Sémantique statique d'un DSML avec OCL

Version d'Eclipse à utiliser : `/mnt/n7fs/ens/tp_dupont/modelling-2023-09/eclipse/eclipse`

Les outils OCL ne fonctionnent plus correctement. Nous allons donc utiliser directement Java (et l'API des streams) pour implanter la sémantique statique. Ceci est fait dans la partie 1. La partie 2 est l'ancien sujet qui a été conservé pour que vous donner une idée de comment fonctionne OCL qui est utilisé pour exprimer la sémantique statique et la vérifier sur un modèle. Enfin, la partie 3 consiste à définir la sémantique statique de l'extension de SimplePDL qui intègre les ressources et sur les réseaux de Petri. Elle sera implantée en Java. Il est toutefois conseillé d'exprimer d'abord ces propriétés en utilisant OCL.

1 Implanter en Java les contraites statiques

Exercice 1 : Comprendre le code Java fourni

Le code fourni doit être ajouté au projet `fr.n7.simplePDL` dans le dossier [1] `src`, dans un paquetage `simplepdl.validation`.

Répondre aux questions suivantes qui aideront à comprendre le code fourni. On pourra dessiner le diagramme de classe de ce système tout en répondant aux questions suivantes.

1.1. En lisant la classe principale `ValidateSimplepdl`, indiquer ce que représentent les arguments (paramètres de `main`).

Expliquer ce qui permet de charger en mémoire un modèle.

1.2. La classe `ValidationResult` est utilisée pour enregistrer toutes les erreurs qui sont détectées lors de l'analyse d'un modèle. Lister les méthodes que l'on pourra utiliser sur cette classe.

1.3. Lors de la génération du code du modèle grâce à EMF, une classe `SimplepdlSwitch` est engendrée dans le paquetage `simplepdl.util`. Expliquer ce que fait la méthode `doSwitch` de cette classe. Il est à noter que cette méthode est appelée par la méthode `doSwitch(EObject)` définie dans `org.eclipse.emf.ecore.util.Switch`.

1.4. La classe `SimplePDLValidator` réalise la vérification effective du modèle : elle parcourt tous les objets du modèle et, pour chaque objet, elle évalue les invariants définis sur la classe de cet objet. Les invariants non vérifiés conduisent à l'enregistrement d'un message d'erreur.

Comment se fait le parcours des objets du modèle ?

Quelles sont les vérifications implantées ?

Exercice 2 : L'API des streams de Java

L'API des streams favorise une approche fonctionnelle qui conduit souvent à des expressions plus simples à lire/écrire et plus efficaces à évaluer que l'utilisation de boucles classiques.

Lire les explications données dans Processing Data with Java SE 8 Streams, Part 1 et Aggregate Operations de tutoriel sur les collections.

1. Plutôt que d'utiliser `src` qui contient le code engendré par EMF, on pourrait créer un dossier `src-validation` et l'ajouter au *Source PATH*.

L'interface Stream présente les principales opérations sur les streams, en particulier : filter, map, distinct, sorted, limit, etc. Ces méthodes s'appliquent à un stream et produisent un stream.

D'autres méthodes terminent le stream et permettent généralement de produire un objet à partir du stream. Certaines sont générales comme `collect` et `reduce`. D'autres sont « spécialisées » comme :

1. count, sum, max, min, empty,
2. findFirst
3. allMatch, anyMatch, noneMatch
4. toList

Que font les méthodes précédentes ?

Regarder le code fourni qui utilise déjà les streams (SimplePDLValidator et ValidationResult en particulier).

Exercice 3 : Compléter le code fourni
Compléter le code Java fourni pour implanter les contraintes suivantes :

1. Une Guidance doit avoir un texte non vide.
2. On ne doit pas avoir deux dépendances différentes de même type entre deux mêmes activités. Par exemple, `A1 --s2s-> A2` et `A1 --s2s-> A2` est interdit dans un même modèle.

2 Compléter un méta-modèle par des contraintes statiques

Exercice 4 : Vérification statique avec OCL
Nous avons utilisé Ecore (ou EMOF) pour définir un méta-modèle pour les processus. Il est rappelé à la figure 1. Certaines contraintes sur les modèles de processus ont pu être exprimées. C'est par exemple le cas de celles qui concernent les multiplicités. La propriété `opposite` permet aussi d'exprimer une contrainte entre deux références pour retrouver une notion proche de la relation d'association bidirectionnelle d'UML.

Cependant, le langage de méta-modélisation (Ecore ou EMOF) ne permet pas d'exprimer toutes les contraintes que doivent respecter les modèles de processus. Aussi, on complète la description structurelle du méta-modèle par des contraintes exprimées en OCL.

Le méta-modèle Ecore et les contraintes OCL définissent la **syntaxe abstraite** du langage de modélisation considéré.

4.1. *OCL.* Expliquer les éléments apparaissant sur le fichier OCL du listing 1 correspondant au métamodèle de SimplePDL.

4.2. *Évaluation des contraintes OCL sur un modèle.* Vérifions maintenant notre modèle (process1-ko.xmi) par rapport à ces contraintes OCL.

4.2.1. Ajouter le fichier `process1-ko.xmi` au projet et le valider (clic droit sur l'élément racine Process, puis *Validate*). Seules les contraintes EMF sont vérifiées.

4.2.2. Pour prendre en compte un fichier de contraintes OCL, il faut commencer par le charger. On fait un clic droit (par exemple sur la première ligne du modèle dans l'éditeur arborescent) et

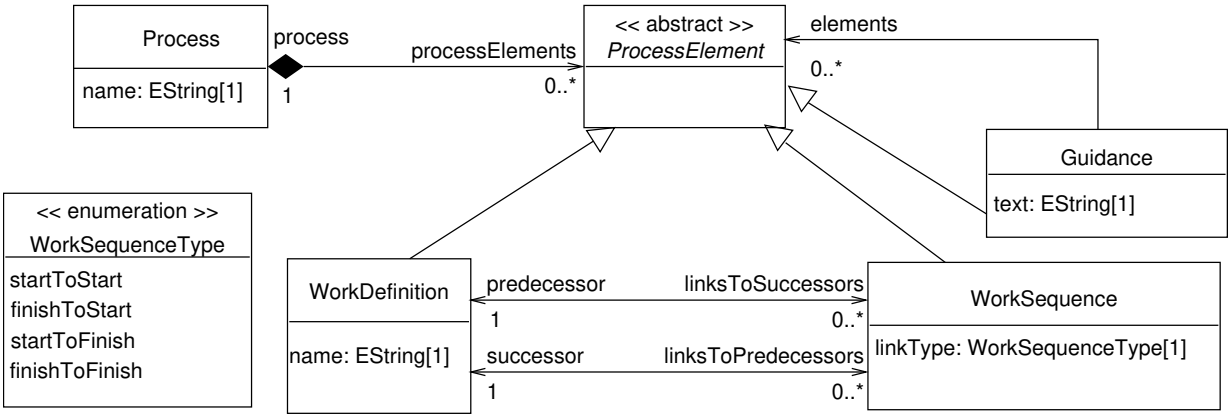

FIGURE 1 – Nouveau méta-modèle de SimplePDL

Listing 1 – Quelques contraintes OCL sur Le métamodèle SimplePDL

```
import 'SimplePDL.ecore'

package simplepdl

context Process
inv warningSeverity: false
inv withMessage('Explicit␣message␣in␣process␣' + self.name + '␣(withMessage)'): false
inv errorSeverity: null

context Process
inv validName('Invalid␣name:␣' + self.name):
   self.name.matches('[A-Za-z_][A-Za-z0-9_]*')

context ProcessElement
def: process(): Process =
   Process.allInstances()
      ->select(p | p.processElements->includes(self))
      ->asSequence()->first()

context WorkSequence
inv successorAndPredecessorInSameProcess('Activities␣not␣in␣the␣same␣process␣:␣'
      + self.predecessor.name + '␣in␣' + self.predecessor.process().name+ '␣and␣'
      + self.successor.name + '␣in␣' + self.successor.process().name):
   self.process() = self.successor.process()
   and self.process() = self.predecessor.process()

endpackage
```

on fait *OCL > Load Document*. On peut alors choisir le fichier OCL qui est alors ajouté dans les ressources à la fin du modèle.

Valider le modèle (clic droit sur l'élément racine Process, puis *Validate*), constater que le modèle est invalide et comprendre l'origine des messages d'avertissements et d'erreurs affichés.

On constate que tous les avertissements ne sont pas affichés (une erreur stoppe l'évaluation des autres expressions du même contexte). Mettre en commentaire l'invariant `errorSeverity` dans le fichier OCL et valider à nouveau le modèle.

4.3. *La console OCL.* La console OCL permet d'exécuter une expression OCL sur un élément d'un modèle. Ceci est pratique pour évaluer par morceaux une expression compliquée et ainsi la mettre au point.

Deux consoles existent, l'historique *OCL Console* et la nouvelle *Xtext OCL Console*. On préfèrera la seconde. Pour les obtenir, on fait un clic droit sur un élément du modèle (par exemple l'élément racine Process) et on choisit *OCL > Show Xtext OCL Console*.

Dans la partie inférieure de la console, on peut écrire une expression OCL (*self* correspond à l'élément sélectionné du modèle). Sa valeur s'affichera dans la partie supérieure. Écrire successivement les deux expressions suivantes :

```
self.name
self.processElements
```

4.4. *Nouveau fichier OCL.* Pour créer un nouveau fichier OCL, on peut sélectionner le métamodèle concerné dans l'explorateur, faire un clic droit puis *New > Others...*, choisir *Complete OCL File*, adapter le nom proposé et faire *Finish*. Le fichier est créé avec un exemple de contrainte.

Créer le fichier nouveau.ocl pour définir des contraintes sur SimplePDL.

4.5. *Compléter les contraintes de SimplePDL.* Exprimer les contraintes suivantes sur SimplePDL et les évaluer sur des exemples de modèles de processus :

1. deux activités différentes d'un même processus ne peuvent pas avoir le même nom.
2. une dépendance ne peut pas être réflexive.
3. le nom d'une activité doit être composé d'au moins deux caractères.
4. le nom d'une activité ne doit être composé que de lettres, chiffres ou soulignés, un chiffre ne peut pas être première position.

Exercice 5 : Contraintes sur les ressources (mini-projet)
Définir les contraintes appropriées concernant les ressources.

3 Application au mini-projet

Exercice 6 : Contraintes sur les ressources (mini-projet)
Définir les contraintes appropriées concernant les ressources.

Exercice 7 : Sémantique statique des réseaux de Petri
Définir les contraintes OCL définissant la sémantique statique du métamodèle des réseaux de Petri. Les valider sur différents modèles de réseaux de Petri.