Génie du Logiciel et des Systèmes : Chaîne de vérification de modèles de processus

Hajem Yassine et Adam Bouchiha

Département Sciences du Numérique - Deuxième année
2023-2024

Table des matières

1 Introduction **4**

2 Définition des métamodèles **5**
2.1 Définition du métamodèle de SimplePDL 5
2.2 Définition du métamodèle de PetriNet 5

3 Complétion des métamodèles par des contraintes statiques avec OCL **6**
3.1 Sémantique statique de SimplePDL 6
3.2 Sémantique statique de PetriNet 7

4 Syntaxes concrètes de SimplePDL **7**
4.1 Syntaxe concrète graphique avec Sirius 7
4.2 Syntaxe concrète textuelle avec XText 8

5 Transformations modèle à modèle (M2M) **11**
5.1 Principe de traduction de SimplePDL à PetriNet 11
5.2 Transformation modèle à modèle avec Java 12
5.3 Transformation modèle à modèle avec ATL 12

6 Transformations modèle à texte (M2T) avec Acceleo **12**
6.1 Définition d'une transformation PetriNet vers Tina 12
6.2 Génération des propriétés LTL permettant de vérifier la terminaison d'un processus . 12
6.3 Génération des propriétés LTL correspondant aux invariants de SimplePDL 13

7 Validations et Tests **13**
7.1 Tester la transformation SimplePDL vers PetriNet obtenue en utilisant EMF/Java et en utilisant ATL 14
7.2 Tester la transformation PetriNet vers Tina produite en utilisant Acceleo 14
7.3 Valider les résultats de la transformation M2M avec les propriétés LTL 14

8 Conclusion **16**

Table des figures

1 Schema de la chaine de production 4
2 Modèle SimplePDL 5
3 Modèle PetriNet 6
4 Syntaxe concrète graphique du SimplePDL 8
5 Modèle SimplePdlXtext 9
6 Exemple du syntaxe textuelle SimplePDLXtext 10
7 Transformation de modèle de la figure 6 en SimplePDL 10
8 Exemple de modèle de réseau de Petri obtenu après la transformation d'un modèle de SimplePDL 11
9 Modèle TestRapport 14
10 Résultat de la transformation M2M en utilisant EMF/Java ou ATL 14
11 Description textuelle de l'exemple en figure 10 15
12 Propriétés LTL vérifiant la terminaison du processus TestRapport 9 15
13 Propriétés LTL correspondant aux invariants SimplePDL pour TestRapport 9 . . . 15
14 Résultats de vérification des propriétés LTL (figure 12) 16

1 Introduction

L'objectif de notre 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 traduisons un modèle de processus en un réseau de Pétri, puis utilisons les outils de model-checking définis sur les réseaux de Petri au travers de la boîte à outils Tina.

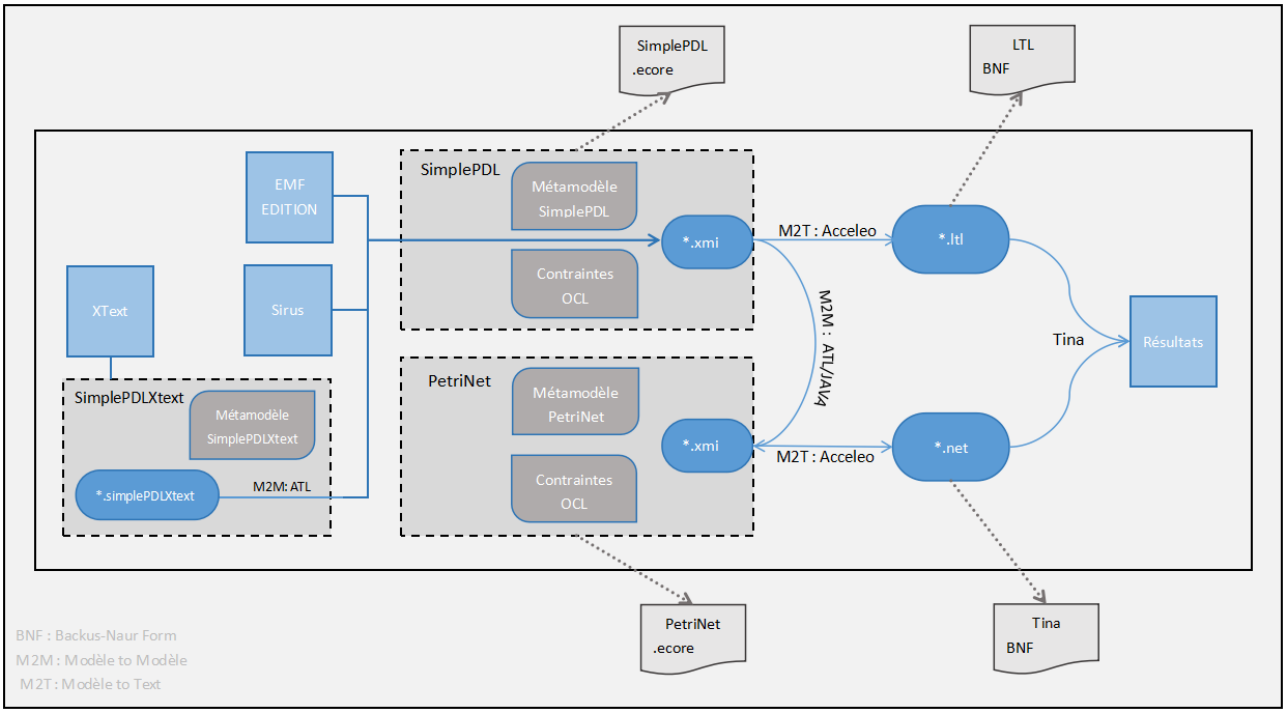

FIGURE 1 – Schema de la chaine de production

Nous vous proposons tout d'abord de décrire le principe de fonctionnement attendu pour la chaîne de vérification.

Il commence avec la génération d'un modèle de processus sur Eclipse au format ".xmi". Cela peut se faire grâce à l'éditeur Ecore d'Eclipse EMF, grâce à l'outil Sirius qui permet de définir une syntaxe concrète graphique de notre métamodèle, ou encore grâce à l'outil XText qui permet de définir une syntaxe concrète textuelle.

La syntaxe abstraite d'un métamodèle est définie sur Eclipse par un métamodèle Ecore (format ".ecore") et des contraintes OCL (format ".ocl"). Le modèle est donc soumis à une double vérification pour vérifier sa conformité à SimplePDL.

Ensuite, d'une part, on génère le modèle de réseau de Petri associé au modèle de processus, et on le soumet à une double vérification pour vérifier sa conformité à PetriNet. Puis on génère à partir de ce modèle un fichier ".net" via Acceleo, exploitable par Tina pour représenter un réseau de Petri.

D'autre part, on engendre avec Acceleo et à partir du modèle de processus, les propriétés LTL que l'on cherche à vérifier avec Tina sur le réseau de Petri obtenu (fichier ".net") exprimées dans des fichiers ".ltl". Ces propriétés sont de deux types. Il y a celles qui répondent à l'objectif premier de notre projet, c'est à dire vérifier la terminaison d'un processus. Mais il est aussi nécessaire d'engendrer les propriétés LTL correspondant aux invariants de SimplePDL pour valider la transformation de modèle.

L'outil selt de Tina permet de vérifier ces propriétés et de conclure sur la terminaison du modèle de processus.

Dans ce rapport, nous vous présentons notre travail effectué pour remplir l'objectif du projet.

2 Définition des métamodèles

2.1 Définition du métamodèle de SimplePDL

Sur Eclipse, nous avons défini le métamodèle de SimplePDL à l'aide de l'éditeur arborescent Ecore d'EMF. Il est défini dans le fichier "SimplePDL.ecore".

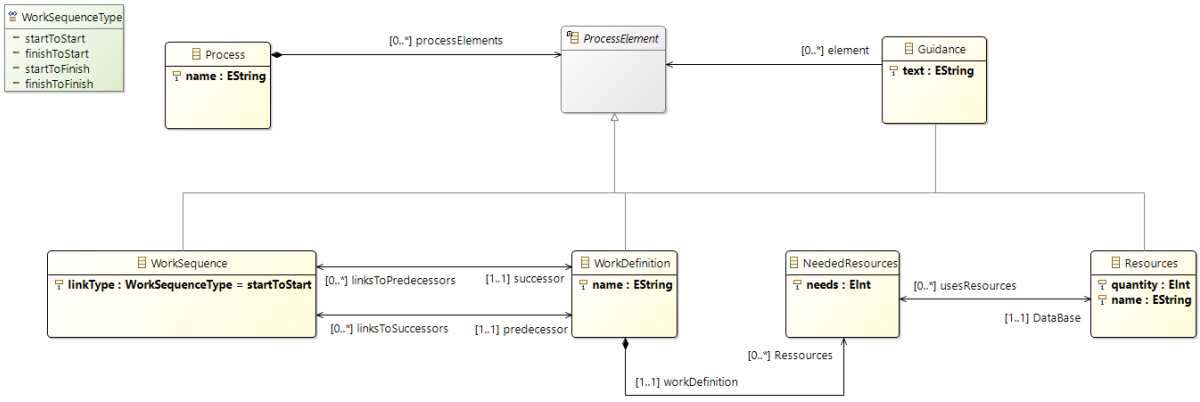

FIGURE 2 – Modèle SimplePDL

Nous avons défini l'élément Process afin qu'il soit l'élément racine des modèles engendrés. La notion de ProcessElement permet de généraliser les fils de l'élément racine d'un modèle. Un processus est donc un ensemble d'éléments de processus qui sont soit des activités (WorkDefinition), soit des dépendances (WorkSequence), soit des ressources (Resources), soit des guidances (Guidance). D'où la relation de composition entre Process et ProcessElement.

Une WorkSequence définit une dépendance entre deux WorkDefinitions (predecessor et successor). Cependant, une WorkDefinition peut dépendre d'un nombre quelconque de WorkDefinition (éventuellement 0), et un nombre quelconque de WorkDefinition peuvent dépendre d'elle (éventuellement 0). Cela se traduit par les deux relations bidirectionelles entre WorkDefinition et WorkSequence sur le diagramme.

WorkSequenceType est défini pour indiquer le type d'une dépendance symbolisé par l'attribut linkType d'une WorkSequence.

Une Guidance traduit une information sur un élément du processus.

L'élément Resources traduit la notion de ressource, dont peuvent avoir besoin les activités pour s'effectuer. Chaque ressource a un nom (attribut "name") et est présente en une quantité fixe (attribut "quantity").

Cependant, la notion de ressource est incomplète sans information sur les besoins en ressource des activités. C'est pour cela que nous avons ajouté l'élément NeededResources. Une WorkDefinition a une NeededResources associée par ressource nécessaire. Cela se traduit par une relation de composition entre WorkDefinition et NeededResources. Chaque instance de NeededResources contient les informations sur la demande exprimée, c'est-à-dire quelle ressource est concernée (relation "DataBase") et en quelle quantité (attribut "needs").

2.2 Définition du métamodèle de PetriNet

Sur Eclipse, nous avons défini le métamodèle PetriNet à l'aide de l'éditeur arborescent Ecore d'EMF. Il est défini dans le fichier "PetriNet.ecore".

Nous avons définit l'élément PetriNet afin qu'il soit l'élément racine des modèles engendrés. Node permet de généraliser les éléments Place et Transition (qui correspondent aux deux types de noeuds d'un réseau de Petri).

Enfin, l'élément Arc sert à représenter un arc d'un réseau de Petri. Il est en relation avec son noeud source et son noeud cible. Un attribut indique son poids et un autre indique son type (simple ou read-arc) défini dans ArcType.

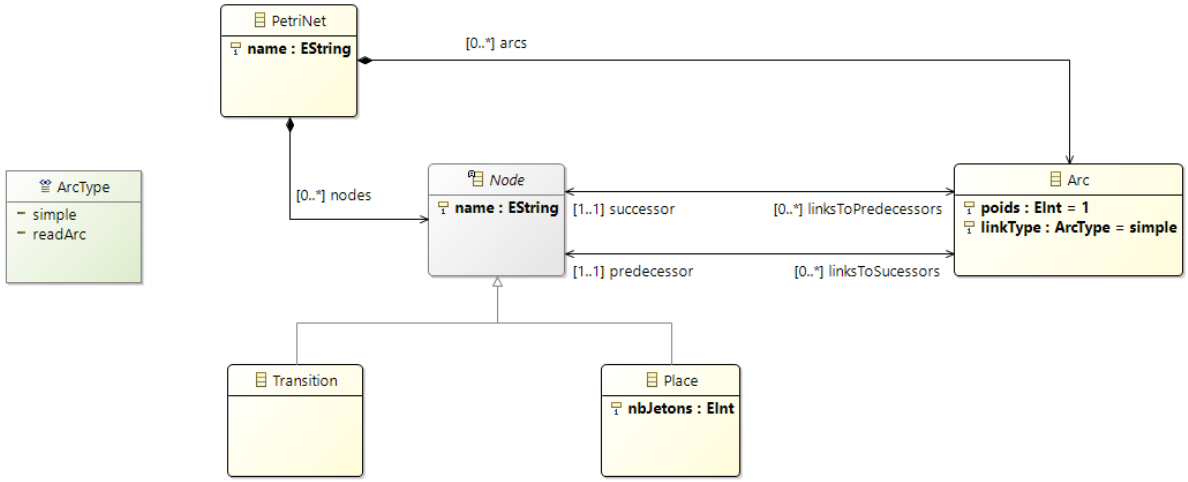

FIGURE 3 – Modèle PetriNet

Etant donné que les places et les transitions d'un réseau de Petri ont un nom, on peut mettre cet attribut commun dans Node. Et comme ces éléments ont aussi tous deux une liste d'arcs entrants ainsi qu'une liste d'arcs sortant, on a fait figurer ces relations avec des arcs dans Node, ce qui donne les flèches bidirectionelles entre Node et Arc sur le diagramme.

3 Complétion des métamodèles par des contraintes statiques avec OCL

Nous avons défini des contraintes OCL pour capturer les contraintes qui n'ont pu l'être par les deux métamodèles créés (SimplePDL et PetriNet).

3.1 Sémantique statique de SimplePDL

Les contraintes figurent dans le fichier OCLSimplePDL.ocl.

Nous avons défini une requête process() applicable sur un élément ProcessElement, qui renvoie le Process auquel il appartient. Cela est utile dans la suite, pour pouvoir depuis un ProcessElement, accéder aux autres ProcessElement du Process. Par exemple si une WorkSequence veut obtenir la liste des WorkSequence de son Process, elle appelle cette requête et obtient la liste de ProcessElement de son Process, puis ne garde que les éléments qui sont des WorkSequence dans cette liste.

Nous avons défini des contraintes pour s'assurer de la validité des noms des Process, des WorkDefinition et des Resources. Leurs noms doivent comporter une lettre ou un underscore, puis au moins une lettre, un chiffre ou un underscore. Pour Resources et WorkDefinition, nous avons aussi ajouté des contraintes pour garantir que deux Resources ou deux WorkDefinition ne portent pas le même nom, car c'est par leurs noms qu'on les distingue.

Nous avons défini la contrainte nonReflexiveDependency pour garantir qu'une WorkSequence n'est pas réflexive (une activité ne dépend pas d'elle même), et la contrainte noIdenticalWS qui permet de s'assurer qu'une WorkSequence n'apparaît pas deux fois dans un Process. La contrainte successorAndPredecessorInSameProcess permet de s'assurer qu'une WorkSequence relie deux WorkDefinition d'un même Process. Cette contrainte nous a été fournie en TP. Nous nous sommes questionnés sur l'utilité d'une telle contrainte car nous ne manipulons que des modèles avec un seul Process. Mais un défaut de programmation dans la définition des syntaxes concrètes pourrait donner lieu à des définitions erronées de modèles ne vérifiant pas cette contrainte. En toute rigueur, il faudrait aussi vérifier que tous les ProcessElement liés entre eux appartiennent au même Process.

Et aussi vérifier qu'une NeededResources lie une WorkDefinition et une Resources qui figurent dans le même Process. Pour éviter d'alourdir le code, nous n'avons pas écrit ces autres contraintes

La contrainte quantityPositif sert à s'assurer qu'une ressource est présente en quantité strictement positive. Il serait inutile de faire apparaître une ressource sans occurrence. Nous avions défini, quand nous débutions notre projet, un invariant quantityNotOverused. Nous voulions ainsi garantir qu'une ressource n'est jamais utilisée plus qu'elle ne peut l'être par les activités. Nous avons ensuite réalisé que nous n'avions pas à essayer de capturer une telle contrainte avec OCL. Nous avons décidé de laisser cet invariant en commentaire, car il illustre bien le fait qu'OCL permet seulement de vérifier des propriétés statiques, d'où l'intérêt de transformer notre modèle pour bénéficier des outils de Tina.

Nous avons défini 3 contraintes pour une NeededResources. La première sert à s'assurer que l'attribut "needs" est strictement positif, sinon l'élément n'apporterait pas d'information. La seconde sert à s'assurer que les besoins d'une activité en une certaine ressource ne soient pas exprimés dans plusieurs NeededResources, mais dans une seule. Nous avons pensé à cette contrainte vers la fin de notre projet. Elle tient son intérêt du principe de transformation de SimplePDL à PetriNet qui sera détaillé dans la partie 5.1, et du fait qu'une place et une transition ne peuvent pas être reliées par deux arcs ayant la même orientation. Un autre intérêt est d'éviter la dispersion d'informations dans le modèle. La troisième permet de garantir que la demande exprimée dans NeededResources ne dépasse pas le nombre total d'occurrences de la ressource en question. Il est évident qu'un modèle ne respectant pas cette contrainte ne peut pas terminer.

3.2 Sémantique statique de PetriNet

Les contraintes figurent dans le fichier OCLPetriNet.ocl.

Nous avons défini une requête petrinet() applicable sur un élément Node, qui renvoie le PetriNet auquel il appartient. Cela est utile pour exprimer certaines contraintes car cela permet depuis un Node d'accéder aux autres Node du PetriNet. Nous avons défini une requête similaire pour les Arcs.

Nous avons défini des contraintes pour s'assurer de la validité des noms des PetriNet et des Nodes. Comme les noms des éléments de SimplePDL, ils doivent comporter une lettre ou un underscore, puis au moins une lettre, un chiffre ou un underscore. Pour les Node, nous avons aussi ajouté une contrainte pour garantir que deux Node ne portent pas le même nom. Dans la transformation modèle à texte de PetriNet vers Tina, les noeuds créés portent le nom de l'élément Noeud correspondant dans le modèle. Or une place et une transition ne peuvent pas avoir le même nom sous Tina. Nous avons donc écrit cette contrainte pour nous conformer à Tina.

Une contrainte exprime le fait qu'une place contient un nombre de jetons positif ou nul.

Le reste des contraintes porte sur les arcs. Un contrainte sert à s'assurer qu'un arc ne lie pas deux places ou deux transitions. Une autre sert à s'assurer que le poids de l'arc est bien strictement positif conformément à la définition d'un arc dans un réseau de Petri. Une dernière contrainte permet de garantir que des arcs qui relient deux même Node n'ont pas la même orientation.

4 Syntaxes concrètes de SimplePDL

4.1 Syntaxe concrète graphique avec Sirius

Une syntaxe concrète graphique fournit un moyen de visualiser et/ou éditer agréablement et efficacement un modèle. Nous utilisons l'outil Eclipse Sirius1 développé par les sociétés Obeo et Thales, et basé sur les technologies Eclipse Modeling comme EMF et GMF2. Il permet de définir une syntaxe graphique pour un langage de modélisation décrit en Ecore et d'engendrer un éditeur graphique intégré à Eclipse.

À partir de notre métamodèle de SimplePDL (SimplePDL.ecore) on définit,dans le fichier " SimplePDL.design", la syntaxe concrète graphique souhaitée sous la forme d'un graphe composé de nœuds et d'arcs.

Pour visualiser les éléments de nos modèles dans l'éditeur graphique, il est nécessaire de définir leur représentation graphique. Lors de la conception d'un éditeur graphique, il peut être difficile de maintenir la compréhension du modèle lorsque le nombre d'éléments à afficher devient important.

Sirius propose une solution à ce problème en introduisant l'utilisation de calques. Un calque est responsable de l'affichage des éléments graphiques spécifiques. Cette fonctionnalité permet de gérer la complexité visuelle en offrant la flexibilité de choisir quels éléments afficher à un moment donné, contribuant ainsi à une meilleure compréhension du modèle affiché.

Dans une palette, on définit ensuite les outils nécessaires pour manipuler le modèle au travers des objets graphiques de cette vue. Ces outils sont regroupés en deux sections :

- Section Elements : cette partie de la palette offre les moyens pour la création des différents éléments : les WorkDefinitions, les Resources et les Guidance.
- Section Links : Dans cette section, on dispose les outils pour la mise en place des relations entre les éléments comme "NeededResources" qui relie les WorkDefintions et les resource nécessaires et "WorkSequences" qui représente la relation entre deux WrokDefinition.

Dans la figure 4, on peut voir les deux sections de la palette d'outils. On utilise la palette pour la création d'un modèle simplePDl. Le WorkDefinition "Programmation" par exemple utilise trois unités des ressources "Ordinateurs". Pour arriver à ce résultat, on a suivi les étapes suivantes :

1. avec Elements > WDCreation, on crée un WorkDefinition qu'on nomme Programmation.
2. avec Elements > ResourceCreation, on crée une Resource Ordinateurs et on lui donne une quantité de 5.
3. avec Links > NeededResourcesCreation, on établit la relation de Programmation avec Ordinateurs et on définit la quantité dont il a besoin.

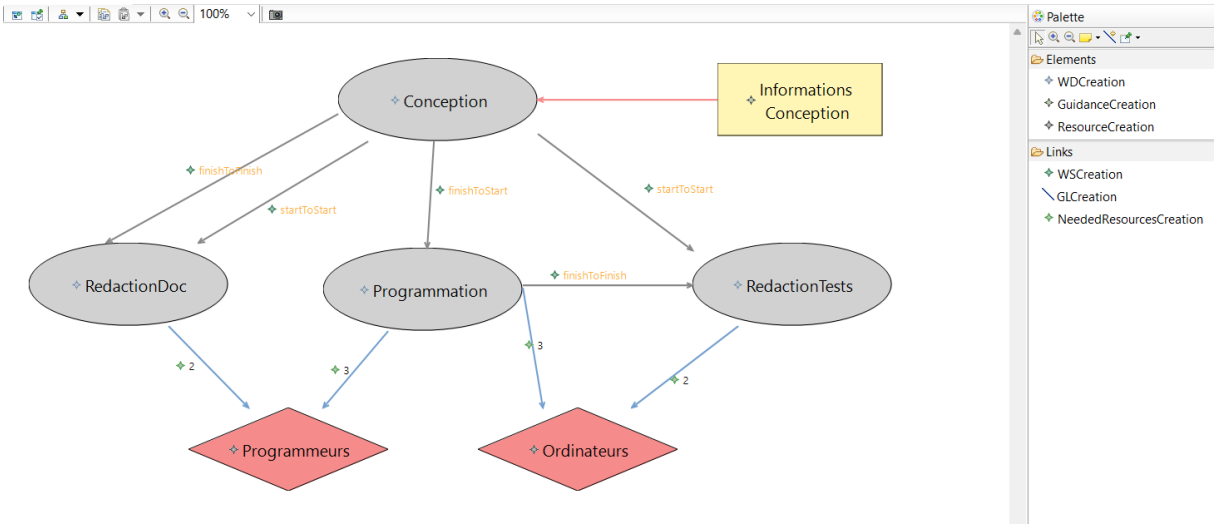

FIGURE 4 – Syntaxe concrète graphique du SimplePDL

4.2 Syntaxe concrète textuelle avec XText

L'outil Xtext permet de définir une syntaxe concrète textuelle d'un métamodèle qui lui appartient. Grâce à Xtext on génère un nouveau métamodèle "SimplePDLXtext", qui va beaucoup ressembler à celui de "SimplePdl", et sur lequel on peut appliquer une transformation modèle à modèle pour obtenir un modèle de SimplePdl. Nous avons défini la syntaxe textuelle concrète de SimplePDLXtext dans le fichier "SimplePDLXtext.xtext".

Le facteur de départ est l'élément "Process", qui encapsule les caractéristiques essentielles d'un processus, notamment son nom et les différents éléments qui le composent "processElements" puis "neededResources". La définition d'un process est reconnue par le mot clef "Process : " suivi de son nom.

Les "NeededResources" définissent le besoin d'une workDefintion en une ressource, ils sont définis par le mot "NeededResources : " suivi par son nom , la quantité dont la workDefinition aura besoin et les ressources en question.

L'abstraction fournie par l'élément "ProcessElement" simplifie la définition du processus en incluant les aspects clés tels que les "workdefinitions", les "worksequences", les "Guidance" et les "NeededRessources".

Le segment "WorkDefinition" est identifié de manière précise par un nom (ID) et des listes éventuellement vides de prédécesseurs, successeurs et ressources.

De même, la section "WorkSequence" offre la possibilité de définir les relations de dépendance entre ces tâches par : "Worksequence : " suivie de son nom, son prédécesseur, son type et son successeur.

La gestion des ressources, élément fondamental dans de nombreux processus, est traitée de manière exhaustive par les sections "Ressources".

Enfin, la section dédiée aux indications ("Guidance") offre la possibilité d'ajouter des notes informatives, facilitant la documentation et la compréhension du processus.

L'ensemble de cette grammaire Xtext contribue à la création d'un langage formel, permettant une modélisation précise (voir figure 5).

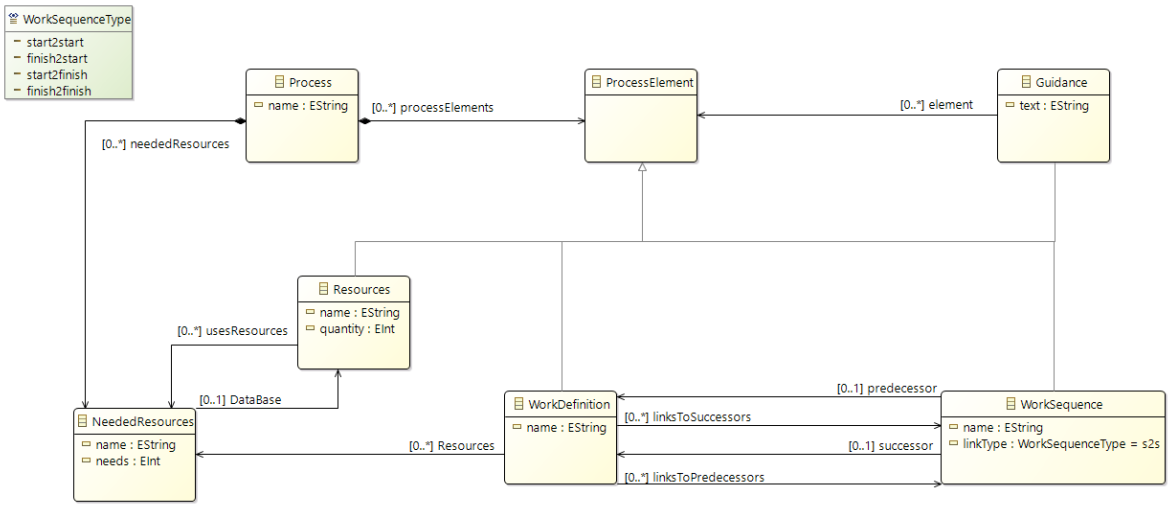

FIGURE 5 – Modèle SimplePdlXtext

L'eclipse de déploiement lancée ensuite reconnait le langage de "SimplePDLXtext" et on a l'exemple dans la figure 6.

Pour qu'on puisse à partir de ce modèle graphique SimplePDLXtext passer à un modèle SimplePdl, on a ajouté dans le dossier "ATL" dans l'eclipse de déploiement une transformation M2M de SimplePDLXtext vers SimplePdl sous le nom "Xtext2SimplePDL.atl" (voir partie 5.3).

Ce fichier permet de générer à partir de notre exemple "exemple.simplePDLXtext" un fichier de style "exemple.simplepdl" (voir figure 7) compatible avec notre modèle SimplePdl.

```
Process MyProcess {
    processElements {
        WorkDefinition TaskA {
            linksToPredecessors( wsBA)
        },
        WorkDefinition TaskB {
            linksToSuccessors( wsBA , wsBC)
            Resources ( nrb )
        },
        WorkDefinition TaskC {
            linksToPredecessors (wsBC)
            Resources ( nrc )
        },
        WorkSequence wsBA{
            predecessor TaskB
            linkType s2s
            successor TaskA
        },
        WorkSequence wsBC{
            predecessor TaskB
            linkType s2f
            successor TaskC
        },
        Resources ResourceB {
            quantity 10
            usesResources ( nrb )
        },
        Guidance {
            text "Guidance pour TaskA et TaskB"
            element (TaskA,TaskB)
        },
        NeededResources nrc{
            needs 5
            DataBase ResourceB
        },
        NeededResources nrb{
            needs 5
            DataBase ResourceB
        }
    }
}
```

FIGURE 6 – Exemple du syntaxe textuelle SimplePDLXtext

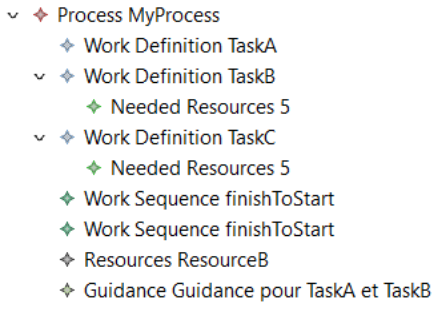

FIGURE 7 – Transformation de modèle de la figure 6 en SimplePDL

5 Transformations modèle à modèle (M2M)

5.1 Principe de traduction de SimplePDL à PetriNet

Avant de définir une transformation modèle à modèle, il faut se demander comment se traduit chaque élément du modèle d'origine.

On veut, depuis un modèle de processus, obtenir un modèle de réseau de Petri correspondant à l'état initial du processus.

Une activité A1 peut être dans trois états. On doit donc créer trois places pour indiquer l'état de l'activité : A1_ready, A1_running et A1_finished. Mais il faut aussi une place indiquant si l'activité a déjà commencé (A1_started) afin que les activités qui dépendent du fait que A1 ait commencé puissent savoir quand c'est le cas. La place A1_ready est la seule à comporter un jeton car cela correspond à l'état initial. Il faut ensuite lier ces places entre elles, cela se fait naturellement en ajoutant deux transitions A1_start et A1_finish et des arcs simples de poids 1.

Une WorkSequence traduit une dépendance. Si A2 dépend de A1, on ajoutera un read_arc de poids 1 qui part de A1_started si A1 doit avoir commencé ou de A1_finished si A1 doit avoir fini, et qui pointe vers A2_start si c'est pour que A2 commence, ou vers A2_finish si c'est pour que A2 termine.

Une ressource se traduit par une place, comportant comme nombre de jetons le nombre d'occurrences de cette ressource. Le nombre de jetons présents dans cette place représentera le nombre d'occurrences disponibles de cette ressource.

Une NeededResources se traduit par deux arcs. Si A1 a besoin de n ressources R, on a un arc de poids n allant de la place R à A1_start (emprunt de ressources), et un arc de poids n allant de A1_finish à R (restitution des ressources).

Le modèle ci-dessous correspond à la transformation en réseau de Petri d'un modèle de Processus comprenant deux activités A1 et A2, où A2 requiert que A1 ait commencé pour finir, et où A1 a besoin de deux ressources de R, qui compte 3 occurences, pour s'effectuer.

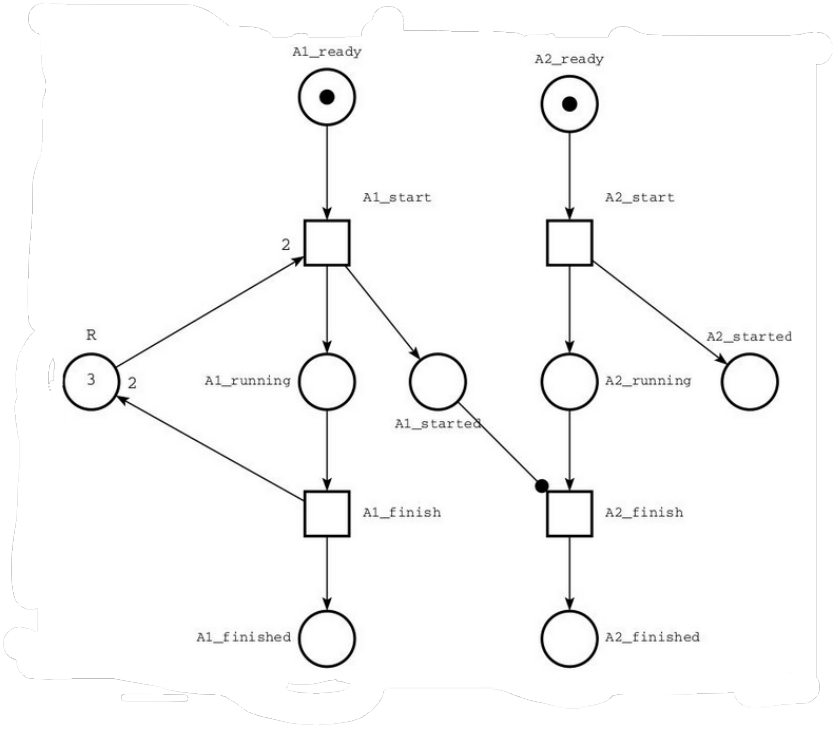

FIGURE 8 – Exemple de modèle de réseau de Petri obtenu après la transformation d'un modèle de SimplePDL

5.2 Transformation modèle à modèle avec Java

Nous avons défini dans un premier temps la transformation M2M de SimplePDL vers PetriNet avec EMF/Java. Nous avons utilisé une bibliothèque permettant de lire et produire des informations dans des fichiers au format ".xmi". Le programme Java définissant cette transformation est écrit dans le fichier "SimplePDL2PetriNet.java". Ce programme suit le principe de transformation décrit dans la partie 5.1. Il est nécessaire de respecter un certain ordre dans le parcours des informations du modèle cible. Par exemple, pour traduire une WorkSequence, il faut déjà avoir créé au préalable les places liées aux deux WorkDefinition qu'elle lie.

5.3 Transformation modèle à modèle avec ATL

Nous avons défini une deuxième transformation M2M de SimplePDL vers PetriNet, en utilisant ATL. Elle est définie dans le fichier "SimplePDL2PetriNet.atl". ATL, en tant que langage spécialisé pour les transformations de modèles, facilite l'extraction et la production d'information des modèles. Le principe d'ATL est de définir des règles, chacune transformant un motif du modèle source en un motif du modèle cible, tout en conservant un lien de traçabilité entre eux. Ces liens de traçabilités et les liens entre les motifs sources, sont exploités pour faciliter la définition de l'assemblage des motifs cibles dans l'écriture des règles. Le code de la transformation ATL s'est donc avéré moins fastidieux à écrire que celui du programme Java.

6 Transformations modèle à texte (M2T) avec Acceleo

6.1 Définition d'une transformation PetriNet vers Tina

Nous avons créé une template Acceleo pour effectuer une transformation modèle à texte (M2T) d'un modèle de réseau de Petri au format ".xmi" à son fichier ".net" associé exploitable avec Tina.

Cette template se trouve dans le fichier "toTina.mtl".

Pour effectuer la transformation, nous commençons par traduire une à une, ligne par ligne, les places du réseau de Petri. Nous avons créé pour cela une requête qui renvoie les places du réseau de Petri sous forme de liste. La ligne résultant de la place considérée dans le parcours de la liste requiert simplement les attributs de la place. Ensuite, nous traduisons une à une et ligne par ligne les transitions. Nous avons donc créé une requête qui renvoie les places du réseau de Petri sous forme de liste. Pour chaque transition de la liste, nous récupérons son nom, et nous appelons une template qui parcourt les arcs entrants de la transition, puis une template qui parcourt les arcs sortants.

6.2 Génération des propriétés LTL permettant de vérifier la terminaison d'un processus

Pour vérifier la terminaison d'un processus, nous utilisons Tina, avec son outil selt, qui permet de vérifier des propriétés exprimées en LTL (Logique Temporelle Linéaire) sur un réseau de Pétri. Les propriétés LTL doivent être définies dans un fichier au format ".ltl".

Nous avons donc eu recours à une template Acceleo pour pouvoir engendrer les propriétés de terminaison d'un modèle de processus au format ".xmi" dans un fichier au format ".ltl".

La template est définie dans le projet simplePDL.toTerminaisonLTL.

Le fichier à engendrer est très court. Il ne doit comporter que 3 lignes, quel que soit le modèle de processus considéré.

La première ligne définit l'opérateur "fin" correspondant à un état du réseau de Petri associé dans lequel chaque activité a terminé (toutes les places "A_finished", où A désigne une activité, comportent un jeton). Dans le cas où le processus ne comporte pas d'activité, nous remplaçons la formule de l'opérateur fin par le mot-clef "dead". Ce mot-clef traduit un état dans lequel le réseau de Petri ne peut pas évoluer S'il n'y a pas d'activité, alors c'est dans cet état que l'on est censé se trouver en permanence.

La seconde ligne exprime une propriété P1 vraie si et seulement si le processus peut toujours terminer (on peut toujours atteindre l'état fin). La troisième exprime une propriété P2 vraie si et seulement si le processus ne peut jamais terminer (on ne peut jamais atteindre l'état fin).

Ces deux propositions sont suffisantes pour répondre aux questions que l'on se pose sur la terminaison d'un processus.

On peut se demander si le processus finit toujours. Cela est vrai si P1 est vraie, et si ce n'est pas le cas, selt nous renvoie un exemple de processus qui ne termine pas.

Mais on peut aussi se demander s'il y a au moins une évolution possible du processus lui permettant de se terminer. Dans ce cas, on s'intéresse à la propriété P2. Si elle renvoie FALSE, alors c'est le cas, et selt nous donne un exemple de processus qui termine.

6.3 Génération des propriétés LTL correspondant aux invariants de SimplePDL

Afin de valider la transformation SimplePDL vers PetriNet, on souhaite vérifier que les invariants sur le modèle de processus sont préservés sur le modèle de réseau de Petri correspondant.

Pour cela, nous avons créé une template Acceleo dans le projet simplePDL.toTerminaisonLTL, permettant d'engendrer les propriétés de terminaison d'un modèle de processus au format ".xmi" dans un fichier au format ".ltl". Nous avons fait le choix d'engendrer deux fichiers distincts pour les tâches 10 et 11 afin de pouvoir effectuer les vérifications séparément.

Dans le fichier engendré, on définit le même opérateur "fin" que dans la tâche précédente. Il traduit l'état dans lequel un process est terminé.

Ensuite figurent les différents invariants :

- un invariant pour s'assurer que le système n'évolue plus si on atteint l'état final
- un invariant par activité, pour s'assurer que activité ne peut pas être indiquée comme n'ayant pas commencé et comme ayant commencé simultanément
- un invariant par activité pour s'assurer que chaque activité est soit non commencée, soit en cours, soit terminée
- un invariant par activité pour s'assurer qu'une activité terminée n'évolue plus
- un invariant par activité pour s'assurer qu'une activité ayant commencé sera toujours indiquée comme ayant commencé
- un invariant par ressource pour s'assurer que si une ressource R est en une certaine quantité initialement, alors cette même quantité doit être présente une fois le processus terminé
- un invariant par ressource pour s'assurer que si une ressource présente en une certaine quantité initialement, elle est présente en cette même quantité à tout instant dans le processus (on considère pour les occurences d'un type de ressource la somme de celles utilisées et de celles inutilisées)

Nous nous sommes limités à cela. Nous pensons qu'il pourrait aussi être utile de rajouter des propositions vérifiant que l'état initial du réseau de Petri est correct, car cela est possible avec Tina. Il serait aussi intéressant de pouvoir exprimer une contrainte qui impose l'ordre d'évolution entre les états d'une actvité (A_ready puis A_running puis A_finished), mais cela n'est pas possible avec les opérateurs LTL dont nous disposons.

7 Validations et Tests

Afin de valider notre chaîne de vérification, nous avons testé ses différents éléments avec plusieurs modèles de processus. On commençait par exemple par vérifier leur bon fonctionnement pour des processus ne contenant que des activités, puis on y ajoutait des dépendances entre activités, des ressources... Quand nous définissions des contraintes que les modèles devaient respecter, nous vérifiions aussi qu'elles relevaient bien les cas d'erreur.

Nous vous proposons de retracer en partie ces étapes, avec un exemple de modèle de processus conforme.

7.1 Tester la transformation SimplePDL vers PetriNet obtenue en utilisant EMF/Java et en utilisant ATL

Prenons par exemple un modèle SimplePDL "TestRapport.xmi". Le modèle est le suivant :

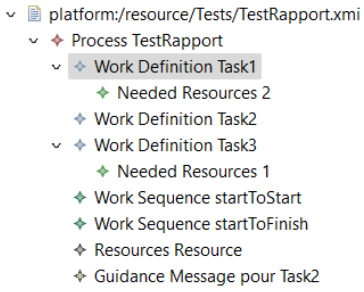

FIGURE 9 – Modèle TestRapport

Après s'être assuré que ce modèle est bien conforme au métamodèle SimplePDL et aux contraintes OCL, on lui applique les transformations par EMF/Java et par ATL. On obtient dans les deux cas les résultats présentés dans la figure 10

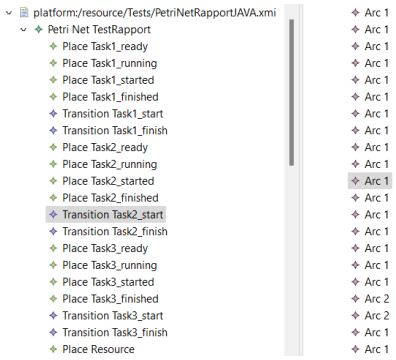

FIGURE 10 – Résultat de la transformation M2M en utilisant EMF/Java ou ATL

On observe bien que pour chaque WorkDefinition WD, on crée 4 places de PetriNet : WD_ready , WD_running, WD_started et WD_finished et 2 transitions : WD_start et WD_finish. On peut voir aussi la place pour la ressource "Ressource" et les différents arcs qui relient les places et les transitions. On vérifie que le modèle est conforme à PetriNet et aux contraintes OCL associées.

7.2 Tester la transformation PetriNet vers Tina produite en utilisant Acceleo

Sur le résultat obtenu précédemment (figure 10), on applique la transformation du PetriNet vers Tina avec Acceleo et on obtient sa syntaxe textuelle dans la figure 11

7.3 Valider les résultats de la transformation M2M avec les propriétés LTL

Pour l'exemple de Tina obtenu précédemment, on a généré avec nos transformations M2T à partir du modèle 9 les propriétés LTL correspondantes. Les résultat obtenus sont affichés dans les figures 12 et 13.

```
net TestRapport
pl Task1_ready (1)
pl Task1_running (0)
pl Task1_started (0)
pl Task1_finished (0)
pl Task2_ready (1)
pl Task2_running (0)
pl Task2_started (0)
pl Task2_finished (0)
pl Task3_ready (1)
pl Task3_running (0)
pl Task3_started (0)
pl Task3_finished (0)
pl Resource (5)
tr Task1_start Task1_ready Resource*2 -> Task1_running Task1_started
tr Task1_finish Task1_running -> Task1_finished Resource*2
tr Task2_start Task2_ready Task1_started?1 -> Task2_running Task2_started
tr Task2_finish Task2_running -> Task2_finished
tr Task3_start Task3_ready Resource -> Task3_running Task3_started
tr Task3_finish Task3_running Task2_started?1 -> Task3_finished Resource
```

FIGURE 11 – Description textuelle de l'exemple en figure 10

```
op fin = Task1_finished /\ Task2_finished /\ Task3_finished;

<> fin; TRUE si le processus termine forcément

- <> fin; TRUE si le processus ne peut pas terminer
```

FIGURE 12 – Propriétés LTL vérifiant la terminaison du processus TestRapport 9

```
op fin = Task1_finished /\ Task2_finished /\ Task3_finished;

[] (fin => dead);

[] (Task1_ready <=> -Task1_started);
[] (Task2_ready <=> -Task2_started);
[] (Task3_ready <=> -Task3_started);

[] (Task1_ready + Task1_running + Task1_finished =1);
[] (Task2_ready + Task2_running + Task2_finished =1);
[] (Task3_ready + Task3_running + Task3_finished =1);

[] (Task1_finished => [] Task1_finished);
[] (Task2_finished => [] Task2_finished);
[] (Task3_finished => [] Task3_finished);

[] (Task1_started => [] Task1_started);
[] (Task2_started => [] Task2_started);
[] (Task3_started => [] Task3_started);

[] (fin => Resource = 5);

[] (Resource + 2 * Task1_running + 1 * Task3_running = 5);
```

FIGURE 13 – Propriétés LTL correspondant aux invariants SimplePDL pour TestRapport 9

Pour garantir qu'on a les bons résultats, on a verifié ces propriétés avec la commande selt. On obtient TRUE pour tous les invariants. On peut alors s'intéresser à la terminaison du processus. Nous voyons qu'il finit toujours. On obtient (voir figure 14) dans la première ligne un TRUE qui indique que le processus se termine toujours, et la seconde renvoie FALSE vu que le processus peut

finir et nous montre un contre-exemple de notre supposition.

```
Selt version 3.7.5 -- 03/29/23 -- LAAS/CNRS
ktz loaded, 18 states, 31 transitions
0.000s

- source TestRapport_terminaison.ltl;
operator fin : prop
TRUE
FALSE
  state 0: L.scc*17 Resource*5 Task1_ready Task2_ready Task3_ready
  -Task1_start->
  state 1: L.scc*15 Resource*3 Task1_running Task1_started Task2_ready Task3_ready
  -Task1_finish->
  state 2: L.scc*7 Resource*5 Task1_finished Task1_started Task2_ready Task3_ready
  -Task2_start->
  state 3: L.scc*5 Resource*5 Task1_finished Task1_started Task2_running Task2_started Task3_ready
  -Task2_finish->
  state 4: L.scc*2 Resource*5 Task1_finished Task1_started Task2_finished Task2_started Task3_ready
  -Task3_start->
  state 5: L.scc Resource*4 Task1_finished Task1_started Task2_finished Task2_started Task3_running Task3_started
  -Task3_finish->
  state 6: L.dead Resource*5 Task1_finished Task1_started Task2_finished Task2_started Task3_finished Task3_started
  -L.deadlock->
  state 7: L.dead Resource*5 Task1_finished Task1_started Task2_finished Task2_started Task3_finished Task3_started
  [accepting all]
0.000s
```

FIGURE 14 – Résultats de vérification des propriétés LTL (figure 12)

8 Conclusion

Nous avons consacré beaucoup de temps à ce projet. Nous dûmes comprendre le fonctionnement d'Eclipse Modeling Framework et parfois gérer certains bugs. Il arrivait aussi que l'on se rende compte après coup d'erreurs ou de cas non fonctionnels auxquels nous n'avions pas pensé en écrivant nos fichiers sources, ce qui nous a convaincu de l'utilité des outils vérifiant automatiquement la conformité. Tout ce temps consacré à ce projet nous a permis de bien assimiler les notions vues en cours et en TP. Nous sommes donc contents d'avoir mené ce projet à son terme, et nous sommes très satisfaits de la chaîne de vérification que nous avons obtenue, fonctionnelle de bout en bout.