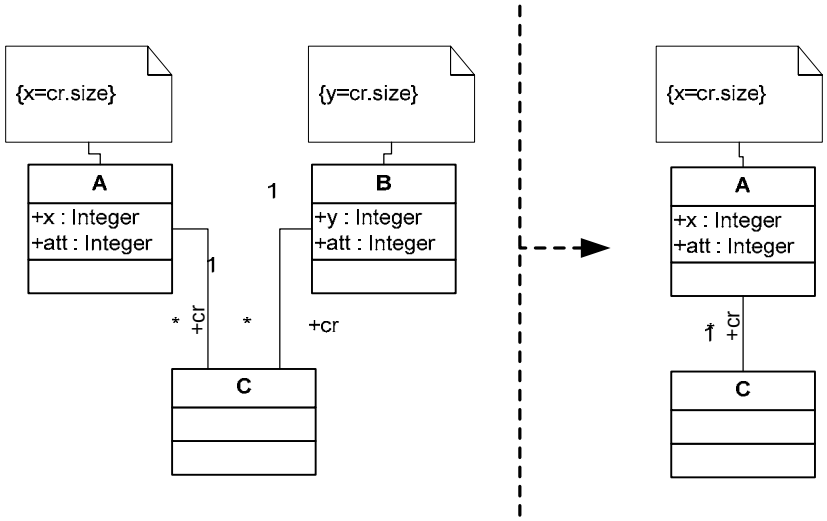
	ATL Transformation Catalogue of Model Transformations	Author Eric Simon eric.simon3 <*at*> gmail.com
INRIA	**Documentation**	Aug 7th 2006

1. ATL Transformation Example: Remove redundant classes

This example is extract from Catalogue of Model Transformations by K. Lano.
Section 2.4: Removal of many-many associations, page 19.



2. ATL Transformation overview

2.1. Description

"Classes may be redundant because they are essentially duplicates of other classes in the model but with a different name (synonyms), or because they are not needed in the system being defined."

2.2. Purpose

"Duplication of classes will lead to over-complex models which are difficult to modify and analyse."

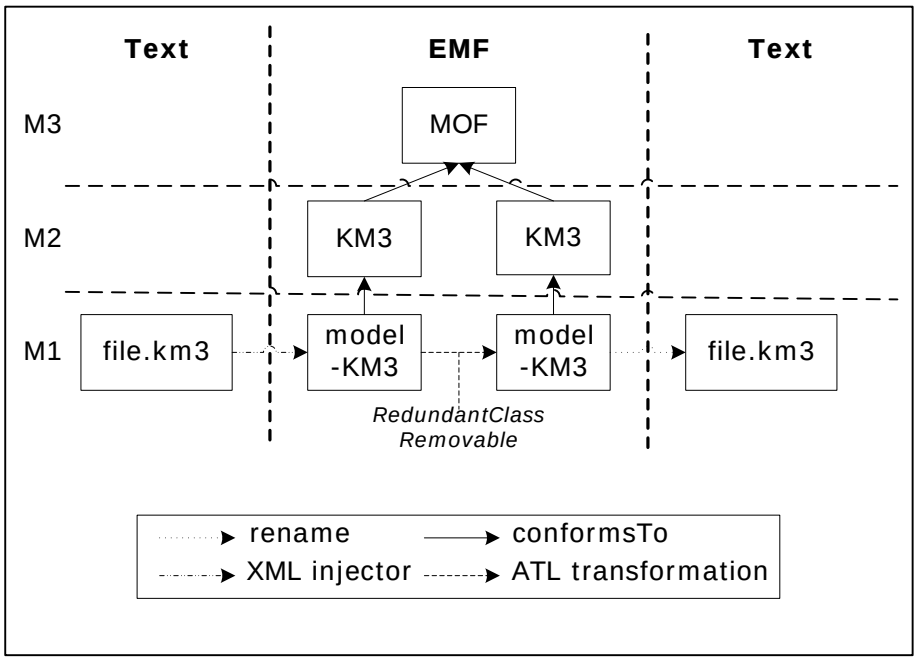


Fig 1. Overview of the transformation

2.3. Rules specification

The transformation has the same metamodel for the source and the target: UML2. However, we choice two different name: UML2 and UML2Target, indeed there is a confusion with the rule ocl: UML2!<nameElement>->allInstances() which returns all the class appertain to the source **and** the target.

The definition of a redundant class is subjective, and so it is necessary to adapt the criterions according to the context.

A *Class* element is considered as redundant if another *Class* element has the same properties (attribute number, constraint number …). So before to generate the target model, the redundant *Class* elements are searched and stored in a set.

- For each *Model* element, another *Model* element is created with the following elements:
  - the attribute *name* is the same,
  - the reference *ownedMember* owns all instances which do not appertain to a class redundant .
- For each *DataType* element, another *DataType* element is created with the following element:
  - the attribute name is the same.

- For each *LiteralNull* element, if it does not appertain to a class redundant:
  - another *LiteralNull* element is created.
- For each *LiteralInteger* element, if it does not appertain to a class redundant:
  - another *LiteralInteger* element is created with the following element:
    - the attribute *value* is the same.
- For each *LiteralUnlimitedNatural* element, if it does not appertain to a class redundant:
  - another *LiteralUnlimitedNatural* element is created with the following element:
    - the attribute *value* is the same
- For each *LiteralString* element, if it does not appertain to a class redundant:
  - another *LiteralString* element is created with the following element:
    - the attribute *value* is the same.
- For each *Association* element, if it does not appertain to a class redundant:
  - another *Association* element is created with the following elements:
    - the attribute *name* is the same,
    - the reference *memberEnd* is the same one as source.
- For each *Property* element, if it does not appertain to a class redundant:
  - another *Property* element is created with the following elements:
    - the attribute *name* is the same,
    - the reference *type* is the same one as the source.
- For each *Constraint* element, if it does not appertain to a class redundant:
  - another *Constraint* element is created with the following elements:
    - the attribute *name* is the same,
    - the reference *namespace* is the same one as the source.
- For each *Class* element, if it does not appertain to a class redundant:
  - another *Class* element is created with the following elements:
    - the attributes *name* and *isActive* are the same,
    - the references ownedOperation, nestedClassifier, ownedReception and ownedAttribute are the same one as the source.

2.4. ATL Code

```
module RedundantClassRemovable; -- Module Template
create OUT : UML2Target from IN : UML2;


helper def: assoMap : Map(UML2!Class, Sequence(UML2!Class)) = Map{};
rule isAlreadyConsidered(ref1 : UML2!Class, ref2 : UML2!Class) {

   do {
      if (not thisModule.assoMap.get(ref2).oclIsUndefined()) {
         if (thisModule.assoMap.get(ref2)->includes(ref1)) {
            true;
         }
         else {
             if (not thisModule.assoMap.get(ref1).oclIsUndefined()) {
               thisModule.assoMap <-
thisModule.assoMap.including(ref1,thisModule.assoMap.get(ref1)->including(ref2));
               false;
             }
            else {
               thisModule.assoMap <- thisModule.assoMap.including(ref1, Sequence{ref2});
               false;
            }
         }
      }
      else  {
            if (not thisModule.assoMap.get(ref1).oclIsUndefined()) {
               thisModule.assoMap <-
thisModule.assoMap.including(ref1,thisModule.assoMap.get(ref1)->including(ref2));
               false;
             }
            else {
               thisModule.assoMap <- thisModule.assoMap.including(ref1, Sequence{ref2});
               false;
            }
      }

   }
}

-- @comment this helper returns a boolean, true if a class can be considered as redundant else
false. The criterion to consider that a class is redundant is not optimal, so it must
strengthen the criterions according to context.
helper def: isRedundantClass : Set(UML2!Class) =
   UML2!Class->allInstances()->select(c|c.oclIsTypeOf(UML2!Class))->
      iterate(inputC1; acc : Sequence(UML2!Class) = Sequence{} | acc->including(UML2!Class-
>allInstances()->
      select(c|c.oclIsTypeOf(UML2!Class))->
      iterate(inputC2; acc1 : Sequence(UML2!Class) = Sequence{} |
      acc1->including(
      if
         (inputC1<> inputC2
         and inputC1.ownedAttribute->size() = inputC2.ownedAttribute->size()
         and inputC1.ownedRule->size() = inputC2.ownedRule->size()
         and inputC1.ownedAttribute->collect(a|a.type)->asSet() = inputC2.ownedAttribute-
>collect(a|a.type)->asSet()
         and (not thisModule.isAlreadyConsidered(inputC1, inputC2)))
      then
         inputC1
      else
         Sequence{}
```

```
      endif
      )
   )
   )->flatten())->flatten()
   ;
```

2.4.1. -- @begin Model

```
rule Model {
   from
      inputM : UML2!Model
   to
      outputM : UML2Target!Model (
         name <- inputM.name,
         ownedMember <- inputM.ownedMember->select(c|c.oclIsTypeOf(UML2!Class))->select(c| not
thisModule.isRedundantClass->includes(c)),
         ownedMember <- inputM.ownedMember->select(c|not c.oclIsTypeOf(UML2!Class))
      )
}
-- @end Model
```

2.4.2. -- @begin DataType

```
rule DataType {
   from
      inputC : UML2!DataType
   to
      outputC : UML2Target!DataType (
      name <- inputC.name
      )
}
-- @end DataType
```

2.4.3. -- @begin LiteralNull

```
rule LiteralNull {
   from
      inputLN : UML2!LiteralNull
      (if inputLN.owner.oclIsTypeOf(UML2!Constraint)
         then
            not (thisModule.isRedundantClass->
                includes(inputLN.owner.namespace))
         else
            not (if inputLN.owner.owningAssociation->oclIsUndefined()
                  then true
                  else inputLN.owner.owningAssociation.member->
                     exists(p| thisModule.isRedundantClass->includes(p.type))
                  endif)
         endif
      )
   to
      outputLN : UML2Target!LiteralNull
}
-- @end LiteralNull
```

2.4.4. -- @begin LiteralInteger

```
rule LiteralInteger {
   from
      inputLI : UML2!LiteralInteger
      (if inputLI.owner.oclIsTypeOf(UML2!Constraint)
         then
            not (thisModule.isRedundantClass->
                includes(inputLI.owner.namespace))
         else
            not (if inputLI.owner.owningAssociation->oclIsUndefined()
                  then true
```

```
                else inputLI.owner.owningAssociation.member->
                   exists(p| thisModule.isRedundantClass->includes(p.type))
                endif)
         endif
      )
   to
      outputLI : UML2Target!LiteralInteger (
         value <- inputLI.value
      )
}
-- @end LiteralInteger

2.4.5.  -- @begin LiteralUnlimitedNatural
rule LiteralUnlimitedNatural {
   from
      inputLUN : UML2!LiteralUnlimitedNatural
      (if inputLUN.owner.oclIsTypeOf(UML2!Constraint)
         then
            not (thisModule.isRedundantClass->
                includes(inputLUN.owner.namespace))
         else
            not (if inputLUN.owner.owningAssociation->oclIsUndefined()
                  then true
                  else inputLUN.owner.owningAssociation.member->
                     exists(p| thisModule.isRedundantClass->includes(p.type))
                  endif)
         endif
      )
   to
      outputLUN : UML2Target!LiteralUnlimitedNatural (
         value <- inputLUN.value
      )
}
-- @end LiteralUnlimitedNatural

2.4.6.  -- @begin LiteralString
rule LiteralString {
   from
      inputLS : UML2!LiteralString
      (if inputLS.owner.oclIsTypeOf(UML2!Constraint)
         then
            not (thisModule.isRedundantClass->
                includes(inputLS.owner.namespace))
         else
            not (if inputLS.owner.owningAssociation->oclIsUndefined()
                  then true
                  else inputLS.owner.owningAssociation.member->
                     exists(p| thisModule.isRedundantClass->includes(p.type))
                  endif)
         endif
      )
   to
      outputLS : UML2Target!LiteralString (
         value <- inputLS.value
      )
}
-- @end LiteralString

2.4.7.  -- @begin Association
rule Association {
   from
      inputA : UML2!Association
      (not inputA.member->exists(p| thisModule.isRedundantClass->includes(p.type)))
   to
```

```
      outputA : UML2Target!Association (
         name <- inputA.name,
         memberEnd <- inputA.memberEnd
   )
}
-- @end Association

2.4.8.  -- @begin Property
rule Property {
   from
      inputP : UML2!Property
      (not (thisModule.isRedundantClass->includes(inputP.class_)
         or thisModule.isRedundantClass->includes(inputP.type)
         or (if inputP.owningAssociation->oclIsUndefined()
            then false
            else inputP.owningAssociation.member->
         exists(p| thisModule.isRedundantClass->includes(p.type))
         endif)
         ))
   to
      outputP : UML2Target!Property (
         owningAssociation <- inputP.owningAssociation,
         name <- inputP.name,
         type <- inputP.type,
         upperValue <- inputP.upperValue,
         lowerValue <- inputP.lowerValue,
         defaultValue <-inputP.defaultValue
      )
}
-- @end Property

2.4.9.  -- @begin Constraint
rule Constraint {
   from
      inputC : UML2!Constraint
      (not thisModule.isRedundantClass->includes(inputC.namespace))
   to
      outputC : UML2Target!Constraint (
         name <-  inputC.name,
         namespace <- inputC.namespace,
         specification <- inputC.specification
      )
}
-- @end Constraint

2.4.10. -- @begin Class
rule Class {
   from
      inputC : UML2!Class
      (not thisModule.isRedundantClass->includes(inputC))
   to
      outputC : UML2Target!Class (
         name <- inputC.name,
         ownedOperation <- inputC.ownedOperation,
         nestedClassifier <-  inputC.nestedClassifier,
         isActive <-  inputC.isActive,
         ownedReception <-  inputC.ownedReception,
         ownedAttribute <- inputC.ownedAttribute
      )
}
-- @end Class
```

3. References

[1] Catalogue of Model Transformations
http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf