Ingénierie dirigée par les Modèles

Mini-Projet

HPCBD - B2

Issam Habibi - Badr Sajid

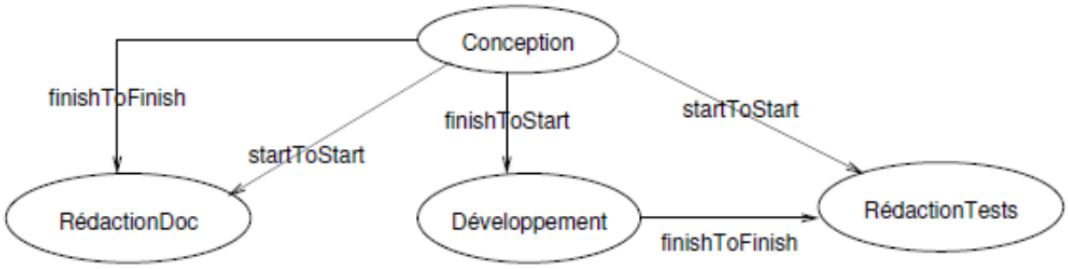
2020

Contents

Introduction

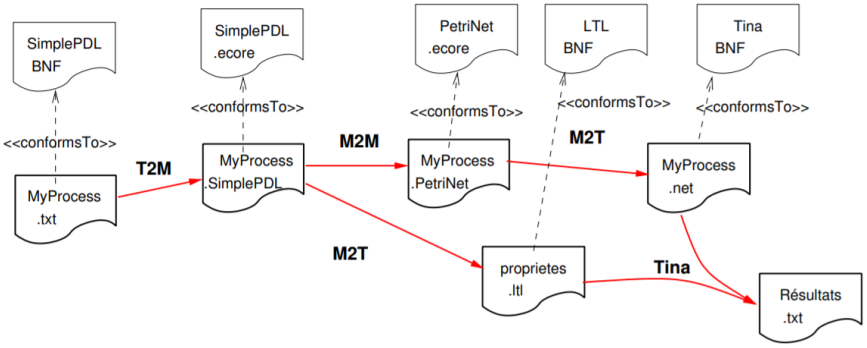
Il est toujours important de détecter les erreurs éventuelles dans un projet le plus tôt possible et avant d'entamer ses phases les plus profondes, c'est pour cela qu'on peut considérer la méta modélisation et sa simulation comme étape importante pour la validation des modèles décrivant le projet.

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



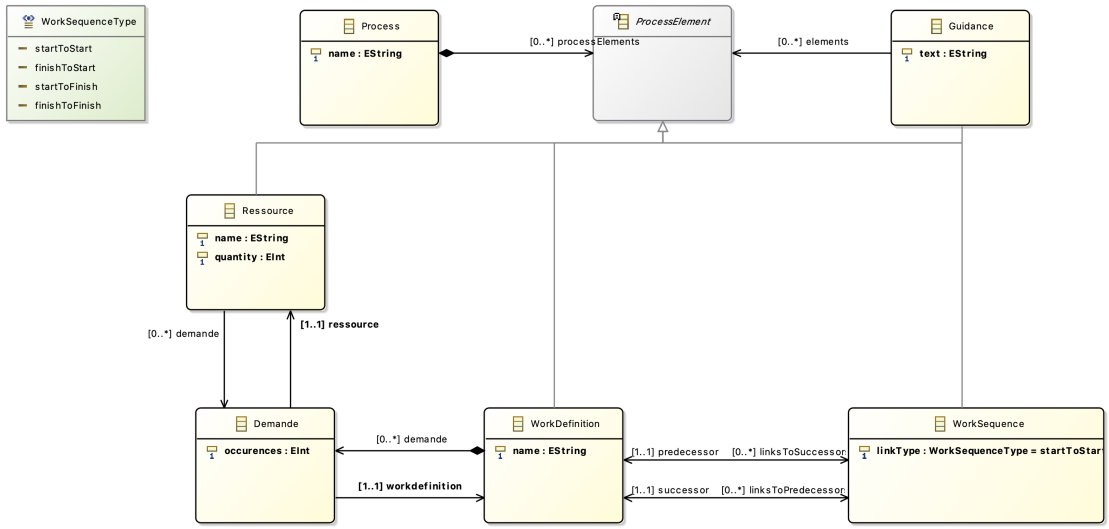
Nous allons essayer donc de produire une chaîne de vérification de modèles de processus SimplePDL en général pour vérifier si un processus peut se terminer ou pas. Nous passerons par plusieurs étapes afin de réaliser cette vérification:

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



Le métamodèle simplePDL

La figure ci dessous représnete notre métamodèle **SimplePDL**



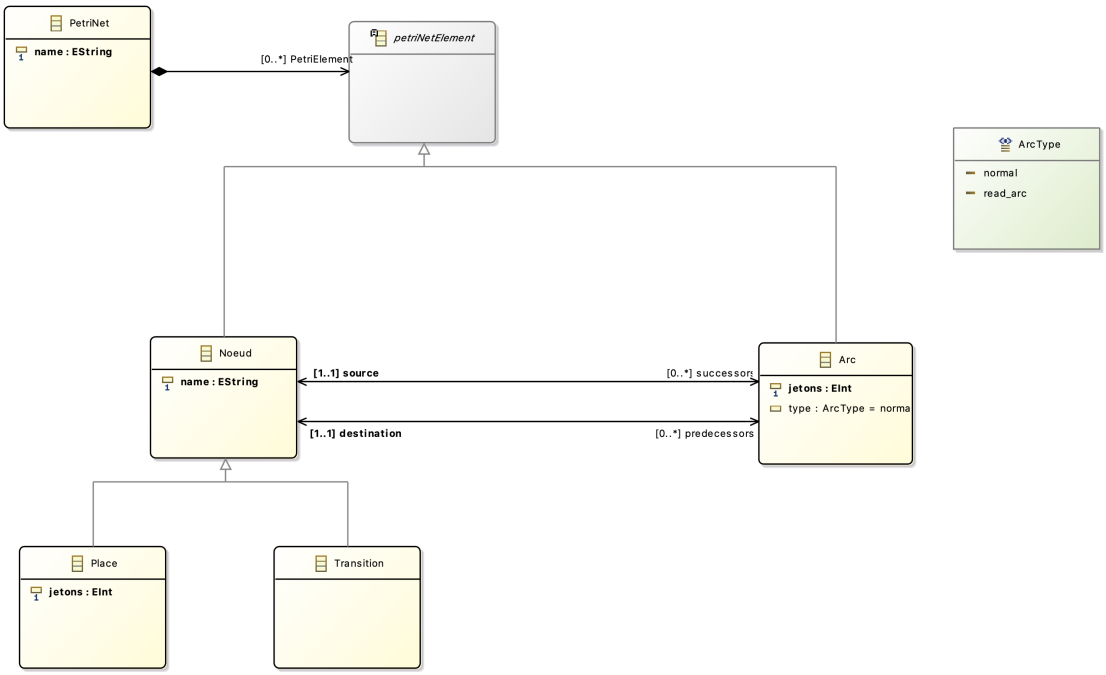
On retrouve donc le concept de processus (**Process**) qui se compose des activités (**WorkDefinition**), des dépendences (**WorkSequence**) et des ressources (**Ressource**). Les activités sont les tâches que le processus doit réaliser, les dépendences relient les activités entre eux avec des relations d'ordonnancement représentées par l'énumération (**WorkSequenceType**) qui caractérise le démarrage et la fin d'une certaine activité et possèdant quatre valeurs (startToStart, finishTosTart, startToFinisih, finishToFinish). Dans notre cas concrêt, l'activité "Conception" est par exemple reliée à l'activité "RédactionTests" avec la relation **startToStart**: la rédaction des tests ne peut commencé que dans le cas où la conception a commencé.

La réalisation d'une activité peut nécessiter plusieurs ou aucune ressource. Ces ressources sont caractérisées par leurs types (l'attribut name) et leurs quantités disponibles (l'attribut quantity) . Par exemple, la réalisation de l'activité programmation peut nécessiter trois développeurs (acteur humain) et quatres machines (outil machine). Cependant, une activité doit demander une quantité disponible des ressources d'où le recours au classe (**Demande**) qui contrôle ceci.

Plusieurs règles peuvent être définies sur le métamodèle SimplePDL: La validité des noms du processus et de ses composants (y compris le fait que deux sous activités d'un même procesuss ne peuvent pas avoir le même nom), la non-refléxivité des dépendences, la disponibilité des ressources demandées par les activités, la quantité non nulle des ressources disponible dans un processus, l'unicité des noms des ressources...etc.

Le métamodèle PetriNet

La figure ci dessous représnete notre métamodèle **PetriNet**



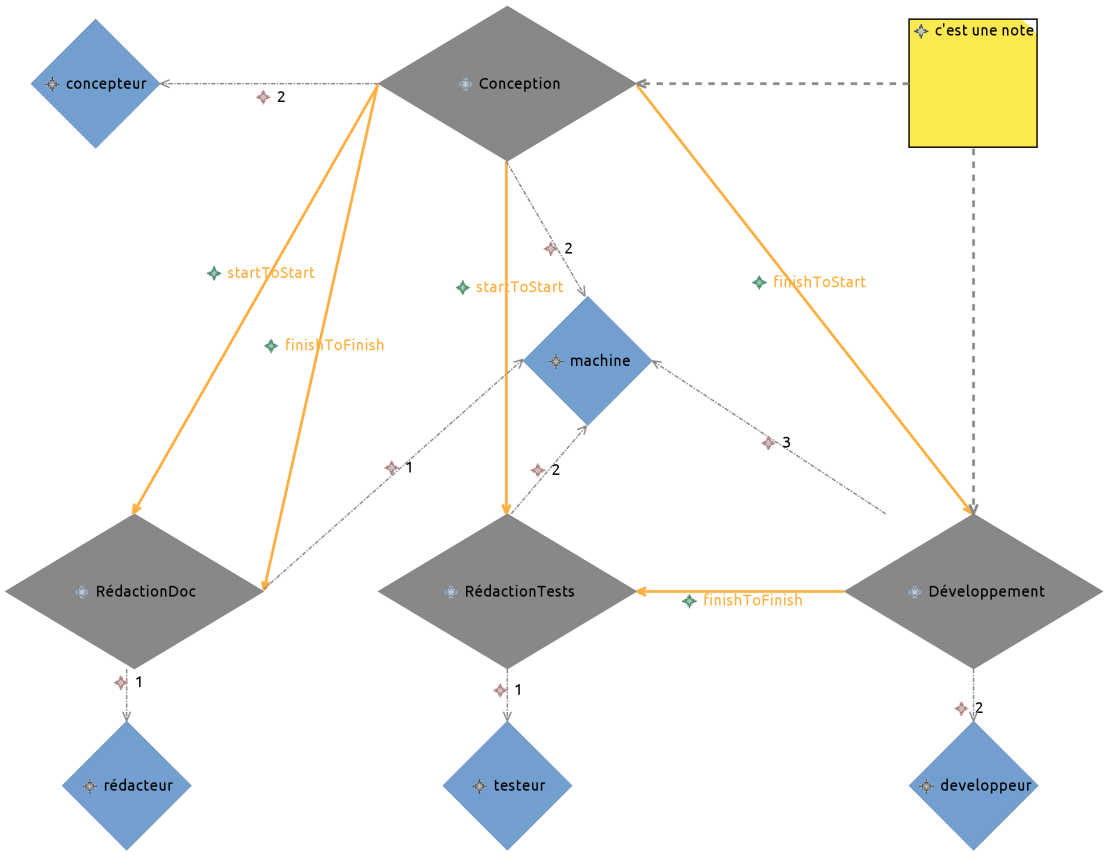
On retrouve dans notre réseau de Petri le concept principal (**PetriNet**) qui se compose des arcs (**Arc**) reliant des noeuds (**Noeud**) qui sont soit des places (**Place**) ou des transitions (**Transition**). Les places contiennent un nombre initial de jetons (l'attribut jetons), les arcs possèdent deux types possibles regroupés dans l'énumération (**ArcType**): normal (qui connecte une place et une transition) et readArc (qui connecte une place d'entrée à une transition). Les arcs possèdent aussi un nombre de jetons (l'attribut jetons) qui sera consommé d'une place vers une transition oubien l'inverse.

Le réseau de Petri doit respecter plusieurs contraintes: Le marquage initiale (le nombre initial des jetons dans les places) doit être valide, la consommation des jetons d'un noeud vers un autre doit aussi être valide, les arcs ne peuvent pas connecter des places avec des places ou des transitions avec des transitions, les noeuds ne peuvent pas être nommé de la même manière, un noeud ne peut pas être isolé (il est obligatoirement suivi ou précédé par un noeud ou les deux en même temps)...etc.

L'éditeur graphique simplePDL

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 Comme le montre la figure ci-dessous.



Définition d'une syntax 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...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
}
```

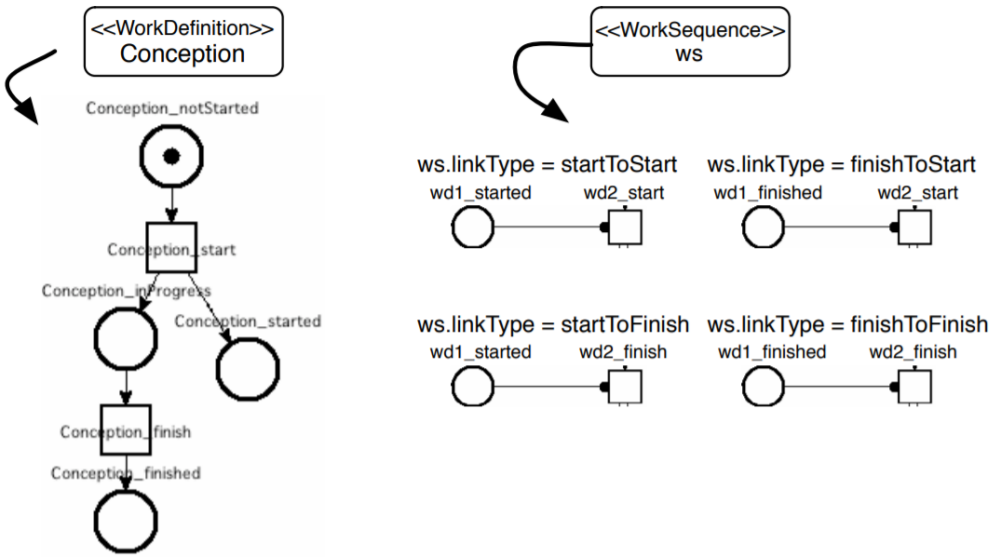
On remarque bien l'existence d'une coloration par exemple pour : process, wd , ws, f2s, from to et Ressource. Aussi bien que des suggestions pendant la génération de l'exemple.

La transformation SimplePDL2PetriNet en utilisant EMF/Java

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:

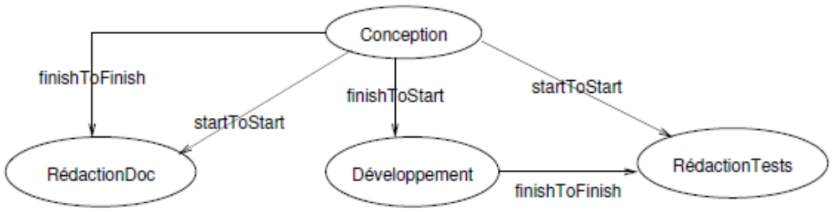
Une **WorkDefinition** est équivalente à 4 places: PlaceReady, PlaceStarted, PlaceRunning et PlaceFinished, 2 transitions: TransitionStart, TransitionFinish et 5 arcs: ArcReadyTostart, ArcStartToStarted, ArcStartToRunning, ArcRunningToFinish et ArcFinishToFinished. Une **WorkSequence** sera un arc reliant les places PlaceStarted ou PlaceFinished avec les transitions TransitionStart ou TransitionFinish selon la nature de la liaison.

La figure ci-dessous représente un exemple de transformation pour l'activité **Conception**.



Les **Ressources** de leurs part sont transformées en places avec comme marquage initial (jetons) la quantité disponible des ressources et les **Demandes** qui contrôlent l'accès à ces ressources sont transoformées en arcs.

On a essayé cette transformation sur l'exemple du sujet (fichier **sujet-pdl.xmi** modélisant ce processus:



Avec les ressources suivantes:

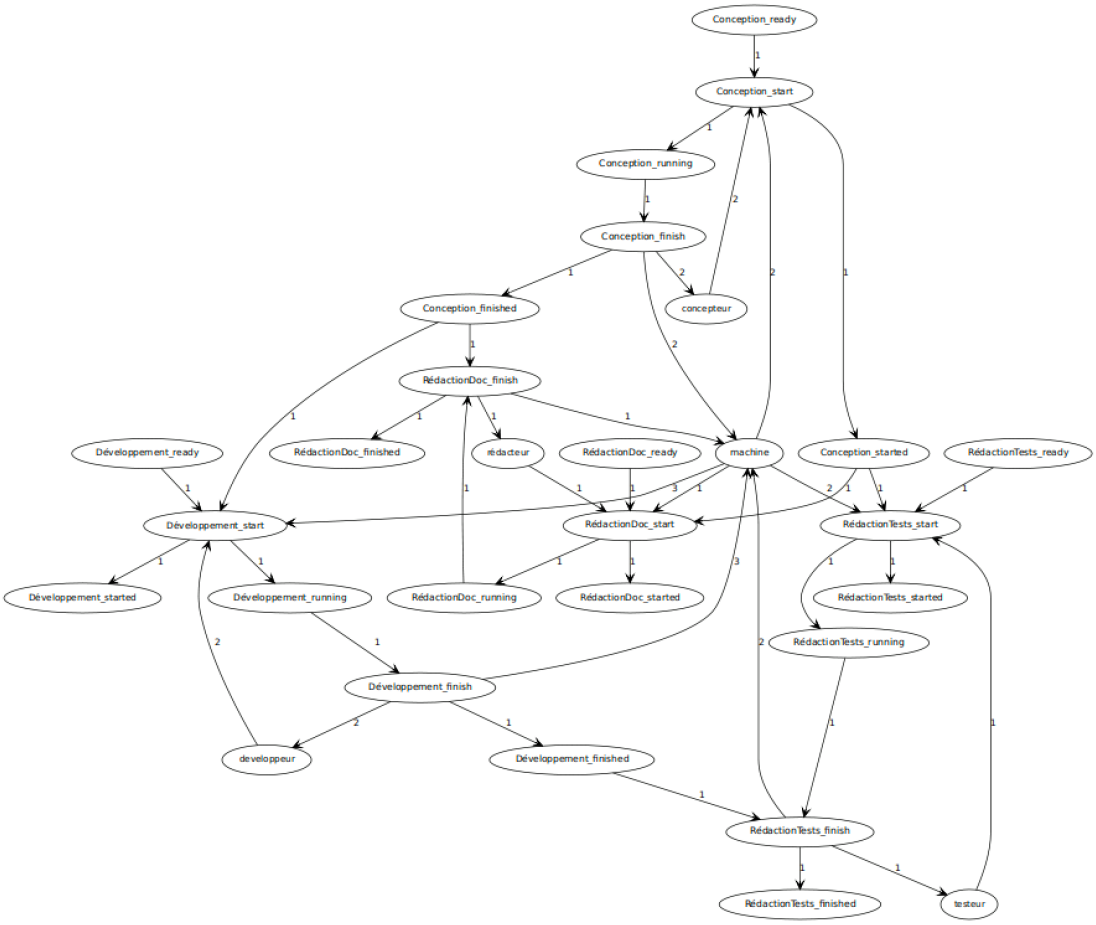
	Quantité	Conception	RédactionDoc	Programmation	RédactionTest
concepteur	3	2			
développeur	2			2	
machine	4	2	1	3	2
rédacteur	1		1		
testeur	2				1

qui se résume dans l'éditeur arborescent en:

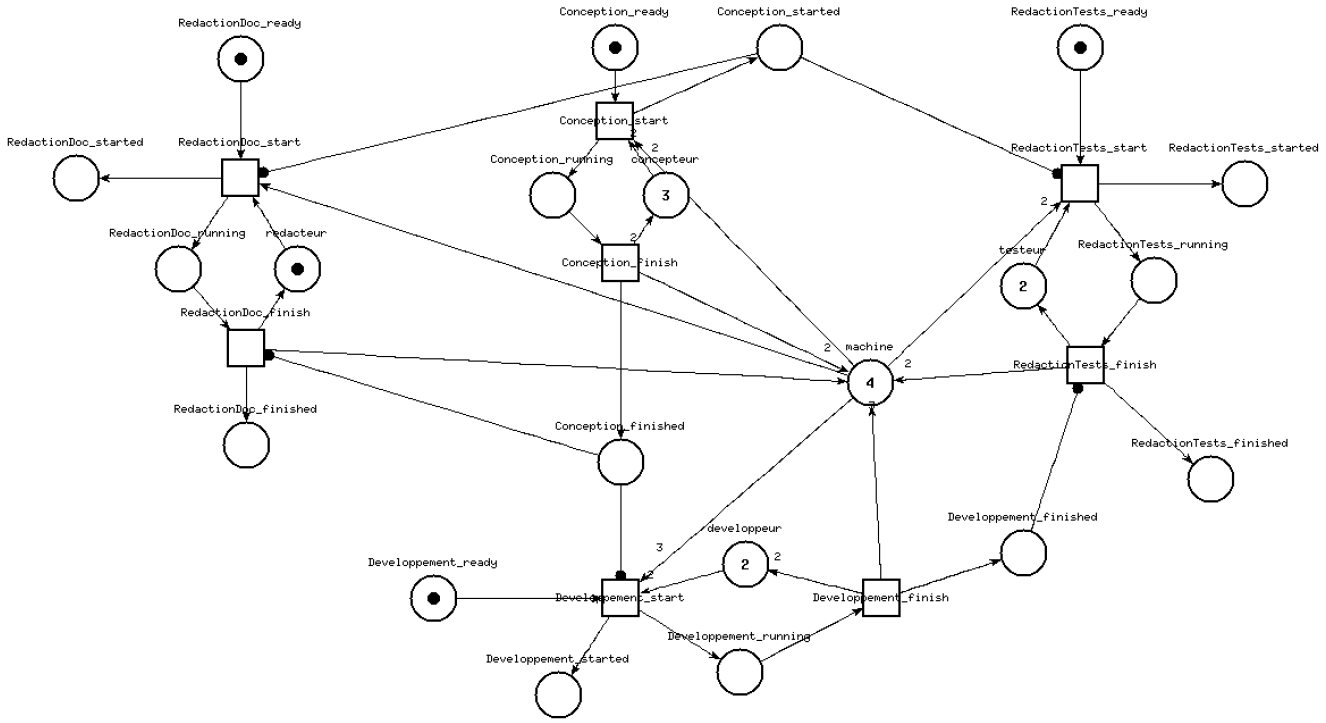
- platform:/resource/fr.n7.simplePDL/sujet-pdl.xmi
  - Process Process
    - Work Definition Conception
      - Demande 2
      - Demande 2
    - Work Definition RédactionDoc
      - Demande 1
      - Demande 1
    - Work Definition Développement
      - Demande 2
      - Demande 3
    - Work Definition RédactionTests
      - Demande 2
      - Demande 1
    - Work Sequence finishToFinish
    - Work Sequence startToStart
    - Work Sequence finishToStart
    - Work Sequence startToStart
    - Work Sequence finishToFinish
    - Ressource concepteur
    - Ressource developpeur
    - Ressource machine
    - Ressource rédacteur
    - Ressource testeur

Ensuite, nous avons appliqué la transformation java SimplePDL2PetriNet sur cet exemple et on a pu généré son équivalent en PetriNet: (L'éditeur arborescent était trop long donc on l'a divisé sur 2 parties)

Ou encore, sous une représentation graphique (.dot)



Oubien, avec la représentation graphique Tina.



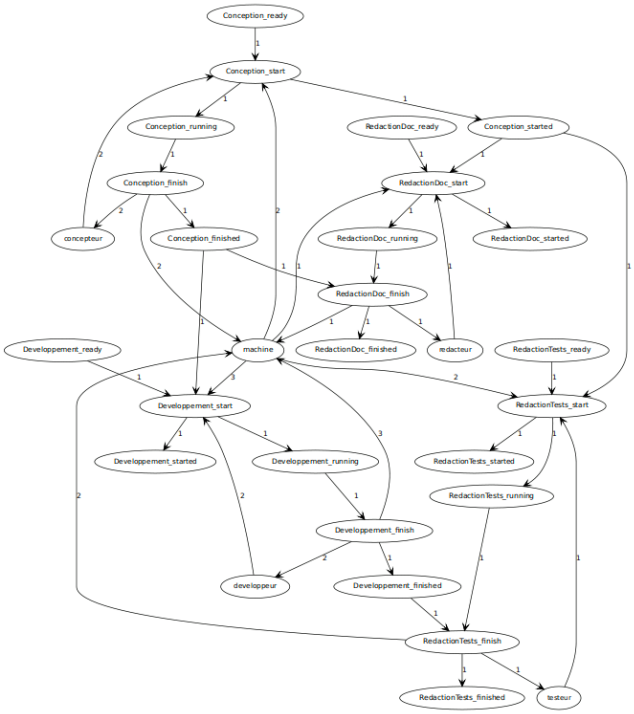
La transformation SimplePDL2PetriNet en utilisant ATL

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

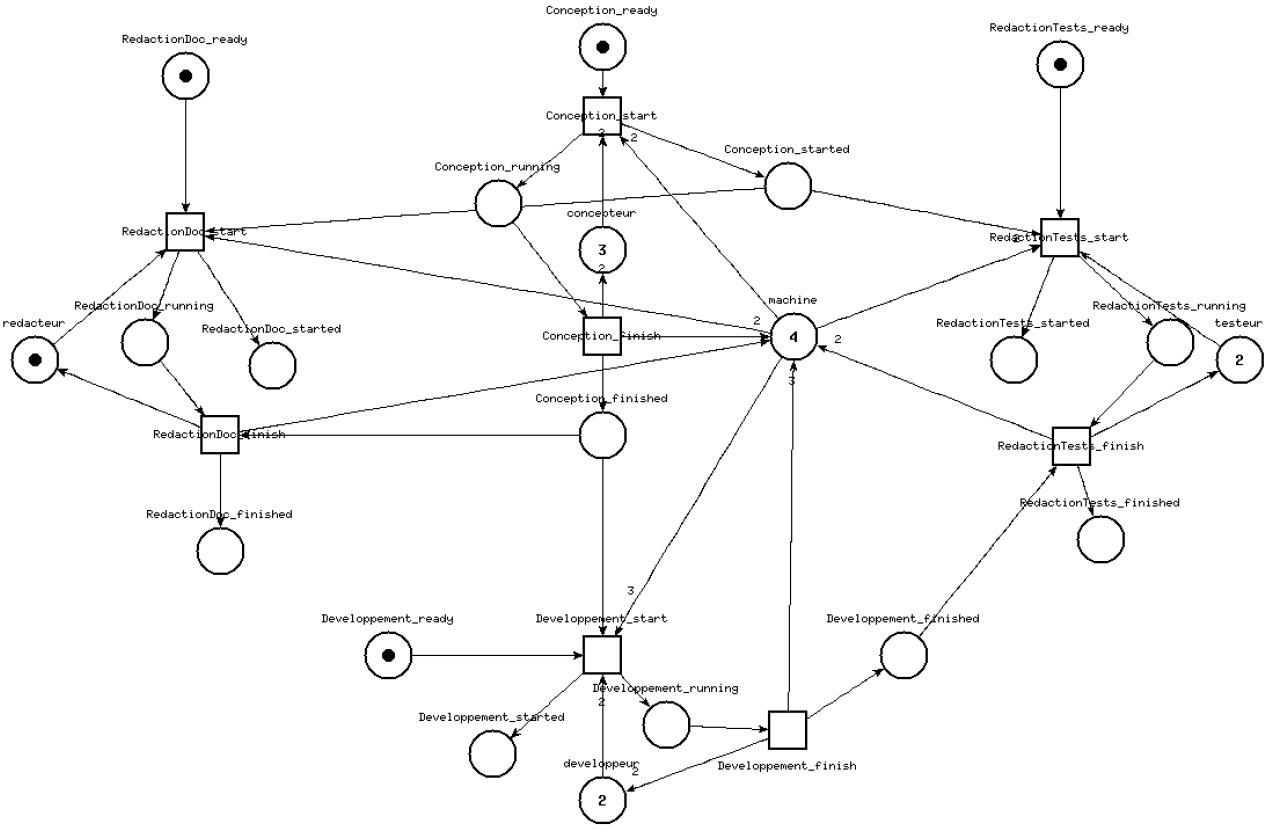
Là encore, le principe est très semblable à celui utilisé lors de la transformation java avec des différence de syntaxe, on commence par traduire le Process en un PetriNet portant le même nom. Ensuite, nous transformons dans le code chaque **WorkDefinition** en 4 places: p_ready , p_started,p_running 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 relian 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** reliants ces ressourcs 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:



en version tina:

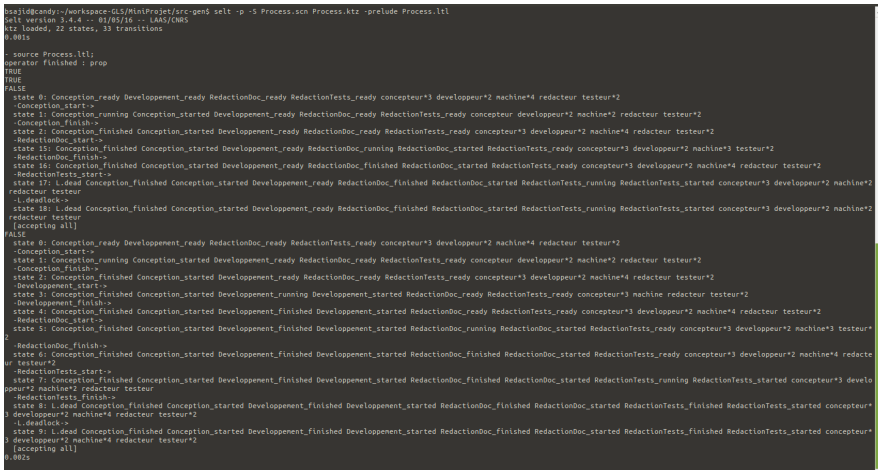


Propriétés LTL et terminaison d'un processus

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.

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



On Remarque que la Troisième propriété [] **(dead => finished)** n'est pas vérifier dans notre cas. Cela est du 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 RédactionTests_finish 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.

Conclusion

Ce projet nous a permis de mettre en pratique les notions vu en cours/TP pour intervenir sur l'ensemble des éléments de la chaîne de transformation SimplePDL -> PetriNet. Ceci nous a reflèté 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.