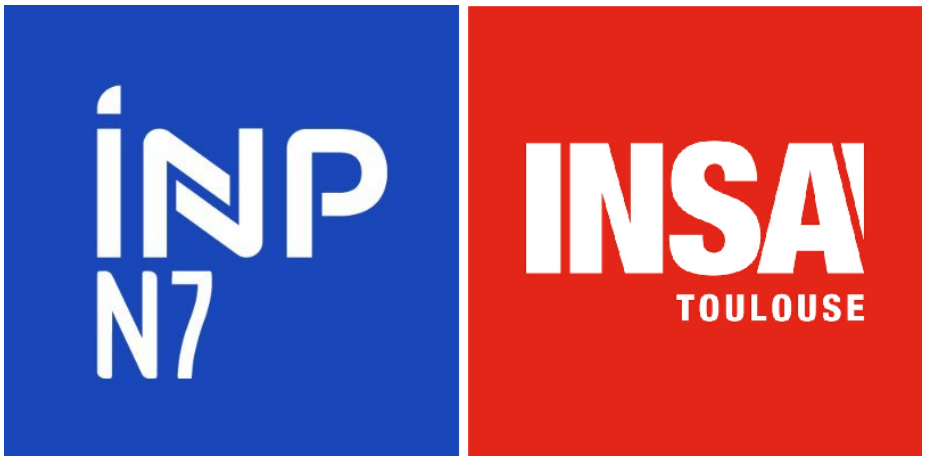

Projet

Une chaîne de vérification pour modèles de procédés

realised by

Yanis Mac
Hicham Laghmam

INP ENSEEIHT - 5ModIA
Système de confiance

2025

Contents

1 Introduction

La modélisation de procédés constitue un enjeu central dans de nombreux domaines industriels et informatiques, en tant qu'outil permettant de décrire, analyser et optimiser des chaînes d'activités complexes impliquant activités, ressources et dépendances. Toutefois, la conception de tels modèles reste une activité délicate, en particulier lorsqu'elle doit être accessible à des utilisateurs n'ayant pas nécessairement de formation en informatique.

Notre objectif dans ce projet est de concevoir un écosystème complet pour la création, la manipulation et la vérification de modèles de procédés. Cet écosystème permet à l'utilisateur de spécifier des procédés à l'aide de syntaxes variées (graphiques, textuelles, arborescentes), de vérifier automatiquement la conformité de ces modèles à un méta-modèle donné, et d'en extraire des représentations équivalentes, notamment sous forme de réseaux de Pétri.

Afin d'atteindre ces objectifs, nous avons utilisé le logiciel Eclipse, dessus, nous avons développé plusieurs composants : méta-modèles (SimplePDL et réseau de Pétri), éditeurs (graphiques avec Sirius, textuels avec Xtext, arborescents), transformations modèles-à-modèles (Java, ATL) et modèles-à-textes (Acceleo), ainsi qu'un ensemble d'exemples démonstratifs. Ces outils offrent une assistance à la fois pour la modélisation, la simulation, et la validation des modèles, notamment via l'analyse de propriétés formelles comme la terminaison ou l'atteignabilité.

2 Modèle de procédé

On se propose une étude de cas, nous allons chercher à vérifier la terminaison de processus. Pour cela nous définissons un métamodèle, SimplePDL.
Nous définissons alors un modèle de processus, en commençant par définir ce qu'est un Processus (Process), il est composé de plusieurs éléments:

- des activités (WorkDefinition)
- des dépendances (WorkSequence) entre activités
- des ressources (Ressource)
- et des notes (Guidance)

On retrouve figure 1 un exemple de modèle de procédé sans ressources.

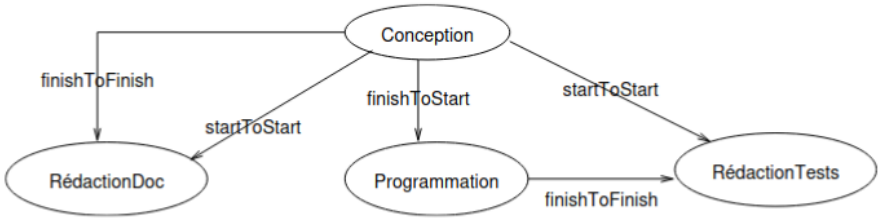

Figure 1: Exemple de processus

La première étape de la création du métamodèle de SimplePDL a été de créer le métamodèle ecore dans Eclipse. On peut retrouver sa représentation graphique figure 2.

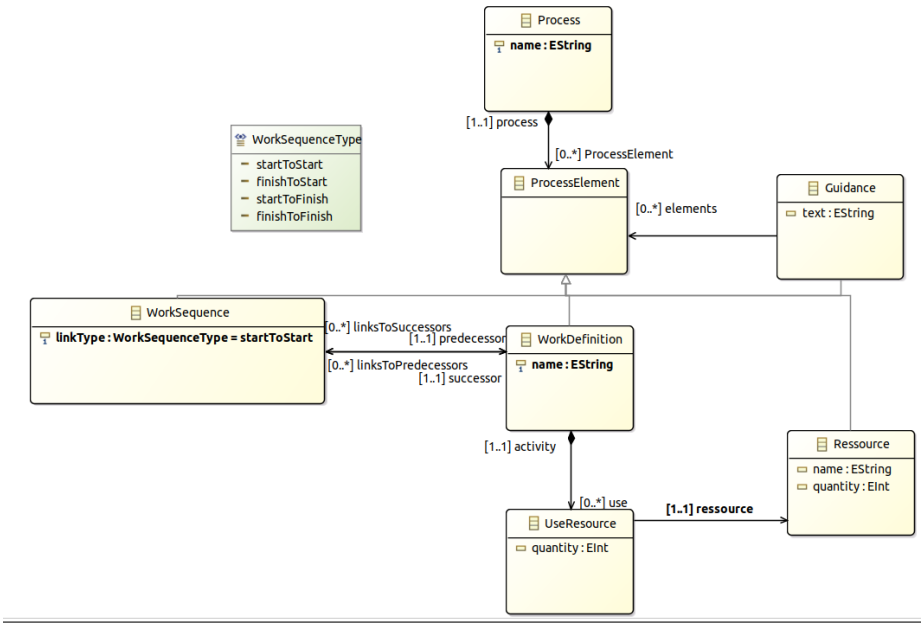

Figure 2: Diagramme du métamodèle de SimplePDL

2.1 Contraintes

Les éléments d'un Processus ont cependant des contraintes, le métamodèle doit donc respecter les contraintes suivantes :

R1.1 Un modèle de procédé se compose d'activités et de ressources

R1.1.1 Une activité est caractérisée par un nom

R1.1.2 Une ressource est caractérisée par un nom et une quantité disponible

R1.2 Les activités peuvent être reliées entre elles avec des dépendances

R1.2.1 Les dépendances entre activités ont un type, qui modélise quel aspect de la activité est visé par la dépendance :

- `start-to-start`: le début de la activité dépendante est conditionné par le début de la activité dont elle dépend
- `finish-to-start`: le début de la activité dépendante est conditionné par la fin de la activité dont elle dépend
- `start-to-finish`: la fin de la activité dépendante est conditionnée par le début de la activité dont elle dépend
- `finish-to-finish`: la fin de la activité dépendante est conditionnée par la fin de la activité dont elle dépend

R1.3 Une activité peut être reliée à une ou plusieurs ressources à l'aide d'une dépendance adaptée

R1.3.1 Une dépendance à une ressource ne concerne qu'une seule activité et une seule ressource

R1.3.2 On doit pouvoir spécifier la quantité de ressource nécessaire à une activité au niveau de la dépendance

R1.4 Un modèle de procédé peut présenter des commentaires

R1.4.1 Un commentaire contient du texte

R1.4.2 Un commentaire peut être relatif/attaché à un élément particulier du modèle, ou être général (attaché à rien)

Nous avons alors créé un package simplepdl.validation dans lesquels nous ajoutons ces contraintes implémenté en java pour le métamodèle SimplePDL.

3 Réseaux de Petri

On aimerait savoir si un processus donné se termine ou non, pour cela on va exprimer la sémantique de SimplePDL en s'appuyant sur un langage formel, ici les réseaux de Petri.
Un réseau de Petri est composé de plusieurs éléments:

- des places (Place)
- des transitions (Transition)
- d'arcs (Arc)

On retrouve figure 3 un exemple de réseau de Petri simple.

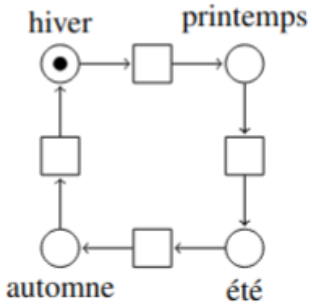

Figure 3: Exemple de réseau de Petri simple

Par la suite, comme pour SimplePDL on a implémenté le métamodèle ecore PetriNet que l'on retrouve figure 4.

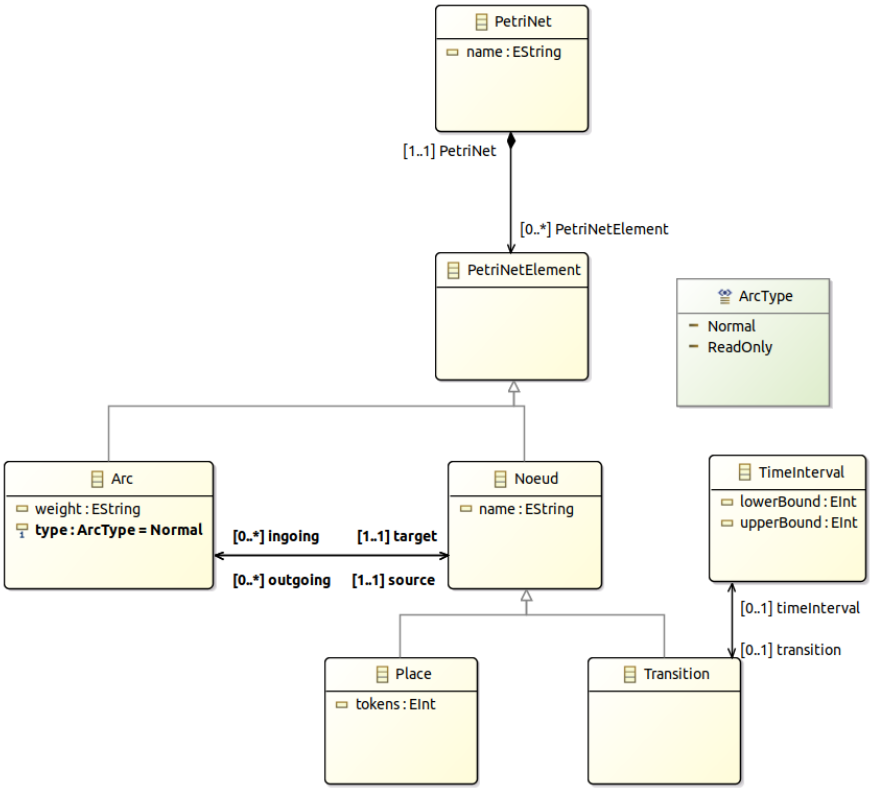

Figure 4: Diagramme du métamodèle de PetriNet

3.1 Contraintes

On a de même que pour SimplePDL eu des contraintes à respecter:

R2.1 Un réseau de Pétri est composé de places et de transitions
R2.2 Une place est nommée et présente des jetons (tokens)
R2.3 Une transition est nommée et peut présenter un intervalle de temps

- **R2.3.1** L'intervalle de temps est optionnel
- **R2.3.2** Un intervalle a une borne inférieure entière
- **R2.3.3** Un intervalle peut avoir une borne supérieure entière, ou pas

R2.4 Les places et les transitions sont reliées entre eux par des arcs pondérés

- **R2.4.1** Un arc est caractérisé par son poids (entier)
- **R2.4.2** Un arc relie une place à une transition ou une transition à une place
- **R2.4.3** Un arc peut présenter un type différent, en particulier l'arc en lecture seule

Nous avons alors créé un package petrinet.validation dans lesquels nous ajoutons ces contraintes implémenté en java pour le métamodèle PetriNet.

R2.5 Un réseau de Pétri a un nom (contrainte de l'outil)

4 Transformation modèle à texte (Acceleo)

Afin de rendre les modèles de réseaux de Pétri utilisables par des outils d'analyse externes, et notamment par la boîte à outils *Tina*, il est nécessaire de convertir les modèles issus de notre transformation modèle-à-modèle en une représentation textuelle conforme à la syntaxe attendue. Pour cela, nous avons utilisé **Acceleo**.

4.1 Utilisation dans notre projet

Dans le cadre de ce projet, nous avons mis en œuvre plusieurs générateurs Acceleo pour assurer différentes tâches :

- Génération de fichiers `.net` à partir des modèles de réseaux de Pétri, afin de permettre leur analyse via Tina.
- Génération de fichiers Dot à partir de modèles SimplePDL, pour visualiser graphiquement les procédés modélisés.

Chaque transformation a été implémentée dans un projet Acceleo dédié, manipulant les éléments du méta-modèle source, appliquant une logique de transformation, et produisant un texte structuré respectant la grammaire attendue. On peut en retrouver une implémentation figure **??**

5 Syntaxes concrètes graphiques (Sirius)

Afin de rendre la modélisation des procédés plus intuitive et accessible pour des utilisateurs non spécialistes de l'informatique, nous avons développé une syntaxe concrète graphique à l'aide de l'outil **Sirius**.

5.1 Présentation de Sirius

Sirius est un outil permettant de définir des éditeurs graphiques personnalisés à partir de méta-modèles EMF. Il permet de spécifier, via un fichier `.odesign`, les différentes représentations visuelles (nœuds, arcs, conteneurs), ainsi que les interactions possibles (création, suppression, modification d'éléments) à travers une palette d'outils graphique.

5.2 Utilisation dans notre projet

Dans notre projet, Sirius a été utilisé pour construire un éditeur graphique des modèles de procédés conformes au méta-modèle `SimplePDL`. Cet éditeur permet

à l'utilisateur de visualiser, modifier et créer des modèles de manière interactive. Les éléments suivants sont représentés graphiquement :

- **Tâches (WDNode)** : chaque tâche du procédé est représentée par un nœud graphique distinct, affiché avec son nom. Ces nœuds constituent les éléments centraux du modèle.
- **Dépendances entre tâches (WSEdge)** : modélisées par des arcs connectant les tâches entre elles. Le style visuel (couleur, flèche, etc.) est adapté en fonction du type de dépendance (start-to-start, finish-to-start, etc.).
- **Ressources (RessourceNode)** : représentées dans un calque distinct, elles indiquent la quantité disponible et peuvent être connectées à des tâches via des arcs ('RessourceEdge') précisant la quantité requise.
- **Commentaires (GuidanceNode)** : affichés dans un calque séparé (`CalqueGuidance`), ils permettent d'associer des annotations aux éléments du modèle sans surcharger la vue principale. Les liens entre commentaires et éléments (via 'GuidanceEdge') sont également affichés de manière différenciée.

5.3 Fonctionnalités mises en place

Nous avons structuré notre éditeur Sirius autour de plusieurs calques fonctionnels, permettant de séparer visuellement les différentes dimensions du modèle :

- **Calque principal (`Default`)** : regroupe les éléments de base du procédé (tâches et dépendances).
- **Calque Ressources (`CalqueRessource`)** : dédié à l'affichage et la manipulation des ressources disponibles et de leurs relations avec les tâches.
- **Calque Guidance (`CalqueGuidance`)** : réservé aux commentaires et à leurs liaisons éventuelles avec les autres éléments du modèle.

La palette Sirius, configurée dans le fichier `.odesign`, permet à l'utilisateur de créer tous les éléments du méta-modèle SimplePDL de manière interactive : tâches, ressources, types de dépendances, et commentaires. Chaque outil de la palette est associé à une opération de création spécifique, garantissant le respect des structures définies dans le méta-modèle.

5.4 Retour sur l'implémentation

La définition de l'éditeur Sirius a nécessité une bonne compréhension de la structure du méta-modèle. Certaines contraintes d'affichage (positionnement des éléments, clarté des dépendances) ont été résolues grâce à des choix de représentation judicieux (formes, couleurs, styles d'arcs). L'éditeur graphique s'est révélé particulièrement utile pour visualiser rapidement des erreurs dans les modèles (par exemple, des dépendances mal définies ou des tâches non reliées).

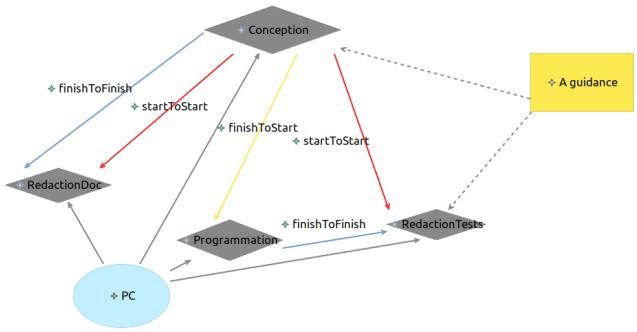

Figure 5: Exemple d'un modèle SimplePDL affiché dans l'éditeur Sirius

6 Transformation modèle à modèle (ATL)

ATL est un outil de l'écosystème Eclipse qui permet de transformer un modèle conforme à un certain métamodèle vers un autre modèle, potentiellement conforme à un autre métamodèle. ATL consiste ainsi à décrire, à l'aide de règles, comment un modèle source peut être transformé en un modèle cible, en respectant les métamodèles respectifs.

Dans notre cas, nous allons transformer un modèle conforme au métamodèle SimplePDL en un modèle conforme au métamodèle PetriNET.

Pour ce faire, nous établissons une liste de règles (trouvable dans le fichier : SpdlToPnet.atl) permettant de transformer chaque élément présent dans notre modèle de départ. Par exemple, chaque WorkSequence est transformée en un ensemble composé de trois places, deux transitions et quatre arcs. Afin d'expliciter les liens entre les éléments créés, nous présentons ci-dessous un schéma explicatif.

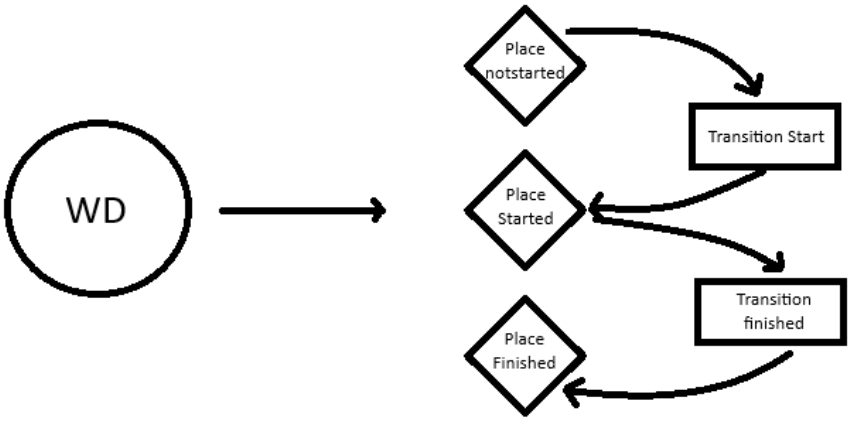

Figure 6: Règle prédéfinie pour la transformation d'une WorkDefinition en un réseau de Petri

Nous implémentons également la transformation des ressources par la création d'une place dédiée. Voici comment il est mis en œuvre :

```
19 -- Traduction Ressource -> Place dans le PetriNet
20 rule Ressource2Place {
21     from r: SPDL!Ressource
22     to p: PNET!Place (
23         name <- 'Ressource_' + r.name,
24         tokens <- r.quantity,
25         PetriNet <- r.getProcess()
26     )
27 }
```

Figure 7: Exemple de définition d'une règle : Une ressource devient une place

Ainsi, chaque UseResource est représentée par deux arcs :

- le premier envoie un jeton (symbolisant la ressource) dans la place correspondante au début de la transition vers la place start.
- le second récupère ce jeton lors de la transition vers la place finished.

7 Syntaxes concrètes textuelles (XText)

Xtext est un outil de l'écosystème Eclipse qui permet de définir facilement une syntaxe spécifiques pour un DSL. Grâce à une description syntaxique, Xtext génère automatiquement tout un environnement de développement dans Eclipse:

éditeur avec coloration, complétion, outline, détection et visualisation des erreurs, etc.

Nous allons donc concevoir 3 codes .xtext qui serviront de descripteurs syntaxiques pour notre modèle SimplePDL. Les syntaxes décrites par nos fichiers xtext seront les suivantes :

```
process ex1 {
    wd a {
        use test quantity 2
    }
    wd b
    wd c
    ws s2s from a to b
    ws f2f from b to c
    res test qtt 10
}
```

Figure 8: Exemple de syntaxe acceptée par les règles PDL1.xtext

```
process ex2 {
    res test qtt 10
    wd a {
        use test quantity 2
    }
    wd b {
        starts if a started
    }
    wd c {
        finishes if b finished
    }
}
```

Figure 9: Exemple de syntaxe acceptée par les règles PDL2.xtext

```
process : ex3
workdefinitions : a uses test quantity 2; b; c;
worksequences : a s2s b; b f2f c;
resources : test qtt 10;
```

Figure 10: Exemple de syntaxe acceptée par les règles PDL3.xtext

Dans cette section, nous nous sommes concentrés sur l'ajout de ressources aux différents fichiers Xtext.

8 Conclusion

Ce projet d'ingénierie dirigée par les modèles nous a permis de développer une chaîne complète de vérification pour les modèles de procédés, depuis leur spécification jusqu'à l'analyse formelle. Les principales difficultés rencontrées ont concerné :

- La prise en main des différents outils (Eclipse Modeling, Sirius, Xtext, Acceleo), le débug étant compliqué en étant peu familier avec ces outils, cela a causé des problèmes à de multiples reprises.

Nous avons réalisé l'essentiel des objectifs initiaux :

- Implémentation complète des méta-modèles SimplePDL et réseaux de Pétri
- Développement des trois éditeurs (arborescent, graphique Sirius, textuel Xtext)
- Transformation modèle-à-modèle vers les réseaux de Pétri en ATL
- Génération des propriétés LTL pour la vérification formelle
- La validation de scénarios d'exécution (pas encore fini)

Certains aspects n'ont pu être finalisés :

- Transformation modèle-à-modèle vers les réseaux de Pétri en Java

Les enseignements majeurs de ce projet incluent :

- L'importance d'une conception rigoureuse des méta-modèles dès la phase initiale
- La nécessité de tests intermédiaires dans les chaînes de transformation
- Les défis d'ergonomie dans les outils destinés à des non-informaticiens

Ce travail confirme le potentiel de l'ingénieurie des modèles pour la vérification formelle de systèmes, tout en soulignant l'importance des choix d'implémentation dans les chaînes d'outils intégrées. Les perspectives incluraient l'extension des types de ressources et l'amélioration des performances d'analyse.