Rapport de projet
Chaîne de vérification des modèles de processus

FAN Yanghai
SADURNI Thomas
ROUX Thibault

Département Sciences du Numérique - Filière Image et Multimédia
2020-2021

Contents

1 Introduction **3**

2 Tâches à réaliser **3**
2.1 Compléter le métamodèle SimplePDL pour prendre en compte les ressources 3
2.2 Définir le métamodèle PetriNet 4
2.3 Définir les contraintes OCL pour capturer les contraintes qui n'ont pu l'être par les métamodèles 5
2.3.1 SimplePDL 5
2.3.2 PetriNet 6
2.4 Donner une syntaxe concrète textuelle de SimplePDL avec Xtext 6
2.5 Développer un éditeur graphique SimplePDL pour saisir graphiquement un modèle de processus, y compris les ressources 6
2.6 Définir une transformation SimplePDL vers PetriNet 7
2.6.1 En utilisant EMF/Java 7
2.6.2 En utilisant ATL 8
2.7 Valider la transformation SimplePDL vers PetriNet Java en faisant des tests 8
2.8 Définir des transformations PetriNet ou SimpleDPL vers un autre modèle à texte en utilisant Acceleo 8
2.8.1 SimplePDL vers HTML 8
2.8.2 PetriNet vers Tina 9
2.8.3 PetriNet vers Dot 9
2.9 Engendrer des propriétés LTL 10
2.9.1 Propriétés permettant de vérifier la terminaison d'un processus et les appliquer sur différents modèles de processus 10
2.9.2 Propriétés correspondant aux invariants de SimplePDL pour valider la transformation écrite 10

3 Mode d'emploi de la conversion à partir d'un SimplePDL : déroulement de l'oral **11**
3.1 Conversion PDL vers PetriNet 11
3.2 Conversion en Tina ou Dot 11
3.3 Synthaxe LTL 11

4 Conclusion **11**

List of Figures

1 Métamodèle SimplePDL prenant en compte les ressources 3
2 Métamodèle PetriNet 4
3 Représentation de pdl-sujet, l'exemple du sujet du projet 7
4 Représentation d'une journée banale 7
5 Représentation d'une journée banale en HTML 9
6 Représentation des saisons en un fichier pour Tina 9
7 Représentation des saisons en un fichier Dot 10

1 Introduction

L'objectif de ce mini-projet était de produire une chaîne de vérification de modèles de processus SimplePDL et de vérifier leur cohérence. Nous avons pour ce faire fait en sorte de pouvoir traduire des modèles de processus en réseaux de Petri.

2 Tâches à réaliser

Ce mini-projet consistait en la réalisation de différentes tâches, telles que la complétion du métamodèle SimplePDL, la définition du métamodèle PetriNet, la définition des contraintes OCL, les transformations modèle à modèle, etc...

2.1 Compléter le métamodèle SimplePDL pour prendre en compte les ressources

Dans cette partie, nous devions compléter le métamodèle SimplePDL fourni en TP en y ajoutant les ressources et leur gestion. Pour cela nous avons decidé d'ajouter la classe "Resources" ainsi que "linksResources" agissant respectivement comme les classes "WorkDefinition" et "WorkSequence". "linksResources" a comme attribut le nombre de ressources dont l'activité a besoin et appelle usedResources pour choisir la ressource correspondante. Une ressource peut être utilisée plusieurs fois, c'est pour cela que nous avons mis dans la classe "Resources" son nombre d'occurences.

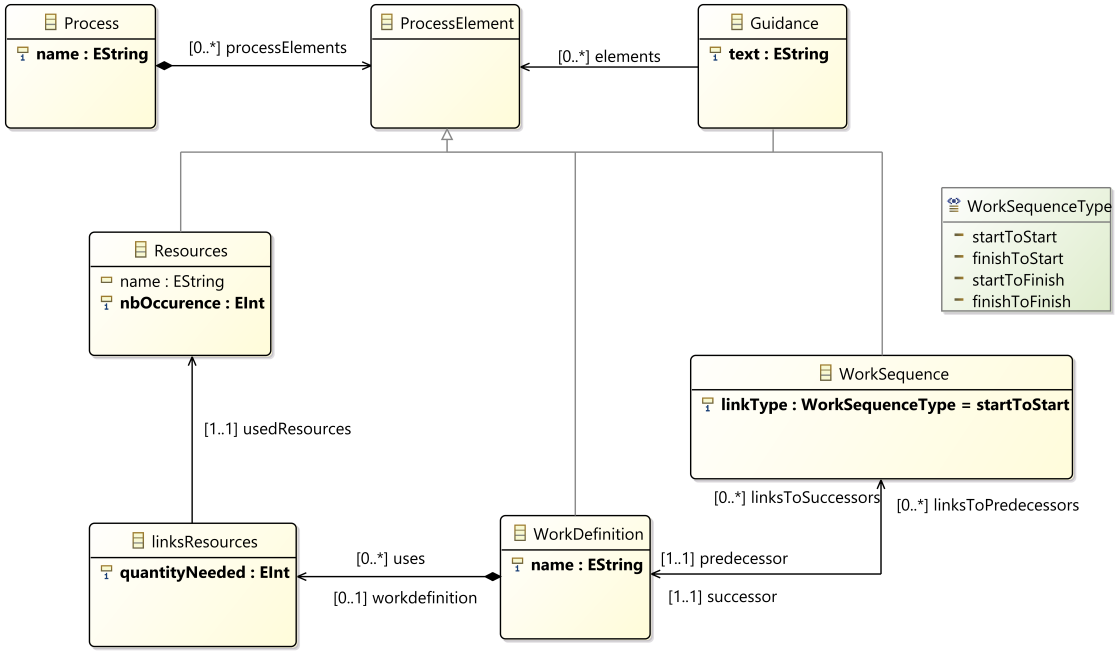


Figure 1: Métamodèle SimplePDL prenant en compte les ressources

Pour valider que notre modélisation était correcte, nous avons créé plusieurs instances de SimplePDL au format xmi. Ces fichiers peuvent aussi être récupérés dans le dossier livrables. Voici les différents exemples de SimplePDL que nous avons instanciés :

- *pdl-sujet.xmi* : L'exemple du sujet du miniprojet. Il est utilisé et représenté en Section 2.5, à la figure 3.
- *pdl-journee-banale.xmi* : Là-encore cet exemple est utilisé et représenté en Section 2.5. On peut le visualiser en figure 4.

2.2 Définir le métamodèle PetriNet

Nous avons ensuite eu à définir entièrement le métamodèle PetriNet en langage Ecore.

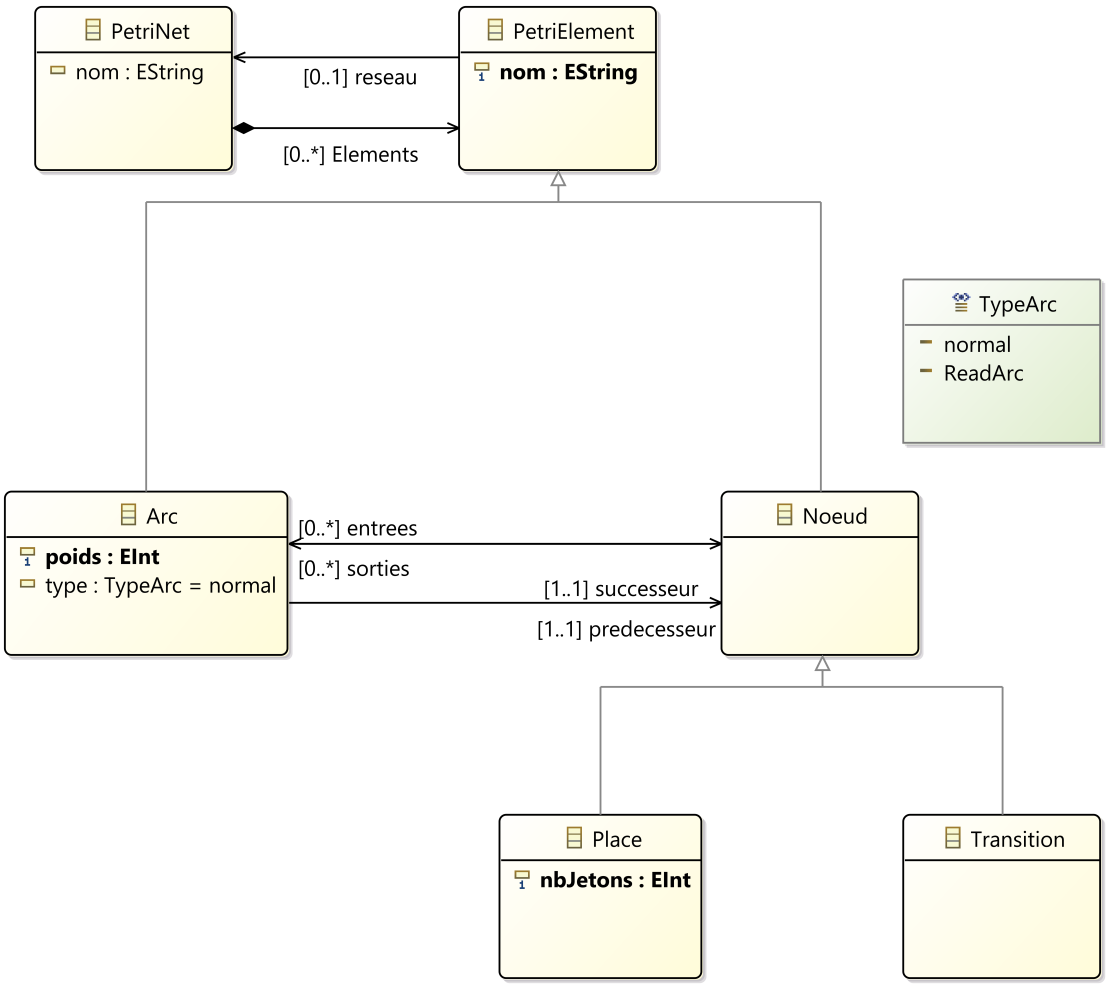


Figure 2: Métamodèle PetriNet

Nous allons alors détailler nos choix de réalisation en s'intéressant aux différentes classes choisies et aux relations entre elles :

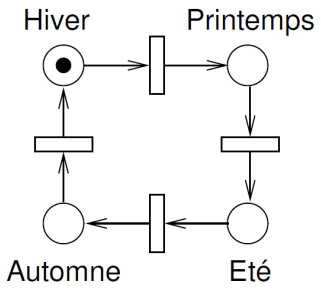
- **PetriNet :** Cette classe est à la racine de notre modèle. Elle a un unique attribut nom, qui permet de nommer notre réseau de Petri. Une relation de composition existe entre cette classe et la classe PetriElement. Un PetriNet peut être composé de 0 à "une infinité" de ProcessElement, d'où la multiplicité [0..*].
- **PetriElement :** Cette classe possède un unique attribut nom, qui permettra de nommer les éléments du réseau de Petri. Elle est super-classe des deux classes suivantes : Arc et Noeud.
- **Arc :** La classe Arc est sous-classe de ProcessElement. Elle possède deux attributs : poids, qui représente le poids d'un arc, et type, qui est de type **TypeArc**, qui est une classe Enumeration qui contient les deux types d'arc (Normal et ReadArc). Un arc de type ReadArc ne consommera pas de jetons, tandis qu'un arc de type Normal le fera. Deux relations d'association existent de Arc vers Noeud. Ces relations permettent de savoir quel Noeud est prédecesseur et lequel est successeur de l'Arc. Leur multiplicité est de 1 car un arc possède un prédecesseur et un successeur.
- **Noeud :** La classe Noeud est également sous-classe de ProcessElement. Elle ne possède pas d'attribut. Deux relations d'association existent de Noeud vers Arc. Elles permettent de savoir quels Arcs sortent du Noeud (sorties), et quels Arcs pointent vers le Noeud (entrées). Leur multiplicité est [0..*] car plusieurs arcs peuvent rentrer/sortir d'un même Noeud. Noeud est super-classe de Transition et de

Place. Nous avons choisi de définir cette classe Noeud pour factoriser les relations des Places et des Transitions avec les Arcs. En effet un Arc a les mêmes relations avec les Places et avec les Transitions.

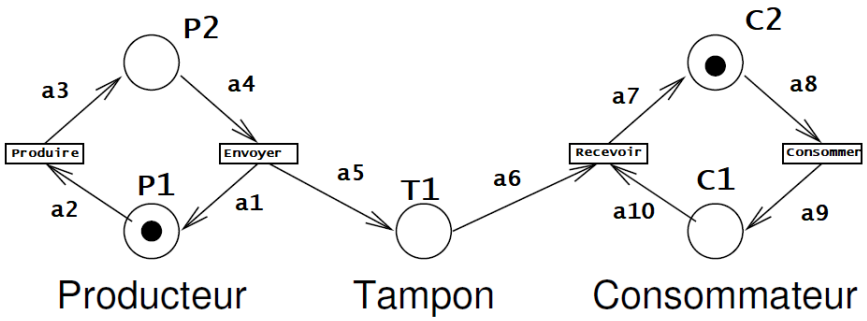
- **Transition :** Transition est sous-classe de la classe Noeud. Elle ne possède pas d'attribut.
- **Place :** Place est également sous-classe de la classe Noeud. Elle possède un attribut nbJetons correspondant au nombre de jetons présents dans la Place.

Pour valider que notre modélisation était correcte, nous avons créé plusieurs instances de PetriNet au format xmi. Ces fichiers peuvent aussi être récupérés dans le dossier livrables. Voici les différents exemples de PetriNet que nous avons instanciés :

- *net-saisons.xmi* : Représentation du cycle des saisons vu en TD1. Nous avons d'ailleurs utilisé cet exemple pour les transformations de Petrinet vers Tina (Section 2.8.2) et Petrinet vers Dot (Section 2.8.3).



- *net-producteur-consommateur.xmi* : Schéma Producteur-consommateur également vu au TD1. Les noms utilisés pour créer le xmi sont indiqués sur la figure ci-après.



- *ko-net-ocl.xmi* : Exemple qui met en évidence les erreurs liées aux contraintes OCL implémentées. (Section qui suit)

2.3 Définir les contraintes OCL pour capturer les contraintes qui n'ont pu l'être par les métamodèles

Les contraintes OCL (Object Constraint Language) permettent de définir les contraintes qui ne sont pas possible de modéliser en language Ecore. Elles permettent de contraindre syntaxiquement les noms des attributs des classes voire même de compléter la description structurelle du métamodèle par des contraintes exprimées en OCL. Elles sont principalement réalisées à l'aide d'invariants.

2.3.1 SimplePDL

Dans le fichier SimplePDL.ocl, on peut retrouver toutes les contraintes que nous avons écrites pour notre SimplePDL.ecore.

Par exemple, chaque classe avec un nom comme attribut possède l'invariant "hasName", qui force l'utilisateur

à attribuer un nom à son activité. De plus, pour les WorkDefinition, on retrouve la fonction "uniqNames" qui assure que chaque WorkDefinition a un nom différent.

Pour la classe Resources que nous avons rajoutée, nous avons écrit la contrainte resourceNonNegative qui stipule que le nombre d'occurrences d'une ressource est forcément positif ou nul.

Enfin, pour la classe linksResources, nous nous sommes inspirés de la fonction déjà implantée process() pour écrire la fonction wd(), qui recherche une activité WorkDefiniton. Nous l'utilisons pour vérifier que la WorkDefinition attribuée est bien celle à laquelle linksResources est attribuée.

La liste complète des contraintes se trouve dans le fichier correspondant.

2.3.2 PetriNet

De la même façon, on peut trouver la liste complète des contraintes OCL pour notre PetriNet.ecore dans le fichier PetriNet.ocl, en voici l'explication de certaines.

On retrouve l'invariant "hasName" pour les classes avec l'attribut "nom".

Le nombre de jetons d'une place ne peut pas être négatif donc l'invariant positiveToken est là pour le vérifier dans le contexte de la Place.

Enfin, "arcWeight" certifie que le poids de l'arc est strictement positif et "goodLink" vérifie que les liens entre les successeurs et les prédecesseurs sont corrects.

2.4 Donner une syntaxe concrète textuelle de SimplePDL avec Xtext

Xtext consiste à définir une syntaxe concrète textuelle au travers d'une grammaire qui doit pouvoir être analysée en LL(k). Pour définir les syntaxes concrètes textuelles de notre SimplePDL, nous avons donc utilisé cet outil et le résulat se trouve dans le fichier SimplePDL.Xtext

2.5 Dévelopeer un éditeur graphique SimplePDL pour saisir graphiquement un modèle de processus, y compris les ressources

À l'instar d'une syntaxe concrète textuelle, une syntaxe concrète graphique fournit un moyen de visualiser et/ou éditer plus agréablement et efficacement un modèle. Nous allons utiliser l'outil Eclipse Sirius développé par les sociétés Obeo et Thales, et basé sur les technologies Eclipse Modeling comme EMF et GMF 2. Il permet de définir une syntaxe graphique pour un langage de modélisation décrit en Ecore et d'engendrer un éditeur graphique intégré à Eclipse. Sirius permet de représenter des modèles graphiquement sous la forme d'un graphe composé de nœuds et d'arcs. Il est possible de choisir la forme d'un nœud (rectangle, ellipse, etc.) et la forme d'un arc (décoration à l'extrémité, tracé du trait, etc.).

En suivant les instructions du TP et en ajoutant nos classes "Resources" et "linksResources" nous obtenons le fichier simplepdl.odesgin.
A partir de ce fichier, nous pouvons créer un exemple de processus de SimplePDL de A à Z.
Avec l'outil WorkDefinitionCreation, nous pouvons créer une nouvelle WorkDefinion. De meme pour WorkSequenceCreation (à condition d'avoir deux WorkDefiniton). Cet outil permet de relier deux WorkDefinition entre elles, nous pouvons choisir ensuite le type de lien qui leur correspond.
GuidanceCreation permet de créer une note à partir de l'onglet Text, que nous avons la possibilité de remplir. Pour relier cette note à son activité, il suffit d'utiliser l'outil GuidanceEdgeCreation.
De même pour ajouter une ressource, ResourceCreation ajoute la ressource au processus et ResourceEdgeCreation permet de relier la ressource à l'activité, il suffit juste de rajouter les quantités nécéssaires.

La représentation graphique de l'exemple du sujet est trouvable en Figure 3.
Une représentation graphique de notre fichier test est visionnable en Figure 4.

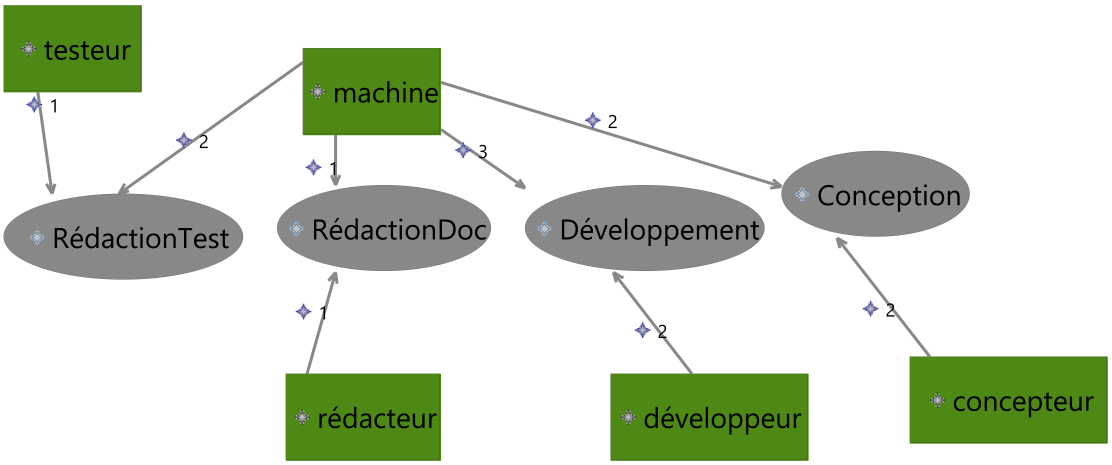


Figure 3: Représentation de pdl-sujet, l'exemple du sujet du projet

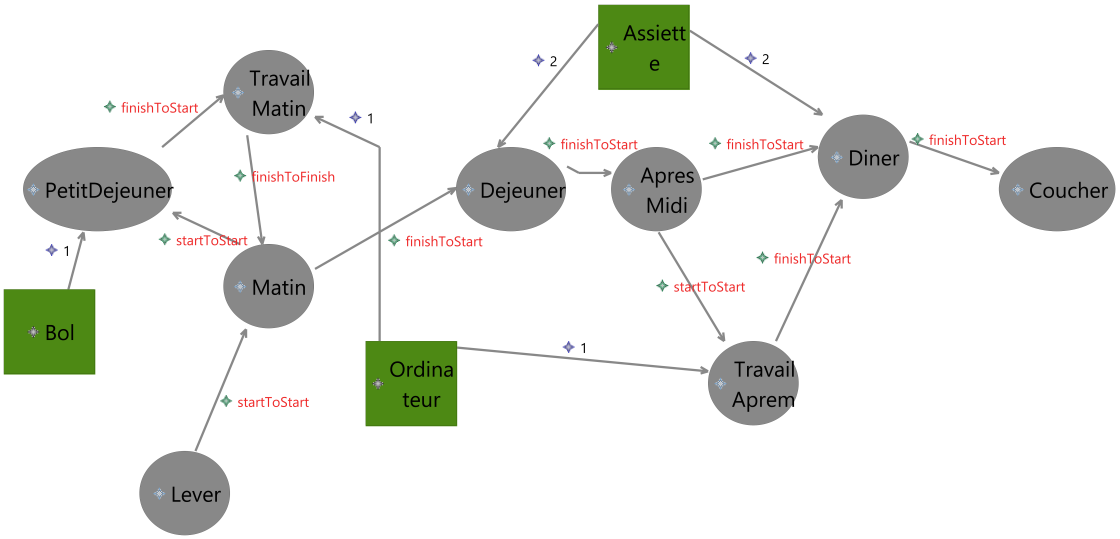


Figure 4: Représentation d'une journée banale

2.6 Définir une transformation SimplePDL vers PetriNet

2.6.1 En utilisant EMF/Java

Dans cette section, nous allons implanter un fichier Java qui transforme les modèles SimplePDL en PetriNet.

Voici l'explication de la structure du fichier :
- Chargement des packages SimplePDL et PetriNet et initialisation du fichier à convertir - Conversion des "WorkDefinition", des "WorkSequence" et des "Resources" en éléments du PetriNet soit 4 places, 2 transitions, et 5 arcs.

Nous avons choisi d'utiliser un HashMap pour stocker chaque élément créé au fur et à mesure de la création.

L'ensemble du code se trouve dans le fichier SimplePDL2PetriNet.java.

Pour tester ceci, nous avons d'abord créé un fichier pdl-journeeBanale.xmi, qui se base sur le modèle SimplePDL.ecore. Le résultat de l'éxécution Java est dans pdltopetrinet-journeeBanale.xmi.

Si vous souhaitez changer les fichiers sources et générés, ceci se passe ligne 56 et 57 du fichier.

2.6.2 En utilisant ATL

ATL (Atlas Transformation Language) est le langage de transformation développé dans le cadre du projet ATLAS. Il permet de convertir un modèle source en un modèle cible. Dans notre cas, un SimplePDL en PetriNet.

Dans un premier temps, nous traduisons un Process en un PetriNet de même nom.
Ensuite, nous traduisons les "WorkDefinition" en "Place" "Transition" ou "Arc", les WorkSequence en Arc que nous relions aux éléments de Petrinet correspondant aux "WorkDefiniton", les "Resources" en "Place" et enfin les "linksResources" en "Arc" que nous relions au éléments du PetriNet correspondant aux ressources.

Pour sauvegarder les éléments créés, nous utilisons la fonction thisModule.resolveTemp.

Pour tester, nous procédons de la même manière que le fichier java. Le fichier généré pour la journée banale est : pdltopetrinetATL-journeeBanale.petrinet, on remarque des les Transitions, Arcs et Places ne sont pas des childs du PetriNet, et nous n'arrivons pas à corriger ceci... Dans la suite de notre projet, nous privilegierons donc la transformation Java.

On remarque cependant que la syntaxe d'écriture et la compréhension du code est plus simple en ATL qu'en Java.

2.7 Valider la transformation SimplePDL vers PetriNet Java en faisant des tests

En passant le fichier pdltopetrinet-journeeBanale.xmi dans les contraintes OCL, la validation est satisfaite, le fichier généré respecte donc la syntaxe OCL.

2.8 Définir des transformations PetriNet ou SimpleDPL vers un autre modèle à texte en utilisant Acceleo

Acceleo est un outil pour créer des générateurs de code à partir d'un modèle EMF. Dans notre cas, nous allons l'utiliser pour transformer notre SimplePDL en format HTML, puis notre PetriNet en format net (tina) et Dot.

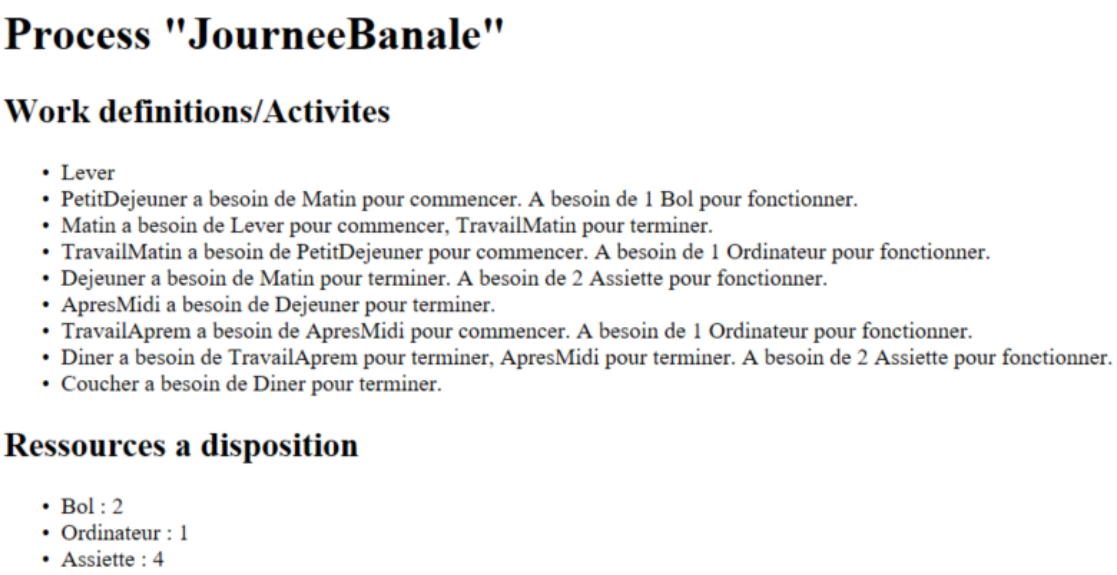
PS: Nous tenons à préciser que les fichiers ont été créés avec les noms "toFORMAT", c'est à dire que si par exemple le SimplePDLtoHTML.mtl ne fonctionne pas, il faut le modifier en toHTML.mtl et cela devrait fonctionner.

2.8.1 SimplePDL vers HTML

Le fichier toHTML nous etait fourni, nous avons rajouté la section avec les ressources et une phrase indiquant l'utilisation d'une ressource par une activité.

Le fichier à ouvrir pour cette section est donc SimplePDLtoHTML.mtl.

Voici ce que donne le fichier pour notre exemple d'une journée :



Process "JourneeBanale"

Work definitions/Activites

- Lever
- PetitDejeuner a besoin de Matin pour commencer. A besoin de 1 Bol pour fonctionner.
- Matin a besoin de Lever pour commencer, TravailMatin pour terminer.
- TravailMatin a besoin de PetitDejeuner pour commencer. A besoin de 1 Ordinateur pour fonctionner.
- Dejeuner a besoin de Matin pour terminer. A besoin de 2 Assiette pour fonctionner.
- ApresMidi a besoin de Dejeuner pour terminer.
- TravailAprem a besoin de ApresMidi pour commencer. A besoin de 1 Ordinateur pour fonctionner.
- Diner a besoin de TravailAprem pour terminer, ApresMidi pour terminer. A besoin de 2 Assiette pour fonctionner.
- Coucher a besoin de Diner pour terminer.

Ressources a disposition

- Bol : 2
- Ordinateur : 1
- Assiette : 4

Figure 5: Représentation d'une journée banale en HTML

2.8.2 PetriNet vers Tina

Pour la transformation d'un PetriNet en fichier .net utilisable sur Tina nous avons créés deux fonctions getTransitions et getPlaces pour pouvoir récupérer respectivement les transitions et les places d'un PetriElement. Nous utilisons ces deux méthodes dans le corps du template pour pouvoir écrire un fichier qui correspond bien à la syntaxe voulue par Tina.

Pour notre exemple des saisons :

```
|
net Saisons
        pl Hiver (1)
        pl Printemps (0)
        pl Ete (0)
        pl Automne (0)
        tr HP [Hiver*1] Hiver -> Printemps
        tr PE [Printemps*1] Printemps -> Ete
        tr EA [Ete*1] Ete -> Automne
        tr AH [Automne*1] Automne -> Hiver
```

Figure 6: Représentation des saisons en un fichier pour Tina

2.8.3 PetriNet vers Dot

De la même manière que pour les deux autres transformations, le code se trouve dans PetriNetToDot.mtl et on retrouve ci-dessous l'exemple des saisons.

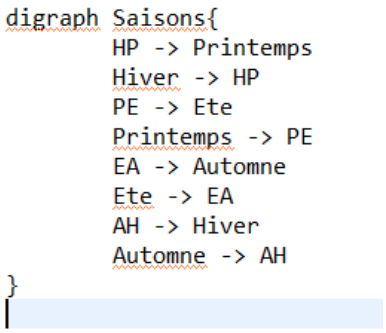


Figure 7: Représentation des saisons en un fichier Dot

2.9 Engendrer des propriétés LTL

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

Les propriétés LTL permettent de vérifier si le processus peut se terminer ainsi que de vérifier l'ordre dans lequel il faut commencer et terminer chaque activité du processus. Pour cela, nous nous sommes basés sur le TP1 et le cours et nous avons codé le fichier petrinetFinishLTL.mtl qui respect ces propriétés.

1. $[]\ (finished => dead)$: plus aucune action n'est possible lorsque le processus est terminé.
2. $[]\ <>\ dead$: plus aucune action à exécuter.
3. $[]\ (dead => finished)$: s'il n'y a plus d'action à réaliser, alors le processus se termine.
4. $-\ <>\ finished$: le processus ne s'est pas terminé.

2.9.2 Propriétés correspondant aux invariants de SimplePDL pour valider la transformation écrite

Pour valider la transformation ecrite, voici les propriétés à respecter, nous l'avons coder pour les PDL et pour les PetriNet sous les noms respectifs SimplePDLtoLTL.mtl et PetriNetToLTL.mtl :

1. $-\ <>\ finished$: le processus ne s'est pas terminé.
2. $[]\ (finished => -\ <>\ ready)$: une activité finie ne peut pas redémarrer.
3. $[]\ (finished => -\ <>\ running)$: une activité finie n'est pas en fonctionnement.
4. $[]\ (started => -\ <>\ ready)$: une activité commencée ne peut pas redémarrer.
5. $[]\ (finished => started)$: une activité terminée a forcement commencée un moment.

3 Mode d'emploi de la conversion à partir d'un SimplePDL : déroulement de l'oral

Pour ce mode d'emploi, nous baserons nos idées sur le déroulement d'une journée banale. Dans votre cas, si vous voulez tester avec un autre modèle, il faudra que vous vous adaptiez en fonction de vos noms de fichiers, mais le principe reste le même.

3.1 Conversion PDL vers PetriNet

Nous vous conseillons d'avoir fait au préalable un fichier .xmi qui déroule votre exemple PDL.
Dans notre cas, nous prenons le fichier pdl-journeeBanale.xmi (disponible dans les livrables).

Pour généré le fichier de notre exemple en version PetriNet, prenez les fichiers SimplePDL2PetriNet.java et SimplePDL2PetriNet.atl au choix (disponibles aussi dans les livrables) et executez-les. Pour un soucis de clareté, tous les fichiers que nous mentionneront dans la suite de cette demonstration seront disponibles dans les livrables.
Dans notre cas, les fichiers créés sont respectivement pdltopetrinet-journeeBanale.xmi et pdltopetrinetATL-journeeBanale.petrinet.
Si vous avez opté pour l'option Java, le fichier créé n'est pas un .petrinet mais un .xmi, veuillez le convertir en .petrinet pour suivre les étapes suivantes de ce mode d'emploi.

3.2 Conversion en Tina ou Dot

Maintenant que vous avez à disposition vos fichiers .petrinet, nous pouvons les convertir en .net.

Pour cela, executez le fichier PetriNetToTina.mtl ou PetriNetToDot.mtl (si vous avez des erreurs d'execution, cf la partie de ce rapport correspondant à ces sections). Dans notre cas, nous obtenons respectivement les fichiers JourneeBanale.net et JourneeBanale.dot.

3.3 Synthaxe LTL

Enfin, pour la dernière partie de ce mode d'emploi, vous aurez besoin du fichier de départ en .simplepdl. Prenez soin de le convertir si ce n'est pas dejà fait (nous avons dans notre cas SimplePDLtoLTL).

Executez le sur votre fichier .simplepdl (pdl-journeBanale.simplepdl) et obtenez votre fichier .ltl (JourneeBanale.ltl), vous pouvez aller le vérifier sur Tina.

4 Conclusion

Ce projet nous a permis de mieux comprendre, analyser et développer les méta-modèles et leurs transformations. Nous avons pu élaborer à partir des deux modèles Ecore/EMF, leurs nombreuses contraintes OCL, des syntaxes concrètes textuelles, un éditeur graphique permmetant de manipuler plus simplement les exemples de processus, les transformations modèle à texte (M2T) puis modèle à modèle (M2M) et enfin les vérifications LTL de ces processus.

Nous avons évidemment passé énormémement de temps sur ce projet, en essayant dans un premier temps de comprendre ce qui était demandé, puis ensuite de corriger les erreurs venant d'Eclipse sur nos machines. En revanche ce projet nous a permis de découvrir les vastes possibilités qu'offre ce logiciel, mais aussi sa limitation : par exemple la modification d'un élément du fichier Ecore entraine une modification de tous les autres fichiers déjà créés, en outre, se rendre compte d'un oubli en fin de projet nous a fait changé la majorité des fichiers déjà implanté, ce qui est assez dommage.

Enfin, ce projet nous a permis de nous rendre compte qu'il faut bien penser, au tout début du projet, à tout ce qui adviendra au fur et à mesure de son avancement, et dans notre cas nous avons eu assez de chance sur ce point.