Rapport Mini-Projet

N7 2SN

Génie du Logiciel et des Systèmes

Groupe L34-14

MEMBRES:

MOUFE DEFFO Ingrid Darline
MORIGINE Annamaria

TABLE DE MATIÈRE

TABLE DE FIGURES

1. Introduction

1.1. Contexte de l'IDM (Ingénierie Dirigée par les Modèles)

L'**Ingénierie Dirigée par les Modèles**, également connue sous le nom de *Model-Driven Engineering* (MDE), est une **approche de développement** qui vise à placer le **modèle au centre du processus**. L'objectif principal de l'IDM est de **favoriser l'interopérabilité par les modèles**.

Les modèles sont utilisés principalement pour **mieux comprendre** un système et les **systèmes complexes**. Un modèle est une simplification et une **abstraction** du système réel. Selon Minsky, un modèle M d'un objet O aide un observateur A à répondre aux questions qu'il se pose sur O.

Ceci sera exploité pour bâtir une **chaîne de vérification** complète de modèles de processus (SimplePDL) en définissant les **méta-modèles** (Ecore) et les **contraintes statiques** (OCL/Java), puis en réalisant une **traduction automatique de modèle à modèle (M2M, via ATL)** de SimplePDL vers des **réseaux de Petri** pour pouvoir ensuite générer, via **transformation modèle à texte (M2T, avec Acceleo)**, la syntaxe requise par l'outil Tina afin de vérifier la cohérence et la terminaison des processus…

1.2. Objectifs du Mini-Projet

Ce projet vise à **développer une chaîne de validation** des **modèles de processus** créés avec SimplePDL et **vérifier la cohérence** des processus.

En utilisant les **outils** explorés lors des travaux pratiques, nous chercherons à déterminer si les processus peuvent parvenir à un terme, en nous appuyant principalement sur la boîte à outils **Tina**.
Afin d'optimiser l'utilisation de Tina, il a été nécessaire de convertir un modèle SimplePDL en un **réseau de Petri** (PetriNet).
Cette conversion a été réalisée grâce à un ensemble d'outils de modélisation intégrés dans l'environnement Eclipse.

Notre approche repose sur plusieurs technologies complémentaires : l'infrastructure **EMF** et le méta-modèle **Ecore** pour la modélisation de base, **Xtext** pour la syntaxe textuelle, **Sirius** pour la représentation graphique, **Java** pour la gestion des contraintes, ainsi que **ATL** et **Acceleo** pour les transformations de modèles.

1.3. Outils d'Implémentation IDM:

En résumé, nous avons utilisé:

- **Ecore/EMF** (pour la syntaxe abstraite)
- **OCL/Java** (pour la sémantique statique)
- **Xtext (**pour les syntaxes concrètes textuelle)
- **Sirius** (pour les syntaxes concrètes graphique)
- **ATL** (pour la transformation M2M)
- **Acceleo** (pour la transformation M2T)

2. Définition des Méta-modèles

2.1 SimplePDL

SimplePDL est conçu pour la modélisation simplifiée des processus de développement qui respecte la spécification Ecore. Il repose sur un métamodèle décrivant les **activités** (WorkDefinition) et leurs enchaînements (WorkSequence). La réalisation d'une activité est conditionnée par l'accès à des **ressources** : une activité alloue des **occurrences** de ces ressources au moment où elle commence, et ces occurrences lui restent affectées de manière exclusive pendant toute son exécution, jusqu'à sa complétion.
Dans le cadre de l'évolution de notre projet, nous avons intégré la gestion des ressources au processus. Cette implémentation s'articule autour de deux composants principaux :

1. Une Eclass “Ressource” qui englobe : la définition de la ressource, un compteur indiquant la quantité disponible “number”

2. Un gestionnaire d'allocation “RessourceRequirement” qui détermine : le volume de ressources requis pour chaque tâche et l'association entre les ressources et les activités

Dans notre modèle, nous avons établi que :

- Les ressources font partie intégrante du flux de processus
- Une activité peut fonctionner avec un nombre variable de gestionnaires d'allocation (de zéro à plusieurs)
- Chaque activité peut nécessiter différents types et quantités de ressources

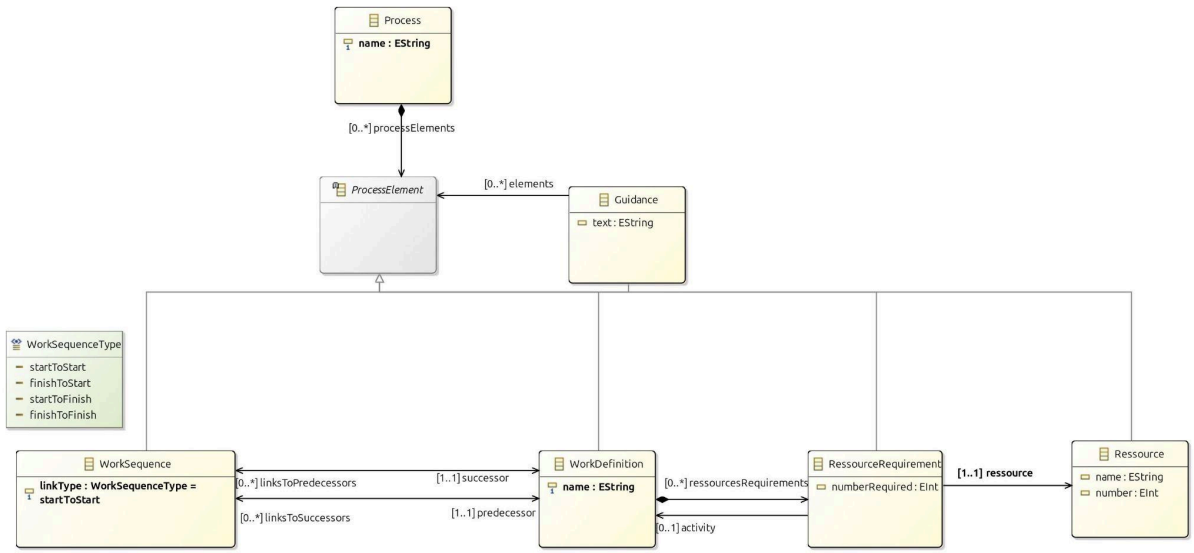

Figure 1 : Métamodèle SimplePDL avec ressources

Un modèle nommé “Developpement” nous a été fourni dans le cadre du mini-projet et nous l’avons compléte en utilisant notre métamodèle afin de rajouter les ressources:

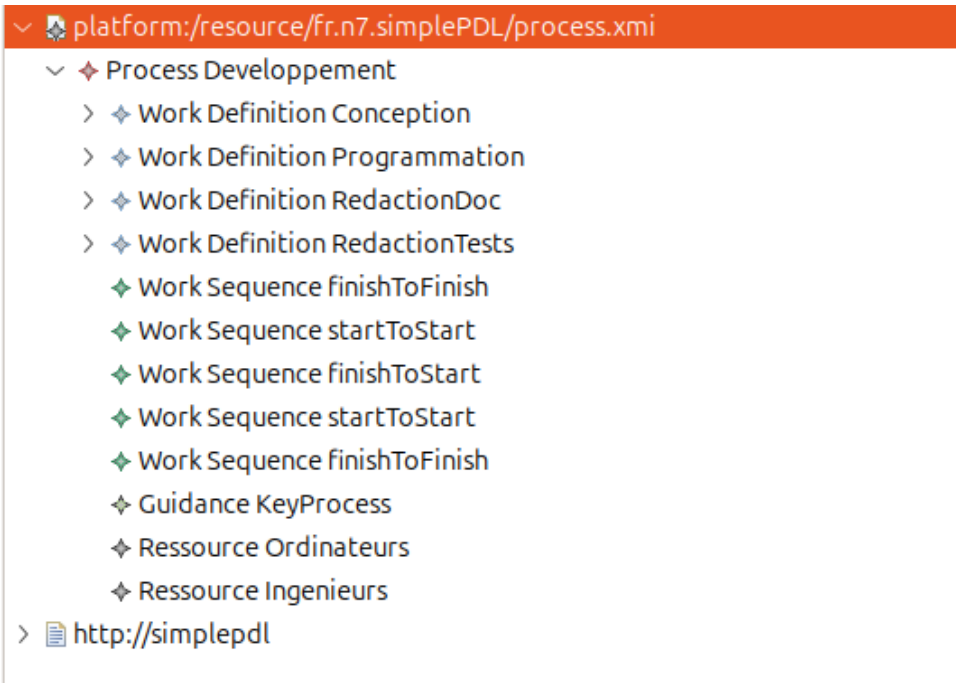

Figure 2 : Modèle de processus process.xmi

2.2 PetriNet

Le méta-modèle Ecore pour les Réseaux de Petri permet de formaliser et de structurer les éléments fondamentaux de ce type de réseau dans un cadre orienté objet.

Il repose sur une modélisation explicite des principaux concepts :

- les **places**, qui représentent des états ou des ressources ;
- les **transitions**, qui modélisent les événements ou actions provoquant des changements d'état ;
- les **arcs**, qui relient les places aux transitions (ou inversement) et définissent le flux entre ces éléments ;
- et enfin, le **marquage initial**, qui spécifie la distribution de jetons dans les places au début de la simulation.

Ce méta-modèle constitue une base rigoureuse pour la définition, la manipulation et l'analyse formelle des Réseaux de Petri dans des outils de modélisation basés sur EMF (Eclipse Modeling Framework).

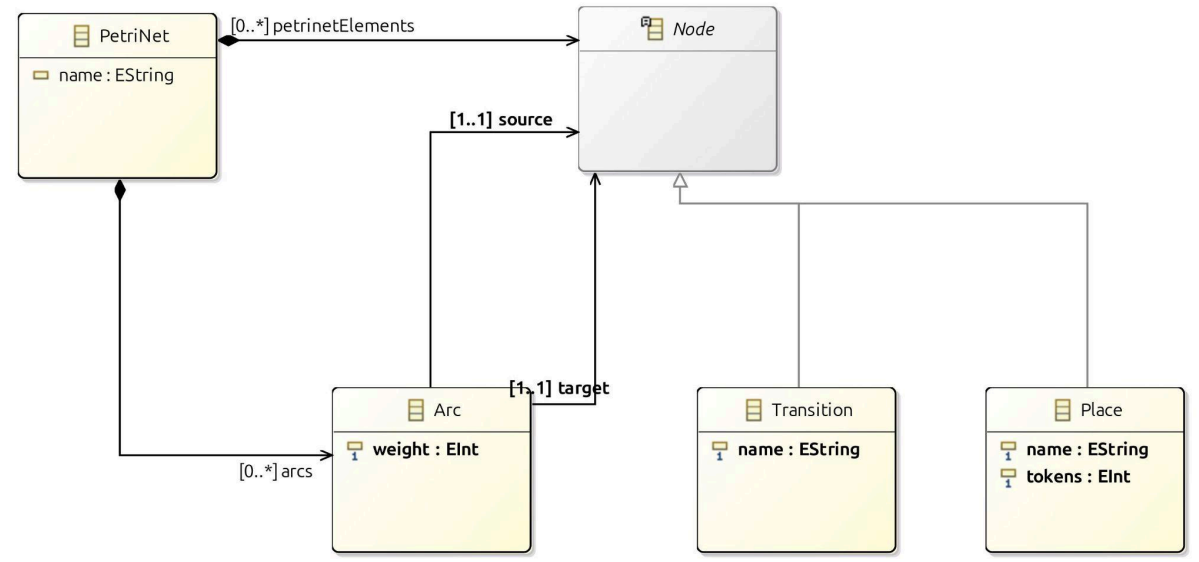

Figure 3 : Métamodèle PetriNet

3 Développement du Langage

3.1. Sémantique Statique (Contraintes Java)

L'objectif est d'utiliser le langage Java pour imposer des limites strictes (des contraintes) sur les modèles de processus, car le langage de méta-modélisation Ecore ne suffit pas. En créant ces règles du jeu, nous définissons précisément ce qu'un modèle peut ou ne peut pas faire. Ce faisant, nous simplifions grandement l'audit du modèle, nous facilitons le suivi des modifications et nous garantissons une construction du système plus fiable et mieux structurée.

3.1.1. Contraintes sur le modèle SimplePDL

Les règles développées pour SimplePDL sont:

- **Les noms (processus, activités et ressources) doivent respecter les conventions Java:**

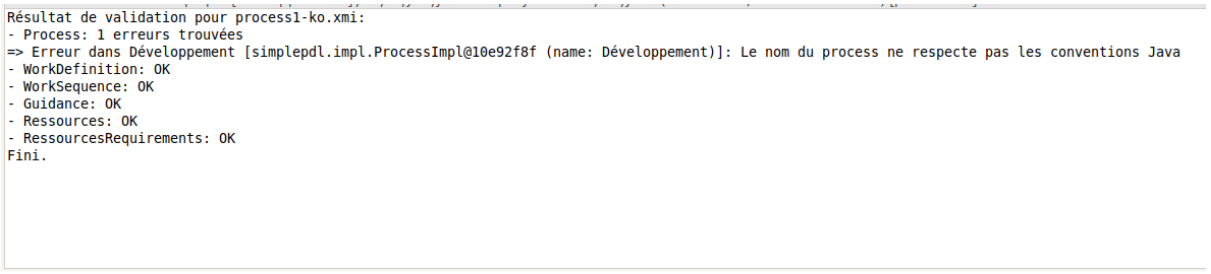

Figure 4 : résultat du test de validation du modèle ko-pdl.xmi

- **Les ressources, les activités doivent avoir des noms différents. Ces noms doivent être uniques:**

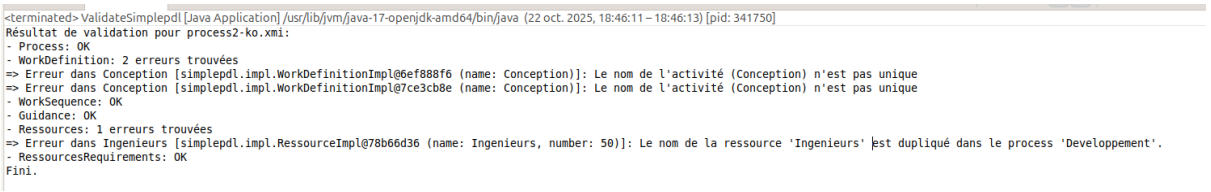

Figure 5 : résultat du test de validation du modèle ko-pdl-2.xmi

- Une dépendance ne doit pas relier une activité à elle-même (une activité ne doit pas dépendre d'elle même) :

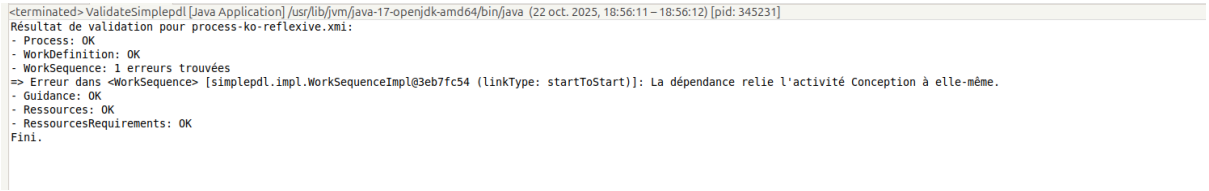

Figure 6 : résultat du test de validation du modèle ko-pdl-reflexive.xmi

- Deux dépendances différentes de même type entre deux mêmes activités ne peuvent coexister :

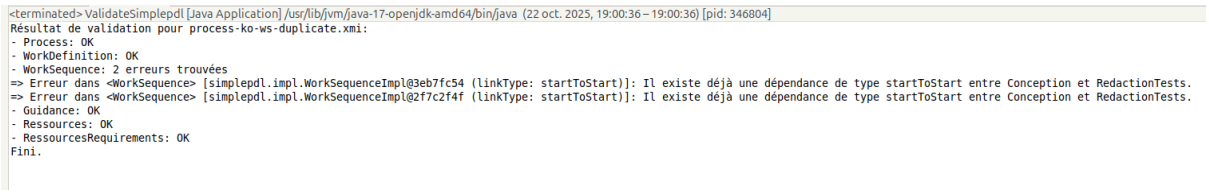

Figure 7 : résultat du test de validation du modèle ko-pdl-ws-duplicate.xmi

- La quantité en besoin d'une ressource et le nombre d'occurence d'une ressource doivent être des entiers naturels. La quantité en besoin ne peut pas être nulle et ne doit pas excéder le nombre de ressources disponibles:

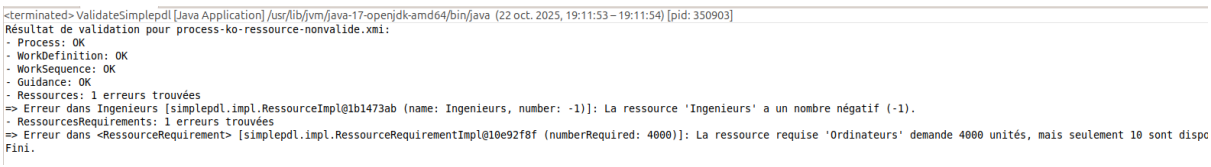

Figure 8 : résultat du test de validation du modèle ko-pdl-ressource-nonvalide.xmi

- Une guidance ne doit pas avoir un texte vide :

Figure 9 : résultat du test de validation du modèle ko-pdl-guidance-nonconforme.xmi

3.1.2. Contraintes sur le modèle PetriNet

Pour les modèles Petrinet, les règles que nous avons définies sont:

- Un arc ne doit pas relier une transition à une transition ou une place à une place:

```
<terminated> ValidatePetriNet [Java Application] /usr/lib/jvm/java-17-openjdk-amd64/bin/java  (22 oct. 2025, 19:25:21 – 19:25:22) [pid: 355690]

Résultat de validation pour : model/petrinet-ko-violation-arc.xmi
- PetriNet: OK
- Place: OK
- Transition: OK
- Arc: 1 erreur(s) trouvée(s)
=> Erreur dans petriNet1.impl.ArcImpl@659a969b (weight: 2): Un arc ne peut pas relier deux éléments du même type (Place→Place ou Transition→Transition).

Validation terminée.
```

Figure 10 : résultat du test de validation du modèle ko-net-violation-arc.xmi

- **Le nombre de jeton d'une place est un entier positif (Toute place doit avoir un nombre entier naturel de jetons):**

```
<terminated> ValidatePetriNet [Java Application] /usr/lib/jvm/java-17-openjdk-amd64/bin/java  (22 oct. 2025, 19:29:41 – 19:29:42) [pid: 357226]

Résultat de validation pour : model/petrinet-ko-violate-token.xmi
- PetriNet: OK
- Place: 1 erreur(s) trouvée(s)
=> Erreur dans Place1 [petriNet1.impl.PlaceImpl@49c386c8 (name: Place1, tokens: -1)]: Le nombre de jetons pour la place 'Place1' ne peut pas être nul ou négatif.
- Transition: OK
- Arc: OK

Validation terminée.
```

Figure 11 : résultat du test de validation du modèle ko-net-violate-token.xmi

- Le nom d'une place ou d'une transition doit être unique:

```
<terminated> ValidatePetriNet [Java Application] /usr/lib/jvm/java-17-openjdk-amd64/bin/java  (22 oct. 2025, 19:38:00 – 19:38:00) [pid: 360169]

Résultat de validation pour : model/petrinet-ko-duplicate.xmi
- PetriNet: OK
- Place: 1 erreur(s) trouvée(s)
=> Erreur dans Place1 [petriNet1.impl.PlaceImpl@6e0dec4a (name: Place1, tokens: 1)]: Le nom de la place 'Place1' est dupliqué.
- Transition: 1 erreur(s) trouvée(s)
=> Erreur dans Transition1 [petriNet1.impl.TransitionImpl@3cc2931c (name: Transition1)]: Le nom de la transition 'Transition1' est dupliqué.
- Arc: OK

Validation terminée.
```

Figure 12 : résultat du test de validation du modèle ko-net-duplicate.xmi

- Le poids d’un arc doit être supérieure ou égal à 1:

```
<terminated> ValidatePetriNet [Java Application] /usr/lib/jvm/java-17-openjdk-amd64/bin/java  (22 oct. 2025, 19:43:17 – 19:43:19) [pid: 362077]

Résultat de validation pour : model/petrinet-ko-violate-poids.xmi
- PetriNet: OK
- Place: OK
- Transition: OK
- Arc: 1 erreur(s) trouvée(s)
=> Erreur dans petriNet1.impl.ArcImpl@659a969b (weight: -1): Le poids de l'arc doit être défini et ≥ 1.

Validation terminée.
```

Figure 13 : résultat du test de validation du modèle ko-net-violate-poids.xmi

3.2. Syntaxe Concrète Textuelle (Xtext)

Nous utiliserons maintenant Xtext pour la définition des syntaxes concrètes textuelles pour SimplePDL (PDL1): c'est structuré autour de mots-clés clairs et de références d'identifiants.

Manipuler directement des modèles complexes avec les éditeurs arborescents d'EMF ou via des classes Java est fastidieux. **Xtext** résout ce problème. Faisant partie du TMF, Xtext permet aux développeurs de **définir une grammaire textuelle personnalisée** pour leurs **DSML** (via un fichier *.xtext*). Cela aboutit à la **génération automatique d'un éditeur Eclipse performant**, qui inclut des aides à la saisie comme la **complétion** et la **détection d'erreurs**, rendant l'**édition et la validation des modèles** non seulement plus simple mais aussi plus naturelle.

3.2.1. Syntaxe de la grammaire

Le principal défi pour créer une syntaxe d'un métamodèle est d'éviter les ambiguïtés. Dans **SimplePDL**, les classes se référencent entre elles, rendant impossible une syntaxe totalement conforme. Par exemple, une **WorkDefinition** dépend des **WorkSequence**, qui dépendent elles-mêmes des **WorkDefinition**.
Pour éviter ce cercle, on a choisi de **supprimer certains attributs**,notamment ceux de **WorkDefinition** et RessourceRequirement de **Ressource**, afin de simplifier la grammaire. Ainsi, la syntaxe finale permet de définir les éléments sans préciser tous leurs attributs.

```
process ex1 {
wd a
wd b
wd c
ws s2s from a to b
ws f2f from b to c

res CPU : nb 4
res RAM : nb 8

req a needs 2 of CPU
req b needs 1 of RAM

note 'GuidanceA'

}
```

Figure 14 : Exemple d'utilisation de la syntaxe

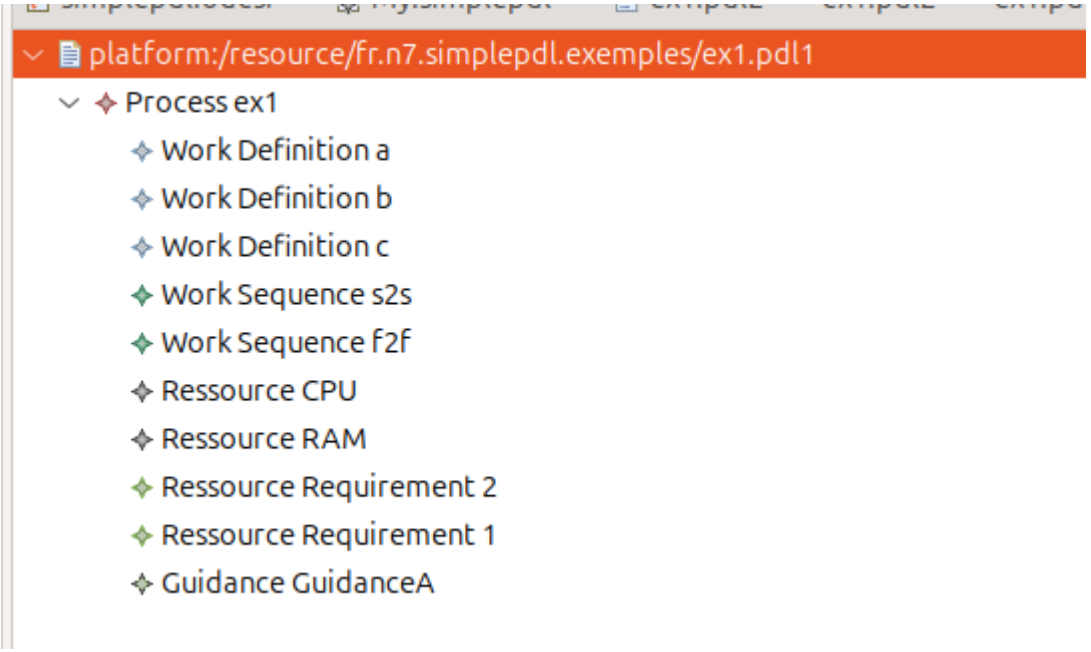

Figure 15 : Modèle ex1.pl1 vu par l'éditeur arborescent

3.3. Syntaxe Concrète Graphique (Sirius)

L'établissement de syntaxes concrètes est nécessaire pour assurer l'exploitabilité du langage métier. Bien qu'une syntaxe textuelle soit utile, la définition d'une syntaxe concrète graphique est cruciale pour mieux comprendre les systèmes complexes et faciliter la communication et la construction des modèles de processus.

L'outil Sirius est employé dans ce mini-projet pour réaliser cette interface visuelle, parce que il permet de visualiser et éditer plus agréablement et efficacement un modèle. Son objectif principal est de présenter graphiquement et d'éditer graphiquement le modèle SimplePDL.

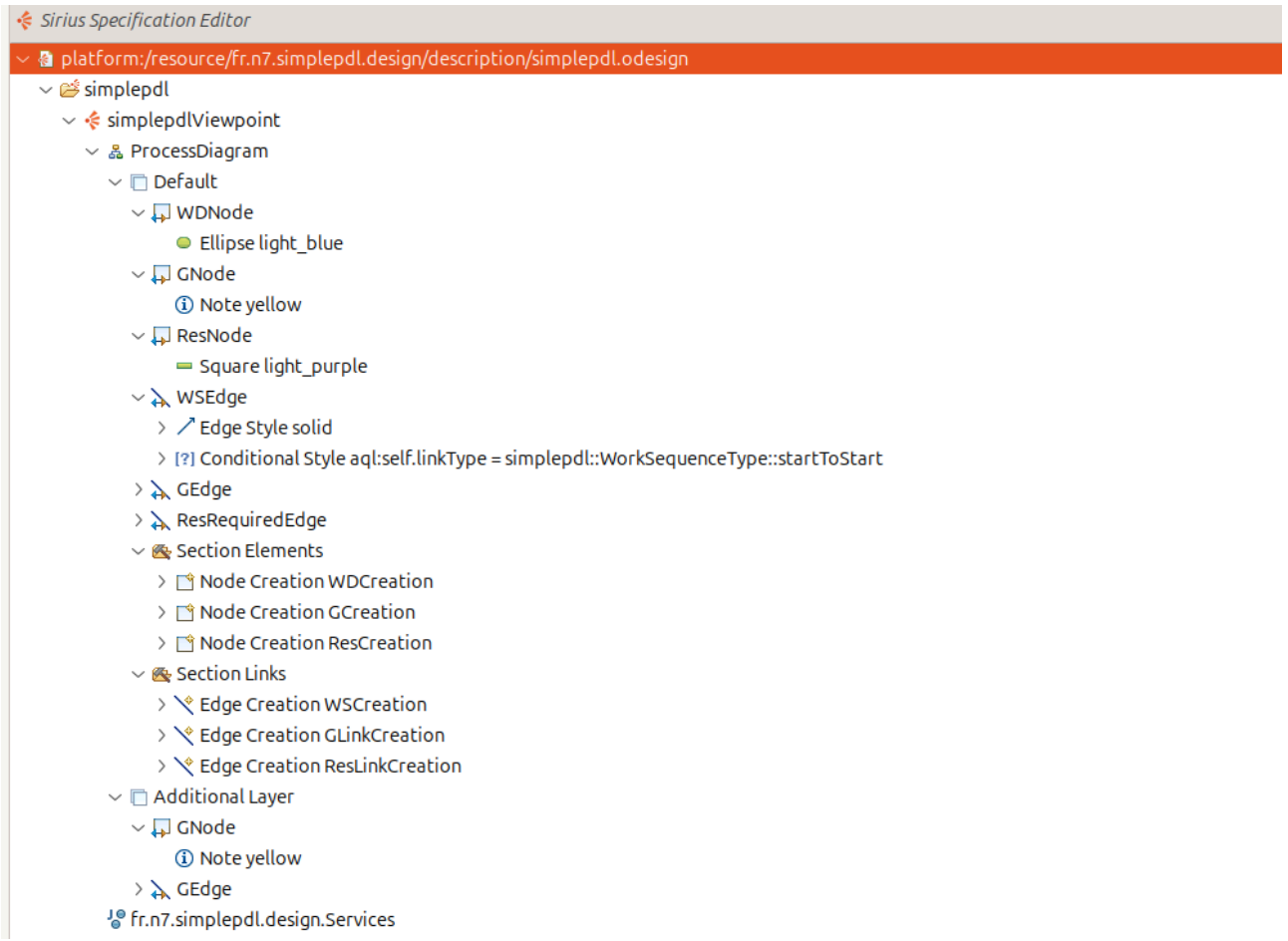

Figure 16 : interface arborescente de Simplepdl avec Sirius

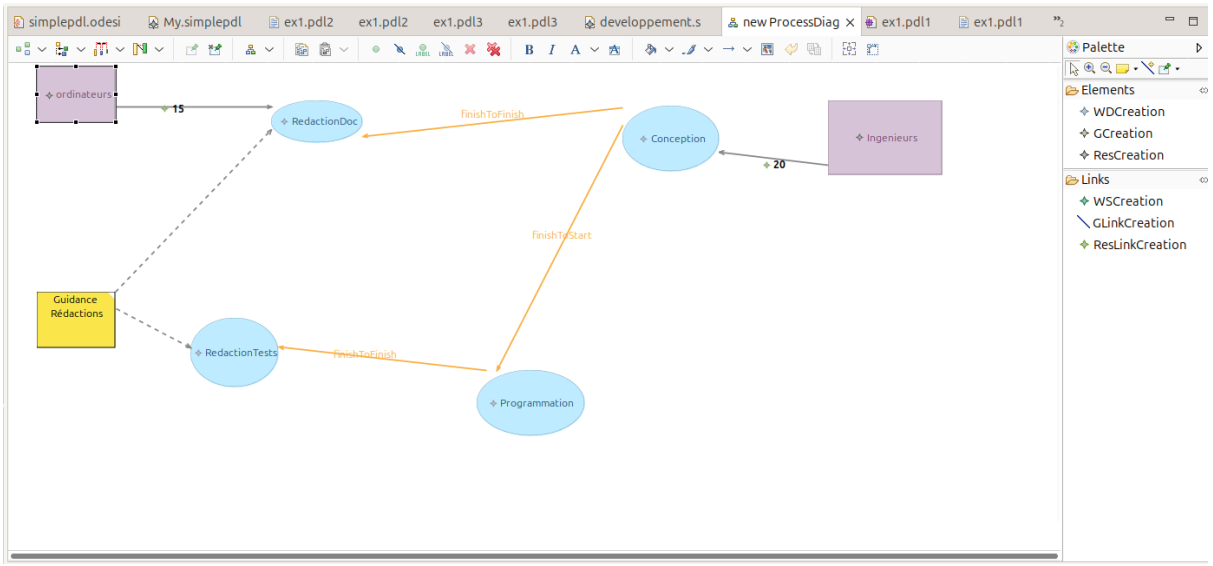

Figure 17 : interface graphique Simplepdl créée avec Sirius

Nous avons décidé de représenter :

- Les activités (WorkDefinition) par une ellipse de fond bleu clair (noeud),
- Les dépendances (WorkSequence) par un arc orienté entre deux activités (lien).
- Les aides (Guidance) par une note avec un fond jaune, attachée aux éléments qu'elle décrit par une flèche en traits interrompus à l'instar des annotations en UML
- Les ressources (Ressource) par un carré violet et les ressources en besoin sont représentées par une flèche entre l'activité concernée et la ressource avec la quantité requise sur la flèche.

Vous pouvez voir à droite les outils pour ajouter de nouveaux éléments. Ces outils ont été mis en œuvre pour ajouter d'autres définitions de WorkDefinition, WorkSequence, Guidance, Ressource et ressource en besoin.

4. La Chaîne de Transformation de Modèles

4.1. Transformation M2M : SimplePDL à PetriNet:

4.1.1 Implémentation EMF/Java

La transformation d'un modèle **SimplePDL** vers un modèle de **Réseau de Pétri** s'effectue automatiquement à l'aide du code Java que nous avons développé dans la classe `SimplePDLToPetriNet`. Cette transformation repose sur l'infrastructure EMF, permettant de charger les métamodèles SimplePDL et PetriNet dans le registre d'Eclipse avant de parcourir les éléments du processus.

L'objectif est de représenter les dépendances entre activités du processus SimplePDL sous la forme d'un réseau de Pétri, où les **WorkDefinition** sont traduites en **places** et **transitions**, et les **WorkSequence** en **arcs** entre ces éléments.

Chaque activité (WorkDefinition) du modèle SimplePDL est transformée en deux places et une transition :

- Une place **`name_ready`**, initialisée avec un jeton, représente l'activité prête à être exécutée.
- Une place **`name_finished`**, sans jeton initial, indique la fin de l'exécution de l'activité.
- Une transition **`name_do`** modélise l'action de réaliser cette activité.

Les arcs suivants sont ensuite créés :

- Un **arc entrant** reliant la place *ready* à la transition *do*, traduisant le déclenchement de l'activité.
- Un **arc sortant** reliant la transition *do* à la place *finished*, indiquant que l'activité est terminée.

Cette structure permet de modéliser simplement le passage de l'état « prêt » à l'état « terminé » pour chaque activité.

Les **WorkSequence** définissent les contraintes d'ordre entre activités (par exemple, « A doit se terminer avant que B ne commence »).

Dans notre transformation, chaque WorkSequence est traduite en un **arc de dépendance** entre une place du prédécesseur et une transition du successeur.
Selon le type de lien (LinkType), la correspondance est la suivante :

- **FINISH_TO_START** : arc entre `pred_finished` → `succ_do`
- **FINISH_TO_FINISH** : arc entre `pred_finished` → `succ_do`
- **START_TO_START** : arc entre `pred_ready` → `succ_do`
- **START_TO_FINISH** : arc entre `pred_ready` → `succ_do`

Ainsi, la dépendance logique du processus est conservée dans le réseau de Pétri grâce à ces arcs de synchronisation.

L'ensemble de la transformation est implémenté dans la méthode `transform(Process process)`, qui crée un objet `PetriNet` contenant tous les éléments du réseau résultant.

Pour chaque élément du processus SimplePDL, les structures correspondantes sont générées et ajoutées au modèle PetriNet, garantissant une correspondance systématique entre les deux représentations.

Cette approche fournit une transformation claire et directe : à partir d'un modèle SimplePDL (.xmi), on obtient automatiquement un modèle PetriNet conforme, facilitant l'analyse comportementale du processus par simulation ou vérification formelle.

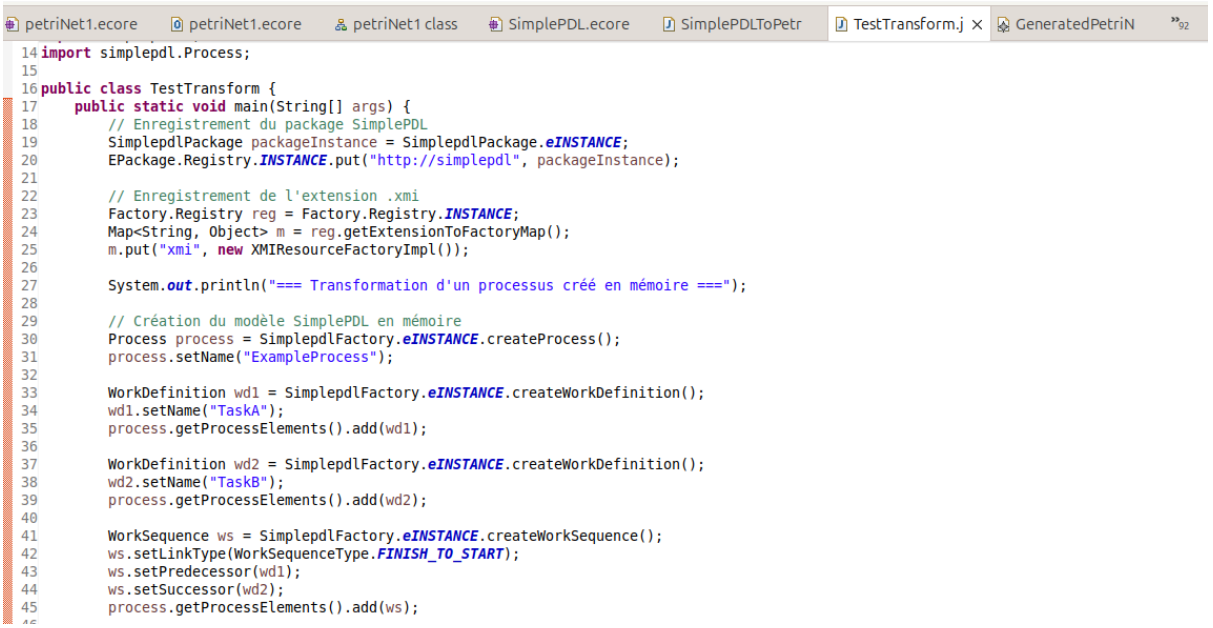

Figure 18 : Création d'un processus en mémoire dans le fichier TestTransform.java

Dans ce même fichier(TestTransform.java) on teste la transformation sur le modèle SimplePDL nouvellement créé.

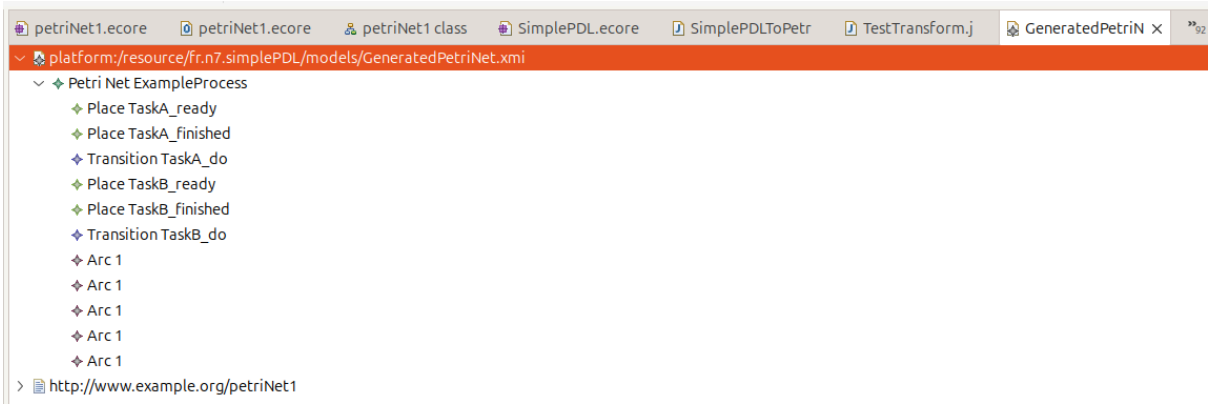

Figure 19 : GeneratedPetrinet.xmi (résultat de la transformation)

4.1.2 ATL

ATL (ATL Transformation Language) est un outil dédié à la **transformation de modèles**, définissant des règles pour convertir un modèle source en un modèle cible, tous deux conformes à leurs métamodèles.

Nous avons transposé en ATL la stratégie de transformation SimplePDL→PetriNet que nous avions initialement mise en œuvre en Java. Cette nouvelle implémentation ATL permet d'effectuer la conversion de

manière déclarative. L'exécution de la transformation est simple : il suffit de configurer le module ATL en lui fournissant le chemin du modèle SimplePDL à transformer et le chemin où le modèle PetriNet résultant doit être généré. Les règles de transformation sont dans le fichier toPetrinet.atl .

La transformation ATL `toPetrinet` permet de convertir automatiquement un modèle **SimplePDL** en un modèle de **Réseau de Pétri** défini dans le métamodèle *petriNet1*. Elle repose sur plusieurs règles qui traduisent les concepts du processus SimplePDL en éléments équivalents du réseau de Pétri.

La règle `Process2PetriNet` crée le réseau global correspondant à un processus SimplePDL, regroupant tous les nœuds et arcs générés. La règle `WD2PetriElements` transforme chaque **WorkDefinition** en un ensemble de places (`_ready`, `_ongoing`, `_finished`) et de transitions (`_start`, `_finish`) reliées par des arcs, représentant les différentes étapes d'exécution d'une activité.

Les **WorkSequence** sont traduites par la règle `WS2Arc`, qui crée des arcs de dépendance entre les transitions et places des activités selon le type de lien (start-to-start, finish-to-start, etc.), afin de préserver les contraintes d'ordre entre tâches.

Les **Ressources** sont quant à elles transformées en **places** avec un nombre initial de jetons égal à la quantité disponible (`Ressource2Place`), tandis que les **RessourceRequirement** deviennent des arcs reliant la ressource à la transition de démarrage de l'activité (`RessourceRequirement2Arc`), avec un poids correspondant au nombre de ressources nécessaires.

Ainsi, l'ensemble de ces règles permet de construire automatiquement un modèle de réseau de Pétri complet à partir d'un modèle SimplePDL, en conservant la logique de dépendances et de gestion des ressources du processus d'origine.

4.2. Transformation M2T

Tout comme il est possible de réaliser des transformations modèle à modèle (M2M) en utilisant des outils comme ATL ou Java, nous pouvons effectuer des transformations modèle à texte (M2T). Pour cela, nous utilisons Acceleo, qui s'appuie sur des gabarits (templates) pour générer les fichiers de sortie souhaités.

4.2.1 PetriNet à Tina

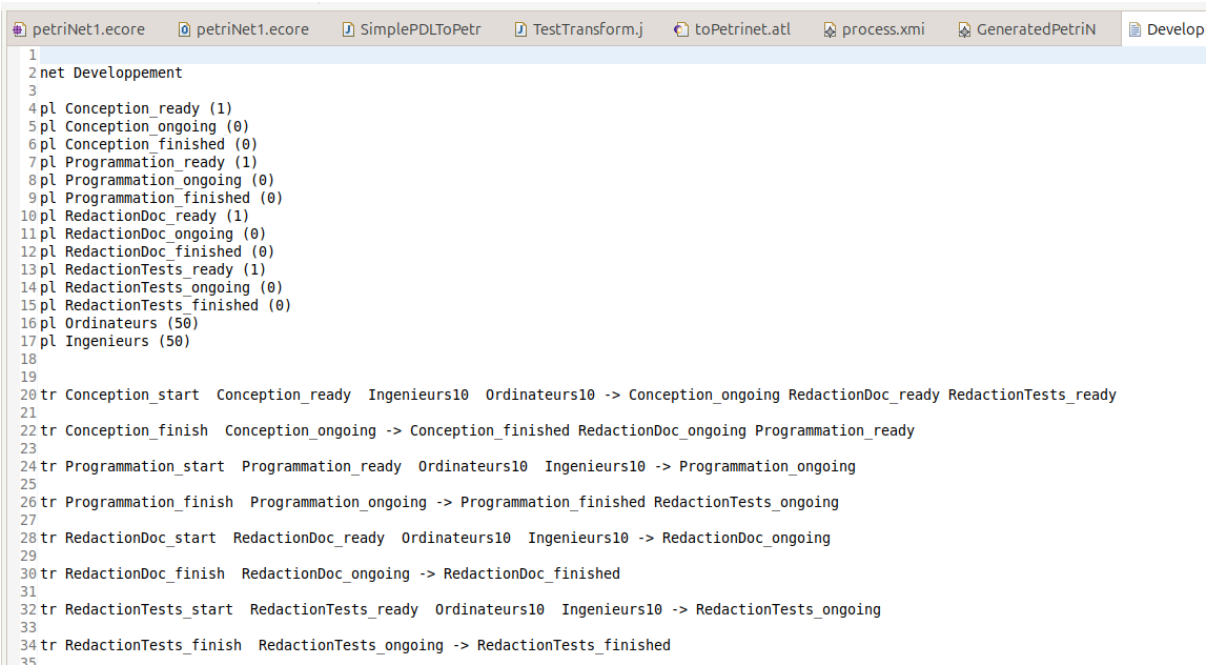

Figure 20 : Résultat de la transformation du modèle ProcessDeveloppementPetrinet.xmi en pdl-sujet.net

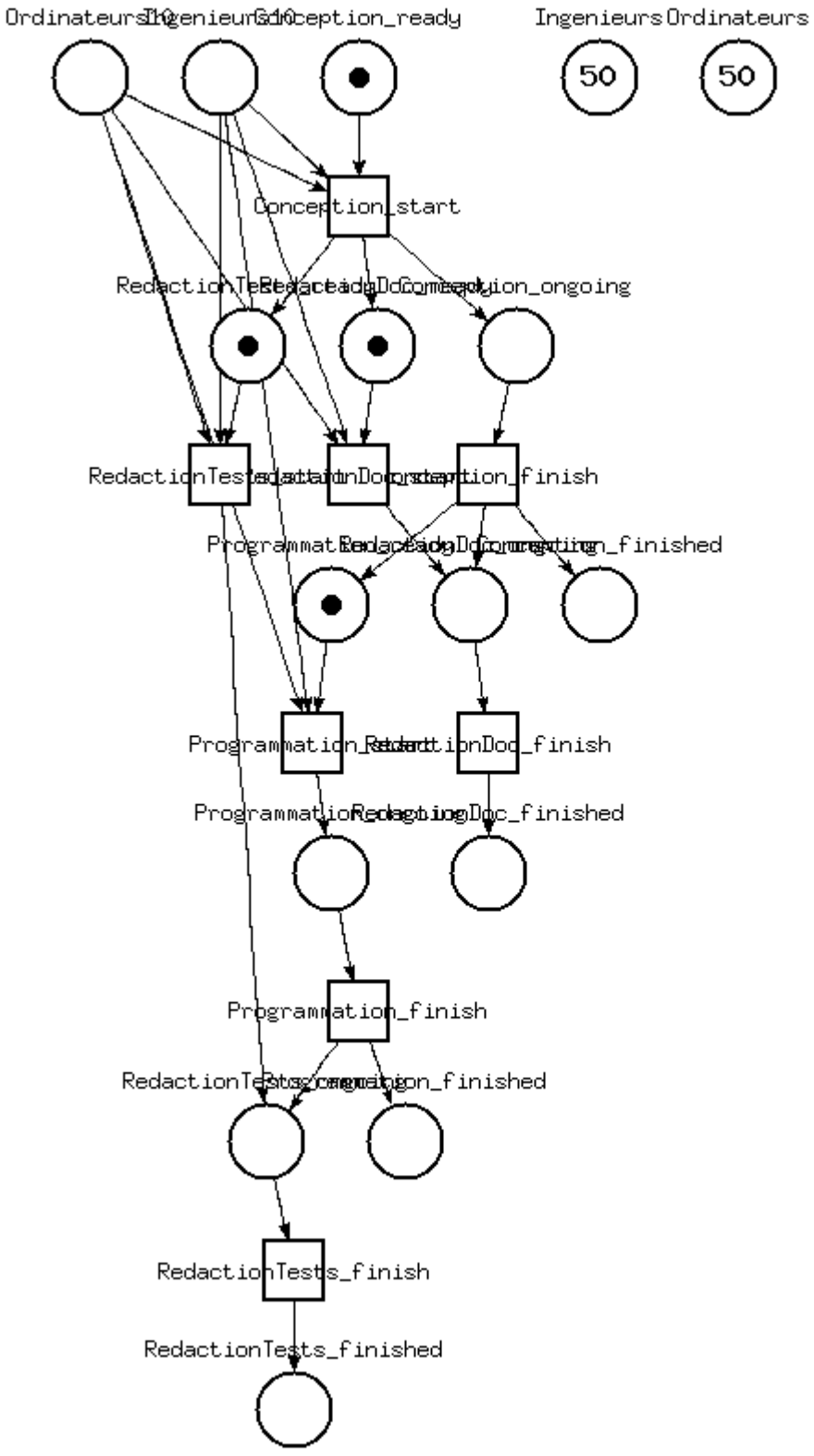

Figure 21 : Visualisation graphique sur Tina de pdl-sujet.net

4.2.2 SimplePDL en ltl

La Logique Temporelle Linéaire (LTL) est employée ici pour l'étape de vérification des propriétés sur le modèle cible, qui est un Réseau de Petri. Les formules LTL permettent de spécifier des contraintes de sécurité (les invariants) et des contraintes de vivacité (comme la terminaison du processus). Ces formules utilisent des variables qui font directement référence aux places du Réseau de Petri.

En s'inspirant de notre implémentation toTina précédente, nous avons créé deux projets de transformation Modèle à Texte (M2T) sous Acceleo. Ces projets traitent un modèle SimplePDL d'entrée et génèrent deux fichiers d'extension .ltl, ces fichiers encapsulant les propriétés de terminaison et les invariants que nous avons définis pour le PetriNet engendré.

- **Terminaison:** Le module parcourt les **WorkDefinition** du processus afin de définir trois opérateurs logiques principaux : `ready`, `ongoing` et `finished`, correspondant respectivement aux états possibles des activités. Ensuite, plusieurs formules LTL sont produites pour exprimer des contraintes de cohérence et de terminaison du processus. Par exemple, la propriété `[] (finished => dead)` signifie que lorsque toutes les activités sont terminées, le processus est dans un état mort (aucune action possible). La formule inverse `[] (dead => finished)` garantit que tout état mort correspond bien à une fin complète du processus. La propriété `<> finished` exprime que la terminaison du processus est **éventuellement atteignable**, tandis que `[] <> dead` vérifie que l'état mort reste **atteignable à tout moment** du déroulement. Ces formules assurent ainsi la **validité, la complétude et la vivacité** du modèle généré, permettant une vérification formelle du comportement du processus. On peut donc vérifier avec l'outil selt (vu en Tp1), sur pdl-sujet.net :

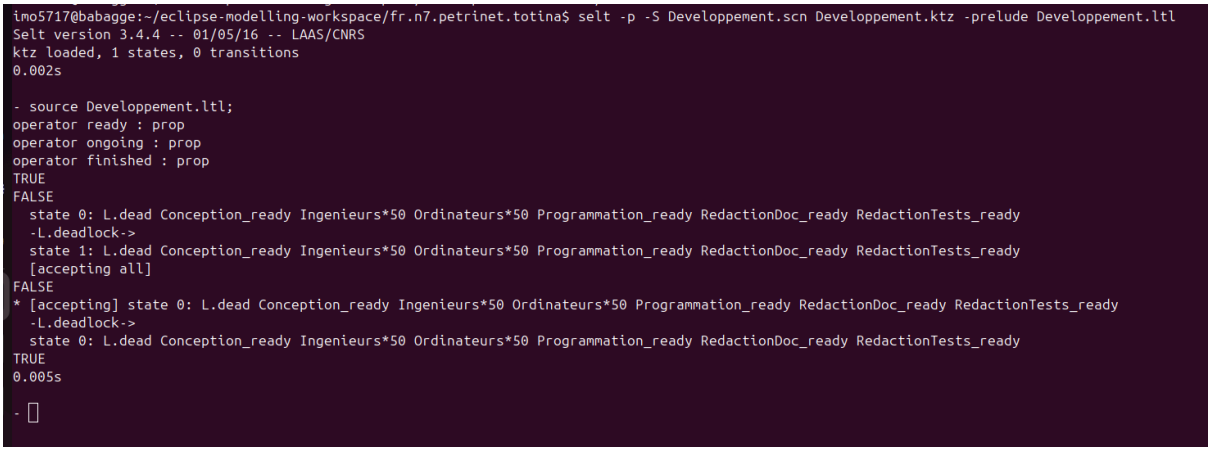

Figure 22 : Vérification de la terminaison du réseau avec selt

- **Invariants:** La transformation génère automatiquement des invariants LTL destinés à vérifier la cohérence interne des activités d'un processus SimplePDL. Pour chaque WorkDefinition, plusieurs propriétés sont créées : la première garantit qu'une activité ne peut être qu'à un seul état à la fois (`ready`, `ongoing` ou `finished`), la seconde et la troisième assurent que les états `ongoing` et `finished` sont permanents une fois atteints, et la dernière vérifie la relation de dépendance entre les états `finished` et `ongoing`. Ces invariants permettent ainsi de contrôler la consistance et la stabilité du comportement du modèle tout au long de son exécution. On peut donc vérifier avec l'outil selt (vu en Tp1), sur pdl-sujet.net :

Figure 23 : Vérification des invariants avec selt

5. Transformation ATL de PDL1 vers SimplePDL

Le **métamodèle PDL1** obtenu à partir de la grammaire définie précédemment présente de légères **divergences** avec le **métamodèle SimplePDL** que nous avions établi initialement. Plus précisément, nous avons dû **retirer certains attributs** du métamodèle PDL1 afin d'assurer l'absence d'ambiguïté syntaxique.

L'utilisation de la nouvelle syntaxe requiert donc une **transformation du modèle PDL1 vers le modèle SimplePDL**. Cette opération est implémentée via une **transformation ATL**, simplifiée par la **proximité structurelle** des deux métamodèles. Le code source de cette transformation est accessible dans le dépôt GitLab.

---

6. Conclusion

En définitive, ce mini-projet illustre la capacité et la puissance de l'IDM, par l'union structurée de la Métamodélisation et des Transformations, à automatiser le passage d'une spécification de haut niveau (DSL) à une vérification formelle rigoureuse.
La chaîne est complète et offre une méthode pour garantir la cohérence des processus modélisés, validant ainsi l'efficacité de l'approche IDM pour l'ingénierie des systèmes complexes.
La réalisation de ce mini-projet a été une expérience particulièrement enrichissante et a permis de consolider une compréhension pratique des défis et des avantages de cette approche.