Projet : Une chaîne de vérification pour modèles de procédés

L'objectif général du projet est de proposer un écosystème permettant à un utilisateur de définir des modèles de procédés, c'est-à-dire des ensembles de tâches et de ressources, ainsi que des dépendances entre tâches, ou entre une tâche et une ressource.

Cet écosystème est à destination d'utilisateurs sans distinction de compréhension du domaine de l'informatique, ce qui nous pousse à développer des outils ergonomiques, avec pour intention de répondre aux questions que l'utilisateur pourrait se poser sur un modèle de procédé.

Afin de simplifier le développement et de rester au plus près des besoins de l'utilisateur et de sa compréhension du domaine, l'utilisation de l'ingénierie dirigée par les modèles a été retenue.

Dans ce projet, nous manipulerons donc diverses technologies afin de proposer un écosystème, une chaîne d'outil pour la gestion et la vérification de modèles de procédés.

1 Objectifs et spécification

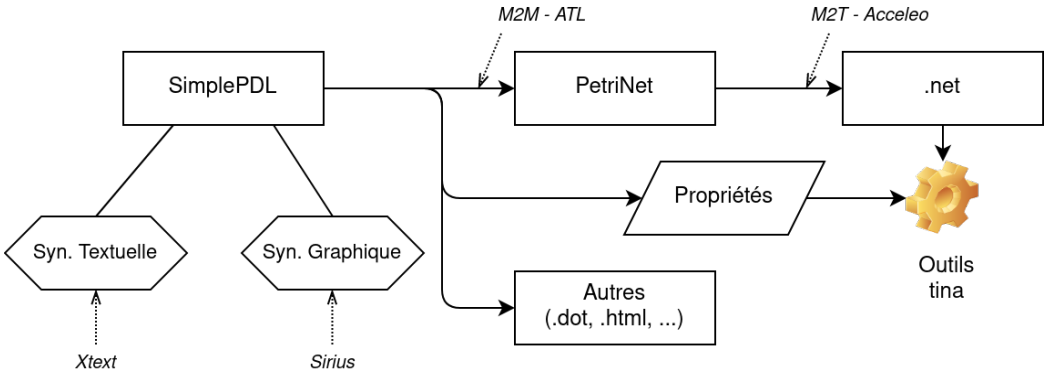

FIGURE 1 – Chaîne de transformation

L'écosystème a pour point d'entrée des *syntaxes concrètes*, qui permettent à l'utilisateur d'afficher, éditer et sérialiser des modèles de procédés. Des contraintes sur ces modèles peuvent être vérifiées, permettant d'exclure les modèles mal formés.

Une fois créés, ces modèles sont transmis à divers outils, qui permettent, notamment, de les afficher sous d'autres formes (HTML, dot), d'en extraire des propriétés, et surtout de les transformer en *réseaux de Pétri*, ce qui permet de vérifier dessus des propriétés plus complexes (terminaison, atteignabilité).

1.1 Méta-modèles et sémantique statique

Modèles de procédés Les modèles de procédés suivent le méta-modèle SIMPLEPDL, esquissé en TP et enrichi pour les besoins de ce projet. Ce méta-modèle doit respecter les contraintes suivantes :

R1.1 Un modèle de procédé se compose de **tâches** et de **ressources**

R1.1.1 Une tâche est caractérisée par un **nom**

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

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

R1.2.1 Les dépendances entre tâches ont un « type », qui modélise quel aspect de la tâche est visé par la dépendance :

- le *début* de la tâche dépendante est conditionné par le *début* de la tâche dont elle dépend (start-to-start)
- le *début* de la tâche dépendante est conditionné par la *fin* de la tâche dont elle dépend (finish-to-start)

— la *fin* de la tâche dépendante est conditionnée par le *début* de la tâche dont elle dépend (start-to-finish)
— la *fin* de la tâche dépendante est conditionnée par la *fin* de la tâche dont elle dépend (finish-to-finish)

R1.3 Une tâche 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 tâche et une seule ressource

R1.3.2 On doit pouvoir spécifier la quantité de ressource nécessaire à une tâche 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)

R1.5 Un modèle de procédé doit respecter un ensemble de contraintes structurelles et sémantiques, qui permettent d'écarter les modèles « mal formés ».

Le méta-modèle SimplePDL doit par ailleurs respecter des contraintes de formes, en particulier :
— il doit être réalisé avec Ecore, et doit être *valide* dans ce méta-méta-modèle
— ses divers aspects (attributs, références, relations) doivent être *pertinents*
— l'architecture du méta-modèle doit être précise et rigoureuse (attention aux arités et à l'appartenance des concepts), et doit aussi être aisément extensible
— le méta-modèle doit respecter les conventions usuelles (pour les identifiants en particulier)

L'exigence R1.5 consiste essentiellement à définir des *contraintes OCL*. On attend de ces contraintes qu'elles limitent au maximum la création de modèles valides mais erronés, qui n'ont pas de sens ou poseront problème dans les outils par la suite.

Attention : à la suite d'un bug critique dans OCL, la réalisation de l'exigence R1.5 se fera à l'aide d'une implémentation Java. Il en va de même pour l'exigence R2.6 (voir plus bas).

Réseaux de Pétri Pour les besoins du projet, il est nécessaire de définir un méta-modèle pour représenter des *réseaux de Pétri*. La forme et l'architecture de ce méta-modèle sont libres, mais doivent respecter les contraintes suivantes :

R2.1 Un réseau de Pétri se compose de **places** et de **transitions**

R2.2 Une place est **nommée** (contrainte de l'outil) et présente des jetons (**marquage**)

R2.3 Une transition est **nommée** (contrainte de l'outil) 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 de borne supérieure (noté ω, ou `w` dans l'outil)

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*

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

R2.6 Un réseau de Pétri doit aussi respecter un ensemble de contraintes structurelles et sémantiques, de sorte qu'un modèle conforme et qui respecte ces contraintes représente un réseau de Pétri bien formé, qui peut être manipulé par les outils de la boîte à outil `tina`

Le méta-modèle des réseaux de Pétri doit par ailleurs respecter les mêmes contraintes de forme que SimplePDL.

Démonstration des méta-modèles Afin de montrer l'utilisation des méta-modèles et de leurs contraintes, il est demandé de réaliser des *exemples*. Ces exemples montreront des cas normaux d'utilisation, mais aussi et surtout les limites de ces méta-modèles : exemples de modèles faux/invalides exclus par le méta-modèle et/ou les contraintes, exemples de modèles faux/invalides mais non exclus par le méta-modèle et/ou les contraintes, exemples de modèles valides, exemples de modèles justes mais exclus par le méta-modèle et/ou les contraintes (le cas échéant).

1.2 Syntaxes concrètes

Pour pouvoir afficher et éditer des modèles de procédés, l'écosystème présente 3 syntaxes concrètes, associées à des outils.

Éditeur arborescent L'éditeur arborescent permet une manipulation basique des modèles. Il présente, sous la forme d'un arbre, le contenu du modèle, avec un nœud pour chaque élément du modèle.

R3.1 L'éditeur arborescent doit représenter toutes les informations du modèle (en particulier, source et cible des dépendances, type pour les dépendances entre tâches et quantité requise pour les dépendance entre tâche et ressource)

Éditeur graphique Cet éditeur, développé avec Sirius, donne une représentation graphique des modèles de procédés, et peut-être utilisé pour les créer et les modifier.

R4.1 L'éditeur doit permettre d'afficher *tous* les éléments d'un modèle de procédé

R4.2 L'éditeur doit, au travers de sa *palette*, permettre de créer n'importe quel élément d'un modèle de procédé

R4.3 Les représentation (nœuds, arcs) pour chaque élément d'un modèle de procédé doivent être pertinentes et différenciées

R4.4 L'outil doit présenter au moins deux *calques*, pour séparer les commentaires du reste du modèle

Éditeur textuel Cet éditeur, développé avec Xtext, représente les modèles de procédés sous la forme de texte structuré (coloration syntaxique).

R5.1 La syntaxe doit représenter *tous* les éléments du méta-modèle

R5.2 Les éléments d'un modèle doivent être introduits à l'aide de mots-clefs pertinents

R5.3 La syntaxe doit être ergonomique et éviter les éléments superflus

R5.4 La syntaxe n'a pas à suivre le méta-modèle, mais une *transformation* modèle-à-modèle doit être fournie pour passer d'un modèle obtenu avec Xtext vers un modèle SimplePDL conforme

1.3 Transformation vers réseaux de Pétri

Transformation modèle à modèle Afin de pouvoir répondre à des questions complexes sur « l'exécution »des modèles de procédés, il est nécessaire de les transformer en réseaux de Pétri. Ces derniers ayant été méta-modélisés, il s'agit en fait de concevoir une transformation de modèle à modèle.

R6.1 La transformation prend en entrée un modèle de procédé conforme à SimplePDL et crée en sortie un modèle de réseau de Pétri conforme au méta-modèle développé

R6.2 La transformation doit prendre en compte *tous* les éléments du modèle de procédé qui soient *sémantiquement significatifs* (pas les commentaires)

R6.3 Le réseau de Pétri correspondant à un modèle de procédé doit être valide (sous hypothèse que le modèle en entrée est lui-même valide)

R6.4 Le réseau de Pétri doit encoder la *sémantique opérationnelle* d'un modèle de procédé, et doit donc respecter ses contraintes et les contraintes induites par les dépendances

Il est demandé d'implémenter la transformation de **deux** façons : avec Java, et avec ATL.

Transformation modèle à texte Le réseau de Pétri obtenu par transformation est conforme au méta-modèle, mais ne peut pas être instrumenté directement par les outils `tina`[1]. Il est donc nécessaire de transformer ce modèle en code qui pourra être interprété par ces outils. Cette tâche est réalisée à l'aide de l'outil Acceleo.

R7.1 La transformation prend en entrée un modèle de réseau de Pétri (supposé valide et respectant les contraintes associées) et crée en sortie un fichier textuel respectant la syntaxe des outils `tina`

R7.2 Tous les éléments du réseau en entrée doivent être retranscrits dans le fichier de sortie

R7.3 Le fichier doit être exempt d'erreur (si le modèle en entrée est valide) ; on fera notamment attention aux contraintes sur les identifiants des éléments dans `tina`

Propriétés à vérifier On veut vérifier, sur un modèle de procédé quelconque, deux aspects importants :
— le procédé peut-il finir ? (atteignabilité)
— le procédé finit-il toujours ? (terminaison)

Ces questions correspondent à des propriétés LTL que l'ont peut demander à l'outil `selt` de vérifier sur le réseau de Pétri correspondant au modèle de procédé.

Ces propriétés dépendent du modèle de procédé. On peut donc les générer automatiquement à l'aide d'une transformation modèle à texte.

R8.1 La transformation prend en entrée un modèle de procédé conforme à SimplePDL et crée en sortie un fichier contenant des propriétés LTL qui peuvent être lues par `selt`

R8.2 Les propriétés générées doivent permettre de répondre aux deux questions ci-dessus

Validation de la transformation Afin de garantir l'exigence 1.3, il est pertinent d'extraire d'un modèle de procédé des propriétés sur son « exécution », que l'on pourra ensuite demander à `tina` de vérifier sur le résultat de la transformation. Si ces propriétés sont respectées, on a alors de bonnes raison de penser que la transformation s'est bien passée, et que le réseau de Pétri résultant a bien du sens vis-à-vis du modèle de procédé en entrée.

Par exemple : une tâche ne peut finir que si elle a commencée, une tâche ne peut plus commencer si elle a finie, une ressource prise au début d'une tâche est restituée à la fin de celle-ci, etc.

Pour cela, on peut définir une transformation Acceleo de SimplePDL vers des propriétés qui seront examinées par `selt`.

R9.1 La transformation prend en entrée un modèle de procédé conforme à SimplePDL et crée en sortie un fichier contenant des propriétés LTL qui peuvent être lues par `selt`

R9.2 Les propriétés générées doivent encoder les particularités des exécutions possibles (des « scénarios ») d'un modèle de procédé

1.4 Autres fonctionnalités

Transformation vers Dot Pour avoir une représentation sous forme d'image d'un modèle de procédé, on définira une transformation modèle à texte vers le format Dot de Graphviz[2].

R10.1 La transformation prend en entrée un modèle de procédé conforme à SimplePDL et crée en sortie un fichier `.dot` conforme au langage Dot

R10.2 Le diagramme Dot ainsi généré doit représenter toutes les informations contenues dans le modèle de procédé (on peut s'inspirer de la représentation développée pour Sirius)

1. `https://projects.laas.fr/tina`
2. `https://graphviz.org/`

2 Modalités de composition, de rendu et d'évaluation

Ce projet se réalise en binôme, éventuellement en trinôme (pas de groupe de 1 ni de groupe de plus de 3 étudiants). La composition des binômes doit être renseignées sur la page Moodle du module.

Le rendu se fera sur un dépôt Moodle situé sur la page du module.

2.1 Livrables

Le rendu consiste en une archive `.tar.gz` (rien d'autre) contenant, **au minimum** :

L1 Le méta-modèle SimplePDL (un fichier `.ecore`)

L2 Le méta-modèle de réseau de Pétri (un fichier `.ecore`)

L3 ~~Les contraintes OCL pour SimplePDL (un ou plusieurs fichiers `.ocl`)~~
⇒ **Le programme de validation des contraintes du méta-modèle SimplePDL (ensemble de sources Java)**

L4 ~~Les contraintes OCL pour les réseaux de Pétri (un ou plusieurs fichiers `.ocl`)~~
⇒ **Le programme de validation des contraintes du méta-modèle de réseau de Pétri (ensemble de sources Java)**

L5 Des exemples de modèles de procédés et de réseaux de Pétri (plusieurs fichiers `.xmi`/`.simplepdl`/`.petrinet`/etc.)

L6 La définition de la syntaxe graphique Sirius (un fichier `.odesign`)

L7 La définition de la syntaxe textuelle (un fichier `.xtext`)

L8 L'éditeur arborescent (projets Eclipse avec les sources du méta-modèle + projet `edit` et `editor`)

L9 Les transformations modèle-à-modèle de SimplePDL vers réseau de Pétri (un fichier `.atl` + un fichier `.java`)

L10 La transformation modèle-à-texte de modèle de réseau de Pétri vers la syntaxe `tina` (un ou plusieurs fichiers `.mtl`)

L11 Les transformations modèle-à-texte de modèle de procédé vers propriétés LTL (deux ou plus fichiers `.mlt`)

L12 La transformation modèle-à-texte de modèle de procédé vers Dot (un ou plusieurs fichiers `.mtl`)

Ce rendu est accompagné d'un **rapport**. Le rapport **doit impérativement respecter les contraintes suivantes** :

- format de fichier PDF, police d'écriture *avec serif* (Times, Garamond, Georgia, etc.), taille 11pt
- langue française + règles de formatage de l'édition française (alinéa à chaque paragraphe notamment)
- une page de garde avec titre, numéro et membres du groupe, année, etc., suivie d'une page blanche (le contenu du rapport commence sur une page impaire)
- un plan (sections, sous-sections, etc.) pertinent et clair + une table des matières après la page de garde
- une introduction, qui rappelle le contexte et la problématique, et annonce dans les grandes lignes ce qui a été fait et le contenu du rapport
- une conclusion, qui explique les principales difficultés rencontrées, ce qui a été fait ou n'a pas été fait, et les enseignements tirés du projet
- des schémas, des morceaux de code (mais pas trop) et des captures d'écran (*au minimum* ***une capture d'écran*** *par outil*)
- les deux méta-modèles (SimplePDL et réseaux de Pétri) sous **forme graphique**, dans des diagrammes Ecore[3] **bien arrangés et lisibles**
- pas de capture d'écran avec fond sombre (par ex., celles issues d'un terminal en mode clair sur sombre)
- une liste détaillée des fichiers livrés, avec nom du fichier et petite description (on peut faire référence aux fichiers dans le rapport mais il faut vraiment une (sous-)section à part qui contient cette liste)

3. Par exemple celui obtenu avec l'outil *Ecore diagram*

Au niveau du contenu, le rapport sert de point d'entrée à votre rendu. Nous attendons à ce qu'il explique ce qui a été fait, **avec vos propres mots**, les problèmes rencontrés, les décisions prises et les choix de conception faits lorsque cela a été nécessaire (par ex. pourquoi telle ou telle contrainte, pourquoi telle architecture de méta-modèle, quels détails d'implémentation pour les transformations, etc.). Essayez d'éviter de paraphraser le sujet de projet ; essayez plutôt de reformuler avec vos propre mots.

L'usage d'un modèle de langage et assimilés (type GPT) pour générer tout ou partie du rapport est strictement interdit dans le cadre de ce projet.

Conformément à l'article L122-5 n° 3 du code la propriété intellectuelle, il est possible d'utiliser une œuvre (textuelle ou picturale) rendue publique pour contribuer à son discours à condition d'en indiquer clairement et sans ambiguïté la provenance et l'auteur, et de clairement identifier la citation en tant que tel.

2.2 Présentation orale

Le rendu s'accompagne d'une présentation orale du projet. Lors de cette présentation, le groupe de projet montre l'utilisation des divers outils/fonctionnalités demandées dans le sujet.

Inutile de montrer la réalisation de ces fonctionnalités (il y a le rendu pour ça) ; l'idée est vraiment de faire une démonstration technique, où l'on peut voir les divers éléments fonctionner dans Eclipse.

Il est vivement conseillé d'élaborer un scénario pour le jour de l'oral !

L'oral en lui-même consiste en 15 minutes de préparation (venir avant son horaire de passage) et 10 minutes de présentation (ferme). Des questions de compréhension pourront être posées, auxquelles il conviendra de répondre précisément et succinctement.

2.3 Modalités d'évaluation

Les critères pour l'évaluation du projet sont donnés à titre indicatif et sont les suivants :

- Méta-modèles (SimplePDL et réseau de Pétri) :
  - Respect des conventions
  - Pertinence structurelle (composition, abstraction, arités, etc.)
  - Pertinence des éléments (classes, attributs, etc.)
  - Respect des exigences (ressources pour SimplePDL notamment...)
  - Exemples
- Contraintes (SimplePDL et réseau de Pétri) :
  - Propriétés de base (validité des attributs, ...)
  - Propriétés sémantiques
  - Formulation correcte, contexte pertinent
  - Exemples/démonstration
- Syntaxe concrète graphique (SimplePDL) :
  - Affichage fonctionnel et exhaustif
    dont ressources et dépendances associées
  - Éditeur (palette) fonctionnel et exhaustif
    dont ressources et dépendances associées
  - Calque différent pour les commentaires (Guidance)
  - Démonstration
- Syntaxe concrète textuelle (SimplePDL) :
  - Grammaire exhaustive
    dont ressources et dépendances associées
  - Transformation de modèle Xtext vers SimplePDL
  - Éditeur avec coloration syntaxique (démonstration)
- Transformation modèle-à-modèle (Java et ATL) :
  - Transformation exhaustive

dont ressources et dépendances associées
  - Structure et qualité du code (modularité, commentaires, helpers/méthodes utilitaires)
  - Démonstration
- Transformation modèle-à-texte :
  - Transformation de modèle de réseau de Pétri à syntaxe tina
  - Transformation de SimplePDL en propriétés LTL (propriétés du modèle de procédé)
  - Transformation de SimplePDL en propriétés LTL (validation de la transformation)
  - Transformation de SimplePDL vers Dot
  - Qualité du code (commentaires, queries et template, etc.)
  - Démonstration
- Autres fonctionnalités : tout autre fonctionnalité peut faire l'objet de points bonus
- Oral :
  - Présentation préparée
  - Qualité des explications
  - Réponses aux questions
- Rapport :
  - Respect des contraintes
  - Pertinence du contenu
  - Précision et rigueur

Pour chaque critère, un *score* est déterminé :

A Respecte parfaitement les exigences, fait correctement

B Respecte bien les exigences, fait mais avec des défauts

C Respecte partiellement les exigences, partiellement fonctionnel

D Insuffisant dans le respect des exigences, non ou peu fonctionnel

F Non fait

Nous n'expliciterons pas la façon dont une lettre est transformée en note, ni le coefficient relatif à chaque critère. Notez simplement que le rapport compte pour au moins un tiers de la note.