1. ATL Transformation Example

1.1. Example: KM3 → Problem

The KM3 to Problem example describes a transformation from a KM3 metamodel [1] into a Problem model. The generated Problem model contains the list of non-structural errors (along with additional warnings) that have been detected within the input KM3 metamodel. The transformation assumes the input KM3 metamodel is structurally correct, as those that have passed a syntactic analysis (for instance, a reference defined with cardinality [1-1] should not be undefined). It may therefore fails when executed on a KM3 metamodel produced from a MOF metamodel that has not been checked.

The input metamodel is based on the KM3 metamodel. It is therefore a KM3 metamodel described by means of the KM3 semantics. The output model is based on the Problem metamodel.

This ATL transformation is based on initial works dealing with model checking with the ATL transformation language [2].

1.2. Metamodels

The KM3 to Problem transformation is based on two distinct metamodels, KM3 and Problem, that are described in the following subsections.

1.2.1. 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 I.

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.

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).

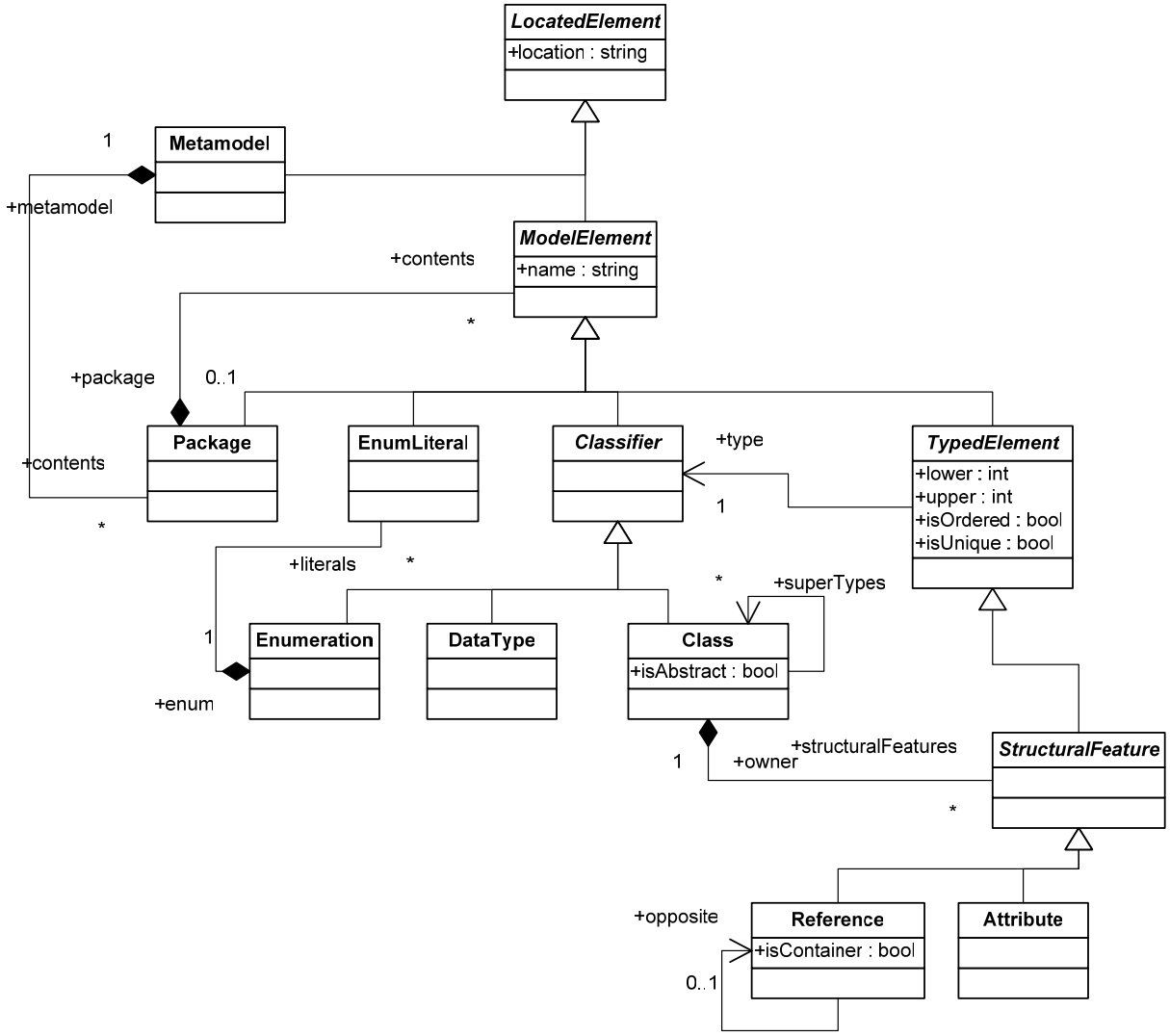

Figure 1. The KM3 metamodel

1.2.1.1. Additional constraints

Figure 1 defines a number of structural constraints on KM3 metamodels. However, in the same way additional constraints can be specified on a MOF metamodel [3] by means of the OCL language [4], KM3 metamodels have to respect a set of non-structural additional constraints.

We describe here the non-structural constraints that have to be respected by KM3 metamodels:

- A Package *name* has to be universally unique.
- A Classifier has to belong to a Package.
- An EnumLiteral has to belong to a Package.
- A Classifier *name* has to be unique within the Package it belongs to.
- A Package can only contain Package and Classifier elements through its *contents* reference.

- A Class is not allowed to be its own direct or indirect supertype.
- A StructuralFeature must be contained by a Class (through the *structuralFeatures* reference), and not by a Package (through its *contents* reference).
- The *name* of a StructuralFeature has to be unique in the Class it belongs to, as well as in the *supertypes* of this Class.
- The *opposite* of the *opposite* of a Reference has to be defined.
- The *opposite* of the *opposite* of a Reference has to be the Reference itself.
- The *type* of the *opposite* of a Reference has to be the *owner* of the Reference.
- The *lower* attribute of a TypedElement cannot be lower than 0.
- The *upper* attribute of a TypedElement has to be unbounded or greater or equal to than 1.
- The *upper* attribute of a TypedElement cannot be lower than its *lower* attribute.
- The *isOrdered* attribute of a TypedElement cannot be true if the *upper* value is 1.
- The *type* of a Reference must be a Class.

1.2.2. The Problem metamodel

The Problem metamodel provides semantics enabling to define, and describe, different kinds of problems (“error”, “warning”, and “critic”). In the scope of the KM3 to Problem transformation, it is used to encode the semantic errors, as well as some warnings and critics, that can be detected over the input KM3 metamodel. Figure 2 provides a description of the Problem metamodel. Its corresponding textual description in the KM3 format is also provided in Appendix II.

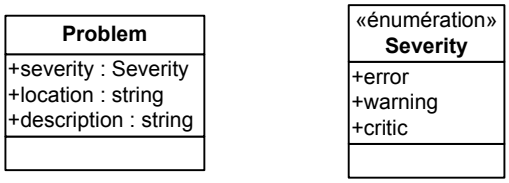

Figure 2. The Problem metamodel

A Problem model corresponds to a set of Problem elements. Each Problem is characterized by a *severity*, a *location* and a *description*. *severity* is of the Severity enumeration type, and can accept “error”, “warning”, and “critic” as value. The *location* and the *description* are both string attributes. The *location* attribute aims to encode the localisation of the Problem in the source file, whereas *description* provides a textual and human-readable description of the Problem.

1.3. An example

The KM3 to Problem transformation is embedded in the KM3 plug-in of the ATL Development Tools (ADT) [5]. It enables to ensure that KM3 non-structural constraints are verified on developed KM3 metamodels. Figure 3 provides an example of this KM3 metamodels development tool.

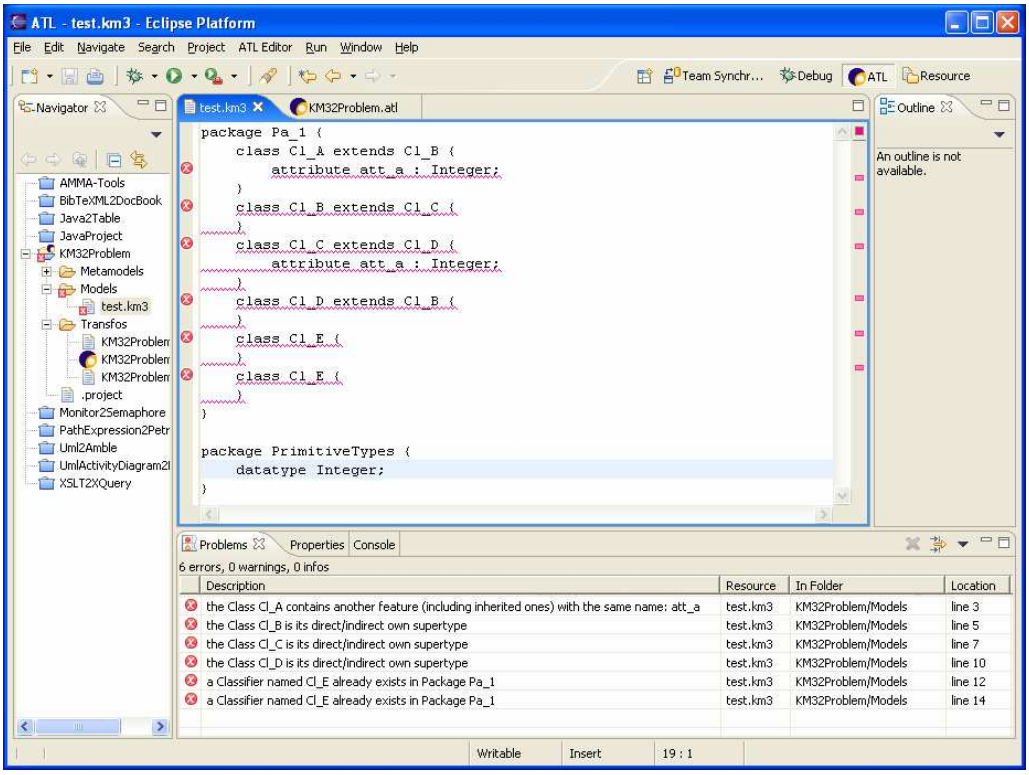

Figure 3. KM3 Problem detection example

The developed metamodel ("test.km3") is composed of a single Package ("Pa_1") that contains 6 classes. This example makes it possible to illustrate different kinds of non-structural errors:

- Class Cl_A defines an attribute ("att_a") that has the same name than an existing attribute of Class Cl_C from which Cl_A indirectly inherits.
- There exists a cycling inheritance definition between Classes Cl_B, Cl_C, and Cl_D (this implies that each one of the involved Classes is its indirect own supertype).
- Two Classes of Package Pa_1 have the same name ("Cl_E").

Note that for each error, the information generated by the KM3 to Problem transformation is displayed in the Problems tab of the windows. This information includes a graphical representation of the Problem type (in this example, we only deal with errors), the description of the Problem, and its location.

1.4. Rules Specification

The KM3 to Problem transformation defines a rule for each type of generated Problem.

Here are the Problems that are currently handled by the KM3 to Problem transformation:

- An error Problem is generated for each Package whose *name* is not unique.
- An error Problem is generated for each Classifier which is not defined within a Package.
- An error Problem is generated for each EnumLiteral which is not defined within a Package.

- An error Problem is generated for each StructuralFeature which is not contained by the *structuralFeatures* reference of the Class entity.
- An error Problem is generated for each Reference whose *opposite* of the *opposite* is not defined.
- An error Problem is generated for each Reference whose *opposite* of the *opposite* is different from itself.
- An error Problem is generated for each Reference whose *type* of the *opposite* is different from the reference's *owner* (i.e. the Class in which it is defined).
- An error Problem is generated for each Classifier whose *name* is not unique in its Package.
- An error Problem is generated for each Class which is its direct or indirect supertype.
- An error Problem is generated for each StructuralFeature whose *name* is not unique in its Class and its supertypes.
- An error Problem is generated for each StructuralFeature whose *lower* value is lower than 0.
- An error Problem is generated for each StructuralFeature whose *upper* value is lower than 1.
- An error Problem is generated for each StructuralFeature whose *upper* value is lower than the *upper* one.
- An error Problem is generated for each StructuralFeature with an *upper* value equal to 1 and the *isOrdered* attribute set to true.
- An error Problem is generated for each Reference that points either to a DataType or an Enumeration element.
- A warning Problem is generated for each Attribute of type Class. Class attributes are indeed supported by the KM3 [1] and MOF 1.4 [3] metamodels, but not by Ecore [6].
- A warning Problem is generated for each abstract Class that has no child.
- A critic Problem is generated for each Classifier whose *name* does not start by an upper case character.

1.5. ATL Code

The ATL code for the KM3 to Problem transformation consists of 6 helpers and 18 rules.

1.5.1. Helpers

The first two helpers are constant helpers, **allPackages**, and **allClasses**, are constant helpers. They simply compute sequences of input model elements (respectively Packages and Classes) that are referred to several times in the transformation. This step makes it possible to save calculations by storing the content of these different sequences into constant helpers.

The remaining four helpers are function helpers. The **allStructuralFeaturesRec(Sequence(KM3!Class))** helper aims to compute the set of all the direct and inherited StructuralFeatures of the contextual Class. It accepts a Sequence of Class as parameter. This Sequence contains the list of Class elements that have already been visited by previous recursive calls and that are not considered anymore (to avoid cycles). The helper first gets

the direct StructuralFeatures of the contextual Class, and performs the union between these StructuralFeatures and those of the supertypes of the contextual Class that have not been already visited (i.e. that do not belong to the Sequence provided as parameter). The **allStructuralFeatures()** helper aims to compute the set of all the direct and inherited StructuralFeatures of the contextual Class. To this end, it simply calls the **allStructuralFeaturesRec()** helper, passing to it a Sequence containing the contextual Class as the list of already visited elements.

The **recursiveInheritanceRec(KM3!Class, Sequence(KM3!Class))** helper aims to compute a Boolean stating whether a recursive inheritance exists for the contextual Class. The helper accepts two parameters: a Class that corresponds to the initial Class, the one for which an inheritance cycle is sought, and a Sequence of Classes that contains the Class elements that have been already visited by previous recursive calls. Note that the helper only looks for inheritance cycles in which the initial Class is involved, but not those that may exist for its supertypes. The helper first checks whether the contextual Class has supertypes. If it does not, it returns false. It the contextual Class has some supertypes, and that the initial Class belongs to them, it returns true. Otherwise, it visits all the supertypes of the contextual Class that have not been yet visited (those that do not belong to the Sequence passed as parameter), and checks whether a recursive inheritance exists between each of them and the reference initial Class. Finally, the **recursiveInheritance()** helper aims to compute a Boolean value stating whether a direct or indirect recursive inheritance is defined for the contextual Class. For this purpose, the helper calls the **recursiveInheritanceRec()** helper, passing to it the contextual Class as the reference Class (for inheritance cycle definition) and an empty Sequence as the list of already visited elements.

1.5.2. Rules

Besides helpers, the Monitor to Semaphore transformation is composed of 17 rules.

The rule **PackageNameUnique** generates an error Problem for each Package whose *name* is not unique. For this purpose, it matches a Package when there exists another Package, distinct from the input one, that has the same name that the input one.

The rule **ClassifierInPackage** generates an error Problem for each Classifier that is not contained by a Package. Thus, it matches a Classifier when its *package* reference is undefined.

The rule **EnumLiteralInPackage** generates an error Problem for each EnumLiteral that is not contained by a Package. Thus, it matches an EnumLiteral when its *package* reference is undefined.

The rule **StructuralFeatureInClass** generates an error Problem for each StructuralFeature that is not contained by a Class. To this end, it matches a StructuralFeature if its *package* reference is not undefined (which means that the StructuralFeature is contained by the Package instead of being contained by a Class).

The rule **OppositeOfOppositeExists** generates an error Problem for each Reference whose *opposite* of the *opposite* is undefined. For this purpose, the rule matches each Reference:

- That has an *opposite* Reference.
- For which the *opposite* of this *opposite* is not defined (this test is performed by the OCL function *oclUndefined()*).

The rule **OppositeOfOppositeIsSelf** generates an error Problem for each Reference whose *opposite* of the *opposite* exists and is different from itself. For this purpose, the rule matches each Reference:

- That has an *opposite* Reference.
- For which an *opposite* of this *opposite* is defined.

- For which the *opposite* of this *opposite* does not point to itself.

The rule **TypeOfOppositeIsOwner** generates an error Problem for each Reference whose *type* of the *opposite* does not point to the Class that contains the Reference (it may, for instance, point to a supertype of this Class). The rule matches each Reference:

- That has an *opposite* Reference.
- For which the *type* of this *opposite* is different from the *owner* of the Reference.

The rule **ClassifierNameUniqueInPackage** generates an error Problem for each Classifier whose *name* is not unique in the Package it belongs to. To this end, the rule matches a Classifier if there exists another Classifier in its Package that has the same name.

The rule **ClassIsNotItsOwnSuperType** generates an error Problem for each Class which is its direct or indirect supertype. To this end, the rule matches each Class for which the **recursiveInheritance()** helper returns true.

The rule **StructuralFeatureNameUniqueInClass** generates an error Problem for each StructuralFeature whose name is not unique in its Class and its supertypes. For this purpose, the rule matches a StructuralFeature when there exists, in the StructuralFeatures sequence returned by the **allStructuralfeatures()** helper, another StructuralFeature that has the same *name*.

The rule **StructuralFeatureLower** generates an error Problem for each StructuralFeature whose *lower* attribute is lower than 0.

The rule **StructuralFeatureUpper** generates an error Problem for each StructuralFeature whose *upper* attribute is lower than 1 or different from 1 (which is used to encode an unbounded value).

The rule **StructuralFeatureLowerUpper** generates an error Problem for each StructuralFeature whose *upper* attribute is lower than its *lower* attribute.

The rule **StructuralFeatureUniqueOrdered** generates an error Problem for each StructuralFeature whose *upper* value is 1 and whose *isOrdered* attribute is true.

The rule **DataTypeReferenceProhibited** generates an error Problem for each Reference that does not point to a Class element. To this end, the rule matches each Reference whose *type* attribute does not target a Class.

Since the Ecore metamodel [6] does not provide support for attributes of type Class, the rule **ClassAttributeUnsupportedByEcore** generates a warning Problem for each Attribute that points to a Class element. To this end, the rule matches each Attribute whose *type* attribute targets a Class.

The rule **AbstractClassShouldHaveChildren** generates a warning Problem for each abstract Class which is the supertype of no other classes. For this purpose, the rule matches each Class whose *isAbstract* attribute is true, and for which there exists no Classes that have the input Class among its set of *supertypes*.

Finally, the rule **ClassifierNameShouldStartWithUpperCase** generates a critic Problem for each Classifier whose *name* does not start by an upper case character.

```
1  module KM32Problem;
2  create OUT : Problem from IN : KM3;
3
4  ------------------------------------------------------------------------
5  -- HELPERS -----------------------------------------------------------
6  ------------------------------------------------------------------------
7
```

```
8   -- This helper computes a Sequence containing all the Packages of the input
9   -- model.
10  -- CONTEXT:  thisModule
11  -- RETURN: Sequence(KM3!Package)
12  helper def: allPackages : Sequence(KM3!Package) =
13     KM3!Package.allInstances()->asSequence();
14
15  -- This helper computes a Sequence containing all the Classes of the input
16  -- model.
17  -- CONTEXT:  thisModule
18  -- RETURN: Sequence(KM3!Class)
19  helper def: allClasses : Sequence(KM3!Class) =
20     KM3!Class.allInstances()->asSequence();
21
22
23  -- This helper computes a Sequence that contains all the direct and inherited
24  -- StructuralFeatures of the contextual Class. The Sequence s which is also
25  -- passed as a parameter contains the KM3!Class elements that have already been
26  -- visited by the recursive process and that are not considered anymore.
27  -- The helper collects the direct StructuralFeatures of the contextual
28  -- Class and, by means of recursice calls, the ones of those of its supertypes
29  -- that do not already belong to the Sequence s.
30  -- CONTEXT:  KM3!Class
31  -- IN:     Sequence(KM3!Class)
32  -- RETURN: Sequence(KM3!StructuralFeature)
33  helper context KM3!Class
34     def: allStructuralFeaturesRec(s : Sequence(KM3!Class)) :
35                                 Sequence(KM3!StructuralFeature) =
36     self.structuralFeatures->union(
37        self.supertypes->iterate(e; res : Sequence(KM3!Class) = Sequence{} |
38           if s->includes(e)
39           then
40              res
41           else
42              res->union( e.allStructuralFeaturesRec(s->append(e)) )
43           endif
44        )
45     );
46
47  -- This helper computes a Sequence that contains all the direct and inherited
48  -- StructuralFeatures of the contextual Class.
49  -- To this end, the helper calls the allStructuralFeaturesRec helper with
50  -- an Sequence (containing the contextual Class) as parameter.
51  -- CONTEXT:  KM3!Class
52  -- RETURN: Sequence(KM3!StructuralFeature)
53  helper context KM3!Class
54     def: allStructuralFeatures() : Sequence(KM3!StructuralFeature) =
55     self.allStructuralFeaturesRec(Sequence{self});
56
57  -- This helper computes a Boolean value stating whether a direct or indirect
58  -- inheritance is defined from the contextual Class to the Class c passed as a
59  -- parameter. The Sequence s which is also passed as a parameter contains the
60  -- KM3!Class elements that have already been visited by the recursive process
61  -- (except the c element) and that are not consdidered anymore.
62  -- To this end, the helper successively tests its own supertypes, and the
63  -- supertypes of its supertypes (by means of a recursive call):
64  --   * if the contextual Class has no supertype, the helper returns false.
65  --   * if the Class passed as a parameter is a supertype of the contextual
66  --     Class, the helper returns true.
67  --   * otherwise, the helper returns the disjunction of the recursive calls of
68  --     the helper on each of its supertypes that has not been already visited
69  --     by the recursive process.
```

```
70    -- CONTEXT:  KM3!Class
71    -- IN:     KM3!Class
72    -- IN:     Sequence(KM3!Class)
73    -- RETURN: Boolean
74    helper context KM3!Class
75       def: recursiveInheritanceRec(c : KM3!Class,
76                            s : Sequence(KM3!Class)) : Boolean =
77       if self.supertypes->isEmpty()
78       then
79          false
80       else
81          if self.supertypes->exists(e | e = c)
82          then
83             true
84          else
85             self.supertypes->iterate(e; res : Boolean = false |
86                if s->includes(e)
87                then
88                   res
89                else
90                   res or e.recursiveInheritanceRec(c, s->append(e))
91                endif
92             )
93          endif
94       endif;
95
96    -- This helper computes a Boolean value stating whether a direct or indirect
97    -- recursive inheritance is defined for the contextual Class.
98    -- To this end, the helper calls the recursiveInheritanceRec helper with
99    -- itself, and an empty Sequence as parameters.
100   -- CONTEXT:  KM3!Class
101   -- RETURN: Boolean
102   helper context KM3!Class def: recursiveInheritance() : Boolean =
103      self.recursiveInheritanceRec(self, Sequence{});
104
105
106   ------------------------------------------------------------------------------
107   -- RULES -----------------------------------------------------------------
108   ------------------------------------------------------------------------------
109
110   -- Rule 'PackageNameUnique'
111   -- This rule generates an 'error' Problem element for each Package whose name
112   -- is not unique.
113   rule PackageNameUnique {
114      from
115         i : KM3!Package (
116            thisModule.allPackages->exists(e | (i <> e) and (i.name = e.name))
117         )
118      to
119         o : Problem!Problem (
120            severity <- #error,
121            description <-
122                 'a Package of the same name already exists: '
123                 + i.name,
124            location <- i.location
125         )
126   }
127
128   -- Rule 'ClassifierInPackage'
129   -- This rule generates an 'error' Problem element for each Classifier which is
130   -- not defined within a Package.
131   rule ClassifierInPackage {
```

```
132      from
133         i : KM3!Classifier (
134            i.package.oclIsUndefined()
135         )
136      to
137         o : Problem!Problem (
138            severity <- #error,
139            description <-
140                 'the Classifier ' + i.name
141                 + ' must be contained by a Package',
142            location <- i.location
143         )
144   }
145
146   -- Rule 'EnumLiteralInPackage'
147   -- This rule generates an 'error' Problem element for each EnumLiteral which is
148   -- not defined within a Package.
149   rule EnumLiteralInPackage {
150      from
151         i : KM3!EnumLiteral (
152            i.package.oclIsUndefined()
153         )
154      to
155         o : Problem!Problem (
156            severity <- #error,
157            description <-
158                 'the EnumLiteral ' + i.name
159                 + ' must be contained by a Package',
160            location <- i.location
161         )
162   }
163
164   -- Rule 'StructuralFeatureInClass'
165   -- This rule generates an 'error' Problem element for each Classifier which is
166   -- not defined within a Class.
167   rule StructuralFeatureInClass {
168      from
169         i : KM3!StructuralFeature (
170            not i.package.oclIsUndefined()
171         )
172      to
173         o : Problem!Problem (
174            severity <- #error,
175            description <-
176                 'the Feature ' + i.name
177                 + ' cannot be contained by a Package',
178            location <- i.location
179         )
180   }
181
182   -- Rule 'OppositeOfOppositeExists'
183   -- This rule generates an 'error' Problem element for each Reference whose
184   -- opposite of the opposite is not defined.
185   rule OppositeOfOppositeExists {
186      from
187         i : KM3!Reference (
188            if i.opposite.oclIsUndefined()
189            then
190               false
191            else
192               i.opposite.opposite.oclIsUndefined()
193            endif
```

```
194         )
195      to
196         o : Problem!Problem (
197            severity <- #error,
198            description <-
199                 'the opposite of the opposite of Reference ' +
200                 i.owner.name + '::' + i.name +
201                 ' should be defined',
202            location <- i.location
203         )
204   }
205
206   -- Rule 'OppositeOfOppositeIsSelf'
207   -- This rule generates an 'error' Problem element for each Reference whose
208   -- opposite of the opposite is different from itself.
209   rule OppositeOfOppositeIsSelf {
210      from
211         i : KM3!Reference (
212            if i.opposite.oclIsUndefined()
213            then
214               false
215            else
216               if i.opposite.opposite.oclIsUndefined()
217               then
218                  false
219               else
220                  i.opposite.opposite <> i
221               endif
222            endif
223         )
224      to
225         o : Problem!Problem (
226            severity <- #error,
227            description <-
228                 'the opposite of the opposite of Reference ' +
229                 i.owner.name + '::' + i.name +
230                 ' should be this very same Reference',
231            location <- i.location
232         )
233   }
234
235   -- Rule 'TypeOfOppositeIsOwner'
236   -- This rule generates an 'error' Problem element for each Reference whose
237   -- type of the opposite is different from its owner.
238   rule TypeOfOppositeIsOwner {
239      from
240         i : KM3!Reference (
241            not
242               if i.opposite.oclIsUndefined() then
243                  true
244               else
245                  i.opposite.type = i.owner
246               endif
247         )
248      to
249         o : Problem!Problem (
250            severity <- #error,
251            description <-
252                 'the type of the opposite of Reference ' +
253                 i.owner.name + '::' + i.name +
254                 ' should be the owner of this Reference (' +
255                 i.owner.name + ')',
```

```
256             location <- i.location
257         )
258     }
259
260     -- Rule 'ClassifierNameUniqueInPackage'
261     -- This rule generates an 'error' Problem element for each Classifier whose
262     -- name is not unique within its Package.
263     rule ClassifierNameUniqueInPackage {
264        from
265           i : KM3!Classifier (
266              i.package.contents->exists(e | (i <> e) and (i.name = e.name))
267           )
268        to
269           o : Problem!Problem (
270              severity <- #error,
271              description <-
272                   'a Classifier named ' + i.name
273                   + ' already exists in Package '
274                   + i.package.name,
275              location <- i.location
276           )
277     }
278
279     -- Rule 'ClassIsNotItsOwnSupertype'
280     -- This rule generates an 'error' Problem element for each Class which is its
281     -- direct or indirect supertype.
282     rule ClassIsNotItsOwnSupertype {
283        from
284           i : KM3!Class (
285              i.recursiveInheritance()
286           )
287        to
288           o : Problem!Problem (
289              severity <- #error,
290              description <- 'the Class ' + i.name
291                        + ' is its direct/indirect own supertype',
292              location <- i.location
293           )
294     }
295
296     -- Rule 'StructuralFeatureNameUniqueInClass'
297     -- This rule generates an 'error' Problem element for each StructuralFeature
298     -- whose name is not unique within its Class (including inherited SFs).
299     rule StructuralFeatureNameUniqueInClass {
300        from
301           i : KM3!StructuralFeature (
302              i.owner.allStructuralFeatures()
303                 ->exists(e | (i <> e) and (i.name = e.name))
304           )
305        to
306           o : Problem!Problem (
307              severity <- #error,
308              description <-
309                   'the Class ' + i.owner.name
310                   + ' contains another feature (including inherited ones) '
311                   + 'with the same name: '
312                   + i.name,
313              location <- i.location
314           )
315     }
316
317     -- Rule 'StructuralFeatureLower'
```

```
318   -- This rule generates an 'error' Problem element for each StructuralFeature
319   -- whose lower attribute is lower than 0.
320   rule StructuralFeatureLower {
321      from
322         i : KM3!StructuralFeature (
323            i.lower < 0
324         )
325      to
326         o : Problem!Problem (
327            severity <- #error,
328            description <-
329                  'Lower bound value of Feature ' + i.owner.name + '::'
330                  + i.name + 'is unvalid (lower than 0)',
331            location <- i.location
332         )
333   }
334
335   -- Rule 'StructuralFeatureUpper'
336   -- This rule generates an 'error' Problem element for each StructuralFeature
337   -- whose upper attribute is lower than 1.
338   rule StructuralFeatureUpper {
339      from
340         i : KM3!StructuralFeature (
341            (i.upper < 1) and (i.upper <> 0-1)
342         )
343      to
344         o : Problem!Problem (
345            severity <- #error,
346            description <-
347                  'Upper bound of Feature ' + i.owner.name + '::'
348                  + i.name + 'is unvalid (lower than 1)',
349            location <- i.location
350         )
351   }
352
353   -- Rule 'StructuralFeatureLowerUpper'
354   -- This rule generates an 'error' Problem element for each StructuralFeature
355   -- whose upper attribute is lower than its upper attribute.
356   rule StructuralFeatureLowerUpper {
357      from
358         i : KM3!StructuralFeature (
359            (i.upper < i.lower) and (i.upper <> 0-1)
360         )
361      to
362         o : Problem!Problem (
363            severity <- #error,
364            description <-
365                  'Upper bound of Feature ' + i.owner.name + '::'
366                  + i.name + ' is lower than its lower bound',
367            location <- i.location
368         )
369   }
370
371   -- Rule 'StructuralFeatureUniqueOrdered'
372   -- This rule generates an 'error' Problem element for each StructuralFeature
373   -- whose upper attribute is 1 and isOrdered attribute is true.
374   rule StructuralFeatureUniqueOrdered {
375      from
376         i : KM3!StructuralFeature (
377            (i.upper = 1) and (i.isOrdered = true)
378         )
379      to
```

```
380         o : Problem!Problem (
381            severity <- #error,
382            description <-
383                 'Feature ' + i.owner.name + '::' + i.name
384                 + ' cannot be ordered with an upper bound equals to 1',
385            location <- i.location
386         )
387   }
388
389   -- Rule 'DataTypeReferenceProhibited'
390   -- This rule generates an 'error' Problem element for each Reference which
391   -- targets a Datatype element.
392   rule DataTypeReferenceProhibited {
393      from
394         i : KM3!Reference (
395            not i.type.oclIsTypeOf(KM3!Class)
396         )
397      to
398         o : Problem!Problem (
399            severity <- #error,
400            description <-
401                 'Reference ' + i.owner.name + '::' + i.name
402                 + ' cannot target a DataType element',
403            location <- i.location
404         )
405   }
406
407   -- Rule 'ClassAttributeUnsupportedByEcore'
408   -- This rule generates an 'warning' Problem element for each Attribute whose
409   -- type is Class.
410   rule ClassAttributeUnsupportedByEcore {
411      from
412         i : KM3!Attribute (
413            i.type.oclIsTypeOf(KM3!Class)
414         )
415      to
416         o : Problem!Problem (
417            severity <- #warning,
418            description <-
419                 'Class ' + i.owner.name + ' defines a class Attribute ('
420                 + i.name
421                 + ') that is not supported by the Ecore metamodel',
422            location <- i.location
423         )
424   }
425
426   -- Rule 'AbstractClassShouldHaveChildren'
427   -- This rule generates an 'error' Problem element for each abstract Class which
428   -- has no child.
429   rule AbstractClassShouldHaveChildren {
430      from
431         i : KM3!Class (
432            i.isAbstract and
433            (thisModule.allClasses
434               ->select(e | e.supertypes->includes(i))
435               ->isEmpty()
436             )
437         )
438      to
439         o : Problem!Problem (
440            severity <- #warning,
441            description <- 'the abstract Class ' + i.name + ' has no children',
```

```
442             location <- i.location
443         )
444     }
445
446     -- Rule 'ClassifierNameShouldStartWithUpperCase'
447     -- This rule generates an 'critic' Problem element for each Classifier whose
448     -- name does not start by an upper case character.
449     rule ClassifierNameShouldStartWithUpperCase {
450        from
451           i : KM3!Classifier (
452              let firstChar : String = i.name.substring(1, 1) in
453                 firstChar <> firstChar.toUpper()
454           )
455        to
456           o : Problem!Problem (
457              severity <- #critic,
458              description <-
459                   'the name of Classifier ' + i.name
460                   + ' should begin with an upper case',
461              location <- i.location
462           )
463     }
```

I. KM3 metamodel in KM3 format

```
package KM3 {
   abstract class LocatedElement {
      attribute location : String;
   }

   abstract class ModelElement extends LocatedElement {
      attribute name : String;
      reference "package" : Package oppositeOf contents;
   }

   class Classifier extends ModelElement {}

   class DataType extends Classifier {}

   class Enumeration extends Classifier {  -- extends DataType in Ecore but if so,
cannot use an abstract template in TCS
      reference literals[*] ordered container : EnumLiteral oppositeOf enum;
   }

   class EnumLiteral extends ModelElement {
      reference enum : Enumeration oppositeOf literals;
   }

-- WARNING, ONLY FOR OCL Standard Library
   class TemplateParameter extends  Classifier {
   }
-- End WARNING

   class Class extends Classifier {
-- WARNING, ONLY FOR OCL Standard Library
      reference parameters[*] ordered container : TemplateParameter;
-- End WARNING

      attribute isAbstract : Boolean;
      reference supertypes[*] : Class;
      reference structuralFeatures[*] ordered container : StructuralFeature
oppositeOf owner;
      reference operations[*] ordered container : Operation oppositeOf owner;
   }

   class TypedElement extends ModelElement {
      attribute lower : Integer;
      attribute upper : Integer;
      attribute isOrdered : Boolean;
      attribute isUnique : Boolean;
      reference type : Classifier;
   }

   class StructuralFeature extends TypedElement {
      reference owner : Class oppositeOf structuralFeatures;
      reference subsetOf[*] : StructuralFeature oppositeOf derivedFrom;
      reference derivedFrom[*] : StructuralFeature oppositeOf subsetOf;
   }

   class Attribute extends StructuralFeature {}

   class Reference extends StructuralFeature {
      attribute isContainer : Boolean;
      reference opposite[0-1] : Reference;
```

```
   }

   class Operation extends TypedElement {
      reference owner : Class oppositeOf operations;
      reference parameters[*] ordered container : Parameter oppositeOf owner;
   }

   class Parameter extends TypedElement {
      reference owner : Operation oppositeOf parameters;
   }

   class Package extends ModelElement {
      reference contents[*] ordered container : ModelElement oppositeOf "package";
      reference metamodel : Metamodel oppositeOf contents;
   }

   class Metamodel extends LocatedElement {
      reference contents[*] ordered container : Package oppositeOf metamodel;
   }
}

package PrimitiveTypes {
   datatype Boolean;
   datatype Integer;
   datatype String;
}
```

II. Problem metamodel in KM3 format

```
package Problem {

   enumeration Severity {
      literal error;
      literal warning;
      literal critic;
   }

   class Problem {
      attribute severity: Severity;
      attribute location: String;
      attribute description: String;
   }
}

package PrimitiveTypes {
   datatype String;
}
```

References

[1] KM3: Kernel MetaMetaModel. Available at http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/doc/atl/index.html.

[2] Bézivin, J, and Jouault, F. Using ATL for Checking Models. To appear in Proceedings of the International Workshop on Graph and Model Transformation (GraMoT), September 2005, Tallinn, Estonia.

[3] OMG/MOF. *Meta Object Facility (MOF)*, v1.4. OMG Document formal/02-04-03, April 2002. Available from www.omg.org.

[4] OMG/OCL Specification, ptc/03-10-14. October 2003. Available from www.omg.org.

[5] F. Allilaire, and T. Idrissi. ADT: Eclipse development tools for ATL. In Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA-2), September 2004, Canterbury, England.

[6] F. Budinsky, and D. Steinberg, and E. Merks, and R. Ellersick, and T. J. Grose: Eclipse Modeling Framework, Chapter 5 *Ecore Modeling Concepts*, Addison-Wesley.