Expressing UML Class Diagrams Properties with OCL

Martin Gogolla, Mark Richters
University of Bremen, FB 3, Computer Science Department
Postfach 330440, D-28334 Bremen, Germany
{gogolla|mr}@informatik.uni-bremen.de

Abstract

The Unified Modeling Language UML is a complex language offering many modeling features. Especially the description of static structures with class diagrams is supported by a rich set of primitives. This paper shows how to transfrom UML class diagrams involving cardinality constraints, qualifiers, association classes, aggregations, compositions, and generalizations into equivalent UML class diagrams employing only binary associations and OCL constraints. Thus we provide a better understanding of UML features. By reducing more complex features in terms of basic ones, we suggest an easy way users can gradually extend the set of UML elements they commonly apply in the modeling process.

1 Introduction

UML [OMG99c, OMG99b, OMG99a] being heavily influenced by OMT [RBP+91], Booch [Boo94], and OOSE [JCJÖ92] is a complex language with many modeling features. But up to now, no overall agreed semantic background has been developed. So naturally the question arises how to deal with the language element variety, for example, to relate the different language elements. In this paper we study UML class diagrams and compare the language elements available there. The general idea here is to explain advanced UML features by some more basic UML constructs. A similar approach [GPP98] has been taken by us recently for UML state diagrams concerning the UML dynamic modeling aspects.

Although UML provides a rich set of language features, many projects are forced to restrict themselves to utilizing only a small set of the language. This is partly due to a current lack of sufficiently reliable and efficient tools supporting the whole language and partly due to the fact that more time and effort is needed to learn and adapt additional features. Indeed, using only a small set of features can reduce the complexity of a design and facilitate communication. Our approach enables a smooth transition from utilizing a basic set of language features to more sophisticated ones by explaining them in terms of already known features. On the other hand, giving a definition in terms of simple features possibly helps identifying certain repeating patterns in existing designs which may better be represented by more suitable special features, thus emphasizing a particular design decision.

Our work is related to recent approaches handling formal aspects of UML and other object-oriented methods. Work has been done on the basis of well-established traditional approaches to specification like Z and VDM: [SF97, FBLPS97] focuses on the UML type system, a general integration of Z with object-orientation is discussed in [Ebe97], and in [Lan96, BLM97] an object calculus enhancing the expressibility of object-oriented notations has been proposed. Other approaches treat in detail the UML predecessor OMT [BCV96, WRC97], and in particular class diagrams [BC95] in connection with Larch are discussed. [BHH+97] sketches a general scenario for most UML diagrams without going into technical details, and [Öov98] presents a general framework for relationships in UML. [WB98] integrates structure and behavior specification on the basis of transition systems. [Cla99] gives a type system for static UML diagrams. [KC99] studies class diagrams by means of using Object-Z. [RACH00] concentrates on active classes, and [TE00] discusses class diagrams in connection with sequence diagrams.

The structure of the rest of this paper is as follows. Sections 2 to 6 point out how to translate n-ary associations with cardinality restrictions, qualifiers, association classes, aggregations and compositions, and generalizations, respectively. The UML subset we use employs general binary associations with additional constraints formulated in OCL [WK98, WK99]. These sections all have the same structure. First, we motivate the handled concept by citing respective parts from the "official" UML material: the semantics document [OMG99c], the notation guide [OMG99b], and the OCL description [OMG99a]. Then, a general translation scheme is presented, which is explained afterwards by an example. The paper ends with some concluding remarks.

2 Binary Associations with Cardinalities

2.1 Statements from UML Material

- *[An] association [denotes] the semantic relationship between two or more classifiers [classes] that involves connections among their instances.* Semantics.
- *A multiplicity item specifies the range of allowable cardinalities that a set may assume.* Notation.
- *[Multiplicity:] When placed on a target end, specifies the number of target instances that may be associated with a single source instance across the given association.* Semantics.

2.2 General Translation

Simple Cardinality Restrictions: In Fig. 1 we have indicated how to translate an association with cardinality restrictions (`l` for low and `h` for high) into an equiv-

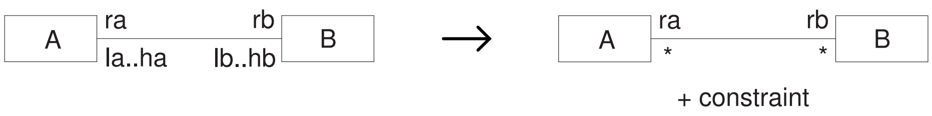

Figure 1: Equivalence Rule for Cardinality Notation

alent association with an explicit constraint. Specifying role names `ra` and `rb` in the diagram implies that the role name identifiers can be used as operations `ra: B -> Set(A)` and `rb: A -> Set(B)`. For example, `ra` returns for an `A` object the set of `B` objects related to the argument. The constraint requires that, for a single `A` object, the size of the set of related `B` objects is restricted by the lower bound `lb` and by the upper bound `hb` (analogously, for the other side of the association).

```
A.allInstances->forAll( a | a.rb->size>=lb and a.rb->size<=hb )
and
B.allInstances->forAll( b | b.ra->size>=la and b.ra->size<=ha )
```

Factoring Out Common Subexpressions: This little example already shows a minor deficiency of OCL: Common subexpressions have to be repeated as often as they are needed. For example, the term `a.rb->size` appears (and must appear) twice in the example. In OCL there is no structuring mechanisms for expressions. An alternative would be to provide a "let" construct in order to factor out common subexpressions. This would make expressions easier to understand especially complex ones with nested quantifiers and queries. Then, the first part of the above constraint could be shortened to the following, more readable expression.

```
let s=a.rb->size in
  A.allInstances->forAll( a | s>=lb and s<=hb )
```

However, in order to be in full accordance with all versions of OCL we refrain from taking advantage of such abbreviations.

Simplifying Resulting Formulas for Special Cases: If the lower bound in a cardinality specification is `0`, then a comparison like `a.rb.size>=0` is always true. In this case the comparison may be left out in the defining formula.

If the upper bound specification is `*` (which means that no finite upper bound is given), then there is no need to include an upper bound restriction in the formula. Thus the UML default cardinality specification `0..*` would be translated into the "empty" restricting constraint.

If lower and upper bounds of the cardinality specification coincide, say the left cardinality specification in Fig. 1 is `h..h` or equivalently only `h`, then the resulting formula can be simplified as follows.

```
B.allInstances->forAll( b | b.ra->size=h )
```

Avoiding Explicit Reference to All Instances: In order to make the OCL expression shorter and more readable, we omit in the following OCL formulas the explicit reference to all instances of a class, if this construction is clear from the context. Thus, for a class `C` the term `C->forAll(...)` is short for `C.allInstances->forAll(...)`. Analogously we define `C->exists(...)`, `C->select(...)` and `C->includes(...)`. Therefore, the above OCL expression is abbreviated to the following formula.

```
A->forAll( a | a.rb->size>=la and a.rb->size<=ha )
and
B->forAll( b | b.ra->size>=lb and b.ra->size<=hb )
```

Ordered Associations: Marking an association end as ordered, for example the left end with the role name `ra`, would change the signature of the operation induced by the role name from set to list, for example from `ra: B -> Set(A)` to `ra: B -> Sequence(A)`, but the constraint would be the same (because `size` is defined for all collection types).

Sequence of Integer Intervals: In general, cardinalities are given as a sequence of integer intervals, not just as a single interval. This extension can be captured by the corresponding constraint, as well. For example, if we had the cardinality specifications `la..ha, la'..ha'` and `lb..hb, lb'..hb'` instead of `la..ha` and `lb..hb`, the constraint would look like:

```
A->forAll( a | a.rb->size>=lb and a.rb->size<=hb
               or
               a.rb->size>=lb' and a.rb->size<=hb' )
and
B->forAll( b | b.ra->size>=la and b.ra->size<=ha
               or
               b.ra->size>=la' and b.ra->size<=ha' )
```

2.3 Example

Figure 2: Example for Cardinality Notation

The class diagram in Fig. 2 refers to the design of a chess game application. A rook protects at least 2 and at most 14 squares. On the other hand, a square can be protected either by no rook or one rook or two rooks. The formula below is a direct translation of the general scheme into the example context.

```
Rook->forAll( r |
  r.protectedSquare->size>=2 and r.protectedSquare->size<=14 )
and
Square->forAll( s |
  s.protectingRook->size>=0 and s.protectingRook->size<=2 )
```

3 N-ary Associations with Cardinalities

3.1 Statements from UML Material

- *An n-ary association is an association among three or more classes (a single class may appear more than once).* Notation.
- *Multiplicity for n-ary associations may be specified but is less obvious than binary multiplicity. The multiplicity of a role represents the potential number of instance tuples in the association when the other N-1 values are fixed.* Notation.

3.2 General Translation

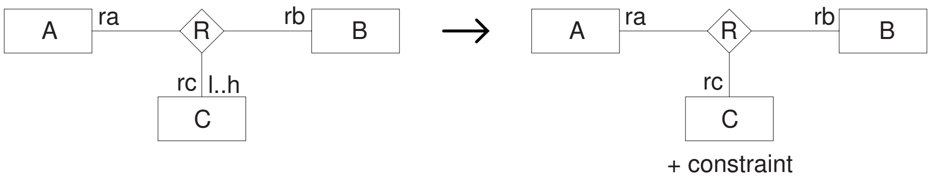

Figure 3: Equivalence Rule for Ternary Association

Ternary Associations: As indicated in the original UML statements, the translation of n-ary associations with cardinality restrictions is a bit more involved in comparison to the binary case. The diagram in Fig. 3 keeps on the right hand side the starting situation except the cardinality requirement. Please note that throughout this paper we assume a multiplicity of * if no explicit specification is given an a class diagram. The formula for the translation fixes the pair of objects on the opposite side of the cardinality specification and requires, analogously to the case with binary associations, that the size of the set of related objects is restricted by the given lower and upper bounds. The formula expresses the connection between three participating objects as `R(a,b,c)`.

```
A->forAll( a |
  B->forAll( b |
    C->select( c | R(a,b,c) )->size>=l
    and
    C->select( c | R(a,b,c) )->size<=h )
  )
)
```

Ternary Associations in OCL: The above constraint does not obey the exact OCL syntax, because the name of an association with appropriate parameters is not allowed as a boolean expression in OCL. It is however necessary to express this, but it cannot be done by using only role names. For example, if we have `R(A,B,C)` and we want to require that `R(a,b,c)` holds for objects `a,b,c`, we cannot express this as

```
a.rb->includes(b) and b.rc->includes(c) and c.ra->includes(a)
```

or

```
a.rb->select( b | b.rc->select(c)->size>0 )->size>0
```

or something similar because

R	A	B	C
	a	b	c'
	a'	b	c
	a	b'	c

fulfills the `includes` and the `select` expression without having an appropriate tuple in `R`.

N-ary Associations: Above, we have shown only the translation for a ternary association. The formula can be generalized to the arbitrary case in Fig. 4 with n participating classes by introducing a universally quantified variable for each of the $n-1$ classes on the opposite side of the cardinality specification.

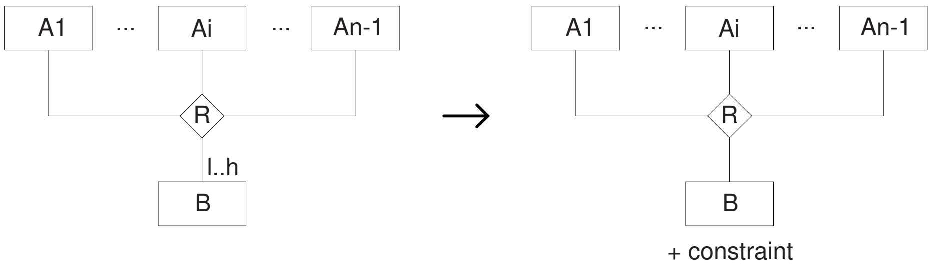

Figure 4: Equivalence Rule for N-ary Association

```
A1->forAll( a1 |
  ...
    Ai->forAll( ai |
      ...
        An-1->forAll( an-1 |
          B->select( b | R(a1,...,ai,...,an-1,b) )->size>=l
          and
```

```
              B->select( b | R(a1,...,ai,...,an-1,b) )->size<=h )
            )
            ...
          )
        ...
      )
```

3.3 Example

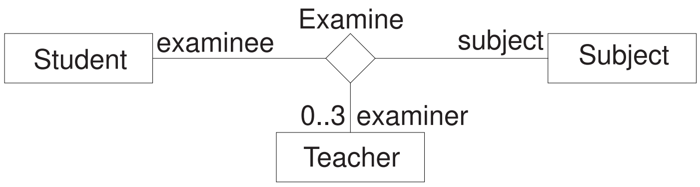

Figure 5: Example for Cardinality Notation in N-ary Association

The example in Fig. 5 shows a ternary association where students can be examined by teachers on certain subjects. The cardinality restriction requires that a student can be examined on a single subject at most three times (each time by a different teacher).

```
Student->forAll( st |
  Subject->forAll( su |
    Teacher->select( t | Examine(st,su,t) )->size>=0
    and
    Teacher->select( t | Examine(st,su,t) )->size<=3
  )
)
```

4 From N-ary to Binary Relationships

4.1 Statements from UML Material

- *An n-ary association is an association among three or more classes (a single class may appear more than once).* Notation.
- *Each instance of the association is an n-tuple of values from the respective classes.* Notation.

4.2 General Translation

Translating N-ary Associations: N-ary associations can be translated strictly to binary associations. Thus, all static structure constructions discussed here could

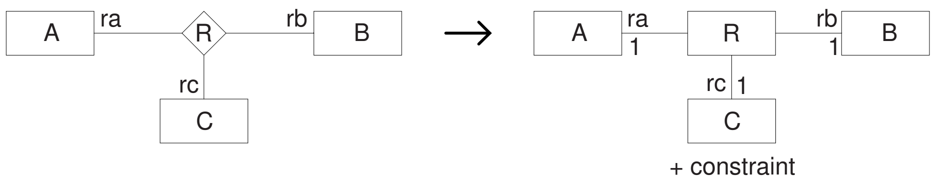

Figure 6: Translation of Ternary Association

finally be reduced to a set of binary associations. As depicted in Fig. 6, a ternary association with unrestricted cardinality translates into three classes plus an additional class representing the association. The cardinalities in the diagram assure that an `R` object, which represents a link, is connected to exactly one `A`, one `B` object, and one `C` object. For representing unique association links we need the following constraint. It states that two different `R` objects are equal, whenever they are connected to the same set of `A`, `B`, and `C` objects. In other words, two links are identical, if they refer to the same source objects.

```
R->forAll( r, r' |
  ( r.ra=r'.ra and r.rb=r'.rb and r.rc=r'.rc ) implies r=r'
)
```

Translating Cardinalities of N-ary Associations: A given cardinality specification `l..h` for the association end (for example) at class `C` can be expressed by the following constraint. The constraint requires that, for a given pair `(a,b)` consisting of an `A` and a `B` object, the set of related `R` objects, i.e. links, is restricted by the given lower and upper bound. The constraint uses operations `ra: R -> A` and `rb: R -> B` which are induced by the role names and the cardinality restrictions.

```
A->forAll( a |
  B->forAll( b |
    R->select( r | r.ra=a and r.rb=b )->size>=l
    and
    R->select( r | r.ra=a and r.rb=b )->size<=h
  )
)
```

Translating Induced Operations of N-ary Associations: It is also possible to give derivations for operations which are induced in the original class diagram by role names. For example, in the original class diagram we have the operation `ra: C -> Set(A)`. For a given `C` object `c` we could form in the original class diagram the term `c.ra` yielding a set of `A` objects connected to object `c`. This term can be expressed in the resulting class diagram as:

```
A->select( a | R->exists( r | r.rc=c and r.ra=a ) )
```

Uniformity versus Semantic Adequacy: Translating complex associations to binary associations has the advantage that the resulting models have a uniform look in that only a small set of features is used and possess equivalent state spaces. This translation has however the drawback that "semantic" information is lost. The notion of semantics here does not refer to formal semantics but to "application" semantics in the following sense: the left class diagram in Fig. 6 clearly states through the explicit association symbol that there are three classes and that there is one association between them; in the right class diagram the information that class `R` is not a "plain" class but really a relationship is only represented implicitly. The translation is in a way analogous to the translation of semantic data model schemata to relational data model schemata. Indeed, the realm of "Semantic Data Models" provides adequate modeling techniques in order to represent different real world phenomena also in different forms in class diagrams (or semantic data model schemata as class diagrams are called there).

4.3 Example

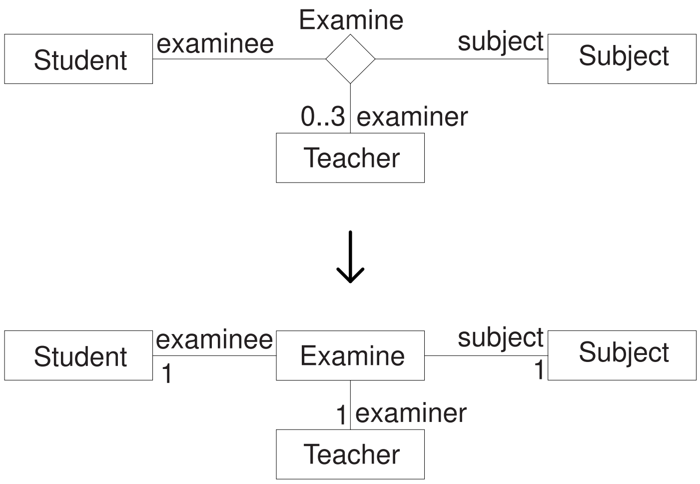

Figure 7: Example for Translation of Ternary Association

In Fig. 7 we show how the Examine example of Fig. 5 is translated. The association translates into a new class, and each "arm" of the old association is transformed into a connection from the new class to one of the original classes. The constraint requires that if two examine objects have the same set of examinees, subjects, and examiners, then the two examine objects coincide.

```
Examine->forAll( e, e' |
  ( e.examinee=e'.examinee and
    e.subject=e'.subject and
    e.examiner=e'.examiner ) implies e=e'
)
```

5 Qualifier

5.1 Statements from UML Material

- *A qualifier is an attribute or list of attributes whose values serve to partition the set of objects associated with an object across an association. The qualifiers are attributes of the association.* Notation.
- *The multiplicity attached to the target role denotes the possible cardinalities of the set of target objects selected by the pairing of a source object and a qualifier value.* Notation.
- *[A] qualifier [is] an association attribute or tuple of attributes whose values partition the set of objects related to an object across an association.* Semantics.

5.2 General Translation

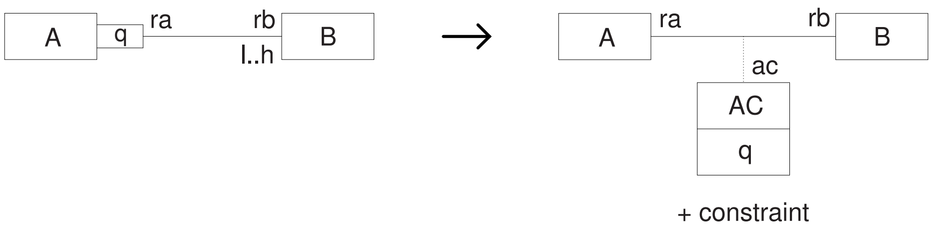

Figure 8: Equivalence Rule for Qualifier

Figure 8 shows that a qualifier is translated into an association class (if the association class is already present, the qualifier translates into an additional attribute of the association class). The respective constraint requires that the size of the set of `B` objects determined by a combination of an `A` object and a `q` attribute value is restricted by the given lower and upper bound. Regarding the operations used in the defining formula, the role name `rb` implies we have an operation `rb: A -> Set(B)`. Due to the additional association class `AC`, we have further operations `ra: AC -> A`, `rb: AC -> B`, `ac: A -> Set(AC)`, and `ac: B -> Set(AC)`.

```
A->forAll( a |
  a.ac->forAll( ac |
    a.rb->select( b |
      b.ac->exists( ac' | ac'.q=ac.q and ac'.ra=a ) )->size>=l
    and
    a.rb->select( b |
      b.ac->exists( ac' | ac'.q=ac.q and ac'.ra=a ) )->size<=h
  )
)
```

In the next section we show how to further reduce association classes to n-ary associations.

5.3 Example

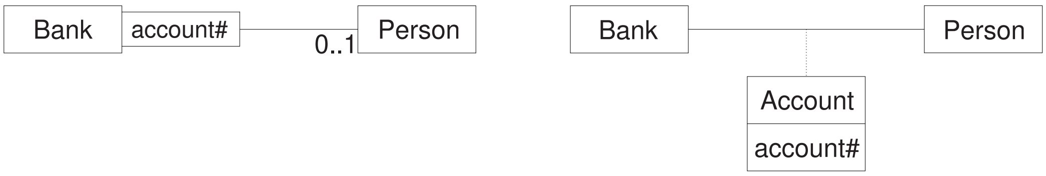

Figure 9: Example for Qualifier

The example in Fig. 9 taken from the UML notatation guide expresses that an account number at a given bank either uniquely determines a person or the account number is not connected to a person at all. The defining formula makes use of implicit role names and induced operations as follows: `person: Bank -> Set(Person)`, `bank: Account -> Bank`, `person: Account -> Person`, `account: Bank -> Set(Account)`, and `account: Person -> Set(Account)`.

The formula expresses the following: (1) For all banks `b` and all accounts `a` at this bank, we select from all persons `p` connected with the bank `b` those who possess an account `a'` at bank `b` having the same account number as account `a`. (2) The size of this set of selected persons is either `0` or `1`.

```
Bank->forAll( b |
  b.account->forAll( a |
    b.person->select( p |
      p.account->exists( a' |
        a'.account#=a.account# and a'.bank=b ) )->size>=0
    and
    b.person->select( p |
      p.account->exists( a' |
        a'.account#=a.account# and a'.bank=b ) )->size<=1
  )
)
```

In this context, the OCL term `b.person[123456]` denotes a person having the account with number `123456` at bank `b`. Therefore, if we denote argument types of a function with underscores _, the notation `_.person[_] : Bank account# -> Person` can be regarded as a partial operation returning for a given bank and a given account number the respective person.

6 Association Classes

6.1 Statements from UML Material

- *An association class is an association that also has class properties (or a class that has association properties).* Notation.

- *An association class is an association that is also a class. It not only connects a set of classifiers [classes] but also defines a set of features that belong to the relationship itself and not any of the classifiers.* Semantics.

- *Navigation from an association class to one of the objects on the association will always deliver exactly one object.* OCL.

6.2 General Translation

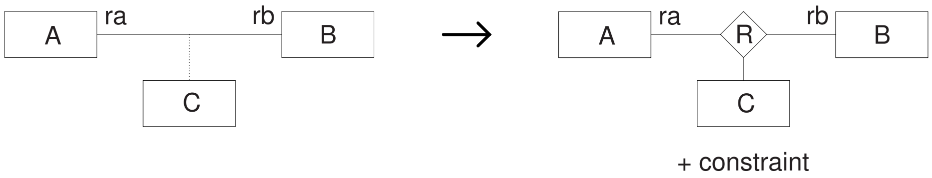

Figure 10: Equivalence Rule for Association Class

In Fig. 10 the translation of association classes into ternary associations is shown. The specified role names induce the operations `ra: C -> Set(A)` and `rb: C -> Set(B)` (and, of course, `ra: B -> Set(A)` and `rb: A -> Set(B)`, but these are not needed in the formula below). The induced formula expresses the idea that each `C` object "points" to a unique pair of `A` and `B` objects. In other words, `R` constitutes an injective function from the class `C` into the product of `A` and `B`.

```
C->forAll( c |
  c.ra->size=1 and c.rb->size=1 and
  C->forAll( c' | (c.ra=c'.ra and c.rb=c'.rb) implies c=c' )
)
```

Alternatively, we can require that (1) each `(A,B)`-pair is connected to exactly one `C` object (the formula below states that for each `(A,B)`-pair the size of the set of connected `C` objects is equal to one) and (2) a pair of `(A,B)` objects is connected to a unique `C` object (the formula below states that if two `C` objects have links to the same pair of `(A,B)` objects, then the two `C` objects coincide).

```
A->forAll( a |
  B->forAll( b |
    ( C->select( c | R(a,b,c) )->size=1 or
      C->select( c | R(a,b,c) )->size=0 )
    and
    C->forAll( c, c' | ( R(a,b,c) and R(a,b,c') ) implies c=c' )
  )
)
```

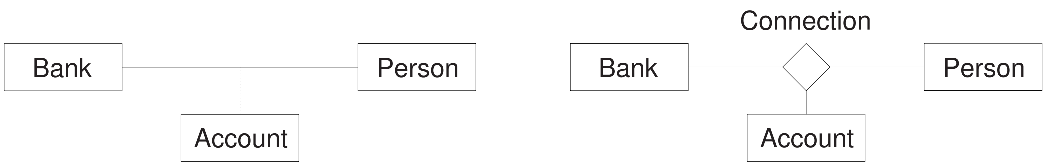

Figure 11: Example for Association Class

6.3 Example

In Fig. 11 the bank example introduced in Fig. 9 is carried out further by paraphrasing it as a ternary association called `Connection` between `Bank`, `Account`, and `Person`. The constraint demands that (1) an account is related to exactly one bank and to exactly one person, and (2) there cannot be a different account with the same links.

```
Account->forAll( a |
  a.bank->size=1 and a.person->size=1
  and
  Account->forAll( a' |
    ( a.bank=a'.bank and a.person=a'.person ) implies a=a'
  )
)
```

The alternative formula would be the following one. This constraint employs the name of the association `Connection` as a predicate symbol.

```
Bank->forAll( b |
  Person->forAll( p |
    ( Account->select( a | Connection(b,p,a) )->size=1 or
      Account->select( a | Connection(b,p,a) )->size=0 )
    and
    Account->forAll( a, a' |
      ( Connection(b,p,a) and Connection(b,p,a') ) implies a=a'
    )
  )
)
```

7 Aggregation and Composition

7.1 Statements from UML Material

- *[A filled diamond] signifies the strong form of aggregation known as composition.* Notation.
- *Composition is a form of aggregation with strong ownership and coincident lifetimes of part with the whole. The multiplicity of the aggregate may not exceed one (it is unshared).* Notation, line -7.

- *[AggregationKind="aggregate"] The part may be contained in other aggregates.* Semantics.
- *[AggregationKind="composite"] The part is strongly owned by the composite and may not be part of any other composite.* Semantics.
- *Aggregation [is] a special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part.* Semantics.
- *Composition [is] a form of aggregation with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it (i.e. they share lifetimes). Such parts can also be explicitly removed before the death of the composite.* Semantics.
- *Both kinds of aggregations define a transitive, antisymmetric relationship, i.e. the instances form a directed, non-cyclic graph. Composition instances form a strict tree (or rather a forest).* Semantics.

7.2 General Translation

Aggregate-part (or whole-part) relationships touch three areas to be discussed: Existential dependency, sharing, and instance reflexivity. (1) In principle, existential dependency comes in two facets: The part can existentially depend on the aggregate (a part can only exist inside an aggregate), or, the other way round, the aggregate can existentially depend on the part (an aggregate can only exist if its parts exist). (2) Forbidding sharing means that two aggregates cannot have a part in common. With respect to sharing, one can further distinguish the situation where the two aggregates are objects of the same class from the situation where it is possible that the two aggregates belong to different classes. (3) Forbidding instance reflexivity means that a part cannot directly or indirectly be part of itself.

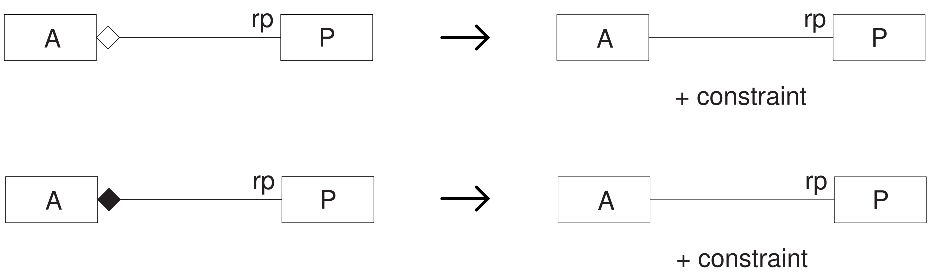

Figure 12: Equivalence Rule for Aggregation and Composition

As indicated in Fig. 12, both UML versions of aggregate-part relationships, namely aggregations and compositions, are translated into binary associations. General associations differ from aggregate-part relationships in that instance reflexivity is forbidden for aggegrations and compositions. For compositions we require in addition the part to be existentially dependent from the aggregate and a strong form of forbidding sharing. Below, we define the facets of aggregate-part relationships independent from these UML requirements as OCL constraints.

Existential Dependency for the Part: The part is existentially dependent from the aggregate. In technical terms, this means that a `P` object can only exist if it is connected to an `A` object.

```
P->forAll( p |
  A->exists( a | a.rp->includes(p) ) )
```

Existential Dependency for the Aggregate: The aggregate is existentially dependent from the part. In technical terms, this means that an `A` object can only exist if it is related to a `P` object.

```
A->forAll( a |
  P->exists( p | a.rp->includes(p) ) )
```

Both forms of existential dependency can be expressed alternatively as a cardinality constraint with a lower bound of one.

Combining Both Forms of Existential Dependency: Both forms of existential dependency can be combined into one notion: Part and aggregate can be existentially coincident. In technical terms, `A` and `P` objects are existentially coincident.

```
P->forAll( p |
  A->exists( a | a.rp->includes(p) ) )
and
P->forAll( p |
  A->exists( a | a.rp->includes(p) ) )
```

Weak Form of Forbidding Sharing: A `P` object cannot be shared by two different `A` objects. In technical terms, if two aggregates comprise one common part, then the aggregates coincide.

```
P->forAll( p |
  A->exists( a, a' |
    ( a.rp->includes(p) and a'.rp->includes(p) )
    implies
    a=a' ) )
```

Strong Form of Forbidding Sharing: A `P` object cannot be shared by two different objects belonging to (potentially) different classes `A1` and `A2`. As shown in Fig. 13, this constraint has to be given for any two classes `A1` and `A2` being potential aggregates for class `P`.

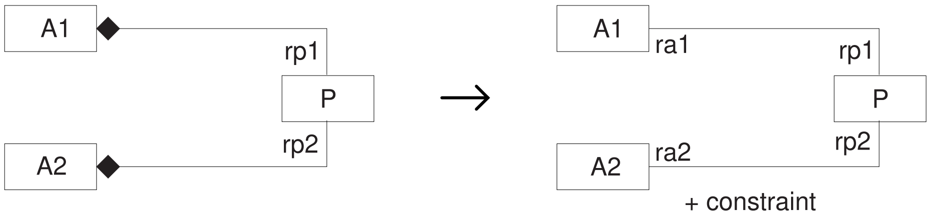

Figure 13: Equivalence Rule for Composition with Strong Form of Forbidding Sharing

```
P->forAll( p |
  A1->exists( a1 |
    A2->exists( a2 |
      ( a1.rp1->includes(p) and a2.rp2->includes(p) )
      implies
      a1=a2 ) ) )
```

Although in the above formula `a1` and `a2` may be objects of different classes, the comparison `a1=a2` is syntactically well-formed, because in OCL we have `A1<Any` and `A2<Any`, and `=: Any x Any -> Bool`.

As an alternative, but equivalent formulation we can set up a constraint requiring that a part `p` cannot have connections to both an `A1` object and an `A2` object.

```
P->forAll( p | p.ra1->isEmpty or p.ra2->isEmpty )
```

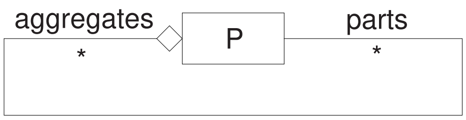

Figure 14: Reflexive Aggregation

Forbidding Instance Reflexivity for Aggregation: A `P` object cannot be part of itself. In Fig. 14 we have a reflexive aggregation (one class participates twice) where the role names define operations `parts: P -> Set(P)` and `aggregates: P -> Set(P)`. Without any further restriction, a `P` object `p` can be a *direct* part of itself, i.e. `p.parts->includes(p)` is possible. The formula below disallows this. It goes even further by forbidding that p is an *indirect* part of itself by using the transitive closure of the operation `parts`, here called `partsClosure: P -> Set(P)`. The term `partsClosure(p)` yields the parts of `p`, the parts of parts of `p` and so on. This intuition is reflected by the following "informal" definition.

```
partsClosure(p) =
  parts(p) +
```

```
  [ parts(p') | p.parts->includes(p') ] +
  [ parts(p'') | p.parts->includes(p') and
                 p'.parts->includes(p'') ] +
  ...
```

If we want to define the transitive closure formally as in the defintion below, we have to use the iterate function of OCL. The operation `partsClosure` has one parameter, the part whose (sub-)parts we want to calculate, and it returns a set of parts. First, we consider all direct parts of the given part and initiate an iteration over this set with the variable `p`. In the variable `acc` (which is able to hold a set of parts) we accumulate the result step by step beginning with the direct parts of the given part. If the considered part in the "iteration" variable `p` is already present in the accumulator `acc`, the accumulator does not change. If the value of `p` is not already in `acc`, we build the union of the accumulator and the result of the recursive application of `partsClosure` to the value of `p`.

```
partsClosure(start : P) : Set(P) =
  start.parts->iterate(
    p : P;
    acc : Set(P) = start.parts;
    if acc.includes(p)
      then acc else acc.union(p.partsClosure) endif
  )
```

After having defined the transitive closure of `parts` formally, we can now simply state the requirement that a part cannot contain itself as a part by forbiddding that a part lies in `partsClosure` applied to the part.

```
P->forAll( p | not( p.partsClosure->includes(p) ) )
```

The situation in Fig. 14 is more or less the easy to understand situation where it is syntactically possible that a part can be a part of itself. The situation can be more involved if the reflexivity on the class diagram level is established with some intermediate steps as shown in Fig. 15. But nevertheless, the corresponding definition can be generalized to this case.

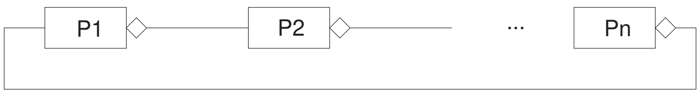

Figure 15: General Case for Reflexive Association

Another complication comes into the picture because aggregation and generalization (see below) can interact with each other. As the left side of Fig. 16 shows even if there is no direct reflexive aggregation, due to the syntactic specialities of generalizations, a part could be a part of itself in this context. As indicated in

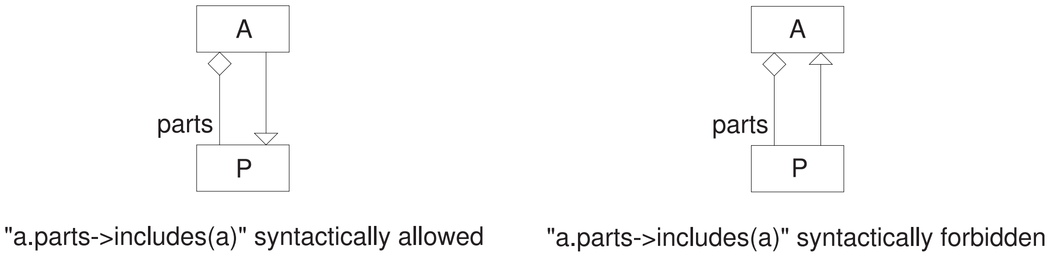

Figure 16: Combination of Aggregation and Generalization

the right side of Fig. 16, if the generalization goes into the "other" direction, this complication does not occur. Through examples following the schema sketched in Fig. 17 we will see that the left side of Fig. 16 is a common situation when one models with both aggregation and generalization.

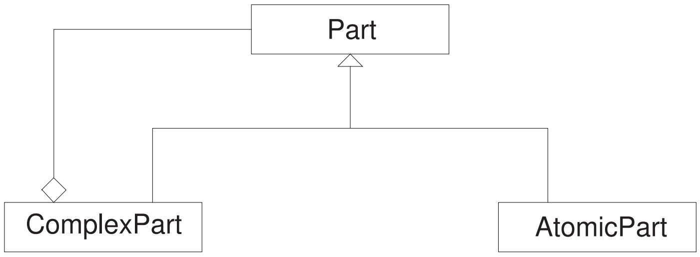

Figure 17: General Example for Combination of Aggregation and Generalization

7.3 Example

7.3.1 The "Paper" Example

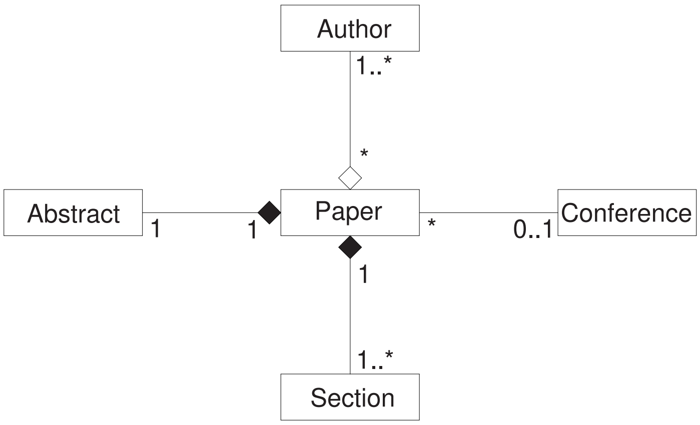

Figure 18: Example for Ordinary Association, Aggregation, and Composition

The example in Fig. 18 shows one ordinary association, one aggregation, and two compositions. As an example for an ordinary association, we see that a paper can be connected to a conference (for instance, submitted to a conference). The components of a paper are described by aggregation and composition. Thus a paper consists of (1) one or more authors, (2) exactly one abstract, and (3) one or more sections. The association between a paper and an author is classified as an aggregation, the ones connecting a paper with abstracts and sections as compositions. This means two different papers can share an author, but an abstract and the paper's sections exclusively belong to one paper, thus sharing is not possible. However, a paper has coincident lifetime with its components (authors, abstract, and sections), but a paper can exist without any connection to a conference (and conferences can exist without being connected to a paper). Please note, instance reflexivity is not applicable in this example.

Existential Dependency for the Aggregate: A paper is existentially dependent from its components, i.e. a paper cannot exist without authors, abstract, and sections.

```
Paper->forAll( p |
  Author->exists( au | p.author->includes(au) ) )
Paper->forAll( p |
  Abstract->exists( ab | p.abstract->includes(ab) ) )
Paper->forAll( p |
  Section->exists( s | p.section->includes(s) ) )
```

Existential Dependency for the Part: A paper component which is classified by composition is existentially dependent from the paper, i.e. an abstract or a section cannot exist without a corresponding paper.

```
Abstract->forAll( a |
  Paper->exists( p | p.abstract->includes(a) ) )
Section->forAll( s |
  Paper->exists( p | p.section->includes(s) ) )
```

We have taken the same restricting formula inside the existentially quantified formula for the whole-dependent-from-part and the part-dependent-from-whole cases because, for example, `p.author->includes(au)` is equivalent to `au.paper->includes(p)`.

Forbidding Sharing for Composition: An abstract and a section cannot be shared by two different papers. In this example the weak and the strong form of forbidding sharing coincide because neither `Abstract` nor `Section` participate as parts in other compositions.

```
Abstract->forAll( a |
  Paper->exists( p, p' |
    ( p.abstract->includes(a) and p'.abstract->includes(a) )
```

```
      implies
      p=p' ) )
  Section->forAll( s |
    Paper->exists( p, p' |
      ( p.section->includes(s) and p'.section->includes(s) )
      implies
      p=p' ) )
```

7.3.2 The "Brain" Example

The class diagram in Fig. 19 attempts to show in a single domain three kinds of relationships: plain association, aggregation, and composition. Please try to take it with a portion of humor. The diagram shows the classes `Human` and `Brain`. The association `KnowsByName` reflects that humans know other humans by name. In general, this association has no special properties. For example, it is not reflexive, symmetric, or transitive. The aggregation involved is the `Fatherhood` relationship. In a mental sense, the child can be considered as a part of the father. But what is more interesting from the technical point of view, the fatherhood relationship (and its transitive closure) is not reflexive, i.e. nobody is the father of himself. The composition `Possession` shown between `Human` and `Brain` reflects that humans exclusively possess brains: sharing of brains is not possible. We discuss the formalities of the induced formulas and some object states for the diagram below.

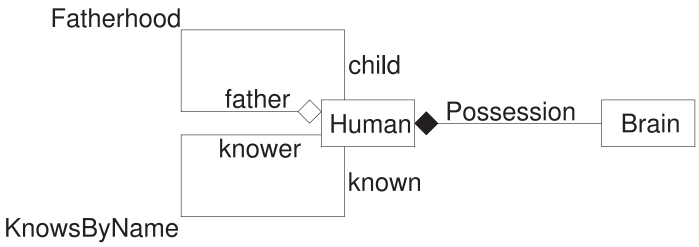

Figure 19: Brain Example - Class Diagram

Existential Dependency for Composition: All brains are possessed by humans.

```
  Brain->forAll( b |
    Human->exists( h | h.brain->includes(b) ) )
```

Forbidding Sharing for Composition: A brain is possessed by only one human.

```
  Brain->forAll( b |
    Human->exists( h, h' |
      ( h.brain->includes(b) and h'.brain->includes(b) )
      implies
      h=h' ) )
```

Forbidding Instance Reflexivity for Aggregation: A human cannot be a descendant of herself or himself. For the formalization of this constraint we need the transitive closure of the operation `child: Human -> Set(Human)`. The operation `child` yields all children of a given father, and, in consequence, the operation `childClosure: Human -> Set(Human)` returns for a given father his children plus his children's children and so on. This operation is a derived operation for the class `Human`.

```
Human->forAll( h | not( h.childClosure->includes(h) ) )
```

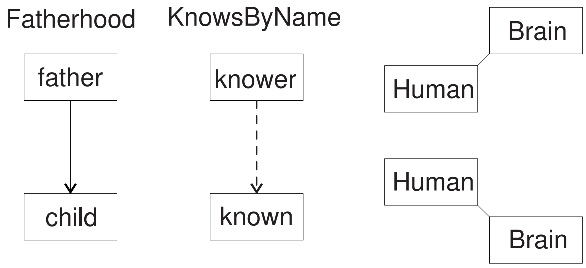

Figure 20: Brain Example - Object Diagram Conventions

The object diagrams hereafter follow the layout conventions sketched in Fig. 20: `Fatherhood` links are denoted by solid arrows, `KnowsByName` links by dotted arrows, and `Possession` links are visualized by showing the brain object in the upper or lower right diagonal of the corresponding human.

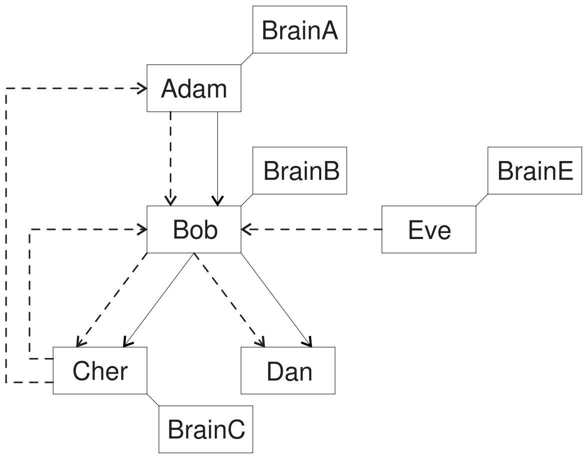

Figure 21: Brain Example - Valid Object Diagram

In Fig. 21 we have depicted a *valid* object diagram for the class diagram in Fig. 19. As we see there, if a human has a brain, then it has its own brain, i.e. brain sharing is not possible. However, there are humans without brains, but there are no brains without humans. Two different humans can have the same father, i.e. sharing is possible for

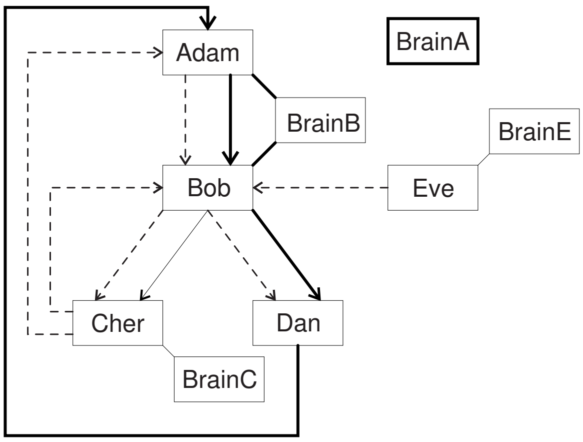

Figure 22: Brain Example - Invalid Object Diagram

`Fatherhood` links. For `KnowsByName` links, arbitrary (even cyclic) connections are allowed.

In Fig. 22 we have depicted an *invalid* object diagram for the class diagram in Fig. 19. There are three reasons for this invalidity (in the diagram we have shown the respective parts with thick lines). (1) There is a brain which is not possessed by a human. (2) There is a brain (`brainB`) shared by two humans. (3) There is a cycle (`Adam -> Bob -> Dan -> Adam`) in the `Fatherhood` links. These three items correspond to violations against the three above given constraints (in that order). To yield a valid object diagram, all three violations must be removed.

8 Generalization

8.1 Statements from UML Material

- *Generalization is the taxonomic relationship between a more general element and a more specific element that is fully consistent with the first element and that adds information.* Notation; Semantics.
- *An instance of the more specific element may be used where the more general element is allowed.* Semantics.
- *The following constraints are predefined. Overlapping: A descendent may be descended from more than one of the subclasses. Disjoint: A descendent may not be descendent from more than one of the subclasses. Complete: All subclasses have been specified (...); no additional subclasses are expected. Incomplete: Some subclasses have been specified but the list is known to be incomplete.* Notation.

8.2 General Translation

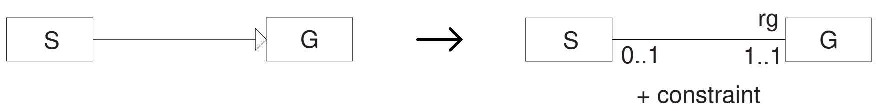

Figure 23: Equivalence Rule for Generalization

Simple Generalizations: As shown in Fig. 23, UML generalizations are transformed to special binary associations. The cardinalities make sure that each specialized object is related with exactly one general object, although not necessarily every general object has a link to a special object. In other words, we have a total mapping from special to general objects. An additional constraint assures the injectivity of the mapping, or, in other words, a special object is associated with a unique general object (no two special objects are associated with the same general object). The defining formula employs `rg` as the role name with an induced operation `rg: S -> G`.

```
S->forAll( s, s' | s<>s' implies s.rg<>s'.rg )
```

This simple translation for generalizations does even handle type substitutability in the context of subtype polymorphism: Wherever an object of the general class `G` is expected (for example as an argument of an operation), we can substitute a specialized object of class `S` after applying the "type cast" `rg`. For example, if `op: boolean` is an operation in class `G`, the expression `s.rg.op` would be allowed for the `S` object `s`.

Disjoint Generalizations: Generalization relationships can further be classified as disjoint or complete. We explain these classifications for the case with two subclasses. The cases with more subclasses can be developed analogously. A disjoint generalization has to partition specialized classes into disjoint sets. We can realize this requirement by demanding that any two special objects from different classes map into different objects in the general class.

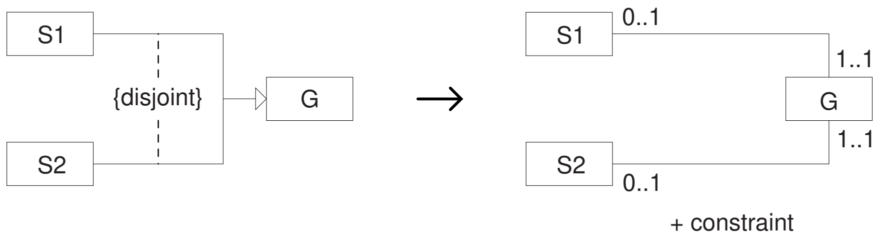

Figure 24: Equivalence Rule for Disjoint Generalization

```
S1->forAll( s1 | S2->forAll( s2 | s1.g<>s2.g ) )
```

Complete Generalizations: Complete generalizations serve to describe the case that a general class completely covers its subclasses. It is forbidden that there is

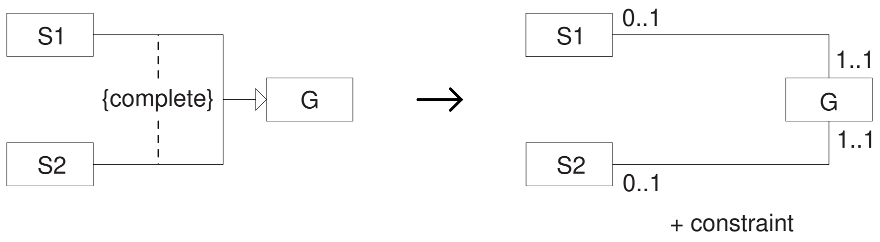

Figure 25: Equivalence Rule for Complete Generalization

a general object which is not represented as a special object in one of the special classes.

```
G->forAll( g |
  S1->exists( s1 | s1.g=g )
  or
  S2->exists( s2 | s2.g=g )
)
```

Overlapping and Incomplete Generalizations: The other syntactic possibilities, namely overlapping and incomplete generalizations, can be regarded as the default cases which hold when no further classification about the nature of the generalization is made. No constraints have to be given under these circumstances. Overlapping corresponds to "not disjoint" and incomplete (of course) to "not complete". Thus, overlapping does not mean that the respective sets actually must overlap but that they are allowed to do so.

8.3 Example

First, we explain a simple example where vehicles are specialized to cars. Afterwards, we give class diagrams for the four different combinations of disjoint and complete generalizations:

Disjoint	Complete	Example
+	+	Female, Male → Person
+	−	Oak, Elm, Birch → Tree
−	+	LandVehicle, WaterVehicle, AirVehicle → Vehicle
−	−	I-Citizen, F-Citizen → Person

8.3.1 The "Car" Example

The example in Fig. 26 shows a specialization of vehicles to cars. The constraint requires, that two given distinct cars are mapped to distinct vehicles by the "type cast" `vehicle` (the operation induced by the association).

```
Car->forAll( c, c' | c<>c' implies c.vehicle<>c'.vehicle )
```

Figure 26: Example for Generalization

8.3.2 The "Female/Male" Example

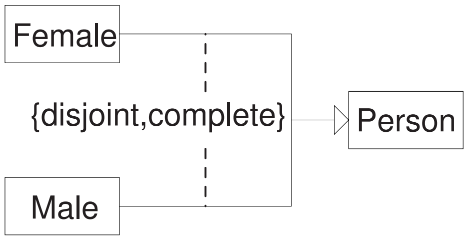

Figure 27: Example for Disjoint and Complete Generalization

The class diagram in Fig. 27 represents a disjoint and complete classification, i.e. a partitioning, of persons by their sex into female and male persons. First we give the constraint for the disjointness condition and afterwards the formula for the completeness requirement.

```
Female->forAll( f |
  Male->forAll( m | f.person<>m.person ) )
```

```
Person->forAll( p |
  Female->exists( f | f.person=p ) or
  Male->exists( m | m.person=p ) )
```

8.3.3 The "Oak/Elm/Birch" Example

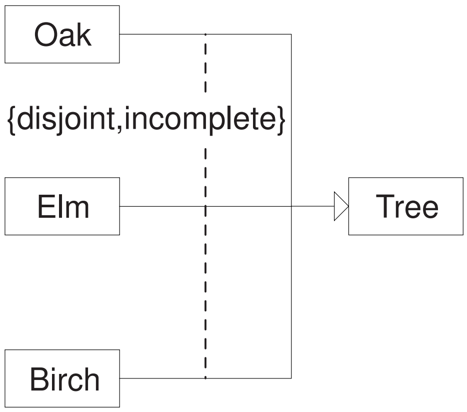

Figure 28: Example for Disjoint and Incomplete Generalization

This example taken from the UML notation guide specializes trees to oaks, elms, and birches. It is a disjoint but incomplete generalization because, for instance, palms are trees but they are not covered by any of the special classes. The constraint formulates the disjointness condition pairwise for each two classes.

```
Oak->forAll( o |
  Elm->forAll( e |
    Birch->forAll( b |
      o.tree<>e.tree and
      o.tree<>b.tree and
      e.tree<>b.tree ) ) )
```

8.3.4 The "Amphibian Vehicle" Example

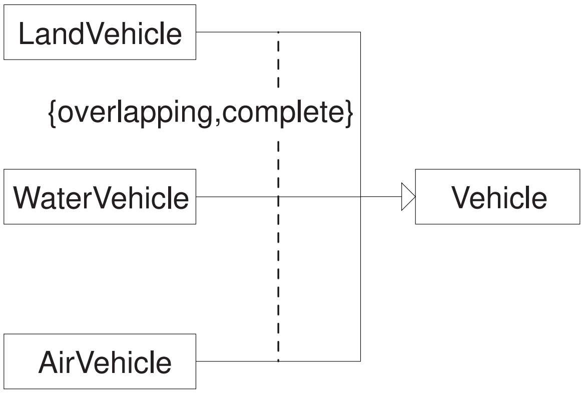

Figure 29: Example for Overlapping and Complete Generalization

The example in Fig. 29 is the inevitable amphibian vehicle example. It represents an overlapping and complete generalization. It is overlapping due to the existence of, say, planes who are able to land or water. It is complete because land, air, and water are the only "media" where transportation can take place on earth. The formula requires that each vehicle is also represented as a land, water, or air vehicle.

```
Vehicle->forAll( v |
  LandVehicle->exists( l | l.vehicle=v ) or
  WaterVehicle->exists( w | w.vehicle=v ) or
  AirVehicle->exists( a | a.vehicle=v ) )
```

8.3.5 The "Citizenship" Example

The example in Fig. 30 describes specializations of class `Person` to Italian and French citizens. No restricting constraint has to be given in this case. The class diagram is overlapping because people are allowed to have both the Italian and the French citizenship. Assuming a "European" context, it is incomplete because, for example, Spanish people are not covered.

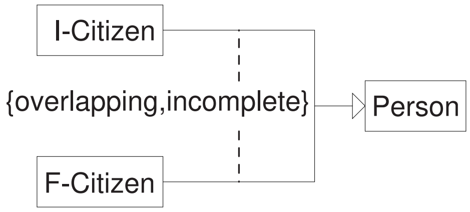

Figure 30: Example for Overlapping and Incomplete Generalization

9 Conclusion

We have achieved guiding rules for UML designers in order to cope with the UML diagram variety. Our approach can be seen as a way to give semantics to an advanced UML language layer. What remains to be done is to give semantics to the "low-level" UML layer by stating a translation into an abstract model. Such a model together with a semantics for the UML constraint language OCL has already been worked out [RG98].

We have translated all discussed UML features into n-ary associations with additional constraints. Due to the modeling power of associations and OCL this seems natural, especially because the association concept is a very general one which is able to model many situations. But one can go even further and transform all n-ary associations into an additional class (plus the classes given before) resolving the n-ary association into n binary associations. This is subject to further consideration.

References

[BC95] R. Bourdeau and B. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, 1995.

[BCV96] E. Bertino, D. Castelli, and F. Vitale. A Formal Representation for State Diagrams in the OMT Methodology. In K.G. Jeffery, J. Kral, and M. Bartosek, editors, *Proc. Seminar Theory and Practice of Informatics (SOFSEM'96)*, pages 327–341. Springer, Berlin, LNCS 1175, 1996.

[BHH+97] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a Formalization of the Unified Modeling Language. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. 11th European Conf. Object-Oriented Programming (ECOOP'97)*, pages 344–366. Springer, Berlin, LNCS 1241, 1997.

[BLM97] J.C. Bicarregui, Kevin Lano, and T.S.E. Maibaum. Objects, Associations and Subsystems: A Hierarchical Approach to Encapsulation. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. 11th European Conf. Object-Oriented Programming (ECOOP'97)*, pages 324–343. Springer, Berlin, LNCS 1241, 1997.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, Redwood City, 1994.

[Cla99] Tony Clark. Type checking UML static diagrams. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 503–517. Springer, 1999.

[Ebe97] Jürgen Ebert. Integration of Z-Based Semantics of OO-Notations. In Haim Kilov and Bernhard Rumpe, editors, *Proc. ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques.* Technische Universität München, Informatik-Bericht TUM-I9725, 1997.

[FBLPS97] R. France, J.M. Bruel, M. Larrondo-Petrie, and M. Shroff. Exploring the Semantics of UML type structures with Z. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 247–260. Chapman and Hall, London, 1997.

[GPP98] Martin Gogolla and Francesco Parisi-Presicee. State Diagrams in UML: A Formal Semantics using Graph Transformations. In Bernhard Rumpe, Manfred Broy, Derek Coleman, and Tom S.E. Maibaum, editors, *Proc. ICSE'98 Workshop on Precise Semantics of Modeling Techniques (PSMT'98)*, 1998. `http://www4. informatik. tu-muenchen. de/ rumpe/icse98-ws/`.

[JCJÖ92] Ivar Jacobsen, Magnus Christerson, Patrik Jonsson, and G.G. Övergaard. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[KC99] Soon-Kyeong Kim and David Carrington. Formalizing the UML class diagram using object-z. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 83–98. Springer, 1999.

[Lan96] Kevin Lano. Enhancing Object-Oriented Methods with Formal Notations. *Theory and Practice of Object Systems*, 2(4):247–268, 1996.

[OMG99a] OMG, editor. *Object Constraint Language (Version 1.3).* OMG, 1999. `http://www.omg.org`.

[OMG99b] OMG, editor. *UML Notation Guide (Version 1.3).* OMG, 1999. `http://www.omg.org`.

[OMG99c] OMG, editor. *UML Semantics (Version 1.3).* OMG, 1999. `http://www.omg.org`.

[Öov98] Gunnar Öovergaard. A Formal Approach to Relationships in The Unified Modeling Language. In Bernhard Rumpe, Manfred Broy, Derek Coleman, and Tom S.E. Maibaum, editors, *Proc. ICSE'98 Workshop on Precise Semantics of Modeling Techniques (PSMT'98)*, 1998. `http://www4. informatik. tu-muenchen. de/ rumpe/icse98-ws/`.

[RACH00] Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hussmann. Analysing UML active classes and associated state machines – A lightweight formal approach. In Tom Maibaum, editor, *Proc. Fundamental Approaches to Software Engineering (FASE 2000), Berlin, Germany*, volume 1783 of *LNCS*. Springer, 2000.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[RG98] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, pages 449–464. Springer, Berlin, LNCS 1507, 1998.

[SF97] M. Shro and R. B. France. Towards a Formalization of UML Class Structures in Z. In *Proc. 21st Annual Int. Computer Software and Applications Conference (COMPSAC'97)*, pages 646–651. IEEE, 1997.

[TE00] A. Tsiolakis and H. Ehrig. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer, editors, *Proc. of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, Berlin, March 2000*, 2000. Technical Report no. 2000/2, Technical University of Berlin.

[WB98] Roel Wieringa and Jan Broersen. A minimal transition system semantics for lightweight class- and behavior diagrams. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.

[WK98] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[WK99] Jos Warmer and Anneke Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, May 1999.

[WRC97] Enoch Y. Wang, Heather A. Richter, and Betty H. C. Cheng. Formalizing and Integrating the Dynamic Model within OMT. In *Proc. 19th Int. Conf. on Software Engineering (ICSE'97)*, pages 45–55. ACM Press, 1997.