Mini-projet IDM

OURKIA ABDELHAKIM
AZDAD BILAL
SARHANE ABDELMOUHAIMEN

Département Sciences du Numérique - Deuxième année
2022-2023

Contents

1 Introduction

Le mini-projet s'inscrit dans le cadre de la modélisation et de la transformation de modèles. L'objectif principal est de finaliser une chaîne de transformation et de validation. Le contexte implique l'utilisation d'Eclipse et d'outils tels que ATL, Acceleo, etc. pour définir les métamodèles, les contraintes OCL, et effectuer des transformations modèle à modèle et modèle à texte. La chaîne de transformation se concentre sur les métamodèles SimplePDL et PetriNet, avec l'ajout de la notion de ressource à SimplePDL.

2 Les Métamodèles

2.1 SimplePDL

Le métamodèle SimplePDL a été étendu pour intégrer la notion de ressource. La classe `Resource` a été introduite, représentant le nombre total de ressources disponibles. La classe intermédiaire `Amount` facilite la gestion des ressources nécessaires pour une `WorkDefinition`. Les ressources sont associées à une `WorkDefinition` via des montants de besoins.

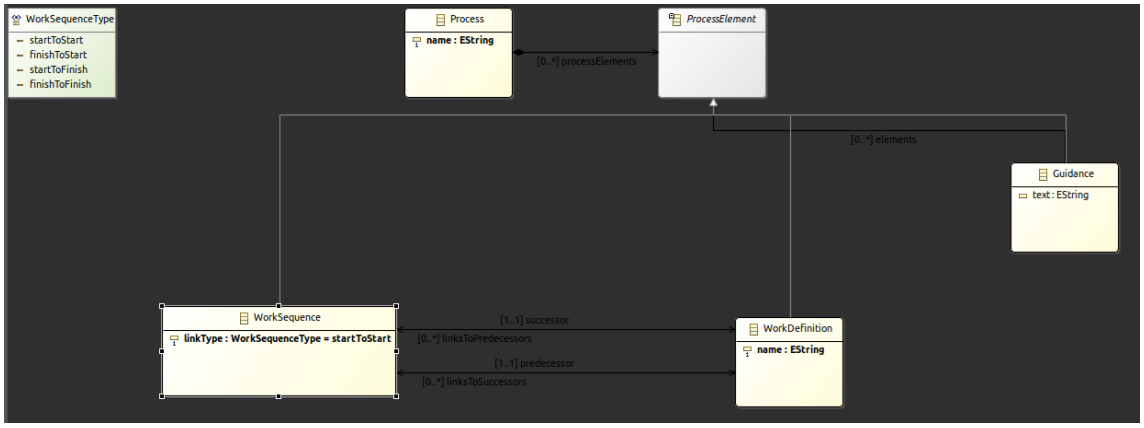

Figure 1: Le méta-modèle de SimplePDL sans ressources ajoutées

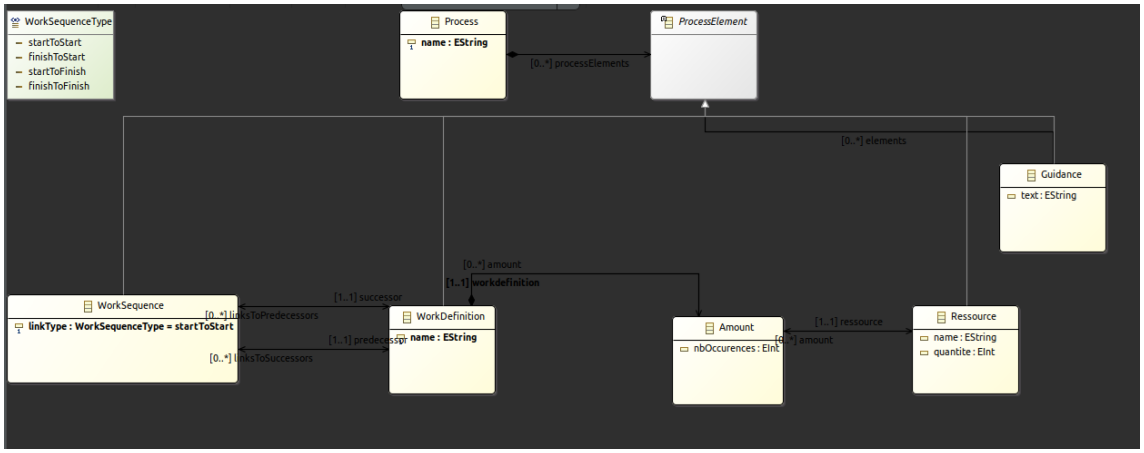

Figure 2: Le méta-modèle de SimplePDL avec ressources ajoutées

2.2 PetriNet

Le métamodèle PetriNet représente formellement les systèmes à éléments discrets. Il comprend des places (`Place`), des transitions (`Transition`), et des arcs (`Arc`) qui les relient. Les arcs

peuvent être de types `normal` ou `readArc`, indiquant des flux entre places et transitions. Chaque place a un attribut `jetons` représentant le nombre initial de jetons.

Ce modèle permet de représenter de manière formelle les interactions entre les différentes composantes d'un système à éléments discrets, en capturant les transitions d'état et les flux de jetons entre les places et les transitions.

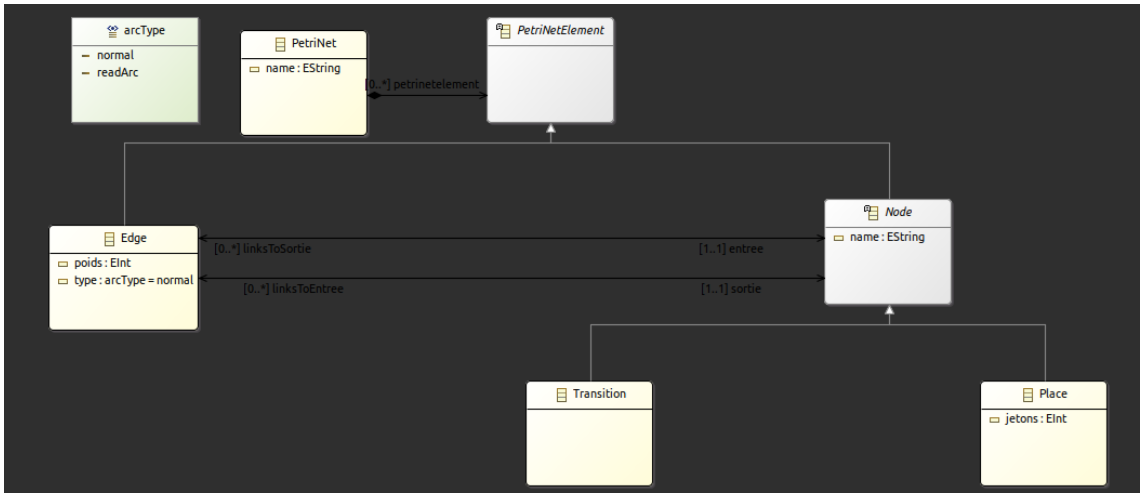

Figure 3: Le métamodèle PetriNet

3 Les Contraintes OCL

3.1 Définition des contraintes

Le langage de méta-modélisation ne permet pas d'exprimer toutes les contraintes que doivent respecter les modèles. Aussi, on complète la description des méta-modèle par des contraintes exprimées en OCL et parmi les contraintes que doit respecter un modèle de processus, nous avons : la validité du nom d'une activité, l'unicité des noms de chaque activité, la non-réflexivité des dépendances et la quantité non nulle des ressources disponibles dans un processus. Aussi, le modèle du réseau de petri doit respecter les contraintes suivantes : la validité du nombre initial des jetons dans les places, Deux places ne peuvent pas être reliées entre elles, ni deux transitions et la validité du nom.

3.2 Violation des contraintes

Des modèles intentionnellement générés ont été utilisés pour violer les contraintes OCL. Par exemple, des modèles avec un nombre nul de ressources disponibles ou des prédécesseurs identiques aux successeurs.

4 Transformation Modèle à Modèle (SimplePDL vers PetriNet)

4.1 ATL et EMF/JAVA

La transformation modèle à modèle de SimplePDL à PetriNet est réalisée à l'aide de l'outil ATL ou bien EMF permettant la génération du code java pour SimplePDL et PetriNet en modélisant chaque `WorkDefinition` à l'aide de 4 place telle que chacune représente un état d'avancement de la `WorkDefinition` .Ces états sont les états prêts (ready), commencée (running) et terminée (finished) ,et nous avons ajouté une place started qui mémorise que l'activité a été commencée . Ces places sont reliées par deux transitions qui représentent les actions que l'on peut faire sur

une activité qui sont commencer (start) et terminer (finish). Une WorkSequence est représenté par un arc de type read_arc qui en fonction de la nature de la liaison entre deux WorkDéfinition peut relier les places Place_Started ou Place_Finished avec les transitions Transition_Start ou Transition_Finish . Pour modéliser les ressources, nous avons choisi d'ajouter une place pour chaque ressource, avec le nombre de jetons représentant son nombre d'occurrences.

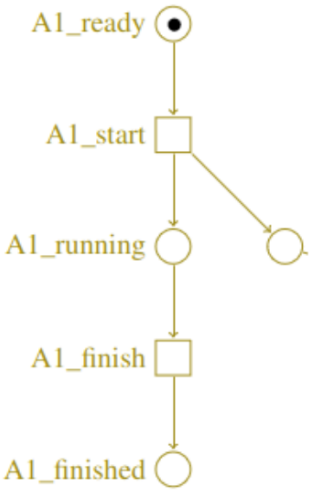

Figure 4: Modélisation d'une WorkDefinition

5 Transformation Modèle à Texte avec Acceleo

5.1 ToTina

La transformation vers la syntaxe de Tina (`.net`) depuis le modèle PetriNet implique la représentation de chaque place avec le mot clé `pl` suivi du nombre de jetons entre parenthèses. Les transitions sont transformées qui prend en compte les poids des transitions et vérifie si les arcs sont de type `readArc`.

```
net developpement
pl Conception_ready (1)
pl Conception_running (0)
pl Conception_finished (0)
pl Conception_started (0)
pl RedactionDoc_ready (1)
pl RedactionDoc_running (0)
pl RedactionDoc_finished (0)
pl RedactionDoc_started (0)
pl Programmation_ready (1)
pl Programmation_running (0)
pl Programmation_finished (0)
pl Programmation_started (0)
pl RedactionTests_ready (1)
pl RedactionTests_running (0)
pl RedactionTests_finished (0)
pl RedactionTests_started (0)
pl concepteur (3)
pl developpeur (2)
pl machine (4)
pl redacteur (1)
pl testeur (2)
tr Conception_start Conception_ready    concepteur*2    machine*2 -> Conception_started   Conception_running
tr Conception_finish Conception_running -> Conception_finished   concepteur*2   machine*2
tr RedactionDoc_start RedactionDoc_ready    Conception_started?1    machine    redacteur -> RedactionDoc_started   RedactionDoc_running
tr RedactionDoc_finish RedactionDoc_running    Conception_finished?1 -> RedactionDoc_finished   machine   redacteur
tr Programmation_start Programmation_ready    Conception_finished?1    developpeur*2    machine*3 -> Programmation_started   Programmation_running
tr Programmation_finish Programmation_running -> Programmation_finished   developpeur*2   machine*3
tr RedactionTests_start RedactionTests_ready    Conception_started?1    machine*2    testeur -> RedactionTests_started   RedactionTests_running
tr RedactionTests_finish RedactionTests_running    Programmation_finished?1 -> RedactionTests_finished   machine*2   testeur

```

Figure 5: Modèle Développement avec syntaxe de Tina (.net)

5.2 ToLTL

Les propriétés LTL sont générées à partir de SimplePDL pour la vérification avec l'outil Selt de Tina. Ces propriétés vérifient que les processus se terminent, qu'une activité ne se trouve pas dans plusieurs états simultanément, et qu'un processus démarré reste démarré . L'outil selt permettra alors de vérifier si ces propriétés sont effectivement satisfaites sur le modèle de réseau de Petri.

Nous avons fait le choix d'utiliser la propriété - <> finished au lieu d'utiliser <> finished pour avoir la réponse False et pour que l'outil selt nous donne un contre-exemple et le chemin à suivre pour avoir une terminaison

```
1
2 op finished = Conception_finished/\RedactionDoc_finished/\Programmation_finished/\RedactionTests_finished;
3
4
5 [] (finished => dead);
6 [] <> dead;
7 [] (dead => finished);
8 - <> finished;
```

Figure 6: Proprietes LTL

6 Syntaxe Graphique avec Sirius

Les syntaxes graphiques sont cruciales pour la visualisation des modèles de manière plus lisible et agréable. C'est pourquoi nous avons utilisé l'outil Sirius d'Eclipse, basé sur les technologies Eclipse Modeling (EMF), permettant de générer un éditeur graphique à partir d'un modèle Ecore.

Nous avons réussi à définir une syntaxe graphique pour le métamodèle défini par l'Ecore, dans laquelle il est possible de définir graphiquement, pour un Process, les `WorkDefinitions`, les `WorkSequences` ainsi que les `Ressources`, comme le montre la figure ci-dessous.

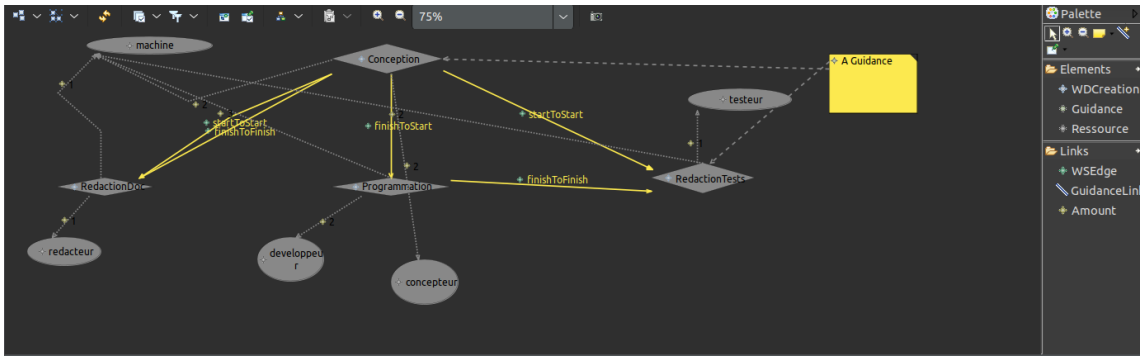

Figure 7: Le modèle graphique généré par Sirius

Le modèle Sirius est utilisé pour créer une représentation graphique des éléments du processus. Les `Nodes` représentent des `WorkDefinitions`, `Guidance`, et `Resources`. Des relations sont établies entre les éléments, comme les `WorkSequences` et la classe `Amount`.

Nous avons créé un projet de nature Sirius qui contient un modèle conforme à SimplePDL, qui servira à tester la syntaxe graphique. Nous avons mis en place le modèle de description de la syntaxe graphique, qui consiste à initialiser le modèle de description de la syntaxe graphique souhaitée et l'utiliser avec le modèle de test, ainsi qu'à définir la définition de la partie graphique de l'éditeur, afin d'afficher sur l'éditeur les différents éléments de nos modèles.

Finalement, nous avons défini la palette pour manipuler le modèle à travers des objets graphiques de cette vue. Ces outils sont regroupés en sections.

7 Syntaxe Textuelle avec Xtext

La syntaxe textuelle définie pour le méta-modèle SimplePDL avec Xtext offre une manière pratique et accessible de construire et de modifier des modèles conformes au métamodèle. Xtext, faisant partie du Textual Modeling Framework (TMF), fournit un éditeur syntaxique doté de nombreuses fonctionnalités telles que la coloration, la complétion, la détection des erreurs, etc.

Le fichier PDL.xtext contient la description Xtext de la syntaxe associée à SimplePDL.

Nous avons défini une syntaxe textuelle pour le méta-modèle SimplePDL. Par exemple, `Process -> process {}` représente un processus contenant des `WorkDefinitions`, et `WorkDefinition -> wd ID {}` représente une `WorkDefinition` avec un identifiant. Voici un exemple de la syntaxe textuelle générée pour un modèle SimplePDL donné :

```
process developpement {
    rs concepteur 3
    rs developpeur 2
    rs machine 4
    rs redacteur 1
    rs testeur 2

    wd Conception amountRessources (
        value 2 ofRessource concepteur
        value 2 ofRessource machine
    )

    wd RedactionDoc amountRessources (
        value 1 ofRessource machine
        value 1 ofRessource redacteur
    )

    wd Programmation amountRessources (
        value 2 ofRessource developpeur
        value 3 ofRessource machine
    )

    wd RedactionTests amountRessources (
        value 2 ofRessource machine
        value 1 ofRessource testeur
    )

    ws f2f from Conception to RedactionDoc
    ws f2s from Conception to Programmation
    ws f2f from Programmation to RedactionTests
    ws f2f from RedactionDoc to RedactionTests
}
```

Figure 8: Le modèle textuel avec l'éditeur Xtext suivant la syntaxe textuelle

8 Conclusion

Le mini-projet a permis de relever divers défis liés à la modélisation et à la transformation de modèles. La chaîne de transformation a été développée avec succès, couvrant la définition des métamodèles, l'application des contraintes OCL, et les transformations modèle à modèle et modèle à texte. Les résultats obtenus ont démontré une compréhension approfondie des concepts enseignés.

9 Glossaire des Technologies Utilisées

- **Tina:** Syntaxe concrète de représentation de modèle, utilisée pour représenter les réseaux de Petri.
- **Ecore:** Langage de méta-modélisation. (`SimplePDL.ecore` et `PetriNet.ecore`)
- **EMF/JAVA:** Framework basé sur Ecore pour la création, le maintien et la manipulation de modèles. (`SimplePDL2PetriNet.javal`)
- **XText:** Outil pour définir une syntaxe concrète textuelle pour un méta-modèle. (`PDL1.xtext` et `developpement.pdl1`)
- **ATL:** Langage de transformation modèle à modèle.(`SimplePDL2PetriNet.atl`)
- **Acceleo:** Outil de génération de texte à partir de modèles.(`toLTL.mtl` et `toLTL.mtl`)
- **Sirius:** Outil pour créer des éditeurs graphiques personnalisés. (`simplepdl.odesign`)