Chaîne de vérification de modèles de processus

Imad Abakarim
Souhail Amghar

Département Sciences Numériques - Deuxième année
Novembre 2020

Table des matières

Table des figures

1 Introduction

L'objectif de mini-projet est de produire une chaîne de vérification des modèles de processus SimplePDL afin de vérifier leur cohérence. Pour cela, nous utilisons les outils de *model-checking* définis sur les réseaux Petri en utilisant la boîte à outil Tina. Il nous faudra donc traduire un modèle de processus en un réseau de Petri. Le travail se fera suivant les étapes suivantes :

1. Définition des méta-modèles avec Ecore.
2. Définition de la sémantique statique avec OCL (Complete OCL).
3. Utilisation de l'infrastructure fournie par EMF pour manipuler les modèles.
4. Définition de transformations modèle à texte (M2T) avec Acceleo, par exemple pour engendrer la syntaxe attendue par Tina à partir d'un modèle de réseau de Petri ou engendrer les propriétés LTL à partir d'un modèle de processus.
5. Définition d'une transformation de modèle à modèle (M2M) avec EMF/Java et avec ATL.
6. Définition de syntaxes concrètes textuelles avec Xtext.
7. Définition de syntaxes concrètes graphiques avec Sirius.

2 Description des modèles de processus

Dans la figure 1, on donne un exemple de processus qui comprend quatre activités : Conception, RédactionDoc, Programmation et RédactionTests, et quatre dépendances : startToStart, startToFinish, finishToStart et finishToFinish. Les ellipses représentent les activités, est arcs représentent les dépendances.

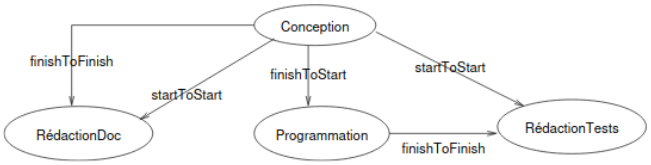

FIGURE 1 – Exemple d'un modèle de processus

3 Définition des méta-modèles avec Ecore

3.1 Définition des méta-modèles avec Ecore pour SimplePDL

On définit le modèle SimplePDL avec quatre activités (Conception, RédactionDoc, Programmation et RédactionTests), quatre dépendances (startToStart,

startToFinish, finishToStart et finishToFinish), les ressources, leurs paramètres et guidance (commentaires).

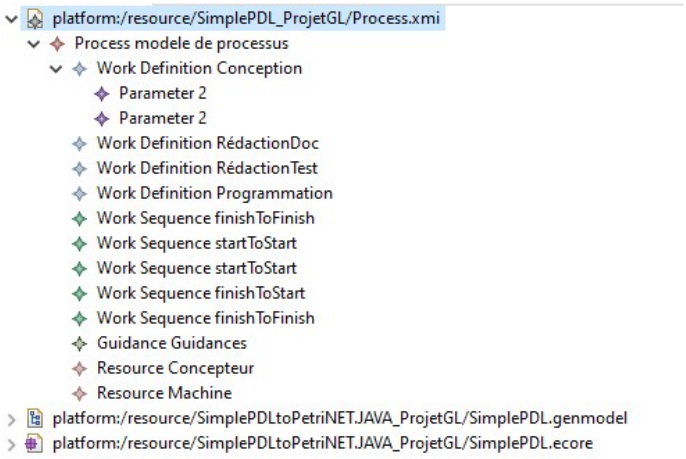

FIGURE 2 – Modèle SimplePDL

On visualise le modèle SimplePDL, et on obtient le méta-modèle dans la figure 3.

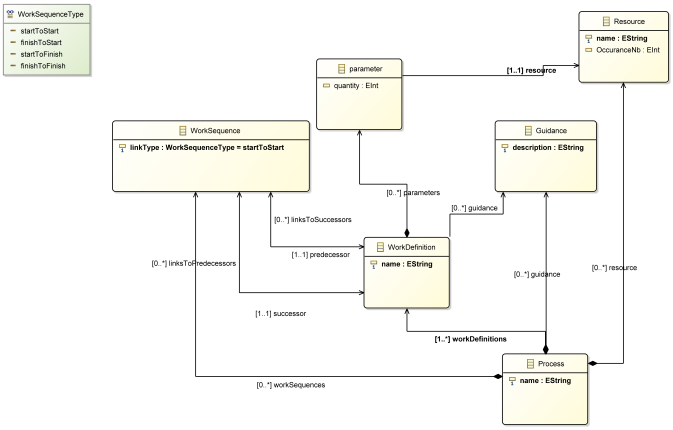

FIGURE 3 – Méta-modèle SimplePDL

On donne aussi l'architecture du modèle.

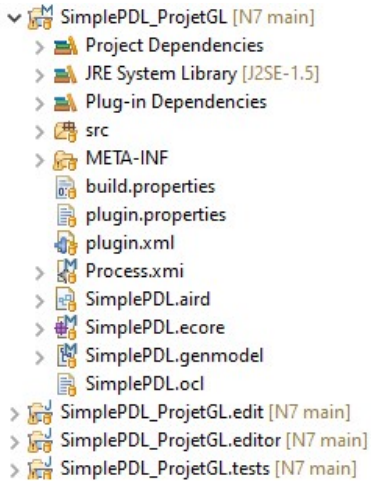

FIGURE 4 – Architecture SimplePDL

3.2 Définition des méta-modèles avec Ecore pour PetriNet

Un réseau Petri contient 3 éléments : des places, des transitions et des arcs reliant une place à une transition ou une transition à une place. Chaque place peut comporter un nombre positif ou nul de jetons.

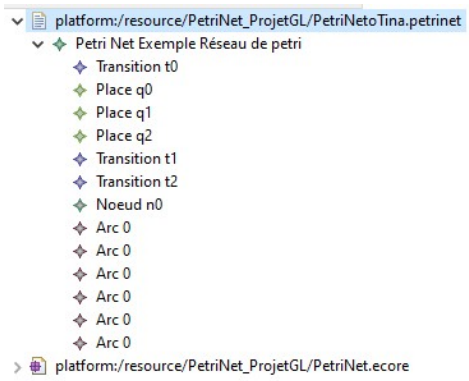

FIGURE 5 – Exemple d'un réseau Petri

On visualise le méta-modèle Petri, et on obtient le diagramme suivant :

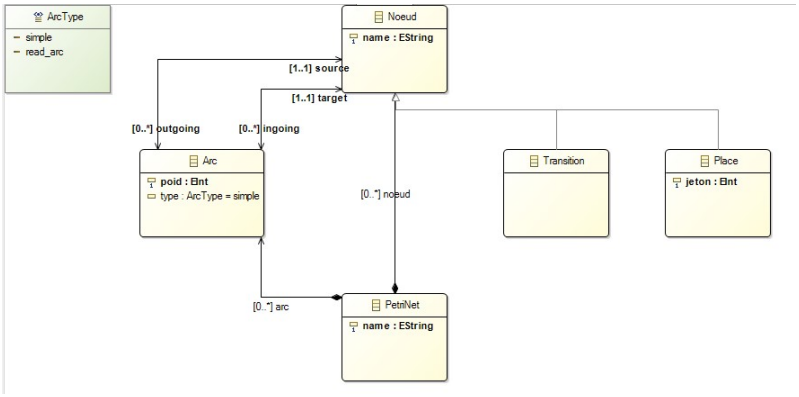

FIGURE 6 – Méta-modèle Petri

On donne l'architecture du réseau Petri.

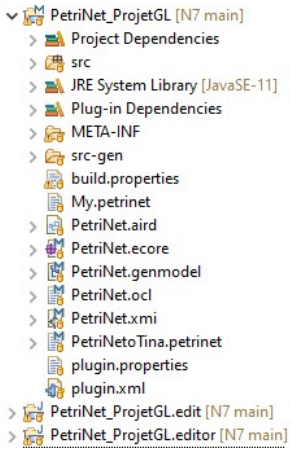

FIGURE 7 – Architecture du réseau Petri

4 Définitions de la sémantique statique avec OCL

4.1 Définitions de la sémantique statique avec OCL pour SimplePDL

On définit des contraintes OCL pour le méta-modèle simplePDL comme suit :

— Les noms doivent respectés la règle suivante : $[A-Za-z\_][A-Za-z0-9\_]*$

— Deux activités et deux ressources ne doivent pas avoir le même nom.

— Une activité ne doit pas dépendre d'elle même.

— La taille du nom d'une activité doit dépasser un caractère.

— Le nombre d'occurrence et la quantité des ressources doivent être positifs.

4.2 de la sémantique statique avec OCL pour le réseau Petri

A fin de de vérifier le bon fonctionnement du réseau Petri, on a définit des contraintes OCL qui doivent être vérifier avant. Les contraintes définies sont :

— Deux noeuds différents ne doivent pas avoir le même nom.

— Un arc ne doit jamais relier deux noeuds de même type.

— Le nombre de jetons dans une place doit être toujours positif ou nul.

5 Syntaxe concrète de SimplePDL

La syntaxe abstraite d'un DSML exprimée en Ecore ne peut pas être manipulé directement. Ainsi, on définit des syntaxes concrètes associée à la syntaxe abstraite pour faciliter la construction et la modification des modèles. Ces syntaxes peuvent être textuelles ou graphiques.

5.1 Syntaxe concrète textuelle

En utilisant l'outil Xtext, on définit une syntaxe concrète textuelle de SimplePDL avec Xtext. Pour chaque élément du modèle, on lui définit une grammaire qui peut être analysée en LL(k), ainsi on génère le méta-modèle SimplePDL étudié.

```
grammar project.simple.PDL with org.eclipse.xtext.common.Terminals

generate pDL "http://www.simple.project/PDL"

Process :
    'Process'
    name=ID
    '{'
    'workDefinitions' '{' workDefinitions+=WorkDefinition ( "," workDefinitions+=WorkDefinition)* '}'
        ('workSequences' '{' workSequences+=WorkSequence ( "," workSequences+=WorkSequence)* '}' )?
        ('guidance' '{' guidance+=Guidance ( "," guidance+=Guidance)* '}' )?
        ('resource' '{' resource+=Resource ( "," resource+=Resource)* '}' )?
    '}';
```

FIGURE 8 – Syntaxe textuelle de Process

```
WorkDefinition :
    {WorkDefinition}
    'WorkDefinition'
    name=ID
    '{'
        ('linksToPredecessors' '(' linksToPredecessors+=[WorkSequence] ( "," linksToPredecessors+=[WorkSequence])* ')' )?
        ('linksToSuccessors' '(' linksToSuccessors+=[WorkSequence] ( "," linksToSuccessors+=[WorkSequence])* ')' )?
        ('guidance' '(' guidance+=[Guidance] ( "," guidance+=[Guidance])* ')' )?
        ('parameters' '{' parameters+=parameter ( "," parameters+=parameter)* '}' )?
    '}';
```

FIGURE 9 – Syntaxe textuelle d'une activité

```
WorkSequence :
    'WorkSequence'
    '{'
        'linkType' linkType=WorkSequenceType
        'from' 'predecessor' predecessor=[WorkDefinition]
        'to' 'successor' successor=[WorkDefinition]
    '}';

Guidance :
    'Guidance'
    '{'
        'description' description=ID
    '}';

Resource :
    {Resource}
    'Resource'
    name=ID
    '{'
        ('OccuranceNb' OccuranceNb=INT)?
    '}'
;
```

FIGURE 10 – Syntaxe textuelle d'une dépendance, guidance et ressource

```
parameter :
    'parameter'
    '{'
        ('quantity' quantity=INT)?
        'resource' resource=[Resource]
    '}';


enum WorkSequenceType :
startToStart = 'startToStart' | finishToStart = 'finishToStart' | startToFinish = 'startToFinish' | finishToFinish = 'finishToFinish';
```

FIGURE 11 – Syntaxe textuelle d'un paramètre

5.2 Syntaxe concrète graphique

A l'instar de la syntaxe concrète textuelle, on peut définir une syntaxe concrète graphique pour visualiser ou éditer le modèle étudié. Pour cela, nous avons utilisé l'outil Siruis fournit par Eclipse.

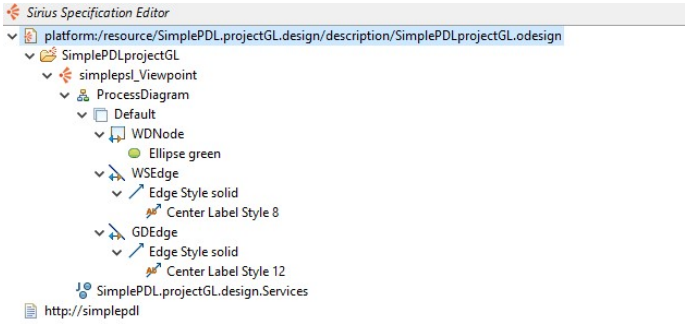

FIGURE 12 – L'éditeur graphique Siruis

La syntaxe concrète graphique du modèle SimplePDL étudié est dans la figure ci-dessous.

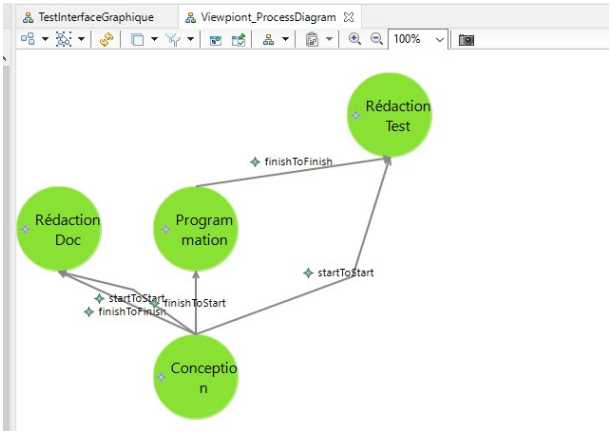

FIGURE 13 – Syntaxe graphique du modèle étudié

6 Définition de la transformation modèle à modèle (M2M)

Dans cette partie, nous avons défini des transformations du modèle SimplePDL vers le réseau Petri. Pour cela, nous avons utilisé deux langages, Java et ATL.

6.1 Transformation modèle à modèle avec Java

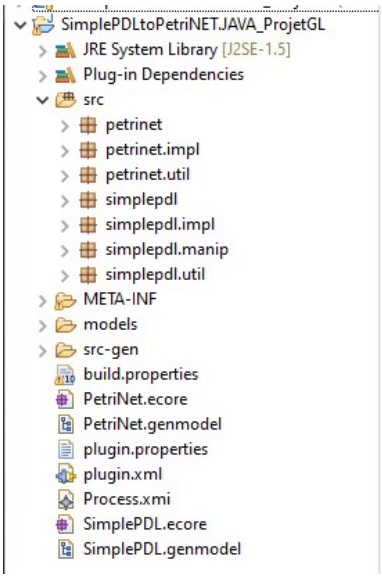

FIGURE 14 – Architecture de la transformation M2M avec Java

En utilisant le langage Java, nous avons défini la transformation SimplePDL au réseau Petri. Pour tester le programme, nous avons créé un modèle simple comme ci dessous.

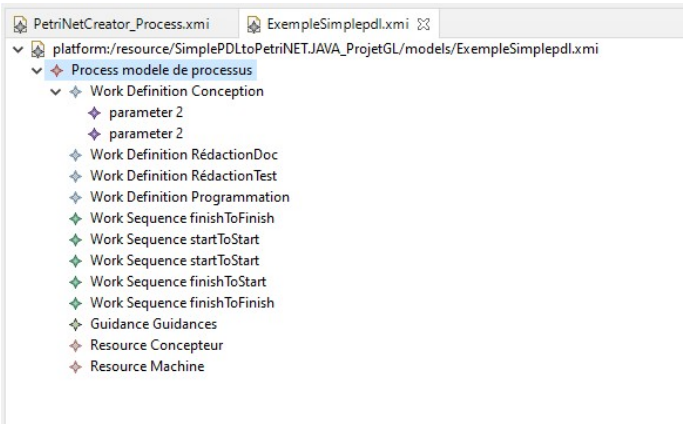

FIGURE 15 – Exemple d'un modèle SimplePDL

En appliquant la transformation en un réseau de Petri avec Java, nous obtenons le résultat suivant :

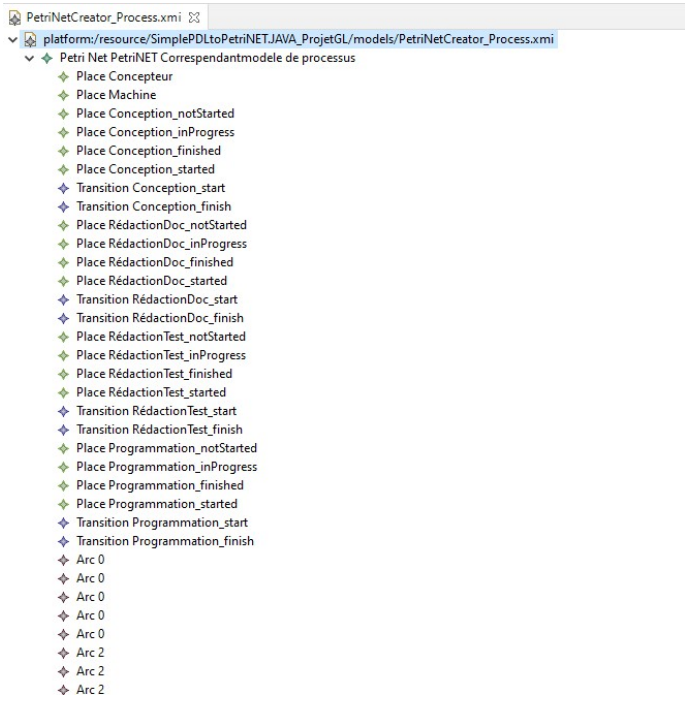

FIGURE 16 – Résultat de la transformation avec Java

6.2 Transformation modèle à modèle avec ATL

A l'instar de la transformation M2M avec Java, on défini une transformation d'un modèle SimplePDL à Petri en utilisant le langage ATL. Pour tester l'efficacité du programme, nous avons défini un exemple de modèle comme ci-dessous.

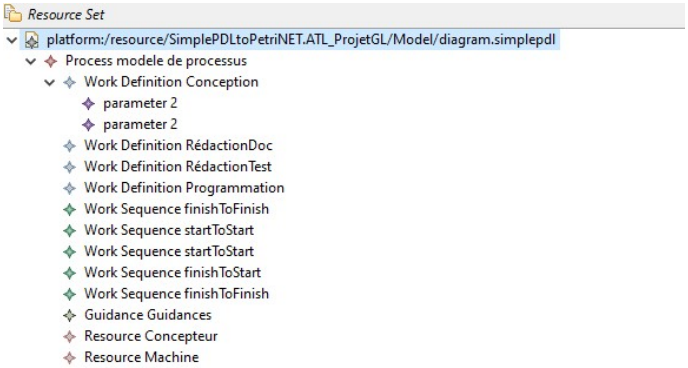

FIGURE 17 – Exemple d'un modèle SimplePDL

En appliquant la transformation en un réseau de Petri avec ATL, nous obtenons le résultat suivant :

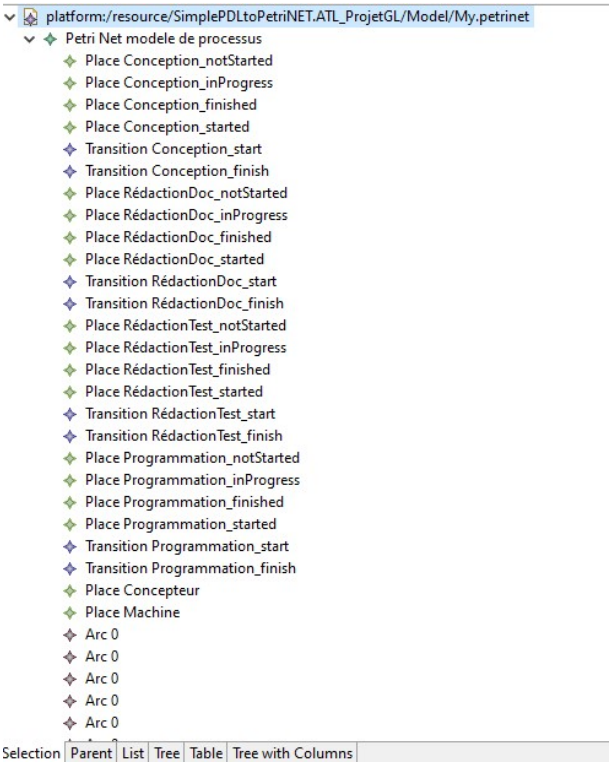

FIGURE 18 – Résultat de la transformation avec ATL

7 Définition de la transformation modèle à texte (M2T)

Nous définissons une transformation modèle à texte avec l'outil Acceleo du réseau Petri vers Tina. Nous commençons par créer un fichier HTML à partir du modèle SimplePDL. Ensuite, nous obtenons le fichier *toHTML.mtl*.

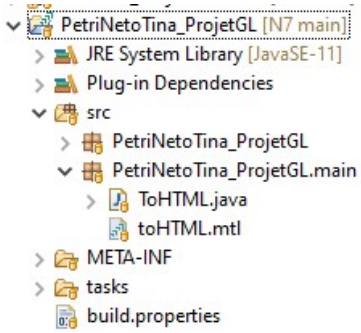

FIGURE 19 – L'architecture de la transformation M2T

Le principe d'Acceleo est de s'appuyer sur des gabarits (templates) des fichiers à engendrer. Le template se trouve dans le fichier *toHTML.mtl*. La figure 20 illustre le résultat obtenu par la transformation.

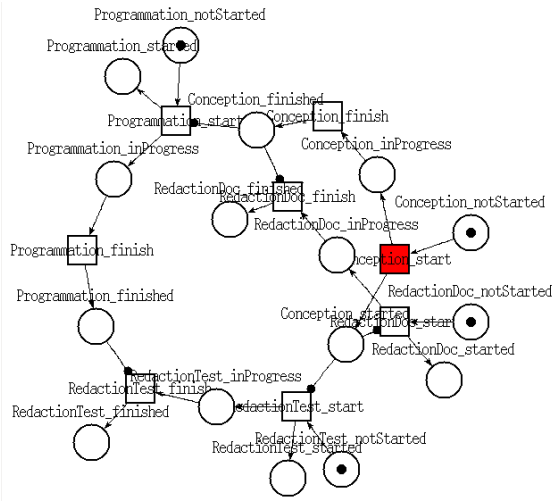

FIGURE 20 – Syntaxe graphique de la transformation M2T

La syntaxe textuelle de ce résultat est dans la figure 21.

```
net PetriNET_Correspondantmodele_de_processus
pl Concepteur (3)
pl Machine (4)
pl Conception_notStarted (1)
pl Conception_inProgress (0)
pl Conception_finished (0)
pl Conception_started (0)
pl RedactionDoc_notStarted (1)
pl RedactionDoc_inProgress (0)
pl RedactionDoc_finished (0)
pl RedactionDoc_started (0)
pl RedactionTest_notStarted (1)
pl RedactionTest_inProgress (0)
pl RedactionTest_finished (0)
pl RedactionTest_started (0)
pl Programmation_notStarted (1)
pl Programmation_inProgress (0)
pl Programmation_finished (0)
pl Programmation_started (0)
tr Conception_start Conception_notStarted Concepteur*2 Machine*2 ->  Conception_inProgress Conception_started
tr Conception_finish Conception_inProgress Concepteur*2 Machine*2 ->  Conception_finished
tr RédactionDoc_start RedactionDoc_notStarted Conception_started?1 ->  RedactionDoc_inProgress RédactionDoc_started
tr RédactionDoc_finish RedactionDoc_inProgress Conception_finished?1 ->  RedactionDoc_finished
tr RédactionTest_start RedactionTest_notStarted Conception_started?1 ->  RedactionTest_inProgress RédactionTest_started
tr RédactionTest_finish RedactionTest_inProgress Programmation_finished?1 ->  RedactionTest_finished
tr Programmation_start Programmation_notStarted Conception_finished?1 ->  Programmation_inProgress Programmation_started
tr Programmation_finish Programmation_inProgress ->  Programmation_finished
```

FIGURE 21 – Syntaxe textuelle de la transformation M2T

8 Validation de la transformation SimplePDL2PetriNet

Pour permettre la vérification de la transformation SimplePDL2PetriNet, TINA dispose d'un *model-checker* pour la logique temporelle LTL (Linear Time Logic). Cette logique permet d'exprimer des propriétés caractéristiques de la transformation et les vérifier.

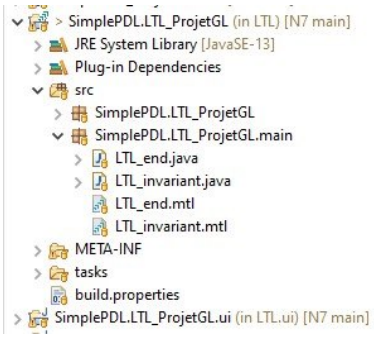

FIGURE 22 – Architecture LTL

Tout d'abord, nous avons engendré les propriétés LTL pour vérifier la terminaison sur les modèles de processus. Nous remarquons la validation de toutes les propriétés définies.

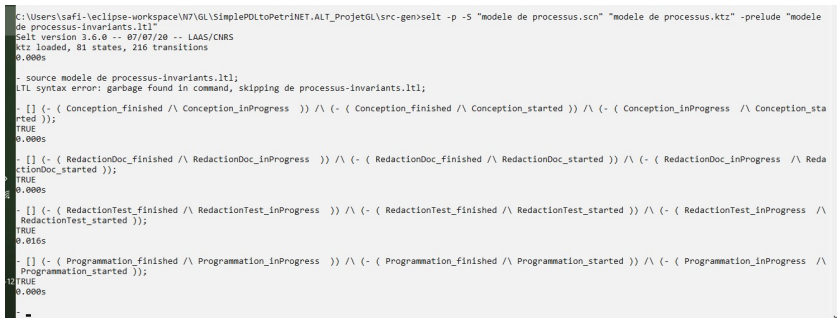

FIGURE 23 – Les propriétés LTL pour vérifier la terminaison

Ensuite, nous avons engendré des propriétés LTL correspondant aux invariants pour valider la transformation SimplePDL vers le réseau Petri.

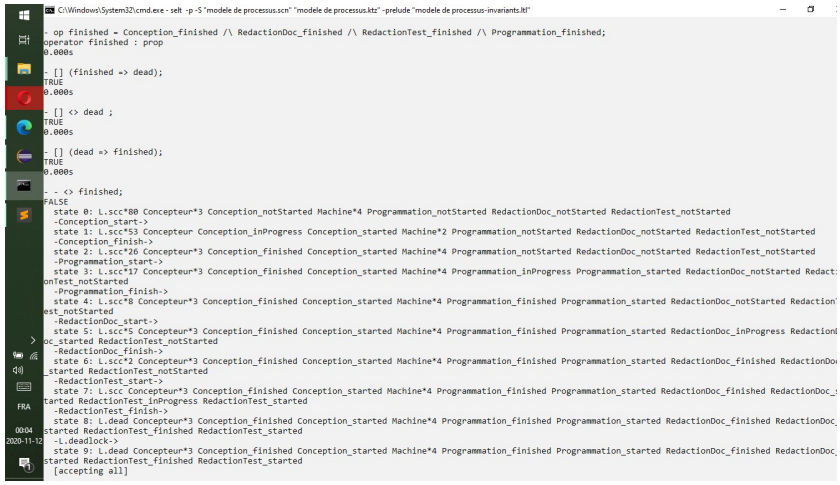

FIGURE 24 – Les propriétés LTL correspondant aux invariants

Nous remarquons que la dernière propriété n'est pas vérifié, c'est propriété qui vérifie si le processus termine ou pas. Cette propriété ne doit pas être vérifiée parce que le processus va terminer dans un temps.

9 Conclusion

Ce mini-projet nous a beaucoup intéressé car il nous a permit de consolider nos compétences en tout ce qui concerne la modélisation des transformations classiques. Toutefois, l'utilisation d'Eclipse nous a posé des problèmes avec ses bugs récurrent.