Chaîne de vérification de modèles de processus

Hamid Oukhnini
Gabriel Gervais

23 octobre 2021

Sommaire:

1 Introduction

L'ingénierie dirigée par les modèles (IDM) a permis plusieurs améliorations significatives dans le développement de systèmes complexes en fournissant des outils d'exécution, de validation et de vérification (statique et dynamique). Ainsi, il est toujours important de détecter les erreurs éventuelles dans un projet le plus tôt possible et avant d'entamer ses phases les plus profondes, c'est pour cela qu'on peut considérer la méta modélisation et sa simulation comme étape importante pour la validation des modèles décrivant le projet.

Parmis ces modèles, le modèle de procédé utilisé principalement dans notre mini-projet sera le suivant:

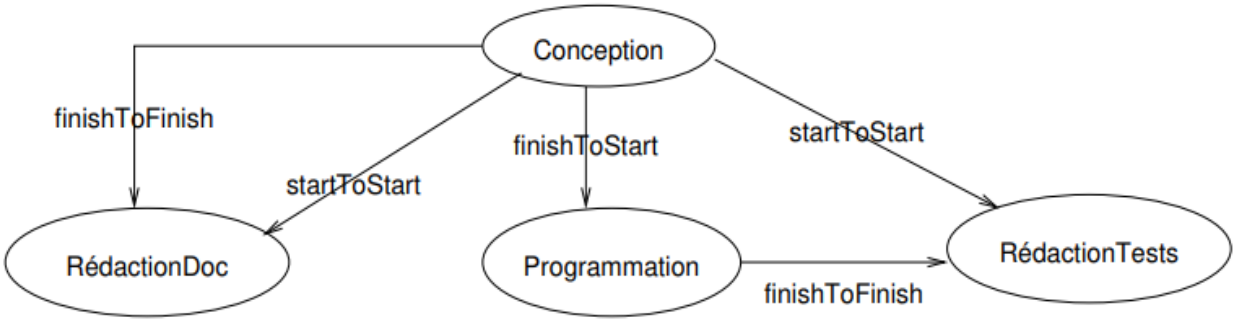
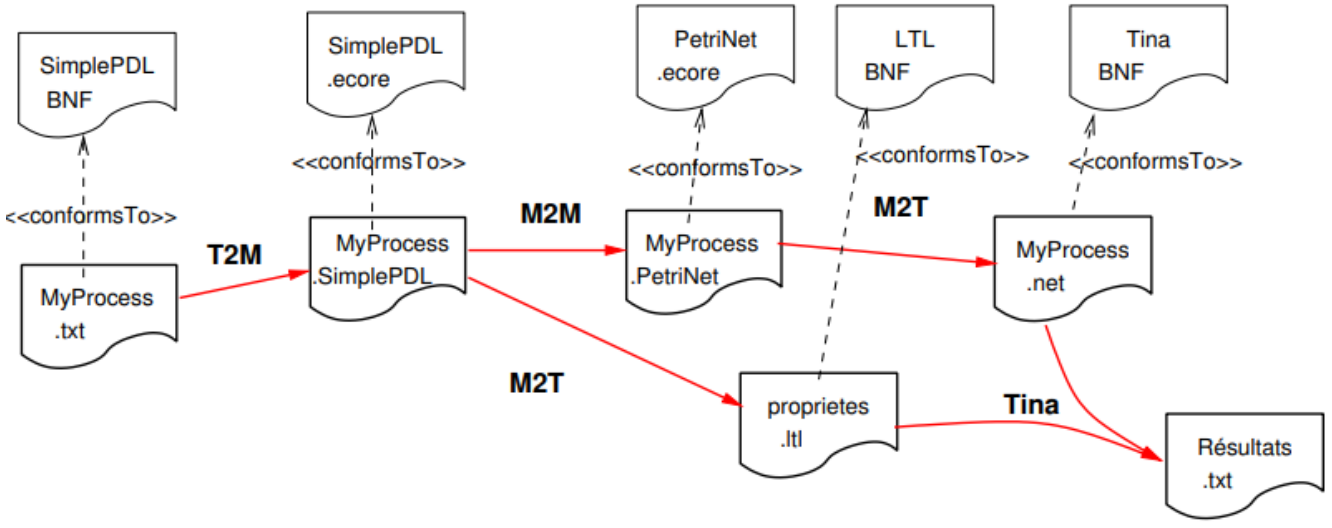


FIGURE 1 – Exemple de modèle de procédé

Nous allons donc essayer de produire une chaîne de vérification de modèles de processus SimplePDL en général pour vérifier si un processus peut se terminer ou pas. Nous passerons par plusieurs étapes afin de réaliser cette vérification:

- La construction des métamodèles avec Ecore.
- La définition des contraintes que ces modèles doivent respecter avec OCL.
- La transformation du modèle de processus en réseau de Petri avec java/ATL.
- Le **modele-checking** en utilisant la boîte à outils Tina.



2 Les métamodèles SimplePDL et PetriNet

2.1 SimplePDL

Le SimplePDL est un langage de métamodélisation qui sert pour décrire des modèles de processus. Il existe deux formes de langages pour le SimplePDL : un langage simplifié de description des procédés de développement et un langage plus avancé résultant de l'ajout de la notion de ProcessElement comme généralisation de WorkDefinition (activité) et WorkSequence (dépendance).

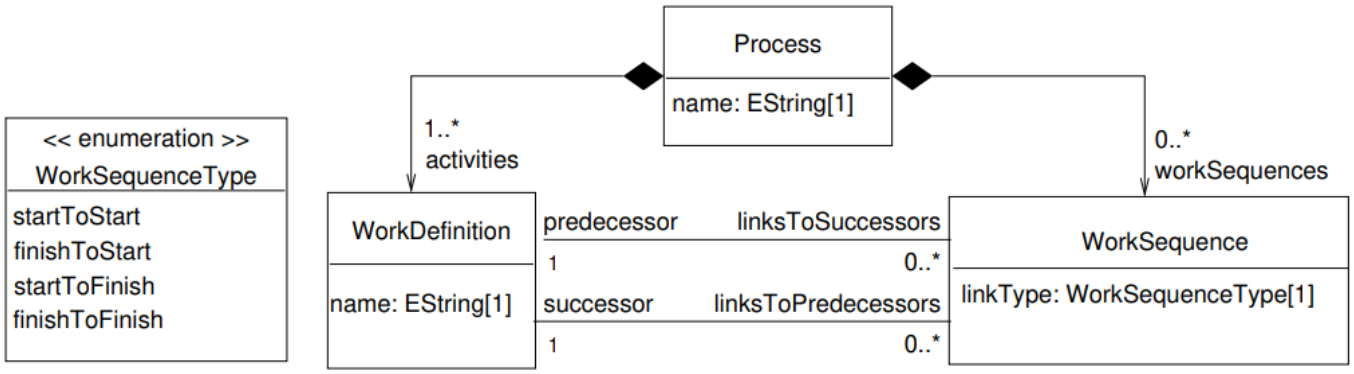


Figure 3 – Métamodèle simplifié de SimplePDL

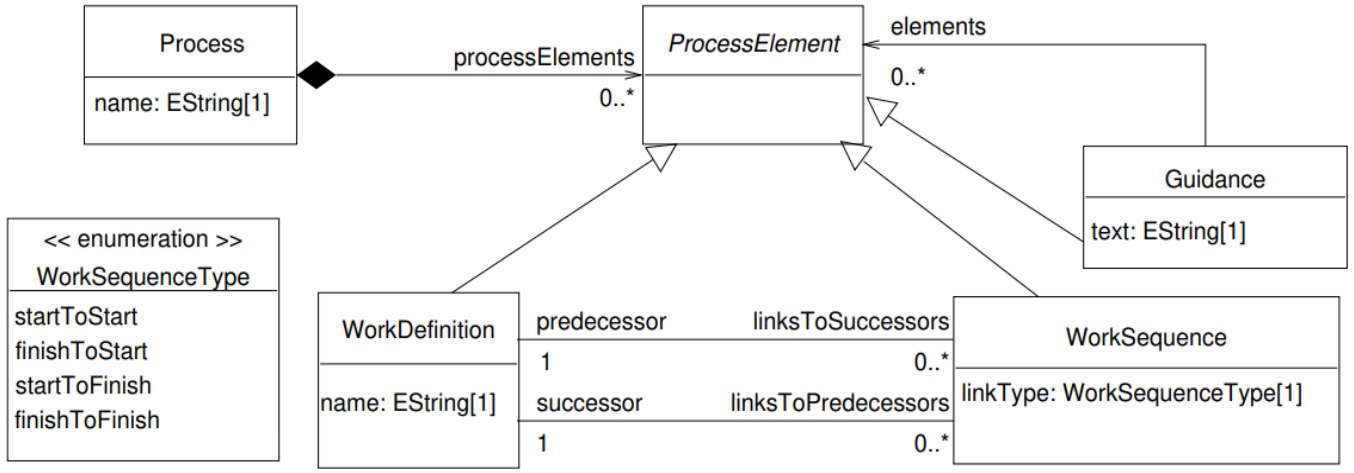


Figure 4 – Métamodèle avancé de SimplePDL

On retrouve donc le concept de processus (**Process**) composé d'un ensemble d'activités (**WorkDefinition**) représentant les différentes tâches à réaliser durant le développement. Une activité peut dépendre d'une autre (**WorkSequence**). Une contrainte d'ordonnancement sur le démarrage ou la fin de la seconde activité est précisée (**attribut linkType**) grâce à l'énumération WorkSequenceType. Par exemple, deux activités A1 et A2 reliées par une relation de précédence de type finishToStart signifie que A2 ne pourra commencer que quand A1 sera terminée.

Enfin, des annotations textuelles (Guidance) peuvent être associées aux activités pour donner plus de détails sur leurs réalisations.

2.3 Exemple de modèle simplePDL

La figure 5 donne un exemple de modèle de processus composé de quatre activités. Le développement ne peut commencer que quand la conception est terminée. La rédaction de la documentation ou des tests peut commencer dès que la conception est commencée (startToStart) mais la documentation ne peut être terminée que si la conception est terminée (finishToFinish) et les tests si le développement est terminé.

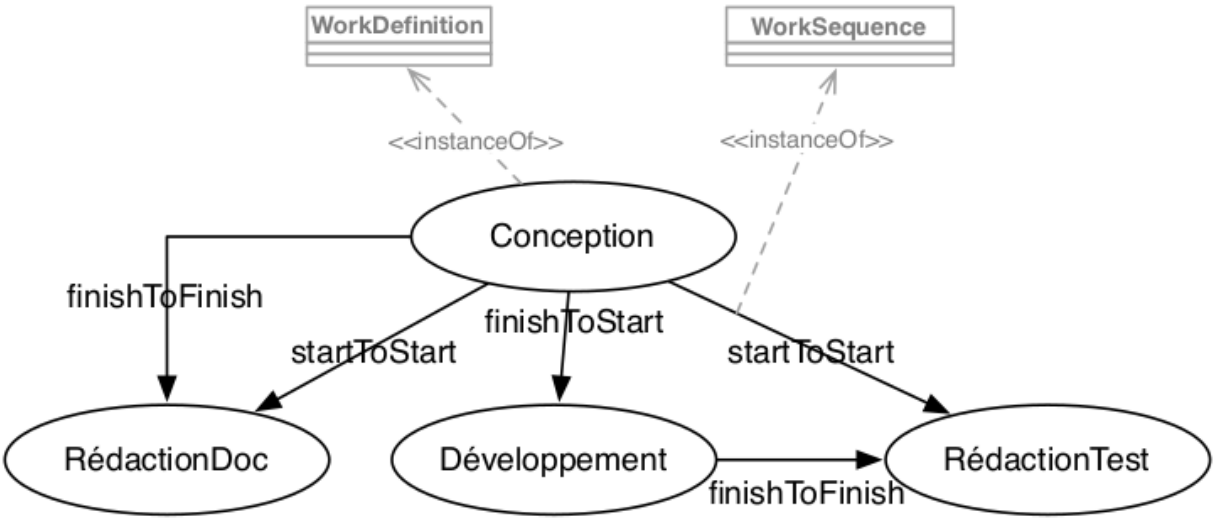


Figure 5: Exemple de modèle simplePDL

Afin qu'une activité soit réalisée, elle aura probablement disposer d'une ou plusieurs ressources pour assurer sa finalité. Une ressource contient des occurrences, une activité loue des occurences d'une ressource au début de son exécution, ces occurences sont alors utilisées exclusivement par cette activité jusqu'à la fin de sa réalisation.

Afin de réaliser cette tâche, on aura besoin d'ajouter deux classes au métamodèle de SimplePDL :

— Ressource : une ressource est définit par son nom (EString) qui décrit son type et ses occurences (Eint).

— Allocate : cette classe représente le nombre d'occurrences prises par une activité parmi les occurences d'une ressource pour effectuer sa réalisation.

A noter qu'une activité (WorkDefinition) aura besoin éventuellement de plusieurs ressources, ainsi elle sera composée de plusieurs classes Allocate

La figure 5 ci dessous donne le métamodèle SimplePDL complet avec l'ajout des ressources

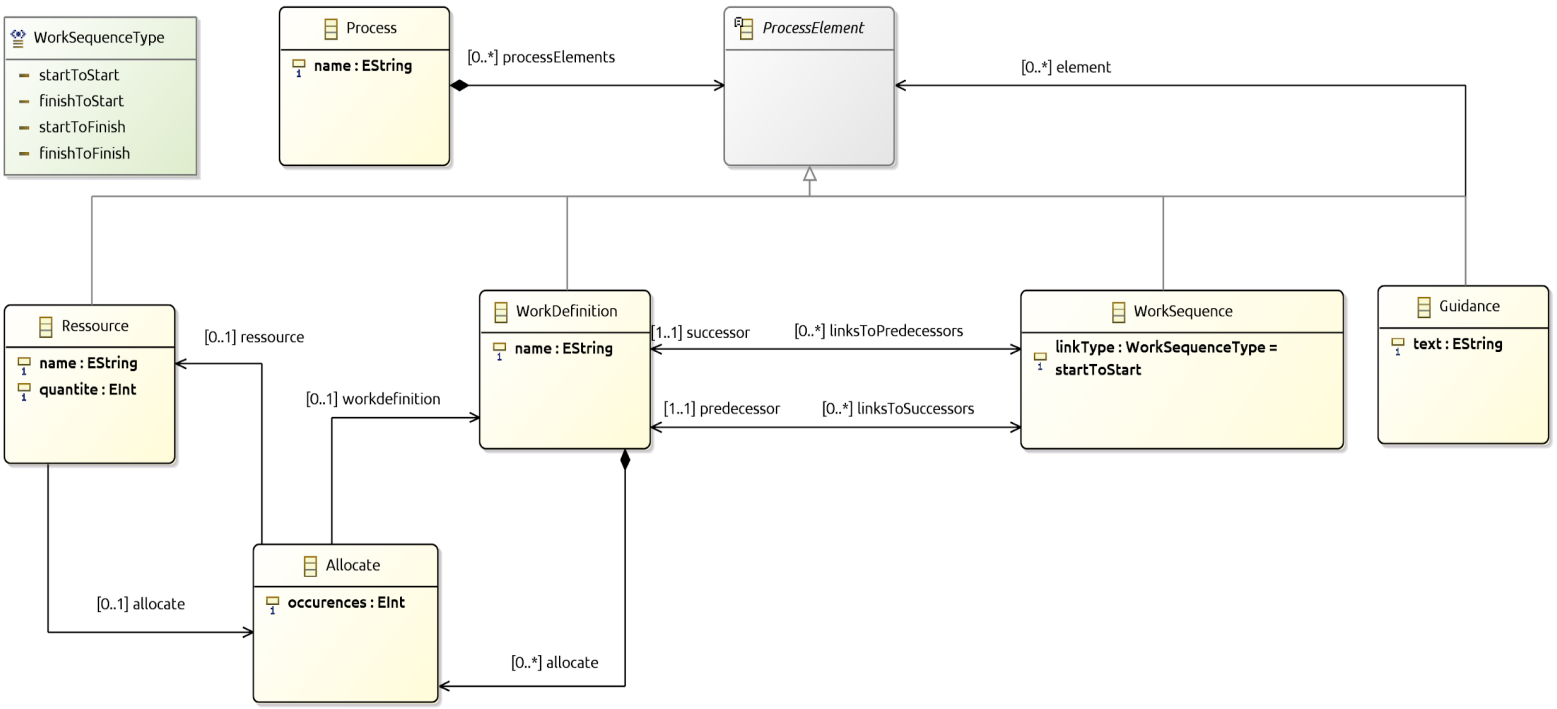


Figure 6 – Le métamodèle SimplePDL avec ressources ajoutées

2.2 PetriNet

Un réseau de Petri (**PetriNet**) est composé de nœuds (**Node**) pouvant être des places (**Place**) ou des transitions (**Transition**). Les nœuds sont reliés par des arcs (**Arc**) pouvant être des arcs normaux ou des readArcs (**ArcKind**). Le poids d'un arc (**weight**) indique le nombre de jetons consommés dans la place source ou ajoutés à la place destination (sauf dans le cas d'un **readArc** où il s'agit simplement de tester la présence dans la place source du nombre de jetons correspondant au poids). Le marquage du réseau de Petri est capturé par l'attribut **tokens** d'une place.

La figure ci dessous représente notre métamodèle **PetriNet:**

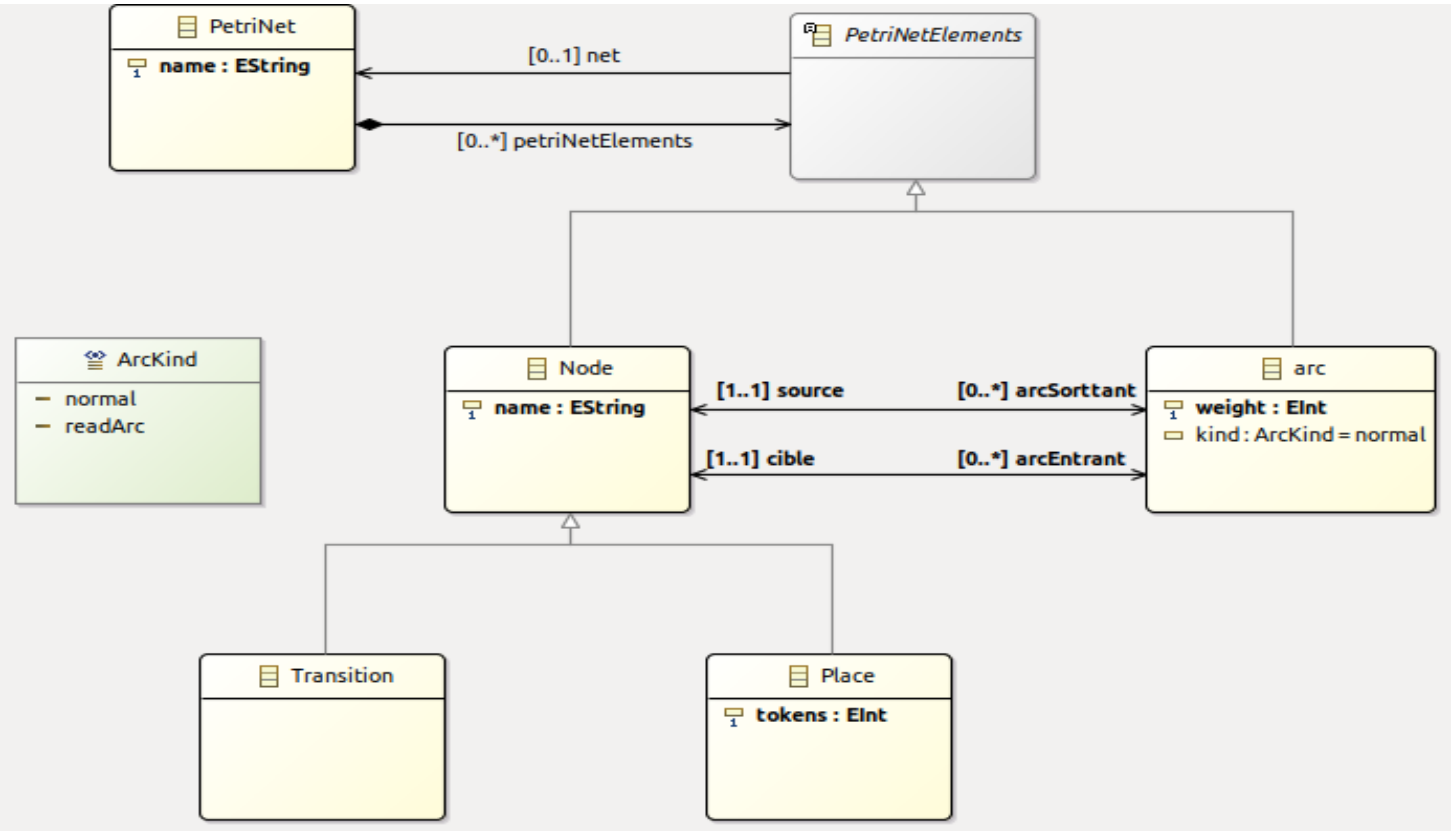


Figure 6 – Le métamodèle PetriNet

3.3 Exemple de réseau de Petri :

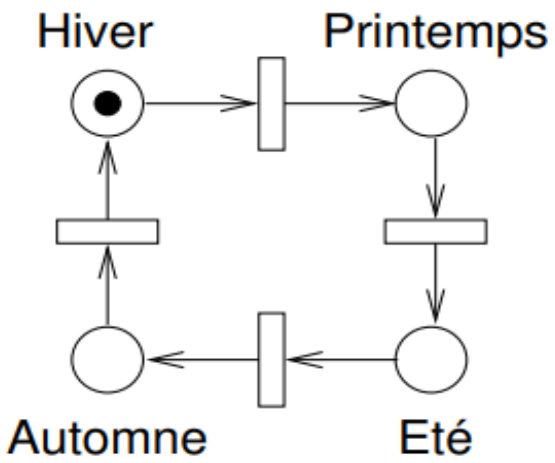


Figure 7 – Exemple de réseau de Petri : Evolution des saisons

3 Les contraintes OCL associés à ces métamodèles

Nous avons utilisé Ecore pour définir un méta-modèle pour les processus.Cependant, le langage de méta-modélisation Ecore ne permet pas d'exprimer toutes les contraintes que doivent respecter les modèles de processus. Aussi, on complète la description structurelle, sémantique statique du méta-modèle réalisée en Ecore par des contraintes exprimées en OCL. Le méta-modèle Ecore et les contraintes OCL définissent la syntaxe abstraite du langage de modélisation considéré.

3.1 Les contraintes ajoutées au SimplePDL :

— contrainte 1:

```
context WorkDefinition
   inv uniqNames: self.Process.processElements
    ->select(pe | pe.oclIsKindOf(WorkDefinition))
    ->collect(pe | pe.oclAsType(WorkDefinition))
    ->forAll(w1, w2 | w1 = w2 or w1.name <> w2.name)
```

Cette contrainte stipule que deux activités différentes d'un même processus ne peuvent pas avoir le même nom : Pour chaque activité WorkDefinition On fait le parcours de tous les ProcessElements, on sélectionne ceux du type WorkDefinition et on réalise le test manifestant que toute WorkDefinition autre que la présente possède un identifiant différent.

— contrainte 2:

```
context WorkSequence
      inv notReflexive: self.predecessor <> self.successor
```

Une dépendance ne peut pas être réflexive : Une WorkSequence ne peut pas lier deux activités identiques.

— contrainte 3:

```
context WorkDefinition
inv validName:
    self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
```

Le nom d'une activité ne doit être composé que de lettres, chiffres ou soulignés, un chiffre ne peut pas être première position.

— contrainte 4:

```
context Allocate
inv ressourceSuffisante:
    self.occurences <= self.ressource.quantite
```

Une activité ne peut pas demander un nombre d'occurrences d'une ressource supérieur aux occurrences que peut offrir cette ressource à la base.

3.2 Les contraintes principales ajoutées au réseau de Petri:

Comme pour le métamodèle SimplePDL, le métamodèle réseau de Petri créé ne représente que la syntaxe statique, on doit en ajouter alors des contraintes OCL, on définit ainsi la syntaxe abstraite du réseau de Petri.

La figure ci-dessous donne Les contraintes OCL du métamodèle PetriNet:

```
    -- Toute Place doit avoir un nombre entier naturel de Jetons,
context Place
 inv marquageInitial : self.tokens >= 0

    -- Un arc ne peut pas lier deux Places ou deux transitions
context arc
     inv arcValid : self.source.oclIsTypeOf(Place) <> self.cible.oclIsTypeOf(Place)
            and self.source.oclIsTypeOf(Transition) <> self.cible.oclIsTypeOf(Transition)

   -- Le nombre de Jetons que transporte un arc doit être un entier naturel.
context arc
   inv : self.weight >= 1

    -- Une place ne peut pas etre isolée
context Node
   inv : self.arcSorttant->size() >= 1 or self.arcEntrant->size() >= 1

    -- Deux places ne peuvent pas avoir le meme nom
context Place
   inv uniqNames: self.PetriNet.petriNetElements
    ->select(pe | pe.oclIsKindOf(Place))
    ->collect(pe | pe.oclAsType(Place))
    ->forAll(w1, w2 | w1 = w2 or w1.name <> w2.name)
```

Figure 8 – Les contraintes OCL du métamodèle PetriNet.

4 L'éditeur graphique Sirius

4.1 Introduction

À l'instar d'une syntaxe concrète textuelle, une syntaxe concrète graphique fournit un moyen de visualiser et/ou éditer plus agréablement et efficacement un modèle. Nous allons utiliser l'outil Sirius développé par les sociétés Obeo et Thales, et basé sur les technologies Eclipse Modeling comme EMF et GMF. Il permet de définir une syntaxe graphique pour un langage de modélisation décrit en Ecore et d'engendrer un éditeur graphique intégré à Eclipse.

4.2 Définition de la syntaxe graphique avec Sirius

notre point de départ était la syntaxe abstraite du DSML définie par le modèle Ecore déjà 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 Guidances, les WorkSequences ainsi que les Ressources Comme le montre la figure ci-dessous.

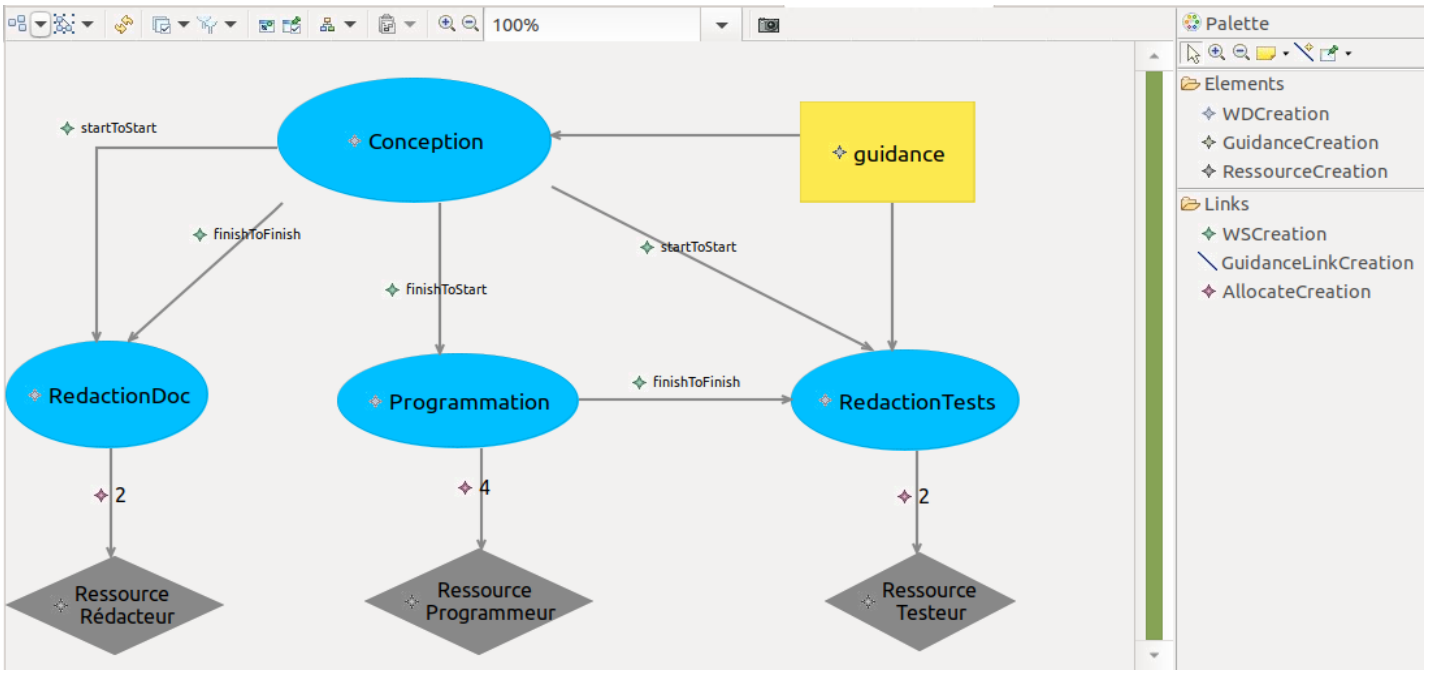


Figure 9 – Le modèle généré par l'éditeur graphique

5 Le modèle Xtext

La syntaxe abstraite d'un DSML (exprimée en Ecore ou un autre langage de métamodélisation) ne peut pas être manipulée directement. Le projet EMF d'Eclipse permet d'engendrer un éditeur arborescent et les classes Java pour manipuler un modèle conforme à un métamodèle. Cependant, ce ne sont pas des outils très pratiques lorsque l'on veut saisir un modèle. Aussi, il est souhaitable d'associer à une syntaxe abstraite une ou plusieurs syntaxes concrètes pour faciliter la construction et la modification des modèles. Ces syntaxes concrètes peuvent être textuelles (outil Xtext) ou graphiques (outil Sirius).

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

```
process p {
    wd a
    wd b
    ws f2s from a to b
    Ressource a 2
}
```

On remarque bien l'existence d'une coloration par exemple pour : process, wd , ws, f2s, from to et Ressource. Aussi bien que des suggestions pendant la génération de l'exemple.

4 La transformation modèle à modèle SimplePDL2PetriNet

4.1 Principe

EMF permet la génération du code java pour SimplePDL et PetriNet une fois qu'on dispose des fichiers Ecore grâce au modèle appelé genmodel. Ensuite, et en se basant sur les packages engendrés, on peut écrire un code Java qui transforme un modèle de processus en un modèle de réseau de Pétri. La transformation suit le principe suivant:

Une WorkDefinition est équivalente à 4 places: Place_Ready, Place_Started, Place_Running et Place_Finished, 2 transitions: Transition_Start, Transition_Finish et 5 arcs: ArcReadyTostart, ArcStartToStarted, ArcStartToRunning, ArcRunningToFinish et ArcFinishToFinished. Une WorkSequence sera un arc reliant les places Place_Started ou Place_Finished avec les transitions Transition_Start ou Transition_Finish selon la nature de la liaison.

La figure ci-dessous représente un exemple de transformation de l'activité Conception:

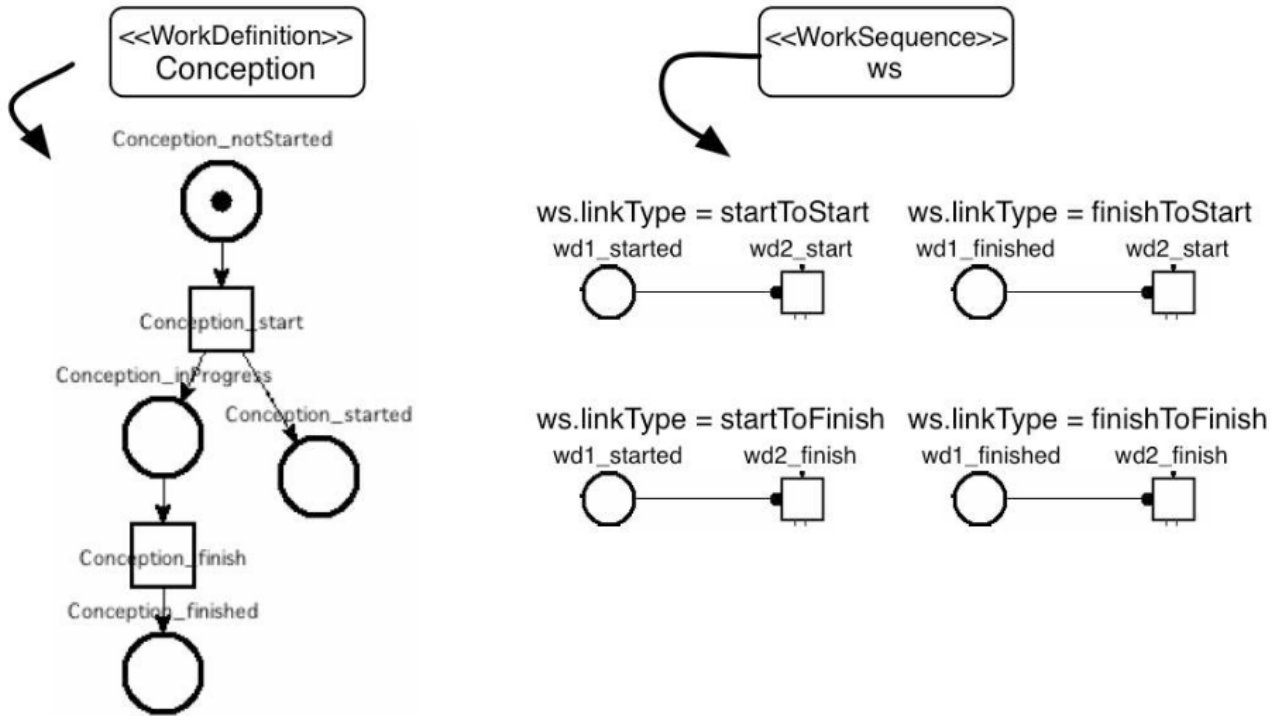


Figure 10 - Principe de transformation d'une activité

Les ressources de leur part sont transformées en places avec comme marquage initial (tokens) la quantité disponible des ressources et les Allocates qui contrôlent l'accès à ces ressources sont transformés en arcs.

4.2 Transformation en java

Nous avons implémenté le code SimplePDL2PetriNet.java qui réalise la transformation d'un modèle en SimplePDL en un modèle PetriNet. Ce code est fourni dans les livrables.

On a essayé cette transformation sur l'exemple suivant :

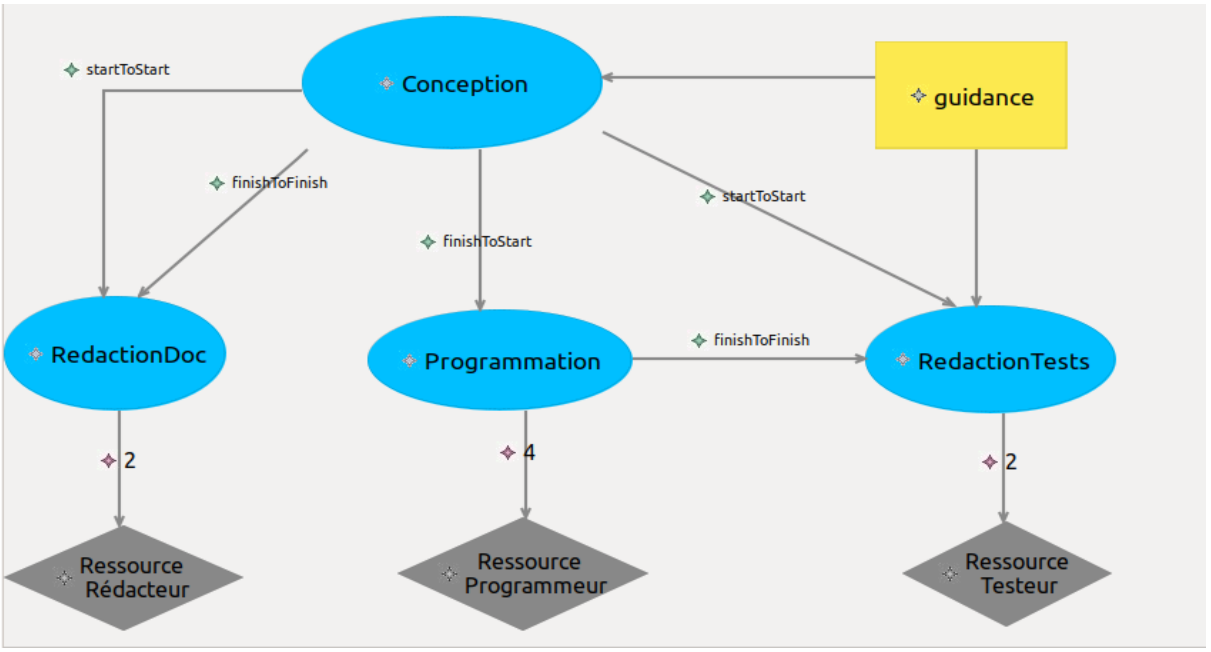
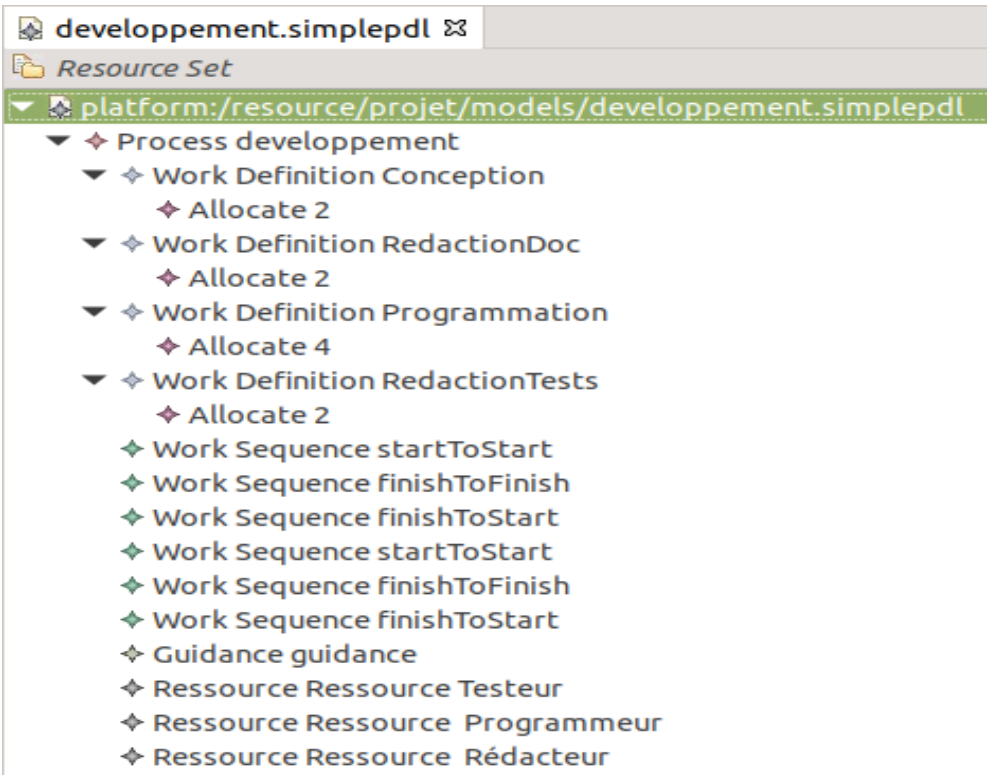


Figure 11 – Modèle de Procédé avec les ressources

Le fichier developpement.simplepdl modélisant ce processus se résume dans l'éditeur arborescent en:



Ensuite, nous avons appliqué la transformation java SimplePDL2PetriNet sur cet exemple et on a pu générer son équivalent en PetriNet: (L'éditeur arborescent était trop long donc on a pris une capture d'écran d'une partie)

Figure 13 - Résultat de la transformation appliqué sur l'exemple ci-dessus

et après à l'aide de la transformation ToTina nous avons generé le fichiet developpemnt.net dont une partie est donnée sur la figure suivant:

developpement.net

```
net developpement
pl ConceptionNot_Started (1)
pl ConceptionStarted (0)
pl ConceptionIn_Progress (0)
pl ConceptionFinished (0)
pl RedactionDocNot_Started (1)
pl RedactionDocStarted (0)
pl RedactionDocIn_Progress (0)
pl RedactionDocFinished (0)
pl ProgrammationNot_Started (1)
pl ProgrammationStarted (0)
pl ProgrammationIn_Progress (0)
pl ProgrammationFinished (0)
pl RedactionTestsNot_Started (1)
pl RedactionTestsStarted (0)
pl RedactionTestsIn_Progress (0)
pl RedactionTestsFinished (0)
pl ressourceTesteur (12)
pl ressourceProgrammeur (2)
pl ressourceRedacteur (4)
tr Conceptionstart ConceptionNot_Started -> ConceptionStarted ConceptionIn_Progress
tr Conceptionfinish ConceptionIn_Progress -> ConceptionFinished
tr RedactionDocstart RedactionDocNot_Started ConceptionStarted?1 -> RedactionDocStarted RedactionDocIn_Progress
tr RedactionDocfinish RedactionDocIn_Progress ConceptionFinished?1 -> RedactionDocFinished
tr ProgrammationStart ProgrammationNot_Started ConceptionFinished?1 -> ProgrammationStarted ProgrammationIn_Progress
```

4.3 En ATL

Il est aussi possible de faire la transformation SimplePDL vers PetriNet en utilisant ATL, un langage de transformation de modèles qui rapporte plus d'efficacité à la transformation.

Là encore, le principe est très semblable à celui utilisé lors de la transformation java avec des différences de syntaxe, on commence par traduire le Process en un PetriNet portant le même nom. Ensuite, nous transformons dans le code chaque WorkDefinition en 4 places: p_ready , p_started,p_running et p_finished, 2 transitions: t_start et t_finish et 5 arcs: a_ready_to_start, a_start_to_started, a_start_to_running, a_running_to_finish et a_finish_to_finished. Ensuite, chaque WorkSequence sera un arc reliant p_started et p_finished avec t_start ou t_finish et finalement chaque Ressource est converti en une place marqué par sa quantité avec les Demandes reliant ces ressources avec les activités.

Nous avons appliqué cette transformation sur l'exemple du sujet (fichier sujet-pdl.xmi) et on a obtenu le résultat suivant :
En version .dot:

Figure 15 - sujet-pdl.dot

en version Tina:

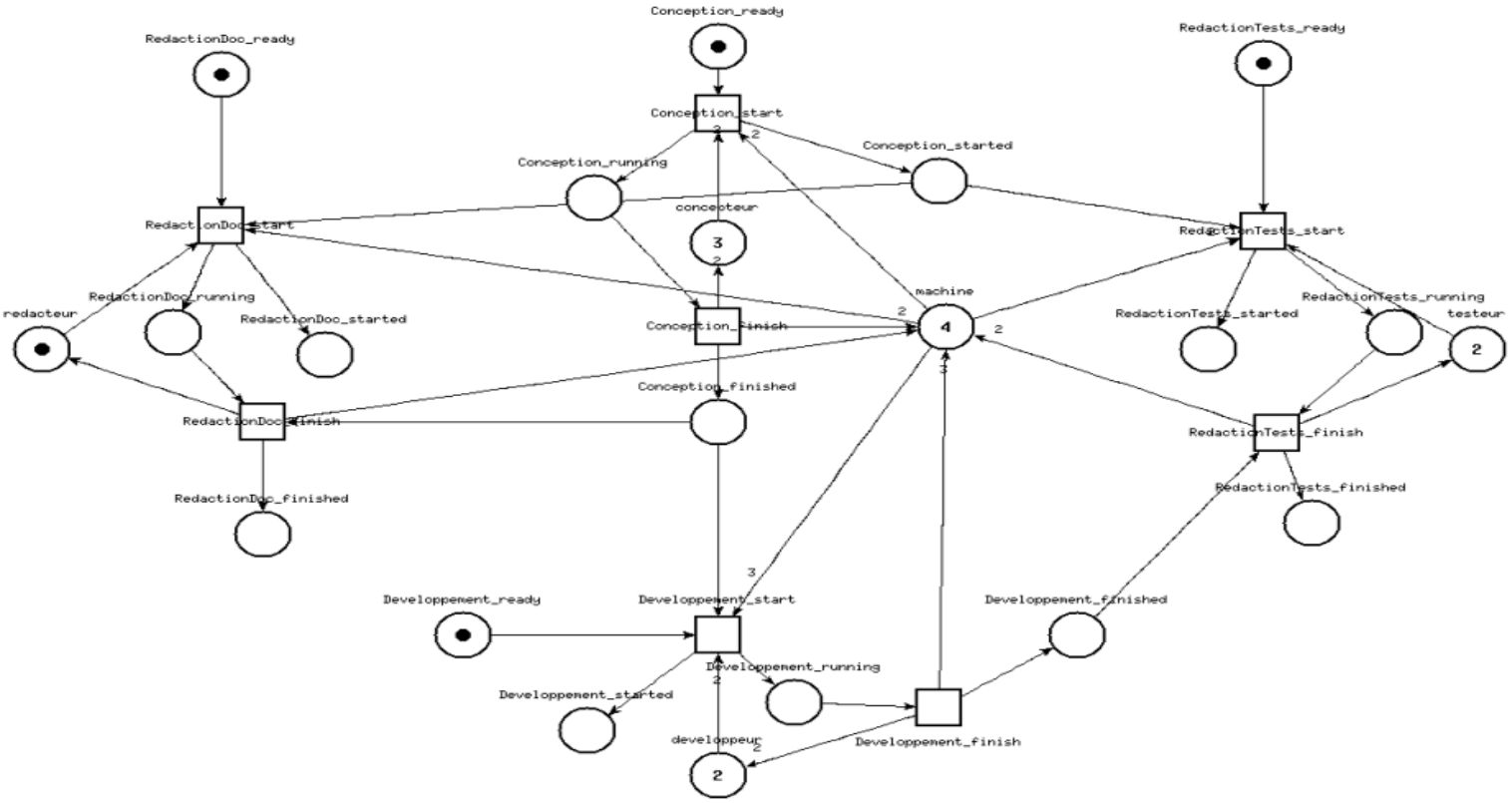


Figure 16 - sujet-pdl.net

5 Propriétés LTL et terminaison d'un processus

Il est important de vérifier la terminaison de chaque processus, pour cela on crée un fichier LTL, avec qui on teste notre tina. Ce fichier LTL est engendré à partir du modèle SimplePDL. Pour vérifier la terminaison, on test l'existence du dead

Le but d'utiliser la propriété **- <> finished** plutôt que **<> finished** est d'avoir la réponse **False** et d'avoir le chemin à suivre pour avoir une terminaison, au lieu d'avoir un résultat **True** et sans aucun chemin à suivre, et ce qu'on remarque d'après le résultat dans la figure ci-dessous.

```
op finished = ConceptionFinished /\ RedactionTestsFinished /\ RedactionDocFinished /\ ProgrammationFinished;

[] (finished => dead);
[] <> dead ;
[] (dead => finished);
- <> finished;
```

```
houkhnin@lannister:~/2Annee/IDM/TinaDemo/bin$ ./tina developpement.net developpement.ktz
# net developpement, 16 places, 8 transitions                          #
# bounded, not live, not reversible                                    #
# abstraction        count      props      psets       dead       live #
#      states           26         16         26          1          1 #
# transitions           47          8          8          0          0 #
houkhnin@lannister:~/2Annee/IDM/TinaDemo/bin$ ./selt -p -S developpement.scn developpement.ktz -prelude developpement.ltl
Selt version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 26 states, 47 transitions
0.002s

- source developpement.ltl;
operator finished : prop
TRUE
TRUE
TRUE
FALSE
  state 0: ConceptionNot_Started ProgrammationNot_Started RedactionDocNot_Started RedactionTestsNot_Started
  -Conceptionstart->
  state 1: ConceptionIn_Progress ConceptionStarted ProgrammationNot_Started RedactionDocNot_Started RedactionTestsNot_Started
  -Conceptionfinish->
  state 2: ConceptionFinished ConceptionStarted ProgrammationNot_Started RedactionDocNot_Started RedactionTestsNot_Started
  -Programmationstart->
  state 3: ConceptionFinished ConceptionStarted ProgrammationIn_Progress ProgrammationStarted RedactionDocNot_Started RedactionTestsNot_Started
  -Programmationfinish->
  state 4: ConceptionFinished ConceptionStarted ProgrammationFinished ProgrammationStarted RedactionDocNot_Started RedactionTestsNot_Started
  -RedactionDocstart->
  state 5: ConceptionFinished ConceptionStarted ProgrammationFinished ProgrammationStarted RedactionDocIn_Progress RedactionDocStarted RedactionTestsNot_Started
  -RedactionDocfinish->
  state 6: ConceptionFinished ConceptionStarted ProgrammationFinished ProgrammationStarted RedactionDocFinished RedactionDocStarted RedactionTestsNot_Started
  -RedactionTestsstart->
  state 7: ConceptionFinished ConceptionStarted ProgrammationFinished ProgrammationStarted RedactionDocFinished RedactionDocStarted RedactionTestsIn_Progress RedactionTestsStarted
  -RedactionTestsfinish->
  state 8: L.dead ConceptionFinished ConceptionStarted ProgrammationFinished ProgrammationStarted RedactionDocFinished RedactionDocStarted RedactionTestsFinished RedactionTestsStarted
  -L.deadlock->
  state 9: L.dead ConceptionFinished ConceptionStarted ProgrammationFinished ProgrammationStarted RedactionDocFinished RedactionDocStarted RedactionTestsFinished RedactionTestsStarted
  [accepting all]
0.003s
```

On remarque que la Troisième propriété [] (dead => finished) n'est pas vérifiée dans notre cas. Cela est dû à un interblocage au niveau du manque de ressource. En effet, en suivant le contre-exemple donné, on s'aperçoit qu'à un moment on ne peut plus avancer car RédactionTestsFinishished attend la fin du développement qui ne peut pas même pas commencer car on ne dispose pas de nombre de machines suffisant. Et donc on se bloque.

6 Conclusion

Ce projet nous a permis de mettre en pratique les notions vues en cours/TP pour intervenir sur l'ensemble des éléments de la chaîne de transformation SimplePDL -> PetriNet. Ceci nous a reflété l'importance de la bonne conception des modèles dans un projet ainsi que les possibilités de model-checking au travers de la boîte à outil Tina.

Il nous a dévoilé aussi l'importance de la modélisation dans la résolution des problèmes ainsi que la maîtrise des processus de développement logiciel en s'appuyant sur des techniques de modélisation, de conception, de développement et de gestion de projet. Ces techniques doivent permettre la conception d'applications modernes, nécessitant l'intégration de composants hétérogènes.