UNIVERSITÉ DE NANTES

Projet MDE

Enseignant :
Gerson SUNYE

Auteurs :
Tristan JARRY
Julien OUVRARD
Thibaut PICHAUD

pour décembre 2015

Table des matières

1 Introduction

Le module MDE (Model Driven Engineering) nous a permis d'apprendre les principes de l'ingénierie des modèles. Afin de concrétiser nos apprentissages nous devions réaliser, dans le cadre d'un projet, un langage décrivant un diagramme UML. Pour cela nous avons utilisé la technologie *Xtext*. Cette technologie permet d'implémenter son propre DSL (Domain-Specific Language) textuel. Autrement grâe à *Xtext* on peut définir son propre langage afin de définir et décrire un certains types de modèles. Plus précisement, dans notre cas, il s'agira d'un diagramme de composants.
Pour illustrer le travail réalisé ce rapport se décompose en 3 parties : un rappel sur les diagrammes de composants, une explication du langage créé et enfin les règles ATL qui permettent de transformer un modèle construit avec notre langage en un modèle conforme au standard UML.
Les sources du projet se trouvent sur github à l'adresse suivante : `https://github.com/masters-info-nantes/compoNantes`

2 Rappels

Un diagramme de composant permet de définir les relations entre plusieurs composants d'un système. On y retrouve donc les composants, qui sont eux même composés de différentes parties : des ports, des interfaces. Entre ces composants peuvent se trouver des connecteurs ou des dépendances qui font la jonction de leurs interfaces. La différence entre les deux jonctions réside dans le fait que le connecteur se place entre les composants interne d'un composant alors que la dépendance va s'exercer à l'extérieur.

3 Langage créé

Notre langage se nomme CompoNantes, il a été créé grâce à XText. Il comporte 6 règles et 3 énumérations.

3.1 Énumérations

3.1.1 Type d'interface

Une interface peut être requise ou fournie c'est pourquoi l'énumération concernant les types s'écrit comme suit :

```
enum InterfaceType:
    REQUIRED='required' | PROVIDED='provided'
;
```

3.1.2 Type de connecteur

Il existe deux types de connecteurs : assembly et delegation. *Assembly* fait la jonction entre les interfaces requise et fournie alors que *delegation* va joindre les interfaces de même type d'un composant interne et celle de son conteneur.

```
enum ConnectorType:
    ASSEMBLY='assembly' | DELEGATION='delegation'
;
```

3.1.3 Type de dépendances

Les dépendances entres composants sont des deux types suivants :

```
enum DependencyType:
    USAGE='usage' | ASSOCIATION='association'
;
```

3.2 Règles

3.2.1 QualifiedName

Cette règle va permettre de définir le nom d'un objet grâce à l'arborescence de ce dernier

```
QualifiedName:
    ID ('.' ID)*
;
```

3.2.2 Component

Elle représente un composant dans notre langage. Elle s'écrit comme suit :

```
Component:
   'component' name=ID '{'
      ports+=Port*
      interfaces+=Interface*
      components+=Component*
      connectors+=Connector*
      dependencies+=Dependency*
   '}'
;
```

On remarque qu'un composant peut avoir des ports, des interfaces, des composants internes, des connecteurs et des dépendances.

3.2.3 Port

Un port dans notre langage n'a qu'un nom

```
Port:
   'port' name=ID
;
```

3.2.4 Interface

```
Interface:
    'interface' type=InterfaceType name=ID 'port' port=[Port]
;
```

Une interface à donc un type qui va permettre de la catégoriser, un nom pour l'identifier ainsi qu'un port auquel elle est reliée.

3.2.5 Connector

```
Connector:
    'connector' type=ConnectorType name=ID '{'
        'from' int_from=[Interface|QualifiedName]
        'to' int_to=[Interface|QualifiedName]
    '}'
;
```

3.2.6 Dependency

```
Dependency:
    'dependency' type=DependencyType name=ID '{'
        'from' component_from=[Interface|QualifiedName]
        'to' component_to=[Interface|QualifiedName]
    '}'
;
```

4 Règles ATL

ATL permet de transformer un modèle conforme à un métamodèle en un modèle identique conforme à un autre métamodèle. L'opération ici consiste à transformer un modèle conforme à CompoNantes en un modèle conforme au diagramme de composants en UML.
Tout d'abord nous définissons deux *helpers* afin de pouvoir récupérer les interfaces requises et fournies d'un composant.

```
helper context COMPO!Component
    def: getProvided() : Set(COMPO!Interface) =
    self.interfaces->collect(e|e.type.toString()='provided')
;
helper context COMPO!Component
    def: getRequired() : Set(COMPO!Interface) =
    self.interfaces->collect(e|e.type.toString()='required')
;
```

Puis après la règle de transformation pour les composants, on assigne à chaque élément d'un composant en UML son alter ego dans notre langage. Ce qui donne :

```
rule CpnComponentToUmlComponent {
    from
        CpnComp : COMPO!Component
    to
        UmlComp : UML!Component (
            name <- CpnComp.name,
            ownedPort <- CpnComp.ports,
            provided <- CpnComp.getProvided(),
            required <- CpnComp.getRequired(),
            ownedConnector <- CpnComp.connectors
        )
}
```

Enfin les règles pour les éléments des composants, avec le même principe que précédemment :

```
rule CpnIntToUmlInt {
    from
        CpnInt : COMPO!Interface
    to
        UmlInt : UML!Interface (
            name <- CpnInt.name
        )
}

rule CpnConToUmlCon {
    from
        CpnCon : COMPO!Connector
    to
        UmlCon : UML!Connector (
            name <- CpnCon.name
        )
}
```

```
rule CpnPrtToUmlPrt {
    from
        CpnPrt : COMPO!Port
    to
        UmlPrt : UML!Port (
            name <- CpnPrt.name
        )
}
```

5 Conclusion

Grâce à la réalisation de ce projet, nous avons pu mieux comprendre les principes du MDE, et ainsi mieux percevoir son utilité. Le travail au cours de se projet s'est décomposé en deux parties. La première à été l'analyse des différents documents nous présentant le diagramme de composant. Après avoir fait cette analyse, une seconde s'est offerte à nous. En effet avant de passer directement à la partie programmation, nous avons du établir toutes les règles que le diagramme de composant doit suivre. Autrement dit nous devions définir quelles étaient ses contraintes. Suite à cela nous avons pu commencer l'implémentation du langage avec *Xtext*. Après ce développement nous sommes passé à la partie concernant l'ATL, cependant, nous n'avons pu réaliser pleinement cette partie. Chaque membre du groupe avait une tâche bien précise ce qui nous a permis d'avancer correctement sur le projet. Tous les membres du groupe ont travaillé de manière efficace avec une bonne communication ce qui a permis de ne pas avoir de conflit ni de problèmes au sein de ce groupe.
Même si nous n'avons pu terminer le projet complètement, il fut très intéressant car comme dit précédemment nous avons mieux comprendre les principes et l'utilité du MDE.