Direction Neutral Language Transformation with Metamodels

Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
gogolla@informatik.uni-bremen.de

Abstract. The aim of this work is to sketch a general metamodel-based frame for describing potentially bidirectional transformations between software languages. We propose to describe a single language with a metamodel consisting of a UML class diagram with classes, attributes and associations and accompanying OCL constraints. A language description is separated into a syntax and a semantics part. The allowed object diagrams of the syntax part correspond to the syntactically allowed words of the language. The semantics can associate with every word of the language certain semantical objects. However, only finite fractions of the semantics can be handled in metamodel tools. Having two languages described by their metamodels, a transformation between them is established by another metamodel, a transformation model. The transformation model can associate a syntax object from one language with a syntax object from the other language in a direction neutral way and opens the possibility for bidirectionality. Analogously, semantical objects can be connected. Transformation properties like 'equivalence' or 'embedding' can and must be expressed with constraints. Thus, the approach describes syntax and semantics of the languages, their transformation and their properties in a uniform way by means of metamodels.

1 Introduction

The aim of this work is to sketch a general metamodel-based frame for describing potentially bidirectional transformations between software languages. We use the term 'software language' in order to distiguish the artificial languages which we consider from 'natural languages'.

2 Frame for Software Language Transformation

Our goals for this work are: descriptive transformation specification, direction neutral transformations, separation of syntax and semantics, and determination of transformation properties.

Descriptive transformation specification: Our first goal is to specify the transformation between two software languages in a descriptive way [BBG+06]. We aim at describing the transformation result, i.e., pairs of words over the two languages which are to be transformed into each other. We call such a descriptive specification a transformation contract, for short tract. Our goal is not to give an operational or procedural description of the transformation process.

Direction neutral transformations: The second goal is to allow for direction neutral, possibly bidirectional transformations [Ste07]. This means that we do not want to restrict ourselves to a source and a target language where the used notions already indicate a transformation direction, but we assume that one language may be transformed into another language and vice versa. Therefore we will use the terms north language and south language when speaking of the two languages. This metaphor is intended to reflect the circumstance that one can go from north to south and vice versa.

Separation of syntax and semantics: A third goal is that we want to separate the languages into a syntax and a semantics part [HR04]. In the syntax part the allowed words of the language are described, in the semantics part a domain is set up and allowed syntactic words of the language are interpreted by establishing connections between the word and possible sub-words and elements of the semantic domain. The language user formulates a word over the syntax part; the semantics part evaluates the word and returns an entity from the semantics domain as the result to the language user. However, usually not all details of the semantics are relevant and shown to the language user, e.g., intermediate results will be blinded out. Besides, not all elements of the semantics part must neccessarily be reachable from the syntax part.

Determination of transformation properties: The fourth goal and last ingredient of our approach is the ability to express transformation properties like embedding or equivalence in a descriptive way by means of constraints [Gog05]. For example, an embedding of north into south will require that every north word will have an equivalent south word but not neccessarily vice versa. An idealized overview on the described ingredients is pictured in Figure 1.

The fundamental metalanguage which we use for formulating the various languages and its parts are metamodels, i.e., UML class diagrams [RBJ05] together with OCL [WK03] constraints. We validate the developed metamodels with our UML and OCL tool USE [GBR07]. USE allows the developer to instantiate class diagrams with object diagrams and to check the validity of UML constructs and OCL constraints with test cases. Thereby, USE supports developers in finding the right metamodels which correspond to their mental, informal expectations.

3 Software Language Transformation by Example

We explain our concepts by means of an example transforming roman numbers into natural numbers [CFH+09]. We have completely specified this transforma-

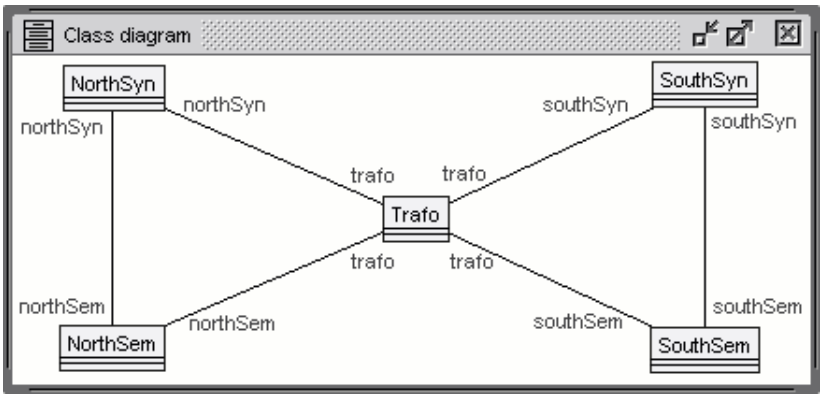

Fig. 1. Transformation between Two Languages

tion in our tool USE. Figure 2 presents a screenshot showing (a) on the left the project browser displaying the central ingredients of the model, namely the names of the classes, associations and invariants, (b) in the upper right the class diagram, and (c) in the lower right an example object diagram.

3.1 Class Diagram and Invariants

The class diagram follows the general software language transformation pattern which we have introduced before. In the left part, we see the first language with classes RomanNumber and Natural, in the right part we identify the second language with classes NaturalNumber and Int (and sub-classes Zero, Neg, and Pos), and in the middle we face the transformation class Trafo. Let us look at the several parts in some detail.

Syntax of Roman Numbers: The syntax of the roman numbers is given by class RomanNumber. It contains one String-returning attribute value which gives the representation of the roman number as a string. The Integer value is computed from the string with the operation value() by means of the auxilliary operations value2MCXI() and MCXI2integer().

Semantics of Roman Numbers: The semantics of the roman numbers is described by (a) the class Natural with a single Integer-returning attribute value and (b) the association connecting Natural and RomanNumber.

Syntax of Natural Numbers: The syntax of the natural numbers is presented by the class NaturalNumber in which the String-returning attribute value states the central content.

Semantics of Natural Numbers: The semantics of the natural numbers is a model with (a) the most complex class, the class Int and (b) the association between Int and NaturalNumber. Int has three subclasses called Zero, Neg, and Pos. The intention is to describe integers with negative and positive numbers and the possibility for specifying zero. There is one association

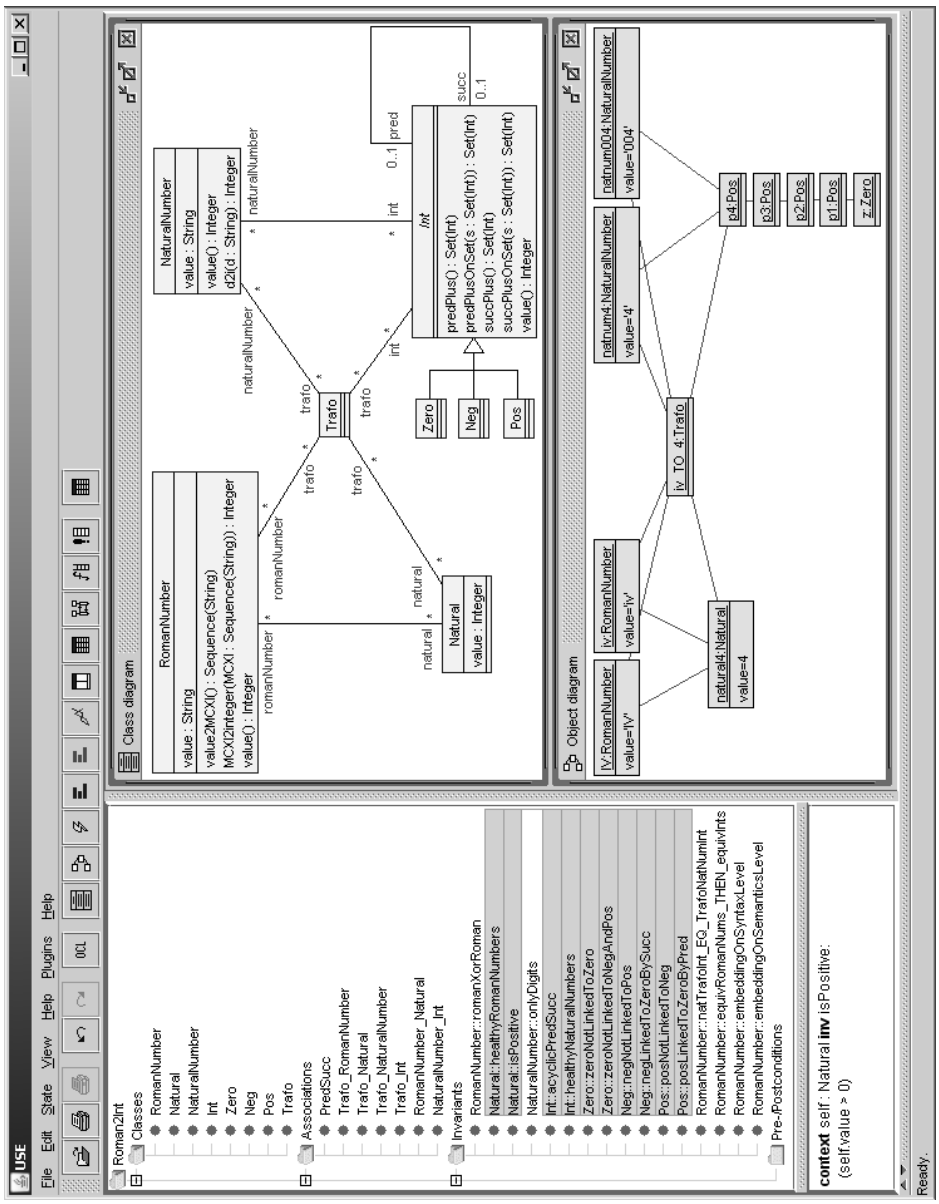

Fig. 2. USE Screenshot for Example Transformation

PredSucc on Int yielding the predeccessor resp. successor of an integer. The valid object diagrams for class Int, which respect the multiplicities and the invariants, are isomorphic to bags of integers. Figure 3 shows the representation of the integer bag Bag{-2,-1,0,1,2,1}.

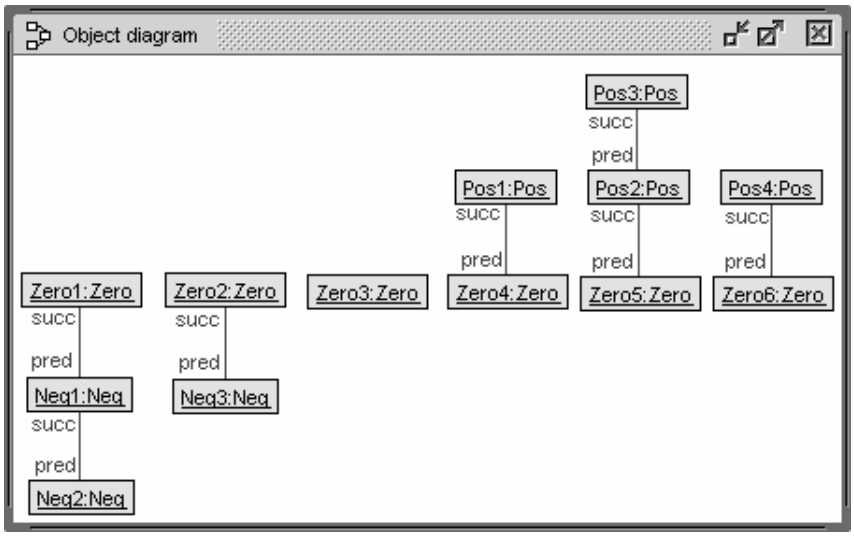

Fig. 3. Int Representation of Example Integers

You may think of a valid object diagram as representing bags of integers which correspond to normal form terms in the standard algebraic specification for the integers, here shown in Maude [CDE+07] syntax.

```
fmod INT is
  sort Int .
  op zero : -> Int .
  ops succ pred : Int -> Int .
  var I : Int .
  eq succ(pred(I)) = I .
  eq pred(succ(I)) = I .
endfm
```

The determined normal form terms are $\{pred^n(zero)\} \cup \{succ^n(zero)\}$. The elements in Figure 3 correspond to the normal forms {pred(pred(zero)), pred(zero), zero, succ(zero), succ(succ(zero)), succ(zero)}. Terms representing not a normal form (like succ(pred(zero))) cause a constraint violation. Note that in Figure 3, the insertion of any additional link would lead to a multiplicity or constraint violation.
The reader may have noticed that our semantics for class NaturalNumber is more complicated than strictly neccessary. Of course, we could have chosen a simple Integer-returning attribute value as in the class Natural. However, we wanted to a give an example of a non-trivial semantic domain which

can be built up with class and object diagrams by using multiplicities and invariants.

Invariants: The invariant names for the example are already present in Figure 2. All details will be found in the appendix where the complete, textual USE model is shown. But we want to go through the used invariants one by one and shortly informally explain their task.

RomanNumber::romanXorRoman requires that the String-returning attribute value contains either a lower or upper letter representation of a roman number between 1 and 3999.

Natural::healthyRomanNumbers assures that a Natural object with Integer-returning attribute value is connected to a RomanNumber object representing the same integer value.

Natural::isPositive states that the Integer-returning attribute value is positive.

NaturalNumber::onlyDigits demands that only digits are allowed to appear in a NaturalNumber object.

Int::acyclicPredSucc rejects cycles in the PredSucc association and captures the intuition that the integers are built with terms, i.e., trees, and not with (possibly cyclic) graphs.

Int::healthyNaturalNumbers assures that an Int object with Integer-returning operation value() is connected to a NaturalNumber object representing the same integer value.

Zero::zeroNotLinkedToZero states that an object of class Zero is not allowed be linked to another Zero object.

Zero::zeroNotLinkedToNegAndPos demands that an object of class Zero is not allowed be linked both to a Neg object and a Pos object.

Neg::negNotLinkedToPos asserts that a Neg object is not linked to a Pos object.

Neg::negLinkedToZeroBySucc requires that a Neg object is linked to a Zero object by employing only the succ role.

Pos::posNotLinkedToNeg asserts that a Pos object is not linked to a Neg object.

Pos::posLinkedToZeroByPred requires that a Pos object is linked to a Zero object by employing only the pred role.

RomanNumber::natTrafoInt_EQ_TrafoNatNumInt states a commutativity law starting with a RomanNumber object going through a Trafo object and ending in an Int object.

RomanNumber::equivRomanNums_THEN_equivInts requires that equivalent roman numbers are transformed into the same integer values.

RomanNumber::embeddingOnSyntaxLevel claims that the transformation of roman numbers into natural numbers is an embedding on the syntactic level (objects of type NaturalNumber are considered), i.e., the transformation can be understood as an injective function.

RomanNumber::embeddingOnSemanticsLevel asserts that the transformation of roman numbers into natural numbers is an embedding on

the semantics level (objects of type Int are considered), i.e., the transformation can be understood as an injective function.

The last four invariants are formulated as invariants on class RomanNumber. However, the invariants make use of Trafo objects. Therefore, these four constraints restrict the transformation and may be understood as the central requirements for determining the transformation properties.

3.2 Example Object Diagram

The object diagram in the lower right part of Figure 2 explains the central ideas of the transformation. Roughly speaking, the object diagram shows six different representations of the value four by the objects IV, iv, natural4, natnum4, natnum004, and p4. We see that the value four is represented by two RomanNumber objects IV and iv with attribute values coinciding with object identity. Semantic links connect objects IV and iv to the object natural4 in the bottom of the object diagram. Two respective NaturalNumber objects called natnum4 and natnum004 contain the value '4' and '004', respectively. The Int representation is given by an object network with five Int objects (one Zero object and four Pos objects). Semantic links connect natnum4 and natnum004 in the top with Pos object p4 in the bottom. In the middle part, the Trafo object iv_TO_4 connects all syntax and semantics objects from the left with the syntax and semantics objects from the right. All invariants are valid.

Note, that instead of having one Trafo object representing both syntax and semantics, we could have chosen to represent the transformation on a syntax and on a semantics level with two class TrafoSyn and TrafoSem and a connecting association. Then we would have two different, but connected objects iv_TO_4_syn and iv_TO_4_sem for syntax and semantics, respectively.

Besides, note that, in order to obtain an injective embedding from roman numbers into natural numbers, there are many alternatives for Trafo[asNatural(n),asInt(n)] as we have decided in the example transformation. For example, one could map a number n onto n+n or n*n: Trafo[asNatural(n),asInt(n+n)], Trafo[asNatural(n),asInt(n*n)].

4 Conclusion

This work sketched a general metamodel-based frame for describing potentially bidirectional transformations between software languages. Languages have been specified with metamodels consisting of UML class diagrams and OCL constraints. Transformations between languages have been established also by metamodels, so-called transformation models. The approach allows to describe syntax and semantics of the languages, their transformation and their properties in a uniform way by means of metamodels.

References

[BBG+06] J. Bezivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, A. Lindow. Model Transformations? Transformation Models! In Nierstrasz et al. (eds.), *Proc. 9th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2006)*. LNCS 4199, Springer, Berlin, 2006.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott (eds.). *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. LNCS 4350. Springer, 2007.

[CFH+09] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In Paige (ed.), *Proc. 2nd Int. Conf. Theory and Practice of Model Transformations (ICMT)*. LNCS 5563, pp. 260–283. Springer, 2009.

[GBR07] M. Gogolla, F. Büttner, M. Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming* 69:27–34, 2007.

[Gog05] M. Gogolla. Tales of ER and RE Syntax and Semantics. In Cordy et al. (eds.), *Transformation Techniques in Software Engineering*. IBFI, Schloss Dagstuhl, Germany, 2005. Dagstuhl Seminar Proceedings 05161. 51 pages.

[HR04] D. Harel, B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer* 37(10):64–72, 2004.

[RBJ05] J. Rumbaugh, G. Booch, I. Jacobson. *The Unified Modeling Language Reference Manual, Second Ed.* Addison-Wesley, Reading, 2005.

[Ste07] P. Stevens. A Landscape of Bidirectional Model Transformations. In Lämmel et al. (eds.), *Proc. Int. Conf. GTTSE*. LNCS 5235, pp. 408–424. Springer, 2007.

[WK03] J. Warmer, A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 2003. 2nd Edition.

Appendix: USE Model for Roman Numbers To Natural Numbers

```
model Roman2Int

class RomanNumber ----------------------------------------------------
attributes
  value:String
operations
  value2MCXI():Sequence(String)=
    let MMM=Set{'','M','MM','MMM'} in
    let CCC=Set{'','C','CC','CCC','CD','D','DC','DCC','DCCC','CM'} in
    let XXX=Set{'','X','XX','XXX','XL','L','LX','LXX','LXXX','XC'} in
    let III=Set{'','I','II','III','IV','V','VI','VII','VIII','IX'} in
    MMM->iterate(A;resA:Sequence(String)=Sequence{}|
      CCC->iterate(B;resB:Sequence(String)=resA|
        XXX->iterate(C;resC:Sequence(String)=resB|
          III->iterate(D;resD:Sequence(String)=resC|
```

```
          if value.toUpper()=A.concat(B).concat(C).concat(D)
            then Sequence{A,B,C,D} else resD endif))))
  MCXI2integer(MCXI:Sequence(String)):Integer=
    let M=MCXI->at(1) in let C=MCXI->at(2) in
    let X=MCXI->at(3) in let I=MCXI->at(4) in
    let dM=if M='' then 0 else if M='M' then 1000 else
           if M='MM' then 2000 else 3000 endif endif endif in
    let dC=if C=''     then 0   else if C='C'   then 100 else
           if C='CC'   then 200 else if C='CCC' then 300 else
           if C='CD'   then 400 else if C='D'   then 500 else
           if C='DC'   then 600 else if C='DCC' then 700 else
           if C='DCCC' then 800 else 900
           endif endif endif endif endif endif endif endif endif in
    let dX=if X=''     then 0  else if X='X'   then 10 else
           if X='XX'   then 20 else if X='XXX' then 30 else
           if X='XL'   then 40 else if X='L'   then 50 else
           if X='LX'   then 60 else if X='LXX' then 70 else
           if X='LXXX' then 80 else 90
           endif endif endif endif endif endif endif endif endif in
    let dI=if I=''     then 0 else if I='I'   then 1 else
           if I='II'   then 2 else if I='III' then 3 else
           if I='IV'   then 4 else if I='V'   then 5 else
           if I='VI'   then 6 else if I='VII' then 7 else
           if I='VIII' then 8 else 9
           endif endif endif endif endif endif endif endif endif in
    dM+dC+dX+dI
  value()=MCXI2integer(value2MCXI())
constraints
  inv romanXorRoman:
    value.size>0 and
    (Set{'','m','mm','mmm'}->exists(A|
     Set{'','c','cc','ccc','cd','d','dc','dcc','dccc','cm'}->exists(B|
     Set{'','x','xx','xxx','xl','l','lx','lxx','lxxx','xc'}->exists(C|
     Set{'','i','ii','iii','iv','v','vi','vii','viii','ix'}->exists(D|
       value=A.concat(B).concat(C).concat(D))))) xor
     Set{'','M','MM','MMM'}->exists(A|
     Set{'','C','CC','CCC','CD','D','DC','DCC','DCCC','CM'}->exists(B|
     Set{'','X','XX','XXX','XL','L','LX','LXX','LXXX','XC'}->exists(C|
     Set{'','I','II','III','IV','V','VI','VII','VIII','IX'}->exists(D|
       value=A.concat(B).concat(C).concat(D))))))
end

class Natural -------------------------------------------------------
attributes
  value:Integer
constraints
  inv healthyRomanNumbers:
    romanNumber->forAll(rn|rn.value()=value)
  inv isPositive: value>0
end
```

```
class NaturalNumber ----------------------------------------------------
attributes
  value:String
operations
  value():Integer=
    Sequence{1..value.size}->iterate(i;res:Integer=0|
      res*10+d2i(value.substring(i,i)) )
  d2i(d:String):Integer=
    if d='0' then 0 else if d='1' then 1 else
    if d='2' then 2 else if d='3' then 3 else
    if d='4' then 4 else if d='5' then 5 else
    if d='6' then 6 else if d='7' then 7 else
    if d='8' then 8 else if d='9' then 9 else oclUndefined(Integer)
    endif endif endif endif endif endif endif endif endif endif
constraints
  inv onlyDigits:
      let digit:Set(String)=
        Set{'0','1','2','3','4','5','6','7','8','9'} in
      Set{1..value.size}->forAll(i |
        digit->includes(value.substring(i,i))) and value.size>0
end

abstract class Int ----------------------------------------------------
operations
predPlus():Set(Int)=
  self.predPlusOnSet(Set{self.pred}->excluding(oclUndefined(Int)))
predPlusOnSet(s:Set(Int)):Set(Int)=
  let oneStep:Set(Int)=s.pred->asSet()->excluding(oclUndefined(Int)) in
  if oneStep->exists(i|s->excludes(i))
    then predPlusOnSet(s->union(oneStep)) else s endif
succPlus():Set(Int)=
  self.succPlusOnSet(Set{self.succ}->excluding(oclUndefined(Int)))
succPlusOnSet(s:Set(Int)):Set(Int)=
  let oneStep:Set(Int)=s.succ->asSet()->excluding(oclUndefined(Int)) in
  if oneStep->exists(i|s->excludes(i))
    then succPlusOnSet(s->union(oneStep)) else s endif
value():Integer=
  if oclIsTypeOf(Zero) then 0 else
    if oclIsTypeOf(Pos) then self.pred.value()+1 else
      if oclIsTypeOf(Neg) then self.succ.value()-1
                          else oclUndefined(Integer) endif endif endif
constraints
  inv acyclicPredSucc:
    self.predPlus()->union(self.succPlus())->excludes(self)
  inv healthyNaturalNumbers:
    naturalNumber->forAll(nn|nn.value()=value())
end
```

```
association PredSucc between
  Int[0..1] role pred
  Int[0..1] role succ
end

class Zero < Int
constraints
  inv zeroNotLinkedToZero:
    not self.predPlus()->union(self.succPlus())->exists(i|
      i.oclIsTypeOf(Zero))
  inv zeroNotLinkedToNegAndPos:
    not self.predPlus()->union(self.succPlus())->exists(n,p|
      n.oclIsTypeOf(Neg) and p.oclIsTypeOf(Pos))
end

class Neg < Int
constraints
  inv negNotLinkedToPos:
    not self.predPlus()->union(self.succPlus())->exists(p|
      p.oclIsTypeOf(Pos))
  inv negLinkedToZeroBySucc:
    self.succPlus()->exists(z|z.oclIsTypeOf(Zero))
end

class Pos < Int
constraints
  inv posNotLinkedToNeg:
    not self.predPlus()->union(self.succPlus())->exists(n|
      n.oclIsTypeOf(Neg))
  inv posLinkedToZeroByPred:
    self.predPlus()->exists(z|z.oclIsTypeOf(Zero))
end

class Trafo ------------------------------------------------------------
end

association Trafo_RomanNumber between
  Trafo[0..*] role trafo
  RomanNumber[0..*] role romanNumber
end

association Trafo_Natural between
  Trafo[0..*] role trafo
  Natural[0..*] role natural
end

association Trafo_NaturalNumber between
  Trafo[0..*] role trafo
  NaturalNumber[0..*] role naturalNumber
end
```

```
association Trafo_Int between
  Trafo[0..*] role trafo
  Int[0..*] role int
end

association RomanNumber_Natural between
  RomanNumber[0..*] role romanNumber
  Natural[0..*] role natural
end

association NaturalNumber_Int between
  NaturalNumber[0..*] role naturalNumber
  Int[0..*] role int
end

constraints ------------------------------------------------------------

-- synob.semantics.trafo = synob.trafo.semantics
context rn:RomanNumber inv natTrafoInt_EQ_TrafoNatNumInt:
  rn.natural.trafo.int->asSet()=rn.trafo.naturalNumber.int->asSet()

-- synob1.semantics=synob2.semantics IMPLIES
--   synob1.trafo.semantics=synob2.trafo.semantics
context rn1:RomanNumber inv equivRomanNums_THEN_equivInts:
  RomanNumber.allInstances->forAll(rn2|
    rn1.natural=rn2.natural implies
    rn1.trafo.naturalNumber.int=rn2.trafo.naturalNumber.int)

-- synob1.semantics<>synob2.semantics IMPLIES
--   synob1.trafo.syntax<>synob2.trafo.syntax
context rn1:RomanNumber inv embeddingOnSyntaxLevel:
  RomanNumber.allInstances->forAll(rn2|
    rn1.natural<>rn2.natural implies
    rn1.trafo.naturalNumber<>rn2.trafo.naturalNumber)

-- synob1.semantics<>synob2.semantics IMPLIES
--   synob1.trafo.semantics<>synob2.trafo.semantics
context rn1:RomanNumber inv embeddingOnSemanticsLevel:
  RomanNumber.allInstances->forAll(rn2|
    rn1.natural<>rn2.natural implies
    rn1.trafo.naturalNumber.int<>rn2.trafo.naturalNumber.int)

------------------------------------------------------------------------

use> open integer.use

use> !create IV:RomanNumber
use> !set IV.value:='IV'
```

```
use> !create iv:RomanNumber
use> !set iv.value:='iv'

use> !create natural4:Natural
use> !set natural4.value:=4

use> !insert (IV,natural4) into RomanNumber_Natural
use> !insert (iv,natural4) into RomanNumber_Natural

use> !create natnum004:NaturalNumber
use> !set natnum004.value:='004'

use> !create natnum4:NaturalNumber
use> !set natnum4.value:='4'

use> !create z:Zero
use> !create p1,p2,p3,p4:Pos

use> !insert (z,p1) into PredSucc
use> !insert (p1,p2) into PredSucc
use> !insert (p2,p3) into PredSucc
use> !insert (p3,p4) into PredSucc

use> !insert (natnum004,p4) into NaturalNumber_Int
use> !insert (natnum4,p4) into NaturalNumber_Int

use> !create iv_TO_4:Trafo

use> !insert (iv_TO_4,IV) into Trafo_RomanNumber
use> !insert (iv_TO_4,iv) into Trafo_RomanNumber

use> !insert (iv_TO_4,natural4) into Trafo_Natural

use> !insert (iv_TO_4,natnum004) into Trafo_NaturalNumber
use> !insert (iv_TO_4,natnum4) into Trafo_NaturalNumber

use> !insert (iv_TO_4,p4) into Trafo_Int

use> check
     checking structure...
     checking invs...
     checking inv (1)  'Int::acyclicPredSucc': OK.
     checking inv (2)  'Int::healthyNaturalNumbers': OK.
     checking inv (3)  'Natural::healthyRomanNumbers': OK.
     checking inv (4)  'Natural::isPositive': OK.
     checking inv (5)  'NaturalNumber::onlyDigits': OK.
     checking inv (6)  'Neg::negLinkedToZeroBySucc': OK.
     checking inv (7)  'Neg::negNotLinkedToPos': OK.
     checking inv (8)  'Pos::posLinkedToZeroByPred': OK.
     checking inv (9)  'Pos::posNotLinkedToNeg': OK.
```

```
     checking inv (10) 'RomanNumber::embeddingOnSemanticsLevel': OK.
     checking inv (11) 'RomanNumber::embeddingOnSyntaxLevel': OK.
     checking inv (12) 'RomanNumber::equivRomanNums_THEN_equivInts': OK.
     checking inv (13) 'RomanNumber::natTrafoInt_EQ_TrafoNatNumInt': OK.
     checking inv (14) 'RomanNumber::romanXorRoman': OK.
     checking inv (15) 'Zero::zeroNotLinkedToNegAndPos': OK.
     checking inv (16) 'Zero::zeroNotLinkedToZero': OK.
     checked 16 invs in 0.094s, 0 failures.

use> ?iv.trafo.naturalNumber
     Bag{@natnum004,@natnum4} : Bag(NaturalNumber)
use> ?IV.trafo.naturalNumber
     Bag{@natnum004,@natnum4} : Bag(NaturalNumber)

use> ?iv.trafo.naturalNumber.int
     Bag{@p4,@p4} : Bag(Pos)
use> ?IV.trafo.naturalNumber.int
     Bag{@p4,@p4} : Bag(Pos)

use> ?iv.trafo.int
     Bag{@p4} : Bag(Pos)
use> ?IV.trafo.int
     Bag{@p4} : Bag(Pos)
```