Projet système de confiance

Dion Thomas / Demarquet Alexandre
Groupe D

MODIA 5ème année
2025

Table des matières

Table des figures

1 Introduction

L'objectif de ce projet est de créer des outils ergonomiques pour définir des modèles de procédés et les éditer avec différentes interfaces (arborescente, graphique ou textuelle) et de les transformer automatiquement en réseaux de Pétri. Le projet s'appuie sur les principes de l'ingénierie dirigée par les modèles (IDM), utilisant les technologies Ecore, Sirius, Xtext, ATL, Acceleo et Java.

Pour ce rapport, on présente d'abord les méta-modèles *SimplePDL* et *PetriNet* ainsi que les contraintes statiques pour assurer la validité des modèles. Ensuite, nous décrivons l'éditeur graphique conçu à l'aide de Sirius pour la création et la visualisation des modèles. Puis, nous présenterons l'éditeur textuel développé avec Xtext permettant de créer des modèles avec différentes syntaxes et de visualiser en arborescence les modèles. Par la suite, nous présenterons les transformations modèle à modèle développées avec ATL et les transformations modèle à texte avec Acceleo. Enfin, nous conclurons et donnons en annexe les fichiers contenus dans le livrable.

2 Méta-Modèles et contrainte

2.1 SimplePDL

Pour le meta modèle simplePDL nous avons reutilisé le metamodèle fournit en TP en l'enrichissant de deux classe `Ressource` et `RessourceAllocation`.

1. `Ressource`

Cette classe permet de représenter un ressource disponible dans un processus. Elle hérite de la classe `ProcessElement` qui la rattache à un `Process`. Deplus chaque ressource possède :

- `name` : un nom de type `EString` pour identifier la ressource ;
- `quantity` : un entier (`EInt`) représentant la quantité totale disponible de cette ressource dans le processus.

2. `RessourceAllocation`

Cette classe permet de faire l'allocation d'une ressource à une tache et est définie par :

- `task` : référence vers la tâche concernée (type `WorkDefinition`) ;
- `ressource` : référence vers la ressource utilisée (type `Ressource`) ;
- `quantity` : entier pour le nombre unités de la ressource qui sont nécessaires à l'exécution de cette tâche.

Cette classe est utilisée en tant que référence contenue dans `WorkDefinition` via la propriété `resourceUsages`, ce qui permet de modéliser plusieurs allocations par tâche.

Afin de modéliser plusieurs allocations par tâche, la classe `RessourceAllocation` est utilisée en tant que référence contenue dans `WorkDefinition` via la propriété `resourceUsages`.

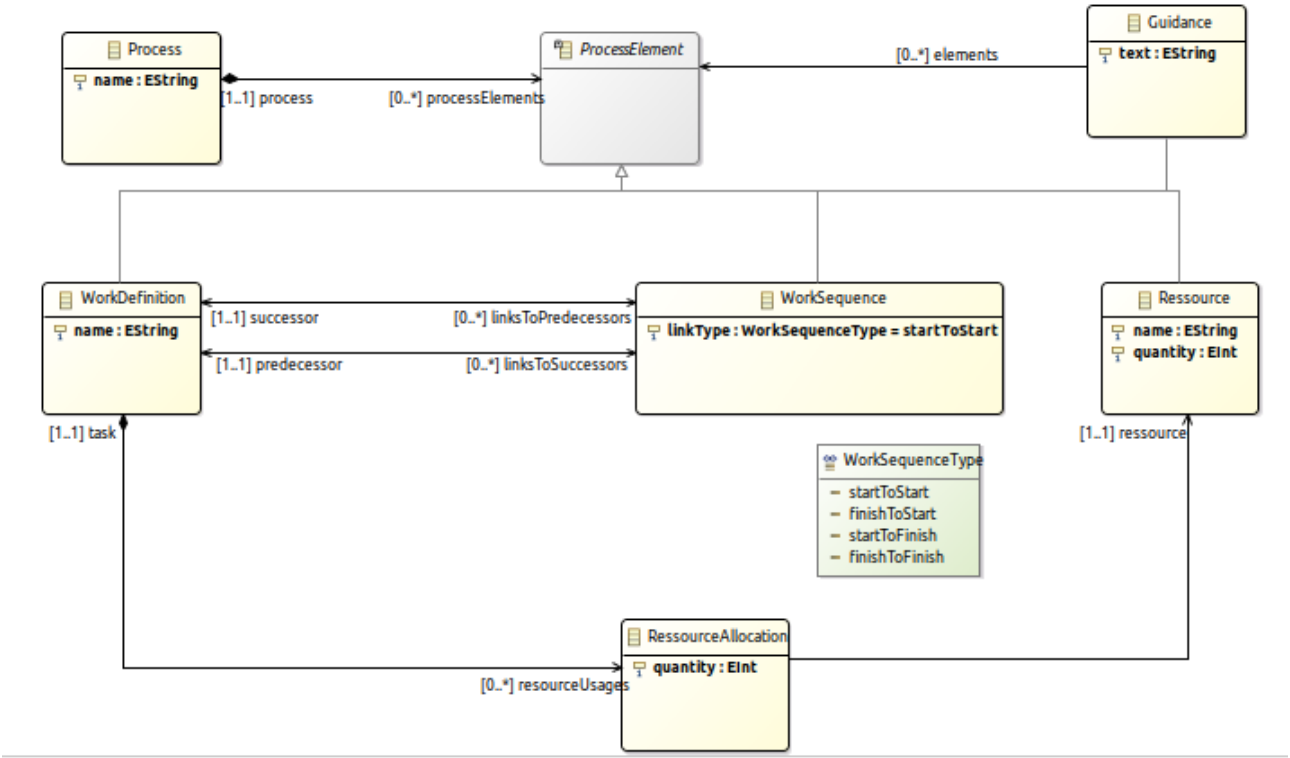

FIGURE 1 – Méta-Modèle SimplePDL (.aird)

2.2 PetriNet

Le méta-modèle `PetriNet` permet de modéliser les réseaux de Petri classiques. Chaque réseau a un nom et contient un ensemble d'éléments `petriElements` qui sont des instances de la classe `PetriElement`.

Il ya deux types de nœuds dans le réseau, représentés par la classe `Node` :

— `Place` contient un attribut `marking` représentant le nombre de jetons présents dans cette place.

— `Transition` modélisant les transitions activables sous certaines conditions.

La classe `Arc` fait les connexions entre les nœuds. Chaque arc permet de relier une source et une cible (les deux sont de type `Node` et ont un poids `weight` indiquant le nombre de jetons transportés). Il y a aussi un attribut booléen `isReadArc` permettant d'identifier les arcs de lecture, qui lisent le marquage sans le consommer. Chaque `Node` possède des références multiples vers ses arcs entrants (`incoming`) et sortants (`outgoing`), tandis que chaque `Arc` référence son nœud source et son nœud cible.

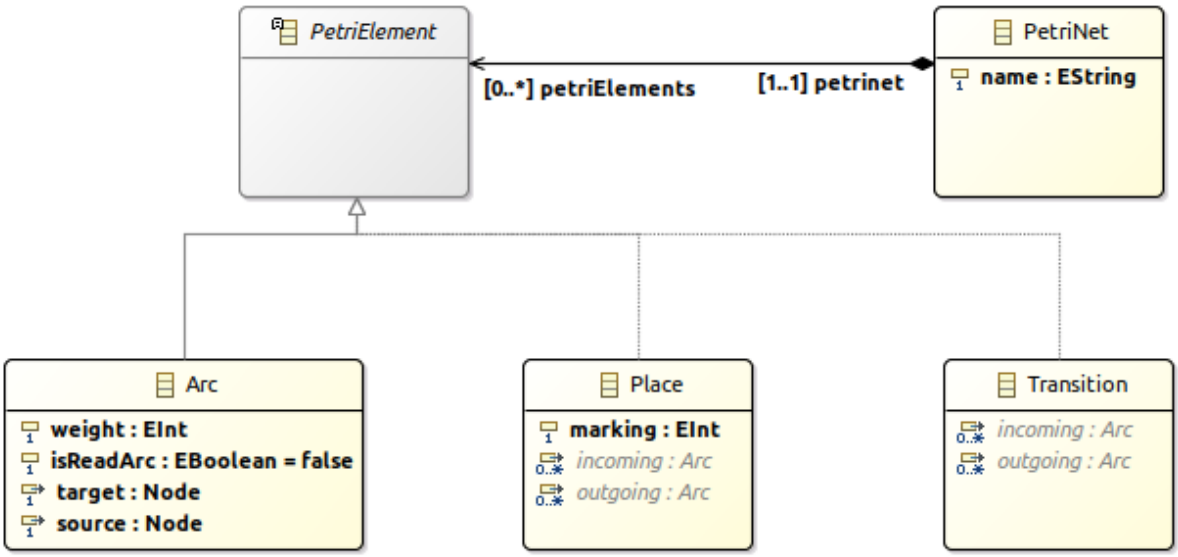

FIGURE 2 – Méta-Modèle PetriNet (.aird)

2.3 Les contraintes

L'objectif de cette partie est de spécifier des invariants qui ne sont pas pris en charge lors de la conception des métamodèles.

2.3.1 SimplePDL

Pour SimplePDL, les invariants développés sont les suivants :

- Le nom du `Process` est valide (au moins deux caractères alphanumériques, commence par une lettre).
- Un prédécesseur et un successeur d'une `WorkSequence` se situent dans le même `Process`.
- Une `WorkSequence` n'est pas réflexive (c'est-à-dire que le prédécesseur et le successeur sont distincts).
- Il n'existe pas deux `WorkSequence` identiques dans un même `Process` (même prédécesseur, même successeur, même type de lien).
- Le nom des `WorkDefinition` est unique dans un même `Process`.
- Le nom des `WorkDefinition` est valide (mêmes règles que pour le `Process`).
- Le nom des `Ressource` est unique dans un même `Process`.
- Le nom des `Ressource` est valide (mêmes règles que pour le `Process`).
- La quantité disponible d'une `Ressource` est supérieure ou égale à 1.
- Pour chaque `RessourceAllocation`, la quantité allouée est strictement positive.
- Pour chaque `RessourceAllocation`, la quantité allouée ne dépasse pas la quantité disponible de la ressource.
- Toute `RessourceAllocation` lie une `Ressource` et une `WorkDefinition` non nulles.
- Lorsqu'une `WorkDefinition` utilise plusieurs allocations pour une même `Ressource`, la somme de toutes les quantités allouées ne dépasse pas la capacité totale de cette ressource.
- Le besoin en `Ressource` se situe dans le bon intervalle (au moins 1, au plus la quantité initialement disponible).
- La `Ressource` et la `WorkDefinition` rattachées à une `RessourceAllocation` appartiennent au même `Process`.
- Le texte d'une `Guidance` n'est pas vide.

Cependant, il reste des limitations notament celui d'un cycle de dépendances qu'illustre le fichier : Process-cyclique.xmi. Le modèle est formellement valide (noms, séquences, ressources, guidance respectent vos contraintes), mais sémantiquement bloquant, car aucune tâche ne peut démarrer car chacune attend la fin d'une autre (FinishToStart entre les WorkDefinitions).

2.3.2 PetriNet

Pour PetriNet, les invariants développés sont les suivants :

— Le nom du `PetriNet` est valide (au moins deux caractères alphanumériques, commence par une lettre).
— Le nom d'un `Node` est valide (mêmes règles que pour le `PetriNet`).
— Les `Node` sont uniques dans un même `PetriNet` (pas deux nœuds de même nom).
— La position d'un `Arc` est valide (il relie une `Place` et une `Transition`, quel que soit le sens).
— Le poids d'un `Arc` est strictement positif.
— La source et la destination d'un `Arc` appartiennent au même `PetriNet`.
— Un `Arc` partant d'une `Transition` ne peut pas être un `ReadArc`.
— Une `Place` a un marquage supérieur ou égal à 0.
— Il n'existe pas d'`Arc` redondant dans un même `PetriNet` (même source et même destination).

Pour petriNet on a le cas limite suivant : une transition T1 reliée seulement par un ReadArc depuis P1 (avec 1 jeton) et aucun arc de sortie, illustré par le fichier petriNet-boucleInf. À chaque tir, T1 reste activable (le ReadArc teste mais ne consomme pas le jeton), n'enlève rien à P1 et ne remet rien ailleurs ainsi le réseau boucle indéfiniment sans jamais évoluer.

3 Éditeur graphique - SIRIUS

Cette partie définie une syntaxe graphique permettant de saisir des modèles conformes à SimplePDL. Le TP nous a fourni l'essentiel, nous avons ajouter les ressources et les besoins aisni que la distinctions des WorkSequence selon leur nature.
Voici l'arborescence de notre syntaxe graphique :

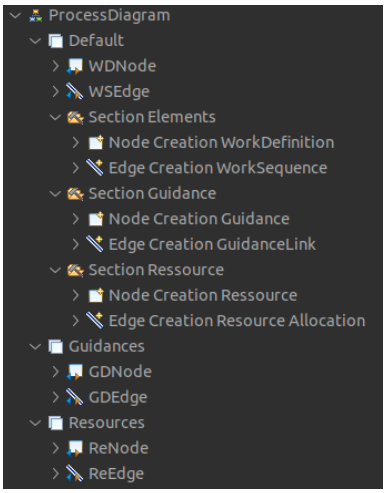

FIGURE 3 – Arborescence de la syntaxe graphique

Notre boite à outil est séparée en trois sections. Une section pour les WorkDefinition et WorkSequence, une section pour les Guidances et une sections pour les Ressources. De plus, nous avons trois calques permettant de ne pas surcharger l'affichage. Le calque principale regroupe les WorkDefinition et WorkSequence.

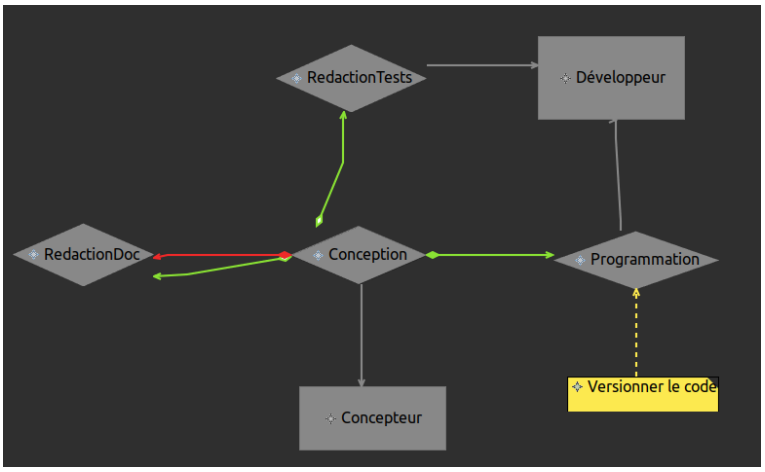

FIGURE 4 – Exemple de modèle SimplePDL avec la syntaxe graphique Sirius

4 Éditeur textuel - XTEXT

Nous définissons ici la syntaxe textuelle Xtext permettant de décrire un processus SimplePDL dans un fichier *.pdl :

— **Création d'une ressource**

```
resource <NomRessource> <QuantitéDisponible>
```

— **Création d'une tâche (WorkDefinition)**

```
wd <NomTâche>
```

— *Spécifier un besoin de ressource*

```
need <NomRessource> <QuantitéRequise>
```

— **Création d'une dépendance entre deux tâches**

```
ws <s2s|s2f|f2s|f2f>
   from <NomTâchePrédécesseur> to <NomTâcheSuccesseur>
```

— **Création d'une note de guidance**

```
note "<TexteDeLaNote>" [ for <Élément1>,<Élément2>,... ]
```

Voici un exemple de code pdl, qui reprend l'exemple de la syntaxe graphique, illustrant les différentes notions de la syntaxe :

```
process developpement {
  resource Concepteur 3
  resource Developpeur 4

  wd Conception need Concepteur 2
  wd RedactionDoc
  wd Programmation need Developpeur 3
  wd RedactionTests need Developpeur 1

  ws s2s from Conception to RedactionDoc
  ws f2f from Conception to RedactionDoc
  ws f2s from Conception to Programmation
  ws s2s from Conception to RedactionTests

  note "Versionner le code" for Programmation
}
```

FIGURE 5 – Exemple de modèle SimplePDL avec la syntaxe textuelle

5 Transformation Modèle à Modèle

Dans ce projet nous avons implémenter à l'aide d'ATL deux transformations Modèle à Modèle ainsi qu'une transformation avec EMF :

1.`SimplePDL` → `PetriNet`

1.1 ATL et EMF

Ici le but est de représenter un SimplePDL sous forme d'un réseau de Pétri. Transformation pour chaque élément de `SimplePDL` :

- **Process → PetriNet**
  Chaque processus SimplePDL est transformé en un réseau de Pétri portant le même nom.
- **WorkDefinition → Places, Transitions et Arcs**
  Pour chaque tâche :
  - 4 *places* : `name_ready`, `name_running`, `name_started`, `name_finished`
  - 2 *transitions* : `name_start`, `name_finish`
  - 5 *arcs* pour modéliser les transitions entre états
- **WorkSequence → Arc de lecture (isReadArc = true)**
  Chaque dépendance entre deux tâches est transformée en un arc de lecture entre une place (`p_started` ou `p_finished`) et une transition (`t_start` ou `t_finish`), selon le type de lien (`startToStart`, `finishToStart`, etc.).
- **Ressource → Place**
  Chaque ressource devient une place contenant un nombre de jetons égal à sa quantité.
- **RessourceAllocation → 2 Arcs**
  On modélise chaque besoin par :
  - Un arc de consommation : de la ressource vers la transition `t_start`
  - Un arc de libération : de la transition `t_finish` vers la ressource

On parcourt les différents éléments de simplePDL afin de leur appliquer une transformation vers des éléments de petriNet. Cette transformation a été implémentée de deux manières différentes : une fois en Java (avec EMF) et une autre avec ATL. Le résultat obtenu est strictement identique pour les deux méthodes.

1.1 Transformation

2.`PDL1` → `SimplePDL`

Notre métamodele de PDL1 ressemble beaucoup à SimplePDL ce qui rend la transformation assez facile. Ainsi, chaque élément de PDL1 est mappé vers son équivalent SimplePDL selon le schéma suivant :

— **Process**
  — Conserve le nom du processus.
  — Ré-attache tous les éléments enfants (tâches, séquences, ressources...) pour éviter de créer des éléments orphelins.
— **WorkDefinition**
  — Conserve le nom de la tâche.
  — Transfère les besoins en ressources vers la propriété `resourceUsages`.
— **WorkSequence**
  — Traduit le type de lien en appelant le helper de conversion.
  — Lie le prédécesseur et le successeur.
— **Ressource**
  — Conserve le nom et la quantité disponible de la ressource.
— **AllocationRessource**
  — Crée une instance de `RessourceAllocation` avec la quantité requise et référence la ressource concernée.
— **Guidance**
  — Transfère le texte de la note.

Difficultés rencontrées

Les difficultés rencontrées dans cette partie sont principalement liées à la syntaxe d'ATL, un peu compliquée à prendre en main, ainsi qu'au débogage, également complexe : la console ne fournissait pas beaucoup d'informations en cas d'erreur. Une autre difficulté a été la seconde transformation de `PDL1` vers `SimplePDL` en ATL, car le modèle .pdl1 n'existe et n'a de sens que dans l'Eclipse de déploiement. Nous avons rencontré plus de difficultés pour exécuter le code avec les bonnes entrées.

Cependant, ce qui nous a pris le plus de temps est la transformation en Java de `SimplePDL` vers `PetriNet`. Nous sommes très peu habitués à coder en Java, surtout en utilisant des fonctions qui ont été auto-générées et non codées par nous-mêmes.

6 Transformation Modèle à Texte

Dans ce projet nous avons implémenté une transformation modèle à texte avec l'outil Acceleo (l'outil TINA et LTL n'ayant pas été vus, seule cette transformation est implémentée).

Cette transforamtion permet de générer automatiquement une réprésentation graphique d'un processus SimplePDL ou d'un petrinet au format DOT. Le format `.dot` est un langage de description de graphes utilisé par l'outil *Graphviz* permettant de représenter des graphes à l'aide d'une syntaxe textuelle simple dans un fichier `.mtl`.

`SimplePDL` → **.dot**

— **WorkDefinition**
  — Représentées par des rectangles.
  — Chaque nœud porte le nom de la tâche.
— **WorkSequence**
  — Flèches orientées du prédécesseur vers le successeur.
— **Guidance**
  — Chaque guidance apparaît sous la forme d'une note.
  — Une flèche relie la note aux tâches concernées.

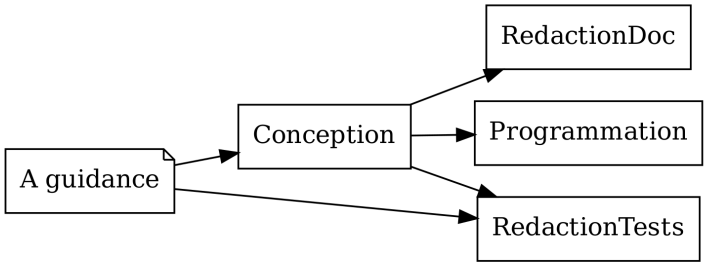

FIGURE 6 – Exemple de .dot à partir d'un modèle simplePDL

`PetriNet` → **.dot**

— **Places**
  — Représentées par des cercles.
  — Chaque cercle porte un libellé indiquant le nombre de jetons.
— **Transitions**
  — Représentées par des rectangles.
  — Chaque rectangle est nommé selon l'identifiant de la transition.
— **Arcs**
  — Flèches reliant l'élément source à l'élément cible.
  — Libellé indiquant son poids.
  — Les arcs de lecture en pointillés.

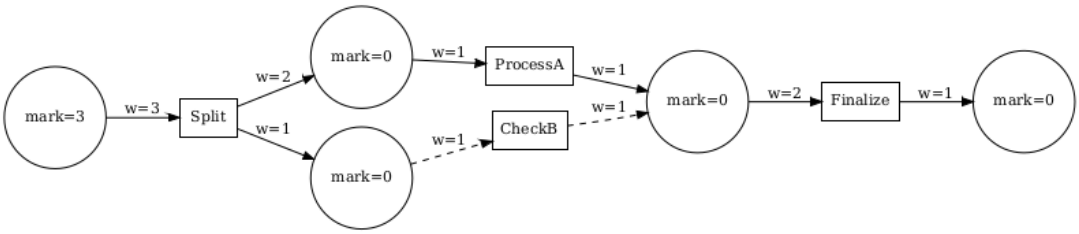

FIGURE 7 – Exemple de .dot à partir d'un modèle petriNet

Difficultés rencontrées

Cette partie était assez compliquée puisqu'il fallait comprendre la syntaxe utilisé dans le .mtl. De ce fait, nous n'avons pas réussi à extraire le type de lien des worksequence.

7 Conclusion

En conclusion , nous avons enrichi et validé les méta-modèles SimplePDL et PetriNet et ajouté des contraintes. Puis conçu deux éditeurs (graphique avec Sirius et textuel avec Xtext) pour faciliter la création et la visualisation des modèles. Nous avons automatisé les transformations SimplePDL vers PetriNet et PDL1 vers SimplePDL en ATL/EMF, ainsi que la génération de fichiers .dot via Acceleo. Les principaux défis ont porté sur la syntaxe d'ATL et MTL ainsi que le debogage en java.

8 ANNEXE - Contenu du livrable

TABLE 1 – Description des fichiers du projet

Projet Eclipse	**Fichiers**	**Description**
fr.n7.simplePDL	SimplePDL.ecore src/simplepdl/validation exemple/Process-cyclique.xmi	Méta-modèle SimplePDL Package pour les contraintes Java exemple de cas limite validé par les contraintes mais erroné
fr.n7.petriNet	PetriNet.ecore src/petrinet/validation exemple/petriNet-boucleInf.xmi	Méta-modèle PetriNet Contraintes de validation Java exemple de cas limite validé par les contraintes mais erroné
fr.n7.simplePDL2petriNetEMF	src/simplepdl2petrinetEMF/ SimplePDL2PetriNetEMF.java /exemple	Code Java de la transformation SimplePDL vers PetriNet exemple d'un simplepdl et le résultat de la transformation vers petrinet
fr.n7.simplePDL2petriNetATL	SimplePDL2PetriNetATL.atl /exemple	Code ATL de la transformation SimplePDL vers PetriNet exemple d'un simplepdl et le résultat de la transformation vers petrinet
fr.n7.simplepdl.design	description/simplepdl.odesign	Syntaxe graphique Sirius de SimplePDL
fr.n7.pdl1	src/fr/n7/PDL1.xtext	Syntaxe textuelle Xtext de PDL1
fr.n7.simplepdl.exemples	ex1.pdl1	exemple d'un pdl1 avec la syntaxe textuelle
fr.n7.pdl12simplepdl	pdl1Tosimplepdl.atl toDot.mtl /exemple	Fichier atl de transformation pdl1 vers simplepdl exemple d'un pdl1 et le résultat de la transformation vers simplepdl
fr.n7.simplepdl.todot	src/fr/n7/simplepdl/todot/main/ toDot.mtl /exemples	Fichier de transformation simplePDL vers dot exemples de .dot avec les pdf associés
fr.n7.petrinet.todot	src/fr/n7/petrinet/todot/main/ toDot.mtl /exemples	Fichier de transformation petriNet vers dot exemples de .dot avec les pdf associés