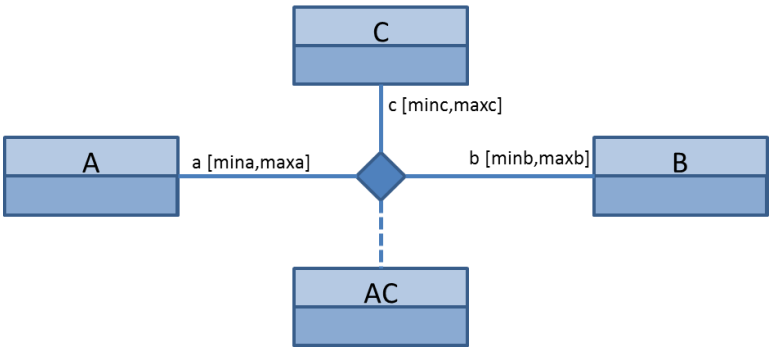
# N-ary associations in UML

This document is a summary of a brainstorming I have had with Ed Willink at the Eclipse OCL forum concerning UML n-ary Association Classes. The context of the discussion was support for parsing of OCl expressions involving navigation of this kind of associations. It reflects my current understanding of the semantics of UML Association Classes, not really an authoritative opinion.

## 1. Informal semantics of association ends' multiplicities

Let us consider the case of the ternary association defined in the following UML class diagram, with its corresponding association ends and multiplicities:




The first question I will address is the meaning of the association ends' multiplicities for this ternary association. According to my own interpretation of the UML specification (see Appendix 1 at the end of the document) the semantics of the multiplicities can be stated in plain English as follows:

- For any given objects `a` and `b`, there exists at least `minc` and at most `maxc` objects `c` such that `{a,b,c}` is an existing instance (link) of the association class AC.
- For any given objects `a` and `c`, there exists at least `minb` and at most `maxb` objects `b` such that `{a,b,c}` is an existing instance (link) of the association class AC.
- For any given objects `b` and `c`, there exists at least `mina` and at most `maxa` objects `a` such that `{a,b,c}` is an existing instance (link) of the association class AC.

A first remark is that the concept of "opposite end" is meaningless for ternary (or in general n-ary) associations. We cannot say that association end `a` is the opposite end of `b` or `c`. So, to verify multiplicities, we place ourselves in the context of a given pair (or in general n-1 tuple), and we evaluate constraints on the set of linked objects at the other end.

Consider for example a scenario with `mina` = `minb` = `minc` = `0` and `maxa` = `maxb` = `maxc` = `1` and the following snapshot of the existing objects and links at a given moment:

```
AC1=(a1,b1,c1)
AC2=(a1,b2,c2)
AC3=(a2,b1,c3)
AC4=(a3,b3,c1)
```

We take every given pair of objects and calculate the set of linked objects at the other end:

```
(a1,b1) ↦ {c1} (a1,b2) ↦ {c2} (a2,b1) ↦ {c3} (a3,b3) ↦ {c1}
(a1,c1) ↦ {b1} (a1,c2) ↦ {b2} (a2,c3) ↦ {b1} (a3,c1) ↦ {b3}
(b1,c1) ↦ {a1} (b2,c2) ↦ {a1} (b1,c3) ↦ {a2} (b3,c1) ↦ {a3}
```

And we can verify whether the cardinalities of the linked sets satisfy the specified multiplicities.

## 2. Navigating n-ary associations with OCL

The second point I want to address is the navigation expressions available in OCL for specifying constraint in an UML model that includes n-ary associations. There are two cases: navigating to an association class and navigating from an association class.

Navigation to an Association class is described in section 7.5.4 of the OCL 2.4 specification:

> "To specify navigation to association classes (Job and Marriage in the example), OCL uses a dot and the name of the association class:
>
> Context Person
> inv: self.Job
>
> The sub-expression self.Job evaluates to a Set of all the jobs a person has with the companies that are his/her employer. In the case of an association class, there is no explicit role name in the class diagram. The name Job used in this navigation is the name of the association class."

An important remark is that the expression always returns a set of instances of the association class, and that the cardinality of this set cannot be easily related to the multiplicities of the association ends. Continuing with the scenario of the previous section we can give some examples of evaluation of OCL expressions that navigate to the Association Class:

```
a1.AC = {AC1,AC2}, a2.AC = {AC3}, a3.AC = {AC4}
b1.AC = {AC1,AC3}, b2.AC = {AC2}, b3.AC = {AC4}
c1.AC = {AC1,AC4}, c2.AC = {AC2}, c3.AC = {AC3}
```

Navigation from an Association class is described in section 7.5.5 of the OCL 2.4 specification:

> "We can navigate from the association class itself to the objects that participate in the association. This is done using the dot-notation and the role-names at the association-ends
>
> context Job
> inv: self.employer.numberOfEmployees >= 1
> inv: self.employee.age > 21
>
> Navigation from an association class to one of the objects on the association will always deliver exactly one object. This is a result of the definition of AssociationClass. Therefore, the result of this navigation is exactly one object, although it can be used as a Set using oclAsSet() or its "->" shorthand."

Notice that the expression always returns one and exactly one object for each association end. An instance of an association class (link) always references a single target object for each association end, and cannot be null. In our running scenario we have the following examples of evaluation:

```
AC1.a = a1, AC1.b = b1, AC2.c = c1
AC2.a = a1, AC2.b = b2, AC2.c = c2
```

```
AC3.a = a2, AC3.b = b1, AC3.c = c3
AC4.a =a3, AC4.b = b3, AC4.c = c1
```

Using these basic expressions, we can easily express navigation from a pair of given objects to the other end of the link. In our running example, we can give the following examples for different combinations of association ends:

```
a1.AC->select(b=b1)->collect(c) = {c1}
a1.AC->select(c=c1)->collect(b) = {b1}
b1.AC->select(c=c1)->collect(a) ={a1}
```

Or equivalently:

```
AC.allInstances->select(a=a1 and b=b1)->collect(c) = {c1}
AC.allInstances->select(a=a1 and c=c1)->collect(b) = {b1}
AC.allInstances->select(b=b1 and c=c1)->collect(a) ={a1}
```

This kind of navigation is very common (it is the equivalent of “navigation to the opposite end” in the binary case), and would deserve some syntactic sugar in forthcoming OCL versions.

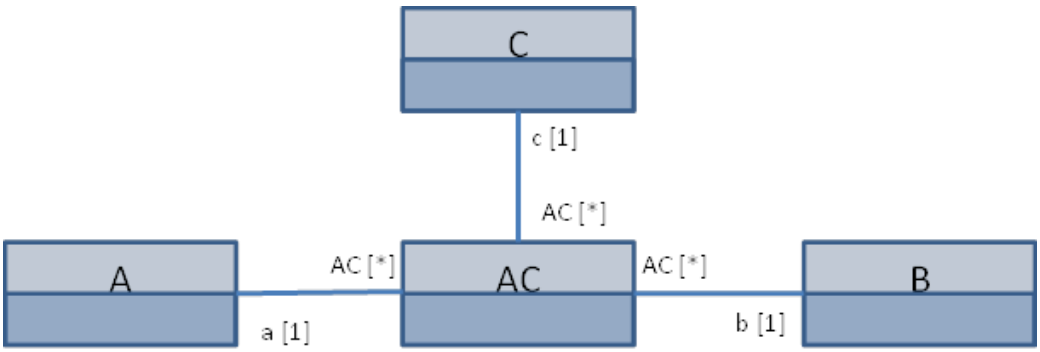
The most important thing to notice is the relationship of this kind of navigation to the multiplicities of the association ends. As a matter of fact, it is the properties of these “navigation to the other end” expressions that are specified by the association end constraints (multiplicities, uniqueness, ordering). This fact can be taken into account by the syntactic sugar to allow smart shortcuts for chained navigation.

The OCL specification also has some special syntax for the case of reflective associations (when the same Class is the target of more than one association end). The only particularity of reflective associations is that some of the expressions presented above become syntactically ambiguous. The specification propose notation to solve this ambiguity in the binary case, but it can be easily extended for the case of n-ary associations.

## 3. Eliminating n-ary associations from a UML model by transformation

Many UML/OCL tools do not fully support n-ary associations natively. However, they fully support the common case of binary associations. A usual strategy to support n-ary associations is then to transform the input UML model into an equivalent model with only binary associations.

Consider support for ternary associations, one first approximation is to transform our original UML model into the following model:



In this transformation, we converted the Association Class into a simple Class and replaced the ternary association by a set of binary associations (one for each of the original association ends).This simple transformation has some very interesting properties from the point of view of support for OCL navigation:

- Instances of the association class naturally have single references to the linked objects.
- Navigation from the Association Class is normal OCL property navigation.
- Navigation to the Association Class is also normal OCL property navigation.
- Navigation to/from the Association Class is well typed.
- It can easily and mindlessly be extended to the case of n-ary associations.
- In the case of the Eclipse OCL implementation, this transformation can be transparently performed when the UML model is converted into the pivot model, without user or programmer intervention.

The main problem of this transformation is that we loosed the important information concerning the association ends' constraints (multiplicity, uniqueness, ordering). One possible workaround is to automatically add invariants to the model that capture the original constraints.

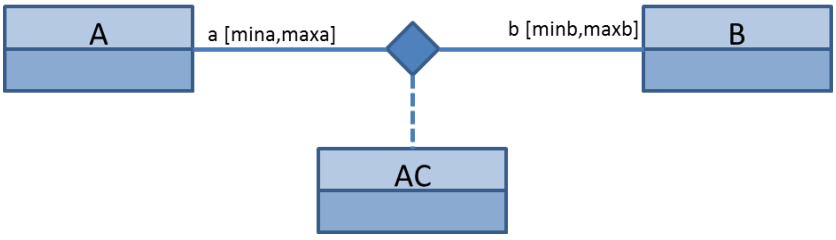
In the case of the ternary association, these invariants may look like this (by literally translating into OCL the English sentences of the first section):

- ```
  A.allInstances->forAll(someA| B.allInstances->forAll(someB|
          let countC = AC.allInstances->select(a=someA and b= someB)->collect(c)->size in
          countC >0 implies countC >= minc and countC<= maxc))
  ```
- ```
  A.allInstances->forAll(someA| C.allInstances->forAll(someC|
          let countB = AC.allInstances->select(a=someA and c= someC)->collect(b)->size in
          countB >0 implies countB >= minb and countB<= maxb))
  ```
- ```
  B.allInstances->forAll(someB| C.allInstances->forAll(someC|
          let countA = AC.allInstances->select(b=someB and c= someC)->collect(a)->size in
          countA >0 implies countA >= mina and countA<= maxa))
  ```

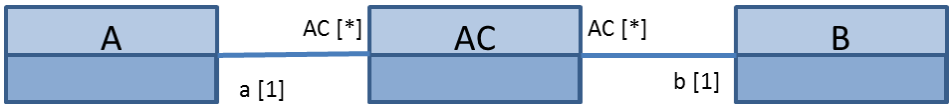
We can see that they can be generated systematically, and extended to the n-ary case. It is worth noting that adding these invariants is not needed for parsing the OCL constraints written by the modeler, but only for validation.

## 4. Binary Association Classes

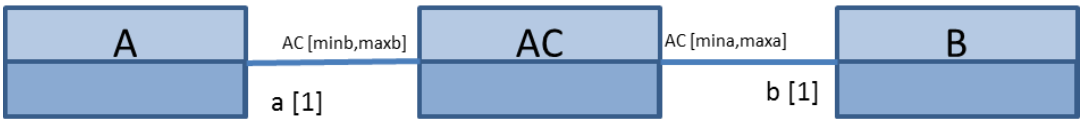
The transformation presented in the previous section only uses simple classes and bidirectional references. It can then be used to add support for binary Association Classes, if they are not supported natively. In this section, we show that in this particular case a most complete support can be offered.


  ```

An UML model using binary Association Classes can be converted, using the transformation already presented, into the following base model:



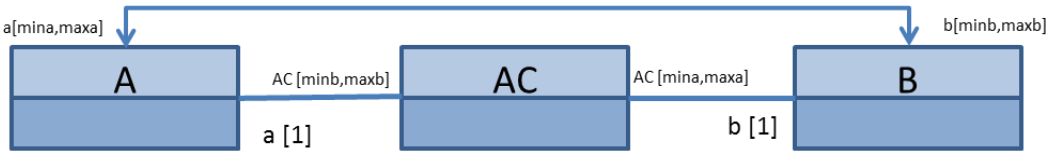

A first remark concerns the multiplicity of the navigation to the Association Class. In this case, because there is a single opposite end, the cardinality of the navigation to the Association Class corresponds exactly to the cardinality of the opposite end. We can therefore make a more precise transformation:




Navigation from/to the Association Class is not only well typed, as in the general case, but it also has the correct type (Set, Bag, reference) as defined in the multiplicities:

```
a1.AC->collect(b) = a1.AC.b
b1.AC->collect(a) = b1.AC.a
```

Moreover, because the association is binary there is a single source and single destination. So we can use Class-owned properties to provide direct navigation to the other end:




And in this case, the navigation expressions become simply `a1.b` and `b1.a`. Not only this expression can be straightforwardly parsed, but the expression can be correctly evaluated if the following invariant is enforced by the runtime:

```
Context A
inv self.b = self.AC.b

Context B
inv self.a = self.AC.a
```

Notice that in this particular case, we can also encode directly the Association ends' multiplicities in the added Class-owned properties, and there is no lost information.

# Appendix 1

## OMG/UML specification of the semantics of association ends' multiplicities

The exact wording of section 11.3.1 of the UML infrastructure specification 2.4 specifying the semantics of the association ends' multiplicities is the following:

> "For an association with N ends, choose any N-1 ends and associate specific instances with those ends. Then the collection of links of the association that refer to these specific instances will identify a collection of instances at the other end. The multiplicity of the association end constrains the size of this collection."

As always, evil is in the details, and many people have identified inconsistencies and imprecisions with this definition, especially for allowing a precise mathematical formalization. UML specification 2.5 is a little more precise and gives the following definition (section 11.5.3.1):

> "For an Association with N memberEnds, choose any N-1 ends. Let the Property that constitutes the other end be called oep, so that the Classifiers at the chosen N-1 ends are the context for oep (see 9.5.3). Associate specific instances with the context ends. Then the collection of links of the Association that refer to these specific instances will identify a set of instances at oep. The value represented by oep (see 9.5.3) is a collection calculated from this set as follows: All of the instances in the set occur in the collection, and nothing else does. If oep is marked as unique, each instance will occur in the collection just once, regardless of how many links connect to it. If oep is marked as nonunique, each instance will occur in the collection once for each link that connects to it. If oep is marked as ordered, the collection will be ordered in accordance with the ordering information in the links. The cardinality of this collection is its size. The multiplicity of oep constrains this cardinality, or in the case of qualified associations, the size of the collection partition that may be associated with a qualifier value."

The context of a property is a concept defined in section 9.5.3 of UML 2.5 specification as follows:

> "When a Property is owned by a Classifier other than an Association via ownedAttribute, then it represents an attribute of the Classifier. When related to an Association via memberEnd it represents an end of the Association. For a binary Association, it may be both at once. In either case, when instantiated a Property represents a value or collection of values associated with an instance of one (or in the case of a ternary or higher-order association, more than one) Classifier. This set of Classifiers is called the context for the Property; in the case of an attribute the context is the owning Classifier, and in the case of an association end the context is the set of Classifiers at the other end or ends of the Association."