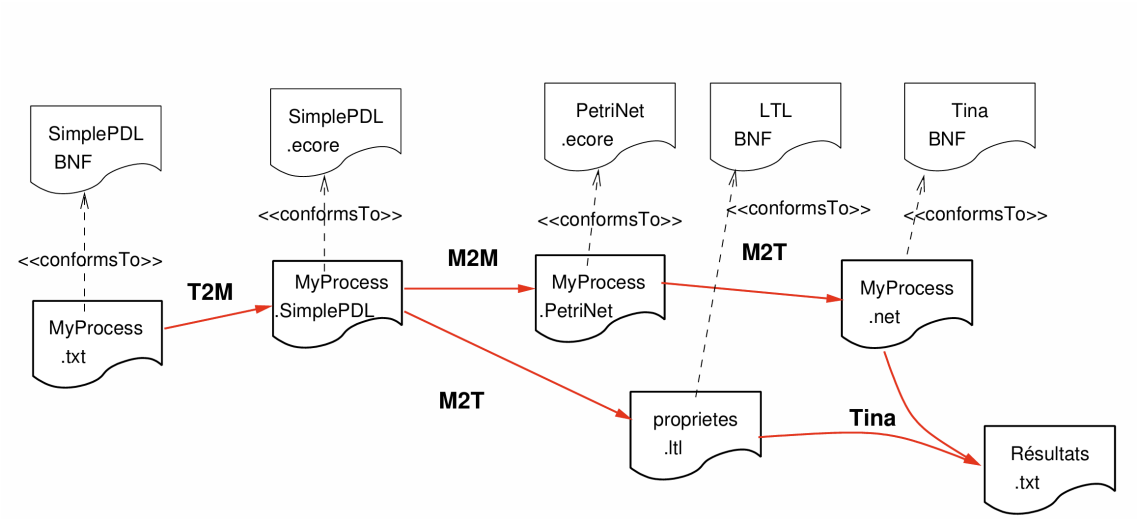
Chaîne de vérification de modèles de processus

Mini Projet en Ingénierie Dirigée par les Modèles



Assala Assellalou
Nour El Houda Mhamedi

Master 1 HPC et Big Data

22 Octobre 2024

Table des matières

1 Introduction

Ce mini-projet a pour objectif de développer une chaîne de vérification pour les modèles de processus SimplePDL, afin d'évaluer leur cohérence et notamment de déterminer si le processus peut se terminer. Pour ce faire, nous utiliserons les outils de model-checking basés sur les réseaux de Petri, en nous appuyant sur la boîte à outils Tina. Nous serons donc chargés de traduire le modèle de processus en un réseau de Petri.
Les principales étapes étant les suivantes :

1. Définition des métamodèles avec Ecore.
2. Définition de la sémantique statique en Java.
3. Définition de syntaxes concrètes graphiques avec Sirius.
4. Définition de syntaxes concrètes textuelles avec Xtext.
5. Définition d'une transformation de modèle à modèle avec Java et avec ATL.
6. Définition de transformations modèle à texte avec Acceleo
7. Définition des propriétés LTL

2 Les métamodèles SimplePDL et PetriNet

2.1 Qu'est-ce qu'un métamodèle ?

Un métamodèle est un modèle qui définit la structure, les contraintes et les règles d'un ensemble de modèles. En d'autres termes, il s'agit d'un modèle de haut niveau qui décrit les éléments et les relations d'un domaine spécifique, permettant ainsi de spécifier comment les modèles concrets doivent être construits.

Les métamodèles sont souvent utilisés dans le cadre de l'ingénierie dirigée par les modèles (IDM) pour assurer la cohérence et la validité des modèles générés. Ils fournissent un cadre permettant de comprendre et de manipuler les concepts d'un domaine, facilitant ainsi la création d'outils, de langages et de méthodes adaptés.

Dans notre contexte, un métamodèle peut inclure des éléments tels que des classes, des attributs, des associations et des règles d'intégrité. Par exemple, dans le cas des modèles de processus, un métamodèle pourrait définir les différentes étapes d'un processus, les acteurs impliqués, et les flux de contrôle qui les relient.

L'objectif de cette partie est de compléter le métamodèle SimplePDL pour prendre en compte les ressources, ainsi que de définir le métamodèle PetriNet.

2.2 SimplePDL

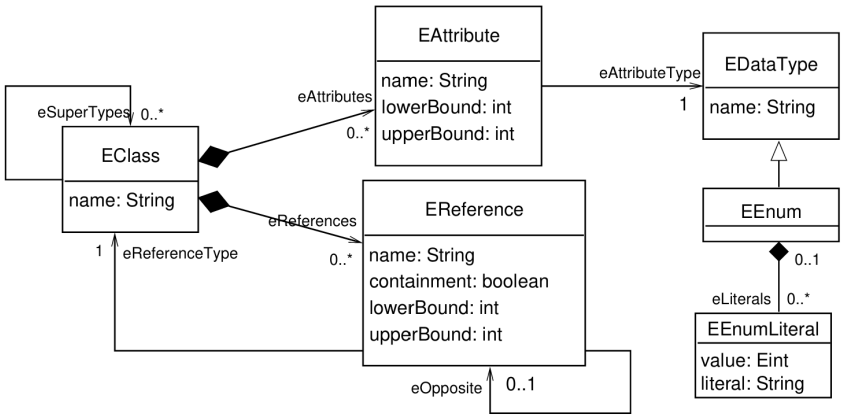
2.2.1 Définition

SimplePDL (Simple Process Description Language) est un langage utilisé pour décrire les processus de manière formelle. Il existe deux versions de SimplePDL, une version simplifiee et une version conforme a EMOF/Ecore.

2.2.2 Version simplifiée

La version simplifiée de SimplePDL est conçue pour être intuitive et accessible, permettant aux utilisateurs de modéliser des processus sans nécessiter une connaissance approfondie. Elle se concentre sur les éléments essentiels des processus, ce qui facilite la création et la compréhension des modèles.

FIGURE 1 – Version très simplifiée du méta-métamodèle Ecore



2.2.3 Version conforme à EMOF/Ecore

La version conforme à EMOF (Eclipse Modeling Framework) et Ecore est une extension de SimplePDL qui offre une représentation plus formelle et rigoureuse des processus. Cette version respecte les standards EMOF/Ecore, permettant une intégration plus facile avec d'autres outils et langages de modélisation. Elle inclut des fonctionnalités avancées pour la définition des métamodèles, la validation des modèles, et la génération de code, offrant ainsi une plus grande flexibilité et puissance dans la modélisation des processus.

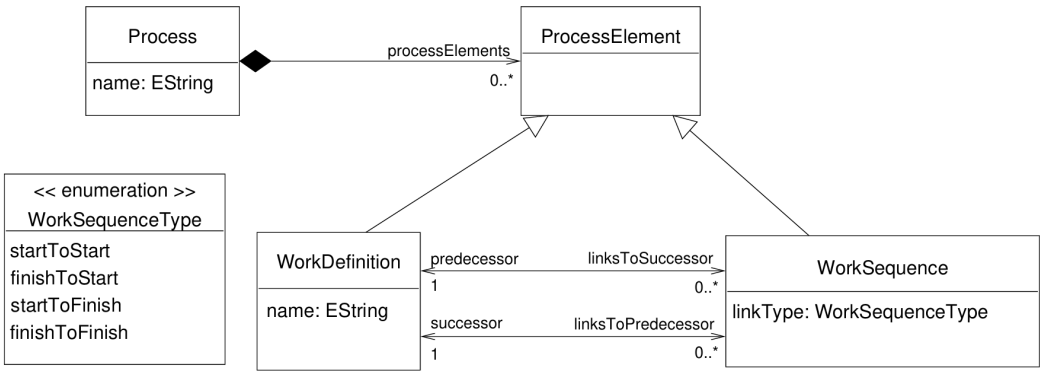


FIGURE 2 – Méta-modèle de SimplePDL conforme à EMOF/Ecore

2.2.4 Syntaxe abstraite (Ecore)

Une première étape consistait à construire le métamodèle en partant des processus et des éléments qui les composent.

Les processus sont modélisés par la classe *Process*, tandis que les éléments d'un processus sont représentés par la classe abstraite *ProcessElement*. Ces éléments peuvent inclure des activités, qui sont spécifiquement modélisées par la classe *WorkDefinition*, ou les dépendances entre ces activités, qui sont représentées par la classe *WorkSequence*.

Pour enrichir le métamodèle SimplePDL, il est essentiel d'intégrer des ressources. , nous avons créé une classe *Ressource avec deux attributs :* **name** *(nom de la ressource) et* **quantity** *(quantité disponible)* qui hérite de la classe abstraite ProcessElement car elle représente un élément du processus. De plus, nous avons introduit une classe dite *Occurrence* possédant un attribut **nb_occurence** (nombre de ressources nécessaires pour une activité). Cette classe *Occurrence* a été conçue comme une sous-classe de *Ressource* afin d'éviter que deux activités utilisent la même ressource simultanément.

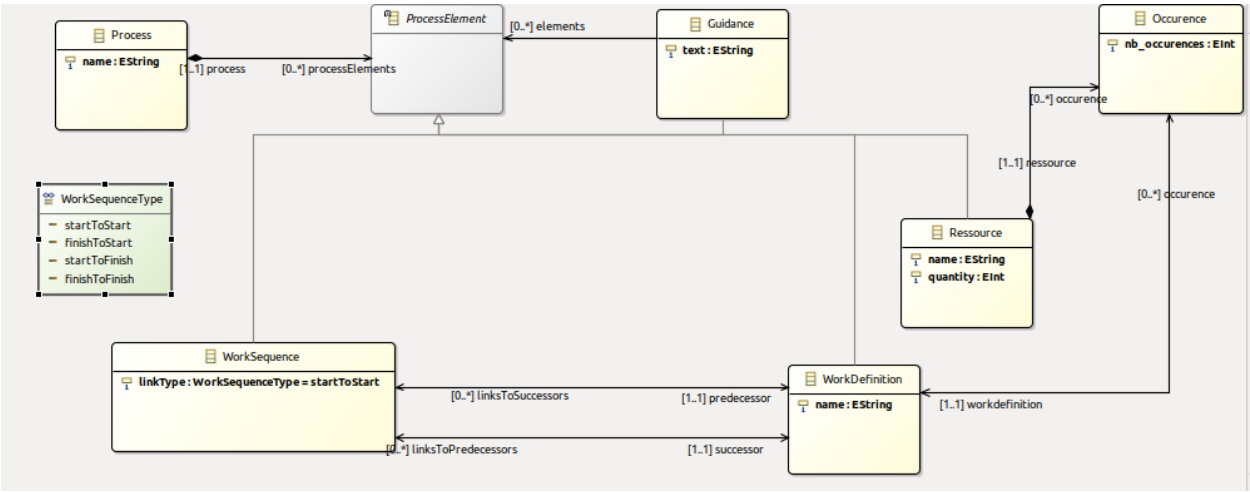


Figure 3 : Le métamodèle SimplePDL avec ressources

2.2.5 Points clés

L'intégration des ressources dans SimplePDL constitue une amélioration significative, car elle permet de mieux représenter les contraintes et les besoins des activités au sein des processus. Cependant, cette intégration peut également complexifier le modèle, en particulier en ce qui concerne la gestion des contraintes OCL/Java . L'ajout de ressources et d'occurrences implique de définir des règles supplémentaires pour garantir que les ressources sont utilisées de manière cohérente.

Pour illustrer ces concepts de manière concrète, nous nous baserons sur un exemple pratique dans les sections suivantes, ce qui nous permettra de démontrer l'impact de l'intégration des ressources sur la modélisation des processus.

2.3 Réseaux de Petri

Les réseaux de Petri sont des modèles mathématiques utilisés pour représenter des systèmes concurrents et dynamiques.Ils permettent de modéliser des processus en montrant comment les différentes activités interagissent les unes avec les autres.

Un réseau de Petri se compose de places, de transitions et d'arcs qui relient ces éléments, formant ainsi un graphe orienté. Les places peuvent contenir des jetons, qui représentent l'état ou les ressources disponibles dans le système. En exécutant des transitions, les jetons se déplacent d'une place à une autre, illustrant ainsi les évolutions du système au fil du temps.

2.3.1 Définition mathématique

Un réseau de Petri est un tuple (S, T, F, M_0, W) où :

- S définit une ou plusieurs **places**.
- T définit une ou plusieurs **transitions**.
- F définit un ou plusieurs **arcs** (flèches).
- Un arc ne peut pas être connecté entre deux places ou deux transitions; plus formellement : $F \subseteq (S \times T) \cup (T \times S)$.
- $M_0 : S \to N$ est appelé **marquage initial** (ou place initiale), où, pour chaque place $s \in S$, il y a $n \in N$ jetons.
- $W : F \to N^+$ est appelé **ensemble des arcs pondérés**, assignant à chaque arc $f \in F$ un entier positif $n \in N^+$ qui indique combien de jetons sont consommés depuis une place vers une transition, ou combien de jetons sont produits par une transition et arrivent pour chaque place.

De nombreuses définitions formelles existent. Cette définition concerne un réseau place-transition (ou P-T).

2.3.2 Représentation graphique

Un réseau de Petri se représente par un graphe orienté composé d'arcs reliant des places et des transitions. Deux places ne peuvent pas être reliées entre elles, ni deux transitions. Les places peuvent contenir des jetons. La distribution des jetons dans les places est appelée le marquage du réseau de Petri.

Les entrées d'une transition sont les places desquelles part une flèche pointant vers cette transition, et les sorties d'une transition sont les places pointées par une flèche ayant pour origine cette transition.

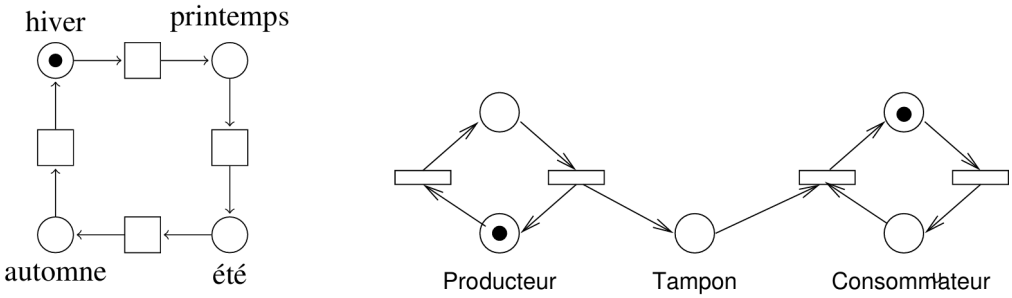


Figure 3 : Exemple de Réseaux de Petri

On a choisit que le metamodele *PetriNet* soit composé des classes PetriNet, Place, Transition, Arc; et d'une enumeration des types d'arcs et des directions de l'arc. Les details des liaisons entre les classes appraissent dans la figure suivante.

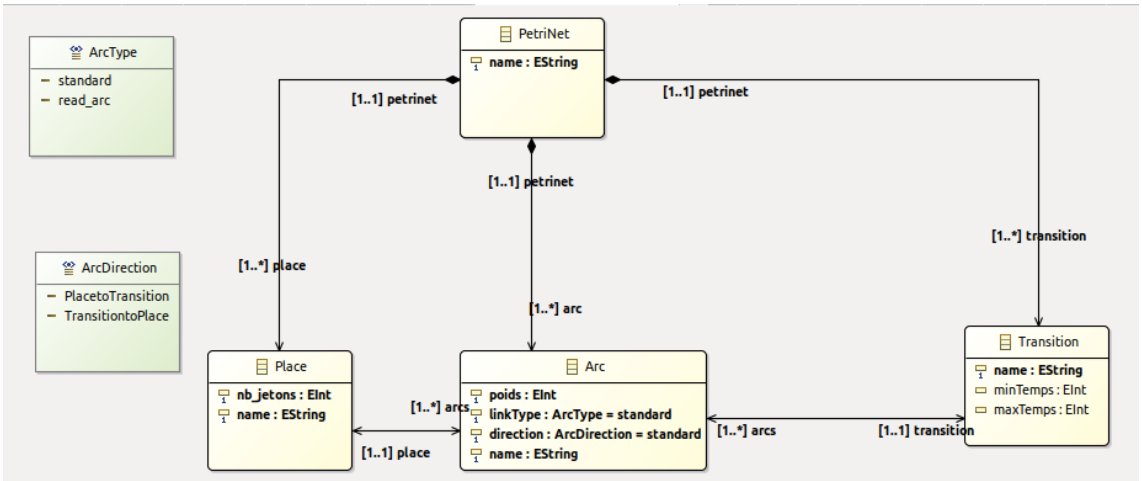


Figure 4 : Editeur Graphique ECore : Métamodèle PetriNet

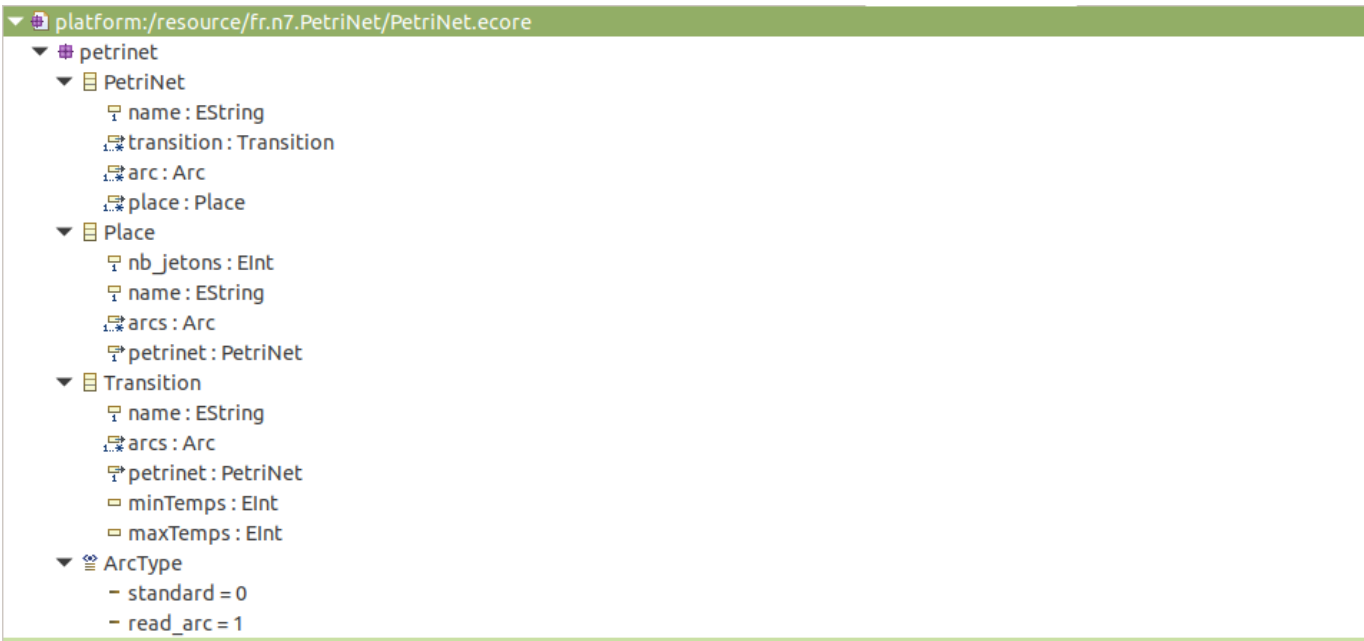


Figure 5.1 :Editeur Arborescent ECore : PetriNet

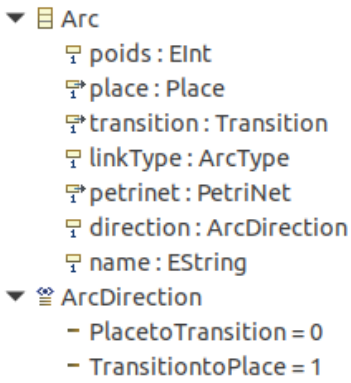


Figure 5.2 : Editeur Arborescent ECore : PetriNet

3 Sémantique statique

Pour assurer la cohérence et la validité des modèles, cette partie porte sur la spécification etla vérification des invariants qui ne sont pas directement pris en charge par la structure des métamodèles.

Plutôt que d'utiliser l'OCL (Object Constraint Language), nous avons ete contraint de choisir Java qui s'avère important pour notre formation.

Les vérifications ont été donc implémentées sous forme de méthodes Java qui valident les modèles en s'assurant que les contraintes définies sont respectée en les utilisant dans les tests automatiques pour détecter rapidement les incohérences.

3.1 SimplePDL

3.1.1 Tests SimplePDL

Ces exemples couvrent différents scénarios qui permettent de tester les contraintes de validation de notre classe SimplePDLValidator. Ils incluent des cas où les noms sont incorrects, où les dépendances sont mal formées, où les ressources ont des problèmes, et où les guidances sont incorrectes.

1. **Exemple 1 : Processus Simple avec des Noms Valides**

- **Nom du Process** : Processvalidite
- **WorkDefinition 1** : a1
- **WorkDefinition 2** : a2
- **WorkSequence** a1 → a2 (startToStart)

Ce modèle est correct et passe la validation.

2. **Exemple 2 : Noms de WorkDefinition Non Conformes**

- **Nom du Process** : Nominvalide
- **WorkDefinition 1** : Activite1
- **WorkDefinition 2** : 123Invalide
- **WorkSequence** : Activite1 → *123Invalide* (startToStart)

Ce modèle échoue à la validation en raison du nom invalide de la deuxième WorkDefinition.

3. **Exemple 3 : Dépendances Réflexives**

- **Nom du Process** : p1
- **WorkDefinition 1** : a1
- **WorkSequence** : a1 →a1 (finishToStart)

Ce modèle échoue car une activité ne peut pas dépendre delle-même.

4. **Exemple 4 : Ressource avec Quantité Négative**

- **Nom du Process** : p1
- **WorkDefinition 1** : a1
- **Ressource** : r1 avec quantité -1

Ce modèle échoue à cause de la ressource qui a une quantité négative.

5. Exemple 5 : Ressource Partagée Simultanément

- **Nom du Process** : p1
- **WorkDefinition 1** : a1
- **WorkDefinition 2** : a2
- **Ressource** : r1 partagée par a1 et a2 simultanément
- **Occurence** de r1 utilisée par a1 quantité 1
-**Occurence** de r1 utilisée a2 quantité 1

Ce modèle échoue à cause du partage simultané de la ressource entre deux activités.

6. Exemple 6 : Processus Valide avec Guidance

- **Nom du Process** : p1
- **WorkDefinition 1** : a1
- **Guidance** : avec un texte "Suivez les instructions"

Ce modèle est correct et passe la validation.

Exemple 7 : Process avec Guidance Vide

Nom du Process : p1
WorkDefinition 1 : a1
Guidance : avec un texte vide

Ce modèle échoue à la validation à cause du texte vide dans la guidance.

3.2.2 Résultats des Tests

```
WARNING: Using incubator modules: jdk.incubator.foreign, jdk.incubator.vector
Résultat de validation pour dependancereflexive.xmi:
- Process: OK
- WorkDefinition: OK
- WorkSequence: 1 erreurs trouvées
=> Erreur dans simplepdl.impl.WorkSequenceImpl@17046283 (linkType: finishToStart): La dépendance relie l'activité a1 à elle-même
- Guidance: OK
Résultat de validation pour Guidancesanstexte.xmi:
- Process: OK
- WorkDefinition: OK
- WorkSequence: OK
- Guidance: 1 erreurs trouvées
=> Erreur dans simplepdl.impl.GuidanceImpl@2667f029 (text: ): Une Guidance doit avoir un texte non vide
Résultat de validation pour Nominvalide.xmi:
- Process: 1 erreurs trouvées
=> Erreur dans 123 [simplepdl.impl.ProcessImpl@67a20f67 (name: 123)]: Le nom du process ne respecte pas les conventions Java
- WorkDefinition: OK
- WorkSequence: OK
- Guidance: OK
Résultat de validation pour Processusavecguidance.xmi:
- Process: OK
- WorkDefinition: OK
- WorkSequence: OK
- Guidance: OK
Résultat de validation pour Processvalidite.xmi:
- Process: OK
- WorkDefinition: OK
- WorkSequence: OK
- Guidance: OK
Fini.
```

Figure 6 : Resultat des Tests SimplePDL

3.2 PetriNet

3.2.1 Tests PetriNet

Voici quelques exemples de modèles PetriNet que nous utilisons pour tester et vérifier que la classe PetriNetValidator fonctionne correctement. Chaque exemple couvre différents scénarios possibles afin de vérifier toutes les règles que nous avons mises en place dans la validation.

Exemple 1 : Modèle valide

Ce modèle doit passer la validation sans erreurs.

- **Place 1** : P1 (2 jetons)
- **Place 2** : P2 (5 jetons)
- **Transition 1** : T1 (minTemps = 1, maxTemps = 5)
- **Arc 1** : De P1 vers T1 (direction PLACETOTRANSITION)
- **Arc 2** : De T1 vers P2 (direction TRANSITIONTOPLACE)

Exemple 2 : Modèle avec jetons négatifs (Place invalide)

Cet exemple contient une place avec un nombre de jetons négatif, ceci génère une erreur.

- **Place 1** : P1 (-3 jetons)
- **Place 2** : P2 (5 jetons)
- **Transition 1** : T1 (minTemps = 0, maxTemps = 2)
- **Arc 1** : De P1 vers T1
- **Arc 2** : De T1 vers P2

Exemple 3 : Transitions sans place en entrée ou en sortie (Transition invalide)

Cet exemple contient une transition sans place en entrée ou en sortie, ceci génère des erreurs.

- **Place 1** : P1 (3 jetons)
- **Transition 1** : T1 (sans place en sortie)
- **Transition 2** : T2 (sans place en entrée)
-**Arc** : T1toP1
-**Arc** : P1toT2

Exemple 4 : Noms de Place et Transition non uniques

Dans cet exemple, deux places et deux transitions ont des noms identiques, ce qui devrait générer des erreurs de nommage.
- **Place 1** : P1 (2 jetons)
- **Place 2** : P1 (5 jetons) (nom dupliqué)
- **Transition 1** : T1 (minTemps = 0, maxTemps = 2)
- **Transition 2** : T1 (minTemps = 0, maxTemps = 3) (nom dupliqué)

Exemple 5 : Temps de Transition négatif (Transition invalide)

Cet exemple contient une transition avec des valeurs de temps négatives, ce qui devrait générer une erreur.
- **Place 1** : P1 (2 jetons)
- **Place 2** : P2 (3 jetons)
- **Transition 1** : T1 (minTemps = -1, maxTemps = 5) (minTemps négatif)
- **Arc 1** : De P1 vers T1
- **Arc 2** : De T1 vers P2

Exemple 6 : Arc avec une source ou une cible nulle (Arc invalide)

Cet exemple contient deux arcs identiques qui relient les memes place et transition associée, ce qui devrait générer une erreur.
- **Place 1** : P1 (2 jetons)
- **Transition 1** : T1 minTemps = 0, maxTemps = 5
- **Arc 1** : De P1 vers T1
- **Arc 2**: De P1 vers T1

3.2.2 Résultats des tests

```
<terminated> ValidatePetriNet [Java Application] /usr/lib/jvm/java-17-openjdk-amd64/bin/java  (26 oct. 2024, 15:52:13 – 15:52:13) [pid: 3167123]
Résultat de validation pour models/Arcspareils.xmi:
- Petrinet: OK
- Place: 1 erreurs trouvées
=> Erreur dans P1 [petrinet.impl.PlaceImpl@5cee5251 (nb_jetons: 1, name: P1)]: Le nom de la place P1 est déjà utilisé par une autre place.
- Transition: 2 erreurs trouvées
=> Erreur dans T1 [petrinet.impl.TransitionImpl@5c909414 (name: T1, minTemps: 0, maxTemps: 5)]: La transition T1 n'a pas de place en entrée.
=> Erreur dans T1 [petrinet.impl.TransitionImpl@5c909414 (name: T1, minTemps: 0, maxTemps: 5)]: Le nom de la transition T1 est déjà utilisé par une autre transition
- Arc: 1 erreurs trouvées
=> Erreur dans T1_P1 [petrinet.impl.ArcImpl@4b14c583 (poids: 0, linkType: standard, direction: TransitiontoPlace, name: T1_P1)]: Il existe déjà un arc reliant la place P1 et la transition T1 dans
Résultat de validation pour models/JetonNegatif.xmi:
- Petrinet: OK
- Place: 2 erreurs trouvées
=> Erreur dans P1 [petrinet.impl.PlaceImpl@65466a6a (nb_jetons: -4, name: P1)]: Le nombre de jetons dans la place P1 ne peut pas être négatif.
=> Erreur dans P2 [petrinet.impl.PlaceImpl@4ddced80 (nb_jetons: 0, name: P2)]: Le nom de la place P2 est déjà utilisé par une autre place.
- Transition: OK
- Arc: 1 erreurs trouvées
=> Erreur dans P2toT1 [petrinet.impl.ArcImpl@1534f01b (poids: 0, linkType: standard, direction: standard, name: P2toT1)]: Il existe déjà un arc reliant la place P2 et la transition T1 dans la même
Résultat de validation pour models/MintempssupMaxtemps.xmi:
- Petrinet: OK
- Place: OK
- Transition: 1 erreurs trouvées
=> Erreur dans T1 [petrinet.impl.TransitionImpl@78e117e3 (name: T1, minTemps: 5, maxTemps: 3)]: Le temps MinTemps dans la transitionT1 ne peut pas être supérieur à MaxTemps.
- Arc: OK
```

Figure 7.1 : Resultats des tests PetriNet

```
Résultat de validation pour models/Noms_identiques.xmi:
- Petrinet: OK
- Place: OK
- Transition: 2 erreurs trouvées
=> Erreur dans T1 [petrinet.impl.TransitionImpl@2ea227af (name: T1, minTemps: 0, maxTemps: 5)]: La transition T1 n'a pas de place en entrée.
=> Erreur dans T1 [petrinet.impl.TransitionImpl@4386f16 (name: T1, minTemps: 0, maxTemps: 0)]: La transition T1 n'a pas de place en sortie.
- Arc: 1 erreurs trouvées
=> Erreur dans A1 [petrinet.impl.ArcImpl@363ee3a2 (poids: 0, linkType: standard, direction: standard, name: A1)]: Il existe déjà un arc reliant la place P1 et la transition T1 dans la même directi
Résultat de validation pour models/PetriNetNormal.xmi:
- Petrinet: OK
- Place: OK
- Transition: OK
- Arc: OK
Résultat de validation pour models/Tempsnegatif.xmi:
- Petrinet: OK
- Place: OK
- Transition: 1 erreurs trouvées
=> Erreur dans T1 [petrinet.impl.TransitionImpl@4690b489 (name: T1, minTemps: -1, maxTemps: 5)]: Le temps dans la transitionT1 ne peut pas être négatif.
- Arc: OK
Résultat de validation pour models/TransitionInvalide.xmi:
- Petrinet: OK
- Place: OK
- Transition: 3 erreurs trouvées
=> Erreur dans T1 [petrinet.impl.TransitionImpl@79b06cab (name: T1, minTemps: 0, maxTemps: 5)]: La transition T1 n'a pas de place en entrée.
=> Erreur dans T2 [petrinet.impl.TransitionImpl@3eb7fc54 (name: T2, minTemps: 0, maxTemps: 0)]: La transition T2 n'a pas de place en sortie.
=> Erreur dans T2 [petrinet.impl.TransitionImpl@3eb7fc54 (name: T2, minTemps: 0, maxTemps: 0)]: Le nom de la transition T2 est déjà utilisé par une autre transition
- Arc: 1 erreurs trouvées
=> Erreur dans P1toT2 [petrinet.impl.ArcImpl@7f552bd3 (poids: 0, linkType: standard, direction: standard, name: P1toT2)]: Il existe déjà un arc reliant la place P1 et la transition T2 dans la même
```

Figure 7.2 : Resultat des tests PetriNet

4 Syntaxe concrète textuelle avec Xtext

L'objectif de cette section est de définir une syntaxe concrète textuelle pour la description d'un processus SimplePDL dans un fichier qu'on a complèté .pdl1, en utilisant Xtext.

Cette approche permet de faciliter la création, l'édition et la validation de modèles SimplePDL sous une forme textuelle, rendant ainsi la modélisation plus facile.

4.1 Présentation de Xtext

Xtext est un framework open-source conçu pour le développement de langages spécifiques de domaine (DSL). Il permet de définir la syntaxe concrète d'un langage ainsi que les outils associés, tels que les éditeurs syntaxiques et les analyseurs.

4.1.1 Éditeurs syntaxiques

Les éditeurs syntaxiques sont des outils intégrés dans les environnements de développement qui aident les utilisateurs à créer et modifier du code en offrant des fonctionnalités telles que la coloration syntaxique, la complétion de code et la validation en temps réel.

4.1.2 Analyseurs

Les analyseurs, ou parseurs, sont des composants qui examinent la syntaxe et la structure du code pour en vérifier la validité. Ils comprennent généralement un analyseur syntaxique, qui construit une représentation de la structure, et un analyseur sémantique, qui s'assure que le code respecte les règles logiques du domaine.

4.2 Exemple d'utilisation

```
process ExempleProcessus {
   //Resources
   ressource r1 have 2
   ressource r2 have 1
    //WorkDefinitions
    wd Conception {
        occurence 1 of r1
    }
    wd RedactionDoc {
        occurence 1 of r1
    }
    wd Programmation {
        occurence 1 of r2
    }
    wd RedactionTests {
        occurence 1 of r2
    }
    //WorkSequences
    ws s2s from Conception to RedactionTests // startToStart
    ws f2s from Conception to Programmation  // finishToStart
    ws s2s from Conception to RedactionDoc // startToStart
    ws f2f from Conception to RedactionDoc // finishToFinish
    ws f2f from Programmation to RedactionTests // finishToFinish
    //Guidances
    guidance "La conception doit être réalisée avant les autres tâches" for Conception
    guidance "Documenter le projet pendant la rédaction de la documentation" for RedactionDoc
    guidance "Programmer les fonctionnalités en respectant les spécifications" for Programmation
    guidance "Les tests doivent être prêts après la programmation" for RedactionTests
}
```

Figure 8 : Exemple de Processus avec code PDL1

5 Syntaxe concrète graphique avec Sirius

L'objectif de cette section est de définir une syntaxe graphique permettant de créer des modèles conformes à SimplePDL. Pour cela, nous avons utilisé l'outil Sirius.

5.1. Présentation de Sirius

Sirius est un framework open-source qui permet de créer des outils de modélisation visuelle pour des langages spécifiques à un domaine (DSL).
Avec Sirius, en développant des éditeurs graphiques, nous pouvons définir des diagrammes et des représentations visuelles des éléments du modèle.

5.2 Présentation de l'éditeur graphique

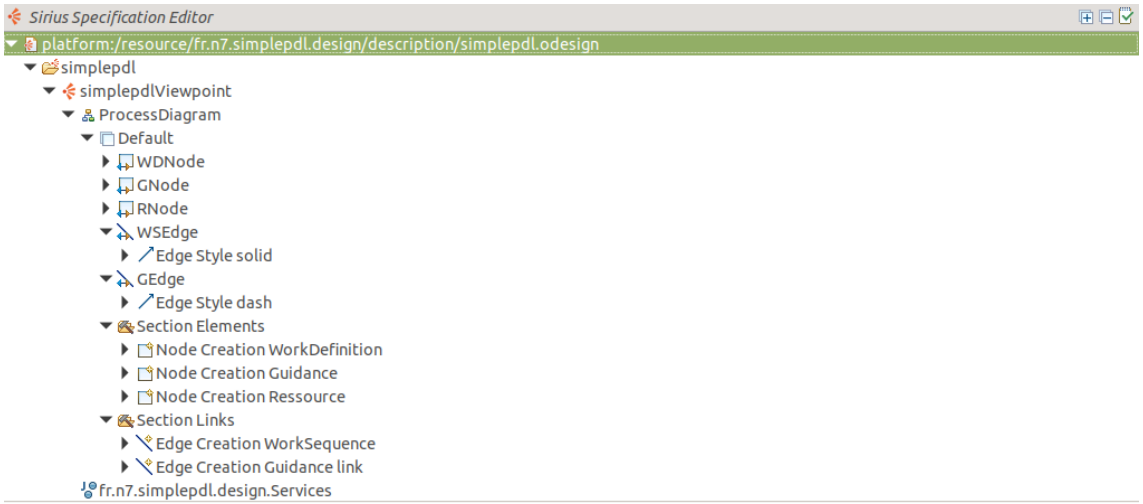


Figure 9 : Vue sur l'editeur graphique

Figure 10.1 : Exemple d'application

Figure 10.2 : Exemple d'application

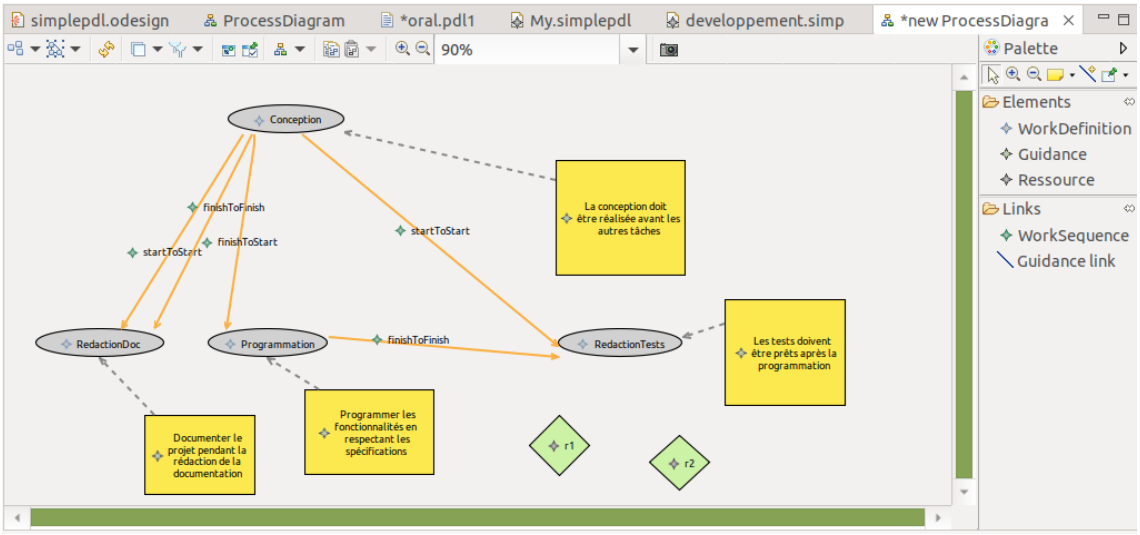


Figure 11 : Le modèle généré par l'éditeur graphique

Remarque :
Il convient de noter que l'outil de création de lien guidé graphiquement semble ne pas fonctionner, même s'il est automatiquement généré pour les exemples. Pour les occurences, elles apparaissent dans les proprietes sans construire l'outil de creation de la classe occurence (oubli).
La creation des noeuds WorkDefinition, Guidance, Ressources et du edge WorkSequence fonctionnent graphiquement sans problème.

6 Transformation Modèle à Modèle

L'objectif de cette section est de réaliser une transformation M2M (Model-to-Model) permettant de convertir un modèle SimplePDL en un modèle PetriNet.

Pour cela, nous utilisons à la fois Java et ATL (Atlas Transformation Language) pour définir et exécuter les règles de transformation

6.1. Principe de la transformation M2M

La transformation M2M consiste à convertir un modèle d'un langage source (SimplePDL) vers un modèle d'un langage cible (PetriNet) tout en préservant la sémantique du modèle original. Cette conversion facilite l'utilisation d'outils d'analyse dédiés aux réseaux de Petri pour vérifier certaines propriétés des processus, comme la possibilité de terminaison.

6.2. Transformation M2M avec le Langage Java

La transformation d'un modèle de processus SimplePDL en un modèle de réseaux de Pétri en Java commence par le chargement des packages correspondants, afin de permettre l'accès aux classes et fonctionnalités nécessaires. Le programme est ensuite configuré pour traiter le modèle SimplePDL en entrée et produire le modèle PetriNet en sortie.

Le processus de conversion s'initie en récupérant l'élément principal du modèle SimplePDL, ce qui permet de préparer la création des éléments équivalents dans le modèle PetriNet. Les éléments de base, tels que les workdefinitions et les ressources, sont alors convertis en places, tandis que les worksequences sont traduites en arcs qui relient ces différentes places.

6.2.1 Mise en Œuvre

Figure 12.1 : Resultat d'une transformation SimplePDL to PetriNet

Figure 12.1 : Resultat d'une transformation SimplePDL to PetriNet

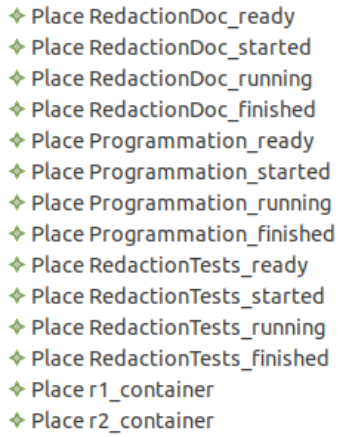


Figure 12.1 : Resultat d'une transformation SimplePDL to PetriNet

6.3. Transformation M2M avec le Langage ATL

Il est aussi possible de faire la transformation SimplePDL vers PetriNet en utilisant ATL, un language de transformation de modèles qui rapporte plus d'efficacité à la transformation .

6.3.1 Présentation du Langage ATL

Le langage ATL permet la traduction de modèle d'un métamodèle vers un modèle d'un autre métamodèle. Le principe d'ATL est de définir un ensemble de règles de transformations permettant de passer d'un élément d'un métamodèle à un élément d'un autre métamodèle.

6.3.2 Principe de la transformation avec ATL

On commence par la création d'un réseau de Pétri qui porte le même nom que le processus d'origine. Les éléments du modèle SimplePDL sont ensuite convertis en composants du réseau de Pétri : les workdefinitions sont transformées en sous-réseaux composés de plusieurs arcs, transitions et places, reliés entre eux. Les worksequences sont traduites en arcs, qui sont associés aux workdefinitions appropriées. Enfin, les ressources sont représentées par des places, intégrées avec les autres éléments du réseau.

7 Transformation Modèle à Texte avec Acceleo

Cette section vise à réaliser une transformation M2T (Model-to-Text) pour convertir un modèle de PetriNet en un format textuel compatible avec la boîte à outils Tina. L'objectif est de générer des fichiers .net qui peuvent être utilisés par Tina pour visualiser et simuler les réseaux de Pétri.

7.1 Présentation d'Acceleo

Acceleo est un outil de génération de code basé sur la technologie MDE (Model-Driven Engineering, ou ingénierie dirigée par les modèles). Il permet de transformer des modèles en code source ou en fichiers texte, dans ce qu'on appelle les transformations M2T (Model-to-Text).
Il est utilisé dans ce mini-projet pour réaliser des transformations M2T, permettant de générer des fichiers textuels à partir de modèles, comme HTML, ou .dot et .net pour Tina.

7.2 Principe de la transformation

La transformation M2T consiste à extraire les éléments du modèle PetriNet et à les convertir en une représentation textuelle suivant la syntaxe exigée par Tina. Le format .net spécifie les composants du réseau, tels que les places, transitions, et arcs, ainsi que leurs relations.

7.3 Mise en Œuvre

```
net developpement
pl Conception_ready (1)
pl Conception_started (0)
pl Conception_running (0)
pl Conception_finished (0)
pl RedactionDoc_ready (1)
pl RedactionDoc_started (0)
pl RedactionDoc_running (0)
pl RedactionDoc_finished (0)
pl Programmation_ready (1)
pl Programmation_started (0)
pl Programmation_running (0)
pl Programmation_finished (0)
pl RedactionTests_ready (1)
pl RedactionTests_started (0)
pl RedactionTests_running (0)
pl RedactionTests_finished (0)
pl r1_container (1)
pl r2_container (0)

tr Conception_start [0,1] Conception_readyr1_container  -> Conception_startedConception_running
tr Conception_finish [2,3] Conception_running -> Conception_finishedr1_container
tr RedactionDoc_start [0,1] RedactionDoc_readyConception_started?1 r1_container  -> RedactionDoc_startedRedactionDoc_running
tr RedactionDoc_finish [2,3] RedactionDoc_runningConception_finished?1  -> RedactionDoc_finishedr1_container
tr Programmation_start [0,1] Programmation_readyConception_finished?1 r2_container  -> Programmation_startedProgrammation_running
tr Programmation_finish [2,3] Programmation_running -> Programmation_finishedr2_container
tr RedactionTests_start [0,1] RedactionTests_readyConception_started?1 r2_container  -> RedactionTests_startedRedactionTests_running
tr RedactionTests_finish [2,3] RedactionTests_runningProgrammation_finished?1  -> RedactionTests_finishedr2_container
```

Figure 13 : Resultat d'une transformation PetriNet to Tino

8 Propriétés LTL et terminaison d'un processus

Il est important de valider la transformation de modèle ce qui est réalisé en créant un fichier LTL (Linear Temporal Logic) que nous utilisons pour tester notre modèle dans Tina. .
Afin de valider la transformation SimplePDL vers PetriNet, une possibilité est de vérifier que les invariants sur le modèle de processus sont préservés sur le modèle de réseau de Petri correspondant. Ces invariants sont appelés propriétés de sûreté.
On peut alors écrire une transformation modèle à texte qui traduit ces propriétés de sûreté sur le modèle de Petri.
L'outil selt permettra alors de vérifier si elles sont effectivement satisfaites sur le modèle de réseau de Petri.

8.1 Présentation de LTL

La logique temporelle linéaire (LTL, Linear Temporal Logic) est un formalisme utilisé pour exprimer des propriétés des systèmes dynamiques, en particulier dans le domaine de la vérification des modèles et de la logique des systèmes.
Elle permet de formuler des assertions sur les comportements futurs d'un système en s'appuyant sur des propositions logiques qui peuvent varier au fil du temps.

8.2 Mise en Œuvre

```
nmi9773@murdock:~/SN2A/IDMFINAL/fr.n7.ORAL$ tina developpement.net developpement.ktz
# net developpement, 32 places, 8 transitions                          #
# bounded, not live, possibly reversible                               #
# abstraction        count      props      psets       dead       live #
#      states            1         32          1          1          1 #
# transitions            0          8          8          8          0 #
nmi9773@murdock:~/SN2A/IDMFINAL/fr.n7.ORAL$ selt -p -S developpement.scn developpement.ktz -prelude developpement_finished.ltl
Selt version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 1 states, 0 transitions
0.002s

- source developpement_finished.ltl;
operator finished : prop
TRUE
TRUE
FALSE
  state 0: L.dead Conception_ready Programmation_ready RedactionDoc_ready RedactionTests_ready r1_container
  -L.deadlock->
  state 1: L.dead Conception_ready Programmation_ready RedactionDoc_ready RedactionTests_ready r1_container
  [accepting all]
TRUE
0.003s
```

Figure 14 : Resultat de la verification de la terminaison du processus developpement

9 Conclusion

Ce mini-projet qui à la fois intéressant et instructif, a démontré l'efficacité de la vérification formelle dans la modélisation des processus via le développement d'une chaîne de vérification pour les modèles SimplePDL. L'intégration des réseaux de Petri a permis d'analyser et de garantir la cohérence des modèles. La spécification de sémantiques en Java a offert un cadre solide pour la validation des contraintes.

Les tests réalisés avec les frameworks Xtext et Sirius ont confirmé l'applicabilité du projet dans divers contextes.

Ce travail ouvre également des perspectives visant à optimiser davantage les processus de modélisation d'une part et les processus de vérification d'autre part.