Chaîne de vérification de modèles de processus

El Bouzekraoui Younes — MDAA Saad

Département Sciences du Numérique - Deuxième année
2020-2021

Contents

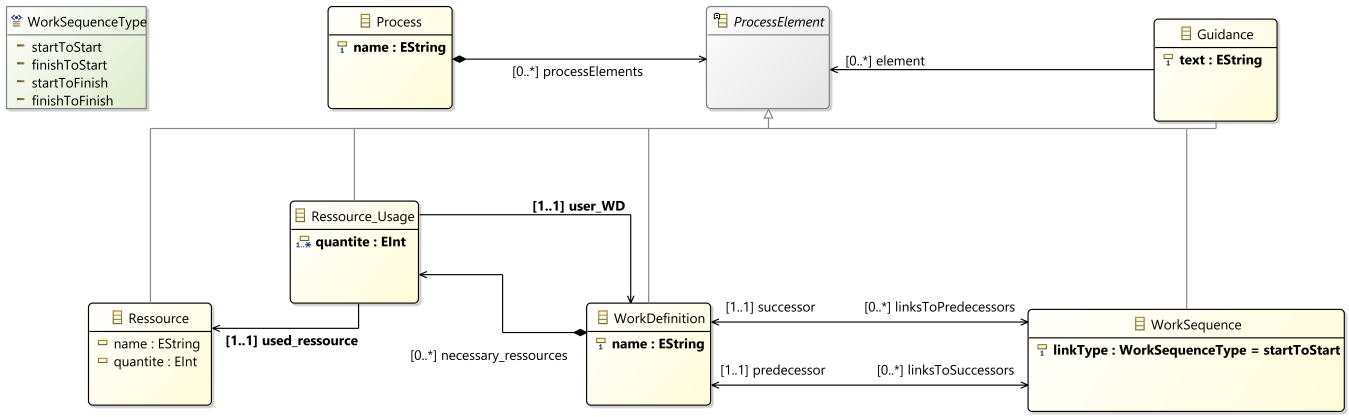
1 Introduction

Ce mini-projet consiste à produire une chaîne de vérification de modèles de processus **SimplePDL** dans le but de vérifier leur cohérence, en particulier pour savoir si le processus décrit peut se terminer ou non. Pour répondre à cette question, nous utilisons les outils de model-checking définis sur les réseaux de **Petri** au travers de la boîte à outils Tina. Il nous faudra donc traduire un modèle de processus en un réseau de Petri.

2 Le Métamodèle SimplePDL

2.1 Modelisation du métamodèle

On a choisi de définir le Métamodèle SimplePDL comme suit (voir **SimplePDL.ecore**):



Notre Métamodèle SimplePDL est composé de :

- Process : le processus qu'on modélise
- ProcessElement : une interface pour faire abstraction sur les éléments d'un processus
- Guidance : des notes écrites
- Workdefinition : une activité ayant un nom de type EString
- WorkSequence : une dépendance entre deux activité
- Ressource : une ressource ayant un nom de type EString et une quantité de type EInt
- Ressource_Usage : le lien entre une activité et une ressource

Pour prendre en compte l'utilisation des ressources on a ajouté les deux classes : **Ressource** et **Ressource_Usage** qui héritent de **ProcessElement**, et on a ajouté un attribut a Workdefinition : **necessary_ressources** qui est une liste des ressources nécessaires pour une activité. Un objet **Ressource_Usage** contient un lien vers une activité et un lien vers une ressource et la quantité consommé.

2.2 Contraintes OCL

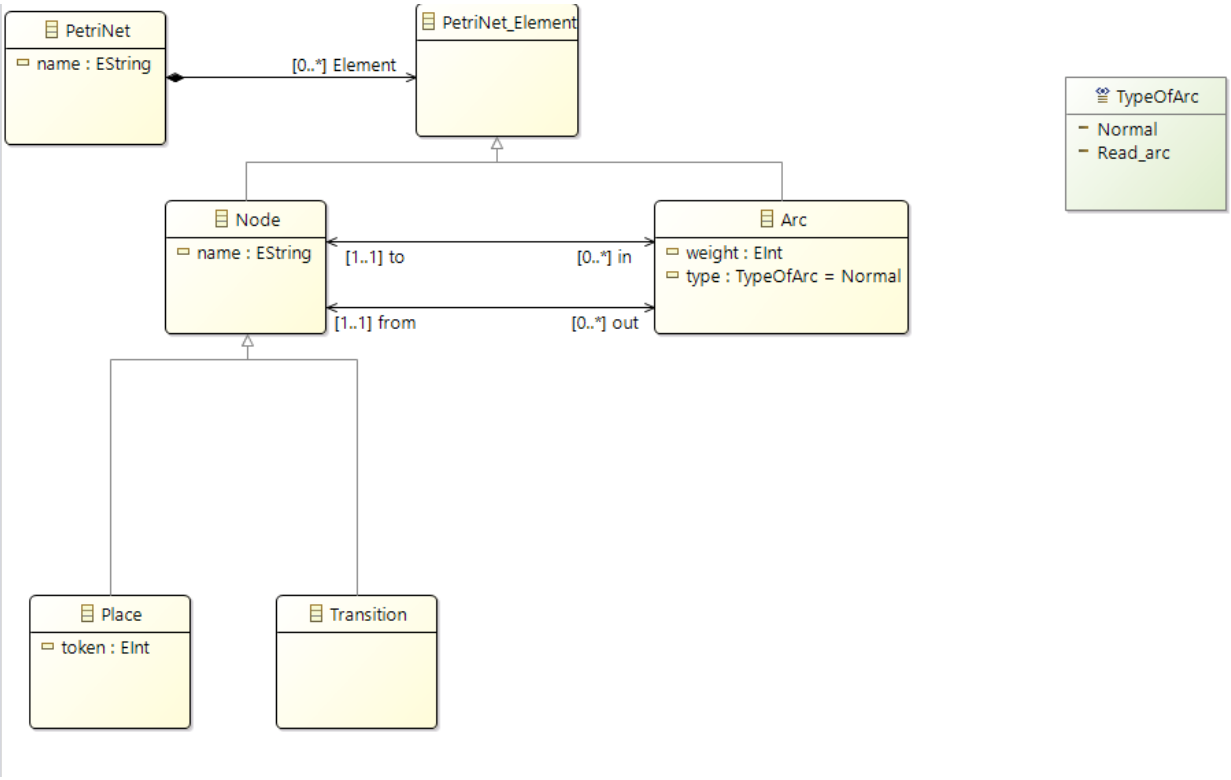
Afin de capturer plus de contraintes On a ajouté les règles ocl suivantes (voir **SimplePDL.ocl**):

- une dépendance ne peut pas être réflexive.
- deux sous-activités différentes d'un même processus ne peuvent pas avoir le même nom.
- le nom d'une activité doit être composé d'au moins un caractère.
- le nom d'une resource doit être composé d'au moins un caractère.
- deux ressources ne peuvent pas avoir le même nom.
- la quantité d'une ressource doit être ≥ 0
- la quantité d'une ressource usé doit être > 0
- la quantité d'un ressource utilisé par une activité doit être inférieur ou égale à la quantité disponible

3 Le Métamodèle PetriNet

3.1 Modelisation du métamodèle

On a choisi de définir le Métamodèle SimplePDL comme suit (voir **PetriNet.ecore**):



Notre Métamodèle PetriNet est composé de :

- PetriNet : le réseau petri qu'on modélise
- PetriNet_Element : une interface pour faire abstraction sur les éléments d'un réseau petri
- Node : une interface pour faire abstraction sur une place et une transition
- Arc : modélise le lien entre une place et une transition de type Normal ou Read_arc
- Place
- Transition

3.2 Contraintes OCL

Afin de capturer plus de contraintes On a ajouté les règles ocl suivantes (voir **PetriNet.ocl**):

- un arc doit suivre le modele suivant place arc transition arc place.
- le poids d'un arc ne peut pas être nul (sinon l'arc ne sert à rien).
- le nombre des jetons dans une place est positive.
- le nom d'une node doit être composé d'au moins un caractère.
- deux nodes différente peuvent pas avoir le même nom.
- le nom d'un réseau Petri doit être composé d'au moins un caractère.

4 Un éditeur graphique SimplePDL

5 Syntaxe concrète textuelle de SimplePDL avec Xtext

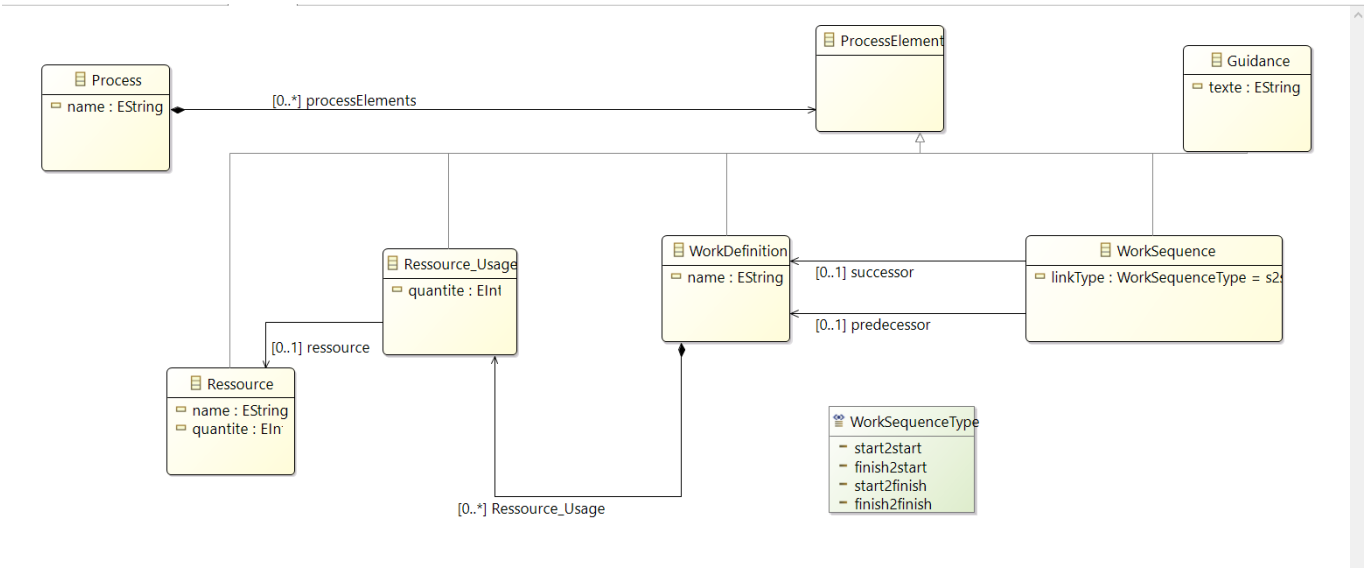
Xtext consiste à définir une syntaxe concrète textuelle pour un DSML au travers d'une grammaire, et il permet aussi de construire un métamodèle conforme à la syntaxe concrète décrite

5.1 Définition de la syntaxe PDL1:

(voir **PDL1.xtext**)
Un exemple de la syntaxe textuelle généré est donnée dans l'exemple ci-dessous.

```
process dev {
    rs concepteur qt 3
    rs developpeur qt 2
    rs machine qt 4
    rs redacteur qt 2
    rs testeur qt 2

    wd Conception needs to get 2 of concepteur get 1 of machine
    wd RedactionDoc needs to get 1 of machine get 1 of redacteur
    wd Programmation needs to get 2 of developpeur get 3 of machine
    wd RedactionTests needs to get 2 of machine get 1 of testeur

    ws f2f from Conception to RedactionDoc
    ws s2s from Conception to RedactionDoc
    ws f2f from Conception to Programmation
    ws s2s from Conception to RedactionTests
    ws f2f from Programmation to RedactionTests
}
```

Le métamodèle généré à partir de cette syntaxe textuelle est donnée ci-dessous :



Ce métamodèle est différent de celui qu'on definit au début.

6 Transformation SimplePDL vers PetriNet en utilisant EMF/Java

EMF nous permet de générer les classes Java a partir des modèles SimplePDL et PetriNet, ce qui nous permet de définir une transformation de SimplePDL vers PetriNet en utilisant Java (voir **SimplePDL2PetriNet.java**)

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

Le principe de la transformation d'un modèle SimplePDL en Réseau de Pétri est le suivant :

- Un élément Process devient un élément PetriNet.
- Une WorkDefinition devient 4 places ready, started, running et finished et deux transitions start et finish.
- 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).
- Une Ressource devient une place, le nombre de jetons indique la quantité
- Un Ressource_Usage devient deux arc , un arc (allocation) dont la multiplicité est la quantité utilisé allant de la place ressource vers la transition start de l'activité . et l'autre arc (déallocation)allant de la transition finish vers la place ressource

7 Transformation SimplePDL vers PetriNet en utilisant ATL

ATL est un langage et une boîte à outils de transformation de modèles. Dans le domaine de l'ingénierie pilotée par les modèles, ATL fournit des moyens de produire un ensemble de modèles cibles à partir d'un ensemble de modèles source.

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

On a suivit les mèmes principes décrit dans la transformation java, on a définit les règles suivantes : (voir **SimplePDL2PetriNet.atl**).
Vue qu'on a pas définit précédemment une référence opposite entre PetriNet et PetriNetElement l'opération suggéré lors du tp:

```
net <- wd.getProcess())
```

n'est pas possible, donc on a ajouté les places / transitions / arc crées lors de la règle

```
Process2PetriNet {..}
```

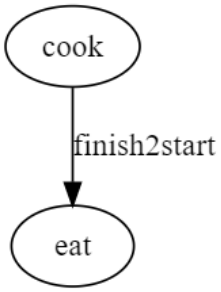
en utilisant

```
thisModule.resolveTemp(..)
```

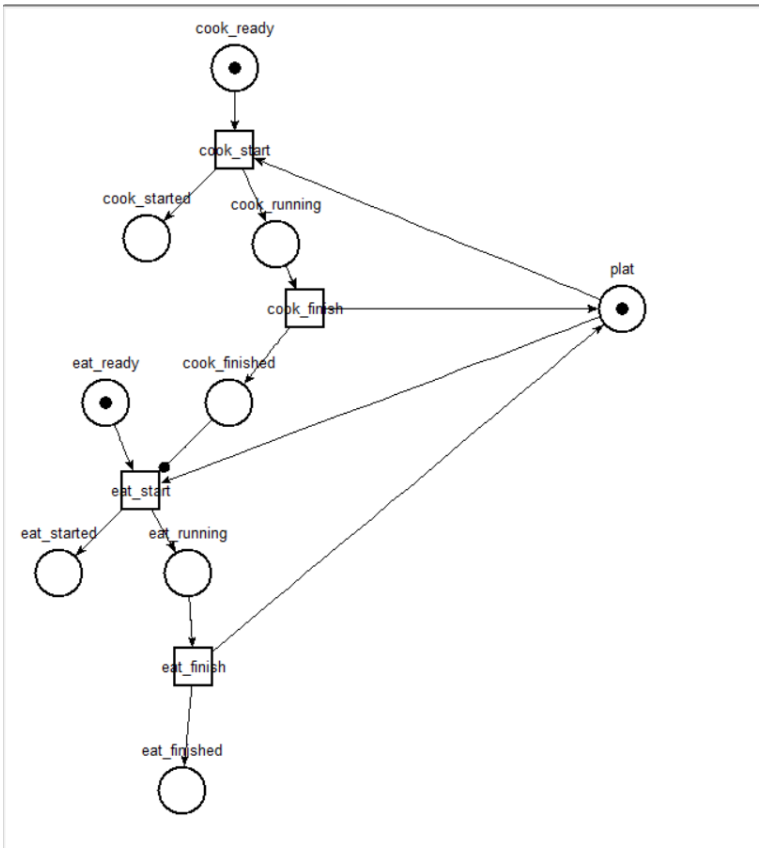
8 Tests de la transformation M2M

8.1 Modèle simple

On commence par tester la transformation sur un modèle simple (voir **pdl-simple.xmi**), il est composé de 2 activités : (cook et eat) ,une dépendance de type finishTostart entre les 2 activités, et une ressource plat utilisé par cook et eat.



la transformation nous donne le modèle Petrinet suivant:



en utilisant le stepper analyser de l'outil netdraw notre modèle Petrinet semble avoir le comportement correcte.

8.1.1 Validation de la transformation avec l'outil selt

On génère les propriétés de sûreté du modèle conforme a SimplePDL et on vérifie la terminaision du process en utilisant l'outil Acceleo (**voir toLTL.mtl**) pour vérifier que les invariants sur le modèle de processus sont préservés sur le modèle de réseau de Petri correspondant.
fichier simple.ltl généré :

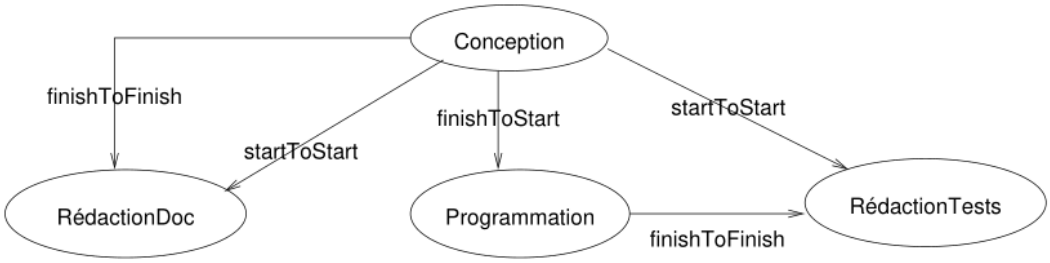
```
op finished = cook_finished /\  eat_finished  ;
[] (finished => dead);
[] <> dead;
[] (dead => finished);
- <> finished;

[] (cook_ready + cook_running + cook_finished = 1);
[] (cook_ready + cook_started = 1);
[] (eat_ready + eat_running + eat_finished = 1);
[] (eat_ready + eat_started = 1);
```

Les 5 premières lignes c'est pour vérifier si toutes les activités se terminent (on attend que propriété sera fausse et le model checker exhibera un contre-exemple qui correspond à un scénario qui correspond à la terminaison du processus)
Les dernières lignes c'est pour vérifier chaque activité est soit non commencée, soit en cours, soit terminée.
Resultat du verification :

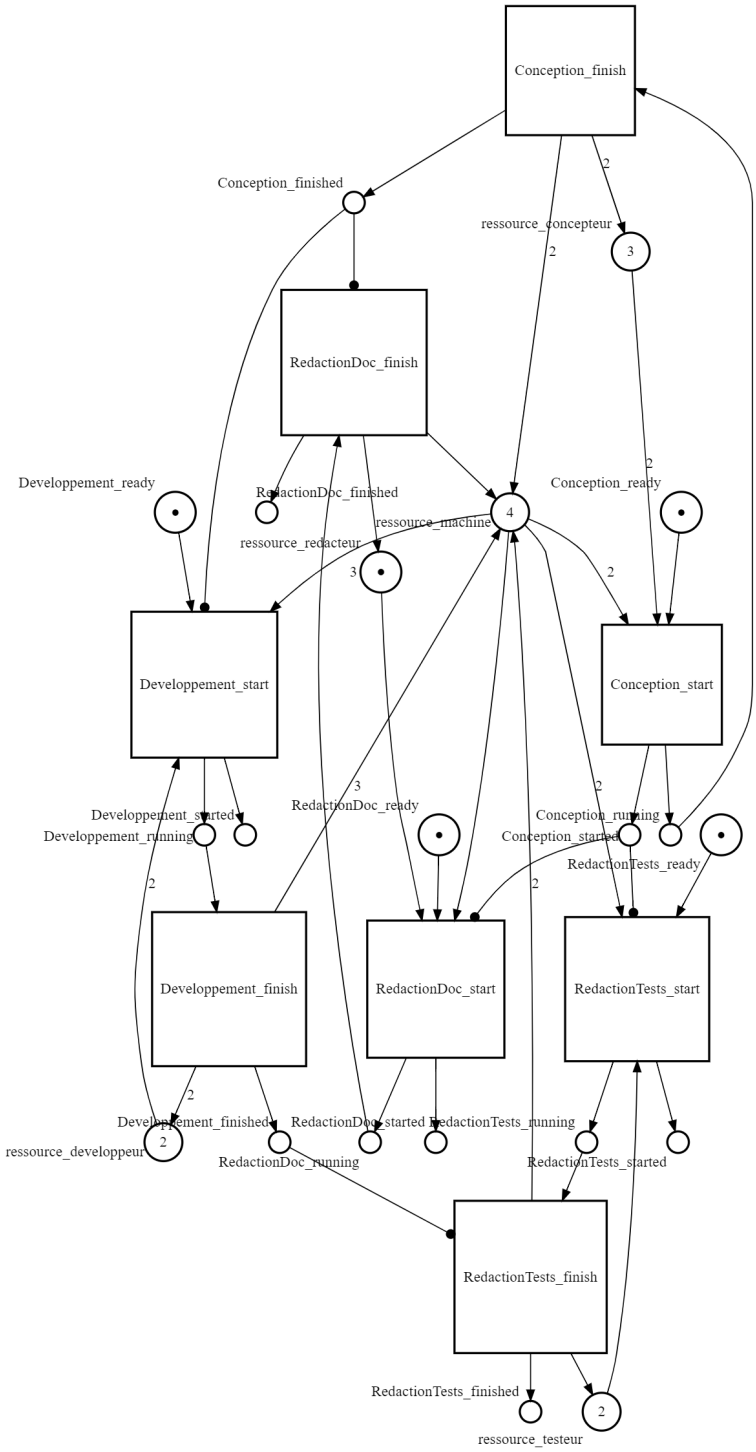
```
- source simple.ltl;
operator finished : prop
TRUE
TRUE
TRUE
FALSE
  state 0: L.scc*4 cook_ready eat_ready plat
  -cook_start->
  state 1: L.scc*3 cook_running cook_started eat_ready
  -cook_finish->
  state 2: L.scc*2 cook_finished cook_started eat_ready plat
  -eat_start->
  state 3: L.scc cook_finished cook_started eat_running eat_started
  -eat_finish->
  state 4: L.dead cook_finished cook_started eat_finished eat_started plat
  -L.deadlock->
  state 5: L.dead cook_finished cook_started eat_finished eat_started plat
  [accepting all]
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
0.047s
```

8.2 Modèle sujet

On teste la transformation sur le modèle inspiré de SPEM donnée dans le sujet :



la transformation nous donne le modèle Petrinet suivant:



On remarque que le process simple termine toujours

8.2.1 Validation de la transformation avec l'outil selt

fichier simple.ltl généré :

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

[] (Conception_ready + Conception_running + Conception_finished = 1);
```

```
[] (Conception_ready + Conception_started = 1);
[] (RedactionDoc_ready + RedactionDoc_running + RedactionDoc_finished = 1);
[] (RedactionDoc_ready + RedactionDoc_started = 1);
[] (Developpement_ready + Developpement_running + Developpement_finished = 1);
[] (Developpement_ready + Developpement_started = 1);
[] (RedactionTests_ready + RedactionTests_running + RedactionTests_finished = 1);
[] (RedactionTests_ready + RedactionTests_started = 1);
```

Resultat du verification :

```
operator finished : prop
TRUE
TRUE
FALSE
  state 0: L.scc*21 Conception_ready Developpement_ready RedactionDoc_ready
  -Conception_start->
  state 1: L.scc*20 Conception_running Conception_started Developpement_rea
  -Conception_finish->
  state 2: L.scc*17 Conception_finished Conception_started Developpement_re
  -RedactionDoc_start->
  state 15: L.scc*15 Conception_finished Conception_started Developpement_r
  -RedactionDoc_finish->
  state 16: L.scc*13 Conception_finished Conception_started Developpement_r
  -RedactionTests_start->
  state 17: L.scc*12 L.dead Conception_finished Conception_started Developp
  -L.deadlock->
  state 18: L.scc*12 L.dead Conception_finished Conception_started Developp
  [accepting all]
FALSE
  state 0: L.scc*21 Conception_ready Developpement_ready RedactionDoc_ready
  -Conception_start->
  state 1: L.scc*20 Conception_running Conception_started Developpement_rea
  -Conception_finish->
  state 2: L.scc*17 Conception_finished Conception_started Developpement_re
  -Developpement_start->
  state 3: L.scc*11 Conception_finished Conception_started Developpement_ru
  -Developpement_finish->
  state 4: L.scc*8 Conception_finished Conception_started Developpement_fin
  -RedactionDoc_start->
  state 5: L.scc*5 Conception_finished Conception_started Developpement_fin
  -RedactionDoc_finish->
  state 6: L.scc*2 Conception_finished Conception_started Developpement_fin
  -RedactionTests_start->
  state 7: L.scc Conception_finished Conception_started Developpement_finis
r testeur
  -RedactionTests_finish->
  state 8: L.dead Conception_finished Conception_started Developpement_fini
eur testeur*2
  -L.deadlock->
  state 9: L.dead Conception_finished Conception_started Developpement_fini
eur testeur*2
  [accepting all]
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
0.000s
```

On remarque que le process sujet peut terminer

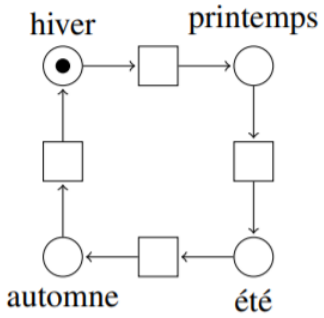
9 Transformation PetriNet vers Tina/dot en utilisant Acceleo

Acceleo est un générateur de code source de la fondation Eclipse permettant de mettre en ouvre l'approche MDA (Model driven architecture) pour réaliser des applications à partir de modèles basés sur EMF

voir **toTINA.mtl** et voir **toDOT.mtl**

9.1 Tests de la transformation M2T

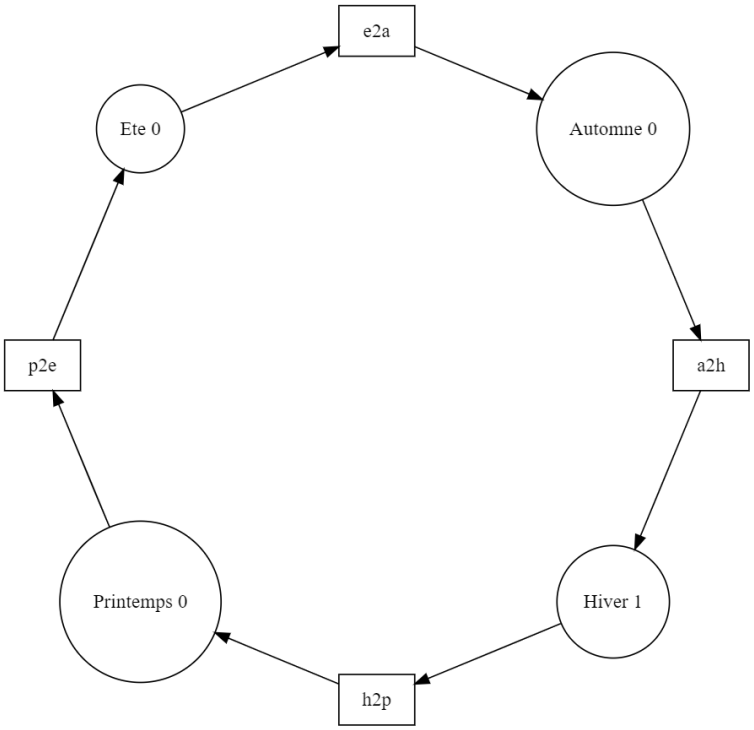
9.1.1 Réseau saisons



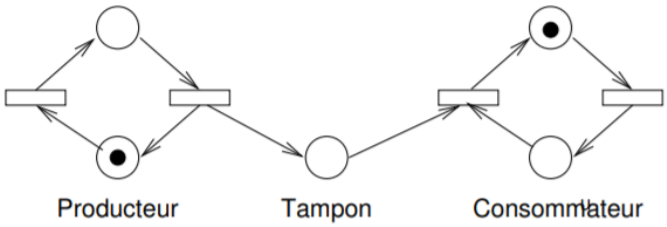
la transformation nous donne le texte tina suivant :

```
net saisons
pl Hiver (1)
pl Printemps (0)
pl Ete (0)
pl Automne (0)

tr h2p  Hiver ->  Printemps
tr p2e  Printemps ->  Ete
tr e2a  Ete ->  Automne
tr a2h  Automne ->  Hiver
```

la transformation nous donne le diagramme dot suivant :



9.1.2 Réseau producteur consomateur



la transformation nous donne le texte tina suivant :

```
net prodcons
pl P1 (1)
pl P2 (0)
pl Tampon (0)
pl C1 (0)
pl C2 (1)

tr P1P2  P1 ->  P2
tr P2P1  P2 ->  P1 Tampon
tr TC2  Tampon C1 ->  C2
tr C2C1  C2 ->  C1
```

la transformation nous donne le diagramme dot suivant :