BE Validation de modèles de procédé

Rapport BE IDM 2012

Alexandre Escudero <escudero@etud.insa-toulouse.fr>
Mathieu Othacéhé <othacehe@etud.insa-toulouse.fr>

5ème année SEC

Résumé du projet

L'objectif de ce mini-projet est de compléter le travail fait en BE IDM et de produire une chaîne de vérification de modèles de processus SimplePDL en utilisant les outils de modelchecking disponibles dans la boîte à outils Tina.

Table des matières

Chapitre 1

Introduction

Dans ce BE, il s'agit d'utiliser les outils de méta-modélisation et de vérification des modèles afin de modéliser des processus déclinés en activités dépendant de ressources et évoluant dans le temps. Différents outils mis à notre disposition nous permettent réaliser ces opérations qui sont décrites dans ce rapport :

- Eclipse
  - Ecore
  - ATL
  - Xtext
  - GMF
  - Epsilon
- Tina
  - Selt

Chapitre 2

Partie 1 : Intégration du temps et des ressources

2.1 T1 : Définition du méta-modèle

Il s'agit dans cette première section de prendre en compte les ressources et le temps dans le méta-modèle SimplePDL. On reprend donc le fichier SimplePDL.ecore définissant le méta-modèle, et l'on y ajoute de nouveaux éléments.

2.1.1 Ajout des ressources

Pour la gestion des ressources :

On rajoute deux nouvelles classes : RessourceDefinition et RessourceInstance. La classe RessourceDefinition sert à définir les différentes ressources et leur nombre total disponible dans le système. Par exemple, si l'on dispose de 4 ressources de type Machine, on va alors déclarer dans le fichier modèle simplepdl une RessourceDefinition, avec un attribut number égal à 4.

La classe RessourceInstance permet d'exprimer le fait qu'une activité va devoir utiliser une certaine quantité de ressources pour fonctionner. Elle possède un lien vers une RessourceDefinition exprimant le type de la ressource ainsi qu'un lien vers l'activité en question. Elle doit aussi posséder un attribut entier indiquant le nombre de ressources nécéssaires à l'activité.

Si l'on affiche le méta-modèle sous forme textuelle (améliorée), on a donc :

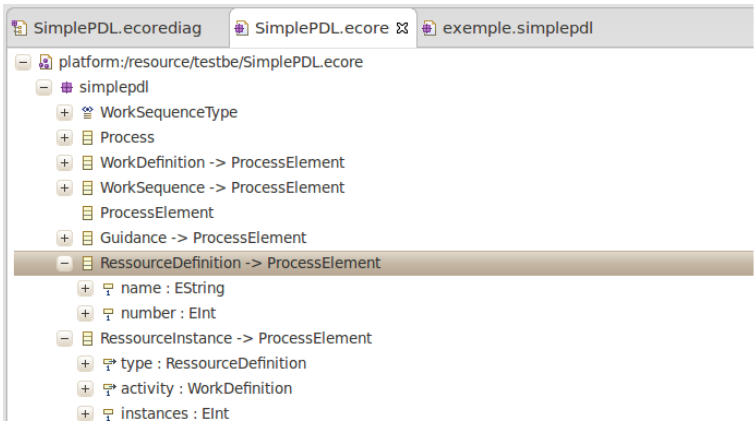

FIGURE 2.1 – Fichier Ecore modifié

On peut également afficher le méta-modèle sous forme graphique Ecorediag :

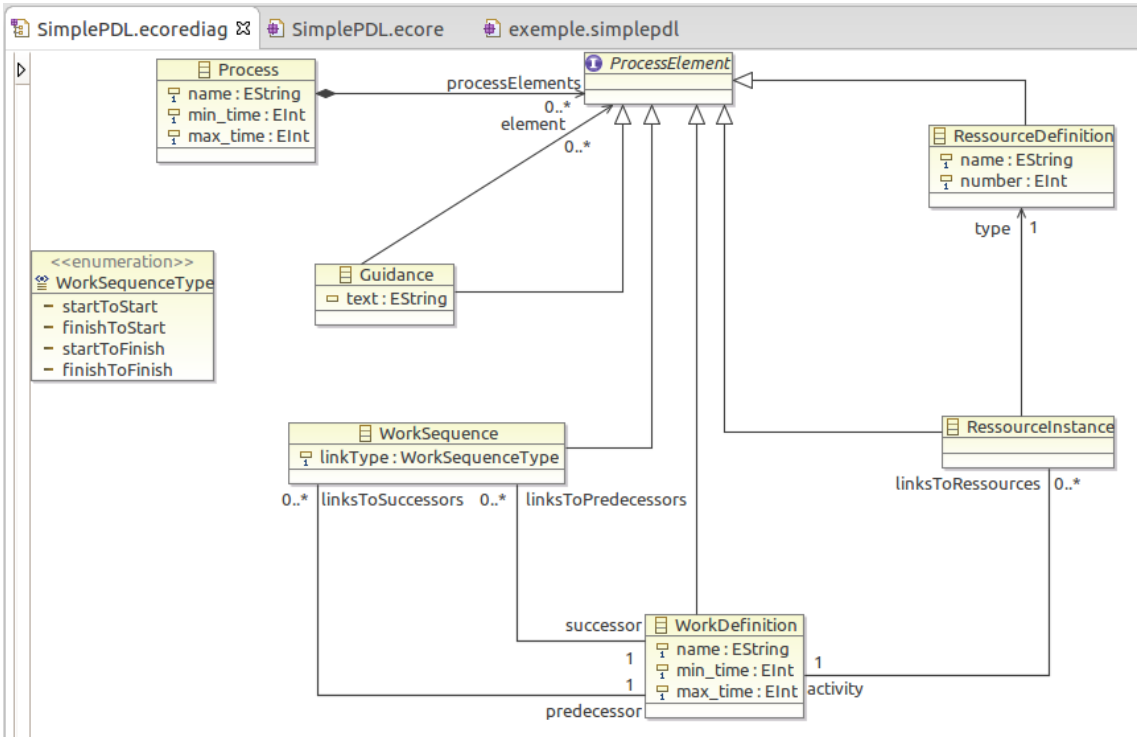

FIGURE 2.2 – Diagramme Ecore modifié

On se retrouve dans une situation assez symétrique avec les classes WorkDefinition et WorkSequence. En effet, ces deux classes héritent toutes deux de ProcessElement et une WorkSequence constitue un lien entre deux WorkDefinition.

Pour la modélisation des ressources disponibles et de leur utilisation par des activités, les classes RessourceDefinition et RessourceInstance héritent également de ProcessElement et une RessourceInstance représentera le lien entre une RessourceDefinition et une WorkDefinition.

2.1.2 Ajout du temps

L'ajout du temps est plus aisé (à ce niveau là en tout cas). Il suffit d'ajouter des attributs temps_min et temps_max aussi bien au niveau du Processus que de la WorkDefinition. Le fichier Ecore modifié est le suivant :

A présent que le fichier Ecore est mis à jour par rapport aux nouvelles spécifications, on va pouvoir modifier l'éditeur graphique GMF pour tenir compte des nouveautés ajoutées.

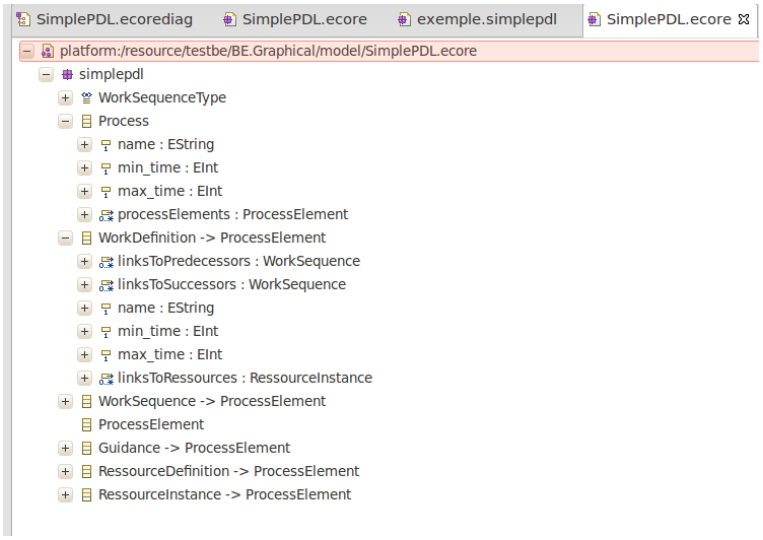

FIGURE 2.3 – Ecore avec gestion du temps

2.2 T2 : Syntaxe concrète graphique

Il s'agit ici d'adapter l'éditeur graphique SimplePDL créé en TP pour permettre de saisir graphiquement les ressources disponibles et l'allocation des ressources nécessaires à chaque activité. On doit également pouvoir spécifier les propriétés temporelles de chaque activité et du processus global.

Pour mettre en oeuvre cet outil graphique on utilise le plugin GMF d'Eclipse qui permet de définir les différentes composantes de l'éditeur à partir du méta-modèle Ecore défini précédemment que l'on considère ici comme une syntaxe abstraite. Après avoir paramétré les composants, l'éditeur graphique permet de construire des processus en définissant des activités, des relations entre elles, des ressources et leurs relations avec les activités. On peut donc reconstruire le modèle suivant, correspondant à l'exemple décrit dans le sujet :

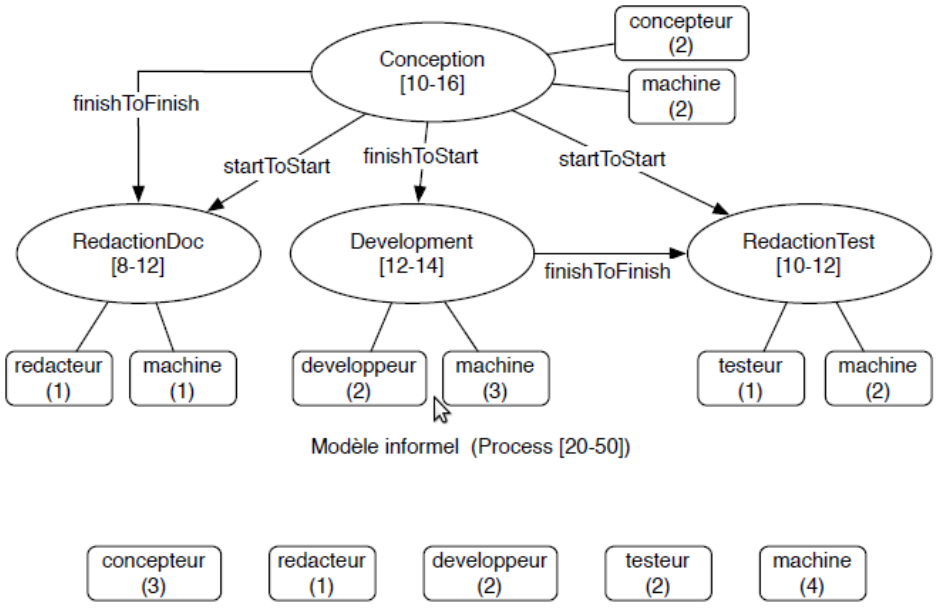

FIGURE 2.4 – Modèle défini dans le sujet

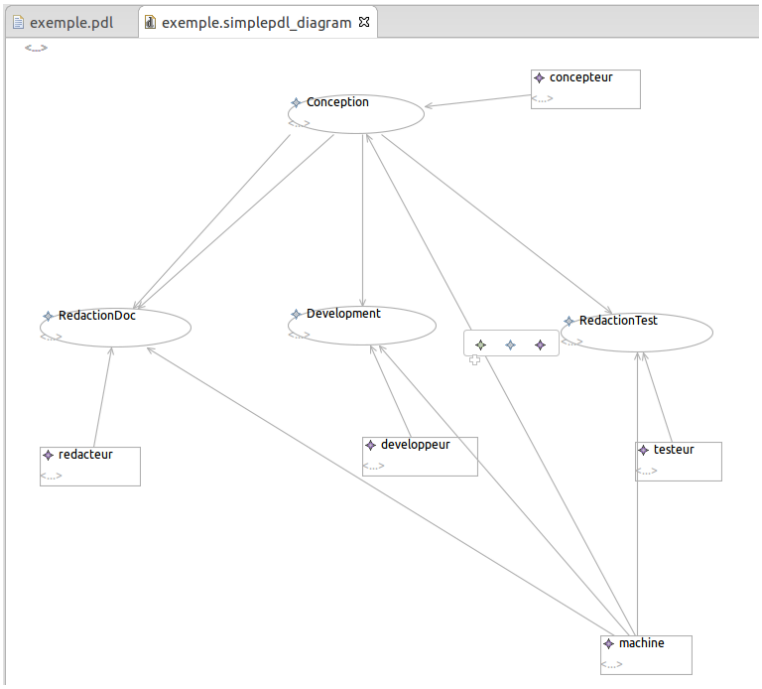

FIGURE 2.5 – Exemple d'utilisation de l'éditeur graphique engendré

Les propriétés telles que :
- le nombre de ressources des éléments de type RessourceDefinition
- le nombre des éléménets de type RessourceInstance
- le type des relations de type WorkSequence
- les intervalles de temps des éléments de type WorkDefinition et Process

sont accessibles depuis l'onglet Propriété.

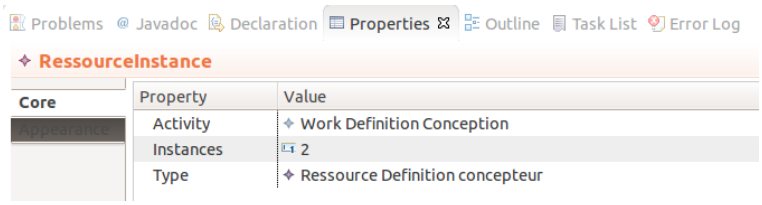

FIGURE 2.6 – Propriétés d'une RessourceInstance

Le modèle généré graphiquement peut être également représenté sous forme arborescente pour être éventuellement exporté ensuite vers un format générique tel que XMI :

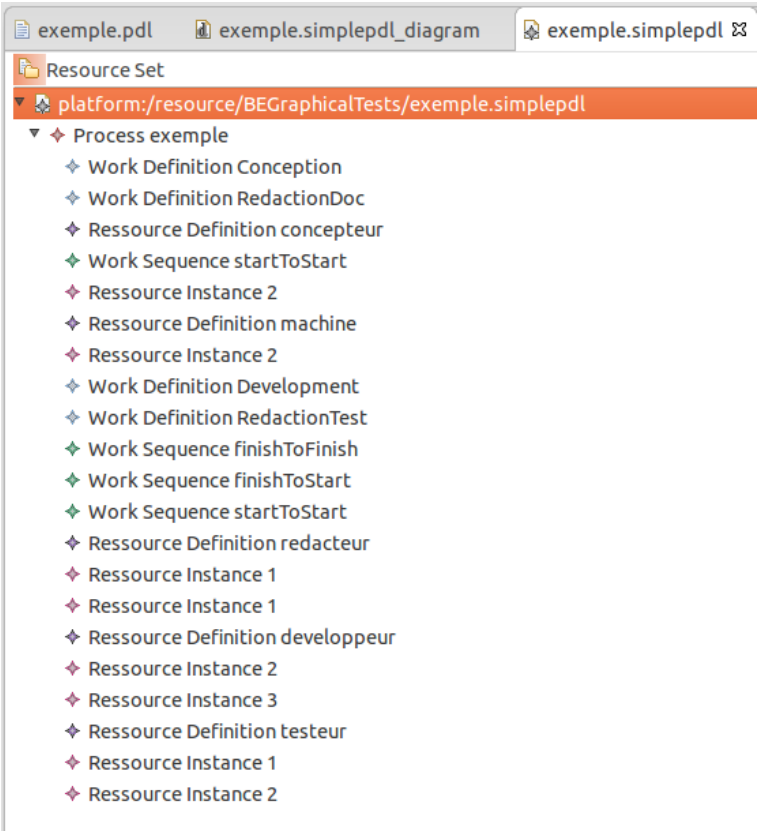

FIGURE 2.7 – Modèle sous forme arborescente généré par l'éditeur graphique

2.3 T3 : Syntaxe concrète textuelle

Nous devons donner une syntaxe concrète textuelle de SimplePDL pour prendre en compte les ressources et le temps et l'outiller avec xText. On va donc reprendre la grammaire xText élaborée lors des séances précédentes.

On rajoute ces lignes dans le fichier xText :

```
RessourceDefinition returns RessourceDefinition:
'rd'
name=EString
'number' number=EInt
;

RessourceInstance returns RessourceInstance:
'ri'
'type' type=[RessourceDefinition|EString]
'activity' activity=[WorkDefinition|EString]
'instances' instances=EInt
;
```

On définit ainsi qu'une RessourceDefinition possède un nom et un chiffre indiquant sa quantité. Une RessourceInstance possède un type, qui n'est autre qu'une WorkDefinition. Elle possède aussi une activité de type WorkDefinition et un nombre d'instances entier.

Il faut aussi modifier la ligne ProcessElement pour indiquer que l'on peut déclarer des éléments de type RessourceDefinition et des RessourceInstance.

```
ProcessElement returns ProcessElement:
WorkDefinition | WorkSequence | Guidance | RessourceDefinition | RessourceInstance
;
```

Enfin, on ajoute au Processus et à la WorkDefinition des attributs min_time et max_time.

```
WorkDefinition returns WorkDefinition:
'wd'
name=EString
'min' min_time=EInt
'max' max_time=EInt
```

et

```
Process returns Process:
{Process}
'process'
name=EString
'{'
'min_time' min_time=EInt
'max_time' max_time=EInt
(processElements+=ProcessElement (processElements+=ProcessElement)* )?
'}'
;
```

On peut à présent créer un fichier s'appuyant sur la grammaire créée et définir un modèle de processus, notamment celui de l'exemple de la partie précédente :

```
process exemple {
min_time 20
max_time 50
wd Conception min 10 max 16
wd RedactionDoc min 8 max 12
wd Development min 12 max 14
wd RedactionTest min 10 max 12
ws s2s from Conception to RedactionDoc
ws f2f from Conception to RedactionDoc
ws f2s from Conception to Development
ws s2s from Conception to RedactionTest
ws f2f from Development to RedactionTest
rd concepteur number 3
rd redacteur number 1
rd developpeur number 2
rd testeur number 2
rd machine number 4
ri type concepteur activity Conception instances 2
ri type machine activity Conception instances 2
ri type redacteur activity RedactionDoc instances 1
ri type machine activity RedactionDoc instances 1
ri type developpeur activity Development instances 2
ri type machine activity Development instances 3
ri type testeur activity RedactionTest instances 1
ri type machine activity RedactionTest instances 2
}
```

A partir du modèle décrit textuellement, on peut l'exporter vers un format générique tel que XMI :

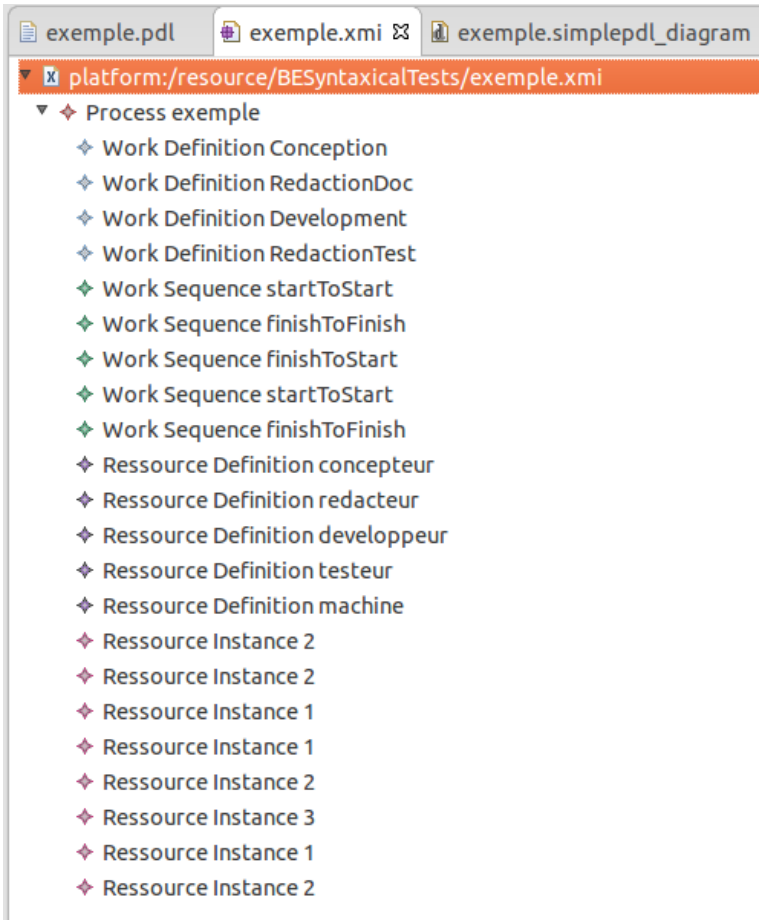

FIGURE 2.8 – Exemple au format XMI sous forme arborescente

2.4 T4 : Contraintes OCL sur le méta-modèle SimplePDL

On complète ici les contraintes OCL pour capturer les contraintes qui n'ont pu l'être par le métamodèle.

2.4.1 Nom des ressources

Comme pour les activités, il s'agit de vérifier que les ressources ont un nom unique. On effectue donc un produit carthésien entre chaque Ressources.

```
-- Invariant vérifiant si toutes les ressources ont un nom unique
inv resourcesNamesAreDifferent:
let elements : Set(ProcessElement) = self.processElements in
     let rds: Set(ProcessElement) = elements->select(e | e.oclIsTypeOf(RessourceDefinition)) in
rds->forAll(rd1, rd2: ProcessElement |
if rd1 <> rd2 then
rd1.oclAsType(RessourceDefinition).name <> rd2.oclAsType(RessourceDefinition).name
else
true
endif
)
```

2.4.2 Verification de la consistance

On peut compléter la requète OCL des TPs, pour s'assurer que toutes les WorkDefinition associées à une RessourceInstance et les activités pointées par une WorkInstance appartiennent bien au même Processus.

```
inv containmentConsistency:
  let elements : Set(ProcessElement) = self.processElements in
    let wds: Set(ProcessElement) = elements->select( e | e.oclIsTypeOf(WorkDefinition)) in
      let wss: Set(ProcessElement) = elements->select( e | e.oclIsTypeOf(WorkSequence)) in
let ris: Set(ProcessElement) = elements->select( e | e.oclIsTypeOf(RessourceInstance)) in
        -- WS linked to a process WD are elements of this process
        wds->forAll(wd: ProcessElement |
            elements->includesAll(wd.oclAsType(WorkDefinition).linksToPredecessors)
            and elements->includesAll(wd.oclAsType(WorkDefinition).linksToSuccessors)
and elements->includesAll(wd.oclAsType(WorkDefinition).linksToRessources))
        -- source and target of a process WS are elements of this process
        and wss->forAll(ws: ProcessElement |
            elements->includes(ws.oclAsType(WorkSequence).predecessor)
            and elements->includes(ws.oclAsType(WorkSequence).successor))
and ris->forAll(ws: ProcessElement |
            elements->includes(ws.oclAsType(RessourceInstance).type)
and elements->includes(ws.oclAsType(RessourceInstance).activity))
```

2.4.3 Consistance des ressources

On doit s'assurer qu'une WorkDefinition ne demande pas plus de ressources que le nombre maximal possible. Par exemple, si on dispose de 4 machines au total et que la WorkDefinition conception en demande 6, le modèle n'est pas correct.

```
context RessourceInstance
inv ressourceCheck:
if self.instances <= self.type.number then
true
else
false
endif
```

2.4.4 Vérification temporelle

On doit s'assurer que la durée minimale d'une processus ou d'une WorkDefinition est bien inférieure à la durée maximale.

```
context WorkDefinition
inv timeCheck1:
self.max_time < self.min_time implies
false
```

2.5 T5 : Transformation SimplePDL vers PetriNet

Il faut à présent engendrer une nouvelle transformation SimplePdl to Petrinet. En effet, on doit reporter les changements réalisés dans méta-modèle SimplePdl, vers le méta-modèle PetriNet.

Nous allons de nouveau distinguer le cas des ressources de celui du temps.

2.5.1 Ressources

Pour les ressources, nous décidons de mettre en place le modèle suivant. Considérons deux types de ressources (RessourceDefinition), machine et testeur. En admettant que une WorkDefintion (Conception dans cette exemple) nécéssite deux machines et un testeur, on aura le modèle suivant en pétri :

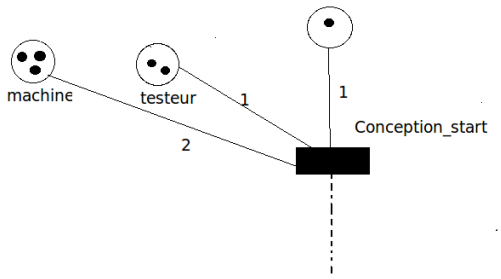

FIGURE 2.9 – Gestion des ressources

Basiquement on aura donc une règle permettant de parcourir les RessourceDefinition, qui seront traduites en places au sens des réseaux de pétri. L'attribut nombre de RessourceDefinition correspond directement au marquage de la place crée.

```
rule RessourceDefinition2PetriNet {
from rd: simplepdl!RessourceDefinition
to
p_ressource: petrinet!Place(
name <- rd.name,
marking <- rd.number,
net <- rd.getProcess())
}
```

Il nous faut ensuite une règle qui pour chaque WorkInstance va créer un arc entre la place de la ressource et la transition de début de la WorkDefinition associée. Il faut aussi crée un arc entre la transition de fin de la WorkDefinition et la place de la ressource. Cet arc traduit le fait que les ressources sont libérées à la fin de l'exécution de l'activité. Il est évident que l'on doit rendre autant de jetons que ceux qui ont été prélevés au démarrage de l'activité.

```
rule RessourceInstance2PetriNet {
from ri: simplepdl!RessourceInstance
to
t_ressource_start: petrinet!Arc(
source <- thisModule.resolveTemp(ri.type, 'p_ressource'),
target <- thisModule.resolveTemp(ri.activity, 't_start'),
weight <- ri.instances,
kind <- #normal,
net <- ri.getProcess()),
t_ressource_finish: petrinet!Arc(
```

```
source <- thisModule.resolveTemp(ri.activity, 't_finish'),
target <- thisModule.resolveTemp(ri.type, 'p_ressource'),
weight <- ri.instances,
kind <- #normal,
net <- ri.getProcess())
}
```

2.5.2 Temps

Pour modéliser la gestion du temps dans un réseau de pétri, nous allons devoir mettre en place un observateur. Le principe de l'observateur est qu'il va nous fournir des informations sur le système sans pour autant en altérer le comportement. Typiquement, l'observateur d'une activité va nous informer sur le fait qu'elle se finit avec une durée correcte (comprise dans l'intervalle [temps_min,temps_max]), trop courte (durée exécution inférieure à la durée minimale) ou trop longue (durée exécution inférieure à la durée maximale).

. . .. . .

TODO : METTRE PHOTO OBSERVATEUR POUR UNE SEULE ACTIVITE ET EXPLIQUER

Le code à fournir pour mettre en place cet observateur est assez long. Il n'est donc pas cité, mais est présent dans les rendus de ce BE.

2.6 T6 : Validation de la transformation SimplePDL vers PetriNet

L'étape de validation consiste à appliquer la transformation précédente et à l'appliquer à un exemple. Prenons le fichier de modèle suivant qui reprend l'exemple du sujet.

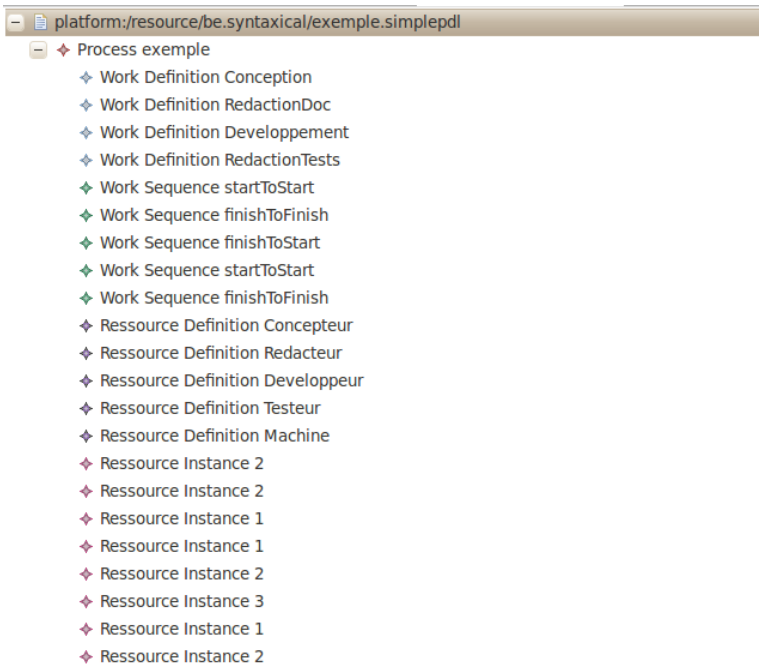

FIGURE 2.10 – Modèle SimplePDL conforme à l'exemple du sujet

Nous obtenons un fichier .PetriNet correspondant bien à ce qui était attendu. Il est valide par rapport au méta-modèle PetriNet. On peut donc valider cette transformation et attaquer la partie PetriNet2Tina.

2.7 T7 : Transformation PetriNet vers Tina

Le méta-modèle PetriNet n'ayant pas évolué, il n'y a pas non plus de raisons de faire évoluer la transformation PetriNet2Tina. Nous avons juste du apporter quelques retouches par rapport à la version des TP. En effet, nous avions considéré que le temps associé par défaut à une transition est par défaut [0,0] sous tina. Pourtant, il s'agit d'un cas particulier puisque nous voulons que par défaut nos transitions soient de type [0,w[.

Dans cette transformation, on doit donc s'assurer que dans le modèle PetriNet si temps_max = -1 (signifie w dans la syntaxe Tina car temps_max forcément entier), alors le fichier de sortie comportera un intervalle terminé par w[.

```
-- build a Tina net file from a Petri net model
query PetriNet2Tina =
let repertoire: String = '/home/alex/workspace/BE/' in
petrinet!PetriNet.allInstances()
->collect(pn | pn.toTina().writeTo(repertoire + pn.name + '.net'));

-- concatenate all the strings of a sequence, adding the before char
-- before each string and the after char after.
helper
def: concatenateStrings(strings: Sequence(String),
before: String, after: String): String =
```

```
strings->iterate(s; acc: String = '' | acc + before + s + after);

-- translate a PetriNet element into the Tina textual syntax
helper
context petrinet!PetriNet
def: toTina(): String =
'net '  + self.name + '\n' + (
let nodesStrings: Sequence(String) = self.nodes->collect(p | p.toTina())
in
thisModule.concatenateStrings(nodesStrings, '', '\n')
)
;

-- translate a place to Tina syntax
helper
context petrinet!Place
def: toTina(): String =
'pl ' + self.name + '(' + self.marking.toString() + ')'
;

-- translate a transition to Tina syntax.
helper
context petrinet!Transition
def: toTina(): String =
let inNodesNames: Sequence(String)
= self.incomings->collect(arc | arc.asTina(true))
in let outNodesNames: Sequence(String)
= self.outgoings->collect(arc | arc.asTina(false))
in
('tr '  + self.name
+ ' [' + self.min_time + ',' + self.max_time + ']'
+ thisModule.concatenateStrings(inNodesNames, ' ', '')
+ ' -> '
+ thisModule.concatenateStrings(outNodesNames, ' ', '')).regexReplaceAll('-1]', 'w[')
;

-- Translate an Arc to Tina syntax. isSource indicates if it is an
-- outgoing arc or an incoming one
helper
context petrinet!Arc
def: asTina(isSource: Boolean): String =
if isSource = true then
self.source.name
else
self.target.name
endif
+
if self.kind = #normal then
'*'
else
'?'
endif
+ self.weight.toString()
;
```

2.8 T8 : Contraintes OCL sur le méta-modèle PetriNet

? ?

2.9 T9 : Propriétés LTL

Pour vérifier la terminaison d'un processus, il s'agit d'écrire plusieurs requètes LTL. Soit l'opérateur finished qui correspond au ET logique entre toutes les places de fin de chaque activité, on a les propriétés suivantes :

[] (finished => dead) ; Si l'on a fini, alors le réseau est mort, doit être Vrai.
[] <> dead ; Il est possible que le réseau soit mort, doit être Vrai.
[] (dead => finished) ;
- <> finished ;

2.10 T10

TO BE DONE

Chapitre 3

Partie 2 : Extensions supplémentaires

Nous avons implémenté deux extensions supplémentaires qui apportent des fonctionnalités supplémentaires mais aussi de la compléxité au système. C'est pourquoi nous avons les avons implémenté en parallèle et façon à ce qu'elles soient indépendantes entre elles.

3.1 Gestion plus fine des ressources

3.1.1 Transformation SimplePDL vers PetrinNet

3.2 Ressources alternatives

3.2.1 Redéfinition du méta-modèle

3.2.2 Transformation SimplePDL vers PetrinNet