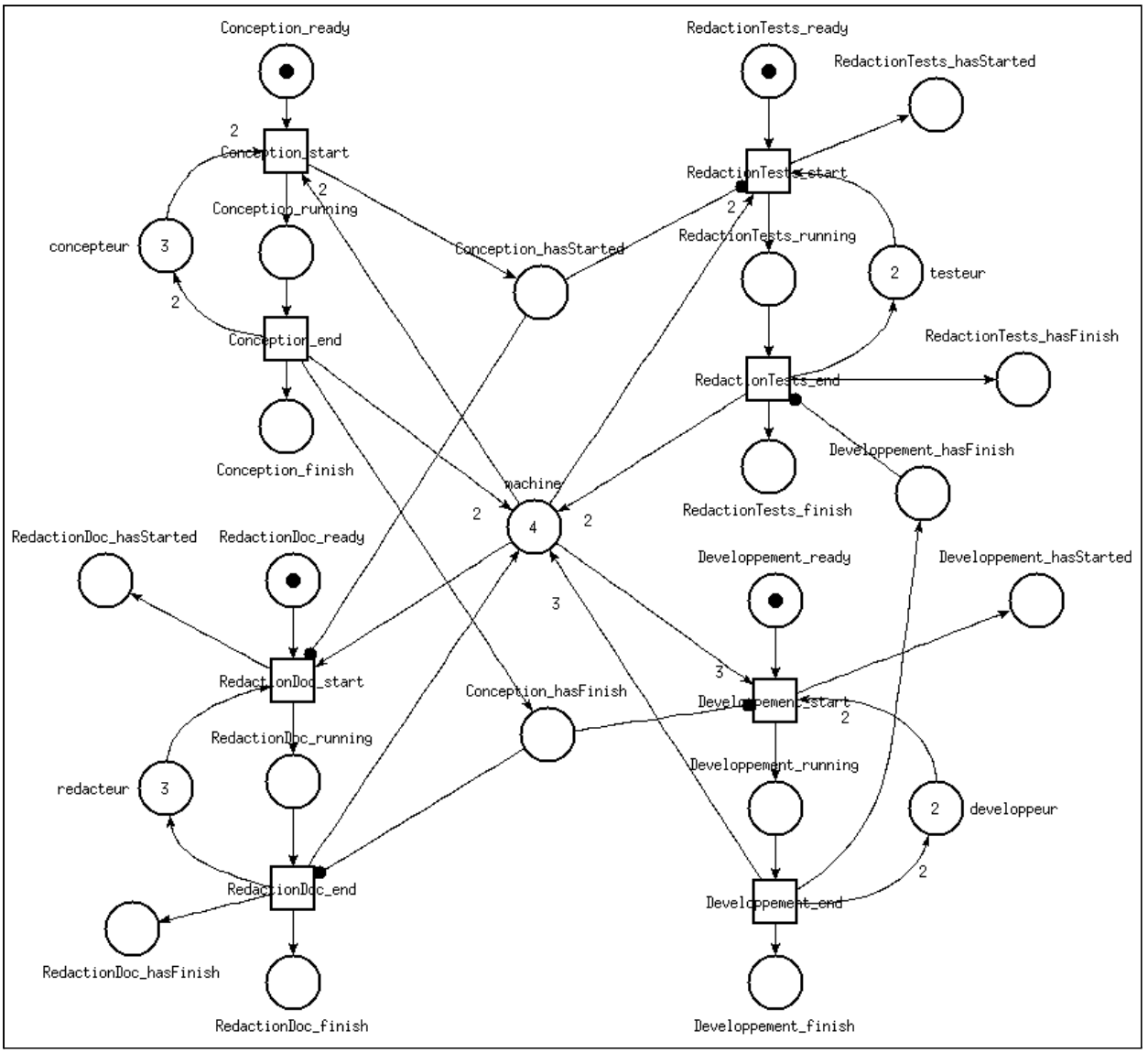

Figure 3 : Graph généré sur Tina