ATL Transformation Examples

The UML to MOF
ATL transformation
- version 0.1 -

September 2005

by
ATLAS group
LINA & INRIA
Nantes
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>The UML to MOF ATL transformation</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Transformation overview</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Metamodels</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Rules specification</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>ATL code</td>
<td></td>
</tr>
<tr>
<td>2.4.1</td>
<td>Helpers</td>
<td></td>
</tr>
<tr>
<td>2.4.2</td>
<td>Rules</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>References</td>
<td></td>
</tr>
<tr>
<td>Appendix A</td>
<td>A simplified UML Core metamodel in KM3 format</td>
<td></td>
</tr>
<tr>
<td>Appendix B</td>
<td>A simplified MOF metamodel in KM3 format</td>
<td></td>
</tr>
<tr>
<td>Appendix C</td>
<td>The UML to MOF ATL code</td>
<td></td>
</tr>
</tbody>
</table>
1 Introduction

The MOF (Meta Object Facility) [3] is an OMG standard enabling to describe metamodels through common semantics. The UML (Unified Modelling Language) Core standard [4] is the OMG common modelling language. Although, the MOF is primarily designed for metamodel definitions and UML Core for the design of models, the two standards handle very close notions. This document describes a transformation enabling to pass from the UML to the MOF semantics. The transformation is based on the UML Profile for MOF OMG specification [1]. Note that a similar UML Profile (for MOF) has been described in the scope of the NetBeans project [2].

2 The UML to MOF ATL transformation

2.1 Transformation overview

The UML to MOF transformation simply transforms a UML model into a MOF model. In the scope of this transformation, we consider the input UML model has been produced by the Poseidon UML tool [5].

2.2 Metamodels

The UML to MOF transformation is based on some subsets of the UML Core and the MOF metamodels. The exhaustive definition of these metamodels can be found in the OMG UML 1.5 specification [3] and OMG MOF 1.4 specification [4]. Appendix A and Appendix B respectively provide, expressed in the KM3 format [6], the UML and MOF metamodels that have been considered in the scope of this transformation.

2.3 Rules specification

The set of rules used to transform a UML model into a MOF model has been derived from the OMG UML Profile for MOF specification [1]:

- A MOF Package is generated from a UML Package;
- A MOF Constraint is generated from a UML Constraint;
- A MOF Constraint is generated from a UML Comment which is associated with the “constraint” stereotype (note that this stereotype does not belong to the UML Profile for MOF defined by the OMG, but has been introduced to ease the definition of constraints under Poseidon);
- A MOF Class is generated from a UML Class whose namespace is associated with the “metamodel” stereotype;
- A MOF Attribute is generated from a UML Attribute;
- A MOF Parameter is generated from a UML Parameter;
- A MOF Operation is generated from a UML Operation;
- A MOF Association is generated from a UML Association;
- A MOF AssociationEnd, and its MOF Reference if the association end is navigable, is generated from a UML AssociationEnd;
- A MOF Tag is generated from a UML TaggedValue;
• A MOF Import is generated from a UML Dependency;
• A MOF PrimitiveType is generated from a UML DataType.

2.4 ATL code

The ATL code for the UML to MOF transformation is provided in Appendix C. It consists of 7 helpers and 12 rules.

2.4.1 Helpers

The getVisibility() and getMOFVisibility() helpers aim to translate a UML VisibilityKind data (vk_public / vk_private / vk_protected) into a MOF VisibilityKind one (public_vis / private_vis / protected_vis). The getVisibility() helper returns the MOF visibility that corresponds to the UML visibility passed as a parameter. The getMOFVisibility() checks whether the visibility of its contextual model element is defined. If not, it returns the public_vis default value. Otherwise, it returns the value provided by the call of the getVisibility() helper.

The getMOFScope() helper aims to translate a UML ScopeKind (sk_instance / sk_classifier) into a MOF ScopeKind (instance_level / classifier_level). For this purpose, it returns the MOF value that corresponds to the UML value.

The getIsChangeable() and getMOFIsChangeable() helpers aim to translate a UML ChangeableKind data (ck_changeable / ck_frozen / ck_addOnly) into a boolean value encoding the MOF changeability. The getIsChangeable() helper returns the boolean value that corresponds to the UML changeability of its contextual model element (true for ck_changeable, false otherwise). The getMOFVisibility() checks whether the changeability of its contextual model element is defined. If not, it returns the true boolean default value. Otherwise, it returns the value provided by the call of the getIsChangeable() helper.

The getMultiplicity() and getMOFMultiplicity() helpers aim to produce a MOF multiplicity from a UML multiplicity and a UML ordering values. The MOF represents multiplicity by means of the Multiplicity entity that encodes the lower and upper bound values, as well as the isOrdered and isUnique characteristics. UML defines two distinct attributes for multiplicity and ordering where 1) the multiplicity contains a sequence of multiplicity range (e.g. a lower and an upper bound) and 2) the ordering is encoded by a constant (ok_unordered / ok_ordered). The getMultiplicity() helper returns a tuple encoding a MOF Multiplicity based on the UML multiplicity, UML ordering and the isUnique boolean value parameters.

The getMOFMultiplicity() first checks whether the multiplicity of its contextual structural feature is defined. If not, it returns a default tuple with lower and upper attributes set to 1, and isOrdered and isUnique attributes set to true. In case the multiplicity is defined, the helper tests whether the ordering attribute of its contextual structural feature is defined. If yes, it returns the tuple value provided by the getMultiplicity() helper called with the UML multiplicity, the UML ordering, and the false constant. If the ordering property is undefined, the helper returns the value provided the getMultiplicity() helper called with the UML multiplicity and the ok_ordered and false constants.

2.4.2 Rules

The Package rule generates a MOF Package from each UML Package that has at least one stereotype named “metamodel”. The container of the generated MOF Package corresponds to the MOF entity generated for the namespace of the input UML Package. Its contents correspond to the elements generated for the ownedElements of the UML Package. Its visibility is computed by the getMOFVisibility() helper. Finally, its supertypes correspond to the entities that are generated from the parents of the generalization of the input Package.
The **Constraint** rule generates a MOF Constraint for each UML Constraint. The container of the generated MOF Constraint corresponds to the MOF entity generated for the namespace of the input UML Constraint. The values of its expression and language attributes are respectively copied from the body and language attribute of the body property of the input UML Constraint.

The **Comment** rule generates a MOF Constraint for each UML Comment which is associated with a “constraint” stereotype. The container of the generated MOF Constraint corresponds to the MOF entity generated for the namespace of the input UML Comment. The value of its expression attribute is initialized with the name of the input Comment, whereas its language attribute is set to the “OCL” default value.

The **Class** rule generates a MOF Class for each UML Class whose namespace is associated with the “metamodel” stereotype. The container of the generated MOF Class corresponds to the MOF entity generated for the namespace of the input UML Class. Its contents correspond to the elements generated for the ownedElements of the UML Class. Its visibility is computed by the getMOFVisibility() helper and its supertypes correspond to the entities that are generated from the parents of the generalization of the input Package. Finally, the isSingleton attribute is set to the false default value since it no corresponding attribute is encoded by the UML Class.

The **Attribute** rule generates a MOF Attribute for each UML Attribute. The container of the generated MOF Attribute corresponds to the MOF entity generated for the owner of the input UML Attribute. Its scope, visibility, multiplicity and isChangeable attributes are respectively computed by the getMOFScope(), getMOFVisibility(), getMOFMultiplicity() and getMOFIsChangeable() helpers. Its isDerived attribute is set to the false default value since it no corresponding attribute is encoded by the UML Attribute.

The **Parameter** rule generates a MOF Parameter for each UML Parameter. The container of the generated MOF Parameter corresponds to the MOF entity generated for the namespace of the input UML Parameter. The value of its direction attribute (in_dir/inout_dir/out_dir/return_dir) is translated from the one of the input UML direction (pdk_in/pdk_inout/pdk_out/pdk_return).

The **Operation** rule generates a MOF Operation for each UML Operation. The container of the generated MOF Operation corresponds to the MOF entity generated for the owner of the input UML Operation. Its contents correspond to the elements generated for the parameter of the UML Operation. Its scope and visibility attributes are respectively computed by the getMOFScope() and getMOFVisibility() helpers. Note that the MOF exceptions, which are not represented in UML, are initialized with an empty set.

The **Association** rule generates a MOF Association for each UML Association. The container of the generated MOF Association corresponds to the MOF entity generated for the namespace of the input UML Association. Its contents correspond to the elements generated for the connections of the UML Association. Its visibility is computed by the getMOFVisibility() helper and its supertypes correspond to the entities that are generated from the parents of the generalization of the input Association.

The **AssociationEnd** rule generates a MOF AssociationEnd, possibly with a MOF Reference, from a UML AssociationEnd. The container of the generated MOF Association corresponds to the MOF entity generated for the association of the input UML Association. Its type corresponds to the participant of the input UML AssociationEnd. The value of its aggregation attribute (shared/composite/none) is translated from the one of the UML aggregation (ak_aggregate/ak_composite/ak_none). Its visibility is computed by a call of the getVisibility() helper with the UML multiplicity, the UML ordering and the true constant as parameters, whereas its isChangeable attribute is provided by the getMOFIsChangeable() helper.

The MOF Reference is only generated for navigable UML AssociationEnd. We assume in this transformation that an Association is always composed of two and only two AssociationEnds. The
container of the generated Reference therefore corresponds to the MOF entity generated for the other AssociationEnd of the Association the input AssociationEnd belongs to.

The TaggedValue rule generates a MOF Tag for each UML TaggedValue whose type is named neither “element.uuid” nor “isValid”. The container of the generated MOF Association corresponds to the MOF entity generated for the namespace of the input UML Association. The tagId of the generated Tag is initialized with the name of the type of the input UML TaggedValue. The model elements associated with the MOF Tag correspond to a sequence containing the only pointed model element of the input UML TaggedValue.

The Dependency rule generates a MOF Import for each UML Dependency that has either an “import” or a “clustering” stereotype. The name of the generated Import corresponds to the name of the imported element (which corresponds to the first client of the input Dependency). Its container corresponds to the importer, that is the first supplier of the input Dependency. The visibility of the generated Import is set to the public_vis default value. Its isClustered property is set to true if the Dependency is associated with the “clustered” stereotype, to false otherwise (i.e. in case it is associated with the “import” stereotype).

The DataType rule generates a MOF PrimitiveType for each UML DataType. The container of the generated MOF PrimitiveType corresponds to the MOF entity generated for the namespace of the input UML DataType. Its contents correspond to the elements generated for the ownedElements of the UML DataType. Since a UML DataType does not have a visibility, the visibility of the generated PrimitiveType is set to the public_vis default value. Finally, the supertypes of the PrimitiveType correspond to the entities that are generated from the parents of the generalization of the input DataType.

3 References
Appendix A  A simplified UML Core metamodel in KM3 format

```java
package Core {

    abstract class Element {
        
        abstract class ModelElement extends Element {
            reference taggedValue[*] container : TaggedValue oppositeOf modelElement;
            reference clientDependency[*] : Dependency oppositeOf client;
            reference constraint[*] : Constraint oppositeOf constrainedElement;
            reference stereotype[*] : Stereotype;
            reference comment[*] : Comment oppositeOf annotatedElement;
            reference sourceFlow[*] : Flow oppositeOf source;
            reference targetFlow[*] : Flow oppositeOf target;
            reference templateParameter[*] ordered container : TemplateParameter oppositeOf template;
            reference namespace[0-1] : Namespace oppositeOf ownedElement;
            attribute name[0-1] : String;
            attribute visibility[0-1] : VisibilityKind;
            attribute isSpecification : Boolean;
        }

        abstract class GeneralizableElement extends ModelElement {
            reference generalization[*] : Generalization oppositeOf child;
            attribute isRoot : Boolean;
            attribute isLeaf : Boolean;
            attribute isAbstract : Boolean;
        }

        abstract class Namespace extends ModelElement {
            reference ownedElement[*] container : ModelElement oppositeOf namespace;
        }

        abstract class Classifier extends GeneralizableElement, Namespace {
            reference powertypeRange[*] : Generalization oppositeOf powertype;
            reference feature[*] ordered container : Feature oppositeOf owner;
        }

        class Class extends Classifier {
            attribute isActive : Boolean;
        }

        class DataType extends Classifier {
        }

        abstract class Feature extends ModelElement {
            reference owner[0-1] : Classifier oppositeOf feature;
            attribute ownerScope : ScopeKind;
        }

        abstract class StructuralFeature extends Feature {
            attribute type : Classifier;
            attribute multiplicity[0-1] : Multiplicity;
            attribute changeability[0-1] : ChangeableKind;
            attribute targetScope[0-1] : ScopeKind;
            attribute ordering[0-1] : OrderingKind;
        }

        class AssociationEnd extends ModelElement {
            
```
class Interface extends Classifier {
    
    class Constraint extends ModelElement {
        reference constrainedElement[*] ordered : ModelElement oppositeOf constraint;
        attribute body[0-1] : BooleanExpression;
    }
}

abstract class Relationship extends ModelElement {
    
    class Association extends GeneralizableElement, Relationship {
        reference connection[2-*] ordered container : AssociationEnd oppositeOf
            association;
    }
}

abstract class BehavioralFeature extends Feature {
    reference parameter[*] ordered container : Parameter oppositeOf
        behavioralFeature;
    attribute isQuery : Boolean;
}

class Operation extends BehavioralFeature {
    attribute concurrency[0-1] : CallConcurrencyKind;
    attribute defaultValue[0-1] : Expression;
    attribute kind : ParameterDirectionKind;
}

class Parameter extends ModelElement {
    reference type : Classifier;
    reference behavioralFeature[0-1] : BehavioralFeature oppositeOf parameter;
    attribute defaultValue[0-1] : Expression;
    attribute kind : ParameterDirectionKind;
}

abstract class Method extends BehavioralFeature {
    reference specification : Operation;
    
    class Generalization extends Relationship {
        reference parent : GeneralizableElement;
        reference powertype[0-1] : Classifier oppositeOf powertypeRange;
        reference child : GeneralizableElement oppositeOf generalization;
        attribute discriminator[0-1] : String;
    }
}

class AssociationClass extends Association, Class {
class Dependency extends Relationship {
    reference client[1-*] : ModelElement oppositeOf clientDependency;
    reference supplier[1-*] : ModelElement;
}

class Abstraction extends Dependency {
    attribute mapping[0-1] : MappingExpression;
}

abstract class PresentationElement extends Element {
    reference subject[*] : ModelElement;
}

class Usage extends Dependency {
}

class Binding extends Dependency {
    reference argument[1-*] ordered container : TemplateArgument oppositeOf binding;
}

class Component extends Classifier {
    reference deploymentLocation[*] : Node oppositeOf deployedComponent;
    reference residentElement[*] container : ElementResidence oppositeOf "container";
    reference implementation[*] : Artifact;
}

class Node extends Classifier {
    reference deployedComponent[*] : Component oppositeOf deploymentLocation;
}

class Permission extends Dependency {
    reference annotatedElement[*] : ModelElement oppositeOf comment;
    attribute body : String;
}

class Flow extends Relationship {
    reference source[*] : ModelElement oppositeOf sourceFlow;
    reference target[*] : ModelElement oppositeOf targetFlow;
}

class ElementResidence {
    reference "container" : Component oppositeOf residentElement;
    reference resident : ModelElement;
    attribute visibility[0-1] : VisibilityKind;
}

class TemplateParameter {
    reference template : ModelElement oppositeOf templateParameter;
    reference container : ModelElement;
    reference defaultElement[0-1] : ModelElement;
}

class Primitive extends DataType {
}

class Enumeration extends DataType {
    reference "literal"[1-*] ordered container : EnumerationLiteral oppositeOf "enumeration";
}

class EnumerationLiteral extends ModelElement {
    reference "enumeration" : Enumeration oppositeOf "literal";
}
class Stereotype extends GeneralizableElement {
   reference stereotypeConstraint[*] container : Constraint;
   reference definedTag[*] container : TagDefinition oppositeOf owner;
   attribute icon[0-1] : String;
   attribute baseClass[1-*] : String;
}

class TagDefinition extends ModelElement {
   reference owner[0-1] : Stereotype oppositeOf definedTag;
   attribute tagType[0-1] : String;
   attribute multiplicity[0-1] : Multiplicity;
}

class TaggedValue extends ModelElement {
   reference type : TagDefinition;
   reference referenceValue[*] : ModelElement;
   reference modelElement : ModelElement oppositeOf taggedValue;
   attribute dataValue[*] : String;
}

class ProgrammingLanguageDataType extends DataType {
   attribute expression : TypeExpression;
}

class Artifact extends Classifier {
}

class TemplateArgument {
   reference binding : Binding oppositeOf argument;
   reference modelElement : ModelElement;
}
Appendix B  A simplified MOF metamodel in KM3 format

package Model {
  abstract class ModelElement {
    -- derived
    reference requiredElements[*] : ModelElement;
    reference constraints[*] : Constraint oppositeOf constrainedElements;
    reference "container"[0-1] : Namespace oppositeOf contents;
    attribute name : String;
  -- derived
    attribute qualifiedName[1-*] ordered : String;
    attribute annotation : String;
  operation findRequiredElements(kinds : String, recursive : Boolean) :
      ModelElement;
  operation isRequiredBecause(otherElement : ModelElement, reason : String) :
      Boolean;
  operation isFrozen() : Boolean;
  operation isVisible(otherElement : ModelElement) : Boolean;
}

enumeration VisibilityKind {
  literal public_vis;
  literal protected_vis;
  literal private_vis;
}

abstract class Namespace extends ModelElement {
  reference contents[*] ordered container : ModelElement oppositeOf "container";
  operation lookupElement(name : String) : ModelElement;
  operation resolveQualifiedName(qualifiedName : String) : ModelElement;
  operation findElementsByType(ofType : Class, includeSubtypes : Boolean) :
      ModelElement;
  operation nameIsValid(proposedName : String) : Boolean;
}

abstract class GeneralizableElement extends Namespace {
  reference supertypes[*] ordered : GeneralizableElement;
  attribute isRoot : Boolean;
  attribute isLeaf : Boolean;
  attribute isAbstract : Boolean;
  attribute visibility : VisibilityKind;
  operation allSupertypes() : GeneralizableElement;
  operation lookupElementExtended(name : String) : ModelElement;
  operation findElementsByTypeExtended(ofType : Class, includeSubtypes : Boolean) :
      ModelElement;
}

abstract class TypedElement extends ModelElement {
  reference type : Classifier;
}

abstract class Classifier extends GeneralizableElement {
}

class Class extends Classifier {
  attribute isSingleton : Boolean;
}

class MultiplicityType {
attribute lower : Integer;
attribute upper : Integer;
attribute isOrdered : Boolean;
attribute isUnique : Boolean;
)

abstract class DataType extends Classifier {
}

class PrimitiveType extends DataType {
}

class EnumerationType extends DataType {
    attribute labels[1-*] ordered : String;
}

class CollectionType extends DataType, TypedElement {
    attribute multiplicity : MultiplicityType;
}

class StructureType extends DataType {
}

class StructureField extends TypedElement {
}

class AliasType extends DataType, TypedElement {
}

enumeration ScopeKind {
    literal instance_level;
    literal classifier_level;
}

abstract class Feature extends ModelElement {
    attribute scope : ScopeKind;
    attribute visibility : VisibilityKind;
}

abstract class StructuralFeature extends Feature, TypedElement {
    attribute multiplicity : MultiplicityType;
    attribute isChangeable : Boolean;
}

class Attribute extends StructuralFeature {
    attribute isDerived : Boolean;
}

class Reference extends StructuralFeature {
    reference referencedEnd : AssociationEnd;
    reference exposedEnd : AssociationEnd;
}

abstract class BehavioralFeature extends Feature, Namespace {
}

class Operation extends BehavioralFeature {
    reference exceptions[*] ordered : Exception;
    attribute isQuery : Boolean;
}

class Exception extends BehavioralFeature {
class Association extends Classifier {
    attribute isDerived : Boolean;
}

equation AggregationKind {
    literal none;
    literal shared;
    literal composite;
}

class AssociationEnd extends TypedElement {
    attribute isNavigable : Boolean;
    attribute aggregation : AggregationKind;
    attribute multiplicity : MultiplicityType;
    attribute isChangeable : Boolean;
    operation otherEnd() : AssociationEnd;
}

class Package extends GeneralizableElement {
}

class Import extends ModelElement {
    reference importedNamespace : Namespace;
    attribute visibility : VisibilityKind;
    attribute isClustered : Boolean;
}

equation DirectionKind {
    literal in_dir;
    literal out_dir;
    literal inout_dir;
    literal return_dir;
}

class Parameter extends TypedElement {
    attribute direction : DirectionKind;
    attribute multiplicity : MultiplicityType;
}

class Constraint extends ModelElement {
    reference constrainedElements[1-*] : ModelElement oppositeOf constraints;
    attribute expression : String;
    attribute language : String;
    attribute evaluationPolicy : EvaluationKind;
}

equation EvaluationKind {
    literal immediate;
    literal deferred;
}

class Tag extends ModelElement {
    reference elements[1-*] : ModelElement;
    attribute tagId : String;
    attribute values[*] ordered : String;
}
Appendix C  The UML to MOF ATL code

```atl
module UML2MOF;
create OUT : MOF from IN : UML;

uses strings;

--- HELPERS ---------------------------------------- ----------------------------

-- This helper computes a MOF!VisibilityKind from a UML!VisibilityKind.
-- CONTEXT: thisModule
-- IN:  UML!VisibilityKind
-- RETURN: MOF!VisibilityKind
helper def: getVisibility(v : UML!VisibilityKind) : MOF!VisibilityKind =
  if v = #vk_public
    #public_vis
  else
    if v = #vk_private
      #private_vis
    else
      if v = #vk_protected
        #protected_vis
      else
        #public_vis -- default
      endif
    endif
  endif;

-- This helper computes the MOF!VisibilityKind that corresponds to the
-- UML!VisibilityKind of the contextual UML!ModelElement. If this visibility
-- kind is undefined, the helper returns 'public_vis' as a default value.
-- CONTEXT: UML!ModelElement
-- RETURN: MOF!VisibilityKind
helper context UML!ModelElement def: getMOFVisibility() : MOF!VisibilityKind =
  let v : UML!VisibilityKind = self.visibility in
  if not v.oclIsUndefined()
    thisModule.getVisibility(v)
  else
    #public_vis
  endif;

-- This helper computes the MOF!ScopeKind that corresponds to the
-- UML!ScopeKind of the contextual UML!Feature.
-- CONTEXT: UML!Feature
-- RETURN: MOF!ScopeKind
helper context UML!Feature def: getMOFScope() : MOF!ScopeKind =
  if self.ownerScope = #sk_instance
    #instance_level
  else
    #classifier_level
  endif;

-- This helper computes the MOF isChangeable boolean attribute that corresponds
-- to the UML!Changeability of the contextual UML!ModelElement (whose
-- changeability is not undefined).
-- CONTEXT: UML!ModelElement
```
64 -- RETURN: Boolean
65 helper context UML!ModelElement def: getIsChangeable() : Boolean =
66 (self.changeability = #ck_changeable);
67
68 -- This helper computes the MOF isChangeable boolean attribute that corresponds
69 -- to the UML!Changeability of the contextual UML!ModelElement. If this
70 -- changeability is undefined, the helper returns true as a default value.
71 -- CONTEXT: UML!ModelElement
72 -- RETURN: Boolean
73 helper context UML!ModelElement def: getMOFIsChangeable() : Boolean =
74 if not self.changeability.oclIsUndefined()
75 then
76 self.getIsChangeable()
77 else
78 true
79 endif;
80
81 -- This helper computes the tuple encoding the MOF multiplicity that
82 -- corresponds to the UML!Multiplicity, UML!OrderingKind, and the isUnique
83 -- boolean provided as parameters.
84 -- CONTEXT: thisModule
85 -- IN:  UML!Multiplicity, UML!OrderingKind, Boolean
86 -- RETURN: TupleType(Integer, Integer, Boolean, Boolean)
87 helper def: getMultiplicity(m : UML!Multiplicity,  
88 o : UML!OrderingKind,  
89 isUnique : Boolean) :
90 TupleType(lower : Integer,  
91 upper : Integer,  
92 isOrdered : Boolean,  
93 isUnique : Boolean) =
94 Tuple{
95 lower = m.range->asSequence()->first().lower,  
96 upper = m.range->asSequence()->first().upper,  
97 isOrdered = (o = 'ok_ordered'),  
98 isUnique = isUnique  
99 };
100
101 -- This helper computes the tuple encoding the MOF multiplicity that
102 -- corresponds to the UML!Multiplicity of the contextual UML!StructuralFeature.
103 -- If the multiplicity of the contextual structural feature is undefined, the
104 -- helper returns (1,1,true,true) as a default tuple. Otherwise, it returns the
105 -- tuple computed by the getMultiplicity helper. Note that if the ordering of
106 -- the contextual structural feature is undefined, it is considered as ordered.
107 -- CONTEXT: UML!StructuralFeature
108 -- RETURN: TupleType(Integer, Integer, Boolean, Boolean)
109 helper context UML!StructuralFeature def: getMOFMultiplicity() :  
110 TupleType(lower : Integer, upper : Integer,  
111 isOrdered : Boolean, isUnique : Boolean) =
112 Boolean) =
113 if not self.multiplicity.oclIsUndefined()  
114 then  
115 if not self.ordering.oclIsUndefined()  
116 then
117 thisModule.getMultiplicity(self.multiplicity, self.ordering, false)
118 else
119 thisModule.getMultiplicity(self.multiplicity, 'ok_ordered', false)
120 endif  
121 else
122 Tuple(lower = 1, upper = 1, isOrdered = true, isUnique = true)
123 endif;
124
125 -- Helper ...
126 -- CONTEXT: UML!ModelElement
127 -- RETURN: String
128 helper context UML!ModelElement def: getMOPQualifiedName() : String =
129 self.name;
130
131 -- --------------------------------------------------------------
-- RULES
---

133 -- Rule 'Package'
134 -- This rule generates a MOF package from each UML package that has a
135 -- stereotype named 'metamodel'.
136 -- Supertypes of the generated package correspond to the parent of the
137 -- generalization of the input UML package.

141 rule Package {
142   from
143   up : UML!Package (up.stereotype->exists(e | e.name = 'metamodel'))
144   to
145   mp : MOF!Package
146 }

147   -- Begin bindings inherited from ModelElement
148   name <- up.name,
149   annotation <- '',
150   container <- up.namespace,
151   constraints <- up.constraint,
152   requiredElements <-
153   -- End of bindings inherited from ModelElement
154
155   -- Begin bindings inherited from Namespace
156   contents <- up.ownedElement,
157   -- End of bindings inherited from Namespace
158
159   -- Begin bindings inherited from GeneralizableElement
160   isRoot <- up.isRoot,
161   isLeaf <- up.isLeaf,
162   isAbstract <- up.isAbstract,
163   visibility <- up.getMOFVisibility(),
164   supertypes <- up.generalization->collect(e | e.parent)
165   -- End of bindings inherited from GeneralizableElement
166
167 }

169 -- Rule 'Constraint'
170 -- This rule generates a MOF constraint from a UML one. Properties of the
171 -- generated constraint, except evaluationPolicy, are copied from the input UML
172 -- constraint.
173 -- The MOF evaluationPolicy property, which has no equivalent in UML, is set to
174 -- the default 'immediate' value.

176 rule Constraint {
178   from
179   uc : UML!Constraint
180   to
181   mc : MOF!Constraint
182   -- Begin bindings inherited from ModelElement
183   name <- uc.name,
184   annotation <- '',
185   container <- uc.namespace,
186   constraints <- uc.constraint,
187   requiredElements <-
188   -- End of bindings inherited from ModelElement
189
190   expression <- uc.body.body,
191   language <- uc.body.language,
192   constrainedElements <- uc.constrainedElement,
193   evaluationPolicy <- 'immediate'
194 }

195 -- Rule 'Comment'
196 -- This rule generates a MOF constraint from each UML Comment that has a
197 -- 'constraint' stereotype.
198 -- The content of the generated constraint corresponds to the body of the input
200 -- UML comment, its language is associated with the OCL default value.

201 rule Comment {
from uc : UML!Comment {
 uc.stereotype->exists(e | e.name = 'constraint')
}
to mc : MOF!Constraint {
   -- Begin bindings inherited from ModelElement
   name <- uc.name,
   annotation <- '',
   container <- uc.namespace,
   constraints <- uc.constraint,
   -- requiredElements <-,
   -- End of bindings inherited from ModelElement
   expression <- uc.body,
   language <- 'OCL',
   constrainedElements <- uc.annotatedElement
}
-- Rule 'Class'
-- This rule generates a MOF class from each UML class whose namespace (which
-- expected to be a Package) has a 'metamodel' stereotype.
-- Properties of the generated class are copied from the input UML class
-- properties.
rule Class {
from uc : UML!Class {
 uc.namespace.stereotype->exists(e | e.name = 'metamodel')
}
to mc : MOF!Class {
   -- Begin bindings inherited from ModelElement
   name <- uc.name,
   annotation <- '',
   container <- uc.namespace,
   constraints <- uc.constraint,
   -- requiredElements <-,
   -- End of bindings inherited from ModelElement
   -- Begin bindings inherited from Namespace
   contents <- uc.ownedElement,
   -- End of bindings inherited from Namespace
   -- Begin bindings inherited from GeneralizableElement
   isRoot <- uc.isRoot,
   isLeaf <- uc.isLeaf,
   isAbstract <- uc.isAbstract,
   visibility <- uc.getMOFVisibility(),
   supertypes <- uc.generalization->collect(e | e.parent),
   -- End of bindings inherited from GeneralizableElement
   isSingleton <- false
}
-- Rule 'Attribute'
-- This rule generates a MOF attribute from each UML attribute.
-- Properties of the generated attribute are copied from the input UML
-- attribute properties. Note that the 'isDerived' attribute is set to the
-- false default value.
rule Attribute {
from ua : UML!Attribute
to ma : MOF!Attribute {
   -- Begin bindings inherited from ModelElement
   name <- ua.name,
   annotation <- '',
   -- Begin bindings inherited from GeneralizableElement
   isRoot <- ua.isRoot,
   isLeaf <- ua.isLeaf,
   isAbstract <- ua.isAbstract,
   visibility <- ua.getMOFVisibility(),
   supertypes <- ua.generalization->collect(e | e.parent),
   -- End of bindings inherited from GeneralizableElement
   isSingleton <- false
}
container <- ua.owner,
constraints <- ua.constraint,
-- End of bindings inherited from ModelElement
-- Begin bindings inherited from Feature
scope <- ua.getMOFScope(),
visibility <- ua.getMOFVisibility(),
-- End of bindings inherited from Feature
-- Begin bindings inherited from StructuralFeature
multiplicity <- ua.getMOFMultiplicity(),
isChangeable <- ua.getMOFIsChangeable(),
-- End of bindings inherited from StructuralFeature
-- Begin bindings inherited from TypedElement
type <- ua.type,
-- End of bindings inherited from TypedElement
isDerived <- false
}

-- Rule 'Parameter'
-- This rule generates a MOF parameter from each UML parameter.
-- Properties of the generated parameter are copied from the input UML
-- parameter properties. Note that the MOF multiplicity attribute is not set
-- since the corresponding information is not available in the UML metamodel.
-- The MOF multiplicity attribute, not encoded in UML, is left undefined.
rule Parameter {
  from up : UML!Parameter
to mp : MOF!Parameter {
    -- Begin bindings inherited from ModelElement
    name <- up.name,
anotation <- '',
container <- up.namespace,
constraints <- up.constraint,
-- End of bindings inherited from ModelElement
-- Begin bindings inherited from TypedElement
type <- up.type,
-- End of bindings inherited from TypedElement

direction <-
if up.kind = #pdk_in
  #in_dir
else if up.kind = #pdk_inout
  #inout_dir
else if up.kind = #pdk_out
  #out_dir
else
  #return_dir
endif

-- Rule 'Operation'
-- This rule generates a MOF operation from each UML operation.
-- Properties of the generated operation are copied from the input UML
-- operation properties. Note that the exceptions property of the generated
-- MOF operation is set to an empty set as a default value.

rule Operation {
  from
    uo : UML!Operation
  to
    mo : MOF!Operation {
      -- Begin bindings inherited from ModelElement
      name <- uo.name,
      annotation <- '',
      container <- uo.owner,
      constraints <- uo.constraint,
      -- requiredElements <-,
      -- End of bindings inherited from ModelElement
      -- Begin bindings inherited from Namespace
      contents <- uo.parameter,
      -- End of bindings inherited from Namespace
      -- Begin bindings inherited from Feature
      scope <- uo.getMOFScope(),
      visibility <- uo.getMOFVisibility(),
      -- End of bindings inherited from Feature
      isQuery <- uo.isQuery,
      exceptions <- Set{ }
    }
  }

-- Rule 'Association'
-- This rule generates a MOF association from each UML association.
-- Properties of the generated association are copied from the input UML
-- association properties. contents of the generated association correspond to
-- the MOF association end generated for the connection of the input UML
-- association.

rule Association {
  from
    ua : UML!Association
  to
    ma : MOF!Association {
      -- Begin bindings inherited from ModelElement
      name <- ua.name,
      annotation <- '',
      container <- ua.namespace,
      constraints <- ua.constraint,
      -- requiredElements <-,
      -- End of bindings inherited from ModelElement
      -- Begin bindings inherited from Namespace
      contents <- ua.connection,
      -- End of bindings inherited from Namespace
      -- Begin bindings inherited from GeneralizableElement
      isRoot <- ua.isRoot,
      isLeaf <- ua.isLeaf,
      isAbstract <- ua.isAbstract,
      visibility <- ua.getMOFVisibility(),
      supertypes <- ua.generalization->collect(e | e.parent)
      -- End of bindings inherited from GeneralizableElement
    }
  }

-- Rule 'AssociationEnd'
-- This rule generates a MOF association end, along with an optional reference,
-- The MOF reference is only generated from navigable UML association ends. For
-- this purpose, the rule iterates through a Sequence that contains 1 element
-- if UML association end is navigable, 0 otherwise.
ATL Transformation Example

UML to MOF

Date 03/11/2005

-- Properties of the generated association end are copied from the input UML
-- association end properties.
-- When generated, the reference has the same name than its associated
-- association end. Its container corresponds to the class that to which is
-- associated the other association end contained by the association that also
-- contains the input UML association end.
-- Its scope and visibility are respectively set to the 'instance_level' and
-- 'public_vis' default values. The values of its type, multiplicity and
-- isChangeable attributes are copied from the input UML association end.
-- The constraints of the generated reference are packed within a single
-- element Sequence for the purpose of compatibility with the reference
-- sequence of the 'foreach' operator.
-- Finally, ...

rule AssociationEnd {
  from
    ua : UML!AssociationEnd
  to
    ma : MOF!AssociationEnd(
      -- Begin bindings inherited from ModelElement
      name <- ua.name,
      annotation <- '',
      container <- ua.association,
      constraints <- ua.constraint,
      -- requiredElements <-,
      -- End of bindings inherited from ModelElement
      -- Begin bindings inherited from TypedElement
      type <- ua.participant,
      -- End of bindings inherited from TypedElement
      isNavigable <- ua.isNavigable,
      aggregation <-
      if ua.aggregation = #ak_aggregate
        then
          #shared
        else
          if ua.aggregation = #ak_composite
            then
              #composite
            else
              #none
          endif
      endif,
      multiplicity <-
        thisModule.getMultiplicity(ua.multiplicity, ua.ordering, true),
      isChangeable <- ua.getMOFIsChangeable()
    ),
  mr : distinct MOF!Reference
    foreach (c
      if ua.isNavigable
        then
          Sequence{true}
        else
          Sequence{}
      endif) ( 
        -- Begin bindings inherited from ModelElement
        name <- ua.name,
        annotation <- '',
        container <- ua.association.connection
        ->select(e | not (e = ua))
        ->first().participant,
        constraints <- Sequence{ua.constraint},
        requiredElements <-,
        -- End of bindings inherited from ModelElement
        -- Begin bindings inherited from Feature
        scope <- #instance_level,
        visibility <- ua.getMOFVisibility(),
        -- End of bindings inherited from Feature
    )}
-- Begin bindings inherited from StructuralFeature
478  
479  -- If the 2 following bindings are exchanged with the referencedEnd
480  -- one, an error may be raised due to MDR inconsisteny checkings.
481  
482  multiplicity <- thisModule.getMultiplicity(ua.multiplicity, ua.ordering, true),
483  
484  isChangeable <- ua.getMOFIsChangeable(),
485  
486  -- End of bindings inherited from StructuralFeature
487  
488  -- Begin bindings inherited from TypedElement
489  
490  type <- ua.participant,
491  
492  -- End of bindings inherited from TypedElement
493  
494  -- The Association corresponding to the Reference is derived: the
495  -- exposedEnd reference should not be assigned.
496  
497  exposedEnd <- ua.association.connection
498  
499  --select(e | not {e = ua})->first(),
500  
501  referencedEnd <- ma
502  
503  
504  rule TaggedValue {
505          
506          from
507          
508          ut : UML!TaggedValue {
509          (ut.type.name <> 'element.uuid') and
510          (ut.type.name <> 'isValid')
511          }
512          
513          using
514  
515          name : String = ut.type.name;
516          
517          annotation <- '',
518          
519          constraints <- ut.constraint,
520          
521          -- End of bindings inherited from ModelElement
522          
523          tagId <- name,
524          
525          values <- ut.dataValue,
526          
527          elements <- Sequence{ut.modelElement}
528          }
529  
530  rule Dependency {
531          
532          from
533          
534          ud : UML!Dependency {
535          
536          ud.stereotype ->exists(e | e.name = 'import' or e.name = 'clustering')
537          }
538          
539          using
540          
541          importer : UML!ModelElement = ud.client->asSequence()->first();
542          
543          imported : UML!ModelElement = ud.supplier->asSequence()->first();
544          
545          
546          
547          }
\[
\text{mi : MOF!Import \{} \\
\quad \text{name <- imported.name,} \\
\quad \text{annotation <- '',} \\
\quad \text{container <- importer,} \\
\quad \text{constraints <- ud.constraint,} \\
\quad \text{-- requiredElements <-,} \\
\quad \text{-- End of bindings inherited from ModelElement} \\
\quad \text{visibility <- \#public_vis,} \\
\quad \text{isClustered <-} \\
\quad \quad \text{if ud.stereotype->exists(e | e.name = 'import')} \\
\quad \quad \text{then} \\
\quad \quad \quad \text{false} \\
\quad \quad \text{else} \\
\quad \quad \quad \text{true} \\
\quad \quad \text{endif,} \\
\quad \quad \text{importedNamespace <- imported} \\
\quad \} \\
\]

\[
\text{-- Rule 'DataType'} \\
\text{-- This rule generates a MOF datatype from each UML datatype.} \\
\text{-- Properties of the generated datatype are copied from the input UML datatype} \\
\text{-- properties. Note that the visibility of the generated MOF datatype is set to} \\
\text{-- the 'public_vis' default value.} \\
\text{rule DataType\{} \\
\quad \text{from} \\
\quad \quad \text{ud : UML!DataType} \\
\quad \text{to} \\
\quad \quad \text{md : MOF!PrimitiveType \{} \\
\quad \quad \quad \text{name <- ud.name,} \\
\quad \quad \quad \text{annotation <- '',} \\
\quad \quad \quad \text{container <- ud.namespace,} \\
\quad \quad \quad \text{contents <- ud.ownedElement,} \\
\quad \quad \quad \text{-- End of bindings inherited from Namespace} \\
\quad \quad \text{ visibility <- \#public_vis,} \\
\quad \quad \text{supertypes <- ud.generalization->collect(e | e.parent)\}} \\
\quad \} \\
\]

547
to
548  mi : MOF!Import {
549  -- Begin bindings inherited from ModelElement
550  name <- imported.name,
551  annotation <- '',
552  container <- importer,
553  constraints <- ud.constraint,
554  -- requiredElements <-,
555  -- End of bindings inherited from ModelElement
556
557  visibility <- \#public_vis,
558  isClustered <-
559  \quad \text{if ud.stereotype->exists(e | e.name = 'import')}
560  \quad \text{then}
561  \quad \quad \text{false}
562  \quad \text{else}
563  \quad \quad \text{true}
564  \quad \text{endif,}
565  importedNamespace <- imported
566  }
567
568
569
570  -- Rule 'DataType'
571  -- This rule generates a MOF datatype from each UML datatype.
572  -- Properties of the generated datatype are copied from the input UML datatype
573  -- properties. Note that the visibility of the generated MOF datatype is set to
574  -- the 'public_vis' default value.
575  rule DataType{
576    from
577      ud : UML!DataType
578    to
579      md : MOF!PrimitiveType {
580      -- Begin bindings inherited from ModelElement
581      name <- ud.name,
582      annotation <- '',
583      container <- ud.namespace,
584      constraints <- ud.constraint,
585      -- requiredElements <-,
586      -- End of bindings inherited from ModelElement
587
588      -- Begin bindings inherited from Namespace
589      contents <- ud.ownedElement,
590      -- End of bindings inherited from Namespace
591
592      -- Begin bindings inherited from GeneralizableElement
593      isRoot <- ud.isRoot,
594      isLeaf <- ud.isLeaf,
595      isAbstract <- ud.isAbstract,
596      visibility <- \#public_vis,
597      supertypes <- ud.generalization->collect(e | e.parent)
598      -- End of bindings inherited from GeneralizableElement
599    }
600  }