

# Chaîne de vérification de modèles de processus

Ce mini-projet consiste à produire une chaîne de vérification de modèles de processus SimplePDL dans le but de vérifier leur cohérence, en particulier pour savoir si le processus décrit peut se terminer ou non. Pour répondre à cette question, nous utilisons les outils de *model-checking* définis sur les réseaux de Petri au travers de la boîte à outils Tina. Il nous faudra donc traduire un modèle de processus en un réseau de Petri.

Les TP présenteront les outils qui devront être utilisés pour réaliser ce mini-projet. Chaque TP traite un aspect développé dans le mini-projet. Le TP est une introduction qui devra être complétée en consultant au moins la documentation des outils utilisés.

Le mini-projet correspond à ce qui est fait dans les TP avec une petite extension (les ressources) de manière à vérifier que les TP ont bien été compris et assimilés.

## 1 Objectifs du mini-projet

Ce mini-projet consiste pour l'essentiel à définir la chaîne de vérification de modèles de processus (dont la description est donnée en section 1.1). Ce travail sera fait en TP, les principales étapes étant les suivantes :

1. Définition des métamodèles avec Ecore.
2. Définition de la sémantique statique avec OCL (Complete OCL).
3. Utilisation de l'infrastructure fournie par EMF pour manipuler les modèles.
4. Définition de transformations modèle à texte (M2T) avec Acceleo, par exemple pour engendrer la syntaxe attendue par Tina à partir d'un modèle de réseau de Petri ou engendrer les propriétés LTL à partir d'un modèle de processus.
5. Définition d'une transformation de modèle à modèle (M2M) avec EMF/Java et avec ATL
6. Définition de syntaxes concrètes textuelles avec Xtext.
7. Définition de syntaxes concrètes graphiques avec Sirius.

Ce travail fait en TP sera complété par la validation de la chaîne de transformation (section 1.2) et l'ajout de ressources au langage de description de processus (section 1.3).

### 1.1 Description des modèles de processus

Le modèle de procédé utilisé est inspiré de SPEM<sup>1</sup>, norme de l'OMG. Nous donnons entre parenthèses le vocabulaire utilisé par cette norme. Dans un premier temps, nous nous intéressons à des processus simples composés seulement d'activités (WorkDefinition) et de dépendances (WorkSequence). La figure 1 donne un exemple de processus qui comprend quatre activités :

---

1. <http://www.omg.org/spec/SPEM/2.0/>

Conception, RédactionDoc, Programmation et RédactionTests. Les activités sont représentées par des ellipses. Les arcs entre activités représentent les dépendances. Une étiquette permet de préciser la nature de la dépendance sous la forme « étatToAction » qui précise l'état qui doit être atteint par l'activité source pour réaliser l'action sur l'activité cible. Par exemple, on ne peut commencer RédactionTests que si Conception est commencée. On ne peut commencer Programmation que si Conception est terminée. On ne peut terminer RédactionTests que si Programmation est terminé.

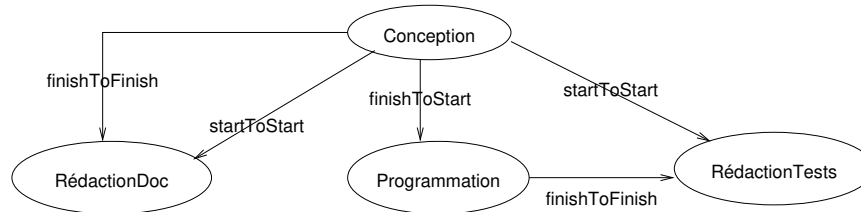


FIGURE 1 – Exemple de modèle de procédé

## 1.2 Validation de la transformation SimplePDL2PetriNet

Comme pour tout programme écrit, il est important de valider la transformation de modèle. Afin de valider la transformation SimplePDL vers PetriNet, une possibilité est de vérifier que les invariants sur le modèle de processus sont préservés sur le modèle de réseau de Petri correspondant. Ces invariants sont appelés *propriétés de sûreté*. En voici quelques exemples :

- chaque activité est soit non commencée, soit en cours, soit terminée ;
- une activité terminée n'évolue plus.

On peut alors écrire une transformation modèle à texte qui traduit ces propriétés de sûreté sur le modèle de Petri. L'outil *selt* permettra alors de vérifier si elles sont effectivement satisfaites sur le modèle de réseau de Petri. Si ce n'est pas le cas, c'est que la traduction contient une erreur ou que l'invariant n'en est pas un !

## 1.3 Ajout des ressources

Pour réaliser une activité, des ressources peuvent être nécessaires. Une ressource peut correspondre à un acteur humain, un outil ou tout autre élément jouant un rôle dans le déroulement de l'activité. Ici, nous nous intéressons simplement aux types de ressources nécessaires et au nombre d'occurrences d'un type de ressource. Par exemple, il peut y avoir deux développeurs, trois machines, un bloc-note, etc. Un type de ressource sera seulement caractérisé par son nom et la quantité d'occurrences de celle-ci.

Pour pouvoir être réalisée, une activité peut nécessiter plusieurs ressources (éventuellement aucune). Par exemple, l'activité *RédactionTest* nécessite un testeur (une occurrence de la ressource de type Testeur) et deux machines (deux occurrences de la ressource Machine). Les occurrences de ressources nécessaires à la réalisation d'une activité sont prises au moment de son démarrage et rendues à la fin de son exécution. Bien entendu, une même occurrence de ressource

ne peut pas être utilisée simultanément par plusieurs activités. Les types de ressource et leur nombre d'occurrences sont définis en même temps que le procédé lui-même.

Voici une affectation possible de ressources pour le processus décrit à la figure 1. Une ligne correspond à un type de ressource. Elle indique la quantité totale de la ressource ainsi que le nombre d'occurrences de cette ressource nécessaire à la réalisation d'une activité.

	Quantité	Conception	RédactionDoc	Programmation	RédactionTest
concepteur	3	2			
développeur	2			2	
machine	4	2	1	3	2
rédacteur	1		1		
testeur	2				1

## 2 Déroulement du mini-projet

Ce mini-projet est un travail en binôme.

Les techniques mises en œuvre doivent être celles présentées dans le module Ingénierie Dirigée par les Modèles (IDM).

### 2.1 Tâches à réaliser

Pour chaque partie, voici les tâches à réaliser et les documents à rendre.

- T<sub>1</sub> Compléter le métamodèle SimplePDL pour prendre en compte les ressources.
- T<sub>2</sub> Définir le métamodèle PetriNet.
- T<sub>3</sub> Développer un éditeur graphique SimplePDL pour saisir graphiquement un modèle de processus, y compris les ressources.
- T<sub>4</sub> Définir les contraintes OCL pour capturer les contraintes qui n'ont pu l'être par les métamodèles (SimplePDL et PetriNet).
- T<sub>5</sub> Donner une syntaxe concrète textuelle de SimplePDL avec Xtext.
- T<sub>6</sub> Définir une transformation SimplePDL vers PetriNet en utilisant EMF/Java.
- T<sub>7</sub> Définir une transformation SimplePDL vers PetriNet en utilisant ATL.
- T<sub>8</sub> Valider la transformation SimplePDL vers PetriNet en faisant des tests.
- T<sub>9</sub> Définir une transformation PetriNet vers Tina en utilisant Acceleo.
- T<sub>10</sub> Engendrer les propriétés LTL permettant de vérifier la terminaison d'un processus et les appliquer sur différents modèles de processus.
- T<sub>11</sub> Engendrer les propriétés LTL correspondant aux invariants de SimplePDL pour valider la transformation écrite (voir section 1.2).

## 2.2 Documents à rendre

Les consignes pour rendre les documents suivants seront données sur la page du module (les documents seront rendus via SVN).

- D<sub>1</sub> Les métamodèles SimplePDL et PetriNet (ainsi que des images de ces métamodèles).
- D<sub>2</sub> Les fichiers de contraintes OCL associés à ces métamodèles, avec des exemples (et contre-exemples) qui montrent la pertinence de ces contraintes.
- D<sub>3</sub> Le code Java de la transformation modèle à modèle.
- D<sub>4</sub> Le code ATL de la transformation modèle à modèle.
- D<sub>5</sub> Le code Aceleo des transformations modèle à texte.
- D<sub>6</sub> Les modèles Sirius décrivant l'éditeur graphique pour SimplePDL.
- D<sub>7</sub> Le modèle Xtext décrivant la syntaxe concrète textuelle de SimplePDL.
- D<sub>8</sub> Des exemples de modèles de processus (en expliquant leur intérêt).
- D<sub>9</sub> Un document concis (rapport) qui explique le travail réalisé. Attention, c'est ce document qui servira de point d'entrée pour lire les éléments rendus.

## 3 Complément optionnel et non évalué

Si vous trouvez ce mini-projet trop court, nous vous proposons l'extension suivante.

La réalisation d'une activité nécessite du temps qui sera, dans notre cas, modélisé comme un intervalle [temps\_min, temps\_max] où temps\_min (respectivement temps\_max) est le temps minimal (respectivement maximal) pour réaliser l'activité.

On veut que tous les comportements précédents (avant l'introduction du temps) puissent encore se produire. Le temps sera donc traité comme un observateur d'une activité qui pourra dire si cette dernière s'est terminée dans le délai imparti, trop tôt ou trop tard. On pourra ainsi vérifier si le processus peut se terminer en respectant les délais impartis.