Rapport Mini-projet : Chaîne de Vérification de Modèles de Processus

ELLAIK Fadwa , OUNCHARI Imane , Groupe L4

Année Académique 2025/2026

Résumé

Ce rapport présente la mise en œuvre d'une chaîne complète d'Ingénierie Dirigée par les Modèles. L'objectif était de créer un outillage permettant la vérification de la cohérence et de la terminaison de modèles de processus **SimplePDL**. La chaîne repose sur l'établissement des métamodèles **SimplePDL** (étendu avec les ressources) et `Réseau de Petri`, sur la définition de contraintes de sémantique statique, et sur la mise en place de transformations Modèle-à-Modèle (M2M, en **Java** et **ATL**) et Modèle-à-Texte (M2T, en **Acceleo**) pour l'intégration de l'outil de *model-checking* **Tina**. Le travail réalisé est analysé en détail, en précisant les étapes d'implémentation et les résultats obtenus.

Table des matières

1 Introduction Générale **3**
1.1 Objectifs et Architecture de la Chaîne 3

2 Phase 1 : Définition Abstraite et Statique des Langages **4**
2.1 Définition des Métamodèles 4
2.1.1 Métamodèle SimplePDL étendu avec Ressources 4
2.1.2 Métamodèle des Réseaux de Petri 5
2.2 Sémantique Statique des Modèles 5
2.2.1 Contraintes sur SimplePDL avec Ressources 5

3 Implémentation des Contraintes Statiques en Java **5**

4 Sémantique Statique des Réseaux de Petri en Java **7**

5 Phase 2 : Définition des Syntaxes Concrètes **9**
5.1 Syntaxe Concrète Textuelle avec Xtext 9
5.2 Éditeur Graphique avec Sirius 9
5.2.1 Mappings Graphiques et Styles 9
5.2.2 Outils de la Palette 9

6 Description et Analyse des Modèles de Processus **10**
6.1 Analyse de la Structure du Modèle 10

7 Phase 3 : Transformation Modèle-à-Modèle (M2M) **10**
7.1 Stratégie de Transformation SimplePDL → PetriNet 10
7.1.1 Modélisation de l'État d'Activité 10
7.1.2 Modélisation des Ressources et de leur Usage 11
7.1.3 Modélisation du Séquencement 11
7.2 Implémentation en EMF/Java 11
7.3 Implémentation en ATL 13

8 Phase 4 : Transformation M2T et Vérification **13**
8.1 Transformation PetriNet → Tina avec Acceleo 13
8.2 Transformation PetriNet → DOT avec Acceleo 13

9 Vérification Formelle par Transformation SimplePDL vers LTL **15**

10 Conclusion **16**

11 Tableau des documents rendus **16**

1 Introduction Générale

Le mini-projet s'inscrit dans le cadre de la vérification formelle des processus métier, un domaine où les modèles jouent un rôle central. L'approche choisie utilise la puissance des Réseaux de Petri pour modéliser le comportement concurrent et les états d'un processus SimplePDL.

1.1 Objectifs et Architecture de la Chaîne

L'objectif principal est de construire une chaîne de vérification, articulée autour des éléments suivants :

(a) **Modélisation Abstraite et Concrète :** Définition des langages SimplePDL et Réseau de Petri et création d'éditeurs graphiques et textuels.

(b) **Cohérence :** Application de règles de sémantique statique pour garantir la validité des modèles d'entrée.

(c) **Transformation:** Transformation du modèle SimplePDL vers le modèle `Réseau de Petri`, en gérant le séquencement et les contraintes de ressources.

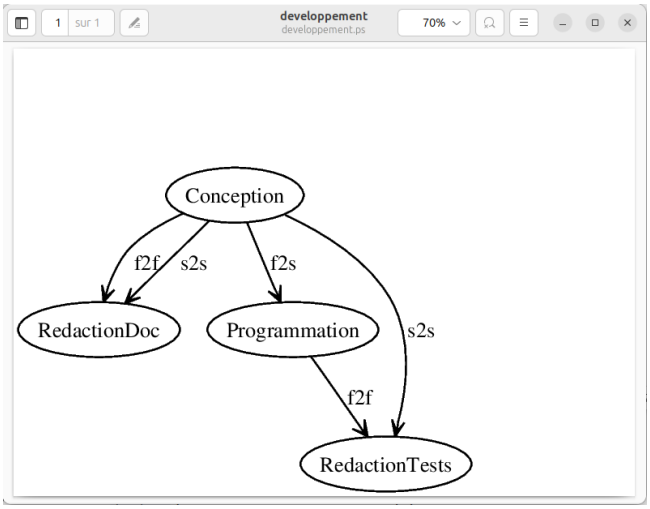
(d) **Vérification :** Traduction du modèle Réseau de Petri en format **Tina** et génération des propriétés LTL pour valider la terminaison et les invariants de sûreté.

2 Phase 1 : Définition Abstraite et Statique des Langages

Cette phase a établi les fondations formelles de la chaîne.

2.1 Définition des Métamodèles

Voici un exemple de modèle comportant les activités : Conception , RedactionDoc , Programmation, RedactionTests



2.1.1 Métamodèle SimplePDL étendu avec Ressources

L'extension principale modélise les ressources nécessaires aux activités. Nous avons introduit les classes `Resource` et `UsageResource` pour lier les besoins de l'activité au stock global. Le diagramme Ecore de notre métamodèle SimplePDL (avec l'extension ressource) est illustré ci-dessous :

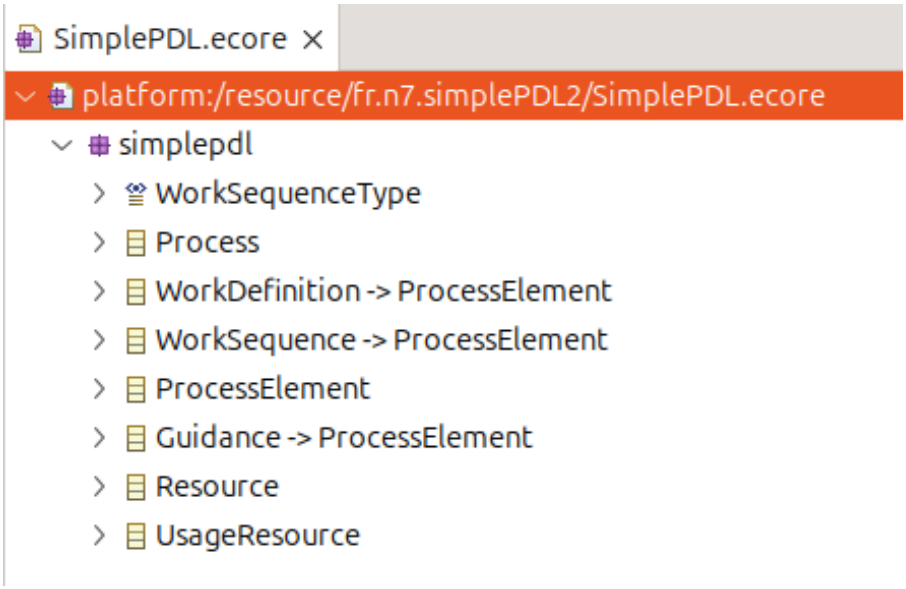


Figure 1: Ecore SimplePDL

- **Classes Clés :** `Resource` (stock global), `UsageResource` (besoin spécifique par activité).
- **Relation :** `WorkDefinition` référence `UsageResource` via `usageResources`, qui lui-même référence la `Resource` globale consommée.

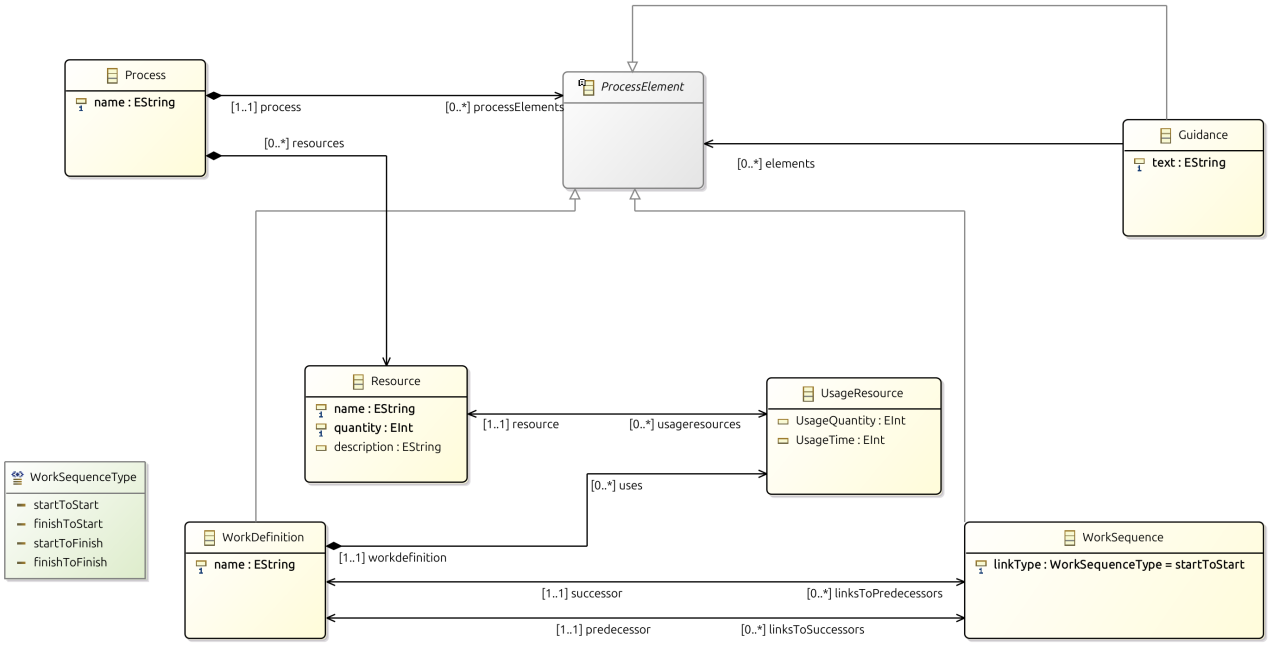


Figure 2: Schéma du Métamodèle SimplePDL (avec Ressources)

2.1.2 Métamodèle des Réseaux de Petri

Le métamodèle cible utilise des classes d'Arc spécialisées pour respecter la bipartition stricte des RdP (Place ↔ Transition), comme suggéré par l'analyse des contre-exemples.

- **Classes d'Arc :** Nous avons utilisé `ArcPlaceTransition` (Arc Entrant vers T) et `ArcTransitionPlace` (Arc Sortant de T).
- Ceci garantit que la source et la cible d'un arc sont toujours d'un type valide, évitant les erreurs de bipartition.

2.2 Sémantique Statique des Modèles

La sémantique statique a été implantée en **Java** (`SimplePDLValidator.java`) en s'appuyant sur l'API des Streams.

2.2.1 Contraintes sur SimplePDL avec Ressources

L'implémentation Java permet de garantir la validité des modèles avant transformation. Voici un extrait de l'approche utilisée pour la contrainte de l'unicité des noms de `WorkDefinition` :

3 Implémentation des Contraintes Statiques en Java

La validation statique du DSML `SimplePDL` est réalisée en Java à l'aide de la classe `SimplePDLValidator`, qui utilise l'API des *streams* pour évaluer les invariants. Voici les contraintes que nous avons ajouté pour Resource et UsageResource

- **Classe `Resource` :**
  - Le nom de la ressource doit être unique au sein du processus.
  - La quantité (`quantity`) de la ressource doit être strictement positive (> 0).

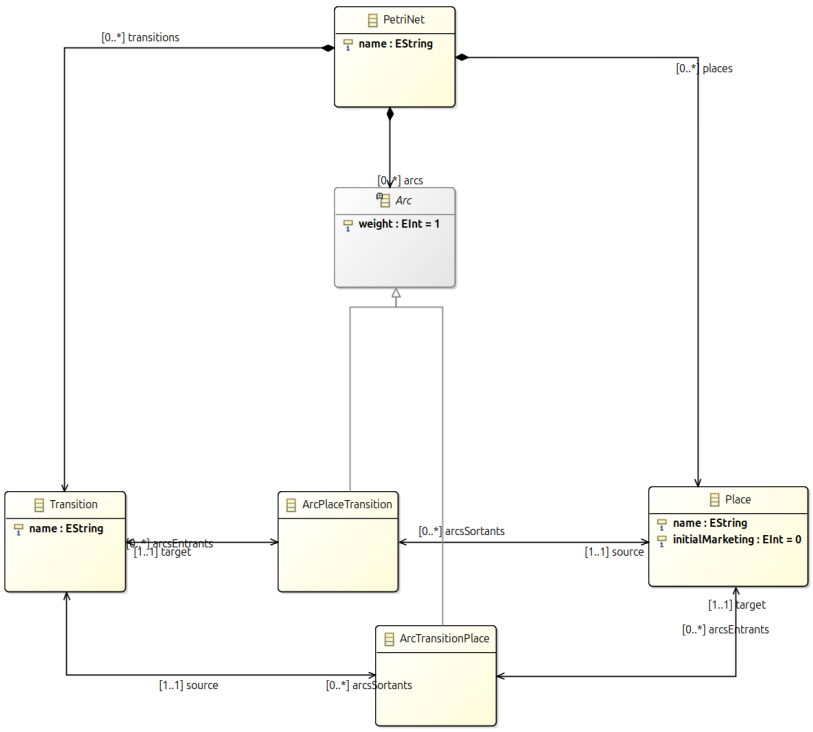


Figure 3: Schéma du Métamodèle Réseau de Petri

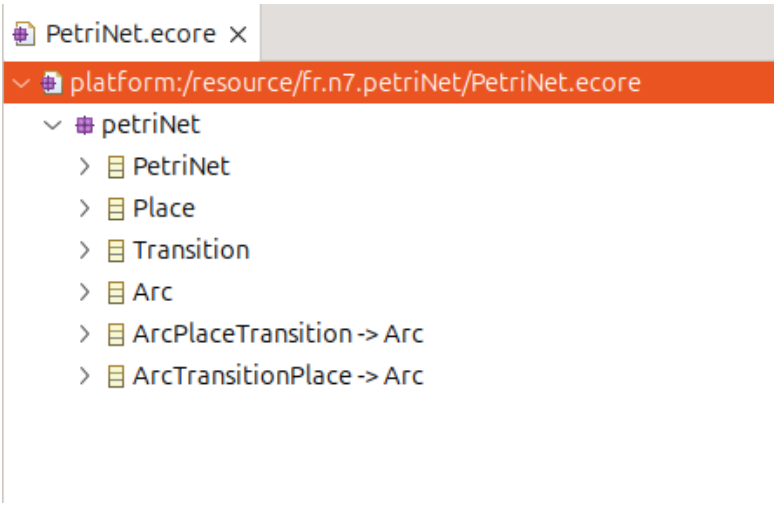


Figure 4: Ecore Petrinet

- **Classe `UsageResource` :**
  - La quantité utilisée (`usageQuantity`) doit être strictement positive (> 0).
  - La ressource référencée doit être définie dans la liste des ressources du processus parent.

L'exemple de code ci-dessous illustre l'implantation de la contrainte d'unicité du nom des ressources dans la méthode `caseResource` du validateur.

Des exemples de modèles invalides seront fournis dans le rendu pour tester ce programme (Comme Resource Nom Unique.xmi , UR Quantité.xmi etc)

Listing 1: Implémentation Java de la contrainte d'unicité (dans SimplePDLValidator.java)

```
1
2    @Override
3    public Boolean caseResource(simplepdl.Resource resource) {
4
5
6        boolean quantityIsPositive = resource.getQuantity() > 0;
7        this.result.recordIfFailed(
8            quantityIsPositive,
9            resource,
10           "La quantit de la ressource '" + resource.getName() + "' doit tre strictement
                 positive (actuel: " + resource.getQuantity() + ")."
11       );
12
13       // Contrainte 1 : Unicit du nom de la ressource.
14       simplepdl.Process process = (simplepdl.Process) resource.eContainer();
15
16       if (process != null) {
17
18           long count = process.getResources().stream()
19               .filter(r -> r.getName() != null && r.getName().equals(resource.getName()))
20               .count();
21
22           // Le nom doit apparatre une seule fois
23           if (count > 1) {
24               this.result.recordError(resource,
25                   "La ressource '" + resource.getName() + "' n'a pas un nom unique dans le
                         processus."
26               );
27           }
28       }
29
30       return null;
31   }
```

4 Sémantique Statique des Réseaux de Petri en Java

La validation statique du DSML des Réseaux de Petri est implantée dans la classe `PetrinetValidator`, qui hérite de `PetriNetSwitch` pour parcourir le modèle. Les contraintes vérifiées sont les suivantes :

- **Classe `PetriNet` :**
  - Les noms des `Places` doivent être uniques entre eux.
  - Les noms des `Transitions` doivent être uniques entre elles.

- Un nom ne peut pas être utilisé à la fois par une `Place` et par une `Transition` (collision inter-types).

- **Classe `Place` :**
  - Le marquage initial (`initialMarketing`) doit être non négatif(≥ 0).

- **Classe `Arc` :**
  - Le poids de l'arc (`weight`) doit être strictement positif (≥ 1).

- **Classes `ArcPlaceTransition` et `ArcTransitionPlace` (Contraintes Spécifiques) :**
  - L'arc doit relier des nœuds du type attendu (`Place` $\rightarrow$ `Transition` ou `Transition` $\rightarrow$ `Place`).

L'exemple de code ci-dessous illustre l'implantation de la contrainte d'unicité des noms (intra-types et inter-types) au niveau du réseau dans la méthode `casePetriNet`.

Listing 2: Implémentation Java de la contrainte d'unicité des noms (dans PetrinetValidator.java)

```
    @Override
    public Boolean casePetriNet(PetriNet net) {

        // 1. Unicit des noms (Places et Transitions) au niveau du rseau
        List<String> placeNames = net.getPlaces().stream()
            .map(Place::getName)
            .collect(Collectors.toList());

        List<String> transitionNames = net.getTransitions().stream()
            .map(Transition::getName)
            .collect(Collectors.toList());

        // Vrifier les doublons internes aux Places et Transitions
        if (placeNames.size() != new HashSet<>(placeNames).size()) {
            result.recordError(net, "Des noms de Places ne sont pas uniques dans le rseau.");
        }
        if (transitionNames.size() != new HashSet<>(transitionNames).size()) {
            result.recordError(net, "Des noms de Transitions ne sont pas uniques dans le
                rseau.");
        }

        // Vrifier les collisions entre Places et Transitions
        Set<String> placeNameSet = new HashSet<>(placeNames);
        for (String tName : transitionNames) {
            if (placeNameSet.contains(tName)) {
                result.recordError(net,
                    "Le nom '" + tName + "' est utilis  la fois par une Place et une
                        Transition.");
            }
        }


        net.getPlaces().forEach(this::doSwitch);
        net.getTransitions().forEach(this::doSwitch);
        net.getArcs().forEach(this::doSwitch);

        return null;
    }
```

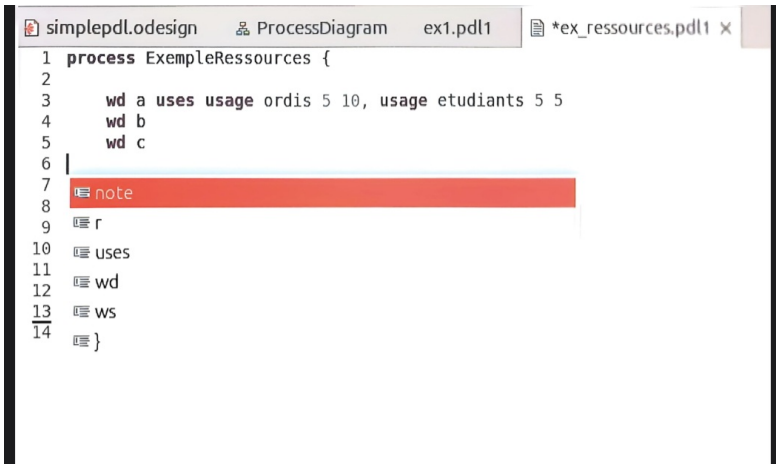
5 Phase 2 : Définition des Syntaxes Concrètes

Cette phase a rendu les modèles utilisables par l'humain à travers des interfaces de saisie et de visualisation adaptées, et signale les erreurs en temps réel.

5.1 Syntaxe Concrète Textuelle avec Xtext

On a utilisé Xtext pour définir facilement une syntaxe textuelle claire pour `SimplePDL`. En écrivant une grammaire, l'outil nous a automatiquement donné le méta-modèle et un éditeur complet.

L'éditeur gère la coloration syntaxique, l'auto-complétion.



5.2 Éditeur Graphique avec Sirius

L'éditeur graphique a été développé avec **Eclipse Sirius**, il offre une visualisation schématique, intuitive et interactive des processus `SimplePDL`. Il a été enrichi pour supporter les extensions du méta-modèle (`Resource`, `UsageResource` et `Guidance`).

5.2.1 Mappings Graphiques et Styles

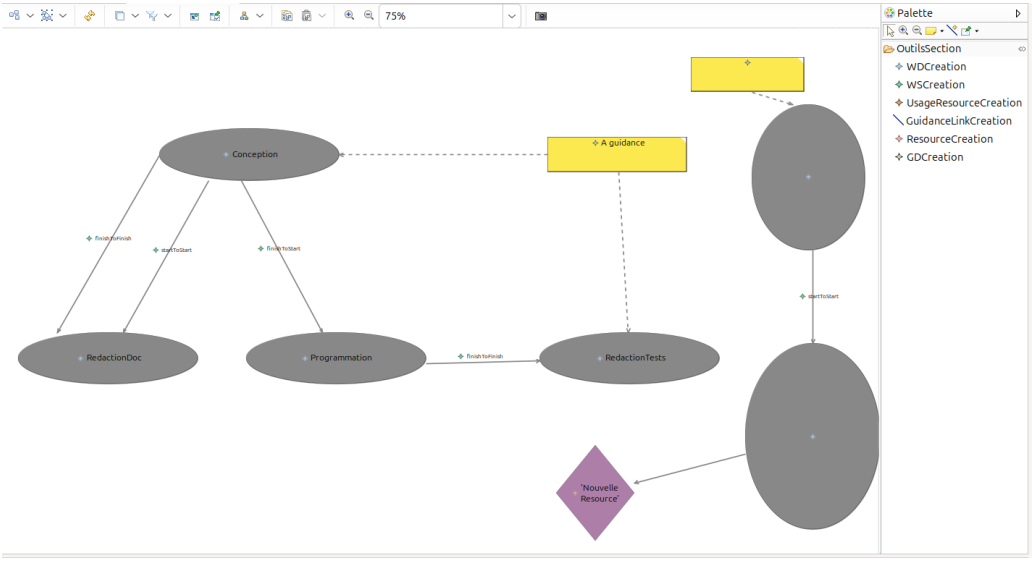
- **WorkDefinition Mapping** : Représenté par une ellipse.
- **WorkSequence Mapping** : Représenté par un arc.
- **Resource Mapping** : La nouvelle entité `Resource` est représentée graphiquement par un losange, permettant de la distinguer clairement des activités.
- **Guidance Mapping** : L'élément `Guidance` est représenté par un rectangle avec un fond de couleur jaune et est relié à l'élément de processus concerné par une ligne pointillée.

5.2.2 Outils de la Palette

La palette d'outils permet la création rapide de tous les éléments du modèle :

- **Création d'éléments** : Inclut les outils pour créer les activités (`WDCreation`), les liens de séquence (`WSCreation`), les ressources (`ResourceCreation`) et les guidances.
- **Création des liens** : Inclut les outils pour lier les séquences de travail (`WSCreation`) et les liens de guidance (`GuidanceLinkCreation`).

- **Gestion des ressources** : L'édition des ressources requises par une activité (`UsageResource`) se fait via la vue *Properties* de la `WorkDefinition` concernée, après avoir créé le lien d'usage (`UsageResourceCreation`) sur le diagramme.



6 Description et Analyse des Modèles de Processus

6.1 Analyse de la Structure du Modèle

Pour illustrer le concept de processus dans le cadre de SimplePDL, un modèle d'exemple a été établi (Figure 5). Il se compose de quatre composantes de travail primaires : `WorkDefinitions`, incluant `Conception`, `RédactionDoc`, `Programmation` et `RédactionTests`. L'enchaînement et les conditions d'activation entre ces activités sont régis par quatre types de dépendances de séquencement :(`WorkSequenceType`) ,(`startToStart`), (`startToFinish`), (`finishToStart`) et (`finishToFinish`).

7 Phase 3 : Transformation Modèle-à-Modèle (M2M)

La transformation du modèle `SimplePDL` vers le modèle `PetriNet` assure la traduction des concepts de gestion de processus (séquencement d'activités, états et consommation de ressources) vers la sémantique formelle des Réseaux de Petri.

7.1 Stratégie de Transformation SimplePDL → PetriNet

La stratégie repose sur une modélisation fine de l'état de chaque élément :

7.1.1 Modélisation de l'État d'Activité

Chaque `WorkDefinition` du modèle `SimplePDL` est décomposée en un motif de Réseau de Petri pour capturer son cycle de vie. Ce motif comprend :

- **Quatre Places (4)** :
  1. `PReady` (`initialMarketing`=1) : Représente l'état « prêt à démarrer ».
  2. `PStarted` : Représente l'état « démarré ».
  3. `PRunning` : Représente l'état « en cours d'exécution ».

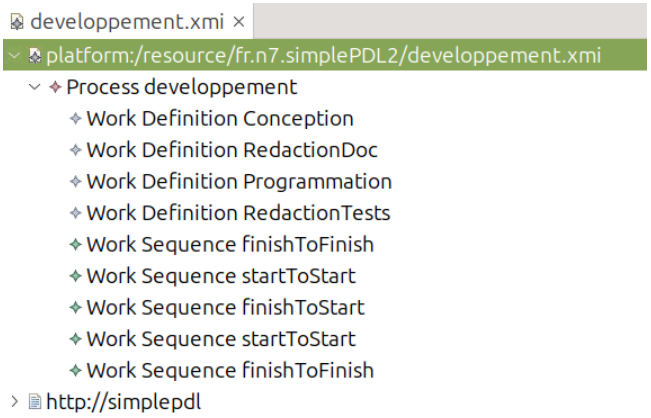


Figure 5: Exemple de modèle SimplePdl

4. `PFinished` : Représente l'état « terminé ».

- **Deux Transitions (2)** : `TStart` et `TFinish`.
- **Cinq Arcs Internes (5)** : Ils lient ces places et transitions, modélisant les transitions d'état (`PReady` → `TStart` → `PStarted` et `PRunning` ; `PRunning` → `TFinish` → `PFinished`).

7.1.2 Modélisation des Ressources et de leur Usage

- Chaque `Resource` de `SimplePDL` est transformée en une seule `Place` dans le Réseau de Petri. Le marquage initial de cette `Place` est directement égal à la quantité (`quantity`) de la ressource.
- L'`UsageResource` est traduite par un couple d'arcs :
  1. Un **Arc Place → Transition** : de la `Place` Ressource vers la transition `TStart` de l'activité, consommant la quantité requise (`usageQuantity`).
  2. Un **Arc Transition → Place** : de la transition `TFinish` vers la `Place` Ressource, libérant la même quantité à la fin de l'activité.

7.1.3 Modélisation du Séquencement

Les `WorkSequence` sont converties en arcs de dépendance entre les places d'état des activités et les transitions de début ou de fin des activités :

- Les `WorkSequenceType` (`f2s`, `s2s`, `f2f`, `s2f`) sont toutes traduites par un **Arc Place → Transition** de poids 1, liant l'état de la tâche prédécesseur (`PStarted` ou `PFinished`) à la transition de début ou de fin de la tâche successeur (`TStart` ou `TFinish`).

7.2 Implémentation en EMF/Java

L'implémentation de la transformation en Java nécessite une manipulation explicite des objets et des références EMF. L'étape cruciale a été l'utilisation de plusieurs structures de type `Map`, notamment :

- `wdToStartTransition` et `wdToFinishTransition` : Pour accéder rapidement aux transitions de début/fin d'une `WorkDefinition` lors de la création des arcs de séquencement.
- `wdToStartedPlace`, `wdToFinishedPlace`, etc. : Pour accéder aux places d'état d'une activité.
- `resourceToPlace` : Pour associer chaque `Resource` du modèle source à la `Place` correspondante dans le modèle cible, ce qui est indispensable pour créer les arcs de consommation/libération de ressources.

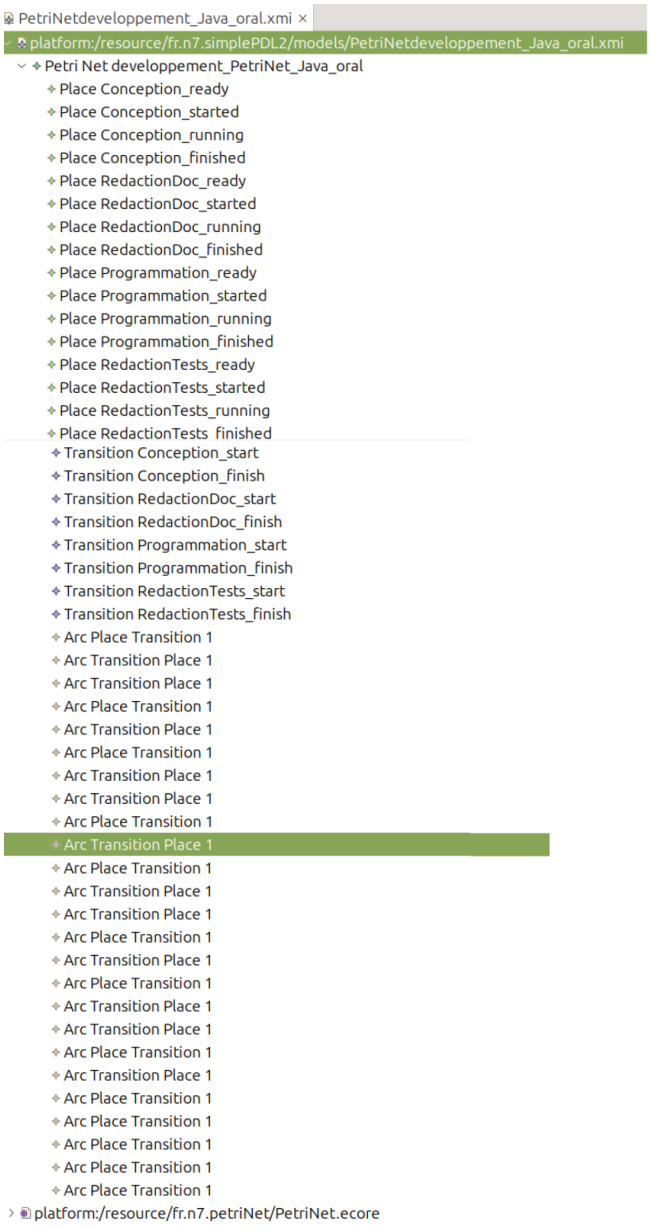


Figure 6: Résultat pour la transformation Java (Identique à celui d'ATL

7.3 Implémentation en ATL

L'implémentation de la transformation SimplePDL → PetriNet a été ensuite en ATL, un langage déclaratif qui utilise des règles pour définir la traduction des éléments du modèle source vers le modèle cible. Cette approche simplifie grandement l'écriture comparativement à une implémentation purement Java.

8 Phase 4 : Transformation M2T et Vérification

Cette phase finalise la chaîne pour l'outil Tina.

8.1 Transformation PetriNet → Tina avec Acceleo

La transformation vers la syntaxe de Tina (`.net`) à partir du modèle `PetriNet` est réalisée par le modèle Acceleo `toTina.mtl`. Ce processus consiste à :

1. Représenter chaque **place** par le mot-clé `pl`, suivi de son nom, puis du nombre de jetons initial entre parenthèses .

2. Convertir les **transitions** en tenant compte des poids associés aux arcs entrants et sortants.

Le modèle Acceleo itère sur les collections de places et de transitions pour générer le fichier `.net` complet.

Le résultat est donc celui des figures suivantes.

Figure 7: Modèle Developpement avec syntaxe de Tina (.net

8.2 Transformation PetriNet → DOT avec Acceleo

La seconde transformation M2T pour les réseaux de Petri est dédiée à la production d'un fichier `.dot`, syntaxe utilisée par l'outil **Graphviz** pour la visualisation graphique.Le template Acceleo `toDot.mtl` prend le modèle `PetriNet` et génère une structure de graphe dirigé (`digraph`). Dans cette représentation, les places et les transitions deviennent des **nœuds** du graphe, tandis que les arcs du réseau de Petri sont traduits en **flèches** (arêtes) entre ces nœuds.

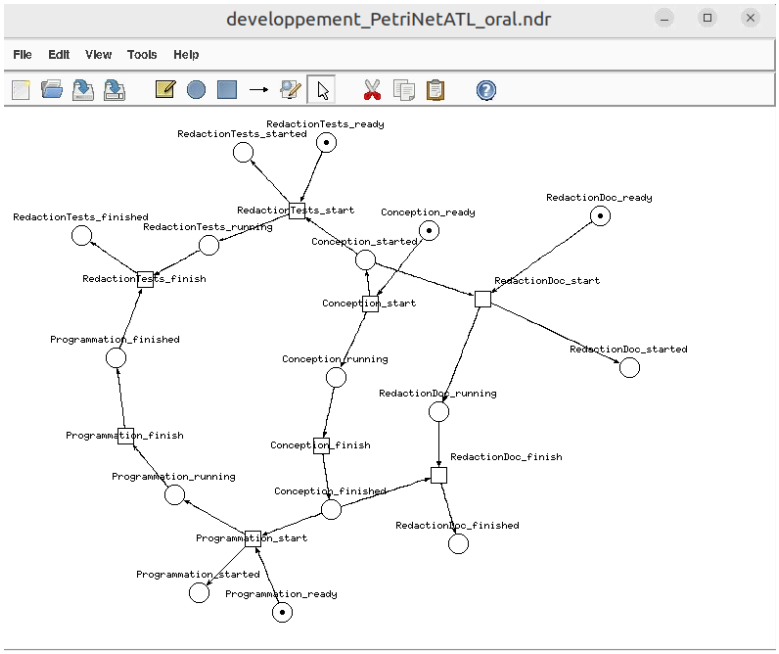


Figure 8: Résultat graphique de la transformation M2T

developpement.dot

```
digraph developpement {
        Conception -> RedactionDoc [arrowhead=vee label=f2f]
        Conception -> RedactionDoc [arrowhead=vee label=s2s]
        Conception -> Programmation [arrowhead=vee label=f2s]
        Conception -> RedactionTests [arrowhead=vee label=s2s]
        Programmation -> RedactionTests [arrowhead=vee label=f2f]
}
```

Figure 9: Résultat de Developpement avec ToDot

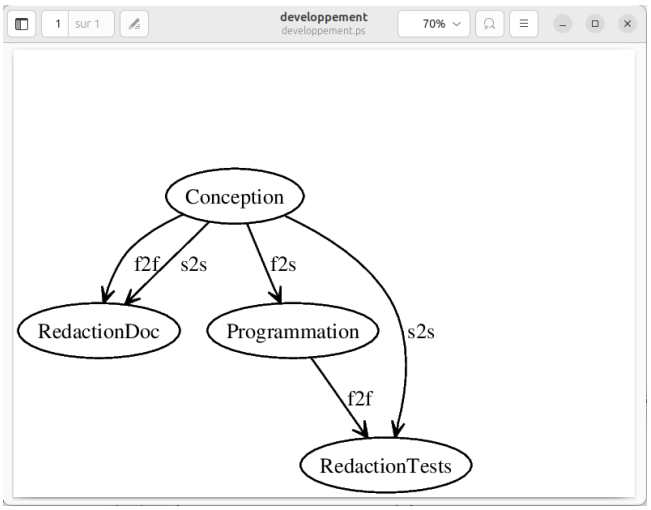


Figure 10: Résultat graphique ToDot

9 Vérification Formelle par Transformation SimplePDL vers LTL

La **conversion SimplePDL en LTL** (Logique Temporelle Linéaire) est essentielle pour la **vérification formelle** des modèles de processus. Cette transformation produit des **formules LTL** vérifiables par l'outil **Selt de Tina**.

Ces propriétés ciblent des aspects cruciaux tels que la **terminaison** des processus, le respect des **invariants** (par exemple, l'exclusivité des états d'activité), et la **continuité** des processus démarrés.

Nous employons deux types de transformations vers LTL:

1. Une dédiée à la vérification de la **terminaison du processus**.
2. Une seconde assurant le respect des **règles d'intégrité (invariants)** de SimplePDL.

Ces formules valident différents aspects du modèle de réseau de Petri.

```
 selt developpement_PetriNetATL_oral.ktz test.ltl

Selt version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 15 states, 20 transitions
0.001s
TRUE
0.000s
TRUE
0.000s
TRUE
0.000s
TRUE
0.000s
TRUE
0.000s
TRUE
0.000s
TRUE
0.000s
TRUE
```

Figure 11: Résultats de vérification des propriétés LTL correspondant aux invariants

```
 selt developpement_PetriNetATL_oral.ktz terminaison.ltl
Selt version 3.4.4 -- 01/05/16 -- LAAS/CNRS
ktz loaded, 15 states, 20 transitions
0.001s
operator finished : prop
0.000s
TRUE
0.000s
TRUE
0.000s
FALSE
  state 0: Conception_ready Programmation_ready RedactionDoc_rea
dy RedactionTests_ready
  -Conception_start ... (preserving T)->
  state 5: L.dead Programmation_finished Programmation_started R
edactionDoc_running RedactionDoc_started RedactionTests_ready
  -L.deadlock ... (preserving (dead /\ - RedactionTests_finished
) \/ (dead /\ - Programmation_finished) \/ (dead /\ - Conception
_finished) \/ dead /\ - RedactionDoc_finished)->
  state 6: L.dead Programmation_finished Programmation_started R
edactionDoc_running RedactionDoc_started RedactionTests_ready
  [accepting all]
0.001s
TRUE
0.000s
```

Figure 12: Résultats de vérification des propriétés LTL correspondant à la terminaison

10 Conclusion

Le mini-projet a permis de maîtriser l'ensemble des technologies de l'IDM autour d'une problématique de vérification concrète. La chaîne de vérification construite, allant de la modélisation abstraite Ecore à la vérification formelle (Tina/LTL), est robuste et met en évidence la puissance des transformations de modèles. L'extension pour la gestion des ressources a complexifié significativement les transformations M2M et la vérification, prouvant ainsi la capacité de la chaîne à traiter des processus métiers sophistiqués.

11 Tableau des documents rendus

Table 1: Association des Documents Requis aux Fichiers du Projet

Document Requis	**Fichiers Correspondants**
D_1: Les métamodèles SimplePDL et PetriNet	`SimplePDL.ecore`, `PetriNet.ecore`, `simplepdl_class_diagram.png`, `petriNet.png`
D_2: Fichiers Java des contraintes et contres exemples	`SimplePDLValidator.java`, `PetrinetValidator.java`, `Resource_Nom_Unique.xmi`, `Resource_Quantite.xmi`, `UR_Quantite.xmi`, `UR_Res_inexistant.xmi`, `Place_NomUnique.xmi`, `Place_InitialMarking.xmi`, `Arc_Poids.xmi`, `PT_Nom.xmi`, `WS.xmi`
D_3: Le code Java de la transformation modèle à modèle	`SimplePDLToPetrinet.java`
D_4: Le code ATL de la transformation modèle à modèle	`SimplePDLToPetriNet.atl`
D_5: Le code Acceleo des transformations modèle à texte	Fichiers toTina.mtl et toDot.mtl dans le dossier PetriNetTo, et fichier toDot.mtl dans le dossier SimplePDLTo.
D_6: Les modèles Sirius	`simplepdl.odesign`
D_7: Le modèle Xtext	ex_ressouces.pdl1 , PDL1.xtext
D_8: Des exemples de modèles de processus	`developpement.simplepdl`, `ex_ressources.pdl1`, `Valide.xmi` (Modèle SimplePDL valide), `PetriNetValide.xmi` (Modèle PetriNet valide)