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 fail 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 define 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).
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.

```
+severity : Severity
+location : string
+description : string

Problem

«énumération» Severity
+error
+warning
+critic
```

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.
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 \texttt{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 \texttt{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.

```plaintext
module KM32Problem;
create OUT : Problem from IN : KM3;

-- HELPERS ---------------------------------------------------------------
```

---
-- This helper computes a Sequence containing all the Packages of the input
-- model.
-- CONTEXT: thisModule
-- RETURN: Sequence(KM3!Package)
helper def: allPackages : Sequence(KM3!Package) =
    KM3!Package.allInstances()->asSequence();

-- This helper computes a Sequence containing all the Classes of the input
-- model.
-- CONTEXT: thisModule
-- RETURN: Sequence(KM3!Class)
helper def: allClasses : Sequence(KM3!Class) =
    KM3!Class.allInstances()->asSequence();

-- This helper computes a Sequence that contains all the direct and inherited
-- StructuralFeatures of the contextual Class. The Sequence s which is also
-- passed as a parameter contains the KM3!Class elements that have already been
-- visited by the recursive process and that are not considered anymore.
-- The helper collects the direct StructuralFeatures of the contextual
-- Class and, by means of recursive calls, the ones of those of its supertypes
-- that do not already belong to the Sequence s.
-- CONTEXT: KM3!Class
-- IN: Sequence(KM3!Class)
-- RETURN: Sequence(KM3!StructuralFeature)
helper context KM3!Class
  def: allStructuralFeaturesRec(s : Sequence(KM3!Class)) : 
    Sequence(KM3!StructuralFeature) =
    self.structuralFeatures->union(
        self.supertypes->iterate(e; res : Sequence(KM3!Class) = Sequence{} |
            if s->includes(e)
            then res
            else res->union( e.allStructuralFeaturesRec(s->append(e)) )
        )
    );

-- This helper computes a Sequence that contains all the direct and inherited
-- StructuralFeatures of the contextual Class.
-- To this end, the helper calls the allStructuralFeaturesRec helper with
-- an Sequence (containing the contextual Class) as parameter.
-- CONTEXT: KM3!Class
-- RETURN: Sequence(KM3!StructuralFeature)
helper context KM3!Class
  def: allStructuralFeatures() : Sequence(KM3!StructuralFeature) =
    self.allStructuralFeaturesRec(Sequence{self});

-- This helper computes a Boolean value stating whether a direct or indirect
-- inheritance is defined from the contextual Class to the Class c passed as a
-- parameter. The Sequence s which is also passed as a parameter contains the
-- KM3!Class elements that have already been visited by the recursive process
-- (except the c element) and that are not considered anymore.
-- To this end, the helper successively tests its own supertypes, and the
-- supertypes of its supertypes (by means of a recursive call):
-- * if the contextual Class has no supertype, the helper returns false.
-- * if the Class passed as a parameter is a supertype of the contextual
--   Class, the helper returns true.
-- * otherwise, the helper returns the disjunction of the recursive calls of
--   the helper on each of its supertypes that has not been already visited
-- by the recursive process.
KT

-- CONTEXT: KM3!Class
-- IN: KM3!Class
-- IN: Sequence(KM3!Class)
-- RETURN: Boolean
helper context KM3!Class
def recursiveInheritanceRec(c : KM3!Class,
                      s : Sequence(KM3!Class)) : Boolean =
    if self.supertypes->isEmpty()
then
    false
else
    if self.supertypes->exists(e | e = c)
then
    true
else
    self.supertypes->iterate(e; res : Boolean = false |
      if s->includes(e)
    then
    res
    else
    res or e.recursiveInheritanceRec(c, s->append(e))
    endif
  )
endif
end;
-- This helper computes a Boolean value stating whether a direct or indirect
-- recursive inheritance is defined for the contextual Class.
-- To this end, the helper calls the recursiveInheritanceRec helper with
-- itself, and an empty Sequence as parameters.
helper context KM3!Class def: recursiveInheritance() : Boolean =
  self.recursiveInheritanceRec(self, Sequence());
-- RULES
-- Rule 'PackageNameUnique'
-- This rule generates an 'error' Problem element for each Package whose name
-- is not unique.
rule PackageNameUnique {
  from
  i : KM3!Package (
         thisModule.allPackages->exists(e | (i <> e) and (i.name = e.name))
  )
  to
    o : Problem!Problem {
    severity <- error,
    description <-
      'a Package of the same name already exists: ' + i.name,
    location <- i.location
  }
}
-- Rule 'ClassifierInPackage'
-- This rule generates an 'error' Problem element for each Classifier which is
-- not defined within a Package.
rule ClassifierInPackage {
from i : KM3!Classifier ( i.package.oclIsUndefined() )

to o : Problem!Problem {
  severity <- #error,
  description <-
    'the Classifier ' + i.name
    + ' must be contained by a Package',
  location <- i.location
}

-- Rule 'EnumLiteralInPackage'
-- This rule generates an 'error' Problem element for each EnumLiteral which is
-- not defined within a Package.
rule EnumLiteralInPackage {
  from i : KM3!EnumLiteral ( i.package.oclIsUndefined() )
  to o : Problem!Problem {
    severity <- #error,
    description <-
      'the EnumLiteral ' + i.name
      + ' must be contained by a Package',
    location <- i.location
  }
}

-- Rule 'StructuralFeatureInClass'
-- This rule generates an 'error' Problem element for each Classifier which is
-- not defined within a Class.
rule StructuralFeatureInClass {
  from i : KM3!StructuralFeature ( not i.package.oclIsUndefined() )
  to o : Problem!Problem {
    severity <- #error,
    description <-
      'the Feature ' + i.name
      + ' cannot be contained by a Package',
    location <- i.location
  }
}

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

256       location <- i.location
257     }
258   }
259   -- Rule 'ClassifierNameUniqueInPackage'
260   -- This rule generates an 'error' Problem element for each Classifier whose
261   -- name is not unique within its Package.
262   rule ClassifierNameUniqueInPackage {
263     from
264     i : KM3!Classifier (i.package.contents->exists(e | (i <> e) and (i.name = e.name))
265     )
266   to
267     o : Problem!Problem (severity <- #error,
268       description <- 'a Classifier named ' + i.name
269       + ' already exists in Package ' + i.package.name,
270       location <- i.location
271     )
272   }
273   -- Rule 'ClassIsNotItsOwnSupertype'
274   -- This rule generates an 'error' Problem element for each Class which is its
275   -- direct or indirect supertype.
276   rule ClassIsNotItsOwnSupertype {
277     from
278     i : KM3!Class (i.recursiveInheritance())
279     )
280   to
281     o : Problem!Problem (severity <- #error,
282       description <- 'the Class ' + i.name
283       + ' is its direct/indirect own supertype',
284       location <- i.location
285     )
286   }
287   -- Rule 'StructuralFeatureNameUniqueInClass'
288   -- This rule generates an 'error' Problem element for each StructuralFeature
289   -- whose name is not unique within its Class (including inherited SFs).
290   rule StructuralFeatureNameUniqueInClass {
291     from
292     i : KM3!StructuralFeature (i.owner.allStructuralFeatures()->exists(e | (i <> e) and (i.name = e.name))
293     )
294   to
295     o : Problem!Problem (severity <- #error,
296       description <- 'the Class ' + i.owner.name
297       + ' contains another feature (including inherited ones) ' + 'with the same name: ' + i.name,
298       location <- i.location
299     )
300   }
301   -- Rule 'StructuralFeatureLower'
```plaintext
-- This rule generates an 'error' Problem element for each StructuralFeature
-- whose lower attribute is lower than 0.
rule StructuralFeatureLower {
  from
    i : KM3!StructuralFeature (i.lower < 0)
  to
    o : Problem!Problem (severity <- #error,
                      description <-
                        'Lower bound value of Feature ' + i.owner.name + '::'
                        + i.name + ' is unvalid (lower than 0)',
                      location <- i.location)
}

-- Rule 'StructuralFeatureUpper'
-- This rule generates an 'error' Problem element for each StructuralFeature
-- whose upper attribute is lower than 1.
rule StructuralFeatureUpper {
  from
    i : KM3!StructuralFeature (i.upper < 1 and i.upper <> 0-1)
  to
    o : Problem!Problem (severity <- #error,
                      description <-
                        'Upper bound of Feature ' + i.owner.name + '::'
                        + i.name + ' is unvalid (lower than 1)',
                      location <- i.location)
}

-- Rule 'StructuralFeatureLowerUpper'
-- This rule generates an 'error' Problem element for each StructuralFeature
-- whose upper attribute is lower than its upper attribute.
rule StructuralFeatureLowerUpper {
  from
    i : KM3!StructuralFeature (i.upper < i.lower and i.upper <> 0-1)
  to
    o : Problem!Problem (severity <- #error,
                      description <-
                        'Upper bound of Feature ' + i.owner.name + '::'
                        + i.name + ' is lower than its lower bound',
                      location <- i.location)
}

-- Rule 'StructuralFeatureUniqueOrdered'
-- This rule generates an 'error' Problem element for each StructuralFeature
-- whose upper attribute is 1 and isOrdered attribute is true.
rule StructuralFeatureUniqueOrdered {
  from
    i : KM3!StructuralFeature (i.upper = 1 and i.isOrdered = true)
  to
```
ATL
Transformation Example

KM3 to Problem

Date 04/08/2005

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

-- Rule 'ClassifierNameShouldStartWithUpperCase'
-- This rule generates an 'critic' Problem element for each Classifier whose
-- name does not start by an upper case character.
rule ClassifierNameShouldStartWithUpperCase {
  from
    i : KM3!Classifier {
      let firstChar : String = i.name.substring(1, 1) in
      firstChar <> firstChar.toUpper()
    }
  to
    o : Problem!Problem {
      severity <- #critic,
      description <-
      'the name of Classifier ' + i.name
        + ' should begin with an upper case',
      location <- i.location
    }
}
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 {
        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


