Projet Génie du Logiciel et des Systèmes
Validation de modèles de procédés

Philippe Leleux
Groupe F

Jessica Hornik
Groupe F

25 janvier 2013

Résumé

Ce projet consiste à compléter la chaîne de vérification de modèles de processus SimplePDL vue en TP et :

- d'y ajouter les ressources et le temps.
- de décomposer une activité et gérer plus finement les ressources en faisant en sorte que chaque activité et sous-activité puisse se mettre en pause, le temps continuant à s'écouler.

Table des matières

I Ajout des ressources et du temps **5**

1 Spécification **5**
1.1 Principe . . . 5
1.1.1 Les ressources . . . 5
1.1.2 Le temps . . . 5
1.2 Solution . . . 5
1.2.1 Les ressources . . . 5
1.2.2 Le temps . . . 5

2 Modèle Ecore **6**
2.1 Modèle de départ . . . 6
2.2 SimplePDL . . . 6
2.3 PetriNet . . . 7

3 Règles OCL **7**
3.1 Règles OCL de départ . . . 7
3.2 SimplePDL . . . 8
3.3 PetriNet . . . 8

4 Éditeur Graphique et xText **8**
4.1 Éditeur Graphique : GMF . . . 8
4.1.1 Affichage . . . 8
4.1.2 Contraintes . . . 8
4.2 xText . . . 9

5 SimplePDL to PetriNet **9**
5.1 Forme de départ . . . 9
5.2 Ajout des Ressources et Paramètres . . . 9
5.3 Ajout du Temps . . . 10

6 PetriNet to Tina **10**

7 Validation de la transformation **10**

8 Propriétés LTL **11**
8.1 Syntaxe . . . 12
8.2 Règles . . . 12
8.2.1 Général . . . 12
8.2.2 Règles sur les `WorkSequence` . . . 13
8.2.3 Ressources . . . 13

II Extensions du modèle de procédés **14**

1 Spécification **14**
1.1 Principe . . . 14
1.1.1 Décomposition d'une activité . . . 14
1.1.2 Gestion plus fine des ressources . . . 14
1.2 Solution . . . 14
1.2.1 Sous-activités . . . 14
1.2.2 La Pause . . . 14

2 Modèles Ecore **15**
2.1 SimplePDL . . . 15
2.1.1 Eclass Activities et `SubWorkDefinition` . . . 15
2.1.2 Eclass `ParameterWD` et `ParameterSWD` . . . 15
2.2 PetriNet . . . 16

3 Règles OCL de SimplePDL **16**
3.1 Adaptation des règles de départ . . . 16
3.2 Règles pour la décomposition des activités . . . 16

4 Éditeur Graphique et xText **16**
4.1 Éditeur Graphique : GMF . . . 16
4.2 xText . . . 16

5 SimplePDL to PetriNet to Tina **17**
5.1 Version finale avec les sous activités . . . 17
5.2 Ajout de la pause . . . 18
5.3 PetriNet to Tina . . . 18

6 Validation de la transformation **19**

7 Propriétés LTL **19**
7.1 Décomposition d'une activité 19
7.2 Gestion des ressources 19
7.3 Compléments 20

Présentation générale

Le but est de compléter le travail fait en TP consistant à créer une chaîne de vérification de modèles de processus SimplePDL. Pour cela nous avons à notre disposition :

- les outils de eclipse-modeling qui permettent de créer des modèles, d'effectuer des transformations d'un modèle un autre, d'un modèle à du texte et le contraire et de vérifier des règles OCL.
- les outils de la boite à outils Tina qui permettent de manipuler des réseaux de Petri, textuellement et graphiquement mais surtout des outils de model-checking pour vérifier des propriétés LTL.

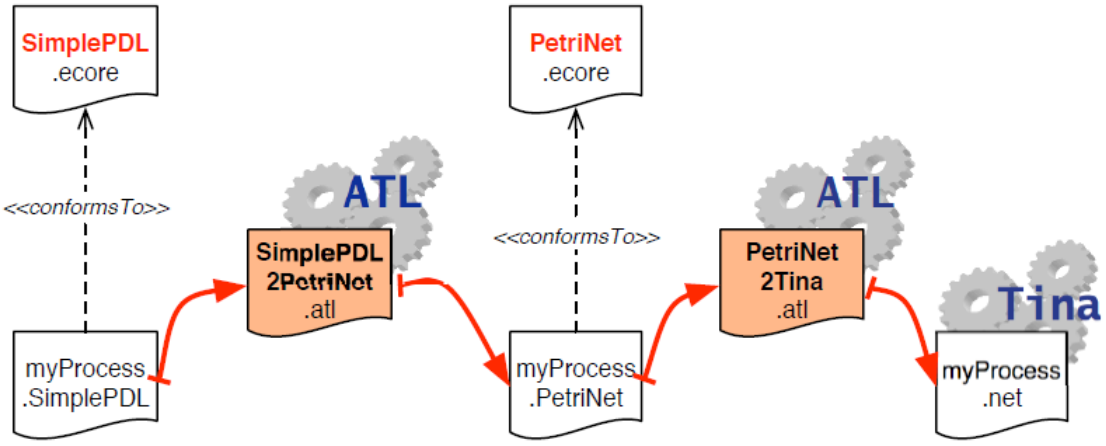

FIGURE 1 – Schéma des transformations de modèles à modèles

Nous ne suivrons pas à la lettre ce schéma : en effet, il nous sera plus profitable d'appliquer une transformation de modèle à texte entre PetriNet et Tina, grâce à `Acceleo`.

Première partie

Ajout des ressources et du temps

1 Spécification

1.1 Principe

1.1.1 Les ressources

Une activité peut avoir besoin de ressources pour pouvoir se dérouler. Ces ressources peuvent correspondre à n'importe quoi, tout ce dont l'activité peut avoir besoin, par exemple pour écrire ce rapport nous avons besoin (d'au moins) un cerveau et d'un ordinateur. Cette ressource sera seulement identifiée par un nom et le nombre de cette ressource disponible pour le processus.

Chaque activité peut avoir besoin de plusieurs ressources pour être réalisée mais une même occurrence de ressource ne peut bien sûr pas être partagée. Une activité ne peut démarrer que si les ressources dont elle a besoin sont disponibles en quantité suffisante. Les ressources nécessaires sont alors prises au moment du démarrage de l'activité et rendues à la fin de l'exécution de celle-ci.

1.1.2 Le temps

Une activité se déroule en un certain temps que l'on modélisera par `min_time` et `max_time`. `min_time` correspond à la durée minimum entre le moment où l'activité est prête à démarrer et celui où elle doit démarrer réellement. `max_time` correspond à la durée entre le moment où l'activité est prête à démarrer et celui avant le quel elle doit s'arrêter. Si l'activité se termine avant `min_time`, elle a fini trop tôt et si elle finit après `max_time` elle finit trop tard.

De manière Similaire, le processus est caractérisé par un temps minimal et un temps maximal et toutes les activités du processus doivent pouvoir s'exécuter dans cet intervalle de temps.

1.2 Solution

1.2.1 Les ressources

Pour modéliser les ressources il faut prendre deux éléments en compte :
- la ressource elle-même par ajout d'une classe.
- Le nombre d'occurrences nécessaires au fonctionnement de l'activité qu'il faudra modéliser par ajout d'une autre classe.

1.2.2 Le temps

Pour le temps tout ce qui va changer c'est la manière de transformer un modèle de SimplePDL à PetriNet. Il va falloir ajouter un réseau de Petri temporel qui va démarrer au démarrage de l'activité, ce réseau va contenir des place spécifiant que le processus est à l'heure, en retard à terminé trop tôt.

Il ne faudra pas oublier d'ajouter le temps de déroulement d'une activité dans la transformation.

2 Modèle Ecore

2.1 Modèle de départ

Les modèles que l'on va prendre au départ sont ceux obtenus en TP et qui permettent :
- de modéliser un `Process` avec `WorkDefinition`, `WorkSequence`, `Guidance` et `ProcessElement`.
- de modéliser un réseau de Petri complet.

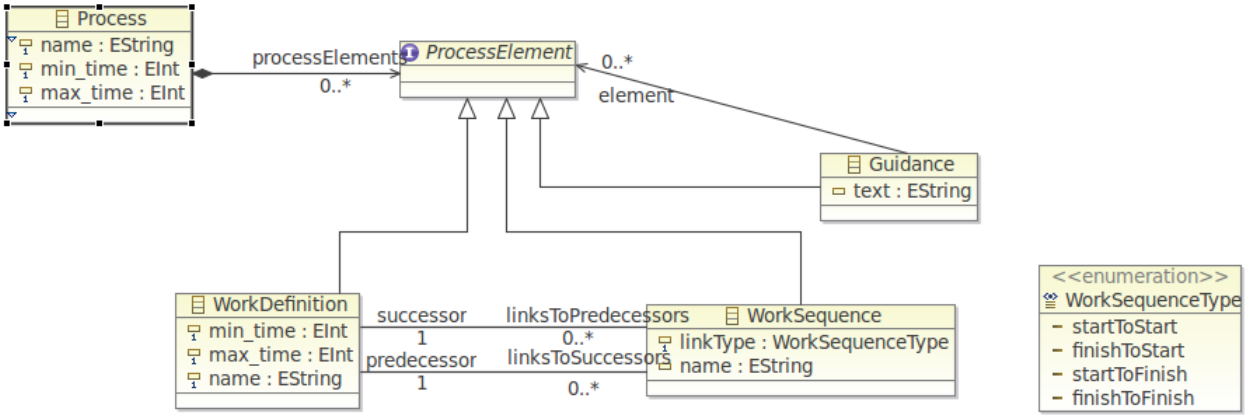

FIGURE 2 – Méta-modèle de SimplePDL

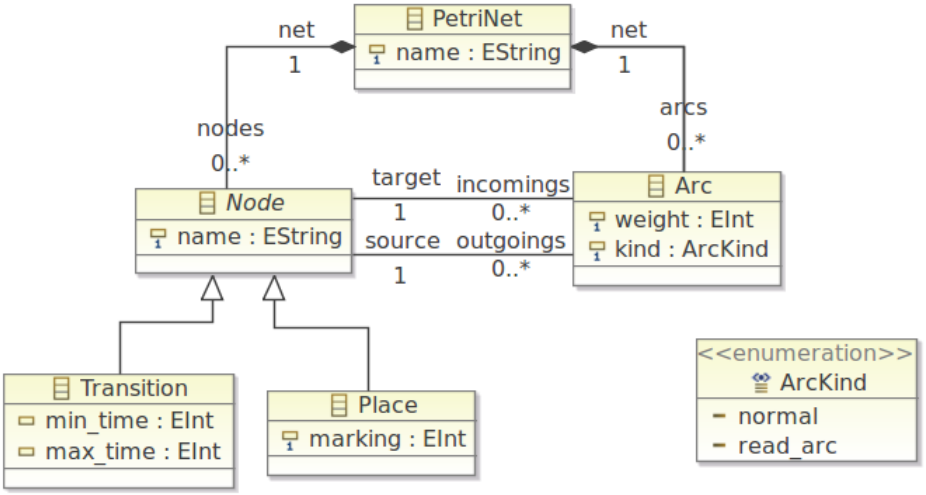

FIGURE 3 – Méta-modèle de PetriNet

2.2 SimplePDL

Pour modéliser les ressources, on va ajouter une classe du nom `Resource` qui aura pour attribut le nombre d'occurrences de départ. Le processus pourra avoir plusieurs ressource qui pourront être utilisés par les activités (`WorkDefinition`). Cependant ce n'est pas suffisant, il faut connaître le nombre d'occurrences de la ressource dont chaque activité a besoin. On va donc ajouter une classe `Parameter`.

Un `Parameter` représente un lien vers la ressource, il est caractérisé par deux attributs : son nom, le nombre d'occurrences de la ressource dont l'activité a besoin, une référence vers une Ressource et une référence vers l'activité à laquelle il est affilié. Chaque activité possèdera une référence vers plusieurs `Parameter`.
Après avoir ajouté ces éléments au modèle, il faudra également modifier les transformations vers PetriNet et les contraintes OCL en accord avec cela.

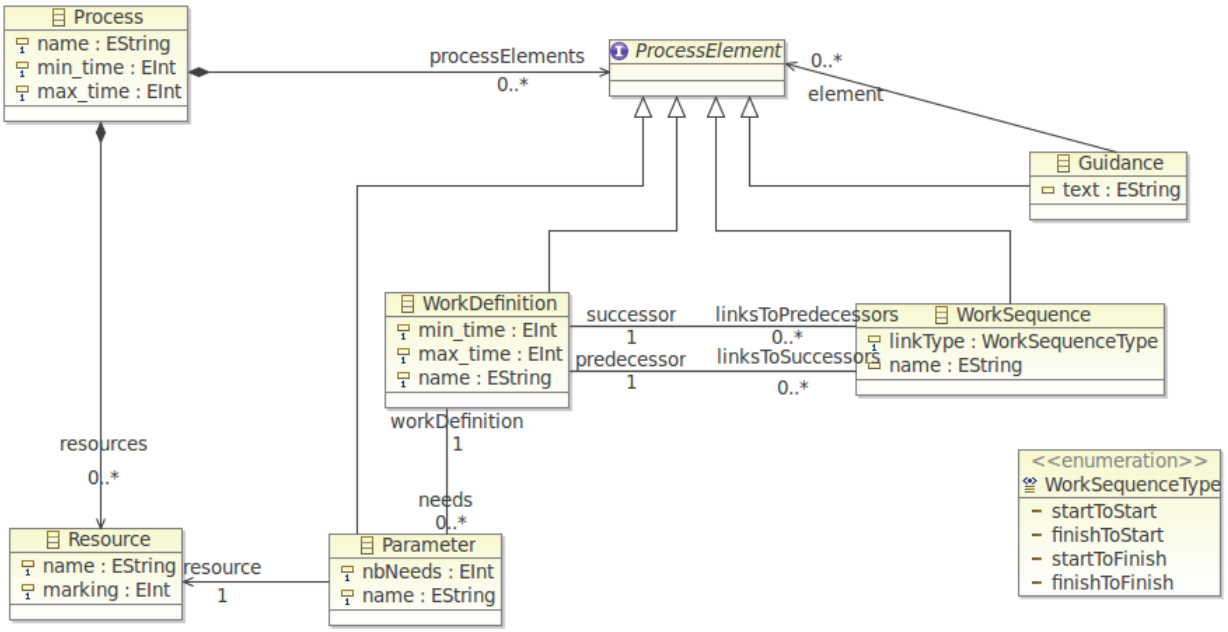

FIGURE 4 – Méta-modèle de SimplePDL avec ressources

2.3 PetriNet

Nous n'avons rien eu à changer sur le modèle dans cette partie du projet.

3 Règles OCL

3.1 Règles OCL de départ

Nous partons au départ avec des contraintes OCL basiques pour SimplePDL :
- Le nom de tous les `ProcessElement` sauf les `WorkSequence` est défini et de taille > 0.
- Les `WorkDefinition` ont toutes un nom différent.
- Les `WorkSequence` ne sont pas réflexives.
- Toutes les `WorkSequence` partant ou arrivant à une `WorkDefinition` et toutes les `WorkDefinition` desquelles partent ou sur lesquelles arrivent les `WorkSequence` appartiennent au `Process`.

Et les contraintes OCL pour PetriNet :
- Le nom du PetriNet et des `Node` est défini et de taille > 0.
- Toutes les places ont un nom différent et les transitions idem.
- Le PetriNet de tous les éléments est bien le PetriNet actuel.
- Le `min_time` des transitions est positif et le `max_time` en est supérieur.
- Le marking des `Place` et le weight des `Arcs` est défini et positif.

- Toutes les transitions lient une place à une transition ou le contraire.
- Tous les `Node` source et target des arcs ainsi que tous les Arc partant ou arrivant à un `Node` appartiennent au PetriNet.

3.2 SimplePDL

Nous avons besoin d'ajouter des règles OCL :
- pour les `Resource` et `Parameter` :
  - Tous les `Parameter` ont un nom différent et toutes les `Resource` idem.

    Les nombres d'occurrences de `Resource` (marking) et de `Parameter` (nbNeeds) sont définis, positifs et celui de `Parameter` est inférieur au marking de la `Resource` correspondante.
  - Deux `Parameter` d'une même `WorkDefinition` ne peuvent avoir la même `Resource`.
  - Tous les `Parameter` liés à une `WorkDefinition`, la `WorkDefinition` liée à chaque `Parameter` et la `Resource` liée à ces même `Parameter` appartiennent au `Process`.
- pour l'ajout du temps :
  - Pour le `Process` et les `WorkDefinition`, `min_time` est positif et `max_time` est soit supérieur à `min_time` soit égal à -1.

    `max_time` du `Process` est soit supérieur au `max_time` de chaque `WorkDefinition` soit égal à -1.

3.3 PetriNet

Il n'y a rien à changer pour le modèle PetriNet.

4 Éditeur Graphique et xText

4.1 Éditeur Graphique : GMF

4.1.1 Affichage

En créant l'éditeur graphique (et en recommençant encore et encore après chaque petit changement du modèle), il faut choisir quels éléments correspondront à un nœud et quels éléments correspondront à un arc orienté.

Au final, nous avons décidé de nous inspirer du schéma Figure1 du sujet et donc de représenter :
- Les `WorkDefinition` par un nœud style Ellipse.
- `Resource` et `Parameter` par un nœud style RoundedRectangle.
- Les `WorkSequence` par des arcs entre predecessor et successor (en affichant le type de la `WorkSequence`).
- Les références `WorkDefinition` des `Parameter` par des arcs entre le `Parameter` et la `WorkDefinition` associée.

4.1.2 Contraintes

Nous avons ajouté à chaque élément Link Mapping (c'est-à-dire chaque arc) un fils Link Constraint spécifiant que la source ne peut pas être égale à la cible pour ne pas pouvoir lier un nœud à lui-même.

Nous avons également décidé d'ajouter des Audit pour que les règles OCL soient vérifiées en direct lors de la création d'un `Process` avec l'éditeur graphique.

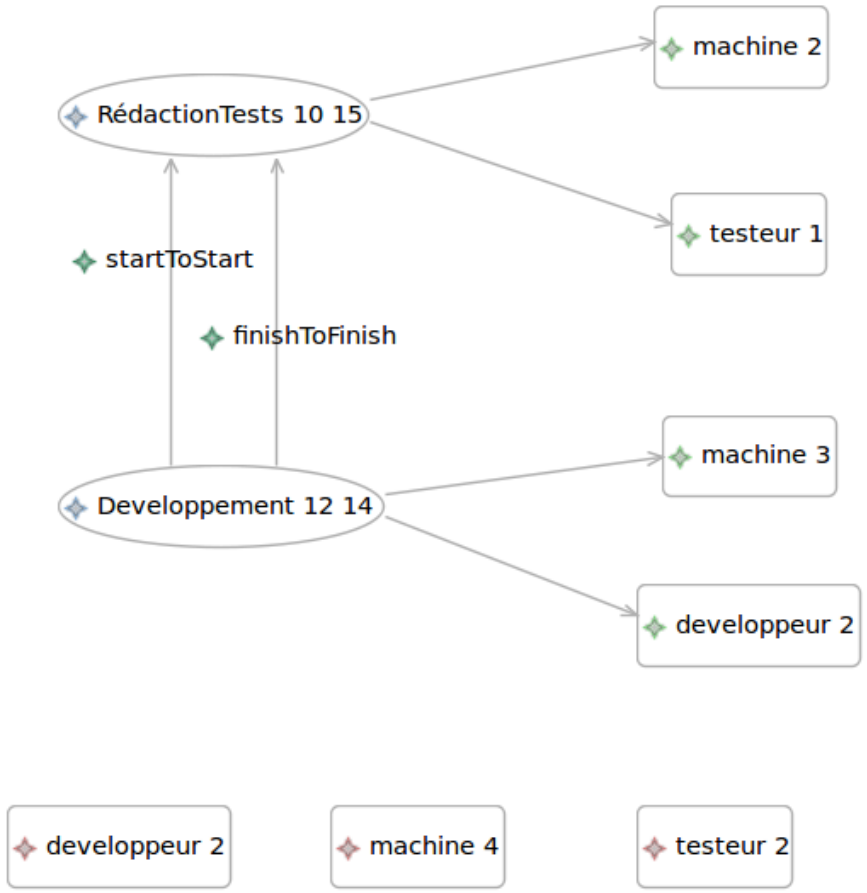

FIGURE 5 – Exemple de processus, affiché avec l'éditeur graphique

4.2 xText

Nous avons décidé de continuer la syntaxe concrète textuelle **simple** en ajoutant les éléments suffisants pour pouvoir, en effectuant le MWE2 WORKFLOW, générer le même modèle que celui qu'on possède. Nous n'avons donc pas utilisé la syntaxe générée automatiquement à partir du modèle de départ.

Voici un exemple illustrant la syntaxe développée :

5 SimplePDL to PetriNet

5.1 Forme de départ

On part d'une transformation créant un réseau de Petri modélisant un `Process` simple, modèle de départ. Voici ce que la transformation de base donne sur un exemple simple (deux `WorkDefinition`, une `WorkSequence`)

5.2 Ajout des Ressources et Paramètres

Une `Resource` est modélisée par une place en réseau de Petri.

Developpement.pdlp

```
process Developpement [20,50] {
    wd Conception [10,16] (par concepteur:2 of concepteur
                           par machine:2 of machine)
    wd RedactionDoc [8,12] (par redacteur:1 of redacteur
                            par machine:1 of machine)
    wd Development [12,14] (par developpeur:2 of developpeur
                            par machine:3 of machine)
    wd RedactionTest [10,12] (par testeur:1 of testeur
                              par machine:2 of machine)
    ws f2f from Conception to RedactionDoc
    ws s2s from Conception to Development
    ws s2s from Conception to RedactionTest
    ws f2f from Development to RedactionTest
} (
    res concepteur:3
    res redacteur:1
    res developpeur:2
    res testeur:2
    res machine:4
)
```

FIGURE 6 – Developpement.pdlp : syntaxe xText

Un `Parameter` correspond à deux arcs liant d'une part la `Resource` à la transition t_start de la `WorkDefinition` (elle ne peut commencer sans la `Resource`) et d'autre part la transition t_finish de la `WorkDefinition` à la `Resource` (lorsqu'elle finit on relâche la ressource).

5.3 Ajout du Temps

Le temps est modélisé par un réseau de Petri annexe qui est lancé lorsque la `WorkDefinition` commence (t_start), et qui a trois transition, une qui est franchie si la `WorkDefinition` se déroule est dans les temps, une si elle est en retard et une dernière si elle est en retard.

Pour bien spécifier que le processus est dans les temps ou en retard, il faut 2 read_arc qui spécifieront que le processus est en cours (p_running) et pour qu'il qu'il soit en avance il faut qu'il soit terminé, signalé par un read_arc à partir de p_finished.

Pour finir il faut que tout cela fonctionne réellement, on ajoute donc le temps sur certains arcs.

6 PetriNet to Tina

Nous avons choisi d'écrire cette transformation en finissant celle du TP écrite dans un projet `Acceleo`, en effet la syntaxe est simple à comprendre et engendre le fichier texte rapidement et facilement.

7 Validation de la transformation

Pour valider la transformation de SimplePDL à PetriNet, on crée une série de tests sous formes de modèles SimplePDL :

– un procédé élémentaire contenant une ressource, une `WorkDefinition` avec un `Parameter`.

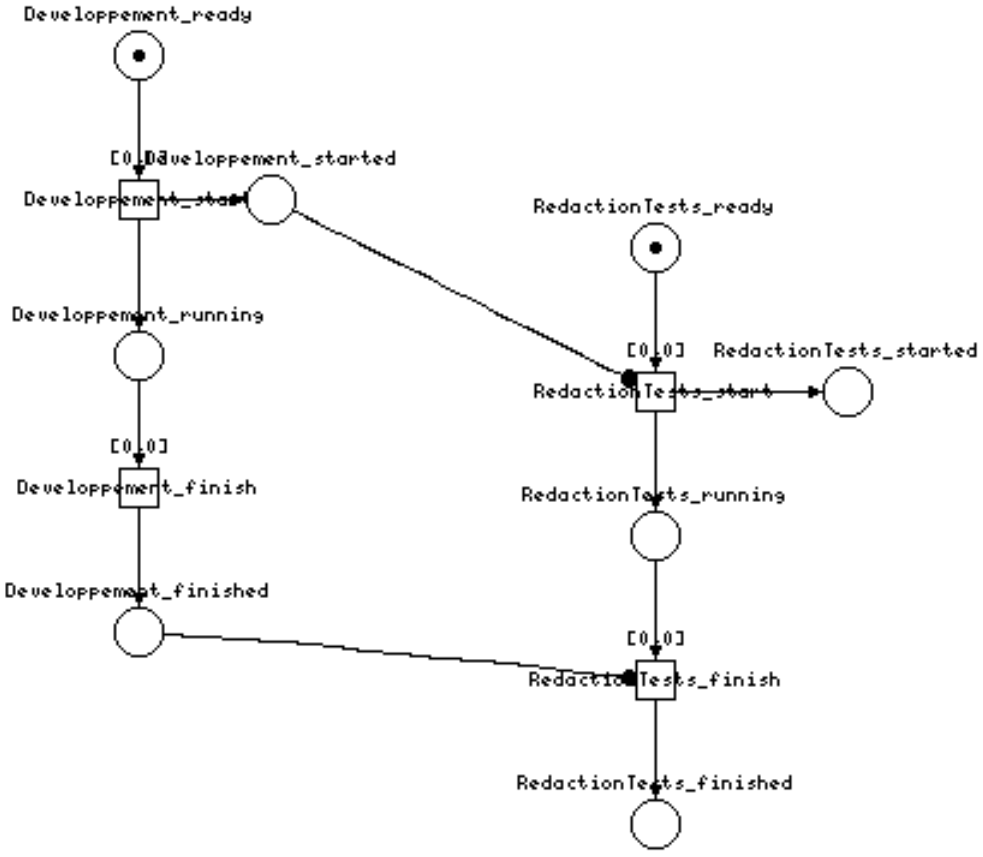

FIGURE 7 – Réseau de pétri représentant un `Process` simple

– l'exemple de la figure 1 du sujet mais sans les `Resource`.
– ce même exemple mais en rajoutant les `Resource` et `Parameter`.

Le point important est que pour nous allons en fait tester la transformation de SimplePDL à PetriNet en même temps que celle de PetriNet vers Tina : il ne faut pas se leurrer pour vérifier les modèles PetriNet obtenus, il faut les observer grâce à nd de tina.

Après avoir affiché les procédés et vérifié qu'ils sont bien compatibles avec nos attentes nous avons lancé le **stepper simulator** de la boîte à outil de Tina. Nous n'avons pas pu vérifier si c'est notre modèle qui a un problème ou simplement le Rand qui n'est pas très aléatoire, cependant en se mettant en mode untimed ou en avançant le temps petit à petit, on ne trouve aucun problème dans la transformation.

8 Propriétés LTL

Nous avons choisi de générer des fichiers de vérification de contraintes temporelles à l'aide d'une transformation modèle to text `Acceleo`.

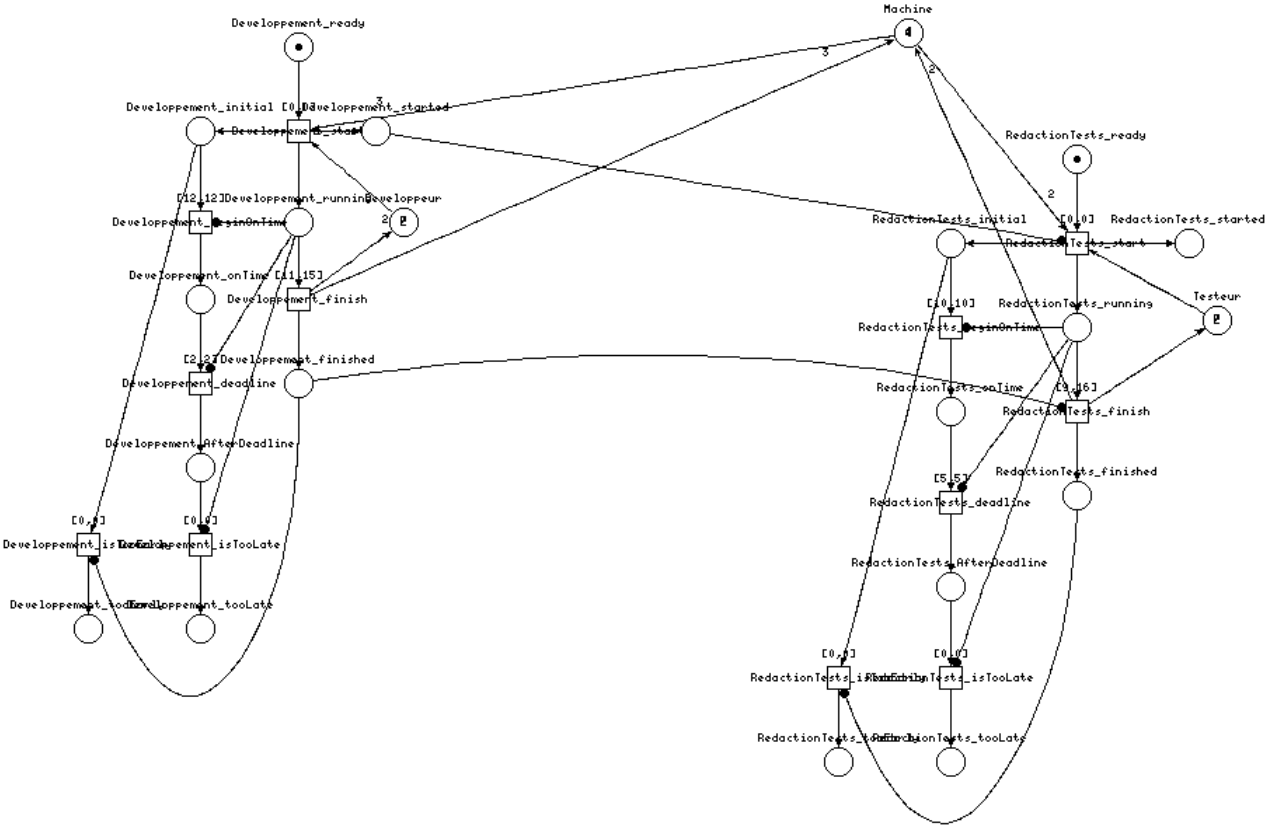

FIGURE 8 – Réseau de pétri représentant un `Process` simple avec ressources et temps

8.1 Syntaxe

Il a été pratique de définir une nouvelle syntaxe à l'aide de la syntaxe basique, afin de simplifier certaines notations. Cette simplification se fait en début du fichier généré. Par exemple,

```
infix p & q = p /\ q; # Redefinition du ET logique
```

On définit une opération `suit` qui dit que si `q suit p`, c'est que si `p` arrive, alors forcémment, il y aura `q`, et une opération `finished` qui renvoie vrai si le `Process` est fini.

8.2 Règles

Nous avons également pu créer des règles LTL directement depuis le modèle SimplePDL :

8.2.1 Général

- Quand le déroulement est bloqué, c'est que le process est terminé
- Quand ça atteint les conditions de `finished`, ça l'atteint définitivement.
- Il ne peut pas y avoir `process_finished` et une `WorkDefinition` non `finished`.
- Le fait d'atteindre la condition de terminaison n'est pas inatteignable. (On invalide la propriété LTL affirmant le contraire)

8.2.2 Règles sur les `WorkSequence`

- On s'assure qu'un lien `finishToStart` fait en sorte qu'on finisse le précédent avant de commencer le suivant
- On procède de même pour les autres types de liens de `WorkSequence`.

8.2.3 Ressources

- Le modèle ne produit pas de ressources.
- Quand la fin est atteinte, les ressources sont restituées.
- Si et quand la fin est atteinte, toutes les activités doivent être en `p_finished` et les ressources restituées

Le modèle ne terminant pas nécessairement (il peut y avoir des pauses de temps indéfini), on définit des règles LTL dépendant du fait que ça finisse. Cependant, ce n'est pas une contrainte aberrante, au vu du fait que la probabilité que le processus termine quand le temps s'étire vers l'infini converge vers 1, si le tirage est aléatoire. De plus, le modèle n'en est que plus réaliste.

Deuxième partie

Extensions du modèle de procédés

1 Spécification

1.1 Principe

Nous avons choisi de traiter les extensions : décomposer et une activité et mettre une activité en pause que nous allons combiner sur la même transformation.

1.1.1 Décomposition d'une activité

Nous envisageons maintenant la possibilité de décomposer une activité. À l'image d'un processus, une activité peut contenir d'autres activités que nous appellerons sous-activités. Bien entendu, une sous-activité peut également être décomposée.

Une sous-activité ne peut utiliser que les ressources réservées par l'activité englobante. Une activité composée est considérée :

- commencée quand ses ressources ont été allouées
- terminée quand toutes ses sous-activités sont terminées (et les ressources libérées).

1.1.2 Gestion plus fine des ressources

Pendant le déroulement d'un procédé, on veut faire en sorte qu'il soit possible de mettre une activité en pause. Ceci permet de libérer les ressources qu'elle utilisait. Elles peuvent alors être affectées à une autre activité.

Pour pouvoir reprendre, l'activité devra retrouver les ressources nécessaires à son déroulement. On considère que le temps de réalisation de l'activité suspendue continue à être décompté.

1.2 Solution

1.2.1 Sous-activités

Une sous-activité (nouvelle classe) va ressembler une activité, les choses qui changent sont :

- une sous-activité ne commence que lorsque l'activité a changé, et elle signale à l'activité englobante lorsqu'elle se termine, il faudra donc changer sa transformation en PetriNet pour prendre cela en compte.
- une sous-activité utilise les ressource de l'activité englobante, cela va également entraîner un changement dans la transformation.

1.2.2 La Pause

Pour la pause nous ne devrons changer que la transformation, il faudra ajouter une transition pour entrer en pause et une autre pour en sortir.

Comme l'extension des sous-activités est combinée, une activité englobante qui se met en pause met les sous-activités en pause. Pour cela nous allons juste empêcher la sous-activité de se terminer si l'activité est en pause.

2 Modèles Ecore

2.1 SimplePDL

2.1.1 Eclass Activities et `SubWorkDefinition`

La classe `SubWorkDefinition` correspond à une sous-activité, elle a une référence vers un parent. Ce parent devrait être de type `WorkDefinition`, sauf qu'une `SubWorkDefinition` peut aussi être décomposée, il faut donc que ce parent puisse également être de ce type. Nous avons pour cela créé la classe abstraite Activities dont héritent `WorkDefinition` et `SubWorkDefinition`. Chacune de ces classes a une référence vers plusieurs activities qui seront leurs enfants.

2.1.2 Eclass `ParameterWD` et `ParameterSWD`

La première idée qui vient lorsqu'on se dit que la `SubWorkDefinition` doit utiliser les ressources allouées par l'activité englobante, c'est que celle-ci met à disposition une ressource interne pour ses enfants.

On pourrait commencer par se dire qu'une `SubWorkDefinition` aura une référence vers un `Parameter` comme défini avant mais il y a un problème pour la transformation : autant la `WorkDefinition` utilise une `Resource` et donc un `Parameter` qui a une référence à `Resource`, autant une `SubWorkDefinition` se sert de la ressource interne du parent, il serait donc normal qu'elle est une nouvelle sorte de paramètre : `ParameterSWD` qui a une référence vers un paramètre. De plus, ce paramètre peut être à la fois celui d'un `WorkDefinition` et d'une `SubWorkDefinition`.

Nous allons donc renommer la classe `Parameter` en `ParameterWD`, créer la classe abstraite `Parameter` dont héritent `ParameterWD` et `ParameterSWD`, classe crée avec une référence vers un `Parameter` (abstrait).

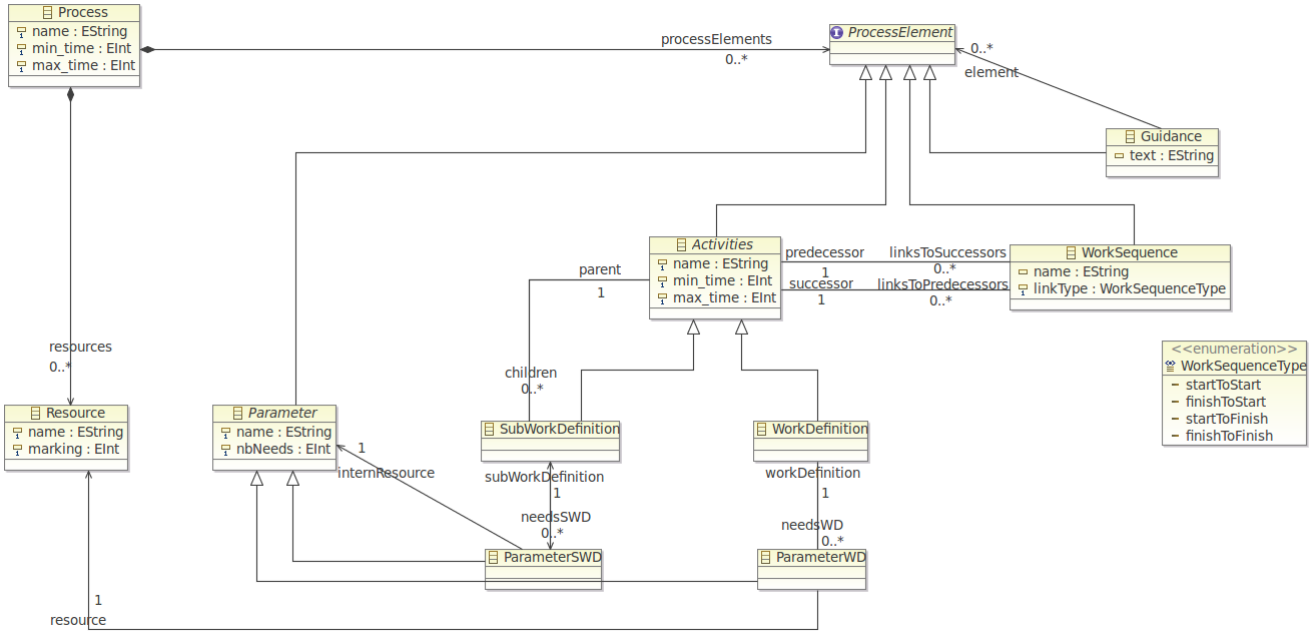

FIGURE 9 – Méta-modèle Ecore du SimplePDL avec `Parameter`

2.2 PetriNet

Le seul changement à apporter est de rajouter un ArcKind : inhibitor.

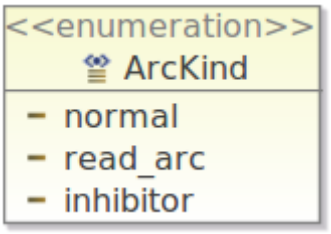

FIGURE 10 – Arc Kind

3 Règles OCL de SimplePDL

3.1 Adaptation des règles de départ

Nous ne préciserons pas dans la suite les règles simples qui ont déjà été définies au départ et qui changent seulement de propriétaires (avec la création de nouvelle classe ou le renommage d'autres classes).

3.2 Règles pour la décomposition des activités

Les règles à ajouter se rapportent aux familles d'activités :
- Une sous-activité ne peut pas être son propre parent.
- Le parent d'une sous-activité et les enfants d'une activité appartiennent au procédé.

4 Éditeur Graphique et xText

4.1 Éditeur Graphique : GMF

Dans l'éditeur graphique, nous avons fait les même choix que précédemment et ces choix s'étendent aux classes de même type, pour les nouvelles classes :
- Les Activities sont des nœuds de type Ellipse.
- Les `Parameter` sont des RoundedRectangle.
- Le lien `WorkDefinition` ou `SubWorkDefinition` d'un `Parameter` est un arc liant le `Parameter` à son activité.
- Le lien parent d'une sous-activité est un arc qui la relie à son parent.

Nous avons également créé à nouveau les Link Constraint et les audit avec les nouvelles contraintes OCL.

4.2 xText

Pour la syntaxe xText nous avons complété la syntaxe de la partie 1, toujours de sorte à ce que le modèle généré soit identique au modèle SimplePDL que l'on a créé.

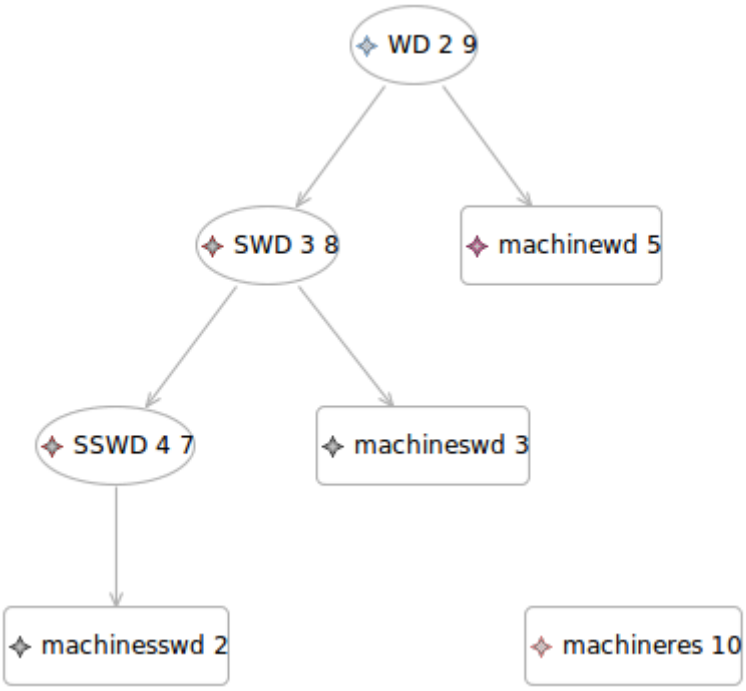

FIGURE 11 – Process vu sous l'éditeur graphique

Developpement2.pdlp

```
process Developpement2 [20,50] {
    wd Developpement [20, 50]
    swd Conception child of Developpement [10,16]
    swd RedactionDoc child of Developpement [8,12]
    swd Development child of Developpement [12,14]
    swd SeCasserLaTete child of Development [13, 13]
    swd EtLEcran child of Development [13, 13]
    swd RedactionTest child of Developpement [10,12]
    ws f2f from Conception to RedactionDoc
    ws s2s from Conception to Development
    ws s2s from Conception to RedactionTest
    ws f2f from Development to RedactionTest
    parWD Developpement_concepteur:3 of concepteur used by Developpement
    parWD Developpement_redacteur:1 of redacteur used by Developpement
    parWD Developpement_developpeur:2 of developpeur used by Developpement
    parWD Developpement_testeur:2 of testeur used by Developpement
    parWD Developpement_machine:4 of machine used by Developpement
    parSWD Conception_concepteur:2 of Developpement_concepteur used by Conception
    parSWD Conception_machine:2 of Developpement_machine used by Conception
    parSWD RedactionDoc_redacteur:1 of Developpement_redacteur used by RedactionDoc
    parSWD RedactionDoc_machine:1 of Developpement_machine used by RedactionDoc
    parSWD Development_developpeur:2 of Developpement_developpeur used by Development
    parSWD Development_machine:3 of Developpement_machine used by Development
    parSWD SeCasserLaTete_developpeur:1 of Developpement_developpeur used by Development
    parSWD SeCasserLaTete_machine:3 of Developpement_machine used by Development
    parSWD EtLEcran_developpeur:1 of Developpement_developpeur used by Development
    parSWD EtLEcran_machine:1 of Developpement_machine used by Development
    parSWD RedactionTest_testeur:1 of Developpement_testeur used by RedactionTest
    parSWD RedactionTest_machine:2 of Developpement_machine used by RedactionTest
} {
    res concepteur:3
    res redacteur:1
    res developpeur:2
    res testeur:2
    res machine:4
}
```

FIGURE 12 – Exemple de vue sous l'éditeur textuel xText

5 SimplePDL to PetriNet to Tina

5.1 Version finale avec les sous activités

La ressource est toujours une place

Une `WorkDefinition` et une `SubWorkDefinition` sont identiques si ce n'est qu'une activité en démarrant permet à une sous-activité de démarrer, d'où un arc et que la sous-activité en se terminant le signale à l'activité englobante pour qu'elle puise se terminer.

La version finale propose un réseau tel que la `WorkDefinition` prend la ressource et la met dans une place `internResource` avant de les reprendre pour rendre la ressource : c'est le `ParameterWD`. La `SubWorkDefinition` prend dans la `internResource` et met dans une autre place `internResource` (sa propre ressource interne pour être décomposée) puis la ressource interne allouée est reprise afin d'être rendue : c'est le `ParameterSWD`.

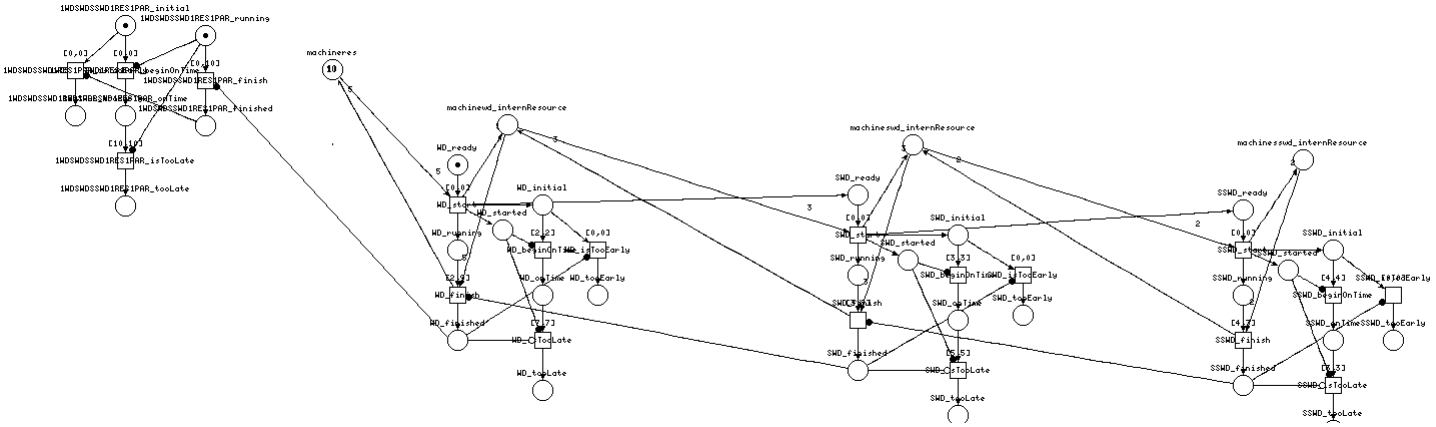

FIGURE 13 – Vue du réseau PetriNet d'un `Process`

5.2 Ajout de la pause

Pour ajouter la pause, on va simplement ajouter une transition (de `max_time` infini) partant de p_running (on met en pause l'état « en cours » de l'activité) ce qui conduit à un place en pause. Une transition nous permettra de sortir de cette pause pour revenir à running. Bien sûr, il faut rendre les ressource lors d'une pause, c'est donc à la première transition de la pause qu'on les rend (lorsqu'on se met en pause) et à la deuxième qu'on les reprend (on ne peut sortir de pause que si on peut prendre les ressources).

Si on met une activité en pause, cela met en pause également les sous-activités théoriquement, mais on doit tout de même garder l'état interne du système, on fait donc comme indiqué plus haut mais simplement si l'activité englobante est en pause, les sous-activités ne peuvent se terminer grâce à un arc inhibitor.

Un dernier point à remarquer est que le temps doit continuer à s'écouler pour l'activité pendant une pause, il faut donc modifier les informations donnés par l'activité au temps. On pourra désormais, en changeant les read_arc, passer la transition «à l'heure» à partir du moment ou on a commencé à un moment, et de même pour «en retard» avec en plus la contrainte que l'activité ne soit pas terminée (usage d'un inhibitor).

5.3 PetriNet to Tina

La transformation est entièrement gardée, à un détail près, il ne faut pas oublier l'ajout de l'ArcKind inhibitor pris en compte en syntaxe Tina comme un weight négatif.

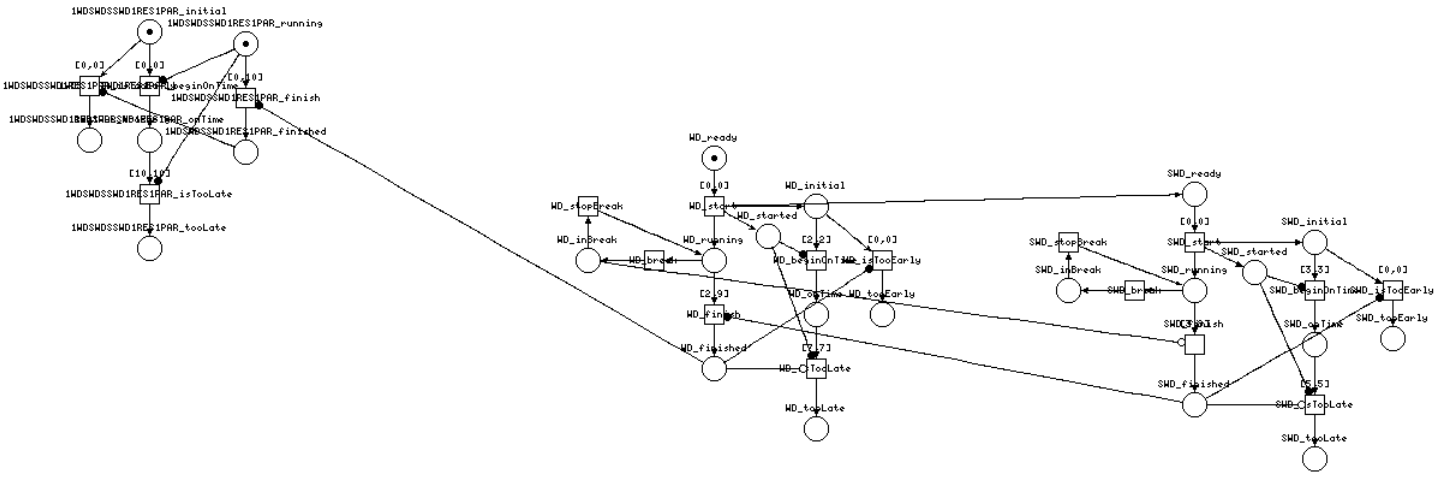

FIGURE 14 – PetriNet d'un `Process` avec pause

6 Validation de la transformation

Ici la série de test est :

- 1 procédé élémentaire avec 1 ressource, 1 `WorkDefinition` avec un `ParameterWD` qui a un enfant `SubWorkDefinition` avec un `ParameterSWD`.
- 1 procédé élémentaire avec 1 ressource, 1 `WorkDefinition` avec un `ParameterWD` qui a un enfant `SubWorkDefinition` avec un `ParameterSWD` qui a elle-même un enfant `SubWorkDefinition` avec un `ParameterSWD`.
- Le même exemple dérivé de la figure 1 du sujet du projet.
- Ce même exemple modifié pour faire apparaître une activité englobant le tout et 2 sous-activités pour une des activités de départ : development.

Après avoir effectué des tests, les résultats semblent concluants : Avec le premier exemple tout se passe comme prévu, les activités se terminent (et le plus souvent dans les temps, aux caprices du random près). Le partage de la ressource et l'effet de la pause sur celui-ci semblent bien maîtrisés et se passent sans problème apparent.

7 Propriétés LTL

Pour cette partie, il s'agit des mêmes propriétés à vérifier que pour la partie I, mais avec quelques unes de plus.

7.1 Décomposition d'une activité

On s'assure qu'une activité enfant ne commence que si l'activité parente est commencée, et ne termine que si l'activité parente a fini.

7.2 Gestion des ressources

- On s'assure que le nombre de ressources interne est toujours inférieur au nombre de ressources disponibles.
- Au moins une des `WorkDefinition` doit être soit en `p_ready`, `p_running`, `p_inBreak`, ou `p_finished`.

– De plus, si une `WorkDefinition` est en pause, alors, forcémment, elle passera en running au bout d'un moment.

7.3 Compléments

Afin de mettre en place ces propriétés, on définit quelques fonctions ("query") complémentaires de recherche d'éléments d'un certain type et vérifiant certains critères.

Conclusion

Ce projet nous a permis de traiter en profondeur de méta-modèles, et des transformations d'un méta-modèle à un autre. Il a été souvent fastidieux, car de manière répétée, il fallait recommencer les mêmes opérations de génération de code, et repérer les problèmes de l'IDE Eclipse, et ce, avec peu de documentation sur le net. Cependant, il a été instructif, sur ce domaine, et nous a permis de créer des modèles, et de les vérifier grâce à différentes méthodes, et par là même, d'en apprendre plus sur ces méthodes de vérification.