Rapport de projet
-
Ingénierie Dirigée par les Modèles

Table des matières

Introduction

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 principales étapes sont les suivantes :

1. Définition des métamodèles avec Ecore
2. Définition de la sémantique statique avec OCL
3. Utilisation de l'infrastructure fournie par EMF pour manipuler les modèles
4. Définition de syntaxes concrètes textuelles avec Xtext
5. Définition de syntaxes concrètes graphiques avec Sirius
6. Définition d'une transformation de modèle à modèle (M2M) avec Java et avec ATL
7. Définition de transformations modèle à texte (M2T) avec Acceleo

Pour ce projet, nous utilisons divers outils tel que Tina ou encore Eclipse Modeling Tools (version Eclipse comportant des outils de modélisation).
C'est donc par le biais de ce rapport que va vous être présenter le travail réalisé.

I – Métamodèles (Ecore)

L'objectif de cette partie est de compléter le métamodèle SimplePDL pour prendre en compte les ressources, ainsi que de définir le métamodèle PetriNet.

Fichiers :

- Métamodèle SimplePDL : *fr.n7.simplePDL/SimplePDL.ecore*
- Image du métamodèle de SimplePDL : *Documents/SimplePDL.jpg*
- Métamodèle PetriNet : *fr.n7.petriNet/petriNet.ecore*
- Image du métamodèle de PetriNet : *Documents/PetriNet.jpg*

SimplePDL

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 : *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

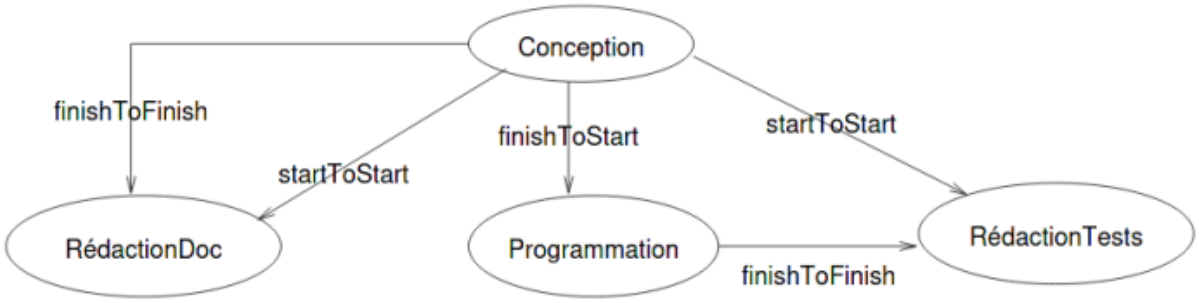

Figure 1 - Exemple de modèle de procédé

Le métamodèle qu'il nous faut compléter est le suivant :

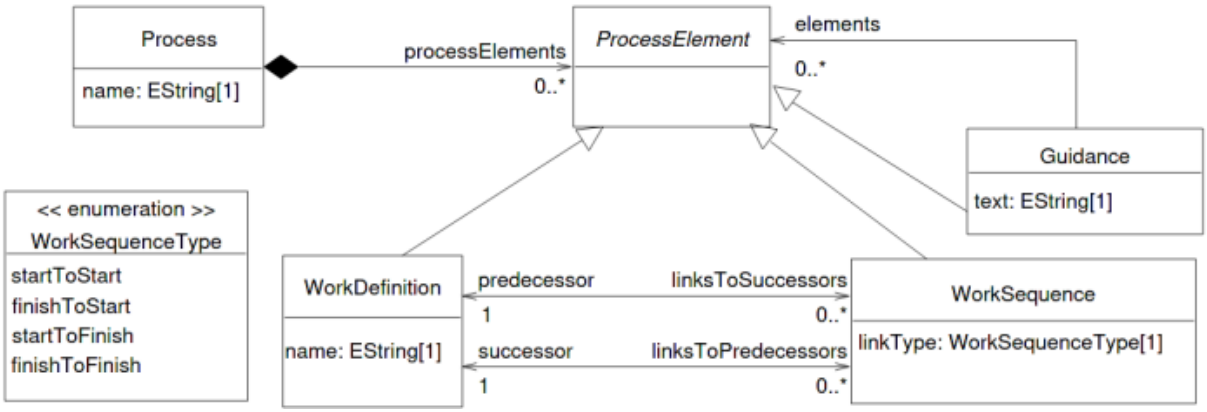

Figure 2 - Métamodèle SimplePDL fournit

Il nous fallait prendre en compte l'utilisation des ressources nécessaires pour effectuer les différentes activités.

Nous avons donc ajouté deux nouvelles classes : *Resource* et *Need*. La première définit les différentes ressources disponibles pour l'ensemble des processus et la seconde indique la quantité de ressource nécessaire pour la réalisation d'une action.

Resource hérite de la classe abstraite `ProcessElement` car elle représente un élément du processus. Cette classe un attribut `name` qui représente son nom, ainsi qu'un attribut `nbAvailableResources` qui représente le nombre de ressources de ce type disponibles.

Need référence la classe Resource pour désigner un type de ressource (`[1..1] resource`). De même, la classe `Ressource` référence aussi la classe `Need`, cependant elle n'a pas de limite d'instances associées (`[0..*] needs`). Cette classe possède un attribut `nbResources` qui indique le nombre de ressources du même type nécessaire. La classe `Need` référence également la classe `WorkDefinition`.

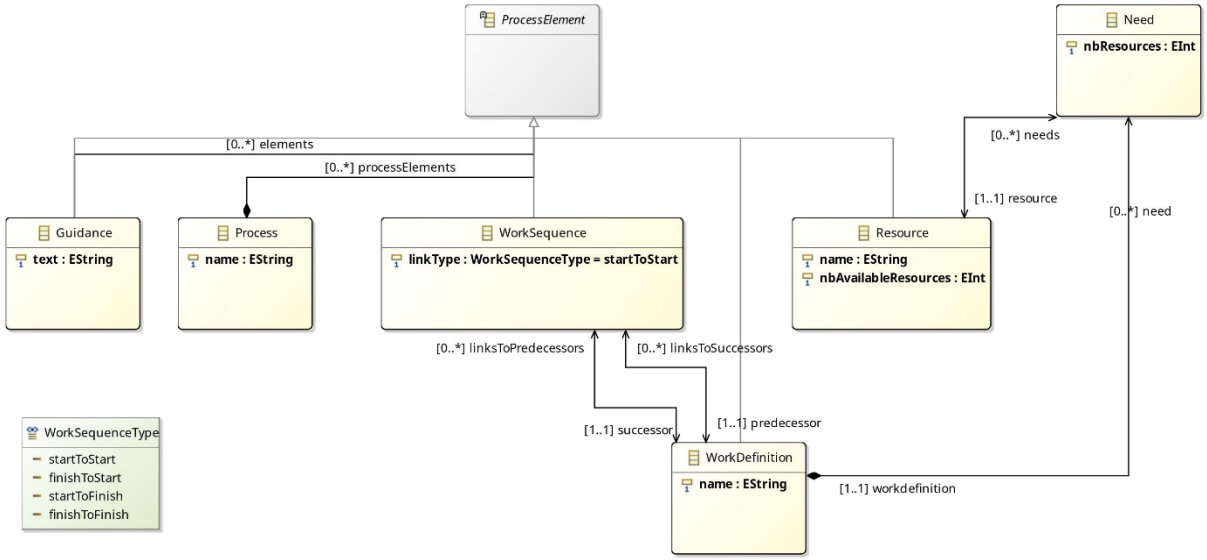

Figure 3 - Métamodèle de SimplePDL

PetriNet

La classe principale de notre réseau de Pétri est `PetriNet`. Cette classe représente le réseau de Pétri entier, en lui attribuant un nom (attribut `name`).

Un réseau de Petri est constitué de plusieurs éléments (ici il s'agit de la classe abstraite `PetriElement`). Cette classe représente un élément du réseau de Pétri parmi tous les éléments possibles (un nœud ou un arc). Cette classe hérite donc des deux éléments qui peuvent être utilisés : la classe `Arc` et la classe `Node`. La première représente un arc orienté (grâce à `source` et `target`) entre une transition et une place. La seconde représente un nœud du réseau de Pétri, nous avons décidé de la faire hériter des éléments qui la compose : la classe `Transition` et la classe `Place`. La première représente une transition dans le réseau de Pétri. La seconde représente une place dans le réseau de Pétri. Elle possède un attribut `marking` qui indique le nombre de jetons initial dans la place représentée.

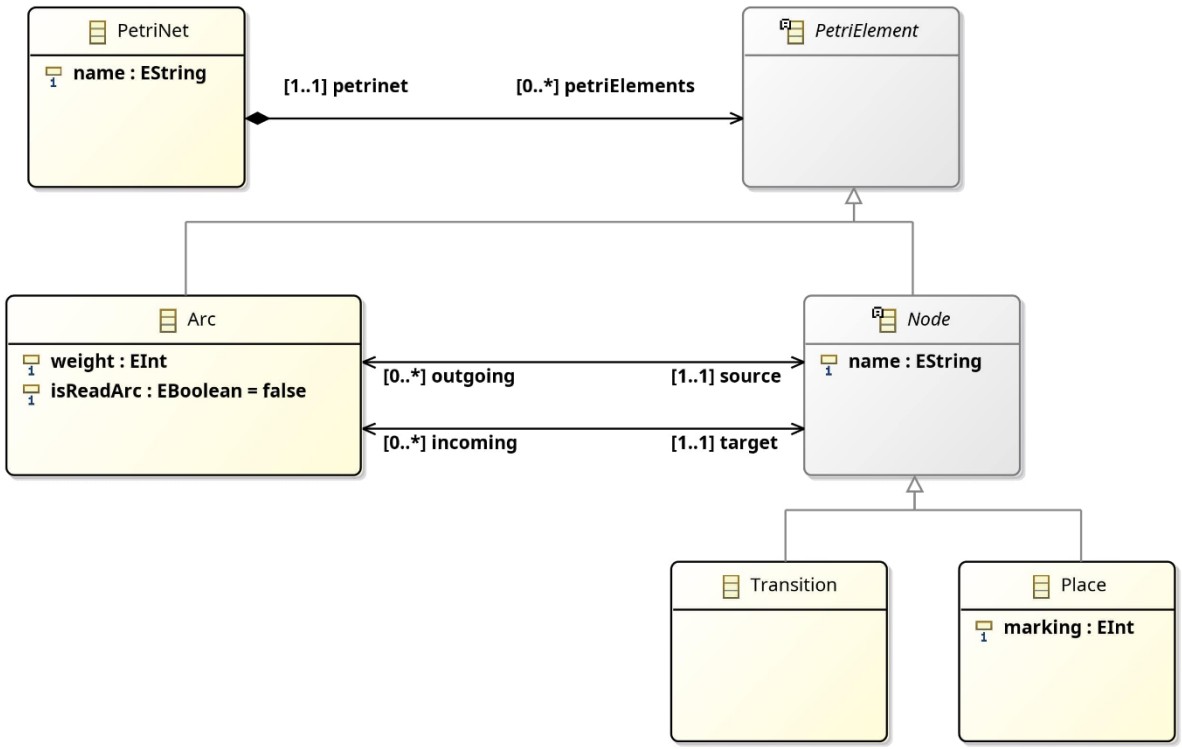

Figure 4 - Métamodèle de PetriNet

II – Sémantique statique (OCL)

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

Fichiers :

- Contrainte OCL de SimplePDL : *fr.n7.simplePDL/simplepdl.ocl*
- Exemples de la pertinence des contraintes :
  - Exemple de bon fonctionnement : *fr.n7.simplePDL.exemples/exemple.xmi*
  - Contre exemple sur la réflexivité : *fr.n7.simplePDL.exemples/exempleFail_reflexivite.xmi*
  - Contre exemple sur les ressources : *fr.n7.simplePDL.exemples/exempleFail_ressources.xmi*
- Contrainte OCL de PetriNet : *fr.n7.petriNet/petriNet.ocl*
- Exemples de la pertinence des contraintes :
  - Exemple pour une transformation ATL : *fr.n7.petriNet.exemples/exempleATL.xmi*
  - Exemple pour une transformation EMF : *fr.n7.petriNet.exemples/exempleEMF.xmi*
  - Contre exemple sur la position d'un arc : *fr.n7.petriNet.exemples/exempleFail_arcPosition.xmi*

SimplePDL

Pour tester le modèle SimplePDL, nous avons défini les invariants suivants :

- Nom du `Process` valide
- Nom des `WorkDefinition` valide
- Nom des Resource valide
- Nom unique des `WorkDefinition`
- Nom unique des `Resource`
- Quantité disponible de ressources
- Possession d'un successor ainsi qu'un `predecessor` d'une `WorkSequence`
- Avoir le `successor` et le `predecessor` distincts pour une `WorkSequence`
- `Need` se trouve dans le bon intervalle
- Une `Resource` et une `WorkDefinition` appartiennent au même `Process`
- Pas de besoins similaires (pas deux `Need` rattachés à la même `Resource` et `WorkDefinition`)

PetriNet

Pour tester le modèle `PetriNet`, nous avons définit les invariants suivants :

- Nom du `PetriNet` valide
- Nom du `Node` valide
- Nom unique des `Node`
- Position d'un `Arc` valide
- `weight` d'un `Arc` strictement positif
- `marking` d'une `Place` strictement positif ou nul
- Une `source` et une `target` d'un `Arc` appartiennent au même `PetriNet`
- Un Arc partant d'une `Transition` n'est pas un `ReadArc`
- Pas d'`Arc` similaires (la même `source` et `target`)

III – Syntaxes concrètes textuelles (Xtext)

L'objectif de cette partie est de définir une syntaxe concrète textuelle permettant de décrire un processus SimplePDL dans un fichier `.pdl` grâce au greffon XText.

Fichier :

- Syntaxes concrètes textuelles : *fr.n7.pdl1/src/fr/n7/PDL1.xtext*

Utilisation de notre syntaxe PDL :

- Création d'un processus :
  - `process <processName> {...}`
- Définition d'une ressource :
  - `create <resourceQuantity> of <resourceName>`
- Définition d'une activité (WorkDefinition) :
  - `task <taskName> {...}`
- Spécification d'un besoin sur une ressource :
  - `need <resourceQuantity> of <resourceName>`
- Initialisation d'une dépendance :
  - `dep <startToStart|startToFinish|finishToStart|finishToFinish> from <taskName> to <taskName>`
- Création d'une note pour un élément :
  - `note <noteText> for <elementName>[,<elementName>,<elementName>,...]`

IV – Syntaxes concrètes graphiques (Sirius)

L'objectif de cette partie est de définir une syntaxe graphique permettant de saisir des modèles conformes à SimplePDL. Nous utilisons l'outil Sirius pour la réalisation de cette syntaxe.

Fichier :

- Syntaxes concrètes graphiques : *fr.n7.simplepdl.design/description/simplepdl.odesign*

Nous avons séparé notre `ProcessDiagram` en trois Layer :

- `Default` (contient les `WorkDefinition` et `WorkSequence`)
  - Section Elements : les nœuds
  - Section Links : les liens
- `Guidances` (ainsi que les liens reliant les guidances aux éléments)
- `Resources` (ainsi que les `Need`)

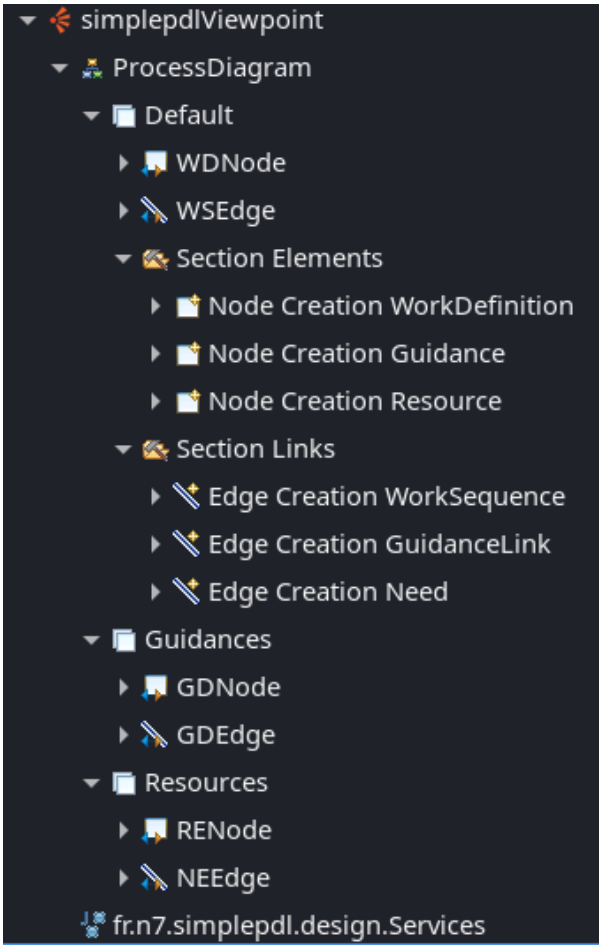

Figure 5 - Arborescence Sirius

Nous avons utilisé le langage *AQL* (*Acceleo Query Language*) pour définir des syntaxes conditionnelles sur les `WorkSequence`.

Malheureusement, nous n'avons pas réussi à faire fonctionner la vue graphique et n'avons donc pas pu essayer notre syntaxe concrète.

V – Transformation M2M

L'objectif de cette partie est de pouvoir passer du modèle de SimplePDL vers celui de PetriNet.

Fichiers :

- Transformation avec EMF : *fr.n7.simplePDL.toPetriNetEMF/src/SimplePDLtoPetriNetEMF/SimplePDLtoPetriNetEMF.java*
- Transformation avec ATL : *fr.n7.simplePDL.toPetriNetATL/SimplePDLtoPetriNet.atl*

EMF

Eclipse Modeling Framework permet de passer d'un modèle à un autre en utilisant du code Java. À partir des deux métamodèles que nous avons écrits, du code Java a été généré permettant d'interagir avec eux : des classes représentant les différents éléments des métamodèles et des méthodes pour les créer.

Notre programme parcoure les éléments du modèle *SimplePDL* et applique les transformations dans le but de créer le modèle *PetriNet* à l'aide des outils générés précédemment. Afin de facilité l'exécution du code pour ne pas avoir à écrire en dur le modèle d'entrée, nous avons mis en place deux paramètres : le premier est le chemin d'accès du modèle *SimplePDL* à transformer et le second est le chemin d'accès du modèle *PetriNet* généré.

ATL

Le langage *ATL* permet la traduction de modèle d'un métamodèle vers un modèle d'un autre métamodèle. Le principe d'*ATL* est de définir un ensemble de règles de transformations permettant de passer d'un élément d'un métamodèle à un élément d'un autre métamodèle.

On peut alors remarquer qu'il est plus simple, pour passer d'un modèle à un autre, d'utiliser *ATL* que *EMF* car il nous suffit de rédiger une règle par élément de *SimplePDL* que l'on veut transformer en *PetriNet*.

VI – Transformation M2T

L'objectif de cette partie est de pouvoir passer du modèle de PetriNet vers le format texte à utiliser dans la boite à outil Tina.

Fichiers :

- Code généré avec Acceleo : *fr.n7.petriNet.exemples/exemple.net*
- Transformation vers Tina : *fr.n7.petriNet.toTina/src/fr/n7/petriNet/toTina/main/toTina.mtl*
- Transformation vers LTL : *fr.n7.simplePDL.toLTL/src/fr/n7/simplePDL/toLTL/main/toLTL.mtl*

Acceleo

Nous avons créé deux méthodes afin d'aider à la rédaction du template principal. La première, `getPlaces`, récupère toutes les places parmi une liste d'instances de `PetriElement` et la seconde, `getTransitions`, réalise la même opération pour récupérer des transitions.
Nous avons ensuite une méthode `toTina` pour transformer une transition en texte.

Par la suite, grâce à Acceleo, on obtient un fichier `.net` qui contient le code Tina utile pour engendrer le graph suivant :

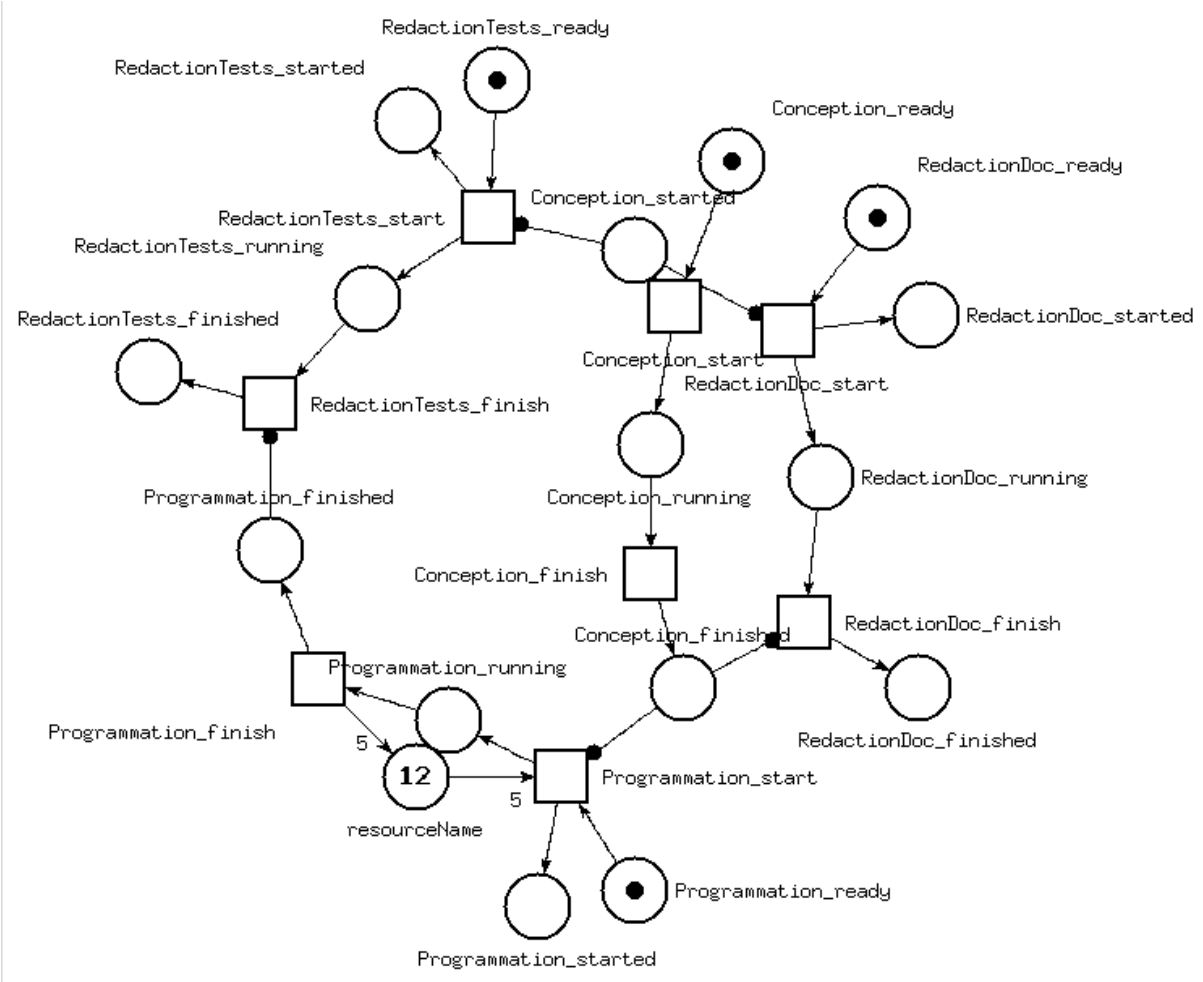

Figure 6 - Graph généré sur Tina

LTL

Pour la rédaction de ce template, nous avons vérifié que toutes les activités arrivent dans l'état `finished`. Par la suite, nous vérifions qu'une activité ne peut être que dans un état à la fois. Enfin, nous vérifions qu'une activité démarrée reste toujours démarrée.

Malheureusement, nous avons rencontré un problème dans le déploiement du greffon et nous n'avons donc pas pu tester notre template.

Conclusion

Pour conclure, ce mini-projet, ainsi que les TPs, nous aurons appris à nous servir de divers outils de modeling. Nous avons malheureusement rencontré de nombreux problèmes liés à *Eclipse Modeling Tools*, mais il a tout de même était intéressant de découvrir toutes les étapes de transformation d'un modèle à un autre.