TOULOUSE INP N7

SCIENCE DU NUMERIQUE

S7 - Ingénierie Dirigée par les Modèles

Mini-Projet

Chaîne de vérification de modèles de processus

Auteurs :
Yvan Charles KIEGAIN DJOKO
Aicha BENNAGHMOUCH

15 novembre 2022

Systèmes Logiciels - Groupe L3
Département Sciences du Numérique - Deuxième année
2022-2023

Table des matières

1 Introduction **2**

2 Les métamodèles de SimplePDL et PetriNet **2**
2.1 SimplePDL 2
2.2 PetriNet 3

3 Les contraintes OCL associées à ces métamodèles **4**
3.1 SimplePDL 4
3.2 PetriNet 5

4 Transformation modèle à modèle (M2M) avec EMF/JAVA et avec ATL **5**
4.1 Transformation SimplePDL vers PetriNet en utilisant EMF/JAVA 5
4.2 Transformation SimplePDL vers PetriNet en utilisant ATL 7

5 Transformation modèle à texte (M2T) avec Acceleo **8**
5.1 Transformation PetriNet vers Tina 8
5.2 Propriétés LTL 10
5.2.1 Propriétés LTL permettant de vérifier la terminaison d'un processus et les appliquer sur différents modèles de processus 10
5.2.2 Propriétés LTL correspondant aux invariants de SimplePDL pour valider la transformation écrite 10

6 Editeur graphique SimplePDL avec Sirius **11**

7 Syntaxe concrète textuelle de SimplePDL avec Xtext **12**

8 Conclusion **13**

Table des figures

1 Métamodèle SimplePDL sans les ressources 2
2 Métamodèle SimplePDL : *simplepdl.png* 3
3 Métamodèle PetriNet : *petrinet.png* 4
4 Modèle CollegeApplication en SimplePDL : *process1.png* 5
5 Modèle CollegeApplication transformé en PetriNet en utilisant EMF/JAVA 6
6 Modèle CollegeApplication transformé en PetriNet en utilisant ATL 7
7 Modèle WriteABook 8
8 Fichier .net généré par la transformation (M2T) PetriNet to Tina 8
9 Graphe associé au fichier .net à l'état initial 9
10 Graphe associé au fichier .net à l'état final 9
11 Propriétés LTL permettant de vérifier la terminaison d'un processus 10
12 Résultat commande selt *-end.ltl 10
13 Propriétés LTL correspondant aux invariants de SimplePDL 10
14 Résultat commande selt *-inv.ltl 11
15 Modèle CollegeApplication : *process1.png* 11
16 Métamodèle SimplePDL : *process2Diagram.png* 12
17 Exemple d'un Process définie avec la syntaxe PDL 12

1 Introduction

Ce mini-projet consiste à produire une chaîne de vérification de modèles de processus SimplePDL dans le but de vérifier leur cohérence, en particulier pour savoir si le processus décrit peut se terminer ou non. Pour répondre à cette question, nous utilisons les outils de model-checking définis sur les réseaux de Petri au travers de la boîte à outils Tina. Il nous faudra donc traduire un modèle de processus en un réseau de Petri.

Ce mini-projet consiste pour l'essentiel à définir la chaîne de vérification de modèles de processus. Les principales étapes sont les suivantes :

1. Définition des métamodèles de SimplePDL et de PetriNet avec Ecore.
2. Définition de la sémantique statique avec OCL.
3. Définition d'une transformation de modèle à modèle (M2M) avec EMF/Java et avec ATL.
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 de syntaxes concrètes graphiques avec Sirius.
6. Définition de syntaxes concrètes textuelles avec Xtext.

2 Les métamodèles de SimplePDL et PetriNet

2.1 SimplePDL

Le SimplePDL est un langage de métamodélisation qui sert pour décrire des modèles de processus. Le métamodèle de SimplePDL contient la notion de ProcessElement comme généralisation de WorkDefinition et WorkSequence. Un processus est donc un ensemble d'éléments de processus qui sont soit des activités, soit des dépendances. La notion de Guidance a été également ajoutée. C'est l'équivalent d'une annotation UML : elle permet d'associer un texte à plusieurs éléments d'un processus.

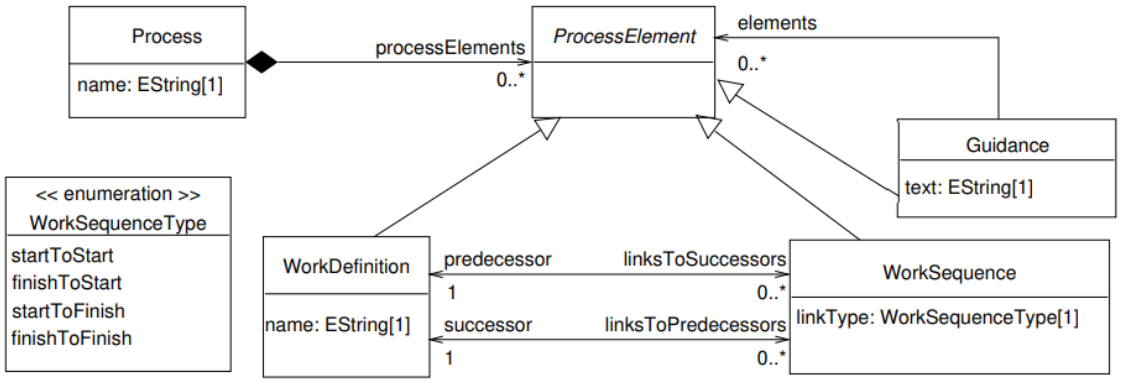

FIGURE 1 – Métamodèle SimplePDL sans les ressources

Un métamodèle de SimplePDL définit différentes activités (WorkDefinition) et dépendances (WorkSequence). Pour réaliser une activité, des ressources peuvent être nécessaires. Nous nous intéressons simplement aux types de ressources nécessaires et au nombre d'occurrences d'un type de ressource. Un type de ressource sera seulement caractérisé par son nom et la quantité d'occurrences de celle-ci. Une ressource contient des occurrences, une activité loue des occurrences d'une ressource au début de son exécution, ces occurrences sont alors utilisés exclusivement par cette activité jusqu'à

la fin de sa réalisation. Afin de réaliser cette tâche, on a donc ajouter deux classes (classes qui héritent de ProcessElement) au métamodèle de SimplePDL :

— Resource : une ressource est défini par son nom (name : EString) qui décrit son type et ses occurrences (quantity : Eint).
— Allocate : cette classe représente le nombre d'occurrences (occurence : EInt) prises par une activité parmi les occurrences d'une ressource pour effectuer sa réalisation.

Les fichiers fournies pour cette partie sont :
— simplepdl.png
— SimplePDL.ecore

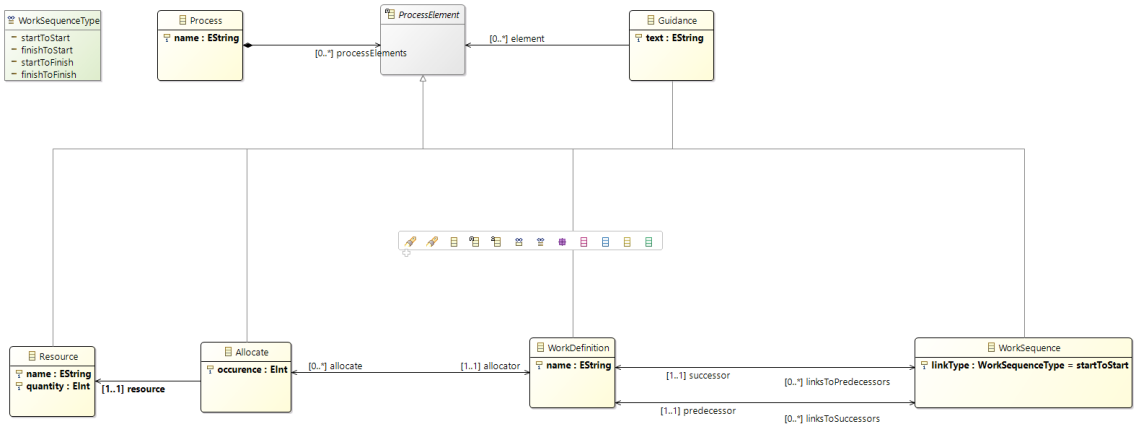

FIGURE 2 – Métamodèle SimplePDL : *simplepdl.png*

2.2 PetriNet

On crée dans un premier temps une classe Net représentant un réseau de Petri. Ensuite, en se basant sur la définition du réseau de Petri, on trouve que ces composants principaux sont des arcs, des places et des transitions. Par analogie au métamodèle de SimplePDL, on crée alors trois classes représentant respectivement les arcs (Arc), les places (Place) et les transitions (Transition) ainsi qu'une classe NetElement dont héritent les trois classes précédentes. NetElement peut donc être soit un arc, soit une place ou une transition, et Net est alors une composition de plusieurs NetElement.
On crée également une classe Node, caractérisée par son nom (name : EString), de laquelle hérite les classes Place et Transition vu que ces dernières ont les mêmes propriétés à part le fait peut qu'une place peut contenir des jetons (token : EInt). Il y a également un arc de sortie (outArc : Arc) et un arc d'entrée (inArc : Arc) pour Node.
Un Arc permet de relier une place et une transition. Il existe deux types d'arcs : un arc classic et un read arc. Afin de différencier ces deux types, on crée une énumeration ArcType et on ajoute un attribut (type : ArcType) à la classe Arc. Un Arc est également caractérisé par son poids (weight : EInt) et son nom (name : EString). Un arc relie deux nodes soit une place soit une transition avec les attributs source et target.

Les fichiers fournies pour cette partie sont :
— petrinet.png
— Petri.ecore

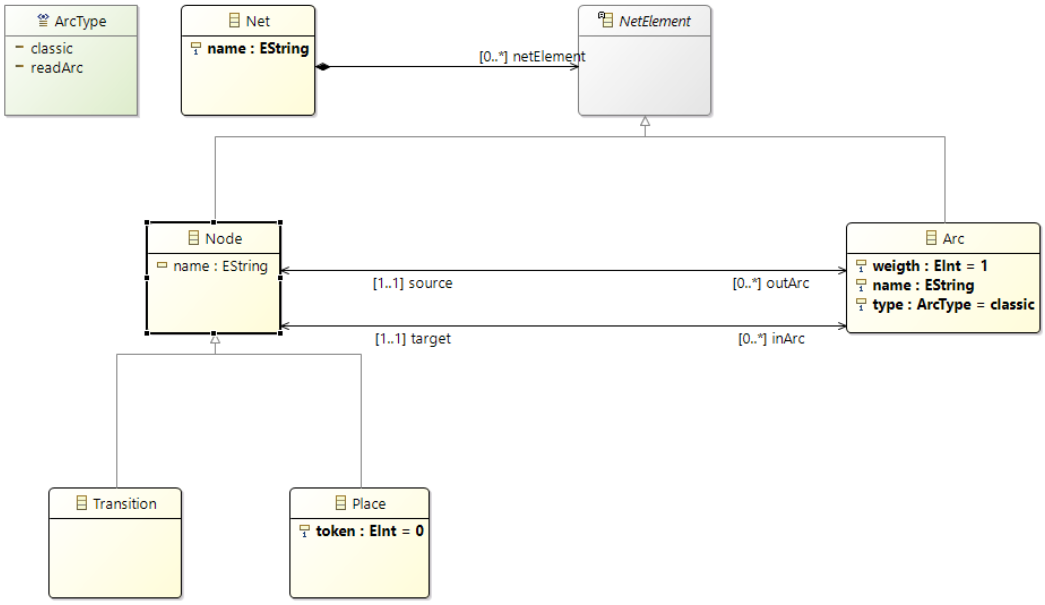

FIGURE 3 – Métamodèle PetriNet : *petrinet.png*

3 Les contraintes OCL associées à ces métamodèles

Le langage de méta-modélisation Ecore ne permet pas d'exprimer toutes les contraintes que doivent respecter les modèles de processus. On complète donc la description structurelle du méta-modèle par des contraintes exprimées en OCL.

3.1 SimplePDL

Pour SimplePDL, les contraintes sont :
— processValidName : Le nom d'un Process ne doit être composé que de lettres, chiffres ou soulignés, un chiffre ne peut pas être première position.
— wdUniqNames : Deux activités différentes d'un même processus ne peuvent pas avoir le même nom.
— atLeastTwoChar : Le nom d'une WorkDefinition doit avoir au moins deux caractères.
— resourceUniqueType : Une WorkDefinition ne peut avoir qu'une seule allocation par ressource
— wdValidName : Le nom d'une activité ne doit être composé que de lettres, chiffres ou soulignés, un chiffre ne peut pas être première position.
— notReflexive : Une dépendance ne peut pas être réflexive (Une WorkSequence ne peut pas lier deux activités identiques).
— successorAndPredecessorInSameProcess : Le predecessor et le successor d'une WorkSequence doivent être dans le même processus.
— positiveResourceQuantity : Pour qu'une Resource soit définie, il faut que son nombre soit strictement positif

— resourceValidName : Le nom d'une ressource ne doit être composé que de lettres, chiffres ou soulignés, un chiffre ne peut pas être première position.
— positiveOccurence : Le nombre d'occurences dans Allocate doit être supérieur ou égal à zéro
— maxQuantity : Pour Allocate, le nombre d'occurence doit être inférieure à la quantité de la Ressource associée.

Les fichiers fournies pour cette partie sont :
— SimplePDL.ocl

3.2 PetriNet

Pour PetriNet, les contraintes sont :
— netValidName : Le nom d'un réseau de pétri ne doit être composé que de lettres, chiffres ou soulignés, un chiffre ne peut pas être première position.
— posToken : Toute Place doit avoir un nombre entier naturel de Jetons.
— placeUniqNames : Deux places différentes d'un même réseau de pétri ne peuvent pas avoir le même nom.
— transitionUniqNames : Deux Transitions différentes d'un même réseau de pétri ne peuvent pas avoir le même nom.
— posWeight : Le poids d'un Arc doit être strictement positif.
— differentTransition et differentPlace Un Arc doit relier une Transition et une Place.
— readArcSourceTarget Un ReadArc doit être entre une Place et une Transition.

Les fichiers fournies pour cette partie sont :
— SimplePDL.ocl

4 Transformation modèle à modèle (M2M) avec EMF/JAVA et avec ATL

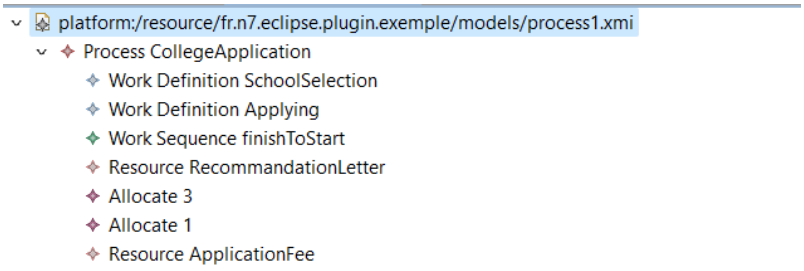

FIGURE 4 – Modèle CollegeApplication en SimplePDL : *process1.png*

4.1 Transformation SimplePDL vers PetriNet en utilisant EMF/JAVA

Il nous a également été demandé de définir une transformation SimplePDL vers PetriNet en utilisant EMF/Java, c'est-à-dire traduire les éléments de SimplePDL (Process, WorkDefinition, ...) en élément du réseau de Petri. La transformation modèle à modèle en Java consistant en la transformation d'un modèle de processus en un modèle de réseaux de Pétri passe par plusieurs étapes :

— Chargement des packages SimplePDL et PetriNet afin de l'enregistrer dans le registre d'Eclipse.
— Configuration de l'input qu'on fournit au programme Java ainsi que l'output qu'il doit nous rendre.
— Récupération du premier élément du modèle process(élément à la racine) et instanciation de la fabrique.

On suivra ensuite la logique suivante pour construire notre réseau de petri :

— Pour chaque WorkDefinition, on crée 4 places dans notre réseau(ready, started, running, finished) correspondant au état d'évolution de cette dernière ainsi que 2 transitions (start, finish) pour passer de ready à running et de running à finished en mettant 1 jetons à la place ready et zéro au reste puis on relie toutes ces nodes (Node : Place et Transition) entre eux avec des arcs.
— Pour chaque Ressource, on crée une Place dont le nombre de jetons sera initialisé par la quantité de cette ressource et on l'associe grâce à un arc à la transition start (pour conditionner le démarrage d'une activité) de chaque WorkDéfinition dont elle est la ressource. Le poids de ces arcs est fixé par l'attribut occurence de la EClass Allocate qui correspond au nombre d'occurrence de la ressource dont le WorkDefinition a besoin.
— Pour chaque WorkSequence, on crée un arc qui reliera, selon le type de la WorkSequence, les places started/finished et les transitions start/finish entre deux activités différentes et ces arcs ont la particularité d'être des readArc qui ne vont pas consommer le jeton pris depuis la place en source.

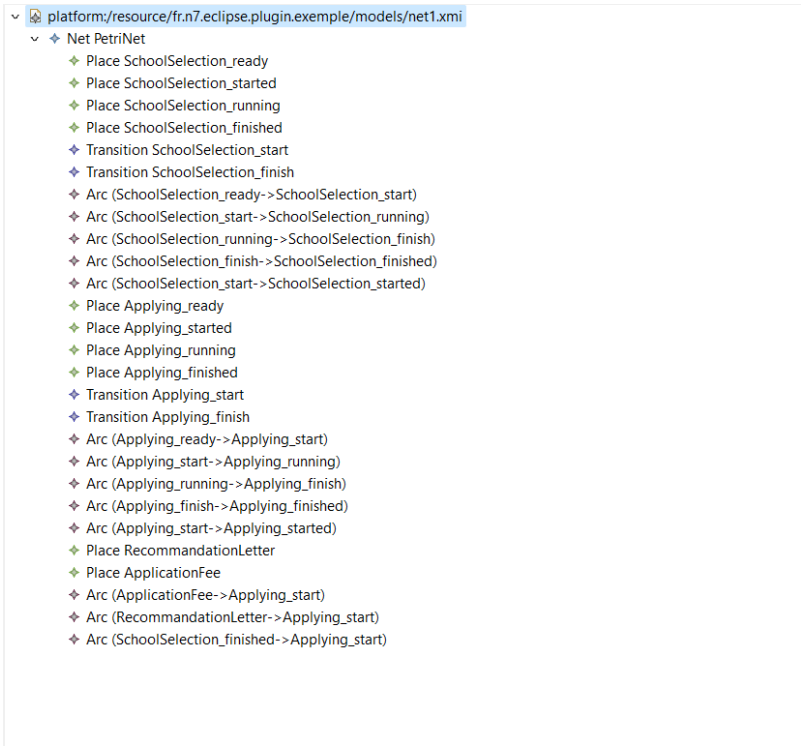

FIGURE 5 – Modèle CollegeApplication transformé en PetriNet en utilisant EMF/JAVA

Les fichiers fournies pour cette partie sont :
— ProcessToPedri.java

4.2 Transformation SimplePDL vers PetriNet en utilisant ATL

Dans la transformation ATL, le raisonnement est resté le même dans l'ensemble. Une différence notable ici est que durant la rédaction des transformations, on a constaté que la syntaxe de ATL avait l'air plus optimisée pour réaliser cette tâche rendant plus facile la transformation d'un élément du processus en un élément du réseau. Vous observerez que la seule différence entre le fichier généré par emf/java et celui par atl se trouve être le nom des arcs qui n'a vraiment rien de significatif et avait à la base uniquement pour but la clarté de nos modèles en mode arborescence.

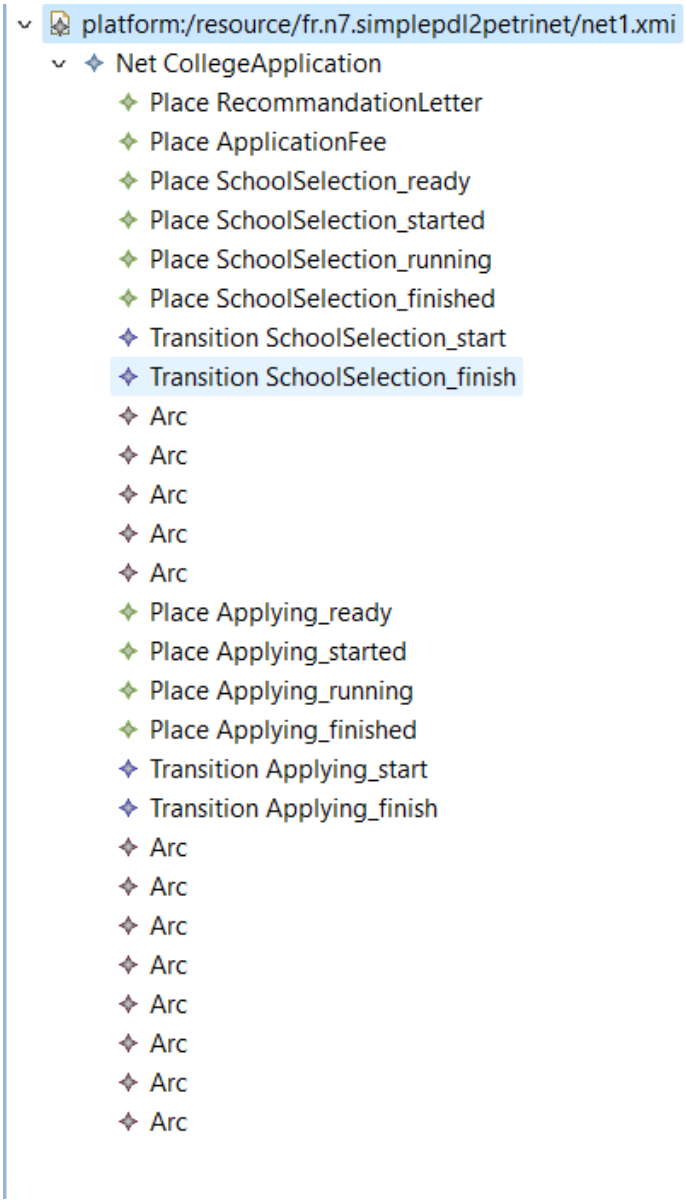

FIGURE 6 – Modèle CollegeApplication transformé en PetriNet en utilisant ATL

Les fichiers fournies pour cette partie sont :
— SimplePDL2PetriNet.atl

5 Transformation modèle à texte (M2T) avec Acceleo

5.1 Transformation PetriNet vers Tina

On a établit une transformation modèle à texte d'un modèle de réseau de Petri en la syntaxe textuelle utilisée par les outils de Tina, à savoir la syntaxe en extension .net . En se basant sur cette syntaxe et en s'inspirant des fichiers toHTML.mtl et toDOT.mtl précédemment réalisés en TP, on a complété le template Acceleo afin de réaliser la transformation d'un réseau de Pétri en tina.PAinsi, On a ensuite pu visualiser graphiquement le modèle l'outil nd (Net Draw).

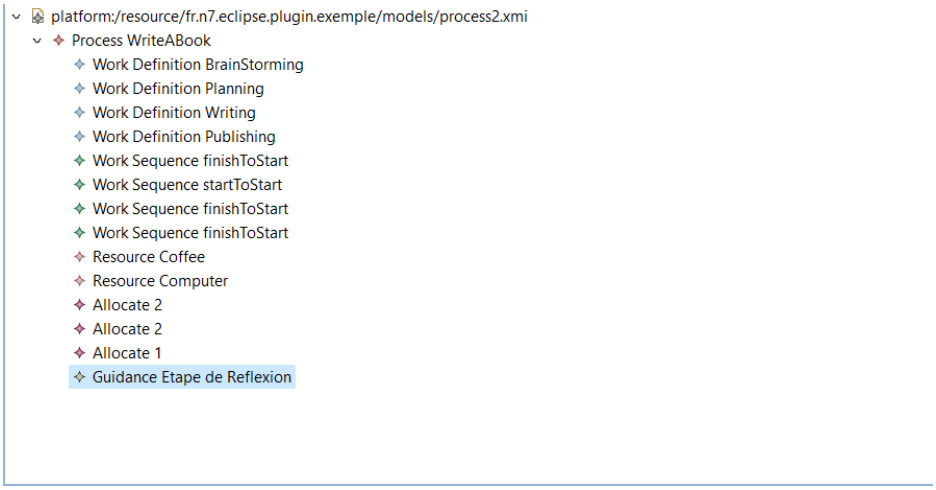

FIGURE 7 – Modèle WriteABook

```
net WriteABook
pl Coffee (4)
pl Computer (1)
pl BrainStorming_ready (1)
pl BrainStorming_started (0)
pl BrainStorming_running (0)
pl BrainStorming_finished (0)
pl Planning_ready (1)
pl Planning_started (0)
pl Planning_running (0)
pl Planning_finished (0)
pl Writing_ready (1)
pl Writing_started (0)
pl Writing_running (0)
pl Writing_finished (0)
pl Publishing_ready (1)
pl Publishing_started (0)
pl Publishing_running (0)
pl Publishing_finished (0)
tr BrainStorming_start BrainStorming_ready Coffee*2 Planning_started?1 -> BrainStorming_running BrainStorming_started
tr BrainStorming_finish BrainStorming_running -> BrainStorming_finished
tr Planning_start Planning_ready Coffee*2 -> Planning_running Planning_started
tr Planning_finish Planning_running -> Planning_finished
tr Writing_start Writing_ready Computer BrainStorming_finished?1 Planning_finished?1 -> Writing_running Writing_started
tr Writing_finish Writing_running -> Writing_finished
tr Publishing_start Publishing_ready Writing_finished?1 -> Publishing_running Publishing_started
tr Publishing_finish Publishing_running -> Publishing_finished
```

FIGURE 8 – Fichier .net généré par la transformation (M2T) PetriNet to Tina

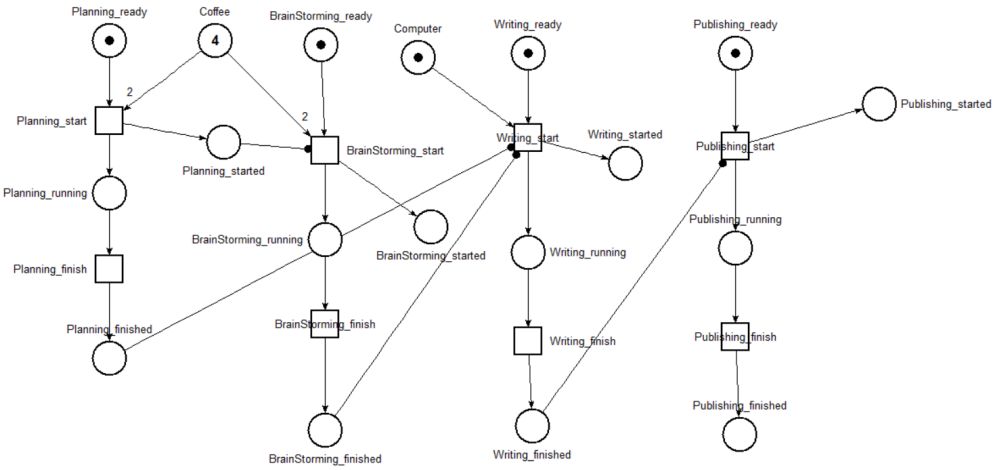

FIGURE 9 – Graphe associé au fichier .net à l'état initial

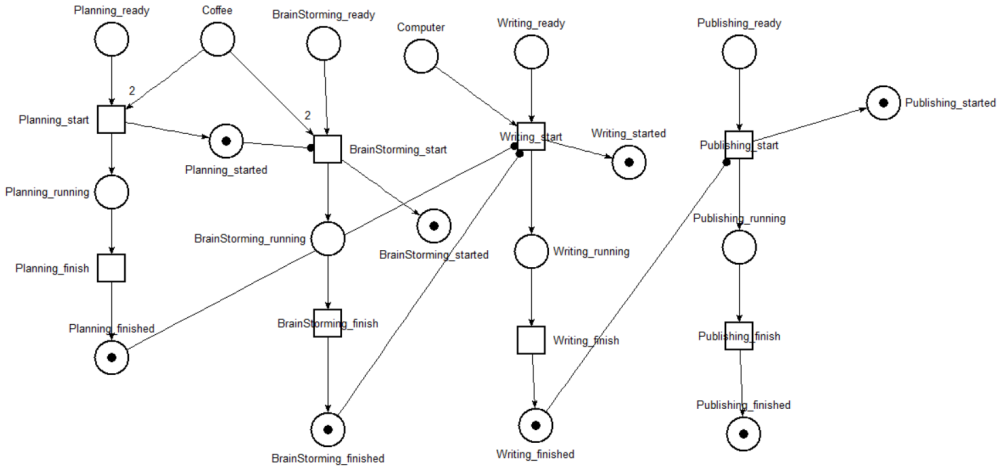

FIGURE 10 – Graphe associé au fichier .net à l'état final

Les fichiers fournies pour cette partie sont :
— toTINA.mtl

5.2 Propriétés LTL

5.2.1 Propriétés LTL permettant de vérifier la terminaison d'un processus et les appliquer sur différents modèles de processus

Le fichier toLTLend.mtl contient la transformation vers la Syntaxe LTL à partir d'un modèle de processus en vérifiant les propriétées de terminaison de toutes les activités du processus. Cette transformation engendre un fichier .ltl qu'on peut vérifier.

exemple1_end.ltl

```
op finished =Conception_finished /\ RedactionDoc_finished /\ Programmation_finished /\ RedactionTests_finished;
[] <> dead ;
[] (dead => finished);
- <> finished;
```

FIGURE 11 – Propriétés LTL permettant de vérifier la terminaison d'un processus

```
ykiegain@ykiegain-VirtualBox:~/Documents$ ./tina-3.7.0/bin/selt -p -S WriteABook.scn WriteABook.ktz -prelude WriteABook_end.ltl
Selt version 3.7.0 -- 05/19/22 -- LAAS/CNRS
ktz loaded, 11 states, 12 transitions
0.011s

- source WriteABook_end.ltl;
operator finished : prop
TRUE
TRUE
FALSE
  state 0: L.scc*10 BrainStorming_ready Coffee*4 Computer Planning_ready Publishing_ready Writing_ready
  -Planning_start->
  state 1: L.scc*9 BrainStorming_ready Coffee*2 Computer Planning_running Planning_started Publishing_ready Writing_ready
  -BrainStorming_start->
  state 2: L.scc*7 BrainStorming_running BrainStorming_started Computer Planning_running Planning_started Publishing_ready Writing_ready
  -BrainStorming_finish->
  state 3: L.scc*5 BrainStorming_finished BrainStorming_started Computer Planning_running Planning_started Publishing_ready Writing_ready
  -Planning_finish->
  state 4: L.scc*4 BrainStorming_finished BrainStorming_started Computer Planning_finished Planning_started Publishing_ready Writing_ready
  -Writing_start->
  state 5: L.scc*3 BrainStorming_finished BrainStorming_started Planning_finished Planning_started Publishing_ready Writing_running Writing_started
  -Writing_finish->
  state 6: L.scc*2 BrainStorming_finished BrainStorming_started Planning_finished Planning_started Publishing_ready Writing_finished Writing_started
  -Publishing_start->
  state 7: L.scc BrainStorming_finished BrainStorming_started Planning_finished Planning_started Publishing_running Publishing_started Writing_finished Writing_started
  -Publishing_finish->
  state 8: L.dead BrainStorming_finished BrainStorming_started Planning_finished Planning_started Publishing_finished Publishing_started Writing_finished Writing_started
  -L.deadlock->
  state 9: L.dead BrainStorming_finished BrainStorming_started Planning_finished Planning_started Publishing_finished Publishing_started Writing_finished Writing_started
  [accepting all]
0.003s
```

FIGURE 12 – Résultat commande selt *-end.ltl

Les fichiers fournies pour cette partie sont :
— toLTLend.mtl

5.2.2 Propriétés LTL correspondant aux invariants de SimplePDL pour valider la transformation écrite

Le fichier toLTLinv.mtl transforme le modèle d'un processus aux propriétées LTL des invariants dans le processus. Cette transformation engendre un fichier .ltl qu'on peut vérifier.

exemple1_inv.ltl

```
[] <> (- ( Conception_ready /\ Conception_running )) /\ (- ( Conception_running /\ Conception_finished )) /\ (- ( Conception_finished /\ Conception_ready ));
[] <> (- ( RedactionDoc_ready /\ RedactionDoc_running )) /\ (- ( RedactionDoc_running /\ RedactionDoc_finished )) /\ (- ( RedactionDoc_finished /\ RedactionDoc_reac
[] <> (- ( Programmation_ready /\ Programmation_running )) /\ (- ( Programmation_running /\ Programmation_finished )) /\ (- ( Programmation_finished /\ Programmatic
[] <> (- ( RedactionTests_ready /\ RedactionTests_running )) /\ (- ( RedactionTests_running /\ RedactionTests_finished )) /\ (- ( RedactionTests_finished /\ Redacti
```

FIGURE 13 – Propriétés LTL correspondant aux invariants de SimplePDL

Les fichiers fournies pour cette partie sont :
— toLTLinv.mtl

```
ykiegain@ykiegain-VirtualBox:~/Documents$ ./tina-3.7.0/bin/selt -p -S WriteABook.scn WriteABook.ktz -prelude WriteABook_inv.ltl
Selt version 3.7.0 -- 05/19/22 -- LAAS/CNRS
ktz loaded, 11 states, 12 transitions
0.001s

- source WriteABook_inv.ltl;
TRUE
TRUE
TRUE
TRUE
0.003s
```

FIGURE 14 – Résultat commande selt *-inv.ltl

6 Editeur graphique SimplePDL avec Sirius

Une syntaxe concrète graphique fournit un moyen plus agréable pour visualiser et éditer un modèle. Sirius nous a permis de développer un éditeur graphique SimplePDL pour saisir graphiquement un modèle de processus, y compris les ressources. Pour ce faire, on a créé un Sirius / Viewpoint Specification Project. On a défini comment représenter les éléments du modèle ainsi que la correspondance entre les éléments du modèle et les éléments graphiques. On a également défini une palette qui propose les outils pour créer les différents éléments graphiques. Ceci permet à l'utilisateur d'ajouter graphiquement des WorkDefinition, des WorkSequence, des Guidance, des Resource et des Allocate.

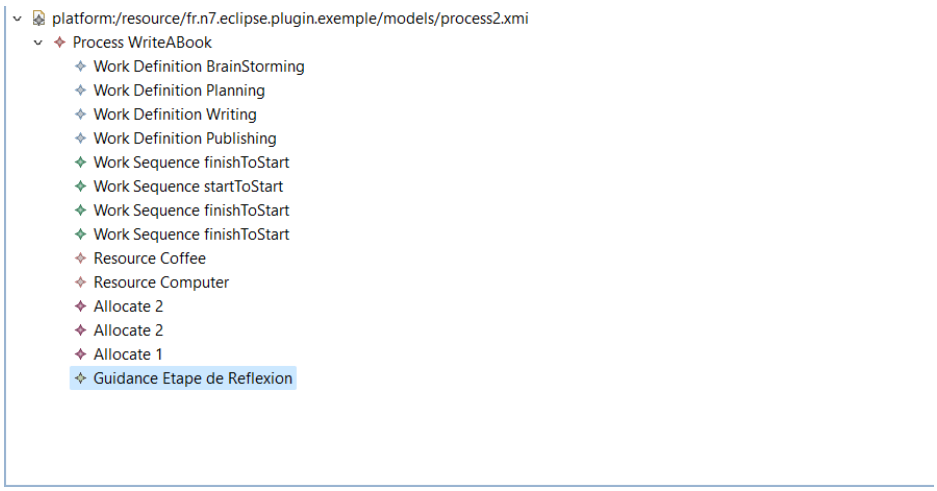

FIGURE 15 – Modèle CollegeApplication : *process1.png*

Les fichiers fournies pour cette partie sont :
— simplepdl.odesign

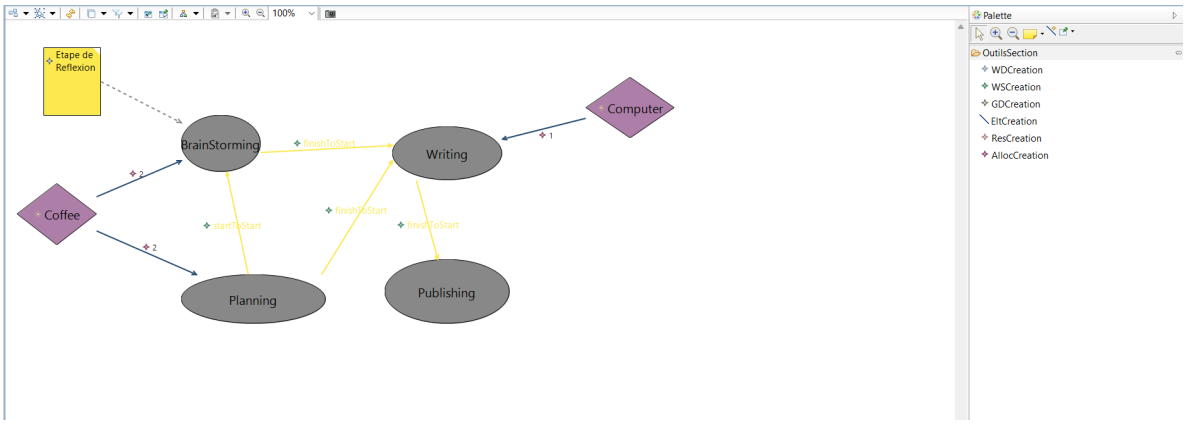

FIGURE 16 – Métamodèle SimplePDL : *process2Diagram.png*

7 Syntaxe concrète textuelle de SimplePDL avec Xtext

En utilisant le métamodèle de SimplePDL, on a pu générer une syntaxe textuelle. On a ensuite fait quelques modifications sur la grammaire proposée par défaut pour qu'elle soit plus facile à rédiger. Pour cela, on s'est inspiré de la grammaire de pdl1 proposée en TP.

exemple.pdl

```
process WriteABook {
wd BrainStorming
wd Planning
wd Writing
wd Publishing
ws f2s from BrainStorming to Writing
ws s2s from Planning to BrainStorming
ws f2s from Planning to Writing
ws f2s from Writing to Publishing
resource Coffee quantity 4
resource Computer quantity 1
note "Reflexion"
}
```

FIGURE 17 – Exemple d'un Process définie avec la syntaxe PDL

Les fichiers fournies pour cette partie sont :
— PDL.xtext

8 Conclusion

Ce mini-projet nous permet de mieux appréhender le cours de Génie et Ingénierie des Systèmes et de manipuler les différentes notions que l'on y trouve (modèle, métamodèles, transformations). Il nous a permis de découvrir plusieurs langages et syntaxes de programmations différentes.
On a également rencontré de nombreuses difficultés de part l'incident qu'a connu l'école au début de l'année et les nombreux bugs survenant durant le projet avec la manipulation des plugins et Eclipse en générale. Mais tout ceci n'aura fait que nous permettre de travailler sur nos capacités d'adaptation et d'autonomie face à diverses situations qui sont des qualités qui représente bien un ingénieur mais surtout un développeur.
Les réseaux de petri, Acceleo, Sirius, Xtext, ATL, OCL, EMF/Java, Eclipse Modeling Tools, Tina, et les modèles ; autant de notions que ce projet nous aura permis de mieux comprendre et assimiler.