Rapport Projet SimplePDL et Réseaux de Pétri

Membres :
Jules Gourio
Felix Foucher de Brandois
27 juin 2025

Table des matières

1 Introduction

Ce projet s'inscrit dans le cadre du module d'Ingénierie Dirigée par les Modèles et vise à développer un écosystème complet pour la définition, la manipulation et la vérification de modèles de procédés. L'objectif principal est de proposer une chaîne d'outils permettant à des utilisateurs, sans distinction de compréhension du domaine informatique, de créer et analyser des modèles de procédés représentant des ensembles de tâches, de ressources et leurs dépendances.

Cette approche repose sur l'utilisation de l'ingénierie dirigée par les modèles pour simplifier le développement et rester au plus près des besoins utilisateur. Le projet couvre la création de méta-modèles, la définition de syntaxes concrètes, la transformation vers les réseaux de Pétri et la vérification de propriétés complexes.

Le présent rapport détaille l'ensemble des réalisations effectuées, les choix de conception adoptés, les difficultés rencontrées et les enseignements tirés de ce projet.

2 Méta-modèles et sémantique statique

2.1 Méta-modèle SimplePDL

2.1.1 Architecture générale

Le méta-modèle SimplePDL respecte les contraintes définies dans le cahier des charges. Il se compose des éléments suivants :

- **Process** : élément racine contenant l'ensemble des éléments du modèle
- **WorkDefinition** : représentation des tâches du procédé
- **WorkSequence** : modélisation des dépendances entre tâches
- **Resource** : représentation des ressources disponibles
- **Need** : liaison entre tâches et ressources requises
- **Guidance** : éléments de documentation et commentaires

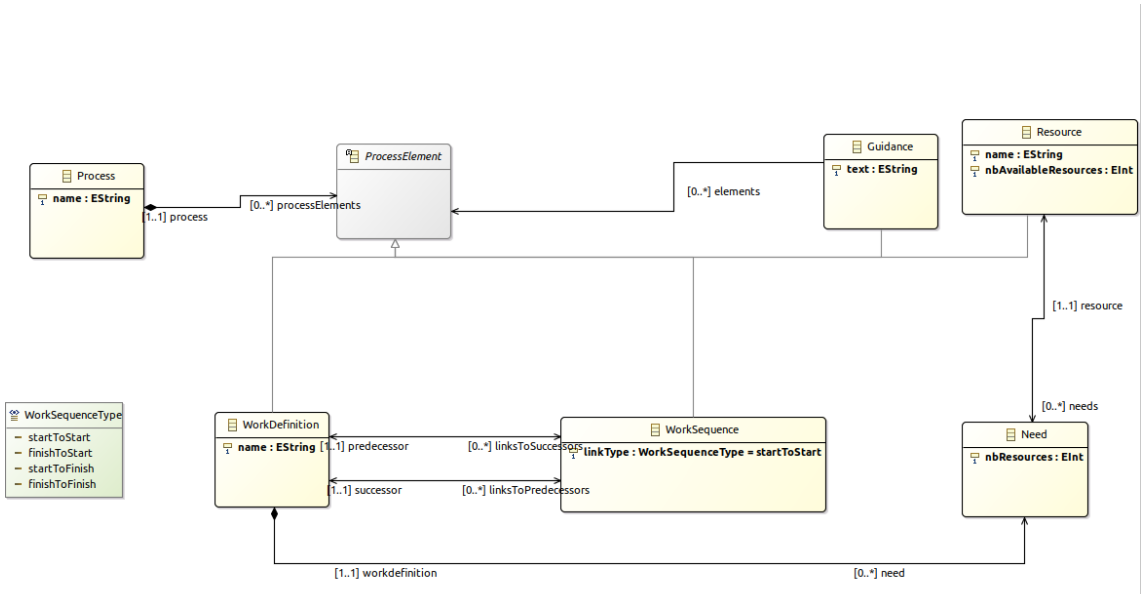

FIGURE 1 – Diagramme Ecore du méta-modèle SimplePDL

2.1.2 Choix de conception pour les ressources

Pour modéliser les ressources, nous avons créé deux classes distinctes qui respectent les contraintes R1.3.1 et R1.3.2 du cahier des charges.

La classe `Resource` hérite de `ProcessElement` et représente une ressource disponible dans le système. Elle contient :

— **name** : nom unique de la ressource
— **nbAvailableResources** : quantité totale disponible (nombre entier positif)

Cette classe centralise donc les informations sur la disponibilité globale d'une ressource.

La classe `Need` représente le lien entre une tâche et une ressource. Elle contient :

— **nbResources** : quantité requise par la tâche (nombre entier positif)
— **resource** : référence vers la ressource concernée
— **workdefinition** : référence vers la tâche qui a besoin de cette ressource

Justification de cette approche :

— Cette séparation en deux classes présente plusieurs avantages :
— Une ressource peut être utilisée par plusieurs tâches avec des quantités différentes
— Chaque tâche peut avoir plusieurs besoins en ressources
— La validation est simplifiée : on peut facilement vérifier que la somme des besoins ne dépasse pas la disponibilité
— Cela respecte la contrainte qui stipule qu'une dépendance ne concerne qu'une seule tâche et une seule ressource
— La référence de composition entre `WorkDefinition` et `Need` garantit que chaque besoin appartient à une seule tâche, maintenant ainsi la cohérence du modèle.

2.2 Contraintes OCL et validation Java

Suite au bug critique dans OCL (mentionné dans le TP 4) , l'implémentation des contraintes s'est faite en Java.

2.2.1 Contraintes implémentées

Les contraintes suivantes ont été développées en Java pour valider les modèles SimplePDL :

— **Validation des noms de processus** : vérification que le nom du processus respecte les conventions Java (lettres, chiffres et underscore uniquement)
— **Unicité des noms d'activités** : contrôle qu'aucune activité dans un même processus n'a le même nom qu'une autre activité
— **Validation des noms d'activités** : vérification que les noms d'activités respectent les conventions Java
— **Prévention des dépendances réflexives** : une WorkSequence ne peut pas relier une activité à elle-même
— **Unicité des dépendances** : interdiction d'avoir plusieurs dépendances identiques (même type, même prédécesseur, même successeur) entre deux activités
— **Validation du contenu des guidances** : vérification que chaque Guidance contient un texte non vide
— **Validation des ressources** : contrôle de la cohérence des quantités de ressources disponibles et requises
— **Validation des besoins** : vérification que chaque Need est correctement lié à une ressource et une activité avec des quantités positives

2.3 Test des contraintes SimplePDL

Pour vérifier l'implémentation de ces contraintes nous avons tester sur deux modèles la validation sur deux modèles simplepdl avec des défaillances et en effet ceux-ci apparaissant bien comme défaillant, l'un qui était construit directement avec le SimplePDLCreator avait des noms qui ne convenait pas aux conventions Java, l'autre avec une Guidance avec un champ textuel vide. Ces deux modèles se trouvent dans fr.n7.simplepdl/models.

2.4 Méta-modèle PetriNet

2.4.1 Architecture du méta-modèle

Le méta-modèle des réseaux de Pétri respecte les contraintes R2.1 à R2.6 :

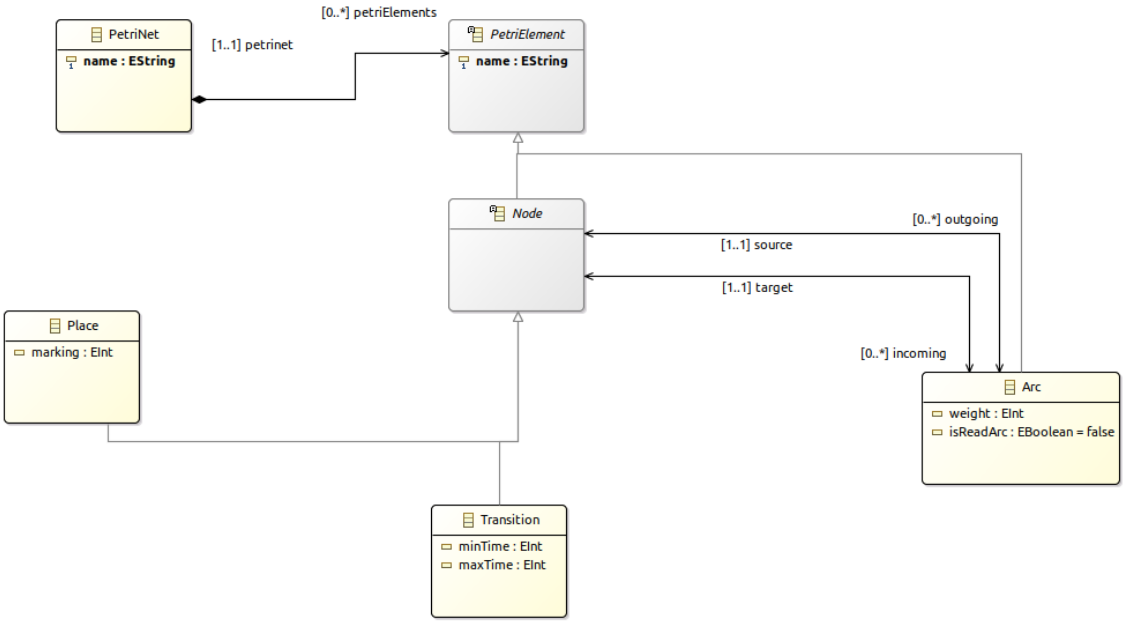

FIGURE 2 – Diagramme Ecore du méta-modèle des réseaux de Pétri

2.4.2 Choix de conception pour la classe PetriNet

La classe `PetriNet` est la classe racine du modèle avec :
— Attribut `name` obligatoire (lowerBound=1) pour identifier le réseau
— Référence de composition `petriElements` vers tous les éléments du réseau (upperBound=-1)
— Relation bidirectionnelle avec `PetriElement` via eOpposite
Cette conception centralise tous les éléments dans un conteneur unique, facilitant la navigation et la validation.

2.4.3 Choix de conception pour les places et transitions

Les classes `Place` et `Transition` héritent de la classe abstraite `Node` qui elle-même hérite de `PetriElement` :

Place :
— Attribut `marking` (type EInt) pour le nombre de jetons
— Pas de lowerBound spécifié, donc marquage peut être 0 par défaut
Transition :
— Attribut `minTime` pour la borne inférieure de l'intervalle temporel
— Attribut `maxTime` pour la borne supérieure (peut être non défini)
— Les deux attributs sont optionnels (pas de lowerBound)
La classe abstraite `Node` factorise les références `incoming` et `outgoing` vers les arcs, évitant la duplication de code.

2.4.4 Choix de conception pour les arcs

La classe `Arc` hérite directement de `PetriElement` et contient :
— **weight** : poids de l'arc (type EInt) respectant la contrainte R2.4.1
— **isReadArc** : booléen pour les arcs en lecture seule (contrainte R2.4.3)
— **source** et **target** : références obligatoires (lowerBound=1) vers les nœuds
— Relations bidirectionnelles avec `Node` via eOpposite pour maintenir la cohérence
Cette modélisation permet de connecter uniquement places et transitions (contrainte R2.4.2) grâce au typage des références vers `Node`.

2.5 Test des contraintes PetriNet

Pour valider l'implémentation des contraintes du méta-modèle PetriNet, nous avons développé une classe `PetriNetValidator` qui vérifie la cohérence structurelle et sémantique des modèles de réseaux de Pétri.

2.5.1 Contraintes implémentées

Les contraintes suivantes ont été développées en Java pour valider les modèles PetriNet :
— **Validation du nom du réseau** : vérification que le réseau de Pétri possède un nom non vide (contrainte R2.5)
— **Validation de la structure** : contrôle que le réseau contient au moins une place et une transition pour être considéré comme valide
— **Validation du marquage des places** : vérification que le marquage (nombre de jetons) est positif ou nul (contrainte R2.2)
— **Validation du poids des arcs** : contrôle que le poids de chaque arc est strictement positif (contrainte R2.4.1)
— **Validation des connexions d'arcs** : vérification qu'un arc relie bien une place à une transition ou vice versa (contrainte R2.4.2)
— **Validation des intervalles temporels** : contrôle de la cohérence des bornes temporelles des transitions, avec vérification que la borne inférieure est positive et que la borne supérieure est supérieure ou égale à la borne inférieure quand elle est définie (contrainte R2.3)

2.5.2 Stratégie de validation

La validation suit une approche récursive similaire à celle utilisée pour SimplePDL :
— Parcours de tous les objets EMF de la ressource

— Application des contraintes spécifiques selon le type d'objet
— Enregistrement des erreurs dans un objet `ValidationResult`
— Validation récursive des éléments contenus

2.5.3 Tests de validation

Pour vérifier l'efficacité de notre système de validation, nous avons créé plusieurs modèles de test, le premier (result_simplepdl2petrinet_echec.xmi) est un modèle avec un marquage négatif (=-1) pour une des places et effectivement le test ne fonctionne pas. Le deuxième qui est un modèle de Petri venant d'une transformation d'un simplepdl passe parfaitement le test (result_simplepdl2petrinet.xmi) tout les deux dans le dossier fr.n7.PetriNet/models .

2.6 Implémentation Java de la transformation

Pour compléter l'approche ATL, nous avons également développé une transformation modèle-à-modèle en Java pur. Cette implémentation offre plus de contrôle sur le processus de transformation et facilite le débogage.

La classe `ProcessToPetriNetTransformer` (dans fr.n7.transformation_simplepdl2petrinet) fonctionne en deux phases :

Phase 1 : Transformation des activités Chaque `WorkDefinition` génère un patron comportemental composé de :

— 4 places représentant les états : `notStarted`, `started`, `running`, `finished`
— 2 transitions contrôlant les changements d'état : `start`, `finish`
— 5 arcs connectant ces éléments selon la sémantique d'exécution

Phase 2 : Transformation des dépendances Chaque `WorkSequence` devient un arc de lecture reliant la place appropriée du prédécesseur à la transition appropriée du successeur, selon le type de dépendance :

— `START_TO_START` : place `started` → transition `start`
— `FINISH_TO_START` : place `finished` → transition `start`
— `START_TO_FINISH` : place `started` → transition `finish`
— `FINISH_TO_FINISH` : place `finished` → transition `finish`

Gestion de la correspondance La classe utilise des `HashMap` pour maintenir la correspondance entre les éléments SimplePDL et les éléments PetriNet créés, permettant de résoudre efficacement les références lors de la transformation des dépendances.

Avantages de cette approche :

— Contrôle fin du processus de transformation avec messages de trace
— Gestion explicite des erreurs et validation en cours de transformation
— Sauvegarde intégrée du modèle résultant au format XMI
— Facilité de débogage et de maintenance

3 Transformation de modèle à Texte

Pour PetriNet ous n'avons pas implémenté les template Acceleo toTina.mtl et toDot.mtl pour pouvoir visualiser de manière textuelle ou de manière graphique les réseaux de Pétri.

subsectionÉditeur graphique avec Sirius

Eclipse Sirius est un framework permettant de créer rapidement des éditeurs graphiques sans programmation complexe. Il fonctionne par séparation entre le modèle sémantique (EMF) et sa représentation graphique, définie de manière déclarative dans un fichier .odesign.

3.0.1 Extension pour les ressources

Pour intégrer les ressources dans notre éditeur SimplePDL existant, nous avons étendu la spécification Sirius avec de nouveaux mappings :

Mapping ResourceNode : Les ressources sont représentées par des ellipses bleu clair avec le nom de la ressource affiché comme label. Ce choix visuel permet de les distinguer clairement des activités (losanges gris) et des guidances (notes jaunes).

Mapping NeedEdge : Les besoins en ressources sont matérialisés par des arcs verts reliant une ressource à une tâche. Le label de l'arc affiche la quantité requise (feature :nbResources), permettant de visualiser directement l'information quantitative.

Outils de création :

Un outil ResourceCreation dans la section "Elements" pour créer de nouvelles ressources

Un outil NeedCreation dans la section "Links" pour établir les liens de dépendance entre ressources et tâches

Configuration des références : Le mapping NeedEdge utilise sourceFinderExpression="feature :resource" et targetFinderExpression="feature :workdefinition" pour résoudre correctement les connexions via l'objet Need intermédiaire.

Cette extension respecte l'architecture existante en ajoutant les ressources comme nouveaux éléments graphiques sans perturber la représentation des activités et dépendances déjà définies. L'utilisateur peut ainsi créer et visualiser des modèles SimplePDL complets incluant la gestion des ressources.

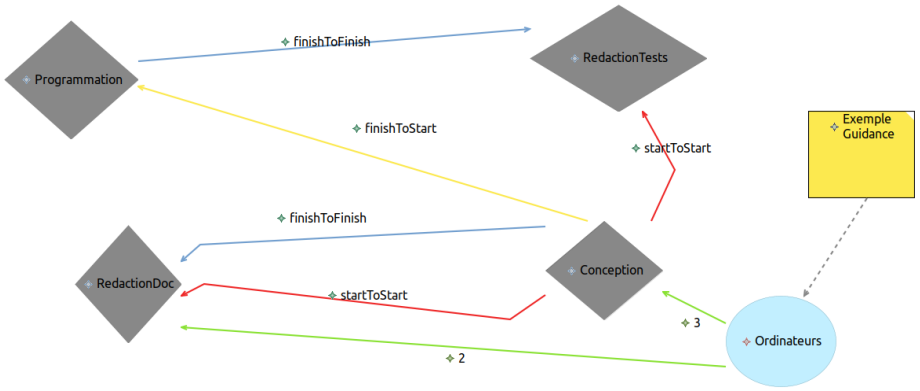

FIGURE 3 – Affichage exemple du developpement.simplepdl avec Sirius

4 Transformation SimplePDL vers TaskMaster avec ATL

Cette transformation convertit un modèle SimplePDL en modèle TaskMaster orienté événements. Elle utilise cinq règles principales pour établir les correspondances entre les deux métamodèles.

4.1 Règles de base

La transformation Process vers EventBundle crée l'élément racine. Chaque WorkDefinition devient deux événements (début et fin) avec une dépendance causale entre eux. Les WorkSequence deviennent des dépendances causales entre les événements appropriés selon leur type.

4.2 Extension pour les ressources

L'ajout des ressources nécessite deux nouvelles règles :

Resource vers Supply : Chaque ressource SimplePDL devient un Supply TaskMaster avec mapping direct du nom et de la quantité disponible.

Need vers Physical : Cette règle constitue l'innovation principale. Chaque Need génère deux Physical requirements : un positif à l'événement de début (acquisition de la ressource) et un négatif à l'événement de fin (libération de la ressource). Cette approche modélise correctement le cycle de vie des ressources pendant l'exécution des tâches.

4.3 Avantages

Cette extension permet une modélisation fine des contraintes d'allocation des ressources tout en préservant la sémantique du modèle SimplePDL original. La transformation ATL garantit la traçabilité et facilite la maintenance grâce à la séparation claire des règles.

5 Transformation SimplePDL vers réseau de Pétri avec ATL

Cette transformation ATL convertit un modèle SimplePDL en réseau de Pétri pour permettre l'analyse de propriétés comme la terminaison et l'atteignabilité. La transformation suit une approche systématique où chaque élément SimplePDL génère des éléments Pétri correspondants.

5.1 Architecture de la transformation

La transformation utilise cinq règles principales qui créent un réseau de Pétri complet à partir du modèle de procédé. Chaque WorkDefinition génère quatre places et deux transitions pour modéliser les états d'exécution (non démarrée, démarrée, en cours, terminée) et les événements de début et fin.

5.2 Modélisation des activités

Chaque activité SimplePDL devient un sous-réseau composé de quatre places représentant les états successifs de l'activité. La place "notStarted" contient initialement un jeton, tandis que les autres places sont vides. Deux transitions "start" et "finish" permettent de faire évoluer l'activité entre ces états.

Les arcs connectent ces éléments selon la sémantique d'exécution : la transition "start" consomme le jeton de "notStarted" et produit des jetons dans "started" et "running". La transition "finish" consomme le jeton de "running" et produit un jeton dans "finished".

5.3 Gestion des dépendances

Les WorkSequence deviennent des arcs de lecture qui contraignent l'ordre d'exécution. Ces arcs connectent les places appropriées selon le type de dépendance : start-to-start relie "started" à la transition "start", finish-to-start relie "finished" à "start", etc. L'utilisation d'arcs de lecture permet de vérifier les conditions sans consommer les jetons.

5.4 Extension pour les ressources

L'intégration des ressources constitue l'extension majeure de cette transformation. Chaque Resource devient une place dont le marquage initial correspond à la quantité disponible. Les Need génèrent deux arcs : un arc "take" de la place ressource vers la transition "start" et un arc "release" de la transition "finish" vers la place ressource.

Cette modélisation respecte parfaitement la sémantique des ressources : elles sont acquises au début de l'activité (consommation de jetons) et libérées à la fin (production de jetons). Le poids des arcs correspond à la quantité de ressource nécessaire.

6 Syntaxes concrètes textuelles avec Xtext

L'écosystème SimplePDL propose trois syntaxes concrètes textuelles développées avec Xtext, chacune offrant une approche différente pour la saisie et la manipulation des modèles de procédés. Ces syntaxes permettent aux utilisateurs de créer des modèles de manière intuitive tout en bénéficiant des fonctionnalités avancées d'Eclipse comme la coloration syntaxique, l'auto-complétion et la validation en temps réel.

6.1 Syntaxe PDL1 : Approche directe et intuitive

La syntaxe PDL1 adopte une approche directe inspirée des langages de programmation modernes. Elle utilise des mots-clés explicites pour chaque type d'élément, rendant la lecture et l'écriture des modèles particulièrement accessibles.

6.1.1 Structure grammaticale

La grammaire PDL1 s'articule autour des éléments suivants :

- **Processus** : délimité par `process nom { ... }` avec une structure de bloc claire
- **Activités** : déclarées avec `wd nom` suivi optionnellement de la clause `uses` pour les ressources
- **Dépendances** : exprimées avec `ws type from source to target`
- **Ressources** : définies avec `resource nom quantity valeur`
- **Guidances** : ajoutées avec `note "texte"`

6.1.2 Exemple de syntaxe PDL1

Listing 1 – Exemple de modèle en syntaxe PDL1

```
process ex_with_resources {
resource cpu quantity 2
resource memory quantity 8
resource disk quantity 100
wd a uses cpu, memory
```

```
wd b uses cpu, disk
wd c uses memory, disk
ws f2s from a to b
ws s2f from b to c
}
```

Cette syntaxe présente l'avantage d'être facilement lisible et de suivre une structure proche du méta-modèle SimplePDL. L'extension pour les ressources a été intégrée naturellement en ajoutant la possibilité d'associer des ressources aux activités via la clause `uses`, permettant de spécifier plusieurs ressources séparées par des virgules.

6.2 Syntaxe PDL2 : Modélisation orientée dépendances

La syntaxe PDL2 adopte une approche radicalement différente en intégrant les dépendances directement dans la définition des activités. Cette approche centre l'information sur chaque activité plutôt que de séparer les éléments.

6.2.1 Caractéristiques principales

Chaque activité peut spécifier ses conditions de démarrage et de fin à l'aide des clauses `starts if` et `finishes if`. Cette approche permet de visualiser immédiatement les contraintes qui s'appliquent à une activité donnée.

6.2.2 Exemple de syntaxe PDL2

Listing 2 – Exemple de modèle en syntaxe PDL2

```
process ex1 {
wd a {
}
wd b {
starts if a started
}
wd c {
starts if b finished
}
}
```

Cette approche présente l'avantage de centraliser l'information au niveau de chaque activité, facilitant la compréhension des contraintes locales. Cependant, elle génère un méta-modèle très différent de SimplePDL original, nécessitant des transformations plus complexes pour maintenir la compatibilité avec l'écosystème existant.

6.3 Syntaxe PDL3 : Structure déclarative par sections

La syntaxe PDL3 organise le modèle en sections distinctes, séparant clairement les différents types d'éléments. Cette approche suit une philosophie de séparation des préoccupations.

6.3.1 Organisation en sections

La syntaxe PDL3 structure le modèle en trois sections principales :
— `workdefinitions` : liste des activités
— `worksequences` : définition des dépendances

— `notes` : commentaires et guidances

6.3.2 Exemple de syntaxe PDL3

Listing 3 – Exemple de modèle en syntaxe PDL3

```
process : ex1
workdefinitions : a; b; c;
worksequences : a s2s b; b f2f c;
```

L'avantage de cette syntaxe réside dans sa structure claire et sa facilité de parsing, bien qu'elle soit moins intuitive pour les utilisateurs non techniques. La séparation en sections facilite également la maintenance et l'évolution des modèles complexes.

7 Transformation PDL1 vers SimplePDL avec ATL

7.1 Architecture générale de la transformation

La transformation PDL12SimplePDL constitue le pont essentiel entre la syntaxe concrète PDL1 et le méta-modèle SimplePDL. Cette transformation ATL utilise un ensemble de règles complémentaires pour assurer une correspondance complète et sémantiquement correcte entre les deux modèles.

L'architecture de la transformation repose sur le principe de correspondance directe pour les éléments simples et sur des mécanismes plus sophistiqués pour gérer les relations complexes, notamment la gestion des ressources qui nécessite la création d'objets intermédiaires.

7.2 Règles de transformation fondamentales

La transformation utilise cinq règles principales qui créent une correspondance complète entre les éléments PDL1 et SimplePDL. La règle de transformation du processus racine préserve le nom et établit la relation de composition avec tous les éléments enfants. La conversion des activités intègre également la gestion des besoins en ressources en créant automatiquement les objets Need correspondants.

La gestion des dépendances utilise un helper de conversion pour mapper correctement les types de dépendances entre les deux méta-modèles, car les énumérations utilisent des conventions de nommage différentes.

7.3 Extension complexe pour la gestion des ressources

L'intégration des ressources constitue l'aspect le plus compliqué de cette transformation, nécessitant la mise en place de trois mécanismes complémentaires qui travaillent en synergie.

7.3.1 Transformation directe et création d'objets intermédiaires

La transformation des ressources s'effectue en deux phases distinctes. D'abord, la conversion simple des objets Resource préserve les informations essentielles tout en adaptant la structure aux conventions du méta-modèle cible. Ensuite, la génération automatique des objets Need pour chaque association entre une activité et une ressource utilise

une condition de filtrage sophistiquée pour s'assurer de l'application aux seules associations valides.

7.3.2 Gestion de la relation many-to-many

La principale difficulté résidait dans la gestion de la relation many-to-many entre activités et ressources. Dans PDL1, cette relation est exprimée par une référence directe, tandis que SimplePDL utilise un objet intermédiaire Need pour modéliser cette association avec des informations quantitatives.

La solution adoptée utilise une approche en deux phases soigneusement orchestrée : d'abord la génération des éléments principaux avec leurs propriétés de base, puis la création des objets Need via une règle conditionnelle qui établit les associations. Cette approche garantit que tous les éléments référencés existent avant la création des liens, évitant ainsi les problèmes de références non résolues.

7.4 Validation de la transformation

La transformation a été validée sur plusieurs modèles de tests avec succès en effet si nous reprenons le modèle avec ressources du PDL1 défini ci dessus et qu'on le transforme on obtiens ceci :

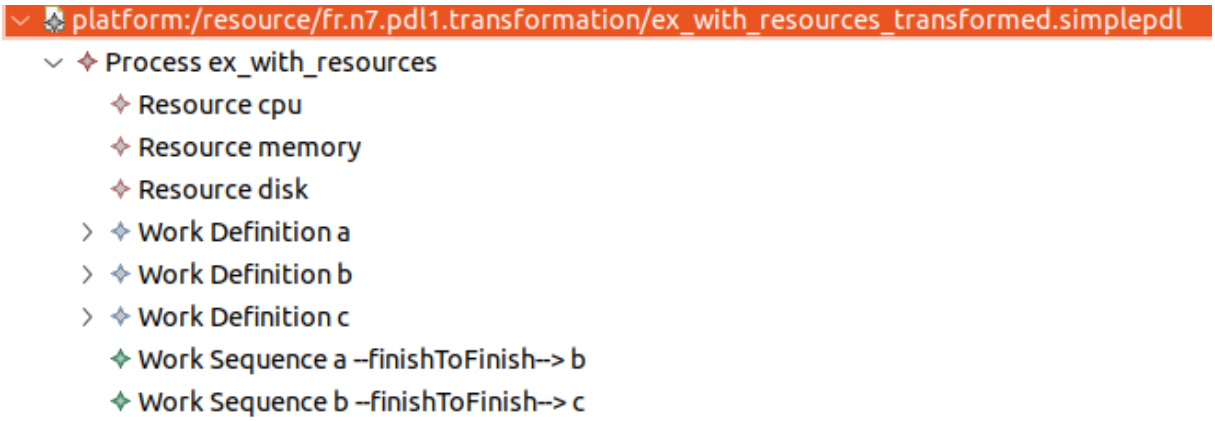

FIGURE 4 – Modèle PDL1 transformé en SimplePDL

Ce modèle est bien un modèle simplepdl conforme en effet on peut le valider et il passe les tests du validator. La transformation et le modèle se trouvent dans le dossier fr.n7.pdl1.transformation

8 Liste détaillée des fichiers livrés

Cette section présente l'ensemble des fichiers constituant le livrable, organisés selon les exigences L1 à L12 du cahier des charges :

8.1 Méta-modèles (L1-L2)

— **L1 - Le méta-modèle SimplePDL** :
`fr.n7.simplepdl/SimplePDL.ecore`

— **L2 - Le méta-modèle de réseau de Pétri** :
`fr.n7.PetriNet/PetriNet.ecore`

8.2 Contraintes et validation (L3-L4)

— **L3 - Le programme de validation des contraintes du méta-modèle SimplePDL** :
`fr.n7.simplepdl/src/simplepdl.validation/SimplePDLValidator.java`
— **L4 - Le programme de validation des contraintes du méta-modèle de réseau de Pétri** :
`fr.n7.petrinet/src/petrinet.validation/PetriNetValidator.java`

8.3 Exemples (L5)

— **L5 - Des exemples de modèles de procédés et de réseaux de Pétri** :
  — `fr.n7.simplepdl/models/developpement.simplepdl`
  — `fr.n7.simplepdl/models/process1-ko.xmi`
  — `fr.n7.petrinet/models/result_simplepdl2petrinet.xmi`
  — `fr.n7.petrinet/models/result_simplepdl2petrinet_echec.xmi`
  — `et bien d'autres dans chaque dossier où il y a une transformation à faire`

8.4 Syntaxes concrètes (L6-L8)

— **L6 - La définition de la syntaxe graphique Sirius** :
`fr.n7.simplepdl.design/description/simplepdl.odesign`
— **L7 - La définition de la syntaxe textuelle** :
`fr.n7.pdl1/src/fr/n7/PDL1.xtext`
— **L8 - L'éditeur arborescent** :
  — `fr.n7.simplepdl/` (projet principal avec méta-modèle)
  — `fr.n7.simplepdl.edit/` (projet edit)
  — `fr.n7.simplepdl.editor/` (projet editor)
  — `fr.n7.PetriNet/` (projet principal avec méta-modèle)
  — `fr.n7.PetriNet.edit/` (projet edit)
  — `fr.n7.PetriNet.editor/` (projet editor)

8.5 Transformations (L9-L12)

— **L9 - Les transformations modèle-à-modèle de SimplePDL vers réseau de Pétri** :
  — `fr.n7.simplepdl2petrinet_ATL/SimplePDL2PetriNet.atl`
  — `fr.n7.transformation_simplepdl2petrinet/src/ProcessToPetriNetTransformer.java`
— **L10 - La transformation modèle-à-texte de modèle de réseau de Pétri vers la syntaxe tina** :
`Non réalisé`
— **L11 - Les transformations modèle-à-texte de modèle de procédé vers propriétés LTL** :
  — `fr.n7.simplepdl.toHTML/src/fr/n7/simplepdl/toHTML/main/toHTML.mtl`

— **L12 - La transformation modèle-à-texte de modèle de procédé vers Dot** :

— `fr.n7.simplepdl.todot/src/fr/n7/simplepdl/todot/main/toDot.mtl`

8.6 Transformations supplémentaires

— **Transformation PDL1 vers SimplePDL** :
`fr.n7.pdl1.transformation/transformations/PDL12SimplePDL.atl`
— **Transformation SimplePDL vers TaskMaster** :
`fr.n7.simplepdl2taskmaster/transformations/SimplePDL2TaskMaster.atl`

9 Conclusion

Ce projet a permis de développer un écosystème complet pour la manipulation et la vérification de modèles de procédés, en utilisant les technologies de l'ingénierie dirigée par les modèles.