

ATL Transformation Examples

The UML to MOF ATL transformation *- version 0.1 -*

September 2005

by
ATLAS group
LINA & INRIA
Nantes

Content

1	Introduction	2
2	The UML to MOF ATL transformation.....	2
2.1	<i>Transformation overview.....</i>	2
2.2	<i>Metamodels</i>	2
2.3	<i>Rules specification</i>	2
2.4	<i>ATL code</i>	2
2.4.1	Helpers	2
2.4.2	Rules.....	2
3	References	2
Appendix A	A simplified UML Core metamodel in KM3 format	2
Appendix B	A simplified MOF metamodel in KM3 format	2
Appendix C	The UML to MOF ATL code.....	2



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;

	ATL Transformation Example	
	UML to MOF	Date 03/11/2005

- 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 `getMOFIsChangeable()` 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.



ATL Transformation Example

UML to MOF

Date 03/11/2005

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

	ATL Transformation Example	
	UML to MOF	Date 03/11/2005

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

- [1] OMG/UML Profile for MOF, OMG Formal Specification. formal/04-02-06, 2004. Available at <http://www.omg.org/docs/formal/04-02-06.pdf>.
- [2] NetBeans/Sun Microsystems. UML Profile for MOF. Available at <http://mdr.netbeans.org/uml2mof/profile.html>.
- [3] OMF/UML (Unified Modeling Language) 1.5 specification. formal/03-03-01, 2003.
- [4] OMG/MOF Meta Object Facility (MOF) 1.4 specification. formal/2002-04-03, 2002.
- [5] Gentleware. Poseidon for UML, information and download available at <http://www.gentleware.com/index.php>.
- [6] KM3 User Manual. The Eclipse Generative Model Transformer (GMT) project, <http://eclipse.org/gmt/>.

Appendix A A simplified UML Core metamodel in KM3 format

```
1 package Core {
2     abstract class Element {
3     }
4
5     abstract class ModelElement extends Element {
6         reference taggedValue[*] container : TaggedValue oppositeOf modelElement;
7         reference clientDependency[*] : Dependency oppositeOf client;
8         reference constraint[*] : Constraint oppositeOf constrainedElement;
9         reference stereotype[*] : Stereotype;
10        reference comment[*] : Comment oppositeOf annotatedElement;
11        reference sourceFlow[*] : Flow oppositeOf source;
12        reference targetFlow[*] : Flow oppositeOf target;
13        reference templateParameter[*] ordered container : TemplateParameter oppositeOf
14    template;
15        reference namespace[0-1] : Namespace oppositeOf ownedElement;
16        attribute name[0-1] : String;
17        attribute visibility[0-1] : VisibilityKind;
18        attribute isSpecification : Boolean;
19    }
20
21
22    abstract class GeneralizableElement extends ModelElement {
23        reference generalization[*] : Generalization oppositeOf child;
24        attribute isRoot : Boolean;
25        attribute isLeaf : Boolean;
26        attribute isAbstract : Boolean;
27    }
28
29
30    abstract class Namespace extends ModelElement {
31        reference ownedElement[*] container : ModelElement oppositeOf namespace;
32    }
33
34    abstract class Classifier extends GeneralizableElement, Namespace {
35        reference powertypeRange[*] : Generalization oppositeOf powertype;
36        reference feature[*] ordered container : Feature oppositeOf owner;
37    }
38
39    class Class extends Classifier {
40        attribute isActive : Boolean;
41    }
42
43    class DataType extends Classifier {
44    }
45
46    abstract class Feature extends ModelElement {
47        reference owner[0-1] : Classifier oppositeOf feature;
48        attribute ownerScope : ScopeKind;
49    }
50
51    abstract class StructuralFeature extends Feature {
52        reference type : Classifier;
53        attribute multiplicity[0-1] : Multiplicity;
54        attribute changeability[0-1] : ChangeableKind;
55        attribute targetScope[0-1] : ScopeKind;
56        attribute ordering[0-1] : OrderingKind;
57    }
58
59    class AssociationEnd extends ModelElement {
```

```
60         reference association : Association oppositeOf connection;
61         reference specification[*] : Classifier;
62         reference participant : Classifier;
63         reference qualifier[*] ordered container : Attribute oppositeOf associationEnd;
64         attribute isNavigable : Boolean;
65         attribute ordering[0-1] : OrderingKind;
66         attribute aggregation[0-1] : AggregationKind;
67         attribute targetScope[0-1] : ScopeKind;
68         attribute multiplicity[0-1] : Multiplicity;
69         attribute changeability[0-1] : ChangeableKind;
70     }
71
72     class Interface extends Classifier {
73     }
74
75     class Constraint extends ModelElement {
76         reference constrainedElement[*] ordered : ModelElement oppositeOf constraint;
77         attribute body[0-1] : BooleanExpression;
78     }
79
80     abstract class Relationship extends ModelElement {
81     }
82
83     class Association extends GeneralizableElement, Relationship {
84         reference connection[2-*] ordered container : AssociationEnd oppositeOf
85 association;
86     }
87
88     class Attribute extends StructuralFeature {
89         reference associationEnd[0-1] : AssociationEnd oppositeOf qualifier;
90         attribute initialValue[0-1] : Expression;
91     }
92
93     abstract class BehavioralFeature extends Feature {
94         reference parameter[*] ordered container : Parameter oppositeOf
95 behavioralFeature;
96         attribute isQuery : Boolean;
97     }
98
99     class Operation extends BehavioralFeature {
100         attribute concurrency[0-1] : CallConcurrencyKind;
101         attribute isRoot : Boolean;
102         attribute isLeaf : Boolean;
103         attribute isAbstract : Boolean;
104         attribute specification[0-1] : String;
105     }
106
107     class Parameter extends ModelElement {
108         reference type : Classifier;
109         reference behavioralFeature[0-1] : BehavioralFeature oppositeOf parameter;
110         attribute defaultValue[0-1] : Expression;
111         attribute kind : ParameterDirectionKind;
112     }
113
114     class Method extends BehavioralFeature {
115         reference specification : Operation;
116         attribute body : ProcedureExpression;
117     }
118
119     class Generalization extends Relationship {
120         reference parent : GeneralizableElement;
121         reference powertype[0-1] : Classifier oppositeOf powertypeRange;
122         reference child : GeneralizableElement oppositeOf generalization;
123         attribute discriminator[0-1] : String;
124     }
125
126     class AssociationClass extends Association, Class {
127     }
128
```




```
129     class Dependency extends Relationship {
130         reference client[1-*] : ModelElement oppositeOf clientDependency;
131         reference supplier[1-*] : ModelElement;
132     }
133
134     class Abstraction extends Dependency {
135         attribute mapping[0-1] : MappingExpression;
136     }
137
138     abstract class PresentationElement extends Element {
139         reference subject[*] : ModelElement;
140     }
141
142     class Usage extends Dependency {
143     }
144
145     class Binding extends Dependency {
146         reference argument[1-*] ordered container : TemplateArgument oppositeOf
147 binding;
148     }
149
150     class Component extends Classifier {
151         reference deploymentLocation[*] : Node oppositeOf deployedComponent;
152         reference residentElement[*] container : ElementResidence oppositeOf
153 "container";
154         reference implementation[*] : Artifact;
155     }
156
157     class Node extends Classifier {
158         reference deployedComponent[*] : Component oppositeOf deploymentLocation;
159     }
160
161     class Permission extends Dependency {
162     }
163
164     class Comment extends ModelElement {
165         reference annotatedElement[*] : ModelElement oppositeOf comment;
166         attribute body : String;
167     }
168
169     class Flow extends Relationship {
170         reference source[*] : ModelElement oppositeOf sourceFlow;
171         reference target[*] : ModelElement oppositeOf targetFlow;
172     }
173
174     class ElementResidence {
175         reference "container" : Component oppositeOf residentElement;
176         reference resident : ModelElement;
177         attribute visibility[0-1] : VisibilityKind;
178     }
179
180     class TemplateParameter {
181         reference template : ModelElement oppositeOf templateParameter;
182         reference parameter container : ModelElement;
183         reference defaultElement[0-1] : ModelElement;
184     }
185
186     class Primitive extends DataType {
187     }
188
189     class Enumeration extends DataType {
190         reference "literal"[1-*] ordered container : EnumerationLiteral oppositeOf
191 "enumeration";
192     }
193
194     class EnumerationLiteral extends ModelElement {
195         reference "enumeration" : Enumeration oppositeOf "literal";
196     }
197
```



ATL Transformation Example

UML to MOF

Date 03/11/2005

```
198     class Stereotype extends GeneralizableElement {
199         reference stereotypeConstraint[*] container : Constraint;
200         reference definedTag[*] container : TagDefinition oppositeOf owner;
201         attribute icon[0-1] : String;
202         attribute baseClass[1-*] : String;
203     }
204
205     class TagDefinition extends ModelElement {
206         reference owner[0-1] : Stereotype oppositeOf definedTag;
207         attribute tagType[0-1] : String;
208         attribute multiplicity[0-1] : Multiplicity;
209     }
210
211     class TaggedValue extends ModelElement {
212         reference type : TagDefinition;
213         reference referenceValue[*] : ModelElement;
214         reference modelElement : ModelElement oppositeOf taggedValue;
215         attribute dataValue[*] : String;
216     }
217
218     class ProgrammingLanguageDataType extends DataType {
219         attribute expression : TypeExpression;
220     }
221
222     class Artifact extends Classifier {
223     }
224
225     class TemplateArgument {
226         reference binding : Binding oppositeOf argument;
227         reference modelElement : ModelElement;
228     }
229 }
```

Appendix B A simplified MOF metamodel in KM3 format

```
1 package Model {
2
3     abstract class ModelElement {
4         -- derived
5         reference requiredElements[*] : ModelElement;
6         reference constraints[*] : Constraint oppositeOf constrainedElements;
7         reference "container"[0-1] : Namespace oppositeOf contents;
8         attribute name : String;
9         -- derived
10        attribute qualifiedName[1-*] ordered : String;
11        attribute annotation : String;
12        operation findRequiredElements(kinds : String, recursive : Boolean) :
13        ModelElement;
14        operation isRequiredBecause(otherElement : ModelElement, reason : String) :
15        Boolean;
16        operation isFrozen() : Boolean;
17        operation isVisible(otherElement : ModelElement) : Boolean;
18    }
19
20    enumeration VisibilityKind {
21        literal public_vis;
22        literal protected_vis;
23        literal private_vis;
24    }
25
26    abstract class Namespace extends ModelElement {
27        reference contents[*] ordered container : ModelElement oppositeOf "container";
28        operation lookupElement(name : String) : ModelElement;
29        operation resolveQualifiedName(qualifiedName : String) : ModelElement;
30        operation findElementsByType(ofType : Class, includeSubtypes : Boolean) :
31        ModelElement;
32        operation nameIsValid(proposedName : String) : Boolean;
33    }
34
35    abstract class GeneralizableElement extends Namespace {
36        reference supertypes[*] ordered : GeneralizableElement;
37        attribute isRoot : Boolean;
38        attribute isLeaf : Boolean;
39        attribute isAbstract : Boolean;
40        attribute visibility : VisibilityKind;
41        operation allSupertypes() : GeneralizableElement;
42        operation lookupElementExtended(name : String) : ModelElement;
43        operation findElementsByTypeExtended(ofType : Class, includeSubtypes : Boolean)
44        : ModelElement;
45    }
46
47    abstract class TypedElement extends ModelElement {
48        reference type : Classifier;
49    }
50
51    abstract class Classifier extends GeneralizableElement {
52
53    }
54
55    class Class extends Classifier {
56        attribute isSingleton : Boolean;
57    }
58
59    class MultiplicityType {
```

```
60         attribute lower : Integer;
61         attribute upper : Integer;
62         attribute isOrdered : Boolean;
63         attribute isUnique : Boolean;
64     }
65
66     abstract class DataType extends Classifier {
67
68     }
69
70     class PrimitiveType extends DataType {
71
72     }
73
74     class EnumerationType extends DataType {
75         attribute labels[1-*] ordered : String;
76     }
77
78     class CollectionType extends DataType, TypedElement {
79         attribute multiplicity : MultiplicityType;
80     }
81
82     class StructureType extends DataType {
83
84     }
85
86     class StructureField extends TypedElement {
87
88     }
89
90     class AliasType extends DataType, TypedElement {
91
92     }
93
94     enumeration ScopeKind {
95         literal instance_level;
96         literal classifier_level;
97     }
98
99     abstract class Feature extends ModelElement {
100         attribute scope : ScopeKind;
101         attribute visibility : VisibilityKind;
102     }
103
104     abstract class StructuralFeature extends Feature, TypedElement {
105         attribute multiplicity : MultiplicityType;
106         attribute isChangeable : Boolean;
107     }
108
109     class Attribute extends StructuralFeature {
110         attribute isDerived : Boolean;
111     }
112
113     class Reference extends StructuralFeature {
114         reference referencedEnd : AssociationEnd;
115         -- derived
116         reference exposedEnd : AssociationEnd;
117     }
118
119     abstract class BehavioralFeature extends Feature, Namespace {
120
121     }
122
123     class Operation extends BehavioralFeature {
124         reference exceptions[*] ordered : Exception;
125         attribute isQuery : Boolean;
126     }
127
128     class Exception extends BehavioralFeature {
```

```
129
130     }
131
132     class Association extends Classifier {
133         attribute isDerived : Boolean;
134     }
135
136     enumeration AggregationKind {
137         literal none;
138         literal shared;
139         literal composite;
140     }
141
142     class AssociationEnd extends TypedElement {
143         attribute isNavigable : Boolean;
144         attribute aggregation : AggregationKind;
145         attribute multiplicity : MultiplicityType;
146         attribute isChangeable : Boolean;
147         operation otherEnd() : AssociationEnd;
148     }
149
150     class Package extends GeneralizableElement {
151     }
152
153
154     class Import extends ModelElement {
155         reference importedNamespace : Namespace;
156         attribute visibility : VisibilityKind;
157         attribute isClustered : Boolean;
158     }
159
160     enumeration DirectionKind {
161         literal in_dir;
162         literal out_dir;
163         literal inout_dir;
164         literal return_dir;
165     }
166
167     class Parameter extends TypedElement {
168         attribute direction : DirectionKind;
169         attribute multiplicity : MultiplicityType;
170     }
171
172     class Constraint extends ModelElement {
173         reference constrainedElements[1-*] : ModelElement oppositeOf constraints;
174         attribute expression : String;
175         attribute language : String;
176         attribute evaluationPolicy : EvaluationKind;
177     }
178
179     enumeration EvaluationKind {
180         literal immediate;
181         literal deferred;
182     }
183
184     class Constant extends TypedElement {
185         attribute value : String;
186     }
187
188     class Tag extends ModelElement {
189         reference elements[1-*] : ModelElement;
190         attribute tagId : String;
191         attribute values[*] ordered : String;
192     }
193 }
```

Appendix C The UML to MOF ATL code

```
1  module UML2MOF;
2  create OUT : MOF from IN : UML;
3
4
5  uses strings;
6
7
8  -----
9  -- HELPERS -----
10 -----
11
12 -- This helper computes a MOF!VisibilityKind from a UML!VisibilityKind.
13 -- CONTEXT: thisModule
14 -- IN:          UML!VisibilityKind
15 -- RETURN:     MOF!VisibilityKind
16 helper def: getVisibility(v : UML!VisibilityKind) : MOF!VisibilityKind =
17     if v = #vk_public
18     then
19         #public_vis
20     else
21         if v = #vk_private
22         then
23             #private_vis
24         else
25             if v = #vk_protected
26             then
27                 #protected_vis
28             else
29                 #public_vis -- default
30             endif
31         endif
32     endif;
33
34 -- This helper computes the MOF!VisibilityKind that corresponds to the
35 -- UML!VisibilityKind of the contextual UML!ModelElement. If this visibility
36 -- kind is undefined, the helper returns 'public_vis' as a default value.
37 -- CONTEXT: UML!ModelElement
38 -- RETURN:   MOF!VisibilityKind
39 helper context UML!ModelElement def: getMOFVisibility() : MOF!VisibilityKind =
40     let v : UML!VisibilityKind = self.visibility in
41     if not v.oclIsUndefined()
42     then
43         thisModule.getVisibility(v)
44     else
45         #public_vis
46     endif;
47
48 -- This helper computes the MOF!ScopeKind that corresponds to the
49 -- UML!ScopeKind of the contextual UML!Feature.
50 -- CONTEXT: UML!Feature
51 -- RETURN:   MOF!ScopeKind
52 helper context UML!Feature def: getMOFScope() : MOF!ScopeKind =
53     if self.ownerScope = #sk_instance
54     then
55         #instance_level
56     else
57         #classifier_level
58     endif;
59
60 -- This helper computes the MOF isChangeable boolean attribute that corresponds
61 -- to the UML!Changeability of the contextual UML!ModelElement (whose
62 -- changeability is not undefined).
63 -- 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.ocIsUndefined()
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
95      Tuple{
96          lower = m.range->asSequence()->first().lower,
97          upper = m.range->asSequence()->first().upper,
98          isOrdered = (o = 'ok_ordered'),
99          isUnique = isUnique
100      };
101
102  -- This helper computes the tuple encoding the MOF multiplicity that
103  -- corresponds to the UML!Multiplicity of the contextual UML!StructuralFeature.
104  -- If the multiplicity of the contextual structural feature is undefined, the
105  -- helper returns (1,1,true,true) as a default tuple. Otherwise, it returns the
106  -- tuple computed by the getMultiplicity helper. Note that if the ordering of
107  -- the contextual structural feature is undefined, it is considered as ordered.
108  -- CONTEXT: UML!StructuralFeature
109  -- RETURN:     TupleType(Integer, Integer, Boolean, Boolean)
110  helper context UML!StructuralFeature def: getMOFMultiplicity() :
111      TupleType(lower : Integer, upper : Integer,
112                isOrdered : Boolean, isUnique :
113      Boolean) =
114      if not self.multiplicity.ocIsUndefined()
115      then
116          if not self.ordering.ocIsUndefined()
117          then
118              thisModule.getMultiplicity(self.multiplicity, self.ordering, false)
119          else
120              thisModule.getMultiplicity(self.multiplicity, 'ok_ordered', false)
121          endif
122      else
123          Tuple{lower = 1, upper = 1, isOrdered = true, isUnique = true}
124      endif;
125
126  -- Helper ...
127  -- CONTEXT: UML!ModelElement
128  -- RETURN:      String
129  --helper context UML!ModelElement def: getMOFQualifiedname() : String =
130      self.name;
131
132  -----
    
```



ATL Transformation Example

UML to MOF

Date 03/11/2005

```
133  -- RULES -----
134  -----
135
136  -- Rule 'Package'
137  -- This rule generates a MOF package from each UML package that has a
138  -- stereotype named 'metamodel'.
139  -- Supertypes of the generated package correspond to the parent of the
140  -- generalization of the input UML package.
141  rule Package {
142      from
143          up : UML!Package (
144              up.stereotype->exists(e | e.name = 'metamodel')
145          )
146      to
147          mp : MOF!Package (
148              -- Begin bindings inherited from ModelElement
149              name <- up.name,
150              annotation <- '',
151              container <- up.namespace,
152              constraints <- up.constraint,
153              requiredElements <- ,
154              -- End of bindings inherited from ModelElement
155
156              -- Begin bindings inherited from Namespace
157              contents <- up.ownedElement,
158              -- End of bindings inherited from Namespace
159
160              -- Begin bindings inherited from GeneralizableElement
161              isRoot <- up.isRoot,
162              isLeaf <- up.isLeaf,
163              isAbstract <- up.isAbstract,
164              visibility <- up.getMOFVisibility(),
165              supertypes <- up.generalization->collect(e | e.parent)
166              -- End of bindings inherited from GeneralizableElement
167          )
168      }
169
170  -- Rule 'Constraint'
171  -- This rule generates a MOF constraint from a UML one. Properties of the
172  -- generated constraint, except evaluationPolicy, are copied from the input UML
173  -- constraint.
174  -- The MOF evaluationPolicy property, which has no equivalent in UML, is set to
175  -- the default 'immediate' value.
176  rule Constraint {
177      from
178          uc : UML!Constraint
179      to
180          mc : MOF!Constraint(
181              -- Begin bindings inherited from ModelElement
182              name <- uc.name,
183              annotation <- '',
184              container <- uc.namespace,
185              constraints <- uc.constraint,
186              requiredElements <- ,
187              -- End of bindings inherited from ModelElement
188
189              expression <- uc.body.body,
190              language <- uc.body.language,
191              constrainedElements <- uc.constrainedElement,
192              evaluationPolicy <- #immediate
193          )
194      }
195
196  -- Rule 'Comment'
197  -- This rule generates a MOF constraint from each UML Comment that has a
198  -- 'constraint' stereotype.
199  -- 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 {
```



```
202         from
203         uc : UML!Comment (
204             uc.stereotype->exists(e | e.name = 'constraint')
205         )
206         to
207         mc : MOF!Constraint(
208             -- Begin bindings inherited from ModelElement
209             name <- uc.name,
210             annotation <- '',
211             container <- uc.namespace,
212             constraints <- uc.constraint,
213             --
214             -- End of bindings inherited from ModelElement
215
216             expression <- uc.body,
217             language <- 'OCL',
218             constrainedElements <- uc.annotatedElement
219         )
220     }
221
222     -- Rule 'Class'
223     -- This rule generates a MOF class from each UML class whose namespace (which
224     -- expected to be a Package) has a 'metamodel' stereotype.
225     -- Properties of the generated class are copied from the input UML class
226     -- properties.
227     rule Class {
228         from
229         uc : UML!Class (
230             uc.namespace.stereotype->exists(e | e.name = 'metamodel')
231         )
232         to
233         mc : MOF!Class (
234             -- Begin bindings inherited from ModelElement
235             name <- uc.name,
236             annotation <- '',
237             container <- uc.namespace,
238             constraints <- uc.constraint,
239             --
240             -- End of bindings inherited from ModelElement
241
242             -- Begin bindings inherited from Namespace
243             contents <- uc.ownedElement,
244             -- End of bindings inherited from Namespace
245
246             -- Begin bindings inherited from GeneralizableElement
247             isRoot <- uc.isRoot,
248             isLeaf <- uc.isLeaf,
249             isAbstract <- uc.isAbstract,
250             visibility <- uc.getMOFVisibility(),
251             supertypes <- uc.generalization->collect(e | e.parent),
252             -- End of bindings inherited from GeneralizableElement
253
254             isSingleton <- false
255         )
256     }
257
258     -- Rule 'Attribute'
259     -- This rule generates a MOF attribute from each UML attribute.
260     -- Properties of the generated attribute are copied from the input UML
261     -- attribute properties. Note that the 'isDerived' attribute is set to the
262     -- false default value.
263     rule Attribute {
264         from
265         ua : UML!Attribute
266         to
267         ma : MOF!Attribute (
268             -- Begin bindings inherited from ModelElement
269             name <- ua.name,
270             annotation <- '',
```

```
271         container <- ua.owner,
272         constraints <- ua.constraint,
273 --         requiredElements <-,
274         -- End of bindings inherited from ModelElement
275
276         -- Begin bindings inherited from Feature
277         scope <- ua.getMOFScope(),
278         visibility <- ua.getMOFVisibility(),
279         -- End of bindings inherited from Feature
280
281         -- Begin bindings inherited from StructuralFeature
282         multiplicity <- ua.getMOFMultiplicity(),
283         isChangeable <- ua.getMOFIsChangeable(),
284         -- End of bindings inherited from StructuralFeature
285
286         -- Begin bindings inherited from TypedElement
287         type <- ua.type,
288         -- End of bindings inherited from TypedElement
289
290         isDerived <- false
291     )
292 }
293
294 -- Rule 'Parameter'
295 -- This rule generates a MOF parameter from each UML parameter.
296 -- Properties of the generated parameter are copied from the input UML
297 -- parameter properties. Note that the MOF multiplicity attribute is not set
298 -- since the corresponding information is not available in the UML metamodel.
299 -- The MOF multiplicity attribute, not encoded in UML, is left undefined.
300 rule Parameter {
301     from
302         up : UML!Parameter
303     to
304         mp : MOF!Parameter (
305             -- Begin bindings inherited from ModelElement
306             name <- up.name,
307             annotation <- '',
308             container <- up.namespace,
309             constraints <- up.constraint,
310 --             requiredElements <-,
311             -- End of bindings inherited from ModelElement
312
313             -- Begin bindings inherited from TypedElement
314             type <- up.type,
315             -- End of bindings inherited from TypedElement
316
317             direction <-
318                 if up.kind = #pdk_in
319                 then
320                     #in_dir
321                 else
322                     if up.kind = #pdk_inout
323                     then
324                         #inout_dir
325                     else
326                         if up.kind = #pdk_out
327                         then
328                             #out_dir
329                         else
330                             #return_dir
331                         endif
332                     endif
333                 endif
334             multiplicity <-
335         )
336 }
337
338 -- Rule 'Operation'
339 -- This rule generates a MOF operation from each UML operation.
```



ATL Transformation Example

UML to MOF

Date 03/11/2005

```
340 -- Properties of the generated operation are copied from the input UML
341 -- operation properties. Note that the exceptions property of the generated
342 -- MOF operation is set to an empty set as a default value.
343 rule Operation {
344     from
345         uo : UML!Operation
346     to
347         mo : MOF!Operation (
348             -- Begin bindings inherited from ModelElement
349             name <- uo.name,
350             annotation <- '',
351             container <- uo.owner,
352             constraints <- uo.constraint,
353             --
354             requiredElements <- ,
355             -- End of bindings inherited from ModelElement
356
357             -- Begin bindings inherited from Namespace
358             contents <- uo.parameter,
359             -- End of bindings inherited from Namespace
360
361             -- Begin bindings inherited from Feature
362             scope <- uo.getMOFScope(),
363             visibility <- uo.getMOFVisibility(),
364             -- End of bindings inherited from Feature
365
366             isQuery <- uo.isQuery,
367             exceptions <- Set{}
368         )
369 }
370
371 -- Rule 'Association'
372 -- This rule generates a MOF association from each UML association.
373 -- Properties of the generated association are copied from the input UML
374 -- association properties. contents of the generated association correspond to
375 -- the MOF association end generated for the connection of the input UML
376 -- association.
377 rule Association {
378     from
379         ua : UML!Association
380     to
381         ma : MOF!Association (
382             -- Begin bindings inherited from ModelElement
383             name <- ua.name,
384             annotation <- '',
385             container <- ua.namespace,
386             constraints <- ua.constraint,
387             --
388             requiredElements <- ,
389             -- End of bindings inherited from ModelElement
390
391             -- Begin bindings inherited from Namespace
392             contents <- ua.connection,
393             -- End of bindings inherited from Namespace
394
395             -- Begin bindings inherited from GeneralizableElement
396             isRoot <- ua.isRoot,
397             isLeaf <- ua.isLeaf,
398             isAbstract <- ua.isAbstract,
399             visibility <- ua.getMOFVisibility(),
400             supertypes <- ua.generalization->collect(e | e.parent)
401             -- End of bindings inherited from GeneralizableElement
402         )
403 }
404
405 -- Rule 'AssociationEnd'
406 -- This rule generates a MOF association end, along with an optional reference,
407 -- from each UML association end.
408 -- The MOF reference is only generated from navigable UML association ends. For
409 -- this purpose, the rule iterates through a Sequence that contains 1 element
410 -- if UML association end is navigable, 0 otherwise.
```

```

409 -- Properties of the generated association end are copied from the input UML
410 -- association end properties.
411 -- When generated, the reference has the same name than its associated
412 -- association end. Its container corresponds to the class that to which is
413 -- associated the other association end contained by the association that also
414 -- contains the input UML association end.
415 -- Its scope and visibility are respectively set to the 'instance_level' and
416 -- 'public_vis' default values. The values of its type, multiplicity and
417 -- isChangeable attributes are copied from the input UML association end.
418 -- The constraints of the generated reference are packed within a single
419 -- element Sequence for the purpose of compatibility with the reference
420 -- sequence of the 'foreach' operator.
421 -- Finally, ...
422 rule AssociationEnd {
423     from
424         ua : UML!AssociationEnd
425     to
426         ma : MOF!AssociationEnd(
427             -- Begin bindings inherited from ModelElement
428             name <- ua.name,
429             annotation <- '',
430             container <- ua.association,
431             constraints <- ua.constraint,
432             --
433             requiredElements <- ,
434             -- End of bindings inherited from ModelElement
435
436             -- Begin bindings inherited from TypedElement
437             type <- ua.participant,
438             -- End of bindings inherited from TypedElement
439
440             isNavigable <- ua.isNavigable,
441             aggregation <-
442                 if ua.aggregation = #ak_aggregate
443                 then
444                     #shared
445                 else
446                     if ua.aggregation = #ak_composite
447                     then
448                         #composite
449                     else
450                         #none
451                     endif
452                 endif,
453             multiplicity <-
454                 thisModule.getMultiplicity(ua.multiplicity, ua.ordering, true),
455             isChangeable <- ua.getMOFIsChangeable()
456         ),
457     mr : distinct MOF!Reference foreach(c in
458         if ua.isNavigable
459         then
460             Sequence{true}
461         else
462             Sequence{}
463         endif) (
464         -- Begin bindings inherited from ModelElement
465         name <- ua.name,
466         annotation <- '',
467         container <- ua.association.connection
468             ->select(e | not (e = ua))
469             ->first().participant,
470         constraints <- Sequence{ua.constraint},
471         --
472         requiredElements <- ,
473         -- End of bindings inherited from ModelElement
474
475         -- Begin bindings inherited from Feature
476         scope <- #instance_level,
477         visibility <- ua.getMOFVisibility(),
478         -- End of bindings inherited from Feature

```

```

478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546

```

```

-- Begin bindings inherited from StructuralFeature
-- If the 2 following bindings are exchanged with the referencedEnd
-- one, an error may be raised due to MDR inconsistency checkings.
multiplicity <-
    thisModule.getMultiplicity(ua.multiplicity, ua.ordering, true),
isChangeable <- ua.getMOFIsChangeable(),
-- End of bindings inherited from StructuralFeature

-- Begin bindings inherited from TypedElement
type <- ua.participant,
-- End of bindings inherited from TypedElement

-- The Association corresponding to the Reference is derived: the
-- exposedEnd reference should not be assigned.
exposedEnd <- ua.association.connection
--
--
->select(e | not (e = ua))->first(),
referencedEnd <- ma
)
}

-- Rule 'TaggedValue'
-- This rule generates a MOF tag from each UML tagged value whose type is
-- neither named 'element.uuid' nor 'isValid'.
-- Properties of the generated tag are copied from the input UML tagged value
-- properties.
rule TaggedValue {
    from
        ut : UML!TaggedValue (
            (ut.type.name <> 'element.uuid') and
            (ut.type.name <> 'isValid')
        )
    using {
        name : String = ut.type.name;
    }
    to
        ot : MOF!Tag (
            -- Begin bindings inherited from ModelElement
            name <- ut.name,
            annotation <- '',
            container <- ut.namespace,
            constraints <- ut.constraint,
            --
            requiredElements <- ,
            -- End of bindings inherited from ModelElement

            tagId <- name,
            values <- ut.dataValue,
            elements <- Sequence{ut.modelElement}
        )
    }

-- Rule 'Dependency'
-- The rule generates a MOF!Import from each input UML!Dependency that has a
-- stereotype of either 'clustering' or 'import' type. Note that input
-- dependencies can have a clustering or an import stereotype, but not both.
-- Properties of the generated import are copied from those of the input
-- dependency. Note that the isClustered attribute is set to false if the
-- input dependency has an 'import' stereotype, true otherwise (i.e. if it has
-- a 'clustering' stereotype).
rule Dependency {
    from
        ud : UML!Dependency (
            ud.stereotype
            ->exists(e | e.name = 'import' or e.name = 'clustering')
        )
    using {
        importer : UML!ModelElement = ud.client->asSequence()->first();
        imported : UML!ModelElement = ud.supplier->asSequence()->first();
    }
}

```



ATL Transformation Example

UML to MOF

Date 03/11/2005

```
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                 if ud.stereotype->exists(e | e.name = 'import')
560                 then
561                     false
562                 else
563                     true
564                 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 }
```