Rapport de Mini-Projet

Génie du Logiciel et des Systèmes

Chaîne de vérification de modèles de processus

Groupe M-02
Fainsin Laurent
Guillotin Damien

Département Sciences du Numérique
Deuxième année
2021 — 2022

Table des matières

1 Métamodèles (avec Ecore)

1.1 simplePDL.ecore

Ce projet se base sur un langage simplifié de modélisation de processus de développement, appelé SimplePDL. Nous sommes donc parti du modèle Ecore de base du modèle SimplePDL auquel nous avons ajouté progressivement des nouveaux éléments. Dans un premier temps, il a fallu ajouter la modélisation des guidances comme indiqué dans le sujet.

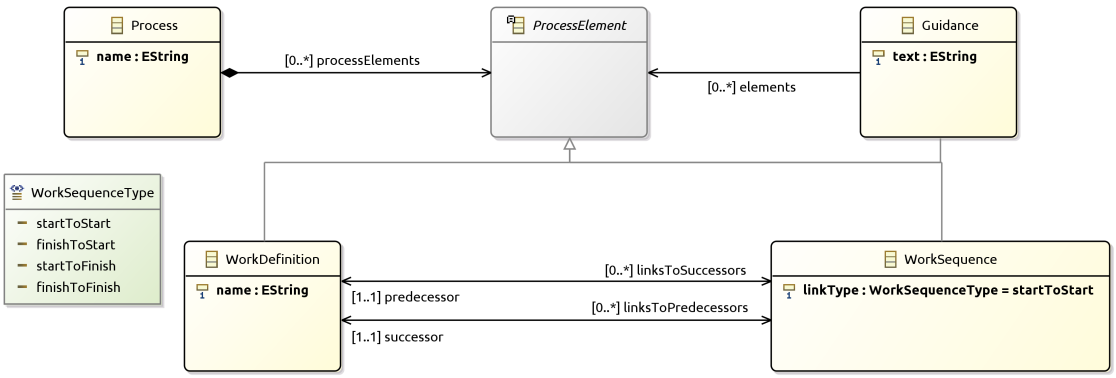

FIGURE 1 – Métamodèle simplePDL avec guidance

Nous avons ensuite dû choisir une manière de modéliser les ressources nécessaires au processus de développement. Notre choix de modélisation s'est porté sur : une Ressource (qui implémente ProcessElement) est demandée par le biai d'une Request elle-même générée par une WorkDefinition. On se retrouve donc avec un nouveau modèle Ecore de la forme :

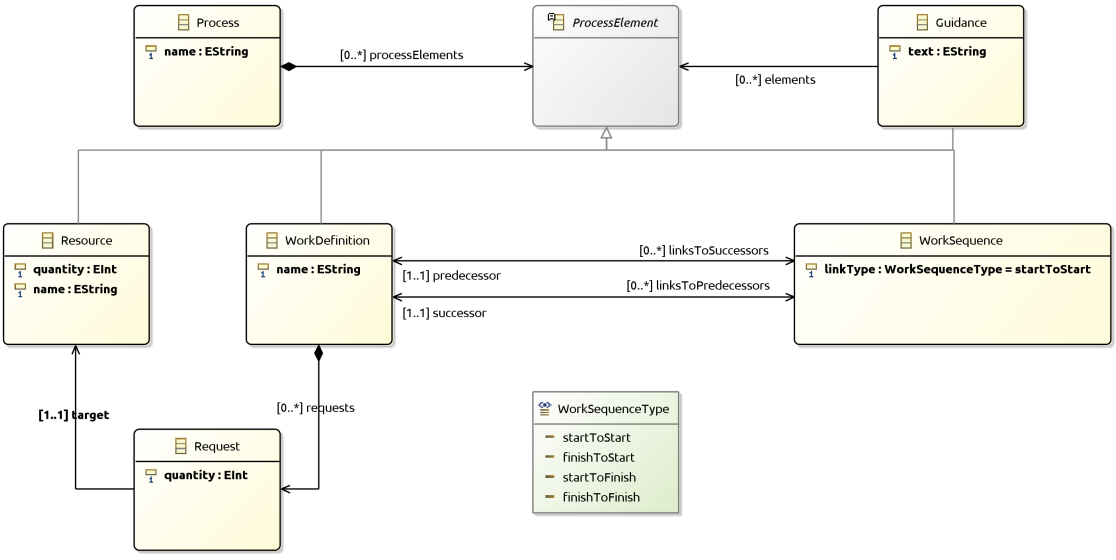

FIGURE 2 – Métamodèle simplePDL avec guidance et ressource

Les relations Ressource-Request et Request-WorkDefinition sont déclarées en EOpposite pour pouvoir facilement passer d'un fils à un parent et vice versa. Le modèle SimplePDL est maintenant complet pour représenter des processus de développement. Un exemple complet d'utilisation de ce modèle serait :

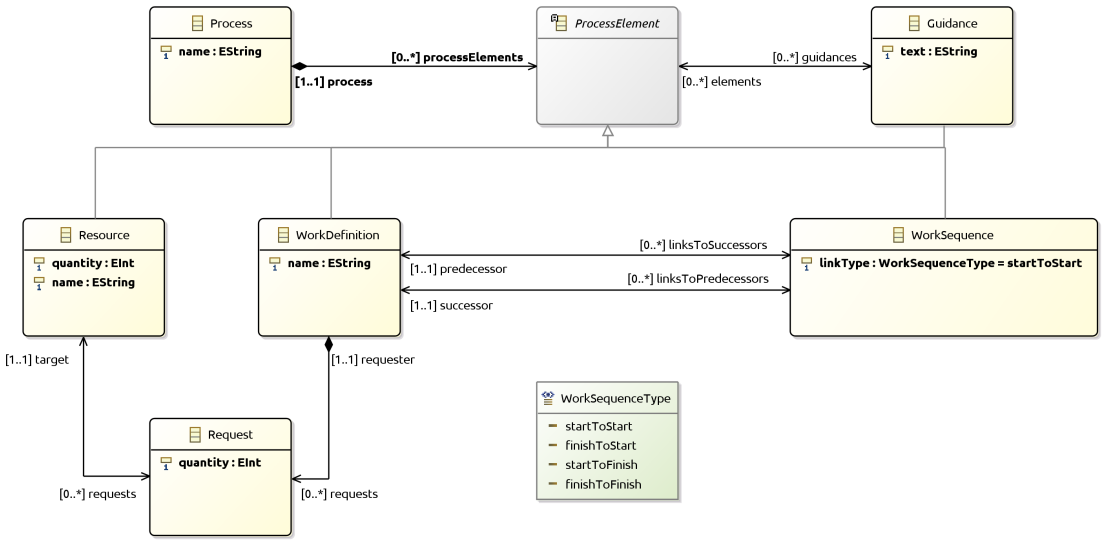

FIGURE 3 – Métamodèle simplePDL complet

1.2 petriNet.ecore

En se basant sur ce que l'on a vu pour le modèle SimplePDL, nous avons créé un modèle Ecore permettant de modéliser les réseaux de pétri. Nous avons modélisé un réseau comme étant composé de nœuds. Ces nœuds peuvent être les places ou des transitions. Ils sont donc nommés et reliés entre eux par des arcs. Ses arcs ont un attribut entier nommé weight pour indiquer le poids de l'arc ainsi qu'un boolean outgoing pour indiquer si ce dernier est dirigé d'une Place vers une Transition ou d'une Transition vers une Place. (Si outgoing est vrai, alors l'arc va de la transition vers la place.)

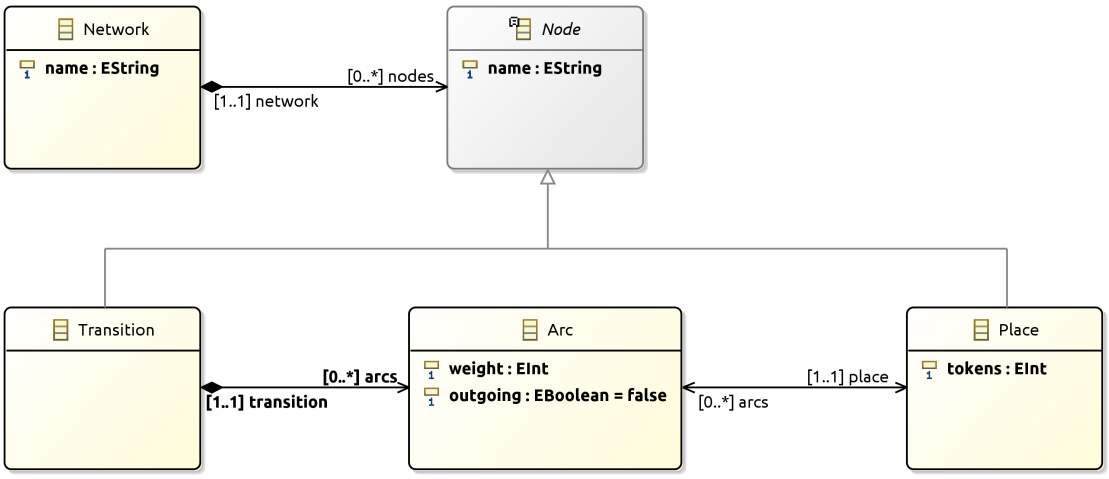

FIGURE 4 – Métamodèle petriNet complet

2 Sémantique statique (avec OCL)

Les contraintes OCL sont là pour vérifier des informations du modèle vis-à-vis du métamodèle. Elles assurent certains points de cohérence et permettent d'éviter les ambiguïtés.

2.1 simplePDL.ocl

Pour les modèles SimplePDL, nous contraignons les noms des Process au format camelCase. De même les noms des WorkDefinitions et des Resources doivent posséder au minimum 2 caractères et ne pas être exclusivement constitués de chiffres ou d'underscores.

```
context Process
inv validName('Invalid name: ' + self.name):
    self.name.matches('[A-Za-z_][A-Za-z0-9_]*')

context WorkDefinition, Resource
inv nameMin2Char: self.name.matches('..+')
inv weirdName: not self.name.matches('([0-9]*|_*)')
```

Pour ne pas avoir d'ambiguité dans le modèle, les noms des WorkDefinitions et des Ressources doivent être uniques.

```
context Process
inv uniqNamesWD: self.processElements
    ->select(pe | pe.oclIsKindOf(WorkDefinition))
    ->collect(pe | pe.oclAsType(WorkDefinition))
    ->forAll(w1, w2 | w1 = w2 or w1.name <> w2.name)
inv uniqNamesRes: self.processElements
    ->select(pe | pe.oclIsKindOf(Resource))
    ->collect(pe | pe.oclAsType(Resource))
    ->forAll(r1, r2 | r1 = r2 or r1.name <> r2.name)
```

Nous avons aussi contraint l'utilisateur à utiliser les WorkSequence sur des WorkDefinition appartenant au même Process. Pour éviter des non-sens, les WorkSequence ne peuvent pas non plus avoir le même successeur et prédécesseur.

```
context WorkSequence
inv successorAndPredecessorInSameProcess('Activities not in the same process : '
        + self.predecessor.name + ' in ' + self.predecessor.process().name+ ' and '
        + self.successor.name + ' in ' + self.successor.process().name):
    self.process() = self.successor.process()
    and self.process() = self.predecessor.process()
inv notReflexive: self.predecessor <> self.successor
```

Nous avons aussi ajouté des contraintes sur les quantités des Resource et Request. En effet, cela n'a pas de sens d'avoir des Resources ou des Requests avec des quantités négatives. De plus, une Request ne peut pas être plus grande que le nombre initial de ressources. (Le nombre initial de ressources est le maximum puisqu'il n'y a pas de création.)

```
context Resource, Request
inv negativeQuantity: self.quantity > 0

context Request
inv greedy: self.quantity <= self.target.quantity
```

2.2 petriNet.ocl

Les modèles PetriNet étant relativement similaires aux modèles SimplePDL, nous avons établi des contraintes OCL similaires. Nous obligeons le Network et les Node à avoir des noms uniques mais également sensés.

```
context Network
inv validName('Invalid name: ' + self.name):
    self.name.matches('[A-Za-z_][A-Za-z0-9_]*')
inv uniqNamesNode: self.nodes
    ->forAll(n1, n2 | n1 = n2 or n1.name <> n2.name)

context Node
inv nameMin2Char: self.name.matches('..+')
inv weirdName: not self.name.matches('([0-9]*|_*)')
```

Le nombre de jetons des Place et le poids des Arc doivent évidemment être positifs.

```
context Place
inv negativeQuantity: self.tokens >= 0
context Arc
inv negativeQuantity: self.weight >= 0
```

3 Eclipse Modeling Framework (EMF)

Pour permettre une meilleur intégration de nos métamodèles dans notre environnement de developpement (sous Eclipse), nous pouvons créer des greffons nous permetttant de les intégrer dans d'autres projets, ainsi que des éditeurs arborescents nous permettant de mieux visualiser/éditer des modèles conformes à nos métamodèles Ecore. Le code java de ces éditeurs arborescent est engendré par nos métamodèles Ecore, mais nous pouvons le modifier manuellement pour que ceux-ci conviennent parfaitement à nos critères. Ces plugins seront déployés dans une Eclipse Application séparée de notre environnement de developpement principale pour ne pas mélanger métamodèles et modèles.

3.1 plugin simplePDL

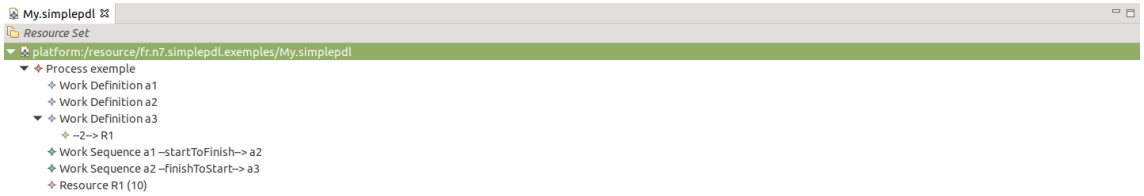

FIGURE 5 – Éditeur arborecent d'un modèle simplePDL

3.2 plugin petriNet

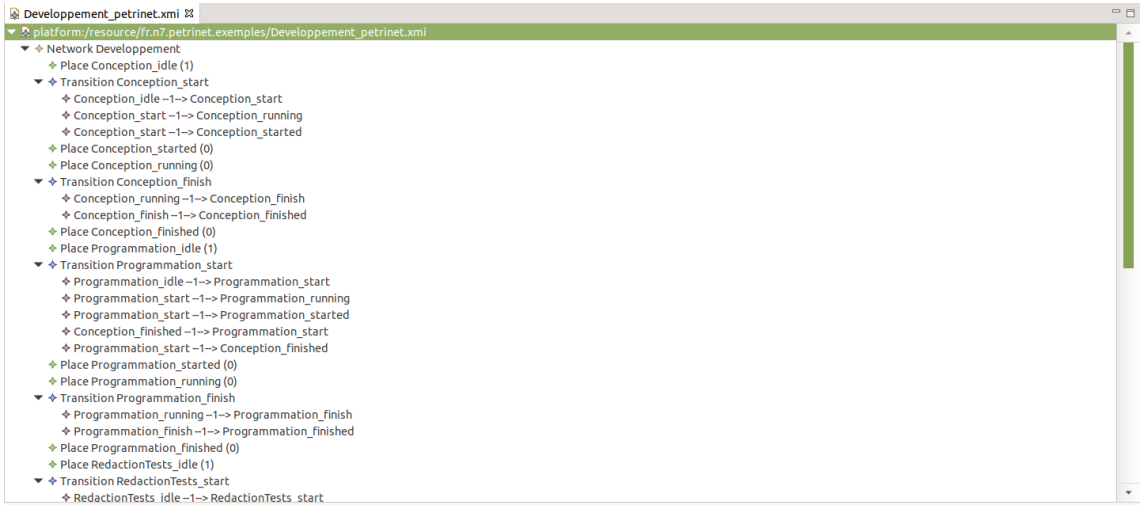

FIGURE 6 – Éditeur arborecent d'un modèle petriNet

3.3 simplePDL → petriNet (avec Java)

Maintenant que nous pouvons charger plus facilement nos métamodèles dans notre environnement de travail, il nous est aussi possible d'importer nos modèles dans un programme Java (grâce aux modules générés automatiquements par EMF). Ainsi en 200 lignes de code, nous pouvons convertir directement un modèles simplePDL en un modèle petriNet.

4 Transformation de modèle à texte (avec Acceleo)

Il nous est possible de transcrire nos modèles vers d'autres formats de fichiers pour nous permettre de les utiliser dans logiciels tierces, et ainsi de les modifier/visualiser plus aisaiment.

4.1 simplePDL → html (toHTML.mtl)

Nous pouvons dans un premier temps de transformer nos modèles simplePDL selon le langage de balisage HTML.

```
<head><title>developpement</title></head>
<body>
    <h1>Process "developpement"</h1>
    <h2>Work definitions</h2>
    <ul>
        <li>Conception</li>
        <li>
            RedactionDoc requires Conception to be finished,
            Conception to be started.
        </li>
        <li>Programmation requires Conception to be finished.</li>
        <li>
            RedactionTests requires Conception to be started,
            Programmation to be finished.
        </li>
    </ul>
</body>
```

4.2 simplePDL → dot (toDOT.mtl)

Nous pouvons de même transformer nos modèles simplePDL selon le langage de description de graphe DOT.

```
digraph "developpement" {
    "Conception" -> "RedactionDoc" [arrowhead=vee label=finishToFinish]
    "Conception" -> "RedactionDoc" [arrowhead=vee label=startToStart]
    "Conception" -> "Programmation" [arrowhead=vee label=finishToStart]
    "Conception" -> "RedactionTests" [arrowhead=vee label=startToStart]
    "Programmation" -> "RedactionTests" [arrowhead=vee label=finishToFinish]
}
```

4.3 petriNet → tina (toTINA.mtl)

Enfin, il nous est possible de transformer nos modèles petriNet selon le language de description de réseau de Petri TINA (format .net).

```
net coolNetwork
pl debut (1)
pl fin (0)
tr debut2fin debut*1 -> fin*1
```

5 Définition de syntaxes concrètes graphiques (avec Sirius)

Tout comme lors de la création d'éditeurs arborescent spécifiques à nos métamodèles (cf EMF), il nous est possible de créer des éditeurs graphiques pour nos métamodèles. Cela nous donne accès à des outils graphiques permettant ainsi à un utilisateur non accoutumé à des outils complexes de créer et modifier des modèles (de processus ou de réseau de petri).

5.1 simplePDL.odesign

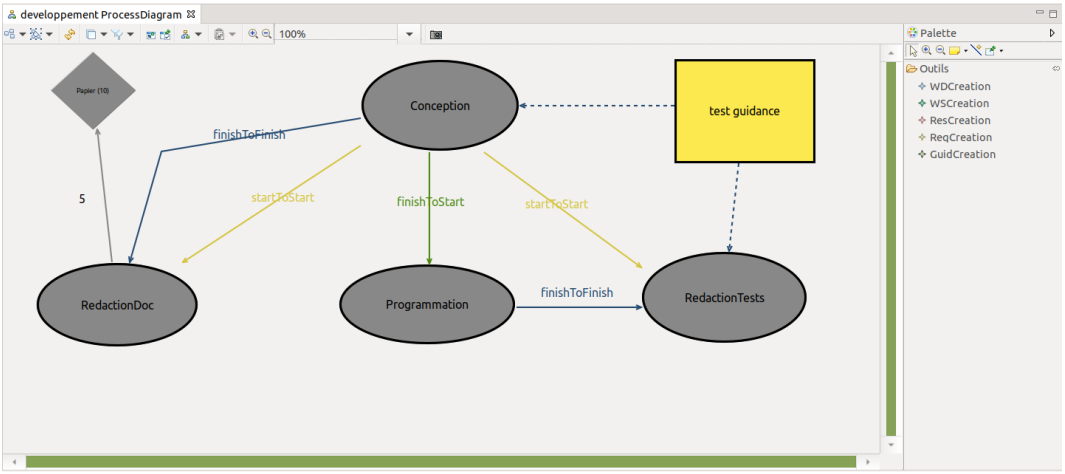

FIGURE 7 – Éditeur graphique d'un modèle simplePDL

5.2 petriNet.odesign

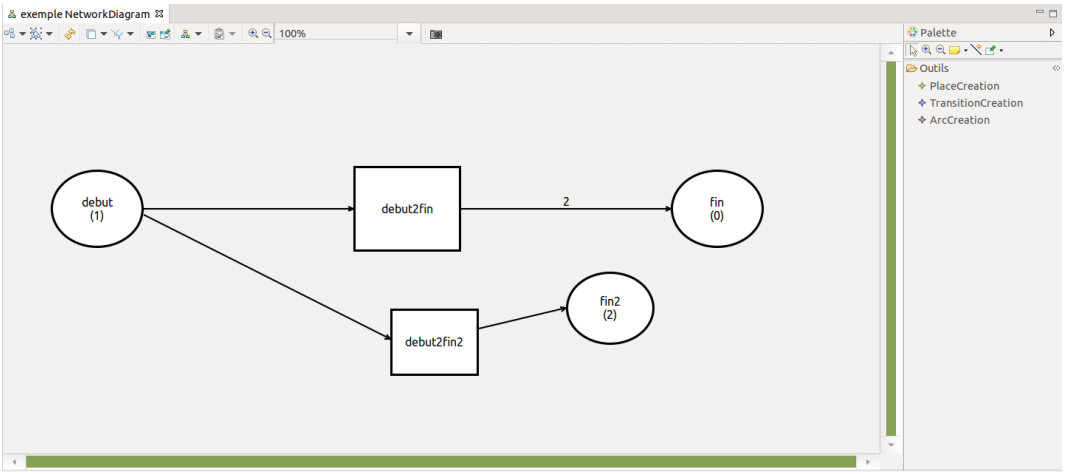

FIGURE 8 – Éditeur graphique d'un modèle petriNet

6 Définition de syntaxes concrètes textuelles (avec Xtext)

Dans la continuité de la création d'outils pour manipuler et visualiser nos modèles, il nous est possible de définir une styntaxe textuelle associée à nos métamodèles.

6.1 simplePDL.xtext

Ainsi pour simplePDL, la syntaxe textuelle suivante permet de manipuler nos modèles facilement, sans passer par des outils graphiques parfois complexes.

```
process Developpement {
    res Crayon 10
    res Papier 20
    wd Conception
        req Crayon 10
        req Papier 5
    wd RedactionTest
    wd RedactionDoc
    wd Programmation
    ws f2s from Conception to Programmation
    ws s2s from Conception to RedactionTest
    ws s2s from Conception to RedactionDoc
    ws f2f from Conception to RedactionDoc
    ws f2f from Programmation to RedactionTest
}
```

7 Transformation de modèle à modèle (avec ATL)

Finalement il nous est possible, tout comme lors de la transformation modèle à modèle via l'écriture d'un programme Java, de transformer un modèle selon un autre métamodèles via l'outil ATL.

7.1 simplePDL → petriNet

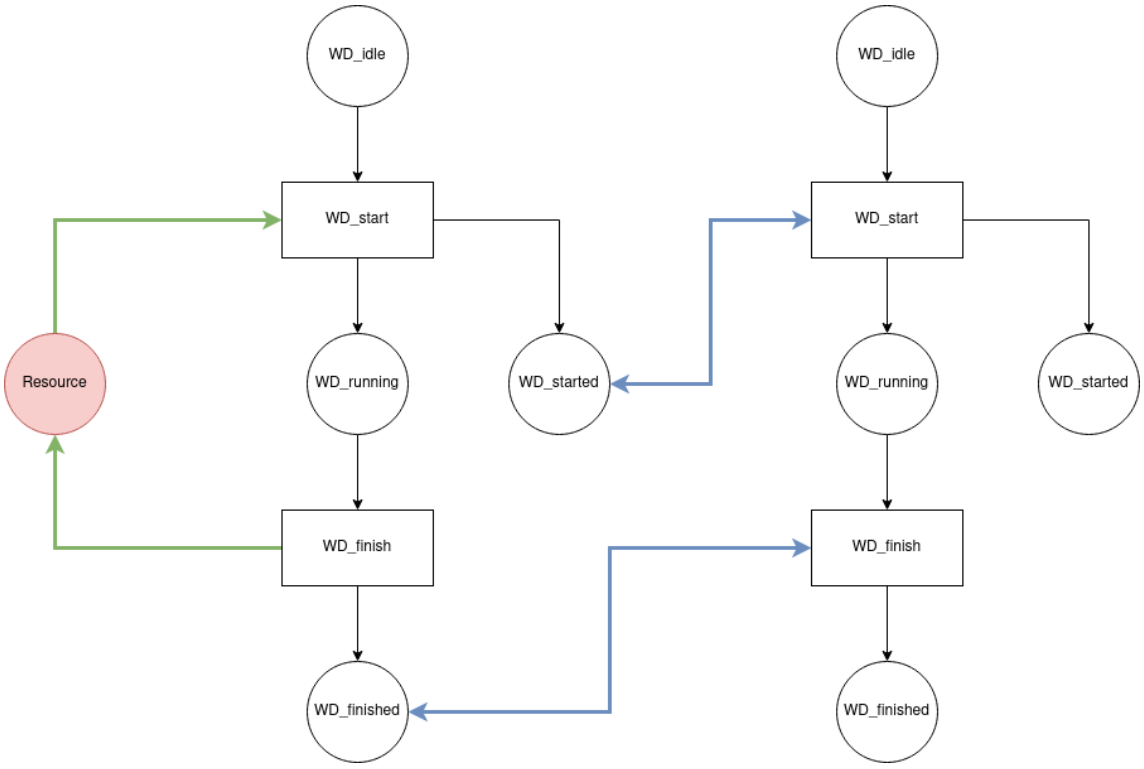

FIGURE 9 – Modèle pour la transformation de modèle à modèle

Pour transformer une WorkDefinition dasn un réseau de Petri, nous créons 4 places (_idle, _running, _started, _finished) ainsi que 2 transitions (_start, _finish). Pour transformer une WorkSequence, nous relions la place du predecesseur à la transition du successeur, par exemple dans le cas d'un linkType start2start nous relions un _started à un _start. Pour transformer une Resource, nous créons simplement une place avec le bon nombre de tokens. Pour ce qui est des Requests d'une WorkDefinition nous relions le _start et le _finish de la WorkDefinition à la ressource avec les poids correspondants.

8 Logique Temporelle Linéaire (LTL)

Dans l'optique d'une vérification de la terminaison et de la transformation correcte de notre modèle simplePDL en modèle petriNET, nous pouvons à partir du modèle simplePDL générer des propositions que nous demanderons par la suite à selte/tina de vérifier.

Pour vérifier la terminaison du processus, nous pouvons écrire :

```
[] <> (WD1_finished /\ WD2_finished /\ WD3_finished);
```

Pour vérifier qu'une WorkDefinition ne soit que dans un seul état à la fois, nous pouvons écrire (via l'intermédiaire d'un xor) :

```
[] (((WD1_idle /\ -WD1_running /\ -WD1_finished)
     \/ (-WD1_idle /\ WD1_running /\ -WD1_finished)
     \/ (-WD1_idle /\ -WD1_running /\ WD1_finished))
     /\ ((WD2_idle /\ -WD2_running /\ -WD2_finished)
     \/ (-WD2_idle /\ WD2_running /\ -WD2_finished)
     \/ (-WD2_idle /\ -WD2_running /\ WD2_finished))
     /\ ((WD3_idle /\ -WD3_running /\ -WD3_finished)
     \/ (-WD3_idle /\ WD3_running /\ -WD3_finished)
     \/ (-WD3_idle /\ -WD3_running /\ WD3_finished)));
```

Pour vérifier qu'une WorkDefinition ne progresse que dans un seul sens, nous pouvons

```
[] ((WD1_finished =>
    [](-WD1_running /\ -WD1_idle /\ WD1_started))
    /\ (WD2_finished => [](-WD2_running /\ -WD2_idle /\ WD2_started))
    /\ (WD3_finished => [](-WD3_running /\ -WD3_idle /\ WD3_started)));
```