Une chaîne de vérification pour modèles de procédés

Bekkare Hanna, Ferreira Mathilde

Deuxième année FISA - MODIA
2024-2025

Table des matières

1 Introduction **3**

2 Spécification des modèles de procédés **3**
2.1 Modèle de procédé : `SimplePDL` . . . 3
2.2 Modèle de réseau de Pétri : `Petrinet` . . . 4

3 Validation **5**
3.1 Validation de SimplePDL . . . 5
3.2 Validation de PetriNet . . . 6
3.3 Validation sur des exemples . . . 6
3.3.1 Modèles SimplePDL . . . 6
3.3.2 Modèles PetriNet . . . 7

4 Syntaxe concrète graphique **7**

5 Syntaxe concrète textuelle **9**
5.1 Syntaxe PDL1 . . . 9
5.2 Transformation PDL1 vers SimplePDL . . . 10
5.3 Limitation et amélioration possible . . . 10

6 Transformation modèle à modèle **10**

7 Transformation vers Dot **11**
7.1 SimplePDL . . . 11
7.2 Petrinet . . . 13

8 Liste des fichiers **14**

9 Conclusion **15**

Table des figures

1 Diagramme Ecore du méta-modèle `SimplePDL` . . . 3
2 Diagramme Ecore du méta-modèle `Petrinet` . . . 4
3 Interface de création de l'éditeur Sirius . . . 8
4 Palette de création des outils dans l'éditeur Sirius . . . 8
5 Exemple de ProcessDiagram pour OfficeSimplePDL . . . 9
6 Exemple de syntaxe textuelle PDL1 pour le modèle *FaireGateau* . . . 9
7 Transformation du modèle FaireGateau simplePDL vers Petrinet . . . 11
8 Code permetant de passer de simplepdl en .dot . . . 11
9 Graphe généré pour le modèle `FaireGateau` . . . 12
10 Code permetant de passer de petrinet en .dot . . . 13
11 Graphe généré pour le modèle `FaireGateau` . . . 14

1 Introduction

L'objectif de ce projet est de concevoir un écosystème complet permettant à un utilisateur de définir des modèles de procédés. Ces modèles incluent des tâches, des ressources, ainsi que les relations de dépendance qui peuvent exister entre elles.

Cet écosystème est destiné à des utilisateurs ne disposant pas nécessairement de compétences avancées en informatique. Cela implique de concevoir des outils à la fois ergonomiques et intuitifs, capables de répondre efficacement aux interrogations que l'utilisateur pourrait se poser à propos d'un modèle de procédé.

Afin de faciliter le développement tout en garantissant la pertinence et la compréhension du modèle pour l'utilisateur final, nous avons opté pour une approche fondée sur l'ingénierie dirigée par les modèles.

Le projet mobilise donc un ensemble de technologies et de méthodes dans le but de proposer une chaîne d'outils cohérente pour la définition, la gestion et la vérification des modèles de procédés.

2 Spécification des modèles de procédés

2.1 Modèle de procédé : `SimplePDL`

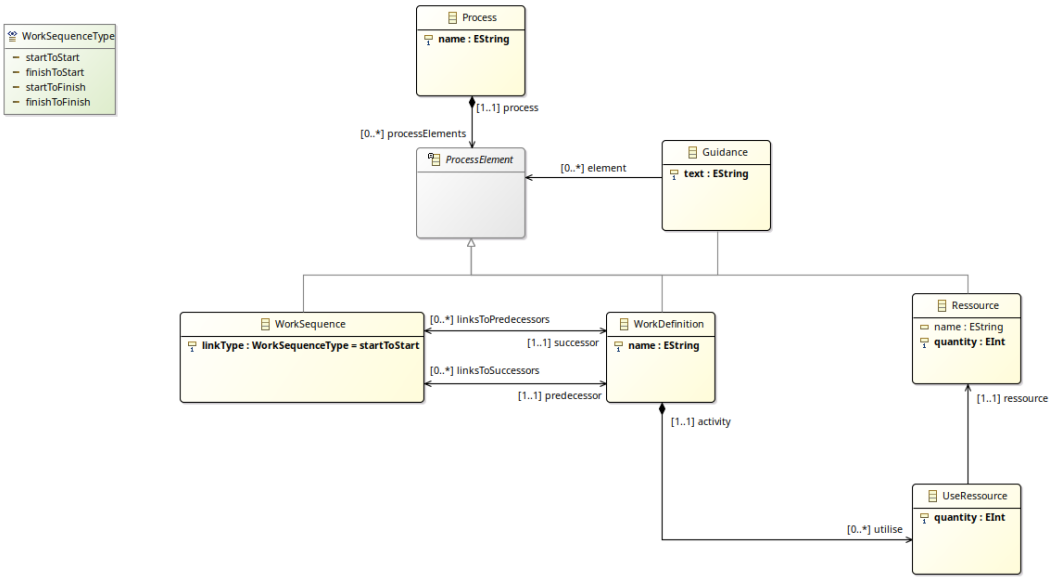

FIGURE 1 – Diagramme Ecore du méta-modèle `SimplePDL`

R1.1 Un modèle de procédé se compose de tâches et de ressources.
- Une tâche est représentée par la classe `WorkDefinition`, caractérisée par un attribut `name`.
- Une ressource est modélisée par la classe `Ressource`, contenant les attributs `name` et `quantity`.

R1.2 Les tâches peuvent être liées par des dépendances.
- Ces dépendances sont représentées par la classe `WorkSequence`, qui possède un attribut `linkType` de type `WorkSequenceType`.
- Les types de dépendances modélisés sont : `start-to-start`, `finish-to-start`, `start-to-finish`, et `finish-to-finish`.

R1.3 Les tâches peuvent utiliser des ressources via des associations spécifiques.
- `UseResource` est une classe d'association entre `WorkDefinition` et `Ressource`, permettant de spécifier la quantité utilisée dans un contexte donné. Cette association modélise une relation plusieurs-à-plusieurs (N :N) tout en maintenant l'information sur la quantité requise.
- La relation entre `WorkDefinition` et `UseResource` est une composition, car la durée de vie de l'objet `UseResource` dépend de celle de la tâche.

R1.4 Le modèle peut contenir des commentaires explicatifs.
- Les commentaires sont modélisés par la classe `Guidance`, qui possède une référence sortante vers un ou plusieurs éléments du procédé (`ProcessElement`). Cela permet d'associer des commentaires à des éléments spécifiques ou de les laisser non attachés.

R1.5 Le modèle doit respecter des contraintes structurelles et sémantiques rigoureuses.
- Le méta-modèle est implémenté avec `Ecore`, en respectant les bonnes pratiques de modélisation : pertinence des attributs et relations, architecture extensible, noms conformes, etc.
- En raison d'un bug critique dans l'OCL, les règles de validation sont implémentées en Java via la classe `SimplePDLValidator`. Cette classe utilise un `SimplepdlSwitch` pour parcourir les éléments et appliquer des règles telles que : unicité des identifiants, cohérence des dépendances, validité des ressources, non-récursivité, etc.
- La description des contraintes se trouve dans la partie **Validation**.

2.2 Modèle de réseau de Pétri : `Petrinet`

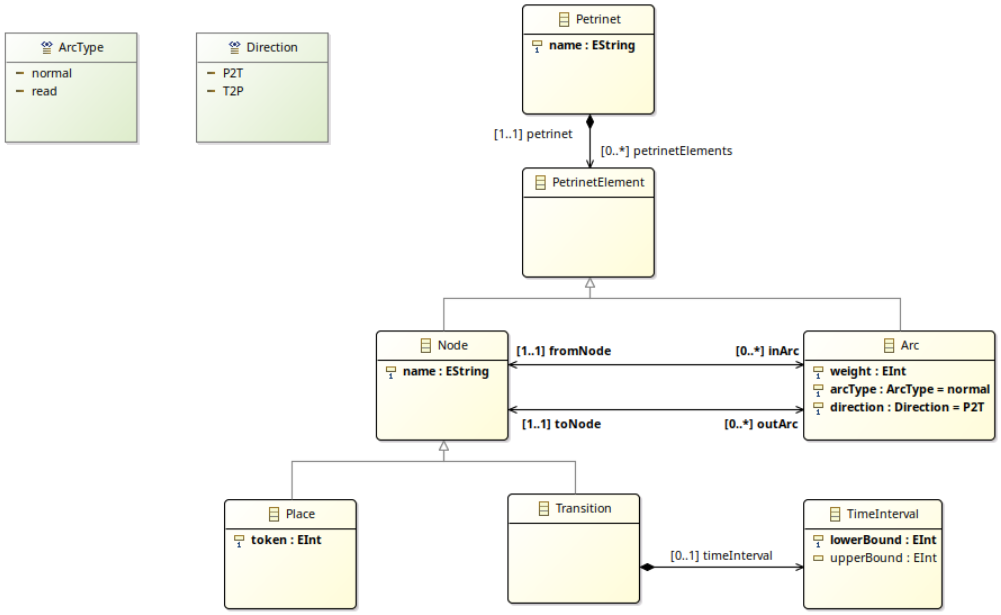

FIGURE 2 – Diagramme Ecore du méta-modèle `Petrinet`

R2.1 Un réseau de Pétri est constitué de places et de transitions.
- La classe abstraite `PetrinetElement` est utilisée pour modéliser les éléments de base.
- `Node` hérite de `PetrinetElement`, et les classes `Place` et `Transition` héritent de `Node`.

R2.2 Une place est nommée et peut contenir un nombre de jetons.
- L'attribut `name` est défini dans `Node`.
- `Place` possède un attribut entier `token` représentant le marquage.

R2.3 Une transition est nommée et peut inclure un intervalle de temps.
- L'attribut `name` est défini dans `Node`.
- Une relation optionnelle (multiplicité [0..1]) de composition est définie vers la classe `TimeInterval`, qui possède :
  - Une borne inférieure obligatoire (`lowerBound`).
  - Une borne supérieure optionnelle (`upperBound`), pouvant être non définie.

R2.4 Les places et transitions sont reliées par des arcs pondérés.
- Les arcs sont représentés par la classe `Arc`, héritant de `PetrinetElement`, et possédant un attribut entier obligatoire `weight`.
- Chaque arc relie soit une place à une transition, soit une transition à une place, grâce à l'enum `Direction` (`P2T`, `T2P`).
- Les arcs peuvent être de type `normal` ou `read`, définis dans l'enum `ArcType`.

R2.5 Le réseau possède un nom.
- L'attribut obligatoire `name` est défini dans la classe `Petrinet`, située à la racine du modèle.

R2.6 Le modèle doit respecter des contraintes structurelles et sémantiques.
- Celles-ci sont implémentées dans le package `n7.petrinet.validation` via une classe Java dédiée (`ValidatePetrinet`).
- La description des contraintes se trouve dans la partie **Validation**.

3 Validation

La phase de validation a pour objectif de s'assurer que les modèles créés respectent bien les contraintes définies par les règles métiers de chaque langage (SimplePDL et PetriNet). Pour cela, une classe de validation a été implémentée afin de parcourir les éléments du modèle et d'y appliquer les différentes vérifications nécessaires.

3.1 Validation de SimplePDL

La validation pour le langage SimplePDL couvre plusieurs types d'éléments : `Process`, `WorkDefinition`, `WorkSequence`, `Ressource`, `UseRessource` et `Guidance`. Voici les règles principales mises en œuvre :
- **Process** : Le nom du processus doit être non nul et respecter les conventions de nommage Java.
- **WorkDefinition** : Chaque activité doit avoir un nom valide et unique au sein d'un même processus.
- **WorkSequence** :
  - Une activité ne peut pas dépendre d'elle-même.
  - Il ne peut y avoir deux séquences de dépendance identiques (même prédécesseur, successeur et type).
- **Ressource** :
  - Le nom doit être valide et unique.
  - La quantité disponible doit être strictement positive.
- **UseRessource** :
  - La quantité utilisée doit être strictement positive.
  - Elle ne doit pas excéder la quantité disponible.
- **Guidance** : Le texte doit être non vide.

3.2 Validation de PetriNet

La validation des modèles de réseaux de Pétri s'applique aux éléments suivants : `Petrinet`, `Place`, `Transition`, `Arc` et `TimeInterval`. Les contraintes sont les suivantes :

- **Petrinet** : Le nom du réseau doit être non nul et conforme aux conventions Java.
- **Place** :
  - Le nom doit être non vide.
  - Le marquage initial doit être supérieur ou égal à zéro.
- **Transition** :
  - Le nom doit être non vide.
  - Si un intervalle de temps est défini :
    - La borne inférieure doit être ≥ 0.
    - La borne supérieure, si elle est définie, doit être supérieur à la borne inférieure.
- **Arc** :
  - Le poids de l'arc doit être strictement positif.
  - L'arc doit relier une place à une transition ou inversement.
  - La direction déclarée doit correspondre à la structure réelle de l'arc.

3.3 Validation sur des exemples

3.3.1 Modèles SimplePDL

FaireGateauSimplePDL Ce modèle est valide et respecte toutes les contraintes :

```
Résultat de validation pour OutFaireGateau.xmi:
- Process: OK
- WorkDefinition: OK
- WorkSequence: OK
- Guidance: OK
- Ressource: OK
- UseRessource: OK
Fini.
```

SimplePDL-ko-1 Ce modèle contient une erreur volontaire : une ressource avec une quantité nulle.

Sortie de validation de SimplePDL-ko-1.xmi

```
Résultat de validation pour SimplePDL-ko-1.xmi:
- Process: OK
- WorkDefinition: OK
- WorkSequence: OK
- Guidance: OK
- Ressource: 1 erreurs trouvées
=> Erreur dans r (name: r, quantity: 0)]:
La quantité de la ressource doit être strictement positive
- UseRessource: OK
Fini.
```

SimplePDL-ko-2 Ce modèle devrait être invalide mais passe la validation. Il présente un cas de "deadlock" entre deux activités et une insuffisance de ressources. Cela met en évidence une limite de la validation statique, qui ne détecte pas certains problèmes dynamiques.

Sortie de validation de SimplePDL-ko-2.xmi

```
Résultat de validation pour SimplePDL-ko-2.xmi:
- Process: OK
- WorkDefinition: OK
- WorkSequence: OK
- Guidance: OK
- Ressource: OK
- UseRessource: OK
Fini.
```

3.3.2 Modèles PetriNet

OfficePetrinet Modèle valide, toutes les entités respectent les règles :

```
Résultat de validation pour OfficePetrinet.xmi:
- Petrinet: OK
- Place: OK
- Transition: OK
- Arc: OK
- TimeInterval: OK
Fini.
```

Petrinet-ko-2 Ce modèle présente plusieurs erreurs :

- Un arc a un poids nul.
- Un arc relie deux places (connexion invalide).
- La direction déclarée d'un arc est incohérente.

Sortie de validation de Petrinet-ko-2.xmi

```
Arc from: PlaceImpl to: TransitionImpl direction: P2T
Arc from: TransitionImpl to: PlaceImpl direction: T2P
Arc from: PlaceImpl to: PlaceImpl direction: P2T
Résultat de validation pour Petrinet-ko-2.xmi:
- petrinet: OK
- Place: OK
- Transition: OK
- Node: OK
- Arc: 3 erreurs trouvées
=> Erreur dans n7.petrinet.impl.ArcImpl (weight: 0, arcType: normal, direction: P2T):
Le poids d'un arc doit être strictement positif
=> Erreur dans n7.petrinet.impl.ArcImpl (weight: 0, arcType: normal, direction: P2T):
Un arc doit relier une place à une transition (ou l'inverse)
=> Erreur dans n7.petrinet.impl.ArcImpl (weight: 0, arcType: normal, direction: P2T):
La direction déclarée de l'arc ne correspond pas à sa structure réelle
- TimeInterval: OK
Fini.
```

4 Syntaxe concrète graphique

Nous avons développé un éditeur graphique basé sur Sirius, permettant une représentation visuelle des modèles de procédés, ainsi que la création et la modification de ces derniers.

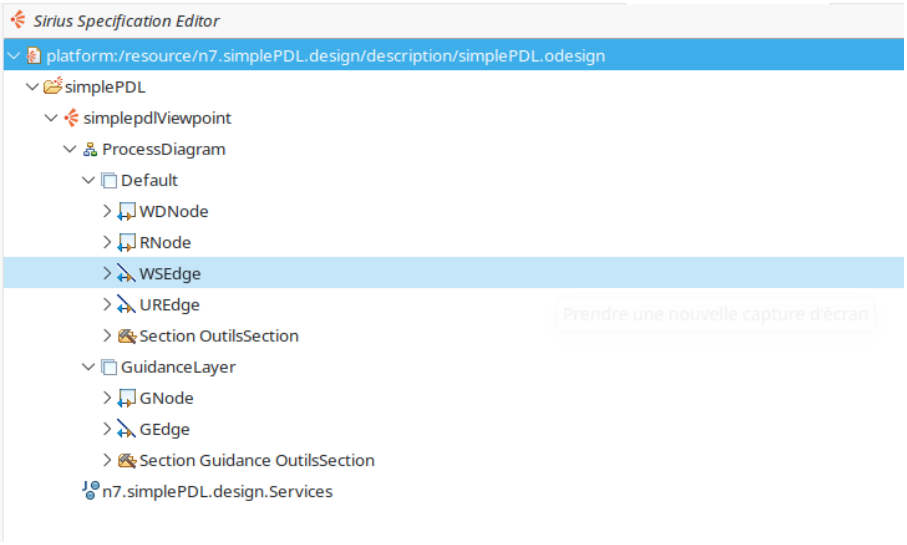

FIGURE 3 – Interface de création de l'éditeur Sirius

R4.1 L'éditeur doit être capable d'afficher tous les éléments d'un modèle de procédé.

— L'éditeur permet d'afficher tous les éléments du modèle SimplePDL.

R4.2 L'éditeur doit offrir, via sa palette, la possibilité de créer n'importe quel élément d'un modèle de procédé.

— Nous avons intégré des outils permettant de créer différents objets du modèle de procédé dans l'éditeur graphique.

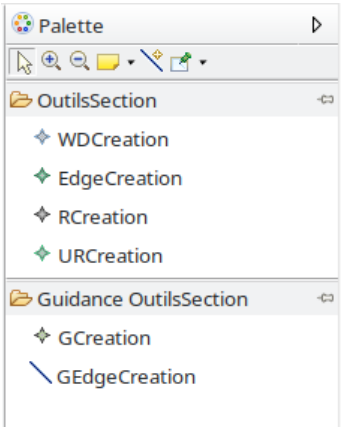

FIGURE 4 – Palette de création des outils dans l'éditeur Sirius

R4.3 Les représentations (nœuds, arcs) des éléments d'un modèle de procédé doivent être pertinentes et distinctes.

— Les WorkDefinitions et Ressources sont représentées par des nœuds dans Sirius.

— Les WorkSequences et UseResources sont des arêtes basées sur des éléments, car elles relient les WorkDefinitions et Ressources.

— Les Guidances sont représentées par des nœuds, et les GEdges sont des arêtes relationnelles, pouvant lier les Guidances aux WorkDefinitions ou non.

R4.4 L'outil doit proposer au moins deux calques, afin de séparer les commentaires du reste du modèle.

— Pour mieux organiser nos éléments dans l'éditeur Sirius, nous avons créé deux calques : un calque par défaut contenant tous les éléments à l'exception des Guidances et GEdges, qui sont placés dans un calque séparé nommé "GuidanceLayer".

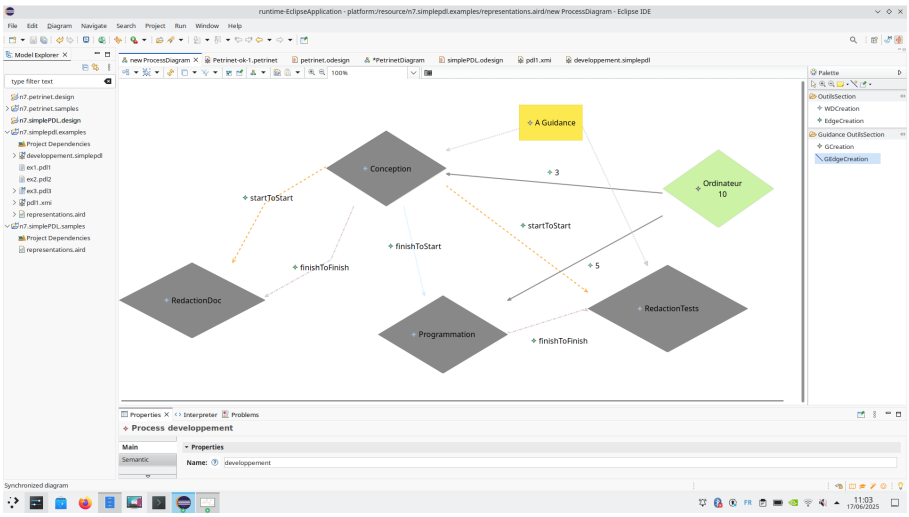

FIGURE 5 – Exemple de ProcessDiagram pour OfficeSimplePDL

5 Syntaxe concrète textuelle

5.1 Syntaxe PDL1

La syntaxe concrète textuelle du langage PDL1 a été définie à l'aide de Xtext. Le fichier `pdl1.xtest` correspond à la grammaire mise en place pour permettre la génération automatique d'un éditeur et la transformation vers un modèle EMF.

Cette grammaire permet de décrire un processus composé de plusieurs éléments : des définitions de tâches (`WorkDefinition`), des séquences de dépendance (`WorkSequence`), des ressources (`Ressource`), des consommations de ressources (`UseRessource`) et des annotations textuelles (`Guidance`).

```
1  process FaireGateau {
2      r cuisinier q 1
3      r gourmand q 1
4      wd MelangerIngredient {
5          useRessources : r cuisinier q 1
6      }
7      wd CuireGateau {
8          useRessources : r cuisinier q 1
9      }
10     wd MangerGateau {
11         useRessources : r gourmand q 1
12     }
13     ws finishToStart from MelangerIngredient to CuireGateau
14     ws finishToStart from CuireGateau to MangerGateau
15     note "Ne mangez pas le gâteau avant qu'il soit cuit !"
16 }
```

FIGURE 6 – Exemple de syntaxe textuelle PDL1 pour le modèle *FaireGateau*

5.2 Transformation PDL1 vers SimplePDL

On crée un fichier pdl1.atl qui nous permet de créer un modèle SimplePDL à partir d'un modèle PDL1. La transformation d'un modèle à un autre avec `atl` sera décrite dans la section **Transformation modèle à modèle**. Une fois le modèle simplepdl obtenue on vérifie sa validité en utilisant `ValidateSimplePDL.java`.

5.3 Limitation et amélioration possible

Actuellement, la syntaxe ne permet pas de relier un élément de type `Guidance` à un élément spécifique du processus (comme une `WorkDefinition`). En effet, le métamodèle de `SimplePDL` autorise cette association, mais la grammaire Xtext ne reflète pas cette relation, ce qui fait que les `Guidance` sont isolées dans le modèle.

Une amélioration future consisterait à modifier la grammaire pour permettre de rattacher directement une note à un élément du processus, ce qui améliorerait la cohérence entre la syntaxe textuelle et le métamodèle.

6 Transformation modèle à modèle

Afin de pouvoir répondre à des questions complexes sur « l'exécution » des modèles de procédés, il est nécessaire de les transformer en réseaux de Pétri. Ces derniers ayant été méta-modélisés, il s'agit de concevoir une transformation de modèle à modèle.

La transformation prend en entrée un modèle de procédé conforme à SimplePDL et crée en sortie un modèle de réseau de Pétri conforme au méta-modèle développé.

Nous avons créer un script `simplePDLToPetrinet.atl`, qui contient les règles de transformation de simplepdl à petrinet. Les règles sont décrite ci-dessous :

Règle `Process2Petrinet` Cette règle transforme un objet `Process` de `SimplePDL` en un objet `Petrinet`. Elle copie le nom du processus et récupère tous les éléments générés dans la sortie (`OUT`) qui sont des `PetrinetElement`, incluant ainsi les places, transitions et arcs associés.

Règle `WD` Cette règle transforme une `WorkDefinition` en cinq éléments du réseau de Petri :
- Une `Place` initiale (avec un token) représentant le début de l'activité.
- Une `Transition` de début (`starting_transition`).
- Une `Place` active (`active_place`) sans jeton.
- Une `Transition` de fin (`finishing_transition`).
- Une `Place` finale (`finished_place`).

Quatre arcs relient ces éléments dans l'ordre logique du déroulement de l'activité. Chaque arc est typé et orienté selon le sens de l'exécution.

Règle `WS` Cette règle transforme une `WorkSequence` en un `Arc` de type `read`. Le nœud source et le nœud cible sont choisis dynamiquement à l'aide des helpers `sourceNode()` et `targetNode()` selon le type de lien (`linkType`).

Règle `R` Cette règle transforme une `Ressource` en une `Place` dans le réseau de Petri. Le nombre de jetons correspond à la quantité disponible de cette ressource.

Règle `UR` Cette règle transforme une `UseRessource` (utilisation d'une ressource par une activité) en deux arcs :
- Un arc normal depuis la place représentant la ressource vers la transition de début de l'activité, avec un poids correspondant à la quantité requise.
- Un arc normal depuis la transition de fin vers la place de la ressource, représentant la restitution.

Les arcs utilisent `refImmediateComposite()` pour retrouver l'activité parente (WorkDEfinition) de l'utilisation.

Helper `sourceNode` et `targetNode` Ces helpers permettent de déterminer dynamiquement les nœuds source et cible d'une `WorkSequence`, en fonction de son `linkType` (start-to-start, finish-to-start, etc.).

Une fois ce script appliqué sur un Simplepdl à travers l'outil ATL on obtient un Petrinet. On valide par la suite le modèle obtenue avec `ValidatePetrinet.atl`.

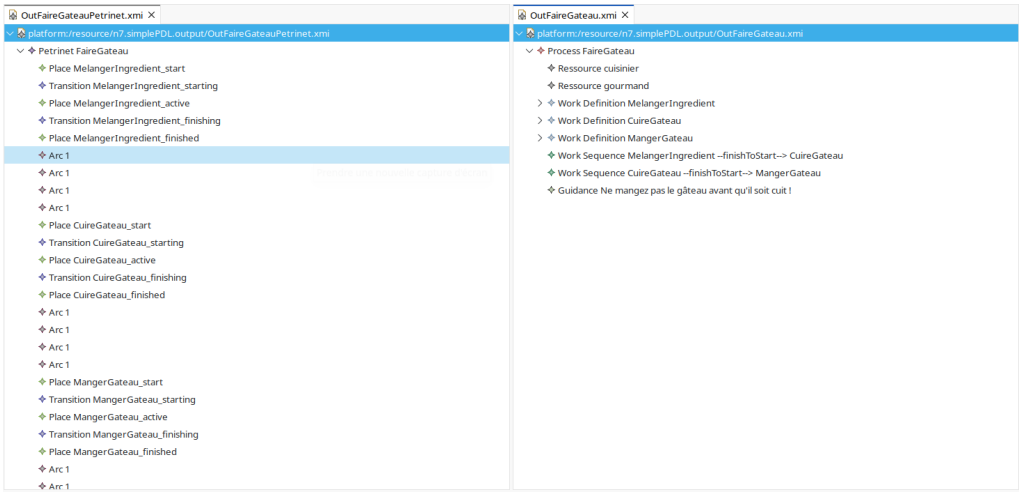

FIGURE 7 – Transformation du modèle FaireGateau simplePDL vers Petrinet

7 Transformation vers Dot

Pour avoir une représentation sous forme d'image d'un modèle de procédé, on définie une transformation modèle à texte vers le format Dot de Graphviz.

7.1 SimplePDL

```
digraph [aProcess.name/] {

[for (ws : WorkSequence | aProcess.processElements
    ->select(e | e.oclIsTypeOf(WorkSequence))
    ->collect(e | e.oclAsType(WorkSequence)))]
  [ws.predecessor.name/] -> [ws.successor.name/] [ '[' /]arrowhead=vee label="[ws.linkType.toLabel()/]"[ ']' /];
[/for]
[for (ur : UseRessource | aProcess.processElements ->select(e | e.oclIsTypeOf(WorkDefinition))
    ->collect(wd | wd.oclAsType(WorkDefinition).utilise)
  ->flatten()
)]
  [ur.ressource.name/] -> [ur.activity.name/]  [ '[' /]arrowhead=vee label="[ur.quantity/]"[ ']' /];
 [/for]
}
```

FIGURE 8 – Code permetant de passer de simplepdl en .dot

Ce code génére étape par étape :

— On parcourt toutes `WorkSequence`, les **nœuds** correspondent aux `WorkDefinition` (predessesor et successor de la `WorkSequence`) et les **flèches** représentent les dépendances

entre activités, avec un label indiquant le type de lien (`startToStart`, `startToFinish`, `finishToStart`, `finishToFinish`).

— On parcourt ensuite toutes les `WorkDefinition` et on recupère le `UseRessource` dans l'argument `utilise` des `WorkDefinition`. Les **nœuds** correspondent aux `Ressource` (`ressource` du `UseRessource`) et les **flèches** représentent les dépendances entre la `Ressource` et la `WorkDefinition` (`activity` du `UseRessource`), avec un label indiquant la quantité.

Chaque arête est traduite dans le langage DOT avec un attribut `arrowhead=vee` pour indiquer la direction.

Un exemple de sortie produite est donné à la figure suivante :

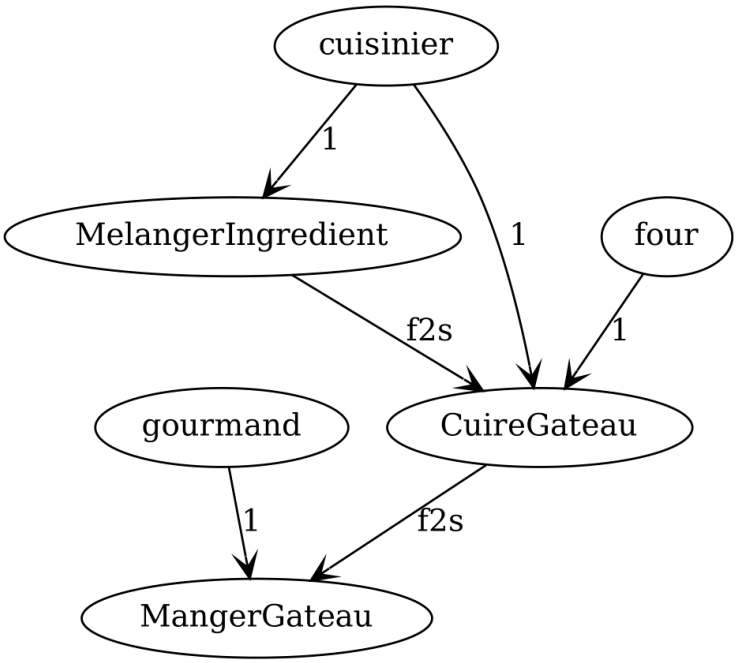

FIGURE 9 – Graphe généré pour le modèle `FaireGateau`

7.2 Petrinet

```
digraph [aPetrinet.name/] {
    // Transitions en rectangle
    [for (t : Transition | aPetrinet.petrinetElements->select(e | e.oclIsTypeOf(Transition)))]
    [t.name/] [shape=rectangle];
    [/for]
    // Places en cercle
    [for (p : Place | aPetrinet.petrinetElements->select(e | e.oclIsTypeOf(Place)))]
    [p.name/];
    [/for]
    // Arcs
    [for (arc : Arc | aPetrinet.petrinetElements->select(e | e.oclIsTypeOf(Arc)))]
    [arc.fromNode.name/] -> [arc.toNode.name/] [label="[arc.arcType.toLabel()/]"];
    [/for]
}
```

FIGURE 10 – Code permetant de passer de petrinet en .dot

Ce code génére étape par étape :
— On parcourt toutes `Transition` pour former des **nœuds** en forme de rectangle.
— On parcourt toutes les Places pour créer des `noeuds` en forme de cercle.
— On parcourt tous les `Arcs`, et on créer des flèches entre les atributs `fromNodes` et `toNode`. On ajoute en label le `ArcType` de l'`Arc`.
Un exemple de sortie produite est donné à la figure suivante :

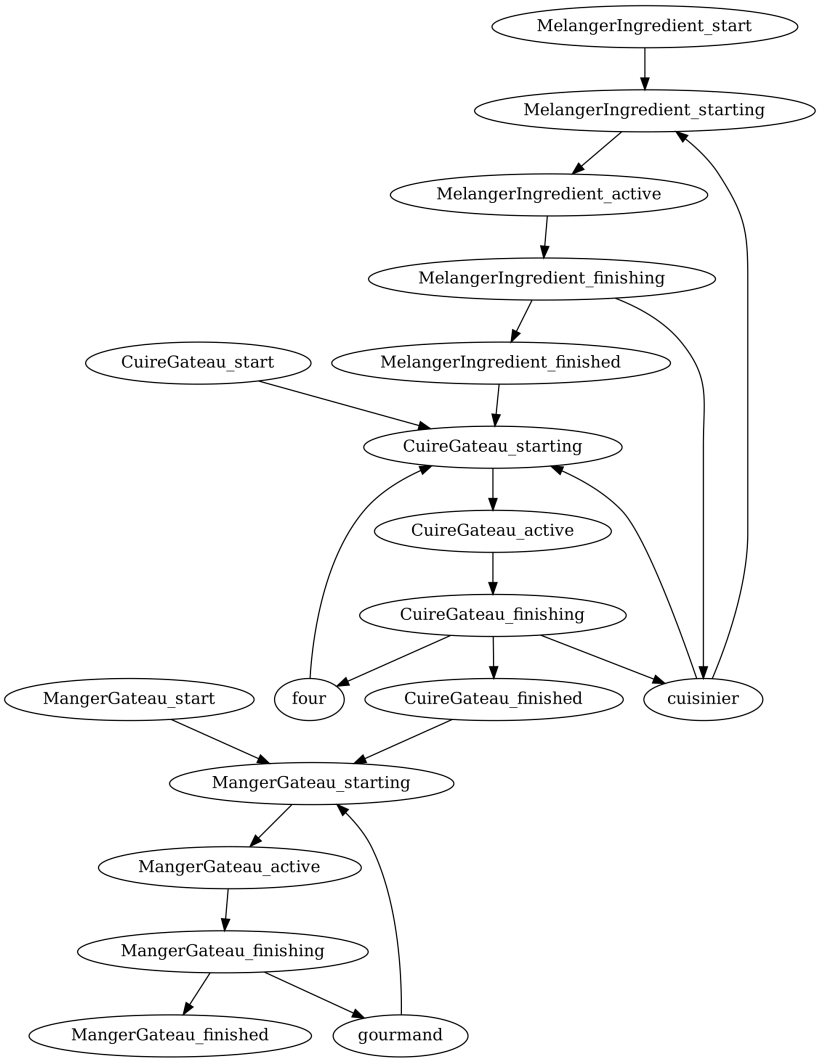

FIGURE 11 – Graphe généré pour le modèle `FaireGateau`

8 Liste des fichiers

Dans l'éditeur de développement

- **n7.Petrinet** : contient le méta-modèle PetriNet (Petrinet.ecore), le genmodel associé, ainsi que le code généré par EMF (interfaces, classes d'implémentation et utilitaires). Inclut également la validation spécifique (ValidatePetrinet, PetrinetValidator, etc.).
- **n7.simplePDL** : contient le méta-modèle SimplePDL (SimplePDL.ecore), le genmodel, les classes générées (WorkDefinition, WorkSequence, etc.), ainsi que le code de validation (SimplePDLValidator, Utils) et des classes de manipulation.
- **n7.simplePDL.editor** : éditeur EMF généré pour SimplePDL, incluant les classes de présentation (SimplepdlEditor, SimplepdlModelWizard) et les icônes.
- **n7.Petrinet.editor** : éditeur EMF généré pour PetriNet, contenant les classes de présentation (PetrinetEditor, PetrinetModelWizard) et les icônes.
- **n7.simplePDL.ATL** : règles de transformation ATL entre le modèle SimplePDL et d'autres

méta-modèles (par exemple pdl1.atl, taskMaster.atl).
— **n7.simplePDL.toDOT** : transformation Acceleo/M2T pour générer un fichier DOT à partir d'un modèle SimplePDL (Todot.java, toDOT.mtl, tâches Ant dans toDOT.xml).
— **n7.PDL1** : langage textuel Xtext généré (grammaire PDL1), incluant les classes générées (parsers, scoping, validation) et les tests.

Dans l'Eclipse de déploiement
— **n7.simplePDL.design** : Projet de description Sirius pour SimplePDL.
  — `description/simplePDL.odesign` — Fichier de configuration du diagramme.
  — `src/n7/simplePDL/design/Services.java` — Services Java pour le diagramme.
— **n7.pdl1.ATL** : Règles de transformation ATL entre SimplePDL et PetriNet.
  — `pdl1.atl`, `simplePDLToPetrinet.atl` — Règles de transformation.
— **n7.simplePDL.samples** : Exemples de modèles SimplePDL.
  — `representations.aird` — Fichier de représentation graphique.
— **n7.petrinet.samples** : Exemples de modèles PetriNet générés.
  — `representations.aird`, `OfficePetrinet.xmi` — Modèles instanciés.
— **n7.simplePDL.output** : Modèles de sortie.
  — `OutFaireGateau.xmi`, `OutFaireGateauPetrinet.xmi` — Modèles générés.

9 Conclusion

Au terme de ce projet, nous avons mis en place une chaîne d'outils complète permettant de modéliser, valider et visualiser des procédés à travers deux formalismes distincts : SimplePDL pour les modèles de procédés, et PetriNet pour les réseaux de Pétri. Nous avons défini et spécifié les méta-modèles de ces deux formalismes à l'aide d'Ecore, en veillant à ce qu'ils respectent les bonnes pratiques de modélisation. Nous avons également conçu un mécanisme de validation rigoureux, garantissant que les modèles respectent les contraintes structurelles et sémantiques imposées. Pour permettre une exploitation dynamique des modèles, nous avons implémenté une transformation modèle à modèle en ATL, qui traduit automatiquement un modèle SimplePDL en un réseau de Petri conforme à notre méta-modèle. Enfin, nous avons développé un éditeur graphique sous Sirius ainsi qu'une syntaxe textuelle (PDL1) facilitant la création, la modification et la visualisation des modèles par des utilisateurs.

Difficultés rencontrées Au cours de la réalisation de ce projet, nous avons été confrontés à plusieurs difficultés. Dans un premier temps, l'installation d'Eclipse et des différents modules nécessaires (Xtext, Sirius, Acceleo, ATL, etc.) a été compliquée par une mauvaise configuration initiale de Java. Cette erreur a entraîné des incompatibilités entre les versions et une instabilité générale de l'environnement de développement, ce qui nous a contraints à recommencer la procédure d'installation.

Lors de la création de la transformation ATL entre SimplePDL et PetriNet, nous avons rencontré un problème de compatibilité lié à un attribut `use` que nous avions défini dans le métamodèle SimplePDL. Ce mot étant réservé par la syntaxe ATL, il a fallu renommer l'attribut et modifier le métamodèle Ecore. Cette modification a nécessité une adaptation manuelle des fichiers .simplepdl existants ainsi que la regénération des artefacts associés, ce qui a engendré des conflits et une phase de débogage fastidieuse.

Par ailleurs, le `genmodel` de PetriNet est devenu inutilisable à la suite de ces changements. Nous avons donc dû le reconstruire, ajuster le namespace (passé à `http://petrinet`) et revoir l'organisation des dossiers générés.

Nous avons également investi beaucoup de temps sur la transformation vers le format `Dot`, notamment pour personnaliser le style graphique des éléments générés. Malgré de nombreuses recherches sur les forums et dans la documentation, ces ajustements visuels n'ont jamais donné le résultat escompté.

Enfin, la veille de la présentation, une partie de l'éditeur PDL1 a disparu de notre installation d'Eclipse. Cette disparition est survenue alors que nous essayions d'alléger le workspace en fermant les projets non utilisés, ce qui a malheureusement désactivé le projet `n7.PDL1.edit`. Dans le même temps, la suppression des fichiers de la syntaxe PDL3, que nous pensions obsolètes, a provoqué des références invalides dans le fichier `representation.aird`. Cette succession d'événements a nécessité une dernière séance de correction d'urgence avant la soutenance.