SmallUML, Un UML simplifié

LUCAS Lenny
MOREAU Benjamin

11 décembre 2016

1 Introduction

L'objectif de notre projet est de définir un langage de modélisation représentant une version simplifiée du diagramme de classe UML. Nous retiendrons seulement les concepts de Classes, Associations, Généralisation/Spécialisation, Attributs, Opérations, DataTypes, Énumérations, Multiplicités.

Nous allons dans un premier temps définir notre modèle en EMF. Puis, nous spécifierons la syntaxe concrète de SmallUML à l'aide de Xtext. Enfin, ATL nous permettra de définir des règles de transformation pour passer automatiquement des instances de SmallUML vers UML.

2 Le model smallUML

Le modèle SmallUML suivant a été définit à l'aide d'EMF Ecore.

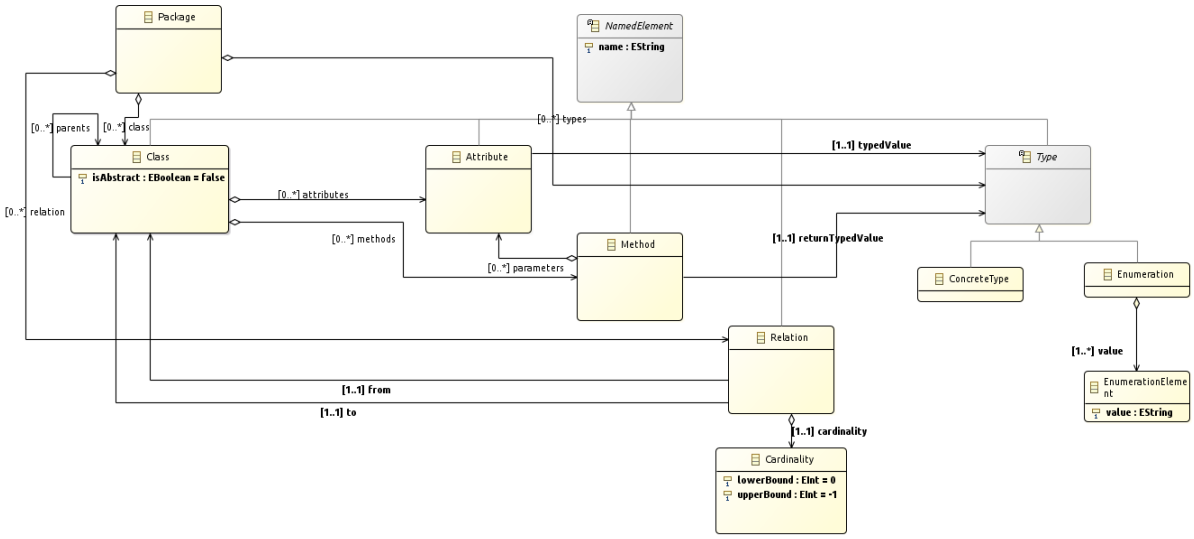

FIGURE 1 – SmallUML

3 La syntaxe concrète

La syntaxe concrète du langage SmallUML à été dans un premier généré automatiquement depuis le modèle Ecore. Nous avons ensuite apporté quelques modifications pour le rendre plus simple à utiliser. Le Xtext permettant de définir la syntaxe concrète est définie ci-dessous.

Listing 1 – Xtext : syntaxe concrète SmallUML

```
grammar org.alma.uml.smalluml.SmallUml with org.eclipse.xtext.common.Terminals

import "http://www.univ-nantes.fr/smalluml"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Package returns Package:
        {Package}
        ( class+=Class ';' )*
        ( relation+=Relation ';')*
        ( types+=Type ';')*

;

Type returns Type:
        Enumeration | ConcreteType;

Class returns Class:
        (isAbstract?='Abstract')?
        'Class'
        name=ID ('extends' parents+=[Class] ( "," parents+=[Class])* )?
        '{'
                ('attributes' '{' attributes+=Attribute ( attributes+=Attribute)* '}' )?
                ('methods' '{' methods+=Method ( methods+=Method)* '}' )?
        '}';

Relation returns Relation:
        'Relation'
        name=ID
        '{'
                'from' from=[Class]
                'to' to=[Class]
                'cardinality' cardinality=Cardinality
        '}';

EString returns ecore::EString:
        STRING | ID;

EBoolean returns ecore::EBoolean:
        'true' | 'false';

Attribute returns Attribute:
        name=EString ':' typedValue=[Type] ';';

Method returns Method:
        {Method}
        name=ID '('( parameters+=Attribute ( "," parameters+=Attribute)*  )?')' ':' (returnTypedValue=[Type])?  ';';


Cardinality returns Cardinality:
        '[' lowerBound=EInt',' upperBound=EInt']';

EInt returns ecore::EInt:
        '-'? INT;

Enumeration returns Enumeration:
        'Enumeration'
        name=ID
        '{'
                value+=EnumerationElement ( "," value+=EnumerationElement)*
        '}';

ConcreteType returns ConcreteType:
        {ConcreteType}
        'ConcreteType'
        name=EString;

EnumerationElement returns EnumerationElement:
        value=ID;
```

Cette grammaire est simple mais puissante. Elle est composée de 3 blocs principaux : une déclaration de toutes les classes, puis de toutes les associations entre ces classes et enfin de tout les types utilisés par ces classes.

Une classe est composée de variables typées et de méthodes qui sont elles-mêmes composées de paramètres typées et d'un type de retour. Cette grammaire permet aussi l'héritage entre classe. Chaque types utilisé ici doit être déclaré au préalable.

Les associations sont, quand a elles, composées d'un nom, de 2 liens vers des classes (une de départ et une d'arrivée) et d'une cardinalité.

Pour les Datatypes, nous avons mis en place des types génériques devant être déclarés avant leur utilisation.

Nous pouvons tester notre langage en modélisant un domaine représentant des animaux et leur propriétaire. C'est aussi ce domaine que nous utiliseront pour tester notre outil de transformation ATL.

Listing 2 – SmallUML : Domaine propriétaires d'animaux

```
Abstract Class Animal {
        attributes {
                nom: String;
        }
        methods {
                crier (voix : String;):String;
                marcher ():Void;
        }
};

Class Chien extends Animal {
        methods {
                ramenerBalle ():Boolean;
        }

};

Class Chat extends Animal {};

Class Personne {
        attributes {
                nom: String;
        }
        methods {
                donnerAManger (nourriture: String;):Boolean;
        }
};

Relation proprietaire {
        from Personne
        to Animal
        cardinality [0,1]

};

Enumeration Etat {
        Mort,
        Vivant,
        Mort-Vivant
};

ConcreteType String;
ConcreteType Boolean;
ConcreteType Void;
```

4 La transformation

Pour la transformation, nous avons créé un .xmi représentant notre modèle Propriétaire-Animal associé au méta-modèle SmallUML. La transformation suivante transforme ce modèle en un autre modèle Propriétaire-Animal au format .xmi associé au méta-modèle UML5.0.

Listing 3 – ATL : Transformation du smallUML vers UML5.0

```
-- @path small=/SmallUML/model/smalluml.ecore
-- @path uml=/UML/model/uml.ecore

module Small2UML;
create OUT : uml from IN : small;

rule package {
        from
                s : small!Package
        to
                t : uml!Package (
                        name <- 'Mainpackage',
                        packagedElement <- s.class.union(s.relation)
                )
}

lazy rule smallAttributes2Property {
        from s : small!Attribute
        to t : uml!Property (
                name <- s.name,
                type <- s.typedValue
```

```
        )
}

rule smallMethods2Operation {
        from s : small!Method
        to t : uml!Operation (
                name <- s.name,
                type <- s.returnTypedValue,
                ownedParameter <- s.parameters -> collect(parameter | thisModule.Parameter2Attribute(parameter))
        )
}

lazy rule Parameter2Attribute {
        from
                s : small!Attribute
        to
                t : uml!Parameter (
                        name <- s.name,
                        type <- s.typedValue
                )
}

rule SmallClass2Class {
        from s : small!Class
        to t : uml!Class (
                isAbstract <- s.isAbstract,
                name <- s.name,
                ownedAttribute <- s.attributes -> collect(attribute | thisModule.smallAttributes2Property(attribute)),
                ownedOperation <- s.methods,
                superClass <- s.parents
        )
}

lazy rule RelationFrom2Property {
        from s : small!Relation
        to t : uml!Property (
                name <- s.from.name,
                lower <- s.cardinality.lowerBound,
                upper <- s.cardinality.upperBound
        )
}

lazy rule SmallClassTo2Property {
        from s : small!Class
        to t : uml!Property (
                name <- s.name,
                lower <- 1,
                upper <- 1
        )
}

rule Relation2Association {
        from s : small!Relation
        to t : uml!Association (
                name <- s.name,
                memberEnd <- Set{thisModule.RelationFrom2Property(s), thisModule.SmallClassTo2Property(s.to)}
        )
}

-- Type transformation.

lazy rule EnumerationElement2Property {
        from s : small!EnumerationElement
        to t : uml!Property (
                name <- s.value
        )
}

rule SmallEnumeration2Enumeration {
        from s : small!Enumeration
        to t : uml!Enumeration (
                name <- s.name,
                ownedAttribute <- s.value -> collect(element | thisModule.EnumerationElement2Property(element))
        )
}

rule ConcreteType2PrimitiveType {
        from s : small!ConcreteType
        to t : uml!PrimitiveType (
                name <- s.name
        )
}
```