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 définissant le méta-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 (*WorkDefinition*) 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 :

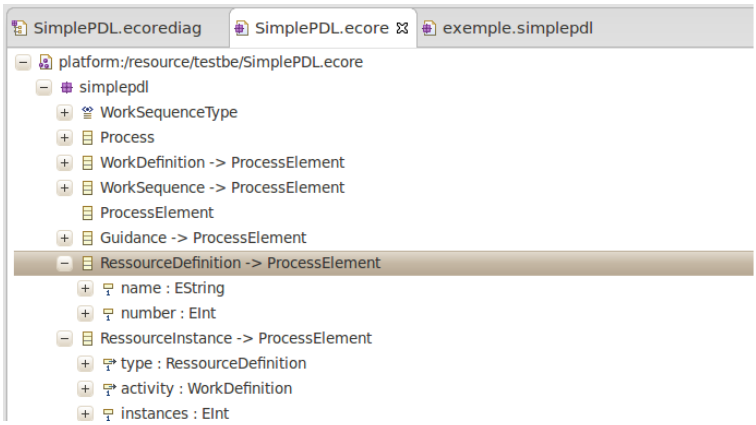

FIGURE 2.1 – Fichier Ecore modifié

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

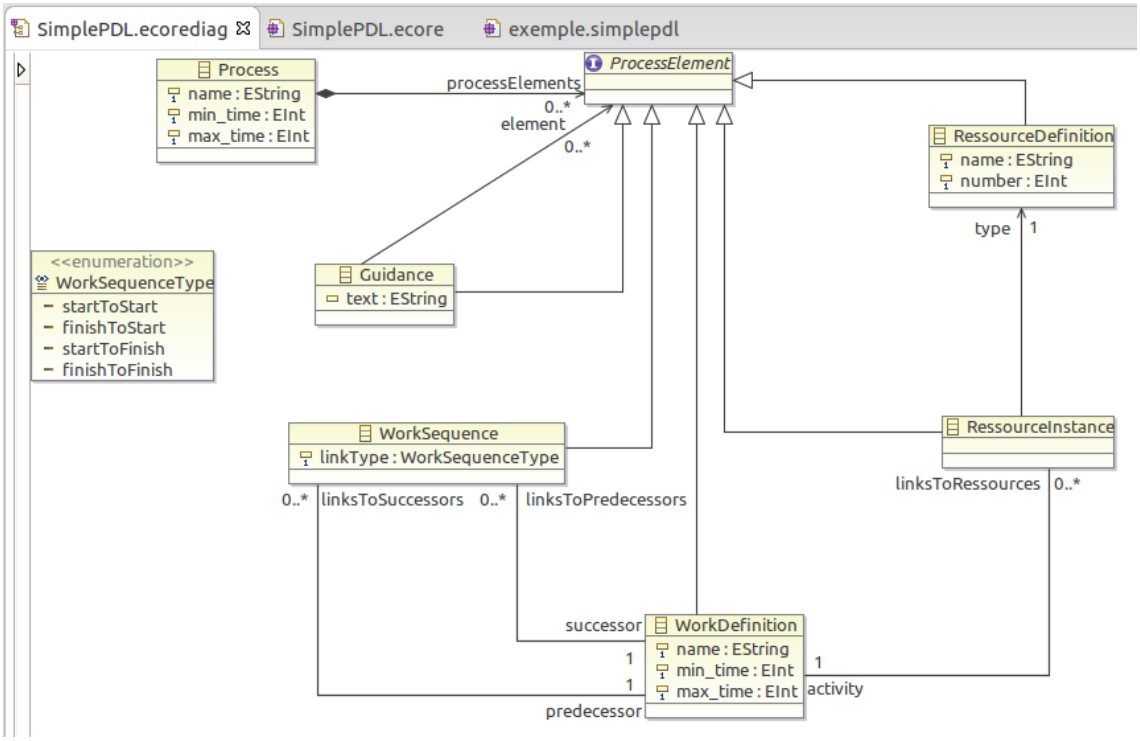

FIGURE 2.2 – Diagramme Ecore modifié

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 *Process* 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.

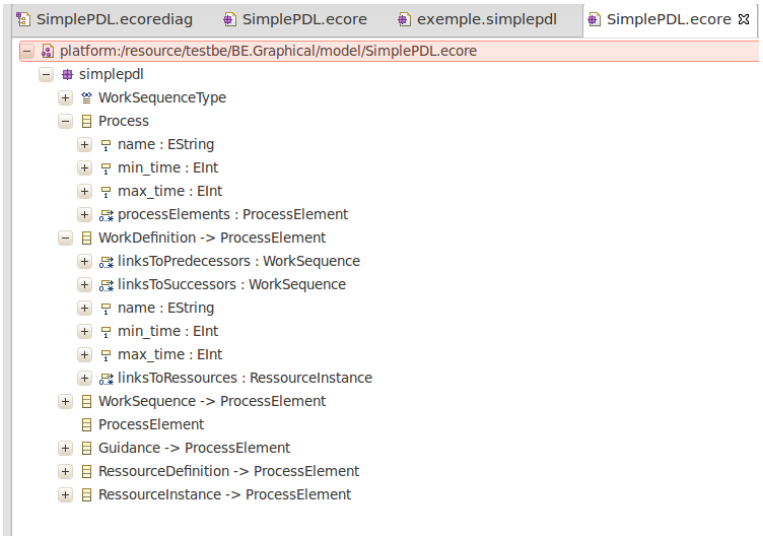

FIGURE 2.3 – Ecore avec gestion du temps

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 :

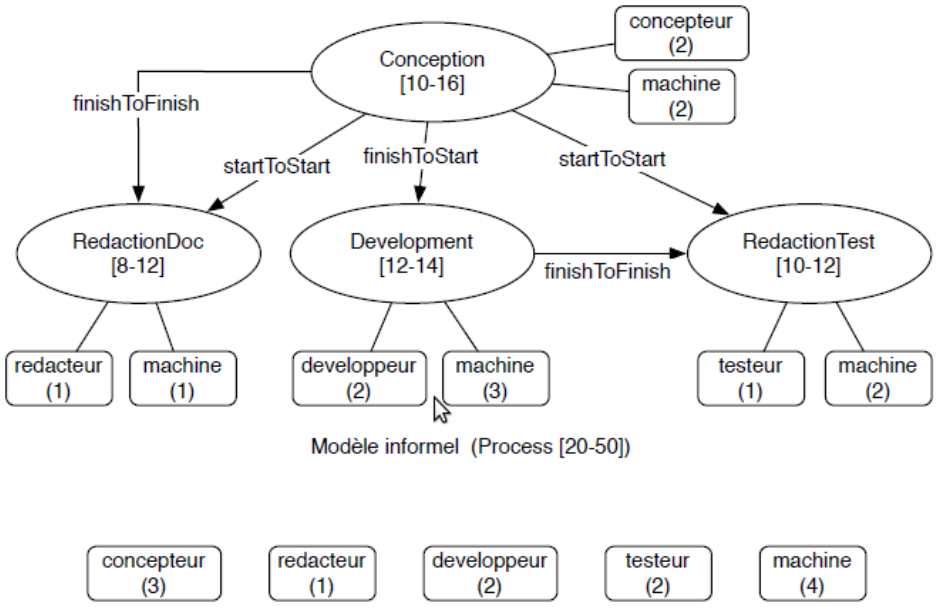

FIGURE 2.4 – Modèle défini dans le sujet

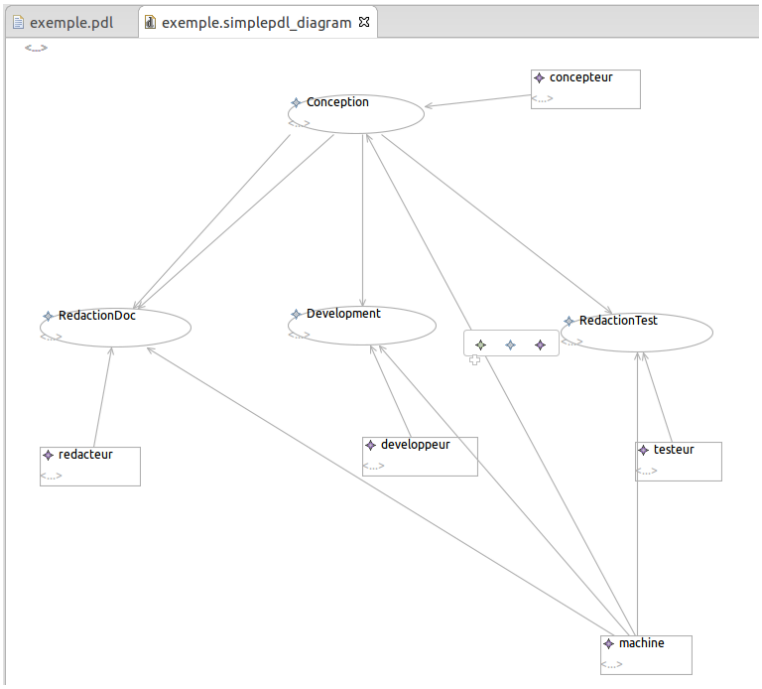

FIGURE 2.5 – Exemple d'utilisation de l'éditeur graphique engendré

On peut observer dans ce modèle édité graphiquement, que l'on représente :
- un élément de type *WorkDefinition* par une ellipse
- un élément de type *WorkSequence* par une flèche entre deux éléments de type *WorkDefinition*
- un élément de type *RessourceDefinition* par un rectangle
- un élément de type *RessourceInstance* par une flèche entre un élément de type *RessourceDfinition* et un de type *WorkDefinition*

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 Propriétés.

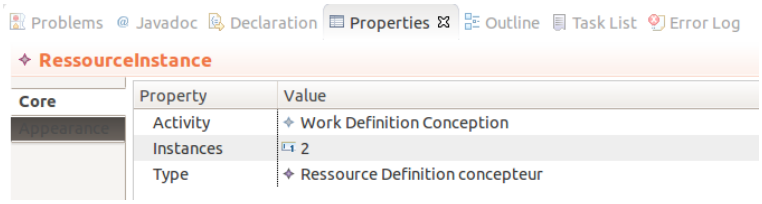

FIGURE 2.6 – Propriétés d'une RessourceInstance

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

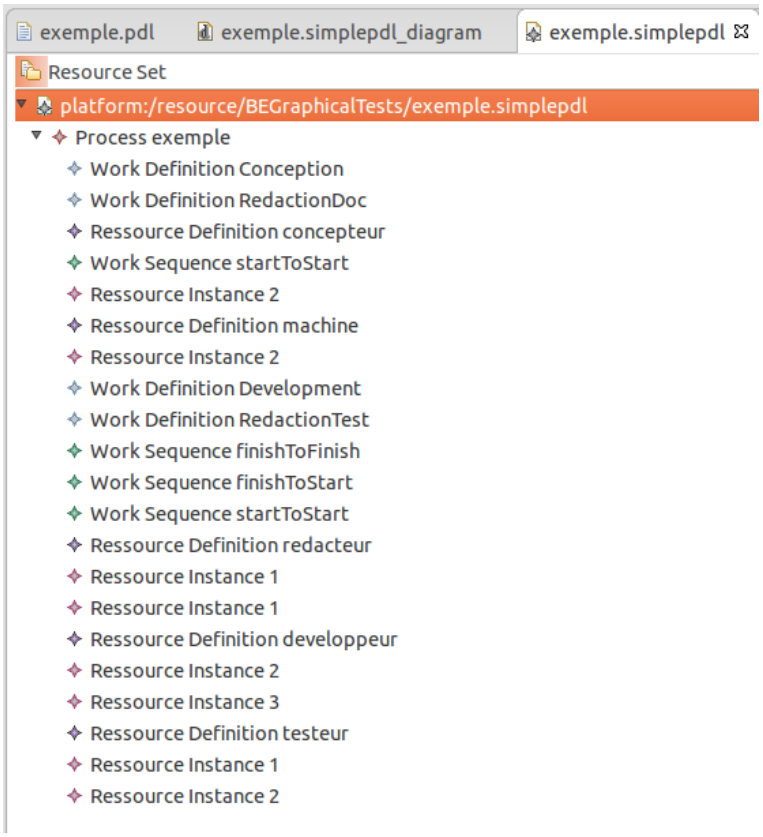

FIGURE 2.7 – Modèle SimplePDL sous forme arborescente généré par l'éditeur graphique

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 pour prendre en compte dans la syntaxe les éléments de type *RessourceDefinition* et *RessourceInstance* :

```
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. Un élément de type *RessourceInstance* possède un type, qui n'est autre qu'un élément de type *RessourceDefinition*. Il possède aussi une activité qui correspond à un élément 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* à l'intérieur de l'élément racine de type *Process*.

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

Enfin, on ajoute au *Process* 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 :

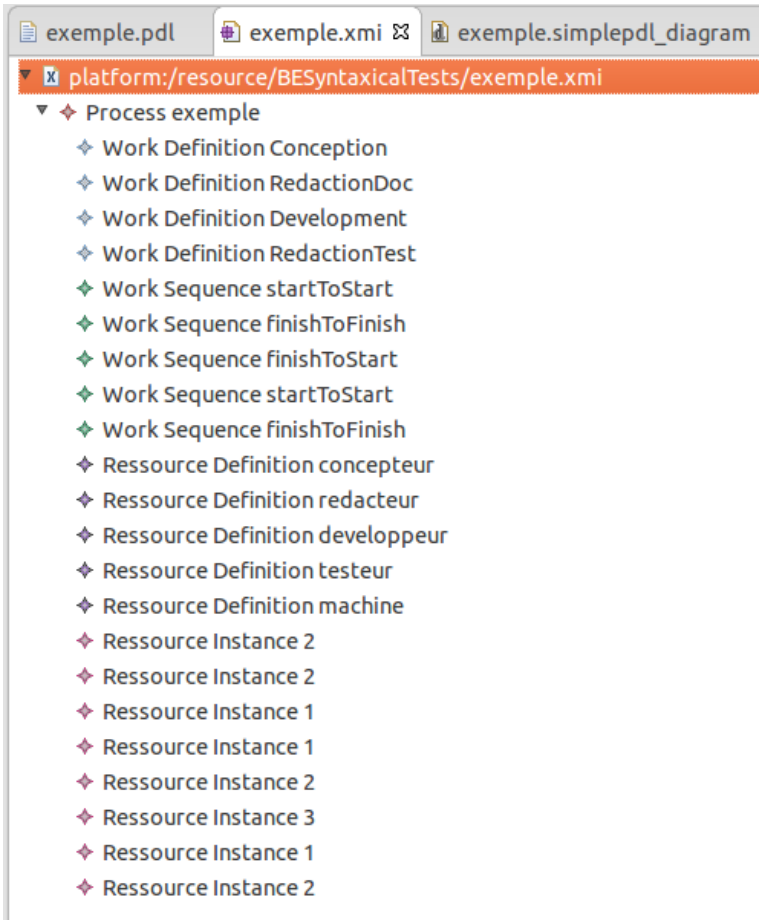

FIGURE 2.8 – Exemple de modèle SimplePDL au format XMI sous forme arborescente

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 méta-modèle.

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 élément de type *RessourceDefinition*.

```
-- 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 tous les éléments de type *WorkDefinition* associés à une *RessourceInstance* et les activités pointées par une *WorkInstance* appartiennent bien au même *Process*.

```
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))
    -- activity and type of a process RI are elements of this process
    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'un élément de type *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 *Process* 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 vers 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 PetriNet :

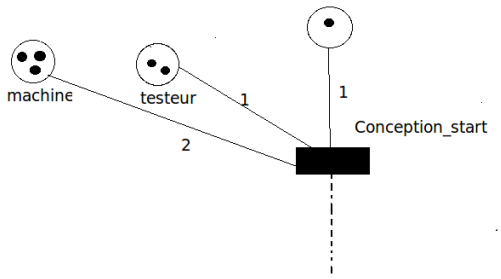

FIGURE 2.9 – Gestion des ressources

Basiquement on aura donc une règle permettant de parcourir les éléments de type *RessourceDefinition*, qui seront traduites en éléments de type *Place* au sens des réseaux de Petri. L'attribut *nombre* de *RessourceDefinition* correspond directement au marquage initial de la place créée.

```
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 élément de type *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 qu'à la libération, 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 Petri, 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).

Prenons l'activité *Conception* à laquelle on adjoint un observateur :

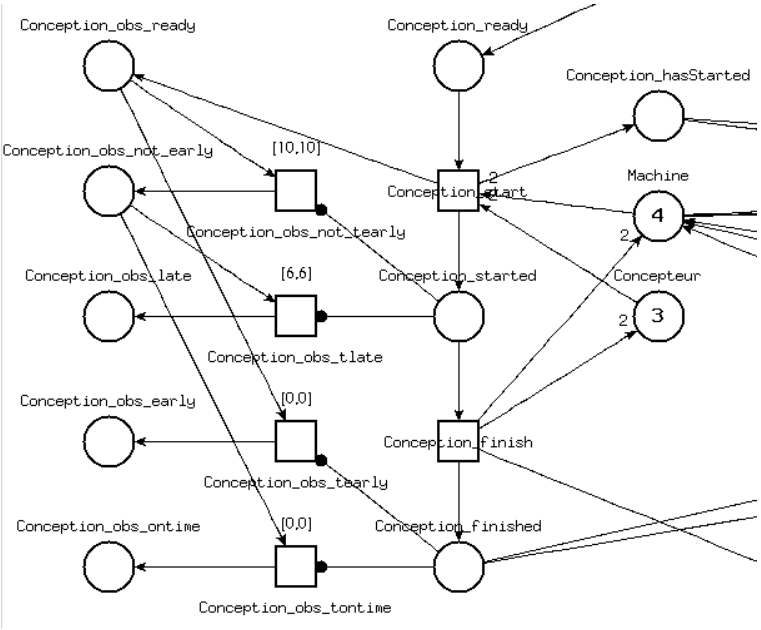

FIGURE 2.10 – Observateur sur l'activité Conception

Les observateurs sont donc présents au niveau de chaque activité puisqu'on veut surveiller pour chacune leur temps d'exécution, mais aussi pour le processus tout entier qui possède également des attributs *time_min* et *time_max*. Pour observer le processus tout entier, le principe est le même que pour une activité, car à l'instar d'une activité, il possède les places *Ready*, *Started* et *Terminated* (équivalent de *Finished* pour les activités),

Dans notre exemple, on obtient le réseau de Petri suivant modélisant la gestion des ressources et l'observation des temps d'exécution :

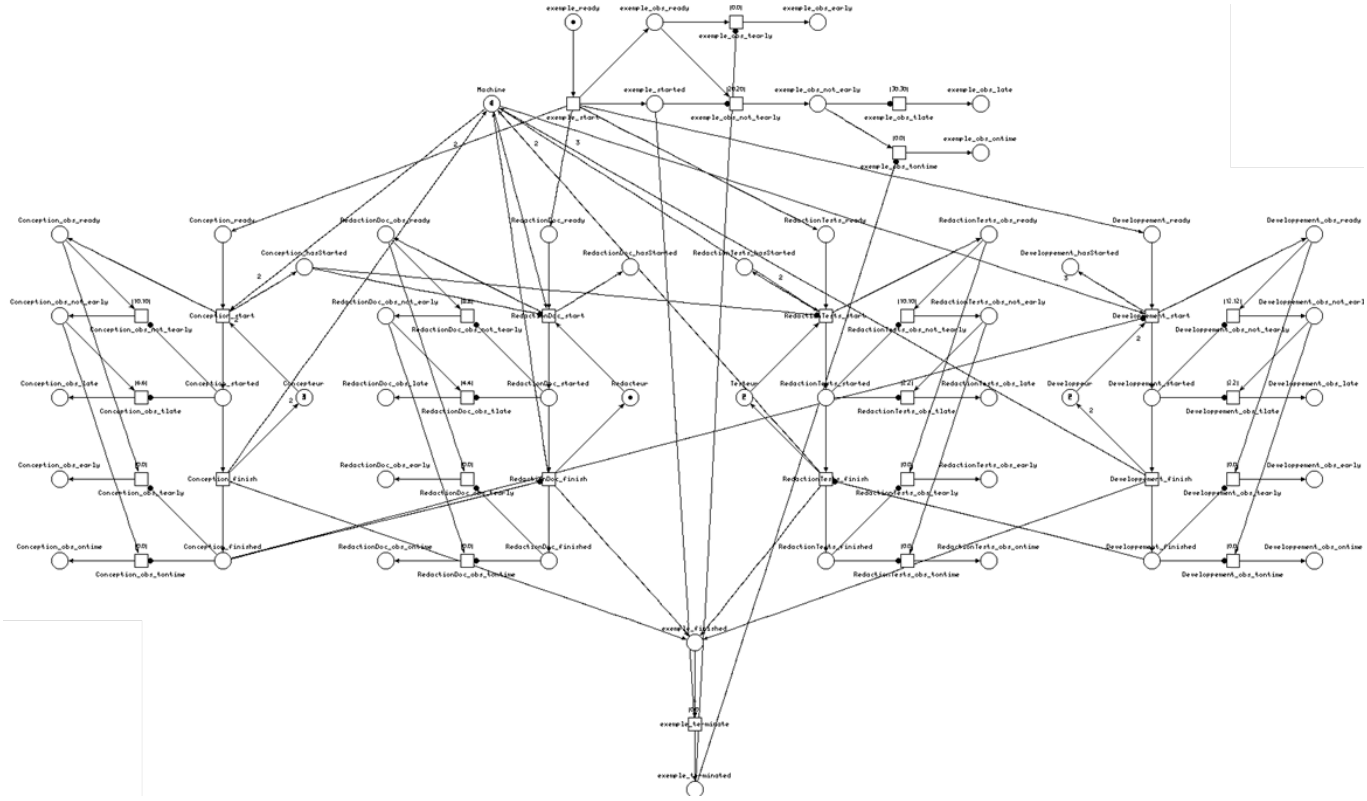

FIGURE 2.11 – Réseau de Petri de l'exemple de processus sous l'outil Tina

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.

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 PetriNet vers Tina. 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[*.

```
-- translate a transition to Tina syntax.
helper
context petrinet!Transition
def: toTina(): String =
```

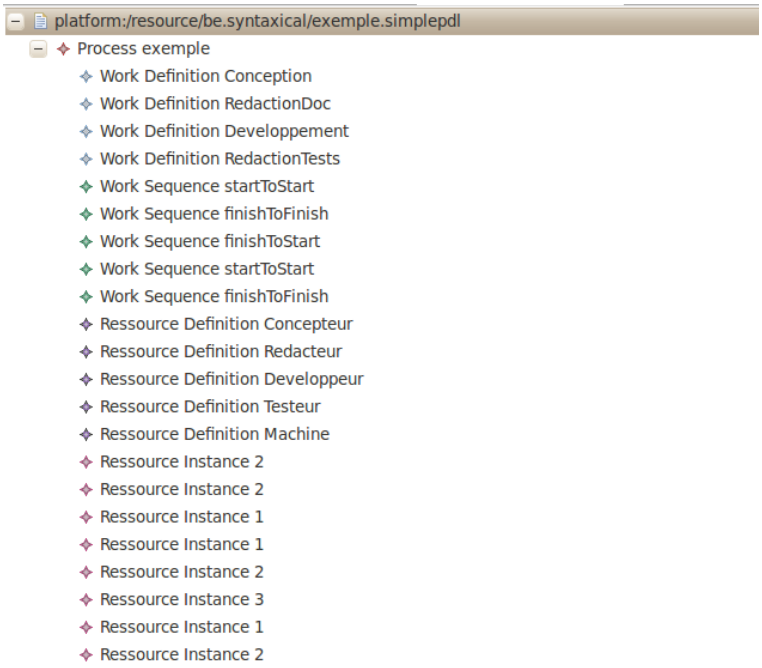

FIGURE 2.12 – Modèle SimplePDL conforme à l'exemple du sujet

```
  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

Il s'agit ici de vérifier certaines propriétés fondammentales que l'on ne peut exprimer dans le méta-modèle PetriNet, c'est pourquoi l'on s'appuie sur des règles OCL qui permettent de rajouter certaines contraintes et de vérifier si les modèles instances du méta-modèle PetriNet les respectent.

2.8.1 Noms du réseau, des places et transitions

Comme pour le méta-modèle SimplePDL, on vérifie que les noms sont du réseau de Petri, des éléments de type *Place* et de type *Transitions* ne sont pas vides. On vérifie également que les noms des éléments de type *Place* sont uniques, de même pour les éléments de type *Transition*.

2.8.2 Liaison des arcs

Il faut vérifier qu'un élément de type *Arc* lie un élément de type *Place* avec un élément de type *Transition*, dans un sens ou dans l'autre. Il faut donc vérifier pour chaque élément de type *Arc* que l'on vérifie l'une des deux conditions suivantes :

– l'attribut *source* est un élément de type *Place* et l'attribut *target* est un élément de type *Transition*
– l'attribut *source* est un élément de type *Transition* et l'attribut *target* est un élément de type *Place*

En langage OCL, cette propriété se vérifie avec la règle suivante :

```
context Arc
inv arcValid:
-- la source d'un read_arc est toujours de type place et la destination de type transition
        (self.kind = ArcKind::read_arc implies
                (self.source.oclIsTypeOf(Place) and self.target.oclIsTypeOf(Transition)))
        and
-- la source et la destination d'un arc normal doivent etre de nature opposées
        (self.kind = ArcKind::normal implies
                (self.source.oclIsTypeOf(Place) and self.target.oclIsTypeOf(Transition)
                or
                self.source.oclIsTypeOf(Transition) and self.target.oclIsTypeOf(Place)))
```

2.8.3 Composantes temporelles des transitions

Il faut également vérifier que si elles sont toutes deux entiers positifs, la borne temporelle inférieure d'une transition doit être inférieure ou égale à la borne supérieure. On peut traduire la propriété en OCL de la manière suivante :

2.9 T9 : Propriétés LTL

Maintenant que nous avons effectué des vérification sur les modèles SimplePDL et PetriNet, il faut maintenant effectuer une vérification des propriétés de sûreté sur le modèle de réseau de Petri servant d'entrée à l'application Tina. Ces vérification peuvent s'effectuer via des requêtes LTL exécutées sur l'outil Selt.

2.9.1 Relation entre terminaison des activités et mort du réseau

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

```
# Si l'on a fini, alors le réseau est mort, doit renvoyer True.
[] (finished => dead);
```

```
# Il est possible que le réseau soit mort, doit renvoyer True.
[] <> dead;

# Si le réseau est mort, alors toutes les activités sont terminées, doit renvoyer True.
[] (dead => finished);

# On atteindra jamais l'état où toutes les activités sont terminées, doit renvoyer False.
- <> finished;
```

2.9.2 Invariance des activités

On souhaite également pouvoir vérifier qu'une fois le processus démarré (*Started*), ses activités seront toujours soit dans l'état :

- non commencée - *Ready*
- en cours - *Started*
- terminée - *Finished*

Dans notre exemple, cette propriété se traduit par les requêtes LTL suivantes :

```
[] (exemple_started => (Conception_ready \/ Conception_started \/ Conception_finished));
[] (exemple_started => (RedactionDoc_ready \/ RedactionDoc_started \/ RedactionDoc_finished));
[] (exemple_started => (Developpement_ready \/ Developpement_started \/ Developpement_finished));
[] (exemple_started => (RedactionTests_ready \/ RedactionTests_started \/ RedactionTests_finished));
```

2.9.3 Irréversibilité des activités terminées

Enfin, on peut vérifier qu'une activité terminée n'évolue plus du tout : c'est-à-dire qu'une activité ayant atteint l'état *Finished* n'atteindra plus jamais les états *Ready* ou *Started*. Dans notre exemple, cela se traduit par les requêtes suivantes :

```
[] (Conception_finished => - <> (Conception_ready \/ Conception_started));
[] (RedactionDoc_finished => - <> (RedactionDoc_ready \/ RedactionDoc_started));
[] (Developpement_finished => - <> (Developpement_ready \/ Developpement_started));
[] (RedactionTests_finished => - <> (RedactionTests_ready \/ RedactionTests_started));
```

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

Il s'agit dans cette extension de permettre qu'une activité en cours d'exécution puisse être interrompue puis reprendre le cours de son exécution. Une telle interruption doit libérer les ressources utilisées par l'activité en question qui deviennent alors disponibles pour une autre activité. Pour une activité interrompue, sa reprise doit être permise à condition que les ressources dont il dépend soient disponibles.
Pour cette extension, il n'est pas nécessaire de modifier la structure du méta-modèle SimplePDL, ce qui implique qu'il n'y a pas de modification à apporter à l'outil de génération de syntaxe concrète textuelle, ni à l'outil de syntaxe concrète graphique. En effet, on agira principalement sur le réseau de Petri résultant de la transformation à partir d'un modèle SimplePDL. Nous allons donc adapter la requête de transformation SimplePDL vers PetriNet afin de permettre l'interruption et la reprise (soumise à condition) d'une activité.

3.1.1 Transformation SimplePDL vers PetriNet

Il est nécessaire de modifier le réseau de Petri mais tout en veillant à toujours respecter la spécification stipulant qu'une activité est soit :
- non commencée - *Ready*
- en cours - *Started*
- terminée - *Finished*

Il ne faut donc pas supprimer les états déjà mis en place dans la partie précédente.

D'autre part, il est préférable de ne pas altérer le fonctionnement général du réseau de Petri engendré par la requête conçue dans la première partie afin de ne pas risquer une regression au niveau des fonctionnalités, particulièrement au niveau des observateurs associés à chaque activité. C'est pourquoi nous avons fait le choix de rajouter des places représentant deux sous états de l'état *Started* :
- en cours d'exécution - *Running*
- interrompue - *Interrupted*

Deux nouvelles transitions sont nécessaire pour permettre à une activité d'osciller entre ces deux sous états :
- interrompre - *Interrupt* : *Running* vers *Interrupted*
- reprendre - *Resume* : *Interrupted* vers *Running*

On remarque que ces nouveaux sous états ne changent rien au fonctionnement des observateurs mis en place dans la première partie, cela permet de répondre à l'exigence qui précise que quelque soit le sous état dans lequel on se trouve (*Running* ou *Interrupted*), le temps continue à être décompté.

Pour cela, il faut donc générer de nouvelles places représentant ces sous-états dans la requête de transformation SimplePDL vers PetriNet, pour tous les éléments *wd* de type *WorkDefinition* parcourus dans le modèle SimplePDL en entrée :

```
p_running: petrinet!Place(
        name <- wd.name + '_running',
        marking <- 0,
        net <- wd.getProcess()),
p_interrupted: petrinet!Place(
        name <- wd.name + '_interrupted',
        marking <- 0,
        net <- wd.getProcess()),
```

Il faut également insérer les transitions, toujours depuis le contexte des éléments de type *WorkDefinition* :

```
t_interrupt: petrinet!Transition(
        name <- wd.name + '_interrupt',
        incomings <- a_r2i,
        outgoings <- a_i2i,
        min_time <- 0,
        max_time <- -1,
        net <- wd.getProcess()),
t_resume: petrinet!Transition(
        name <- wd.name + '_resume',
        incomings <- a_i2r,
        outgoings <- a_r2r,
        min_time <- 0,
        max_time <- -1,
        net <- wd.getProcess()),
```

De même pour les arcs :

```
a_r2i: petrinet!Arc(
        source <- p_running,
        target <- t_interrupt,
        weight <- 1,
        kind <- #normal,
        net <- wd.getProcess()),
a_i2i: petrinet!Arc(
        source <- t_interrupt,
        target <- p_interrupted,
        weight <- 1,
        kind <- #normal,
        net <- wd.getProcess()),
a_i2r: petrinet!Arc(
        source <- p_interrupted,
        target <- t_resume,
        weight <- 1,
        kind <- #normal,
        net <- wd.getProcess()),
a_r2r: petrinet!Arc(
        source <- t_resume,
        target <- p_running,
        weight <- 1,
        kind <- #normal,
        net <- wd.getProcess()),
```

Les arcs suivants sont nécessaires pour la libération lors de l'interruption d'une activité et pour la prise de ressources lors de la reprise du cours de son exécution d'une activité. Dans ce cas-là, on se positionne depuis le contexte des éléments *ri* de type *RessourceInstance* qui font le lien entre une activité (*WorkDefinition*) et un type de ressource (*RessourceDefinition*) et qui contiennent la donnée du nombre de ressources requises (qui seront libérées et reprises successivement) :

```
a_i2r: petrinet!Arc(
        source <- thisModule.resolveTemp(ri.activity, 't_interrupt'),
        target <- thisModule.resolveTemp(ri.type, 'p_ressource'),
        weight <- ri.instances,
        kind <- #normal,
        net <- ri.getProcess()),
a_r2r: petrinet!Arc(
        source <- thisModule.resolveTemp(ri.type, 'p_ressource'),
        target <- thisModule.resolveTemp(ri.activity, 't_resume'),
        weight <- ri.instances,
        kind <- #normal,
net <- ri.getProcess())
```

Afin de ne pas perturber le système de gestion des ressources initial, il est nécessaire de préciser quelques spécifications supplémentaires :

- Une activité qui entre dans l'état *Started*, entre par défaut dans le sous-état *Running*
- Pour pouvoir traverser la transition *Finish* amenant à l'état *Finished*, il faut être dans le sous-état *Running* et évidemment dans l'état *Started*

Cela implique le rajout les arcs suivants :

```
a_s2r: petrinet!Arc(
        source <- t_start,
        target <- p_running,
        weight <- 1,
        kind <- #normal,
        net <- wd.getProcess()),
a_r2f: petrinet!Arc(
        source <- p_running,
        target <- t_finish,
        weight <- 1,
        kind <- #normal,
        net <- wd.getProcess()),
```

3.1.2 Exemple d'exécution

Dans cet exemple d'exécution, on vérifie que :

- si on est dans le sous-état *Running*, les ressources sont prises et que l'on peut accéder aux transitions *Finish* et *Interrupt*
- si on est dans le sous-état *Interrupted*, les ressources sont libérées et que l'on peut accéder à la transition *Resume* et pas à la transition *Finish*

On peut vérifier ces propriétés, avec une exécution en cinq phases sur une même activité, correspondant aux cinq états suivants, à l'aide du steppeur de l'outil Tina :

- *Ready*
- *Started Running*
- *Started Interrupted*
- *Started Running*
- *Finihed*

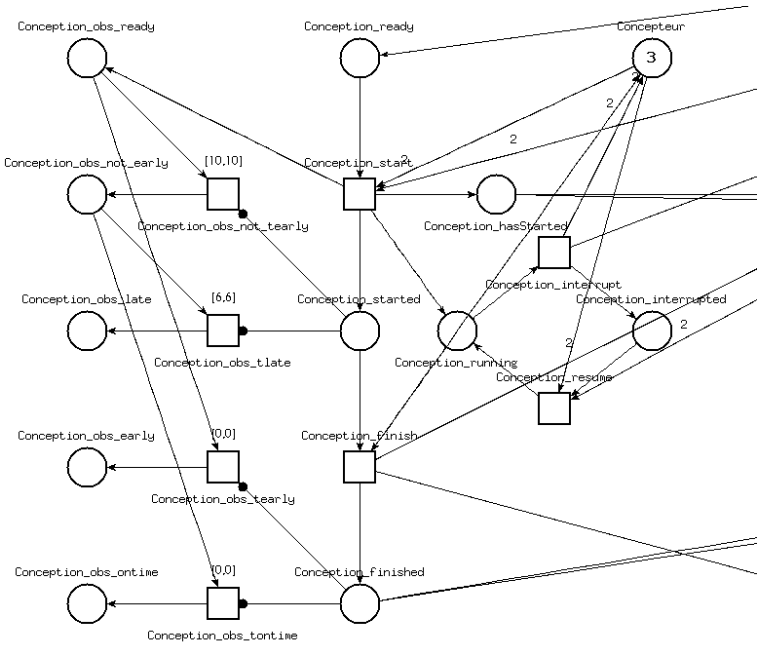

FIGURE 3.1 – Gestion fine des ressources sur l'activité Conception

3.1.3 Contraintes LTL

Afin de montrer que cette extension n'altère en rien les contraintes exiggées dans le document de spécification, on peut exécuter la requête LTL permetant de vérifier que si le processus est en *Started*, alors toutes ses activités sont soit en *Ready*, soit en *Started*, soit en *Finished*. Si l'on exécute les requêtes associées à cette propriété, présentées dans la partie 2.9.2., l'outil Selt renvoie True.

3.2 Ressources alternatives

Il s'agit ici de mettre en oeuvre une gestion alternative des ressources. Le principe est qu'il existe plusieurs configurations possibles de ressources. Ainsi, soit une configuration A nécéssitant 2 machines et 3 développeurs et une configuration B avec 3 machines et 2 testeurs, on pourra exprimer le fait qu'une WorkDefition a besoin que la configuration A ou que la configuration B soit effective.

3.2.1 Redéfinition du méta-modèle

Il est donc clair que l'on doit modifier le méta-modèle pour expliciter cette possibilité de "regroupement des ressources".

Et sous forme graphique,

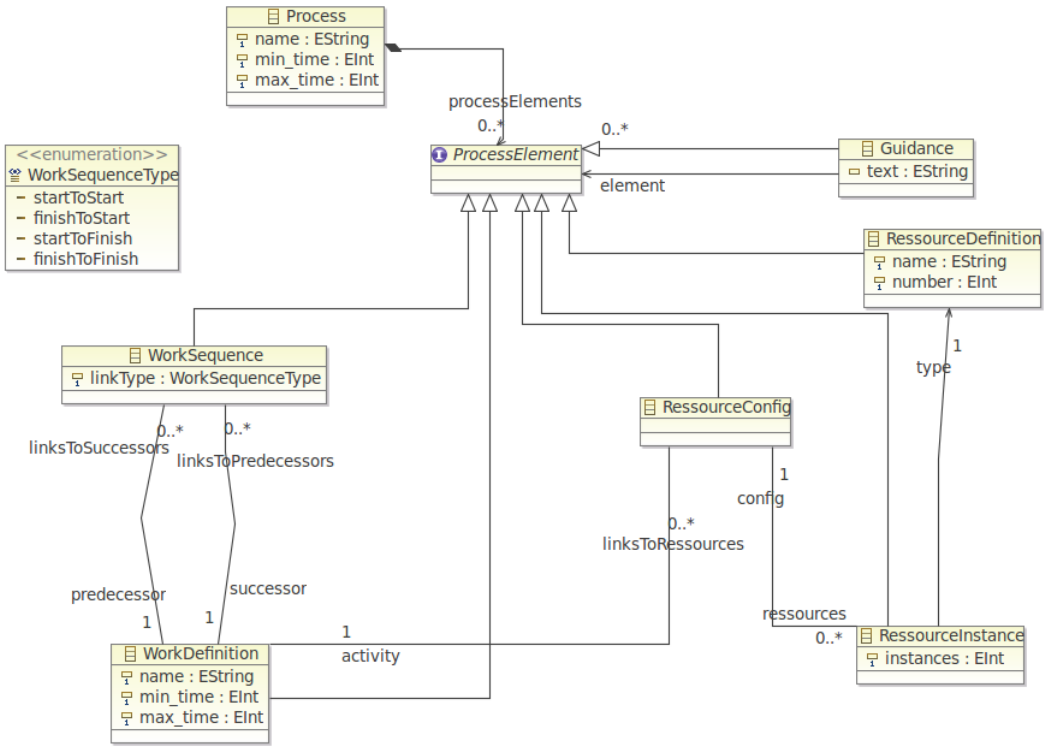

FIGURE 3.2 – Méta-modèle avec ressources alternatives

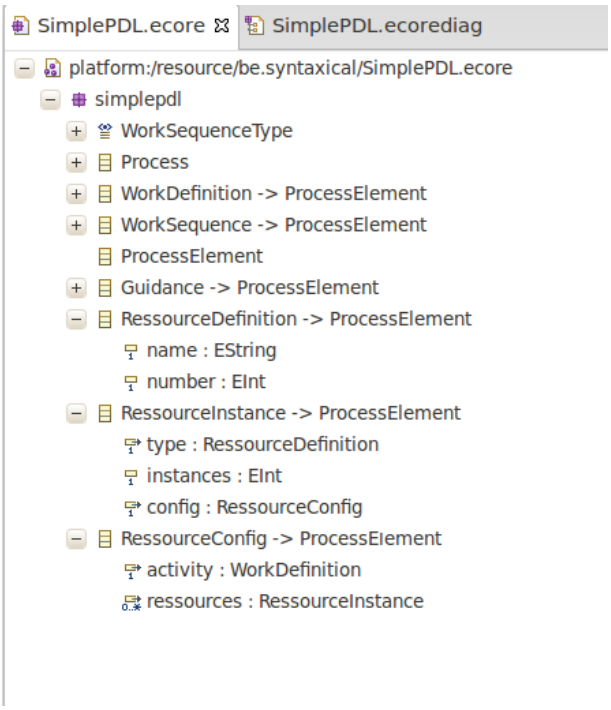

FIGURE 3.3 – Méta-modèle sous forme graphique

Pour permettre ce regroupement, on rajoute une classe RessourceConfig. Cette classe va contenir des liens vers

des RessourceInstance. Ainsi, on va exprimer le fait qu'il peut exister plusieurs configurations mettant en oeuvre différents jeux de ressources.

On remplace donc le lien de WorkDefinition a RessourceInstance, par un lien de WorkDefinition a RessourceConfig. Ce lien a une multiplicité 0.. car on peut avoir plusieurs Configurations pour une même WorkDefinition.

3.2.2 Transformation SimplePDL vers PetriNet

Prenons l'exemple suivant,

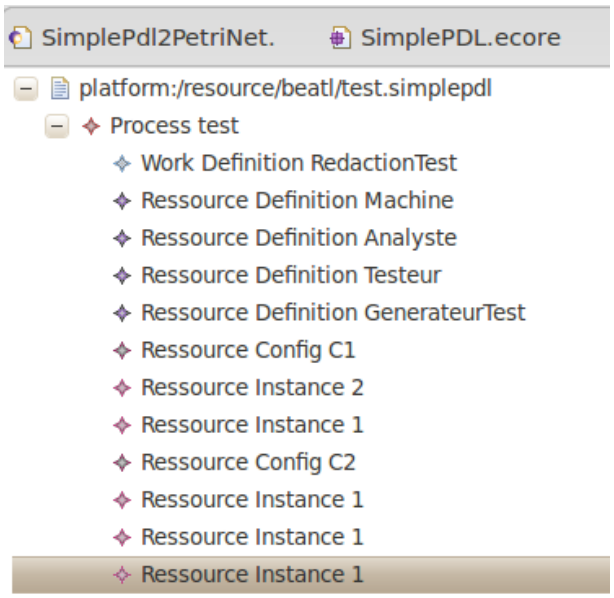

FIGURE 3.4 – Exemple mis en oeuvre

On dispose de 4 ressources : Machine, Analyste, Testeur, GenerateurTest. Par souci de lisibilité, on limite le nombre de WorkDefinition, à une seule : redactionTest. On définit également deux configurations C1 et C2. C1 met en oeuvre deux WorkInstance (Machine, 2) et (GenerateurTest,1). C2 met en oeuvre trois WorkInstance (Analyste,1), (GenerateurTest,1) et (Machine,1).

Cet exemple va nous permettre d'illustrer les transformations qui doivent être effectuées.

Il faut donc reprendre la transformation SimplePDL vers PetriNet et transformer la gestion des Ressources. Toujours par souci de lisibilité, on s'affranchit de la génération d'observateurs.

Le principe du réseau de pétri gérant la configuration des ressources est le suivant :

- Il est possible de choisir entre chacune des configurations (si du moins il y a suffisament de ressources)
- Une fois que l'on a choisi de démarrer avec une configuratio, on ne pourra plus choisir l'autre
- Les Ressources utilisées doivent être restituées à la fin de l'activité, et qui plus est dans la bonne configuration.

Il ressort de cette analyse que l'on va devoir utiliser des places nous indiquant :

- Quelle configuration est choisie : ConfigurationName_used
- Une configuration a été choisie : ProcessName_ress_1

Et des transitions pour :

– Choisir une configuration : ConfigurationName.start
– Rendre les ressources : ConfigurationName.finished

Pour l'exemple cité précédement, on obtient donc le réseau de pétri suivant :

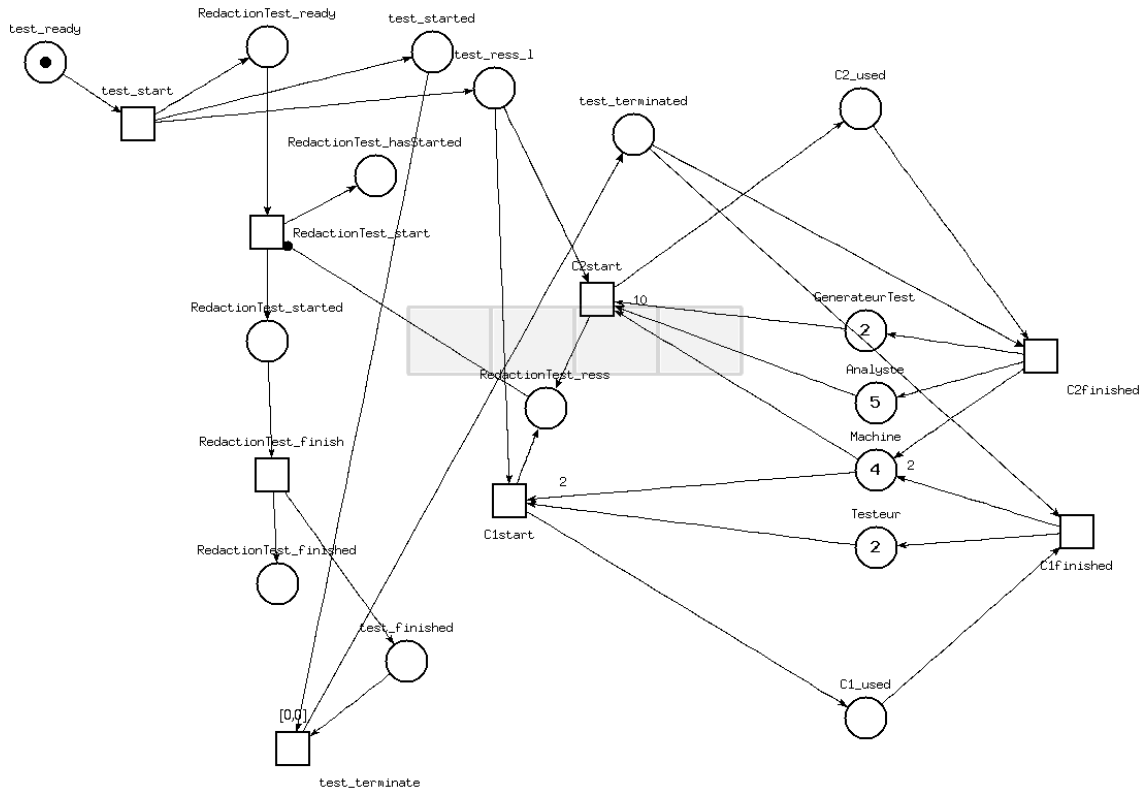

FIGURE 3.5 – Réseau de pétri obtenu

Bien entendu, le code de la transformation a été adapté pour fornir ces changements. Il est disponible en annexe.

Pour ce qui est de la transformation PetriNet2Tina, il n'y a toujours aucune raison de la modifier car, on a pu conserver la même structure pour le méta-modèle PetriNet.ecore.