Une chaîne de vérification pour modèles de procédés

Systèmes de confiance

Groupe B: Ines BESBES & Sara ROOL

5 ModIA, 2024-2025

June 26, 2025

Contents

1 Introduction

Ce projet vise à développer un écosystème permettant aux utilisateurs de définir et de manipuler des modèles de procédés. Ces modèles, composés de tâches, de ressources et de dépendances, sont essentiels pour représenter et analyser des processus complexes dans divers domaines.

L'objectif principal de ce projet est de rendre accessible la modélisation de procédés à des utilisateurs sans expertise particulière en informatique, en développant des outils ergonomiques et intuitifs.

Pour atteindre cet objectif, on a adopté une approche basée sur l'ingénierie dirigée par les modèles (IDM). Cette méthodologie permet de rester proches des besoins des utilisateurs tout en simplifiant le développement des outils nécessaires. On va se concentrer principalement sur deux métamodèles : `SimplePDL` et `PetriNet`.

Plusieurs syntaxes concrètes ont été intégrées pour la visualisation et l'édition des modèles. De même, on a mis en place des transformations permettant de les vérifier et les valider.

2 Métamodèles

2.1 SimplePDL

Le métamodèle `SimplePDL` permet de décrire des processus sous forme de tâches, dépendances, et ressources.

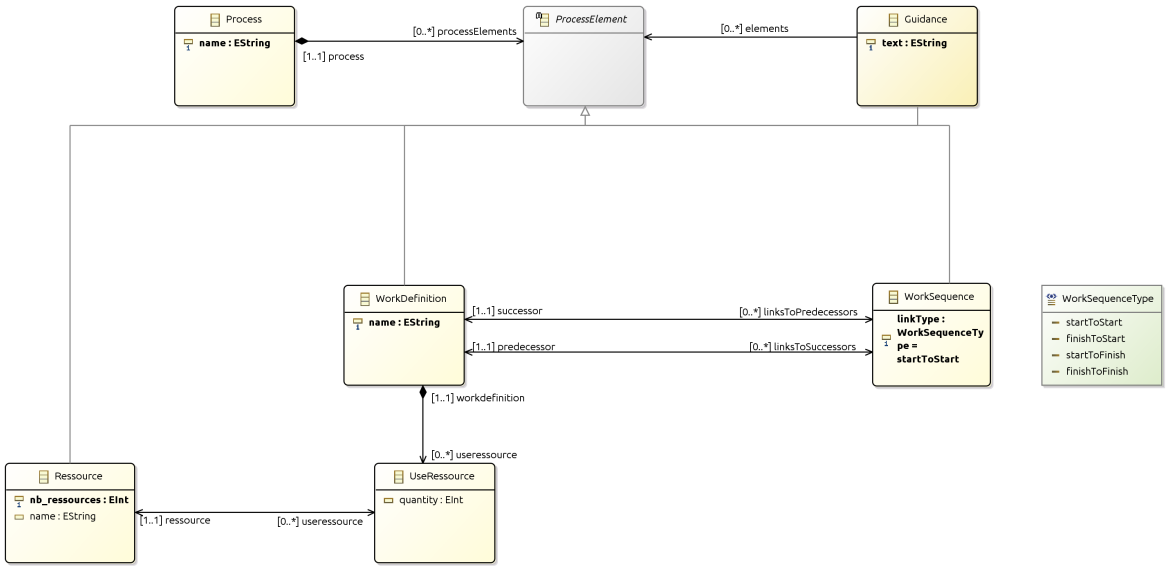


Figure 1: Métamodèle de SimplePDL

Il est structuré comme suit:

- `Process`: représente un processus complet, identifié par un nom (`name`). Contient un ensemble d'éléments de processus (`processElements`).
- `ProcessElement` (classe abstraite): classe mère de tous les éléments appartenant à un processus comme des :
  - `WorkDefinition`
  - `WorkSequence`
  - `Guidance`
  - `Ressource`

  Chaque `ProcessElement` appartient à exactement un processus.
- `WorkDefinition`: représente une tâche ou une activité à exécuter avec un nom (`name`). Chaque `WorkDefinition` peut être liée à plusieurs prédécesseurs/successeurs via des `WorkSequence`. Une `WorkDefinition` utilise des ressources via la classe `UseRessource`.
- `WorkSequence`: représente une relation entre deux `WorkDefinition`. Chaque `WorkSequence` est caractérisée par un attribut `linkType` de type énuméré `WorkSequenceType`:
  - `startToStart`
  - `finishToStart`
  - `startToFinish`
  - `finishToFinish`

Une `WorkSequence` a un prédécesseur et un succeseur.

- `Guidance`: contient un texte d'accompagnement (`text`). Peut être associé à un `ProcessElement` ou à rien du tout.
- `Ressource`: représente une ressource disponible dans le processus. Elle a les attributs suivants :
  - `name`
  - `nb_ressources` (quantité disponible)

  Elle est liée à plusieurs utilisations de ressources. En revanche, chaque `UseRessource` référence exactement une `Ressource`.
- `UseRessource`: spécifie la quantité (`quantity`) d'une `Ressource` à utiliser dans une `WorkDefinition`.

2.2 PetriNet

Le métamodèle de `PetriNet` représente un réseau de Petri, un modèle mathématique servant à représenter divers systèmes (informatiques, industriels...) travaillant sur des variables discrètes.

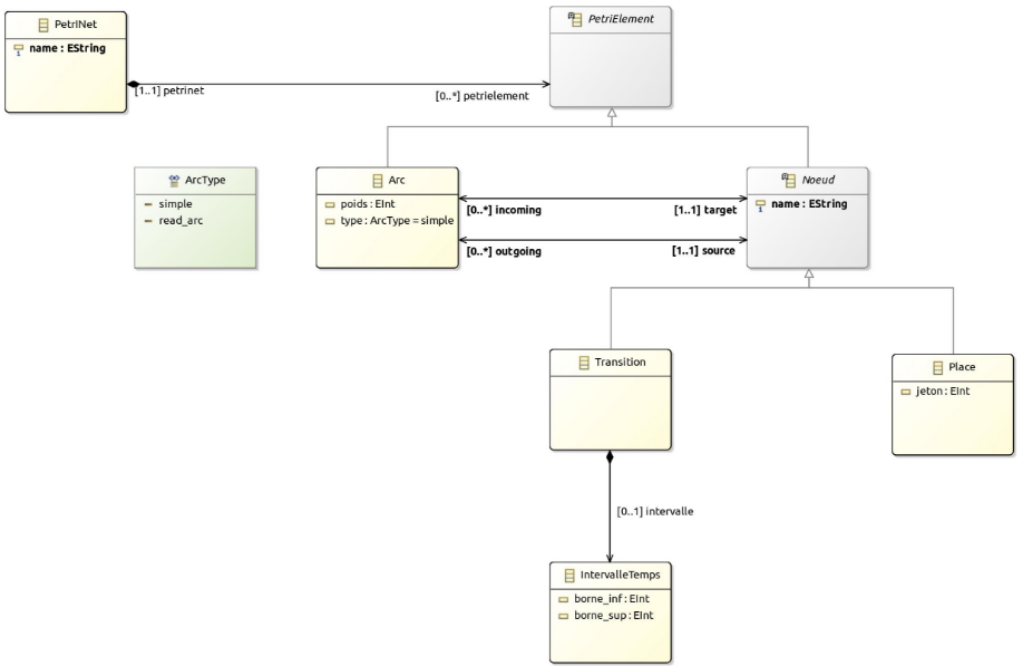


Figure 2: Métamodèle de PetriNet

Le métamodèle est organisé autour des éléments principaux suivants:

- Une racine `PetriNet` contenant l'ensemble des éléments (`PetriElement` qui est une classe abstraite),
- Des `noeuds` (places et transitions),
- Des `arcs` reliant les noeuds,
- Un système de typage pour les arcs,
- Des intervalles de temps pour les transitions (optionnel).

Voici une description détaillée des différents élements:

- `PetriNet` (racine):
  - Attribut: `name`, le nom du réseau,
  - Contient: 1 instance de PetriNet obligatoire et 0 à plusieurs éléments Petri.
- Types d'arcs `ArcType`:
  - `simple`: arc standard qui consomme des jetons,
  - `read_arc`: arc de lecture qui ne consomme pas de jetons.
- Classe `Arc`:
  - Attributs: `poids` qui est le nombre de jetons transportés et `type` qui est le type d'arc,

  - Connexions: 0 à plusieurs arcs entrants, 1 cible obligatoire, 0 à plusieurs arcs sortants et 1 source obligatoire,
  - Un arc lie deux noeuds.

- Classe `Noeud` (abstraite):
  - Attribut: `name`, le nom du noeud,
  - Sous-classes : `Transition` qui peut avoir 0 ou 1 intervalle de temps et `Place` qui a comme attribut `jeton` qui correspond au nombre de jetons dans la place.

- Classe `IntervalleTemps`:
  - Attributs: `borne_inf` qui est la borne inférieure de l'intervalle et `borne_sup` qui est la borne supérieure de l'intervalle (et qui peut être infinie).

3 Contraintes

Un métamodèle, par sa nature même, définit une structure et des règles de base qui imposent des contraintes sur les modèles qui en découlent. Toutefois, pour garantir la robustesse et la cohérence des modèles de processus, il est essentiel de rajouter des contraintes de validation, en plus de celles imposées par le métamodèle. Ces contraintes permettent de s'assurer que les modèles respectent les règles de nommage, d'unicité, et de cohérence logique, ce qui facilite leur implémentation et leur maintenance.

3.1 Contraintes sur SimplePDL

3.1.1 Description des contraintes

- Contrainte générale:
  - Les noms des classes `Process` et `ProcessElement` doivent respecter la convention Java,
- Contraintes sur les `WorkDefinition`:
  - Une `WorkDefinition` ne peut pas être liée à elle-même par une `WorkSequence`,
  - Le nom de chaque `WorkDefinition` doit être unique au sein du processus.
- Contraintes sur les `WorkSequence`:
  - Une `WorkSequence` doit toujours relier des `WorkDefinition` distinctes,
  - Il ne doit pas y avoir de `WorkSequence` dupliquées entre les mêmes `WorkDefinition` avec le même type de lien.
- Contrainte sur les `Guidance`:
  - Le texte de la `Guidance` ne doit pas être vide.
- Contraintes sur les `Ressource`:
  - La quantité de ressource demandée par une même activité (WorkDefinition) ne doit pas dépasser la quantité disponible pour cette ressource,
  - Le nombre de ressources disponibles doit être supérieur ou égal à 1.
- Contrainte sur les `UseRessource`:
  - La quantité de ressource utilisée doit être positive.

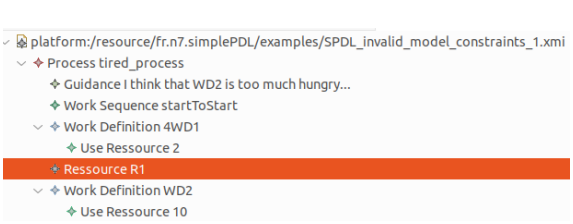
Voici un extrait de code pour une contrainte sur les `Ressource`:

```java
@Override
public Boolean caseRessource(Ressource object) {
    object.getUseressource().stream()
        .collect(Collectors.groupingBy(UseRessource::getWorkdefinition,
                 Collectors.summingInt(UseRessource::getQuantity)))
        .forEach((wd, totalQuantity) -> {
            this.result.recordIfFailed(
                totalQuantity <= object.getNb_ressources(),
                object,
                "L'activité \"" + wd.getName() + "\" demande " + totalQuantity +
                " unités de la ressource \"" + object.getName() +
                "\", ce qui dépasse la quantité disponible (" + object.getNb_ressources() + ")."
            );
        });


    this.result.recordIfFailed(
            object.getNb_ressources() >= 1,
            object,
        "Les ressources ne sont pas suffisantes. Il faut au minimum une ressource."
    );

    this.result.recordIfFailed(
            object.getName() != null && object.getName().matches(IDENT_REGEX),
            object,
            "Le nom de la ressource ne respecte pas les conventions Java."
        );
    return null;
}
```

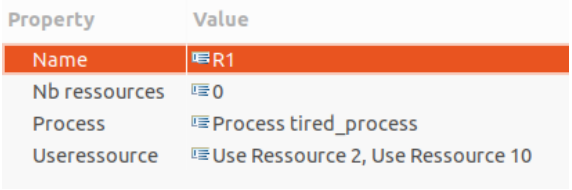
Figure 3: Code des contraintes imposées sur `Ressource`

3.1.2 Exemples

Exemple de modèle invalide par les contraintes:



(a) Arborescence du modèle invalide



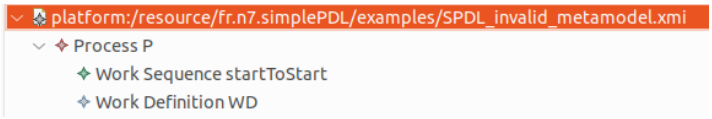
(b) Propriétés de R1 qui ne contient aucune ressource

Figure 4: Propriétés du modèle invalide

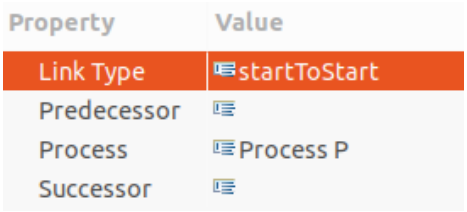
```
Résultat de validation pour examples/SPDL_invalid_model_constraints_1.xmi:
- Process: OK
- WorkDefinition: 1 erreurs trouvées
=> Erreur dans 4WD1 [simplepdl.impl.WorkDefinitionImpl@72a7c7e0 (name: 4WD1)]: Le nom de l'activité ne respecte pas les conventions Java
- WorkSequence: 1 erreurs trouvées
=> Erreur dans simplepdl.impl.WorkSequenceImpl@57c758ac (linkType: startToStart): La dépendance relie l'activité 4WD1 à elle-même
- Guidance: OK
- Ressource: 3 erreurs trouvées
=> Erreur dans R1 [simplepdl.impl.RessourceImpl@a9cd3b1 (nb_ressources: 0, name: R1)]: L'activité "4WD1" demande 2 unités de la ressource "R1", ce qui dépasse la quantité disponible (0).
=> Erreur dans R1 [simplepdl.impl.RessourceImpl@a9cd3b1 (nb_ressources: 0, name: R1)]: L'activité "WD2" demande 10 unités de la ressource "R1", ce qui dépasse la quantité disponible (0).
=> Erreur dans R1 [simplepdl.impl.RessourceImpl@a9cd3b1 (nb_ressources: 0, name: R1)]: Les ressources ne sont pas suffisantes. Il faut au minimum une ressource.
- UseRessource: OK
Fini.
```

Figure 5: Erreur dans le terminal due à l'absence de ressources et à une demande supérieure à l'offre

Exemple de modèle invalide par le métamodèle:



(a) Arborescence du modèle invalide



(b) Propriétés de WorkSequence qui n'est reliée à rien du tout (successor et predecessor sont vides)

Figure 6: Propriétés du modèle invalide

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "Object.equals(Object)" because the return value of "simplepdl.WorkSequence.getPredecessor()" is null
```

Figure 7: Erreur dans le terminal due au fait que la WorkSequence ne possède pas de successor ni de predecessor

3.2 Contraintes sur PetriNet

3.2.1 Description des contraintes

En plus des contraintes instanciées par le métamodèle, on définit les contraintes suivantes:

- Contraintes générales:
  - Les noms des classes `PetriNet` et `Noeuds` doivent respecter la convention Java,
  - Les noms des `Noeuds` doivent être unique au sein d'un même réseau.
- Contraintes sur les `Arcs`:
  - Un arc doit toujours relier des éléments de types différents (une place à une transition ou l'inverse),
  - Le poids d'un arc doit être positif ou nul. On ne peut pas consommer une valeur négative de jetons.
- Contraintes sur les `Transitions`:
  - Si un intervalle de temps est défini, sa bonne inférieure doit être inférieure à sa borne supérieure,
  - Si un intervalle de temps est défini, la borne inférieur doit être positive ou nulle.
- Contraintes sur les `Places`:
  - Le nombre de jetons dans une place doit être positif ou nul,
  - Une place ne peut pas être isolée, elle doit avoir au moins un arc entrant ou sortant.

Ces contraintes sont codées dans un fichier validation dont voici un extrait:

```java
@Override
public Boolean caseTransition(Transition object) {
    IntervalleTemps interval = object.getIntervalle();
    if (interval != null) {
        this.result.recordIfFailed(
            interval.getBorne_inf() <= interval.getBorne_sup() || interval.getBorne_sup() == -1 ,
            object,
            "Intervalle de temps invalide : borne inférieure > borne supérieure");
        this.result.recordIfFailed(
                interval.getBorne_inf() >= 0,
                object,
                "Intervalle de temps invalide : la borne inférieure doit être >=  0");


    }
    return super.caseTransition(object);
}
```

Figure 8: Code pour les contraintes de `Transitions`

3.2.2 Exemples

Exemple d'un modèle invalide vis-à-vis du métamodèle:

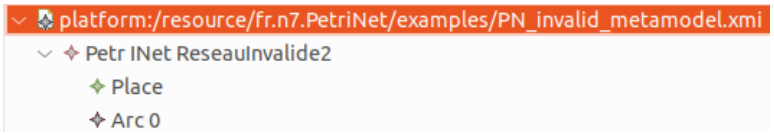


Figure 9: Arborescence du modèle invalide

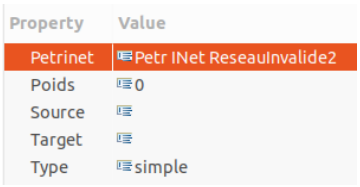


Figure 10: Fenêtre de propriété de l'arc invalide (sans source ni target)

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "Object.getClass()" because "source" is null
```

Figure 11: Erreur dans le terminal

Exemple d'un modèle invalide vis-à-vis d'une contrainte de transition (`borne_inf` > `borne_sup`):

Figure 12: Arborescence du modèle invalide

Figure 13: Fenêtre de propriété de l'intervalle invalide

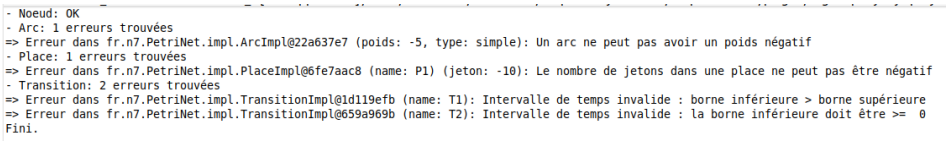


Figure 14: Erreur dans le terminal

4 Définition de syntaxes concrètes graphiques

On va à présent définir une syntaxe concrète graphique qui permet de fournir un moyen de visualiser et/ou éditer plus agréablement et efficacement un modèle. On va utiliser l'outil Eclipse Sirius. Il permet de définir une syntaxe graphique pour un langage de modélisation décrit en Ecore et d'engendrer un éditeur graphique intégré à Eclipse. On va considérer le métamodèle de `SimplePDL`.

4.1 Architecture du fichier `simplepdl.odesign`

Le fichier Sirius utilisé définit :

- un **viewpoint** nommé simplepdlViewpoint, avec un éditeur de diagramme nommé `ProcessDiagram` ;
- une couche principale (Default) contenant les éléments de base du processus ;
- une couche supplémentaire (GuidanceLayer) activée par défaut pour ajouter des annotations textuelles ;
- des **nodeMappings** et **edgeMappings** pour associer les éléments du métamodèle à des représentations graphiques ;
- des **tools** dans la palette pour manipuler ces éléments directement dans l'éditeur Sirius.

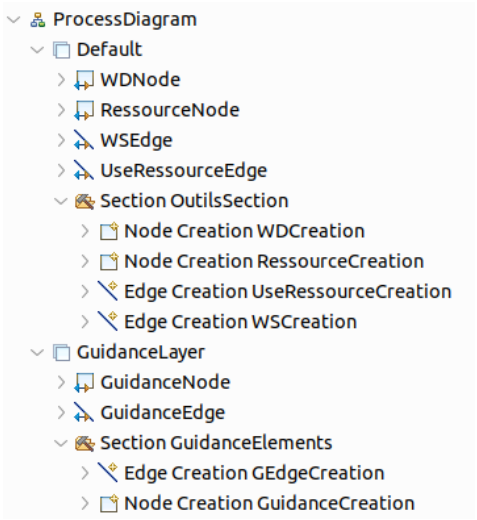


Figure 15: Arborescence de la syntaxe graphique

L'ensemble permet de modéliser graphiquement un processus `SimplePDL` avec ressources, dépendances et annotations, de manière claire et interactive.

4.2 Syntaxe graphique utilisée

Pour commencer, on a défini une seule représentation principale appelée `ProcessDiagram`, qui permet de représenter graphiquement les processus modélisés avec `SimplePDL`.

La syntaxe graphique repose sur une représentation des éléments principaux de `SimplePDL`:

- Les `WorkDefinition` sont représentées sous forme de losanges gris avec une étiquette centrale.

- Les `Ressource` sont représentées sous forme d'ellipses bleu clair.
- Les `Guidance` sont affichées sous forme de notes jaunes avec leur texte comme étiquette.
- Les `WorkSequence` sont représentées par des arêtes colorées en fonction du type de lien:
  - `startToStart`: rouge
  - `startToFinish`: violet clair
  - `finishToStart`: vert foncé
  - `finishToFinish`: orange foncé
- Les arêtes d'utilisation de ressource (`UseRessource`) sont en bleu foncé avec la quantité en label.
- Les liens de `Guidance` sont affichés avec un style de trait pointillé gris.

4.3 Définition de la palette

La palette permet d'instancier facilement les différents éléments du modèle. Elle contient:

- **WDCreation**: pour créer une activité (`WorkDefinition`)
- **RessourceCreation**: pour ajouter une ressource (`Ressource`)
- **UseRessourceCreation**: pour connecter une ressource à une activité via une relation d'utilisation (`UseRessource`)
- **WSCreation**: pour créer des liens de dépendance entre activités (`WorkSequence`)
- **GuidanceCreation**: pour insérer une note (`Guidance`)
- **GEdgeCreation**: pour lier une note à une ou plusieurs activités concernées

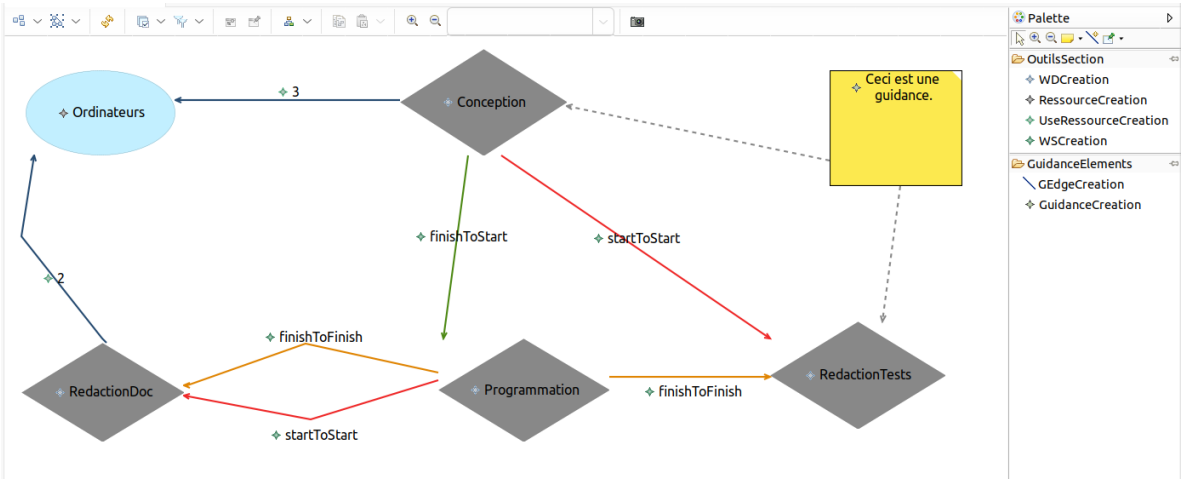


Figure 16: Palette d'outils et exemple de rendu graphique d'un modèle `SimplePDL`

5 Syntaxe textuelle

Dans cette section, on définit une syntaxe textuelle, grâce à Xtext, permettant de créer des modèles `.pdl1`, qui suivront un métamodèle `PDL1`. L'utilisation d'une syntaxe textuelle permet d'instancier autrement un modèle.

On définit la syntaxe textuelle comme suit:

Syntaxe textuelle pour `PDL1`

- Création d'un processus : `process` ⟨*nom_du_processus*⟩ { ... }
  - Création d'une `Ressources`: `resource` ⟨*nom_ressource*⟩ `amount` ⟨*quantité_totale*⟩
  - Création d'une `WorkDefinition`: `wd` ⟨*nom_wd*⟩
    - Indication des ressources nécessaires: `use` ⟨*quantité*⟩ `of` `<nom_ressource>`
  - Création d'une `WorkSequence` entre deux tâches: `ws` ⟨*type*⟩ `from` `<prédécesseur>` `to` `<successeur>`
    - avec ⟨*type*⟩ ∈ `s2s` | `f2s` | `s2f` | `f2f`
  - Ajouter une note au processus: `note` `"texte_entre_guillemets"`

```
process ex1 {

  resource resA amount 5
  resource resB amount 3

  wd a
    use 2 of resA
    use 1 of resB

  wd b
    use 1 of resA

  wd c

  ws s2s from a to b
  ws f2f from b to c

  note "Texte pour la guidance"
}
```

Figure 17: Exemple d'un modèle `.pdl1` défini par syntaxe textuelle

6 Transformation ATL modèle à modèle

La transformation modèle à modèle est réalisée à l'aide du langage ATL (Atlas Transformation Language). Ce langage permet de définir des règles de correspondance entre des éléments de deux métamodèles.

6.1 `PDL1` to `SimplePDL`

Cette section décrit une transformation modèle à modèle permettant de passer d'un `PDL1` à un `SimplePDL`. Tout d'abord voici, les différences majeures entre les deux métamodèles:

`PDL1`	`SimplePDL`
- plus simple et moins contraignant - moins de relations bidirectionnelles explicites - contient une classe `Guidance` avec juste un attribut texte	- plus complet et rigoureux - relations bidirectionnelles explicites (avec `eOpposite`) - contient une classe `Guidance` étendue avec une référence vers des `ProcessElement` - contraintes de cardinalité plus strictes (`lowerBound=1` sur plusieurs éléments)

Table 1: Comparaison entre `PDL1` et `SimplePDL`

6.1.1 Règles de transformation

Transformation directe: La majorité des règles de transformation est une copie directe des éléments avec des règles ATL simples. Par exemple:

```
rule XWD2SWD {
    from xwd : XPDL!WorkDefinition
    to swd : SPDL!WorkDefinition(
        name <- xwd.name,
        useressources <- xwd.useressources
        )
}
```

Figure 18: Code de la transformation ATL pour une `WorkDefinition`

Conversion de l'énumération `WorkSequenceType`: Une des différences syntaxiques/structurelles est la représentation des types de liens dans les dépendances `WorkSequenceType`. Elle est gérée par un helper. Ce helper est ensuite utilisé dans la transformation `XWS2SWS`.

```
helper def: convertLinkType(x : XPDL!WorkSequenceType) : SPDL!WorkSequenceType =
    if x = #s2s then  #startToStart
    else if x = #s2f then #startToFinish
    else if x = #f2s then  #finishToStart
    else #finishToFinish
    endif endif endif;
```

Figure 19: Code du helper `convertLinkType`

Association `UseRessources`: Dans `PDL1`, les `UseRessources` sont définies à l'intérieur des `WorkDefinition`, mais ne possèdent pas explicitement leur workdefinition d'origine. Dans `SimplePDL`, ce lien est nécessaire. Il est récupéré par un accès au conteneur.

Transformation des `Guidance`: Les éléments de type `Guidance` dans `PDL1` sont des annotations textuelles associées au processus, sans relation avec les autres éléments du modèle. Par conséquent, dans la transformation, ils sont mappés directement vers les `Guidance` de `SimplePDL`, en ne conservant que le texte, sans lien à d'autres `ProcessElement`.

6.1.2 Exemple

On effectue la transformation ATL sur le modèle précédemment crée 17 et on obtient un modèle `.simplepdl` avec l'arborescence suivante:

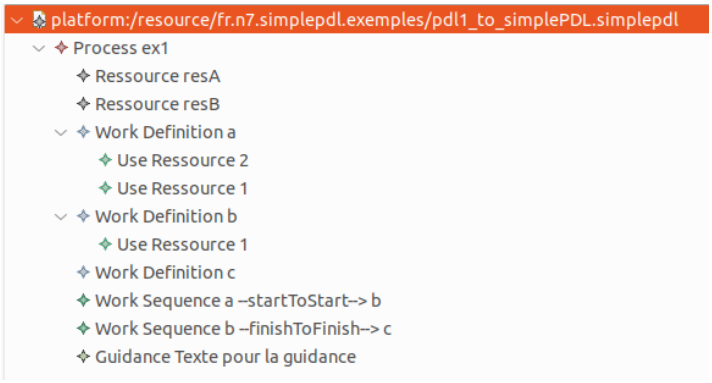


Figure 20: Arborescence d'un modèle `.simplepdl` crée à partir d'une transformation ATL d'un `.pdl1`

6.2 `SimplePDL` to `PetriNet`

Cette section décrit une transformation modèle à modèle permettant de passer d'un `SimplePDL` à un `PetriNet`.

6.2.1 Règles de transformation

Les règles suivantes définissent comment chaque élément du modèle `SimplePDL` est transformé dans le modèle `PetriNet`.

Process2PetriNet: Cette règle transforme un `Process` en un `PetrINet` avec le même nom

```
rule Process2PetriNet {
    from p : SimplePDL!Process
    to pn : PetriNet!PetrINet (
        name <- p.name
    )
```

Figure 21: Code `Process2PetriNet`

WorkDefinition2PetriNet: Chaque `WorkDefinition` est transformée en plusieurs éléments d'un réseau de Petri :

- 4 `Place` qui représentent les états possibles de la tâche:
  - `ready`: pour contrôler l'accès au démarrage,
  - `running`: pour le passage vers `finish`,

  - `started`: pour les dépendances startTo,
  - `finished`: état terminal de la tâche.
- 2 `Transition`: `start`, commencement de la tâche, et `finish`, fin de la tâche, qui modélisent les actions de commencer/terminer
- 5 `Arc`, de type `simple`, qui gèrent le passage de jetons d'un état à l'autre:
  - `ar_ready2start`: la tâche peut commencer seulement si elle est prête (`ready`),
  - `ar_start2running`: la tâche entre dans l'état en cours d'exécution (`running`),
  - `ar_start2started`: en parallèle de la place `running`, on ajoute un jeton dans la place `started` pour marquer que la tâche a commencé.
  - `ar_running2finish`:on peut terminer une tâche que si elle a été démarrée et est en cours.
  - `ar_finish2finished`: la tâche est terminée. C'est utile pour les dépendances finishTo.

WorkSequence2PetriNet: Les `WorkSequence` sont converties en `Arc` de type `read_arc`, connectant une `Place` du prédécesseur à une `Transition` du successeur, selon le type de lien (`startToStart`, `finishToFinish`, `finishToStart`, `startToFinish`).

Ressources2PetriNet: Chaque ressource devient une `Place` contenant un nombre initial de jetons égal à sa quantité disponible.

UseRessources2PetriNet: Chaque relation `UseRessource` donne lieu à deux arcs:

- un arc de type `simple`, de la `Place` ressource vers la `Transition` de début (chargement)
- un arc de type `simple`, de la `Transition` de fin vers la ressource (libération)

Ces arcs permettent de modéliser la consommation et la libération de ressources lors de l'exécution d'une activité.

6.2.2 Fonction auxiliaire

Une fonction helper permet de retrouver le `Process` parent de n'importe quel `ProcessElement`, elle est utilisée pour relier correctement les éléments générés dans le modèle `PetriNet`:

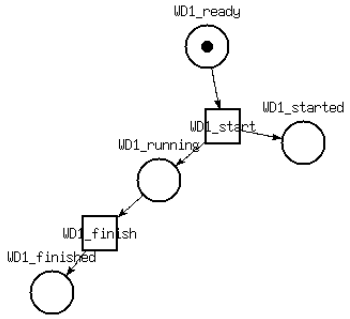
```
helper context SimplePDL!ProcessElement
def: getProcess() : SimplePDL!Process =
    SimplePDL!Process.allInstances()
        ->select(p | p.processElements->includes(self))
        ->asSequence()->first();
```

Figure 22: Code du helper

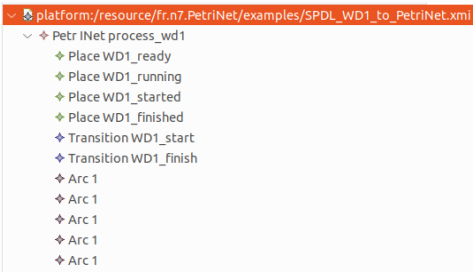
6.2.3 Exemples

Les réprésentations obtenues ci-dessous ont été faites avec `dot` et `Tina`. On decrira ces transformations dans la partie suivante: transformation modèle à texte.

(a) Exemple de modèle `.simplepdl`

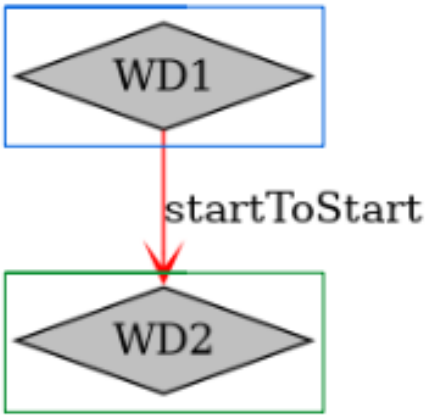


(b) Transformation du modèle `.simplepdl` en un modèle `.petrinet`

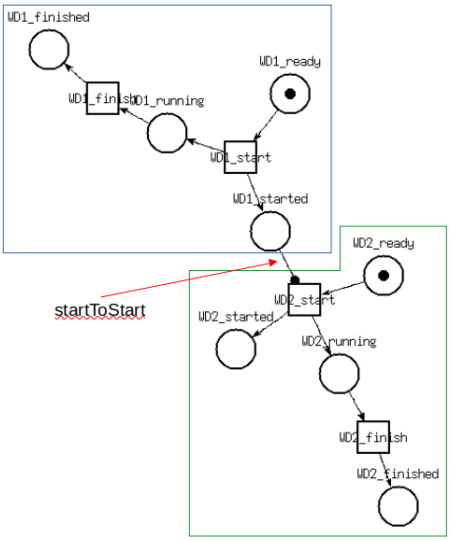


(c) Arborescence obtenue

Figure 23: Transformation ATL 1



(a) Exemple de modèle `.simplepdl`



(b) Transformation du modèle `.simplepdl` en un modèle `.petrinet`

Figure 24: Transformation ATL 2

7 Transformation modèle à texte

On implémente des transformation de modèle à texte (M2T) à l'aide du langage Acceleo.

7.1 toDot

La transformation `toDot` prend en entrée un modèle `SimplePDL` et génère un fichier `.dot` permettant de visualiser le processus sous forme de graphe orienté. On s'inspire du même code couleur développé dans Sirius.

7.1.1 Transformation des éléments du modèle

Élément	Description
`WorkDefinition`	Chaque `WorkDefinition` est représentée comme un noeud de forme losange avec un fond gris.
`WorkSequence`	Les `WorkSequence` sont transformées en arcs fléchés entre les `WorkDefinition`, avec un libellé indiquant le type de lien et une couleur associée.
`Ressource`	Chaque `Ressource` est représentée comme une ellipse bleue.
`UseRessource`	Les arcs entre les `WorkDefinition` et les `Ressource` représentent la consommation de ressources, avec un style pointillé bleu.
`Guidance`	Les objets `Guidance` sont affichés sous forme de notes jaunes contenant leur texte. Ils sont reliés par des arcs gris pointillés aux éléments concernés.

7.1.2 Fonctions utilitaires

Deux fonctions (`query`) sont utilisées pour améliorer la lisibilité du graphe:

- `toLabel`: convertit une valeur `WorkSequenceType` en une chaîne descriptive.
- `getColor`: attribue une couleur d'arc en fonction du type de `WorkSequence`.

7.1.3 Exemples

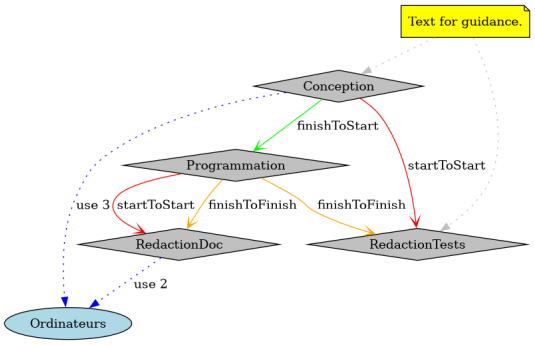
Pour obtenir les graphes, on utilise la commande suivante dans le terminal:

```
dot -Tpng InputFile.dot -o OutputFile.png
```

Cette commande utilise l'outil `dot` de Graphviz pour transformer le fichier d'entrée obtenu grâce à la transformation `InputFile.dot` en une image au format PNG nommée `OutputFile.png`.

```
digraph spdl_valid_model {

  // WorkDefinitions
    Conception [shape=diamond style=filled fillcolor=gray];
    RedactionDoc [shape=diamond style=filled fillcolor=gray];
    Programmation [shape=diamond style=filled fillcolor=gray];
    RedactionTests [shape=diamond style=filled fillcolor=gray];

  // WorkSequences
    Conception -> RedactionTests [label="startToStart" arrowhead=vee color=red];
    Conception -> Programmation [label="finishToStart" arrowhead=vee color=green];
    Programmation -> RedactionDoc [label="finishToFinish" arrowhead=vee color=orange];
    Programmation -> RedactionTests [label="finishToFinish" arrowhead=vee color=orange];
    Programmation -> RedactionDoc [label="startToStart" arrowhead=vee color=red];

  // Ressources
    "res_Ordinateurs" [label="Ordinateurs" shape=ellipse style=filled fillcolor=lightblue];

  // UseRessources
      Conception -> "res_Ordinateurs" [label="use 3" style=dotted color=blue];
      RedactionDoc -> "res_Ordinateurs" [label="use 2" style=dotted color=blue];

  // Guidances
    "guide_Text_for_guidance_" [label="Text for guidance." shape=note style=filled fillcolor=yellow];
        "guide_Text_for_guidance_" -> "Conception" [style=dotted color=gray];
        "guide_Text_for_guidance_" -> "RedactionTests" [style=dotted color=gray];
}
```

(a) Fichier `.dot` obtenu



(b) Graphe obtenu via la transformation `toDot`

Figure 25: Transformation ATL

7.2 toTina

7.2.1 Description de la transformation

On définit une transformation Acceleo qui créer un fichier `.net` à partir d'un modèle `.petrinet`. Ce fichier `.net` est compatible avec l'outil `Tina`.

Syntaxe obtenue après la transformation `PetriNet` vers `Tina`

- Déclaration du réseau :
  - `net` ⟨*nom_du_réseau*⟩
- Définition des places :
  - `pl` ⟨*nom_place*⟩ `(`⟨*nombre_de_jetons*⟩`)` `((n)` est optionnel si `n = 0`)
- Définition des transitions :
  - `tr` ⟨*nom_transition*⟩ `[`⟨*borne_inf*⟩`,`⟨*borne_sup*⟩`]` (optionnel)
  - Entrées de la transition :
    - `<nom_place>` (arc standard)
    - `<nom_place>?<poids>` (arc de lecture)
    - `<nom_place>*<poids>` (arc pondéré)
  - Sorties de la transition :
    - `-> <nom_place>` (arc standard)
    - `-> <nom_place>*<poids>` (arc pondéré)

Une des difficultés rencontrée lors de l'implementation du fichier `.mtl` fût d'obtenir un fichier de sortie bien conforme à la syntaxe `Tina`, par exemple, le respect des sauts de ligne, des crochets pour l'intervalle de temps etc...

7.2.2 Exemples de fichier `.net` obtenu

On reprends les modèles `.simplepdl` transformés en `.petrinet` pour appliquer la transformation `.toTina`.

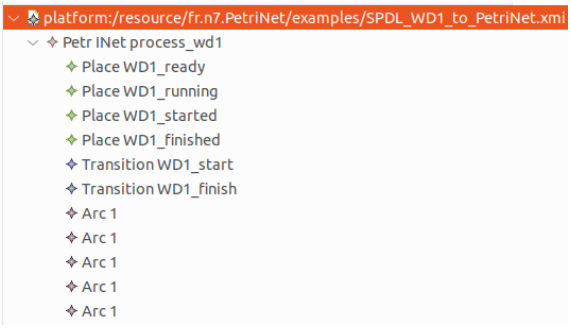


Figure 26: Arborescence du modèle `process_wd1`

```
net process_wd1

pl WD1_ready (1)
pl WD1_running
pl WD1_started
pl WD1_finished

tr WD1_start WD1_ready -> WD1_running WD1_started
tr WD1_finish WD1_running -> WD1_finished
```

Figure 27: Fichier `.net` obtenu

Figure 28: Arborescence du modèle `process_wd2`

```
net process_wd2

pl WD1_ready (1)
pl WD1_running
pl WD1_started
pl WD1_finished
pl WD2_ready (1)
pl WD2_running
pl WD2_started
pl WD2_finished

tr WD1_start WD1_ready -> WD1_running WD1_started
tr WD1_finish WD1_running -> WD1_finished
tr WD2_start WD2_ready WD1_started?1 -> WD2_running WD2_started
tr WD2_finish WD2_running -> WD2_finished
```

Figure 29: Fichier `.net` obtenu

7.2.3 Affichage avec `Tina`

Une fois qu'on a les fichiers `.net`, on peut ouvrir avec la commande `nd nom_du_fichier.net` l'outil `Tina` et afficher nos réseaux de Petri.

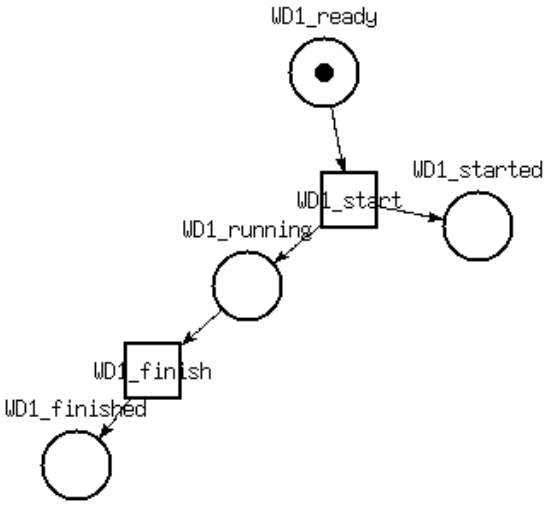


Figure 30: Réseau de Petri de `process_wd1`

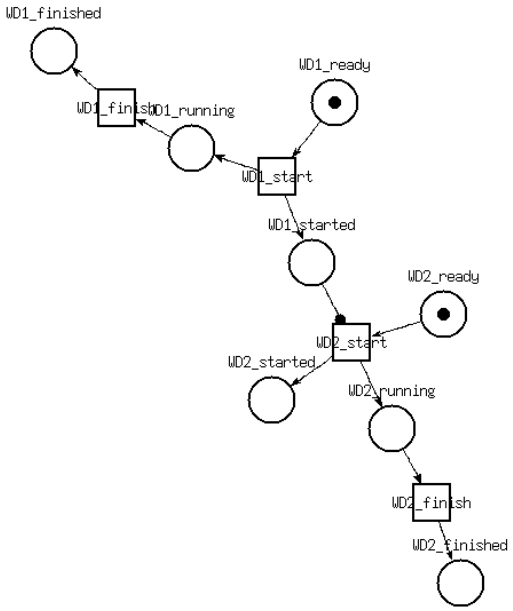


Figure 31: Réseau de Petri de `process_wd2`

8 Conclusion

Ce projet nous a permis de mettre en oeuvre une chaîne complète de vérification pour des modèles de procédés, en combinant plusieurs aspects de l'ingénierie dirigée par les modèles. On a conçu et exploité trois métamodèles (`PDL1`, `SimplePDL` et `PetriNet`) accompagnés de syntaxes concrètes (textuelles via Xtext ou graphiques via Sirius) et de contraintes de validation (via Java), afin de garantir la cohérence et la robustesse des modèles instanciés.

Les transformations mises en place, qu'elles soient modèle à modèle, via ATL, ou modèle à texte, via Acceleo, nous ont permis d'automatiser le passage entre différents niveaux de représentation.

Enfin, on a rencontré certaines difficultés, notamment lors de l'installation d'Eclipse et des problèmes de compatibilité des packages, ce qui a considérablement ralenti notre progression. De plus, n'étant pas très familier avec le langage Java, le debuggage des erreurs a été un peu dur.

9 Où trouver les fichiers

Lien github: `https://github.com/InesBesbes/idm-n7.git`

- `fr.n7.simplePDL/`
  - `SimplePDL.ecore` : métamodèle de SimplePDL
  - `examples/` : dossier contenant des exemples de modèles valides/invalides
  - `bin/simplepdl/validation/SimplePDLValidator.class` : Fichier de validation / définition des contraintes pour `SimplePDL`
- `fr.n7.PetriNet/`
  - `model/PetriNet.ecore` : métamodèle de PetriNet
  - `examples/` : dossier contenant des exemples de modèles valides/invalides
  - `bin/fr/n7/PetriNet/validation/PetriNetValidator.class` : Fichier de validation / définition des contraintes our `PetriNet`
- `fr.n7.pdl1/`
  - `src/fr/n7/PDL1.xtext` : fichier Xtext permettant de définir une syntaxe textuelle pour `PDL1`
  - `model/generated/PDL1.ecore` : métamodèle de `PDL1` auto généré depuis l'eclipse de déploiement
- `fr.n7.simplePDLtoPetriNet.ATL/simplePDL_to_PetriNet.atl` : fichier ATL définissant la transformation modèle à modèle `SimplePDL` à `PetriNet`
- `fr.n7.simplepdl.todot/`
  - `src/fr/n7/simplepdl/todot/main/toDot.mtl` : fichier définissant la transformation modèle à texte d'un modèle `SimplePDL` à un fichier `.dot`
  - `process_wd1.dot`, `process_wd2.dot` et `spdl_valid_model.dot` : exemples de fichier `.dot` obtenus après transformation
  - `process_wd1.png`, `process_wd2.png` et `spdl_valid_model.png` : images obtenues des précédents fichiers `.dot`
- `fr.n7.petrinet.totina/`
  - `src/fr/n7/petrinet/totina/main/toTina.mtl` : fichier définissant la transformation modèle à texte d'un modèle `PetriNet` à un fichier `.net`
  - `process_wd1.net`, `process_wd2.net` et `spdl_valid_model.net` : exemples de fichier `.net` obtenus après transformation
- `fr.n7.simplepdl.exemples/` : fichier contenant l'eclipse de deploiement
  - `fr.n7.pdl1tosimplePDL.ATL/pdl1_to_simplePDL.atl` : fichier ATL définissant la transformation modèle à modèle `PDL1` à `SimplePDL`
  - `fr.n7.simplepdl.design/simplepdl.odesign` : fichier contenant le syntaxe graphique Sirius
  - `example1.pdl1` : modèle `.pdl1` obtenu depuis une définition par syntaxe textuelle

– `pdl1_to_simplePDL.simplepdl` : modèle `.simplepdl` obtenu après la transformation ATL du `.pdl1` précédent