ATL Transformation Examples

The KM3 to Metric ATL transformation

- version 0.1 -

September 2005

by

ATLAS group

LINA & INRIA

Nantes

Content

1 Introduction 1

2 The KM3 to Metrics ATL transformation 1

2.1 Transformation overview 1

2.2 Metamodels 1

2.2.1 The KM3 metamodel 1

2.2.2 The Metrics metamodel 2

2.3 Rules specification 3

2.4 ATL code 3

2.4.1 Helpers 3

2.4.2 Rules 5

3 References 5

Appendix A: The KM3 metamodel in KM3 format 6

Appendix B: The Metrics metamodel in KM3 format 8

Appendix C: The KM3 to Metrics ATL code 9

Figures

Figure 1. The KM3 metamodel 2

Figure 2. The Metrics metamodel 3

1 Introduction

Due to the even growing complexity of metamodels, the establishment of metamodel metrics now requires the use of dedicated tools. This ATL transformation example aims to show how the ATL language may be used to meet this objective by generating metrics models from metamodels.

The Kernel MetaMetaModel (KM3) is a simple textual notation enabling quick definition of metamodels [1]. The KM3 to Metrics ATL example aims to demonstrate the feasibility of computing some metamodel metrics by means of the ATL language. For this purpose, we have developed a basic Metrics metamodel enabling to encode measures of different types.

The KM3 to Metrics ATL transformation defines different metamodel metrics that are relative either to the whole model, or to specific model elements. The metrics provided in the scope of this transformation include simple measurements as well as more complex computations.

2 The KM3 to Metrics ATL transformation

2.1 Transformation overview

The KM3 to Metrics transformation is a single step transformation that produces a Metrics model from a KM3 model.

Users of the ATL Development Tools (ADT) [2] can easily produce their own KM3 input model by 1) entering a textual KM3 metamodel and, 2) exporting the produced textual KM3 metamodel into a KM3 model by means of the *Inject KM3 file to KM3 model* contextual menu option.

2.2 Metamodels

The KM3 to Metrics transformation is based on both the KM3 and Metrics metamodel. The KM3 descriptions of these metamodels can respectively be found in Appendix A: and Appendix B:. They are further described in the following subsections.

2.2.1 The KM3 metamodel

The KM3 metamodel [1] provides semantics for metamodel descriptions. The KM3 metamodel conforms to itself and can therefore be used to define KM3 metamodels. Figure 1 provides a description of a subset of the KM3 metamodel. Its corresponding complete textual description in the KM3 format is also provided in Appendix A:.

A KM3 Metamodel is composed of Packages. A Package contains some abstract ModelElements (TypedElements, Classifiers, EnumLiterals and Packages, since a Package is itself a ModelElement). A ModelElement is characterized by its *name*. The ModelElement entity inherits from the abstract LocatedElement entity. This last defines a *location* attribute that aims to encode, in a string format, the location of the declaration of the corresponding element within its source file.

A Classifier can be either an Enumeration, a DataType or a Class. An Enumeration is composed of EnumLiteral elements. The Class element defines the Boolean *isAbstract* attribute that enables to declare abstract classes. A Class can have direct *supertypes* (Class elements).

A Class is composed of abstract StructuralFeatures. The StructuralFeature element inherits from the abstract TypedElement entity. This entity defines the *lower*, *upper*, *isOrdered* and *isUnique* attributes. The two first attributes defines the minimal and maximal cardinality of a TypedElement. The *isOrdered* and *isUnique* Boolean attributes respectively encode the fact that the different instances of the

TypedElement are ordered and unique. A TypedElement obviously has a *type*, which corresponds to a Classifier element.

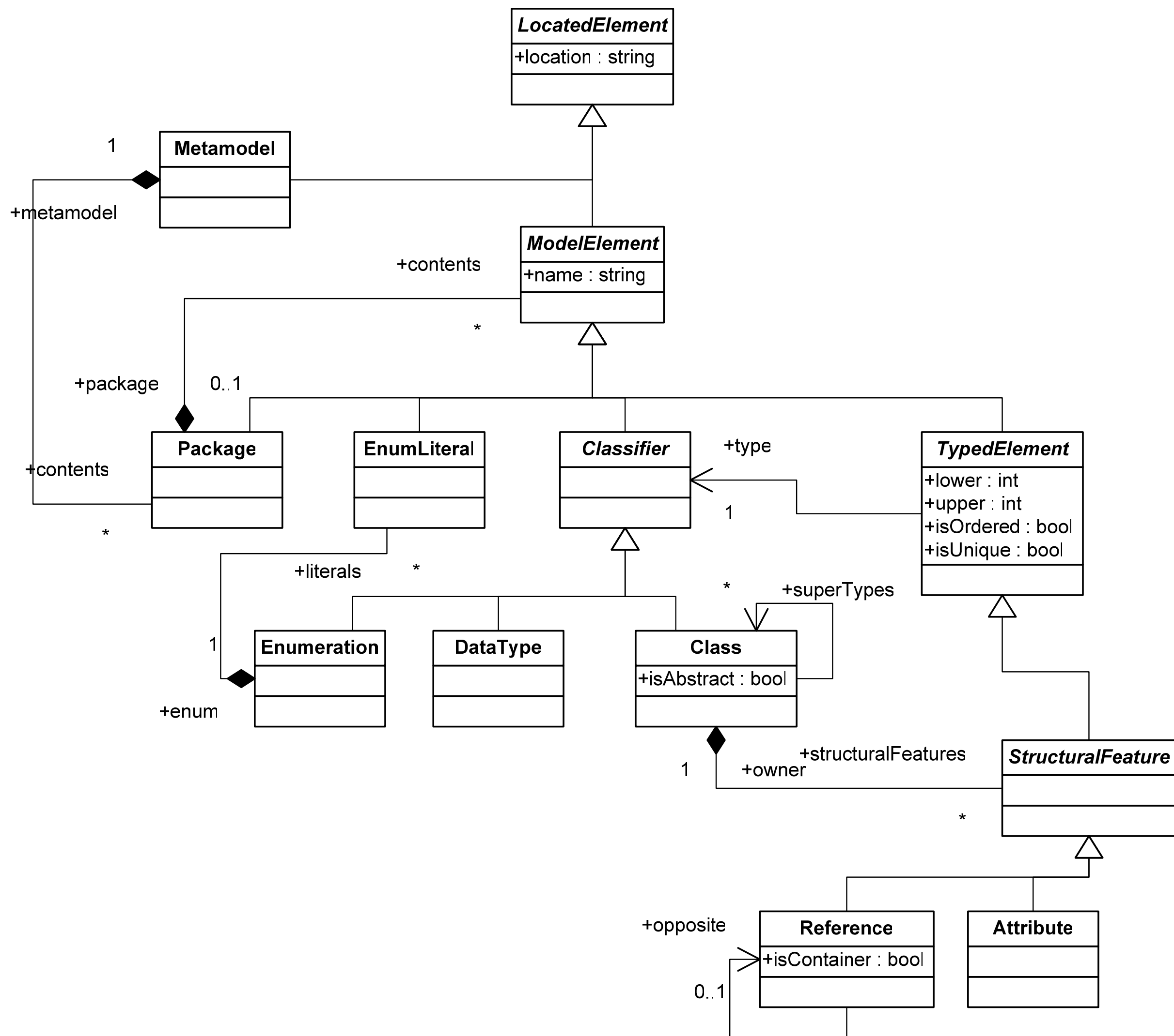

Figure 1. The KM3 metamodel

A StructuralFeature is either a Reference or an Attribute. The Reference element defines the Boolean *isContainer* attribute that encode the fact that the pointed elements are contained by the reference. A Reference can also have an *opposite* reference. Finally, a StructuralFeature has an owner of the type Class (the owner reference is the opposite of the Class *structuralFeatures* reference).

2.2.2 The Metrics metamodel

The Metrics metamodel is a simple metamodel for metrics encoding. Figure 2 provides a description of this metamodel. The main element of the Metrics metamodel is the abstract Metric entity. This entity defines two string attributes that are common to the different types of encoded metrics: the *context* and the *label* of the Metric. The context aims to identify, by means of a string value, the KM3 element the metric relies to. The *label*, as for it, provides a textual description of the metric (e.g. what is measured).

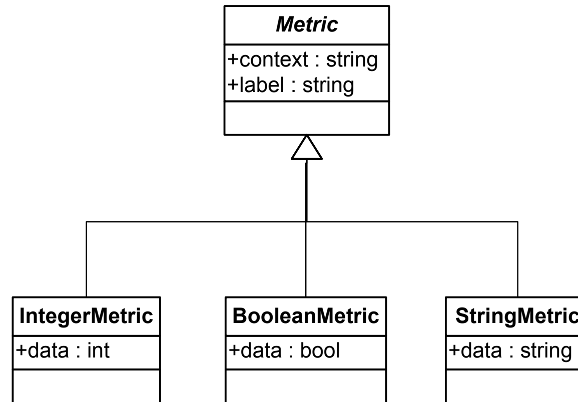

Figure 2. The Metrics metamodel

The Metrics metamodel defines three types of concrete metrics, which all inherit from the abstract Metric entity: the IntegerMetric, the BooleanMetric and the StringMetric. They respectively encode integral, boolean and string measurements. Each of them has a *data* attribute, of the type that corresponds to the metric type, which encodes the metric value.

2.3 Rules specification

Here are the rules used to generate a Metrics model from a KM3 model:

- Five IntegerMetric elements, corresponding to the number of classes and associations of the input KM3 model, as well as its maximal inheritance depth and its number of inheritance trees and graphs, are generated from a KM3 Metamodel;
- Four IntegerMetrics elements, corresponding to the number of own attributes and own references of the input Class element, as well as the total number (including inherited ones) of attributes and references of the Class, are generated from a KM3 Class.

The maximal inheritance depth corresponds to the maximal length of the inheritance chains of the input model.

An independent inheritance tree corresponds to a set of classes of the input model that have the same root class. A class without any supertype forms a single element inheritance tree. Note that a given class may belong to different inheritance trees if it has multiple inheritances.

An independent inheritance graph corresponds to a set of classes of the input model that are linked to each other by means of an inheritance relationship (either supertype or subtype). A class with no supertypes nor subtypes forms a single element inheritance graph. Note that, unlike the with an inheritance tree, a given class belongs to a single inheritance graph. As a consequence, the number of inheritance graphs associated with a given model is lower or equal to its number of inheritance trees.

2.4 ATL code

The ATL code for the UML to MOF transformation is provided in Appendix C:. It consists of 15 helpers and 2 rules.

2.4.1 Helpers

The **allClasses** helper computes the set of all Class elements of the input KM3 model. Since this set is used several times, defining this constant helper enables to avoid multiple calculations of the set of classes.

The **allReferences** helper computes the set of all Reference elements of the input KM3 model. Since this set is used several times, defining this constant helper enables to avoid multiple calculations of the set of references.

The **inheritanceRoots** helper computes the set of all Class elements that correspond to the root of an inheritance tree. These Class elements are those that have no supertypes. Since this set is used several times, defining this constant helper enables to avoid multiple calculations of the set of classes

The **attributeNb** helper computes the number of own attributes of its contextual Class. As a constant helper, it calculates this value only once (at the time it is called for the first time) and stores it as a cached value.

The **referenceNb** helper is similar to the attributeNb helper, except that it deals with the references of its contextual Class.

The **attributeNb2** helper computes the total number of attributes (including inherited ones) of its contextual Class. This helper is a recursive helper which returns the number of attributes of its contextual Class plus the values returned by its recursive calls on each supertype of this contextual Class. As a constant helper, it calculates this value only once (at the time it is called for the first time) and stores it as a cached value.

The **referenceNb2** helper is similar to the attributeNb2 helper, except that it deals with the references of its contextual Class.

The **getInheritanceLenght()** helper aims to compute the maximal length of the inheritance chains that lead to the contextual Class. This helper is a recursive helper. If the contextual Class has no supertypes, it returns 0. Otherwise, it returns the maximal value computed by its different recursive calls on the supertypes of the contextual Class (the length of the inheritance chain is incremented by 1 at each successive recursive call).

The **getInheritanceMaxDepth()** helper aims to compute the maximal length of an inheritance chain within the input KM3 model. For this purpose, it iterates the classes of the input model, calculating for their own inheritance length by calling the getInheritanceLenght helper. It then returns the maximal value computed by this mean.

The **getInheritanceTreeNb()** helper calculates the number of inheritance trees in the input model. This number corresponds to the number of Class elements that are roots of these trees. As a consequence, the helper computes the number of tree root Class elements, that is the number of Class that has no supertypes.

The **subtypes** helper aims to compute the set of direct subtypes of the contextual Class. For this purpose, it iterates the Class elements of the input model, and selects those that have the contextual Class as a supertype. As a constant helper, it calculates this value only once (at the time it is called for the first time) and stores it as a cached value.

The **getTree()** helper calculates the set of Class elements composing the inheritance tree that has the contextual Class as root element. This helper is a recursive helper. If the contextual Class has no supertypes, it returns a set containing the only contextual Class. Otherwise, the helper iterates the supertypes of the contextual Class and returns a set containing this contextual Class along with the results of its recursive calls on the supertypes of the Class.

The **existsHLink()** helper returns a boolean value stating whether there exists any inheritance relationships between the Class elements of the two set of classes it receives as parameters. For this purpose, it iterates the Class elements of the first set and checks whether there exists an inheritance link (either a supertype or a subtype relation) between this current Class and any Class of the second set. The helper returns true if it finds out such an inheritance relation, false otherwise.

The **computeGraphNb()** helper aims to compute the number of inheritance graphs corresponding to the sequence of inheritance trees passed as a parameter. This helper is a recursive helper. If the provided sequence contains a single tree, the helper returns 1, otherwise, it calculates a new

sequence of inheritance trees by trying to merge the first tree of the sequence with the following ones according to the inheritance links that may exist between the classes of these trees. Indeed, two inheritance trees of a same model belong to the same inheritance graph if there exists at least an inheritance relation between one Class of each tree. The helper therefore iterates the input tree sequence and checks whether it exists an inheritance relation between the first tree and the currently iterated one (by calling existsHLink the helper). If so, both trees are merged in order to form the new first tree of the new sequence. If the newly computed sequence contains a single tree, the helper returns 1. Otherwise, the helper returns 1 plus the value returned by its recursive call on a sequence that corresponds to the newly computed sequence without its first tree. Indeed, after the successive mergings, this last corresponds to an inheritance graph of the input model and has no inheritance links with the remaining trees.

The **getInheritanceGraphNb()** helper aims to compute the number of inheritance graphs of the input KM3 model. To this end, its first computes a sequence containing the inheritance trees of the input model. This is simply achieved by iterating the tree root Class elements and calling the getTree helper on each of them. The helper then returns the value returned by the computeGraphNb helper called with the calculated sequence as a parameter.

2.4.2 Rules

The **Model** rule generates Metrics elements that are relative to the input KM3 model. This rule generates 5 IntegerMetric elements. The first one corresponds to the number of Class of the input KM3 model. The second one is associated with the number of associations in the model. This value corresponds to the number of references without an opposite, plus the half of the number of references that have an opposite (since two opposite references represent a same association). The third generated metric corresponds to the maximal inheritance length in the input model, and is computed by the getInheritanceMaxDepth() helper. The two last metrics correspond to the number of inheritance trees and graphs within the KM3 model. They are respectively computed by the getInheritanceTreeNb() and getInheritanceGraphNb() helpers.

The **Class** rule generates Metrics elements that are relative to Class elements of the input KM3 model. This rule generates 4 IntegerMetric elements: one devoted to the number of own attributes of the current class, one to its number of own references, one to its total number of attribute and one to its total number of references. The values of the *data* attribute of the generated Metrics are respectively provided by the attributeNb, referenceNb, attributeNb2 and referenceNb2 helpers.

3 References

[1] KM3 User Manual. The Eclipse Generative Model Transformer (GMT) project, http://eclipse.org/gmt/.

[2] The ATL Development Tools (ADT). The Eclipse Generative Model Transformer (GMT) project, http://eclipse.org/gmt/.

Appendix A: The KM3 metamodel in KM3 format

```
1	package KM3 {
2	
3		abstract class LocatedElement {
4			attribute location : String;
5			attribute commentsBefore[*] ordered : String;
6			attribute commentsAfter[*] ordered : String;
7		}
8	
9		abstract class ModelElement extends LocatedElement {
10			attribute name : String;
11			reference "package" : Package oppositeOf contents;
12		}
13	
14		class Classifier extends ModelElement {
15		}
16	
17		class DataType extends Classifier {
18		}
19	
20		class Enumeration extends Classifier {
21			reference literals[*] ordered container : EnumLiteral oppositeOf enum;
22		}
23	
24		class EnumLiteral extends ModelElement {
25			reference enum : Enumeration oppositeOf literals;
26		}
27	
28		class TemplateParameter extends Classifier {
29		}
30	
31		class Class extends Classifier {
32			reference parameters[*] ordered container : TemplateParameter;
33			attribute isAbstract : Boolean;
34			reference supertypes[*] : Class;
35			reference structuralFeatures[*] ordered container : StructuralFeature
36	oppositeOf owner;
37			reference operations[*] ordered container : Operation oppositeOf owner;
38		}
39	
40		class TypedElement extends ModelElement {
41			attribute lower : Integer;
42			attribute upper : Integer;
43			attribute isOrdered : Boolean;
44			attribute isUnique : Boolean;
45			reference type : Classifier;
46		}
47	
48		class StructuralFeature extends TypedElement {
49			reference owner : Class oppositeOf structuralFeatures;
50			reference subsetOf[*] : StructuralFeature oppositeOf derivedFrom;
51			reference derivedFrom[*] : StructuralFeature oppositeOf subsetOf;
52		}
53	
54		class Attribute extends StructuralFeature {
55		}
56	
57		class Reference extends StructuralFeature {
58			attribute isContainer : Boolean;
59			reference opposite[0-1] : Reference;
```

```
60          }
61
62          class Operation extends TypedElement {
63                  reference owner : Class oppositeOf operations;
64                  reference parameters[*] ordered container : Parameter oppositeOf owner;
65          }
66
67          class Parameter extends TypedElement {
68                  reference owner : Operation oppositeOf parameters;
69          }
70
71          class Package extends ModelElement {
72                  reference contents[*] ordered container : ModelElement oppositeOf "package";
73                  reference metamodel : Metamodel oppositeOf contents;
74          }
75
76          class Metamodel extends LocatedElement {
77                  reference contents[*] ordered container : Package oppositeOf metamodel;
78          }
79  }
80
81  package PrimitiveTypes {
82          datatype Boolean;
83          datatype Integer;
84          datatype String;
85  }
```

Appendix B: The Metrics metamodel in KM3 format

```
1   package Metrics {
2   
3           abstract class Metric {
4                   attribute context : String;
5                   attribute label : String;
6           }
7   
8           class StringMetric extends Metric {
9                   attribute data : String;
10          }
11  
12          class BooleanMetric extends Metric {
13                  attribute data : Boolean;
14          }
15  
16          class IntegerMetric extends Metric {
17                  attribute data : Integer;
18          }
19  }
20  
21  package PrimitiveTypes {
22          datatype Boolean;
23          datatype Integer;
24          datatype String;
25  }
```

Appendix C: The KM3 to Metrics ATL code

```
1   module KM32Metrics;
2   create OUT : Metrics from IN : KM3;
3
4   -------------------------------------------------------------------------------
5   -- HELPERS --------------------------------------------------------------------
6   -------------------------------------------------------------------------------
7
8   -- This helper computes the set of all the Class elements of the input model.
9   -- CONTEXT:    thisModule
10  -- RETURN:     Set(KM3!Class)
11  helper def: allClasses : Set(KM3!Class) = KM3!Class.allInstances();
12
13  -- This helper computes the set of all the Reference elements of the input
14  -- model.
15  -- CONTEXT:    thisModule
16  -- RETURN:     Set(KM3!Reference)
17  helper def: allReferences : Set(KM3!Reference) = KM3!Reference.allInstances();
18
19  -- This helper computes the set of all the Class elements of the input model
20  -- that correspond to the root of the different inheritance trees.
21  -- CONTEXT:    thisModule
22  -- RETURN:     Set(KM3!Class)
23  helper def: inheritanceRoots : Set(KM3!Class) =
24          thisModule.allClasses
25                  ->select(e | e.supertypes->isEmpty());
26
27  -- This helper returns the number of own attributes of the contextual Class.
28  -- CONTEXT:    KM3!Class
29  -- RETURN:     Integer
30  helper context KM3!Class def: attributeNb : Integer =
31          self.structuralFeatures
32                  ->select(e | e.oclIsTypeOf(KM3!Attribute))
33                  ->size();
34
35  -- This helper returns the total number of attributes (including inherited
36  -- ones) of the contextual Class.
37  -- CONTEXT:    KM3!Class
38  -- RETURN:     Integer
39  helper context KM3!Class def: attributeNb2 : Integer =
40          self.attributeNb +
41          self.supertypes
42                  ->iterate(e; sum : Integer = 0 |
43                          sum + e.attributeNb
44                  );
45
46  -- This helper returns the number of own references of the contextual Class.
47  -- CONTEXT:    KM3!Class
48  -- RETURN:     Integer
49  helper context KM3!Class def: referenceNb : Integer =
50          self.structuralFeatures
51                  ->select(e | e.oclIsTypeOf(KM3!Reference))
52                  ->size();
53
54  -- This helper returns the total number of references (including inherited
55  -- ones) of the contextual Class.
56  -- CONTEXT:    KM3!Class
57  -- RETURN:     Integer
58  helper context KM3!Class def: referenceNb2 : Integer =
59          self.referenceNb +
60          self.supertypes
61                  ->iterate(e; sum : Integer = 0 |
62                          sum + e.referenceNb
63                  );
```

```
 64
 65   -- This helper returns the size of the maximum lenght of the inheritance trees
 66   -- that lead to the contextual Class.
 67   -- If the class has no supertype, the helper returns 0. Otherwise, it iterates
 68   -- through the set of supertypes of the contextual Class in order to find out
 69   -- the maximum lenght of inheritance trees (from roots to the contextual
 70   -- Class).
 71   -- CONTEXT:    KM3!Class
 72   -- RETURN:     Integer
 73   helper context KM3!Class def: getInheritanceLenght() : Integer =
 74           if self.supertypes->size() = 0
 75           then
 76                   0
 77           else
 78                   self.supertypes->iterate(e; max : Integer = 0 |
 79                           if e.getInheritanceLenght() + 1 > max
 80                           then
 81                                   e.getInheritanceLenght() + 1
 82                           else
 83                                   max
 84                           endif
 85                   )
 86           endif;
 87
 88   -- This helper returns the maximal depth of an inheritance tree in the input
 89   -- metamodel. For this purpose, it calls the getInheritanceLenght() helper on
 90   -- each Class of the input model and returns the maximum value.
 91   -- CONTEXT:    thisModule
 92   -- RETURN:     Integer
 93   helper def: getInheritanceMaxDepth() : Integer =
 94           thisModule.allClasses
 95                   ->iterate(e; max : Integer = 0 |
 96                           if e.getInheritanceLenght() > max
 97                           then
 98                                   e.getInheritanceLenght()
 99                           else
100                                   max
101                           endif
102                   );
103
104   -- This helper returns the number of inheritance trees in the input model.
105   -- For this purpose, it computes the number of "root" classes, that is the
106   -- classes that donot have any supertype.
107   -- CONTEXT:    thisModule
108   -- RETURN:     Integer
109   helper def: getInheritanceTreeNb() : Integer =
110           thisModule.inheritanceRoots->size();
111
112   -- This helper returns the set of subtypes of its contextual Class.
113   -- For this purpose, it selects among all existing Class elements, those that
114   -- have the contextual Class as a supertype.
115   -- CONTEXT:    KM3!Class
116   -- RETURN:     Set(KM3!Class)
117   helper context KM3!Class def: subTypes : Set(KM3!Class) =
118           thisModule.allClasses
119                   ->select(e | e.supertypes->includes(self))
120                   ->asSet();
121
122   -- This helper computes the inheritance subtree of the contextual Class.
123   -- For this purpose, the helper recursively calls itself for each subtype of
124   -- the contextual Class, adding this contextual Class to the computed result.
125   -- CONTEXT:    KM3!Class
126   -- RETURN:     Set(KM3!Class)
127   helper context KM3!Class def: getTree() : Set(KM3!Class) =
128           self.subTypes->iterate(e; tree : Set(KM3!Class) = Set{self} |
129                   tree->union( e.getTree() )
130           );
131
132   -- This helper computes a boolean value stating whether its exists an (or more)
```

```
133    -- inheritance relation between classes of trees t1 and t2.
134    -- The helper checks whether such a relation (supertype/subtype) exists between
135    -- each class of the tree t1 and the classes of the tree t2.
136    -- Comment sur les asSet()...
137    -- CONTEXT:    thisModule
138    -- IN:         Set(KM3!Class), Set(KM3!Class)
139    -- RETURN:     Boolean
140    helper def: existsHLink(t1 : Set(KM3!Class), t2 : Set(KM3!Class)) : Boolean =
141            t1->iterate(e; res : Boolean = false |
142                    if (e.supertypes->asSet()->intersection(t2)->isEmpty() and
143                            e.subTypes->asSet()->intersection(t2)->isEmpty())
144                    then
145                            res
146                    else
147                            true
148                    endif
149            );
150
151    -- This recursive helper computes the number of independant inheritance graphs
152    -- corresponding to the sequence of inheritance trees passed as a parameter.
153    -- If the input sequence contains a single tree, the helper returns 1.
154    -- Otherwise, the helper checks whether there exists inheritance relationships
155    -- (super/subtype) between classes of the first tree of the input sequence (the
156    -- reference tree), and the following ones. If so, it merges the linked trees
157    -- into the reference tree.
158    -- If the new tree sequence built this way contains a single tree, the helper
159    -- returns 1. Otherwise, the helper returns 1 + the value provided by a
160    -- recursive call of itself on the newly calculated tree sequence without its
161    -- first reference tree.
162    -- CONTEXT:    thisModule
163    -- IN:         Sequence(Set(KM3!Class))
164    -- RETURN:     Integer
165    helper def: computeGraphNb(tree_seq : Sequence(Set(KM3!Class))) : Integer =
166            if tree_seq->size() = 1
167            then
168                    1
169            else
170                    let first_t : Set(KM3!Class) = tree_seq->first() in
171                    let new_seq : Sequence(Set(KM3!Class)) =
172                    tree_seq
173                            ->subSequence(2, tree_seq->size())
174                            ->iterate(e;t_seq : Sequence(Set(KM3!Class)) = Sequence{first_t} |
175                                    if thisModule.existsHLink(first_t, e)
176                                    then
177                                            t_seq
178                                                    ->subSequence(2, t_seq->size())
179                                                    ->prepend(t_seq->first()->including(e))
180                                    else
181                                            t_seq.append(e)
182                                    endif
183                            )
184                    in
185                    if new_seq->size() = 1
186                    then
187                            1
188                    else
189                            thisModule.computeGraphNb(
190                                    new_seq->subSequence(2, new_seq->size())
191                            ) + 1
192                    endif
193            endif;
194
195    -- This helper returns the number of inheritage graphs in the input model.
196    -- For this purpose, it first computes a sequence containing the set of classes
197    -- representing the different inheritance trees of the input model. The helper
198    -- then calls the recursive computeGraphNb with the calculated sequence as a
199    -- parameter.
200    -- CONTEXT:    thisModule
201    -- RETURN:     Integer
```

```
202   helper def: getInheritanceGraphNb() : Integer =
203           let tree_seq : Sequence(Set(KM3!Class)) =
204                   thisModule.inheritanceRoots
205                           ->collect(e | e.getTree())
206                           ->asSequence()
207           in thisModule.computeGraphNb(tree_seq);
208
209
210   ---------------------------------------------------------------------------------
211   -- RULES --------------------------------------------------------------------------
212   ---------------------------------------------------------------------------------
213
214   -- Rule 'Model'
215   -- This rule generates metrics elements that are relative to the input model:
216   -- * the number of classes of the model;
217   -- * the number of associations of the model (this number corresponds to the
218   --   number of references without an opposite, plus the half of the number of
219   --   references that has an opposite);
220   -- * the maximal depth of an inheritance tree (computed by the
221   --   getInheritanceMaxDepth helper);
222   -- * the number of inheritance trees (computed by the getInheritanceTreeNb
223   --   helper);
224   -- * the number of inheritance graphs (computed by the getInheritanceGraphNb
225   --   helper).
226   rule Model {
227           from
228                   i : KM3!Metamodel
229           to
230                   o1 : Metrics!IntegerMetric (
231                           "context" <- 'Model',
232                           label <- 'Number of classes',
233                           data <- thisModule.allClasses->size()
234                   ),
235
236                   o2 : Metrics!IntegerMetric (
237                           "context" <- 'Model',
238                           label <- 'Number of associations',
239                           data <-
240                                   (thisModule.allReferences
241                                           ->select(e | e.opposite = OclUndefined)
242                                           ->size())
243                                   +
244                                   ((thisModule.allReferences
245                                           ->select(e | e.opposite <> OclUndefined)
246                                           ->size()) div 2)
247                   ),
248
249                   o3 : Metrics!IntegerMetric (
250                           "context" <- 'Model',
251                           label <- 'Maximal depth of heritance',
252                           data <- thisModule.getInheritanceMaxDepth()
253                   ),
254
255                   o4 : Metrics!IntegerMetric (
256                           "context" <- 'Model',
257                           label <- 'Number of inheritance trees',
258                           data <- thisModule.getInheritanceTreeNb()
259                   ),
260
261                   o5 : Metrics!IntegerMetric (
262                           "context" <- 'Model',
263                           label <- 'Number of inheritance graphs',
264                           data <- thisModule.getInheritanceGraphNb()
265                   )
266   }
267
268   -- Rule 'Class'
269   -- This rule generates metrics elements taht are relative to a Class of the
270   -- input model. The context of the generated metrics elements therefore
```

```
271    -- includes the name of the class theyr refer to.
272    -- Generated metrics are:
273    -- * the number of own attributes of the class;
274    -- * the number of own references of the class;
275    -- * the total number of attributes of the class (including inherited ones);
276    -- * the total number of references of the class (including inherited ones);
277    rule Class {
278            from
279                    i : KM3!Class
280            to
281                    o1 : Metrics!IntegerMetric (
282                            "context" <- 'Class ' + i.name,
283                            label <- 'Number of own attributes',
284                            data <- i.attributeNb
285                    ),
286
287                    o2 : Metrics!IntegerMetric (
288                            "context" <- 'Class ' + i.name,
289                            label <- 'Number of own references',
290                            data <- i.referenceNb
291                    ),
292
293                    o3 : Metrics!IntegerMetric (
294                            "context" <- 'Class ' + i.name,
295                            label <- 'Total number of attributes',
296                            data <- i.attributeNb2
297                    ),
298
299                    o4 : Metrics!IntegerMetric (
300                            "context" <- 'Class ' + i.name,
301                            label <- 'Total number of references',
302                            data <- i.referenceNb2
303                    )
304    }
```