Department of Computer Science
University of Toronto

Technical Report CSRG-566

Mappings, maps, atlases and tables: a formal semantics for associations in UML 2 ⋆

Zinovy Diskin[1]

zdiskin@cs.toronto.edu

Abstract. In fact, UML2 offers two related yet different definitions of associations. One is implicit in several explanatory sections of the Standard and belongs to UML folklore. It basically says that an association is a set of navigation paths/mappings between the participating classes. The other – official and formal – definition is explicitly fixed by the UML metamodel and shows that there is much more to associations than just mapping. Particularly, association ends can be owned by either participating classes or by the very association, be navigable or not, be unique or not, and may be optionally qualified.
The paper presents a formal framework, based on sets and mappings, in which all notions involved in the both definitions can be accurately explained and formally expounded. Our formal model allows us to reconcile the two views of associations, to present the construct in a remarkably symmetric and unified way and, finally, to detect a few flaws in the association part of the UML2 metamodel.

⋆ Research supported by IBM Eclipse Innovation Grant

> Associations are the glue that ties a system together. Without associations, there are nothing but isolated classes that don't work together.
>
> *James Rumbaugh, Ivar Jacobson and Grady Booch, "The UML Reference Manual", [18]*

1 Introduction

As the epigraph states, associations are amongst the most important modeling constructs. A clear and accurate formal semantics for them would provide guidance for a convenient and precise syntax, and greatly facilitate their adequate usage. Moreover, in the context of model-driven software development, semantics must be crystal clear and syntax has to specify it in an unambiguous and suggestive way. An additional demand for clarifying the meaning of associations comes from UML2 metamodel that is based on binary associations.

Unfortunately, the UML2 specification [1], further referred to as the Standard, does not satisfy these requirements. While complaints about informality of semantics are common for many parts of UML, for associations even their (abstract) syntax fixed in the metamodel seems to be complicated and in some parts really obscure. For example, the meaning of the (meta)associations *ownedEnd* and *navigableOwnedEnd* of the Association (meta)class in the metamodel, and the relationships between ownership and navigability in general, are not clear. In the newest version of the Standard [1], navigability and ownership are declared to be orthogonal concepts (which is explicitly stated as to be in contrast with the previous version of UML2 [14]) while the metamodel is kept unchanged. Surprisingly, it shows that the metamodel is not even intended to follow the conceptual framework and precisely specify it.

The Standard also states that the relationship between ownership and navigability for binary associations is principally different from the case of multiary ($n \geq 3$) associations. In fact, it means that the very definition of Association splits into the binary and multiary cases. The infamous multiplicity problem for multiary associations [9] is another point where the cases of binary and multiary associations are qualitatively different in UML. Also, in the family of constructs related to Association, the *qualified* association appears to be something distinct and treated separately from the binary and multiary cases, which makes the fragmentation even worse. Last but not least, if an Association is a collection of Properties (ends, mappings), i.e., something having a direction, how can it be the classifier for the corresponding *set* of links, i.e., something symmetric and non-directional?

Many important aspects of Association (uniqueness, multiplicities, navigability) are still debated in the community (e.g., [**?**], [13],[2]), and the corresponding part of the Standard is not stable (e.g., the recent revision mentioned above). A sign of a general distortion of the association part of the metamodel is that many modeling tools do not implement multiary associations, not to mention qualified associations - a rarity among the implemented modeling elements.

We will show in the paper that all these problems grow from the same root, and can be fixed as soon as the root problem is recognized and fixed. Roughly

speaking, UML mixes up three different sides of the association construct and for *three* related yet *different* sets containing in total $n+n+2n$ modeling constructs, offers *one* n-element set of terms and formal notions defined in the metamodel. It implies that the same basic term/notion of memberEnd in different parts of the standard implicitly refers to different modeling constructs. As a rough analogy, let us consider using the same term "cylinder" for the following three constructs: a 3D-solid, its net surface and its cylindric surface. Perhaps such an ambiguity may be acceptable in a general discussion (think of the conceptual modeling stage of software design), but technical questions like computing the area of the "cylinder" or its weight (design and implementation) need a precise definition of what is meant by "cylinder". (To make the picture more dramatic, the reader may think of a negotiation process between three parties, each one with its own understanding of the term).

Of course, in real life such ambiguities would hardly be a serious problem. As soon as the first signs of different understandings of the notion by the parties would appear, ambiguities could be easily resolved by presenting a physical model or drawing of a cylinder and pointing out what each party means. It would at once result in precise definitions of the three different notions of the "cylinder", say, Cylinder 1, Cylinder 2, Cylinder 3, and the relationships between them (see Fig. 1). For further references, we call this phenomenon the *Cylinder Syndrome*[1]

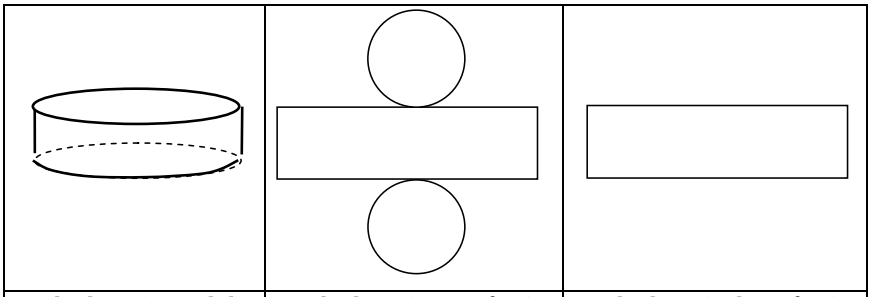


Fig. 1. The *Cylinder Syndrome*: Three meanings of the same term.

[1] Perhaps such a detailed exposition of our cylinder analogy would not be worth the efforts, but in the paper we will encounter this sort of problem several times, and having a brief term for referencing would be convenient. Moreover, the issue appears in other parts of UML and other modeling languages too, and thus acquires the status of a general phenomenon in the modeling world. Therefore we have made an attempt to coin a general term for it: naming means recognition and it is the first, and often crucial, step to solution.

Unfortunately, the simple and reliable method of bringing or drawing a physical cylinder does not work for conceptual constructs ("conceptual Cylinders") like Association. However, we can try to build a formal model of the concept explicating its different understandings in formal terms. Then, if our formal model is well designed and rich enough, each party could recognize and show in the model that party's view of the concept. It appears to be a reasonable (if not the only) way to manage the problem, and this is what we are going to do in the paper. Although the construct of Association is simple enough conceptually, its technical version described by the Standard is not a trivial subject to model and, as far as we know, has not been adequately formalized yet. Hence, our goal is to build a formal model for Association that would be rich enough to explain the construct as defined by the Standard yet comprehensible and mathematically sound.

A word of caution is in order. Formalities as such can be either boring or interesting to play with. When they are intended to model engineering artifacts, the first and crucial requirement for them is to be an adequate and careful formalization of the intuitions behind the artifacts to be modeled. We have paid close attention to deducing our formalization from the Standard rather than from our own perception of what the association should be. To achieve this goal, we have scrutinized the metamodel and read the accompanying explanatory sections of the Standard as carefully as possible. That is why there are many quotations in the paper (they are typed with an indentation and supplied with numbers (Q1),(Q2).. on the left; our own explanatory words within the quotes are placed in square brackets). In addition to reading the Standard, we have discussed possible interpretations with the experts [19, 13] and tried to listen to what might be called the *UML spirit*. The latter is presented in the explanatory parts of the Standards (collected mainly in the Semantics, Description and Notation Sections) and in the UML folklore too. Finally, in cases of essential discrepancies between our formal model and the UML metamodel, we have tried to find out their possible causes. Finding a reasonable *semantic* explanation of why the Standard deviates from a formal model does serve as another justification for adequacy of the latter.[2]

Our plan for the paper is as follows. In the next section we consider in detail an example illustrating the origin of the UML2 association problems and how we are going to fix them. Sections 3 and 4 present the results of our reading and interpreting the Standard, and making it precise in formal terms. Section 3 deals with purely structural aspects without implementation concerns, which are, in turn, are considered in Section 4. In both section, we follow basically the same pattern: analyze the UML metamodel and the respective explanatory parts of the Standard, build a corresponding mathematical model, express it

[2] We recognize that for such an organizationally non-trivial enterprize as standardization of the software industry and UML, there are many factors influencing and shaping the result (political, cultural, personal), which are entirely beyond our investigation. We work with and only with the mathematical substance and engineering intuition underlying the subject.

diagrammatically as a metamodel, render the latter in the UML metamodeling style, and finally, compare the result with the UML metamodel. These comparisons are instructive and their results are presented in sections 3.4,4.4, 4.7. Note also our new consistent and unambiguous notation for associations proposed in Table 3.

Nevertheless, models built in sections 3 and 4 are not quite satisfactory formally and mathematically in that they follow two unfortunate features of the UML metamodeling. The first is that syntax and semantics are not strictly separated. The second is that working with labeled structures goes through labeled bags, and this is not a quite adequate and accurate notion. These two deficiencies are fixed in section 5, where we built a formal model of UML Association according to mathematical standards, and then apply our analysis pattern once again. The result is our metamodel of the association construct in Fig. 8: it is mathematically justified, endowed with a formal semantics and arranged in the UML metamodeling style.

There are a few other aspects of our metamodel to be mentioned. It is built on base of the explanatory sections of the Standards and, in fact, is nothing but an accurate formal explication of the underlying intuitions. Nevertheless, the metamodel is surprisingly compact and observable. It shows enough similarity with the UML metamodel to make the comparison possible, and on the other hand, it shows enough differences with the UML metamodel to make the comparison interesting and productive for UML.

The results of our comparative analysis are remarkable: while informal definitions in the explanatory sections of the Standard are more or less consistent and can be mathematically interpreted and formalized, their specification in the UML metamodel is inaccurate, incomplete and, in fact, inconsistent. This discovery is somewhat astonishing since it is the metamodel that should cast the intuition, or at least a part of it, into a precise specification to be used by tool vendors when they implement UML. The comparative analysis line of the paper is summarized in section 6. We not only demonstrate what is distorted in the UML metamodel of Association but also try to figure out general problems of UML metamodeling that could cause this distortion, and suggest a corresponding treatment.

The reader looking for a shortcut to the final results could look briefly through section 2 and then jump over sections 3 and 4 to section 5 and the final discussion. The reader aiming at detailed motivation of our formalities and understanding the UML style of metamodeling is encouraged to follow its bends and twists and read the intermediate sections too. A reward for this journey is the possibility to feel a special charm of UML modeling and metamodeling. After all, UML is a language and understanding UML is a linguistic activity beyond formal patterns.

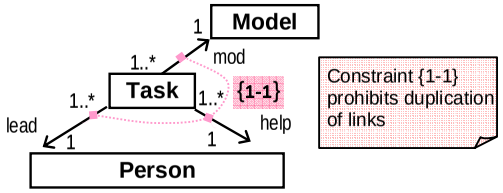
Relation to other work: What is *in* and *not in* the paper. Semantics for the concepts of association/relationship and particularly, of aggregation and role is a well-known research issue that can be traced back to the pioneering works on data semantics by Abrial, Brodie, Chen, Tsichritzis and Lochovsky in seventies-early eighties (see [11] for a survey). Since then a vast body of work on the subject was done and reported in the literature but this angle of viewing associations is far beyond the goals of the present paper. In UML2, modeling these aspects of associations is expressed by the corresponding value of the attribute "aggregation": whether it is aggregation (the white diamond), or composition (the black diamond) or neither of them (an ordinary association end). We leave building formal semantics for white-black diamonds for our next paper on the issue. A suitable mathematical framework is already developed in [4].

We also do not consider the *dynamic* aspects of the association construct [20],[] [8]. In UML2 they are attributed to collaboration diagrams and structured classifiers, and do not influence the Association part of the metamodel. The focus of the present paper is thus on Association as it is defined in the Standard, and covers all its technical aspects apart from the "diamonds". Clarification of technical issues is not a too aspiring research topic, and just a few works focused on them for UML 1.* were published, e.g., [9][7]. Unfortunately, they all became outdated when UML2 essentially reworked the technical aspects of Association. As for Association in UML2, the only published work we know about is [12], which is focused mainly on semantics of the attribute *isUnique* assigned to association ends. We incorporated Milicev's semantics of this attribute and developed it further (section 4.6).

Task		
lead(er)	help(er)	mod(el)
Bob	John	m1
Bob	Mary	m1
Bob	Ann	m2
Bob	John	m2
Bob	Mike	m2
Sue	John	m3
Sue	Mary	m3

|a0) table of records/links

(a| Association as a set of links (table or tabular map), *T = (Task, lead, help, mod)*



|a1) sets-and-mappings specification

An association declares that there can be links between instances of the associated types. A link is a tuple with one value for each end of the association, where each value is an instance of the type of the end.

The Standard [12, sect.7.3.3, p.42]

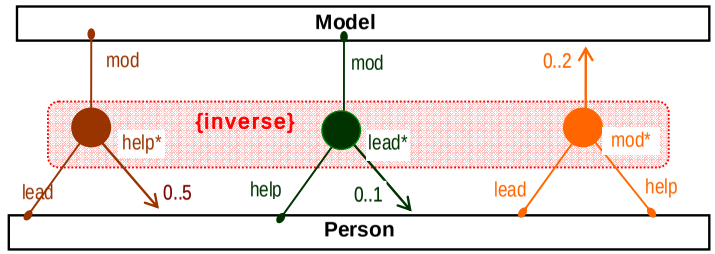
|a2) UML specification

Task-to-mod		
lead(er)	help(er)	mod*(el)
Bob	John	{m1,m2}
Bob	Mary	m1
Bob	Ann	m2
...	...	...

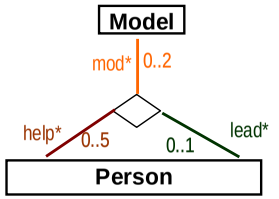
Task-to-help		
lead(er)	help*(er)	mod(el)
Bob	{John, Mary}	m1
Bob	{Ann, John, Mike}	m2
...	...	...

|b0) tables for look-up

(b| Association as a set of structural mappings (structural map), M_S = *(lead*, help*, mod*)*



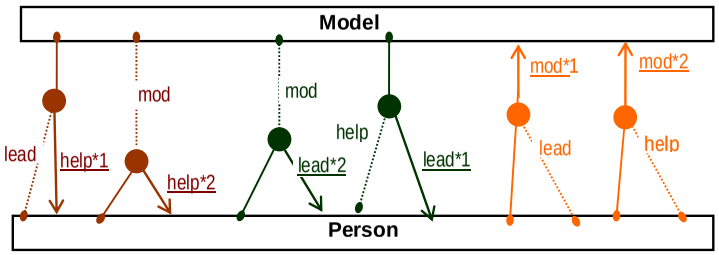
|b1) sets-and-mappings specification



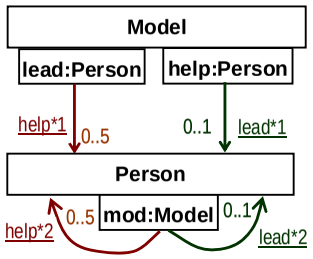
|b2) UML diagram

```
class Model
Person help*1 (Person lead)
 {...// look up the table from
    // mod(el)  towards  help(er)
 }
class Person
Person help*2 (Model mod)
{...// look up the table from
    //  lead(er) towards help(er)
}
```

(c| Association as a set of qualified mappings (operational map), M_Q=*(lead*$_1$,lead*$_2$, help*$_1$,help*$_2$, mod*$_1$, mod*$_2$)*



|c1) sets&mappings specification (constraint {inverse} is not shown),



|c2) UML diagram (not all qualified mappings are shown)

Table 1. A sample association and its three accompanying structures: tabular map (a), structural map (b), qualified map(c)

2 Symptoms of the Cylinder Syndrome for UML2 associations: an example

The following example shows the essence of the problems with UML *associations*.

2.1 Getting started

Suppose a set of *Task*s dealing with model development so that each task is assigned a model, which we will denote by $m1, m2$.... Also, each task is conducted by a *leader* and a few team-members whom we call *helpers*. We thus have a ternary association *Task* with three components:

(1) $\{lead{:}Person,\ help{:}Person,\ mod{:}Model\}$,

where *Person* and *Model* are classes participating in the association and *lead*,*help* and *mod* are their *roles* in the association.

The Standard begins its Semantics section devoted to Association [16, sect.7.3.3, p.42] with the following description:

> (Q1) An association declares that there can be links between instances of the associated types. A link is a tuple with one value for each end of the association, where each value is an instance of the type of the end.[a]

[a] In the UML jargon, classes are often called types.

A set of possible links for our association *Task* can be recorded in a three-column table as shown in cell (a0) in the leftmost column of Table 1. We will call it an *extension* of *Task*. Each row in table (a0) is a link and columns/roles are thus association ends.

Formally, an instantiated table can be specified by a set *Task* of its rows/links together with three projection mappings *lead*,*help*,*mod* into the corresponding classes as shown in cell (a1) of Table 1. The configuration of nodes and arrows in (a1) serves as a classifier for the links of the form shown in (a0) (note that Association is a Classifier according to the UML2 metamodel, see Fig. 2 below). In addition, if duplication of links in the table is not allowed, the triple of projection mappings must be declared *jointly one-one* or a *key*: for any two different links, at least one of the projections gives two different values. We will denote this predicate by symbol **{key}**. Importantly, this predicate/constraint *may* be declared for an association but is not a must: UML does admit extensions with duplicate links. Whether duplication is allowed or not is regulated by setting the attribute *isUnique* of association's ends to be True or False; we will return to this question later in section 4.6.

Thus, cell (a1) models association *Task* as a ternary table, that is, a triple of mappings

(2) $lead\colon Task \to Person, help\colon Task \to Person, mod\colon Task \to Model$,

with a common source type *Task*. We will write such table definitions by expressions $T = (R, p_1, p_2, p_3)$ with R the (name of the) collection of rows or links, which we call the *head* of the table, and p_i are (names of the) the *projection* mappings. Note that the tabular/extensional view of Association is entirely symmetric (non-directional) in that neither of the roles/ends has a preference.

2.2 Navigating associations, I: structural mappings

While the tabular view is usually sufficient for database applications, in the area of programming languages associations are normally understood in a navigational manner as *mappings*. The Standard says [16, sect.7.3.3, p.42]:

> (Q2) 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.

This description brings onto the stage the procedure of looking-up the extension table in one or another direction, see sample (pieces of) tables in cells (b0) in Table 1. Formally speaking, such a procedure amounts to a binary mapping (in the mathematical sense), and a three column table generates three such mappings as shown in cell (b1). Each mapping is denoted by a bold circle (of a different color with a color display), and its target is shown by an arrow. The two arguments are shown by edges/ends incoming into the circle. These argument ends must be named to distinguish between the different arguments of the same type (consider, for example, mapping mod^* whose two arguments have the same type *Person*). Since our mappings have only one output, we can use the mapping's name for it and write, for example, $mod^*(lead\colon Bob, help\colon John)$ to denote the collection of values assigned by the mapping mod^* to the pair of arguments, Bob in the role of *lead*er and John in the role of *help*er. Note that our association does not prohibit for the same *Person* to be a leader in one *Task* and helper in the other, and hence the same person can play different roles in different tasks (though our particular instantiation in cell (a0) does not show this possibility). Note that mappings specified in cell (a2) are different from the projection mappings in cell (a1), they have different sources and they have different multiplicities. Therefore, we name them differently using scripting the role names with stars: $help^*$, $lead^*$ and mod^*. Quote (Q2) also says that multiplicity of an end is the multiplicity of the corresponding *-mapping.

We will call such *-mappings *structural* as they seem to be particular cases of what is called StructuralFeature in UML2. Unfortunately, the Standard does not employ a well-known and reliable notion of mathematical mapping between sets.

Instead, UML uses a vaguely defined notion of Property [16, sect.7.3.44,p.125]

(Q3) A property is a structural feature. ... *Property represents a declared state of one or more instances in terms of a named relationship to a value or values.*[a] When a property is an attribute of a classifier, the value or values are related to the instance of the classifier by being held in slots of the instance. When a property is an association end, the value or values are related to the instance or instances at the other end(s) of the association (see semantics of Association [our quote (Q2)]).

[a] Italic is ours.

Fortunately, the notion of attribute is well understood: an attribute of a class is a mapping, which assigns a value or a collection of values to each instance of the class. It also appears that our *-mapping interpretation of the ends ($lead^*$, $help^*$, mod^*) well matches the second half of the description above. Then it looks reasonable to interpret the general definition cyphered by the italicized phrase in (Q3) by considering the notion of Property as UML's counterpart of the notion of mapping (unless new facts of our investigation will force us to revise the parallel, we will continue our quest below in section 3).

Thus, navigationally, our association is viewed as a triple

$$M_S = (lead^*, help^*, mod^*) \tag{3}$$

of binary mappings

$$\begin{aligned} lead^* &: (help{:}\,Person) \times (mod{:}\,Model) \twoheadrightarrow Person, \\ help^* &: (lead{:}\,Person) \times (mod{:}\,Model) \twoheadrightarrow Person, \\ mod^* &: (lead{:}\,Person) \times (help{:}\,Person) \twoheadrightarrow Model. \end{aligned}$$

whose sources are labeled Cartesian products[3]. Note the double-heads of the arrows: they mean that the mappings are (in general) multi-valued rather than single-valued. The latter case is depicted with single-arrow heads and the mappings are called *functional*, e.g., projections mappings in (2) are functional. Note also that each of the structural mappings is asymmetric and has a designated target/goal class. However, the set M_S of all three mappings (3) retains the symmetry of the tabular view. We will call such sets *structural maps* of associations.

2.3 Navigating associations, II: qualified mappings

When we think about implementation of structural maps, we need to decide (i) which of the possible navigation directions should be implemented efficiently and

[3] which are sets of labeled records and are often denoted by expressions like $\{\![\, help{:}\,Person, mod{:}\,Model \,]\!\}$

(ii), in the OO world where the UML constructs live, which of the classes will implement it. Irrespectively to (i), let us consider the purely structural aspects of realizing an n-ary structural mapping by unary (one-argument) mappings. For example, the binary mapping $help^*$ can be implemented as either a retrieval operation in class *Model* with a formal parameter *lead* of type *Person*,

$$\underline{help}^*_1(lead{:}\,Person){:}\,Model \twoheadrightarrow Person,$$

or as a retrieval operation in class *Person* with a formal parameter *mod* of type *Model*,

$$\underline{help}^*_2(mod{:}\,Model){:}\,Person \twoheadrightarrow Person,$$

see cell (c1) where both possible implementations $\underline{help}^*_1$ and $\underline{help}^*_2$ are shown.[4] We will call such mappings *qualified*, since UML calls formal parameters *qualifiers*. We will call qualified mappings also *qualified ends* of the association. Thus, the same association can be viewed as a six-tuple

$$M_Q = (\underline{help}^*_{1,2},\ \underline{lead}^*_{1,2},\ \underline{mod}^*_{1,2}) \tag{4}$$

of qualified mappings

$\underline{help}^*_1(lead{:}\,Person){:}\,Model \twoheadrightarrow Person$, $\underline{help}^*_2(mod{:}\,Model){:}\,Person \twoheadrightarrow Person$
$\underline{lead}^*_1(help{:}\,Person){:}\,Model \twoheadrightarrow Person$, ...
...

as shown in cell (c1). Note that each of the qualified mappings brings even more asymmetry/navigational details to its structural counterpart yet their full set, M_Q, retains the symmetry of the entire association; we will call such sets *qualified maps* of associations. A bit more accurately, the righthand side of (4) specifies an *atlas* $\boldsymbol{M_Q}$ consisting of three maps, each in turn consisting of two mappings. The map M_Q is the flattened version of $\boldsymbol{M_Q}$, $M_Q = \bigcup \boldsymbol{M_Q}$.

A care should be taken for using the UML's term "qualifier". Unfortunately, UML again mixes two different notions here. The Standard says [16,

[4] in the functional programming style, these mappings would be written as $\underline{help}^*_2 : Person \to [Model \twoheadrightarrow Person]$, and $\underline{help}^*_1 : Model \to [Person \twoheadrightarrow Person]$, where the square brackets denote the space of all possible mappings between the operands.

Sect.7.3.44,p.129]:

(Q4) Given a qualified object and a qualifier instance, the number of objects at the other end of the association is constrained by the declared multiplicity. *In the common case in which the multiplicity is 0..1, the qualifier value is unique with respect to the qualified object, and designates at most one associated object.* In the general case of multiplicity 0..*, the set of associated instances is partitioned into subsets, each selected by a given qualifier instance. *In the case of multiplicity 1 or 0..1, the qualifier has both semantic and implementation consequences.* In the case of multiplicity 0..*, it has no real semantic consequences but suggests an implementation that facilitates easy access of sets of associated instances linked by a given qualifier value. [a]

[a] We have used *italic* and sans serif fonts to separate the two cases.

What is called above the general case is a technical construct of replacing a multi-ary mapping (like our $help^*$) with a unary but parameterized mapping with the same extension (our $\underline{help}^*_{1,2}$). This construction is well known in type theory and functional programming by name of *Currying*. This is how we will understand qualification in the paper.

What is called above *the common case* of using qualifiers, actually refers to a quite different concern of how to build proper models involving associations. The issue is well known in database theory as normalization of relational schemas w.r.t. functional dependencies. We present some details and explanation in Appendix.

To summarize, in general an association is a triple $\boldsymbol{A} = (T, M_S, \boldsymbol{M_Q})$ of mutually derivable components: its extension table T (with projection mappings), its structural map M_S and its qualified atlas $\boldsymbol{M_Q}$. Each of these components in its turn consists of multiple member mappings like those specified in (2), (3), (4); to unify terminology, we could also call a table a *tabular map*.

Unfortunately, for specifying this rich instrumentary of extensional and navigational objects, the UML metamodel offers just one concept of the association *memberEnd*. For example, a ternary association consists of the total of 3+3+6=12 mappings while the UML metamodel states only the existence of its three end Properties. Not surprisingly, that in different parts of the Standard the same notion of *memberEnd* is interpreted as either a projection mapping, e.g., in description (Q1), or a structural mapping, e.g., in (Q2), or a qualified mapping (operation), as we will see below.

Note also that all the three components of an association: tabular, structural and operational, are structurally similar as is well seen from the middle column of Table 1; especially similar are the two navigational maps. In contrast, the UML notation for them (the rightmost column of Table 1) is strikingly dissim-

ilar. Moreover, there is no notation for the tabular component at all despite its basic role for the entire construct. Inevitably, all this leads to ambiguities and misconceptions, especially when the Standard tries to cast the intuition into the precise forms of the metamodel.[5]

3 Geography of Association: The structures accompanying the association construct.

In this section we begin our analysis of the UML metamodel. The parts of the latter specifying the metaclass Association and related constructs are summarized in Fig. 2, whose caption explains how the parts/packages were merged. From now on, the term UML metamodel will refer to that part of the UML metamodel, which is presented in Fig. 2.

3.1 What is a Property? A quest for a proper definition through the Standard

The notion of Property is central for UML modeling and is the cornerstone of the construct of Association as seen from the Fig. 2. Indeed, according to the metamodel, an association A is an n-tuple of Properties $(f_1, ..., f_n)$. $n \geq 2$, called A's *memberEnds*.

(Q5) A property related to an Association by memberEnd represents an end of the association. The type of the property is the type of the end of the association.[a]

[a] The Standard, [16, sect.7.3.44,p.125]

Thus, each of the *memberEnd* Properties has its *type* and we thus come to a tuple of types/classifiers

$$A.endType \stackrel{\text{def}}{=} (f_1.type, \ldots, f_n.type).$$

To shorten wording and formulas, we will often call *memberEnd*s just *ends*, and denote classifiers $f_i.type$ by X_i.

Unfortunately, despite the central role of Property construct, its specification in the metamodel is non-accurate and non-complete. Below we will carefully inspect the metamodel and try to make it into a consistent specification.

[5] It seems that even the much more formally precise OCL did not avoid confusion between the components when it borrowed UML's notation (abstract syntax) for association classes.

Labeled bags vs. lists. In the metamodel, both collections *A.memberEnd* and *A.endType* are considered as tuples/lists (note the attribute *ordered* near the metaassociations ends), which is not justified. Consider, for example, our association *Task* from the previous section: the collection of ends is a *set* $\{lead^*, help^*, mod^*\}$ and correspondingly the collection of classes is a *set* of pairs

$$A.endType = \{lead^* : Person, help^* : Person, mod^* : Model\}. \tag{8}$$

If we omit the labels, the set above would become a bag $\{\![\, Person, Person, Model \,]\!\}$. Such collections of pairs (label:element) are often called labeled records, thus, *A.endType* is a labeled record of classes (class names). Adjusting this terminology to the UML/OCL jargon, we will consider labeled records as bags with additional distinct labels for the elements, and call them *labeled bags*. Thus, we will say that the collection (8) is a labeled bag.

Ordering such collections would be irrelevant and, most likely, it has mistakenly appeared in the metamodel because of the following. In formal considerations, it is convenient to use natural numbers as labels/role names and, say, instead of names *lead*,*help*,*mod* write f_1, f_2, f_3. (Note that instead of $1, 2, 3$, we could well use labels a, b, c or, say, $x, \rho, \Re$ as well). Unfortunately, using natural numbers as labels makes the set of labeled elements ordered thus bringing an accidental construct onto the stage. A proper formulation is to say that *A.endType* is a labeled bag and write

$$A.endType \stackrel{\text{def}}{=} \{f_1 : X_1, .., f_n : X_n\} \tag{9}$$

where, we remind, all f_i are distinct and $\{f_1..f_n\} = A.memberEnd$, X_i denotes $f_i.type$. Often, within the "bag philosophy", the collection above is written shortly as $\{\![\, X_1..X_n \,]\!\}$ assuming that the labels are known from the context.

So far we have said nothing about the source types of the end mappings, which are their crucial parameters. A striking observation is the metamodel indeed says nothing about this component of the construct, and we need to look for a help in the explanatory sections of the Standard.

Looking for Property's source. The Semantics section 7.3.44 in [16, p.128] says:

(Q6) [(i)] When a Property is owned by a classifier other than an association via ownedAttribute, then it represents an attribute of the class or data type. [(ii)] When related to an association via memberEnd, it represents an end of the association. 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) type. 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 types at the other end or ends of the association.[a]

[a] in this piece, the terms "type" and "classifier" are used interchangeably and, hopefully, can be considered synonyms here.

As we do not have a formal model of ownership so far, let us consider the case (ii) of the description (later we will see that (i) is also nothing but its particular case). The description sounds a bit wordy but is actually informative. It says that we can consider a Property as a mapping from some source set, whose elements are labeled tuples ("instances of a collection of types"), to a target set called the *type* of the Property and whose elements play the role of values that the Property takes. The collection of labeled types forming the source is called the *context* of the Property. For instance, the binary mappings $help^*$ from our main example is a Property with context $\{\!\![\, mod : Model, lead : Person \,]\!\!\}$ and type *Person*. Thus, according to the description (Q6), there should be a meta-association *context* from metaclass Property to metaclass Class but, surprisingly, the metamodel does not specify it. This problem can be partially fixed in the following way.

First of all, we collect those Properties that are ends of Associations in a special subclass EndProperty (see the shaded part of Fig. 3). A principal distinction of this subclass is that the meta-end *asson* has the precise multiplicity 1 rather than 0..1 for general Properties. This multiplicity allows us to *derive* the missing metaassociation *context* from other elements specified by the metamodel as follows. We first define a derived metaassociation *coEnd* as specified by expression (10) in Fig. 3 (also correcting the loop metaassociation *opposite* of the metamodel). Then we define a derived metaassociation *context* as specified by expression (11). In our main *Task* example, this procedure would produce for, say, a property $help^*$, the context ($lead^*$:*Person*, mod^*:*Model*).

Invertibility of association ends: A key constraint missing from the metamodel. The shaded part of Fig. 3 presents a plausible view of the structural aspects of the Association construct but it is still essentially incomplete

and lacks a crucial condition. Namely, we need to require that all ends of the association do have the same extension. We will formulate this condition by saying that the ends are *mutually inverse*, meaning that they are mutually derivable from each other by inverting/permuting sources and targets (this condition is well known for the binary case).

Formally, this can be captured in the following way. To simplify presentation, we will take natural numbers to be the labels and then, say, a general ternary association appears as a set of three binary mappings, or *end Properties*, or just *ends*,

$$(18) \qquad f_1 \colon X_2 \times X_3 \twoheadrightarrow X_1,\ f_2 \colon X_1 \times X_3 \twoheadrightarrow X_2, \text{ and } f_3 \colon X_1 \times X_2 \twoheadrightarrow X_3.$$

where the Cartesian products are the contexts. Further, in section 5, we will formalize the general case of arbitrary names as labels.

The double-heads of the arrows mean that the actual target of the mapping is a collection $\mathsf{coll}_{f_i}(X_i)$ of some type (a set, a bag or a list) specified along with f_i and built from elements of X_i. In more detail, if the pair (a_1, a_2) is an argument of mapping f_i, then $f_i(a_1, a_2)$ is a collection built from elements of X_i rather than a single element. Whether this collection is a set or a bag is regulated by an attribute/flag *isUnique* assigned to f_i. The collection is a set if f_i.*isUnique* is set to True and is a bag otherwise. We will also shortly phrase these two cases by saying "f_i is *Unique* or *nonUnique*". It is also convenient to distinguish between the two cases by using a special arrow for set-valued mappings, we will superscript the double-arrow head with ! for that purpose.

3.1 **Definition: mappings, contexts, extensions.** Let $\mathbf{X} = \{\!\![\, X_1 ... X_n \,]\!\!\}$ be a family of classes, that is a labeled bag with natural numbers being the labels. Let $f \colon X_1 \times .. \times X_n \twoheadrightarrow Y$ be a procedure (or a rule or prescription), which for a given tuple/labeled bag of arguments $(x_1, ..., x_n)$, $x_i \in X_i$, returns (specifies) either a collection of elements of class Y or a special value "undefined". Then we say that f is a *(structural) mapping* of *arity* n, of *type* Y and of *(source) context* $\mathbf{X}$. Following UML, we will also call (structural) mappings *(structural) properties*.

A special case, when the value is a singleton (that is, in fact, an element in X_i) will be denoted by a single-arrow head, and such mappings will be called *functional* or *functions*.

The *extension* of mapping f, $\mathsf{ext}(f)$, is the collection of tuples

$$((\mathsf{ext})) \qquad [(x_1, \ldots, x_n, y) : x_1 \in X_1, ..., x_n \in X_n, y \in f(x_1 ... x_n) \in \mathsf{coll}_f(Y)]\,,$$

which is a bag or set iff f is bag-valued or set-valued respectively.[6]

A natural way of presenting collection (ext) is to store it in a table. In fact, we have a mapping $\mathsf{ext} \colon \mathsf{Mapping} \to \mathsf{Table}$ sending any n-ary mapping/property to a $(n{+}1)$-column table recording its extension. To avoid terminological clashes,

[6] If f is list-valued, we can either disregard the ordering information by considering the underlying bag, or consider the extensional set to be partially-ordered.

we will call such mappings between collections of mappings *functors*. Thus, ext is a functor.

Conversely, having a table, we can select one of its columns as a target/goal and look up the table in the direction from the other columns to the target column. This procedure will give us a multi-valued functor $\mathsf{lookUp}\colon \mathsf{Table} \twoheadrightarrow \mathsf{Mapping}$ since we can look-up an n-column table in n diffrent directions. Evidently, for any table $T \in \mathsf{Table}$, the $T.\mathsf{lookUp.ext}$ is the singleton $\{T\}$.

3.2 **Definition: Mutual invertibility of mappings and Structural maps.** Let again $\mathbf{X} = \{\![\, X_1 ... X_n \,]\!\}$ be a family of classes. We say that an (n-1)-ary mapping f is a mapping *over* $\mathbf{X}$ if the source context of f and its type are complementary in $\mathbf{X}$, that is,

$$type(f) = \mathbf{X} \setminus context(f).$$

The family $\mathbf{X}$ is then called the *full* context of f.

Two or more structural mappings $f_1 ... f_k$ over $\mathbf{X}$ are called *mutually inverse* if they have the same extension:

((inverse)) $$\mathsf{ext}(f_1) = \mathsf{ext}(f_2) = ... = \mathsf{ext}(f_k).$$

An n-element set $M_S = \{f_1 ... f_n\}$ of mutually inverse structural mappings over $\mathbf{X}$ is called a *structural map* over $\mathbf{X}$. In other words, a structural map is a maximal set of mutually-inverse structural mappings. The family $\mathbf{X}$ is called the *(full) context* of M_S.

Now we can (and must) add the constraint ((inverse)) to our metamodel of Association, see constraint (13) in Fig. 3. No doubts that this constraint is implicitly assumed by the Standard. Thus, at the current state of our investigation, we may say that the Standard defines associations as nothing but structural maps. Following UML's terminology, we also call the members of structural maps *(structural) ends*.

Our metamodel still misses one more essential component of the notion, namely, realization of structural mappings/properties by qualified mapping/properties (recall our main example in section 2).

3.2 Qualifiers and qualified properties.

Let $g\colon X_1..X_n \twoheadrightarrow Y$ be an n-ary mapping. It can be presented as an unary (one-argument) mapping $\underline{g_i}$ with $(n-1)$-parameters $x_1..x_{n-1}$ in n different ways, $\underline{g_i}(x_1, .., x_{n-1})\colon X_i \twoheadrightarrow Y$, $i = 1..n$. (This process of moving from an n-ary mappings to mappings of lesser arity with parameters is well known and usually called *Currying* (see, e.g., [10]), we will write $\underline{g_i} = \mathsf{Curry}_i(g)$. The corresponding functor $\mathsf{structMapping} \twoheadrightarrow \mathsf{qualifiedMapping}$, sending an n-ary mapping g to the set of its unary parameterized realizations, $\{\mathsf{Curry}_i(g) \mid i = 1..n\}$, will be denoted by Curry.

To form a Curried version of g, we need to pick up a *source* class X_i and consider the rest of arguments as parameters. Importantly, the source X_i is a

pair ($label : className$) rather than just a className as is well illustrated by Currying of mapping mod^* from our main example. It has two Curried versions, $\underline{mod^*}_1(lead{:}Person){:}\ Person \twoheadrightarrow Model$ and $\underline{mod^*}_2(help{:}Person){:}\ Person \twoheadrightarrow Model$, with the same source class but with different roles this class is playing: as the class of *help*(ers) for $\underline{mod^*}_1$ and as the class of *lead*(ers) for $\underline{mod^*}_2$. Correspondingly, they have different qualifiers too. In the general case, if $X_1..X_n$ is the context and X_i is the source of $\underline{g_i}$, the qualifier is nothing but the difference (context \ holder). The inverse operation of converting a qualified mapping into a multi-ary mapping without parameters is also possible, we call it *unCurrying* and write $g = \mathsf{unCurry}(\underline{g_i})$; the corresponding functor is denoted unCurry. Thus, for a qualified end g, equality $f = \mathsf{unCurry}(g)$ mean nothing but $g \in \mathsf{Curry}(f)$, and vice versa. Note that unCurry is a single-valued operation while Curry is set-vlaued.

3.3 **Definition: Curried friends and Curried maps.** We will say that two or more qualified mappings $\underline{g_1}..\underline{g_k}$ are *Curried friends* or *mutually coCurry* if they are qualified versions of the same multi-ary mapping:

$$((\mathsf{coCurry})) \qquad \mathsf{unCurry}(\underline{g_1}) = \mathsf{unCurry}(\underline{g_2}) = ... = \mathsf{unCurry}(\underline{g_k}).$$

A full n-element set M_C of mutually coCurry $(n-1)$-parameterized mappings is called a *Curry map*. Each Curry map generates the only $(n-1)$-ary mapping (because all its members are mutually coCurry)

All the components are mutually derivable/inverse as shown by the following functorial diagram (nodes are sets of mappings and arrows are functors):

$$\mathsf{Table} \underset{\mathsf{ext}}{\overset{\mathsf{lookUp}}{\underset{\mathbf{\{inv\}}}{\rightleftarrows}}} \mathsf{structMapping} \underset{\mathsf{unCurry}}{\overset{\mathsf{Curry}}{\underset{\mathbf{\{inv\}}}{\rightleftarrows}}} \mathsf{qualifiedMapping} \tag{19}$$

.

3.4 **Construction: Adding navigation to tables.** We can enrich tables with "navigational" lookUp information if the corresponding column name will be marked (say, by a star). Similarly, if a table stores the extension of a qualified mapping, we can keep this information by marking the two corresponding columns. In this way we come to the notions of (i) *star-table*, a table with one column specially designated and called the *goal*, and (ii) *double-star table*, a star-table with one more column designated/marked as the *source*.

Now we can formulate the main definition of this section.

3.3 Associations structurally: Main definition and discussion

3.5 **Definition: Full structural view of association.** An n-ary association, *structurally*, is a triple $\boldsymbol{A} = (T, M_S, \boldsymbol{M_Q})$ with the following components:

– $T = (R, p_1..p_n)$ is a table called the extension of the association, R denotes the set of rows/links and p_i are projection mappings or *projection ends* (*pEnds* for short). The span configuration (R is the head and p_i are legs) classifies the links of the association.

– $M_S = \{f_1..f_n\}$ is an n-element set of mutually-inverse $(n-1)$-ary mappings (Properties). Mappings f_i are called the *structural ends (sEnds)* of the association and the set M_S is its *structural map*. Following UML, we will call structural ends just *ends*.

– $\boldsymbol{M_Q} = \{M_{C1}..M_{Cn}\}$ is an n-element atlas of Curried maps $M_{Ci} = \{\underline{g_{i1}}..\underline{g_{i(n-1)}}\}$, $i = 1..n$ [compare with (4)], each one consisting of mutually coCurry qualified mappings/Properties or *qualified ends (qEnds)*.

Moreover, it is supposed that the three components are mutually derivable or *mutually inverse*, that is,

((inverse)*) $\quad \mathsf{unCurry}(M_{Ci}) = f_i$ and $\mathsf{ext}(f_i) = T$ for all $i = 1..n$.

Further we will also work with the flattened set $M_Q = M_{C1} \cup ... \cup M_{Cn}$ of all qualified ends.

Thus, thinking semantically, an association comprises three sets of mappings: a table, a structural map and a qualified map. Because of condition (inverse)*, within each of the maps its member mappings are also mutually derivable and the entire three complex components are mutually derivable by collecting the values of functors in diagram (19).

The metamodel of this definition is presented in Fig. 4. Note that metaclasses are parameterized by the arities of the constructs involved, which makes many structural aspects explicit and allows us to avoid writing down size-related constraints. With the UML style of metamodeling (Fig. 2), these constraints must be added to the metamodel and, in fact, many of them are missing from the Standard. Another advantage of our metamodel Fig. 4 is its clear exposition of structural symmetries of the notion: the central part of the metamodel is built, in fact, from repeated patterns-blocks. Indeed, there are four real operations between the components: to look-up a table and to record extension of a mapping/end, to Curry a mapping and to unCurry a qualified mapping, they all are in the top part of the diagram. All other metaassociations (Curry-unCurry) and (lookUp-ext) are derived from these using grouping of elements by (member-host) metaassociations. Some of the latter could be also derived through the common association source, say, for a structural map M, we have $M.member \stackrel{\text{def}}{=} M.asson.T.lookUp$ but we prefer to keep symmetry and consider all such metaassociations basic. Of course, we then need to add many commutativity constraints to the metamodel. We may consider that all these constraints are embodied into the global constraint **mutually inverse**, whose scope is then the entire metamodel. Note also that the basic notion of association's full context (Definition (3.2)) is made explicit.

UML prefers to work with associations asymmetrically: the metaassociation *sEnd* is considered basic while all the rest is assumed to be derived. To ease the comparison of our formal metamodel with the UML's one, we have arranged the former in a way similar to UML: the result is presented in Fig. 3, whose shaded part was already used above.

3.4 UML metamodel of associations in the light of formalization (A)

The metamodel shown in Fig. 3 accurately describes semantics of the association construct as we have discovered it so far, and presents it in a way similar to the UML metamodel. It is possible and instructive to compare it with the UML metamodel in Fig. 2 (disregarding there, for a while, the navigational and ownership aspects). An immediate comparison reveals a few essential omissions in the UML metamodel. We briefly list them below.

1. The extensional/tabular part of the construct (the right-upper unshaded fragment of our metamodel) is missing from the UML metamodel, which does not allow us to specify Association as a classifier. Another consequence is that metaassociation ext is implicit and hence a major constraint (inverse) in Definition 3.2 cannot be formally specified.
2. The metaassociation *context* of metaclass Property is missing too. If even we introduce the subclass EndProperty for which *context* can be derived, its explicit presence in the metamodel makes the latter much more transparent and less error-prone.
3. The (loop) metaassociation *opposite* is formulated only for the binary case while it makes sense for the general N-ary case as well (our association *coEnd*).
4. The metaassociation qualifier is mistakenly targeted to metaclass Property while its proper target is metaclass Class. Moreover, the entire "qualified" part of the construct (the lower fragment of our metamodel) is missing.
5. A few less crucial omissions are described in the footnotes to Fig. 3.

4 The *operational* view of associations

In this section we consider that part of the UML association metamodel, which specifies navigability and other operational concerns and relationships between Classes, Properties and Associations; we will also consider the two possible meanings of *isUnique* attribute.

4.1 Navigability: holdership vs. ownership

In the previous version of the Standard and the UML folklore, some correlation between navigability and ownership was implicitly assumed. The situation changed in the current version [1, sect.7.3.3,p.45]:

(Q7) Navigability notation was often used in the past according to an informal convention, whereby non-navigable ends were assumed to be owned by the association whereas navigable ends were assumed to be owned by the classifier at the opposite end. This convention is now deprecated. ...navigability and end ownership are orthogonal concepts, each with their own explicit notation.

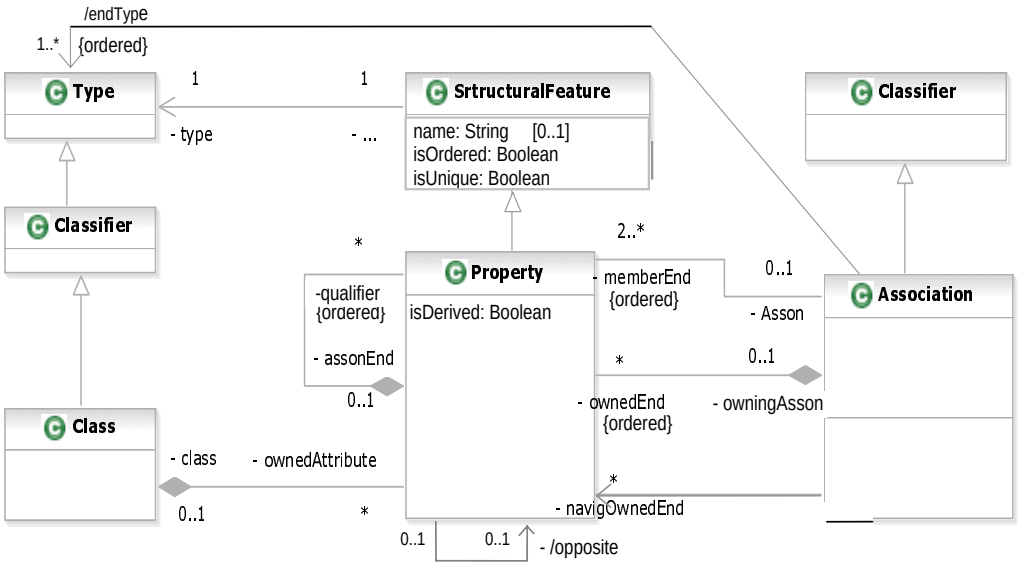
Note that according to the new version, the metamodel should have two *independent* meta-associations from meta-class Association to Property, *ownedEnd* and *navigableEnd*. However, the current version of metamodel is not adjusted to the new semantics and keeps the meta-association *navigableOwnedEnd* subsetting *ownedEnd*. We could guess that there is some cause for this inconsistency. And indeed, separation of ownership and navigability is a right idea but it is done in the Standard in a messy way. We are going to show that the Standard mistakenly mixes two different concepts: (i) ownership of a model element (say, a mapping/end by an association), and (ii) an implementation of a model element, say, a multiary mapping by one of its context classes; in the latter case we call the class a *holder* of the element. The concept (i) is structural and belongs to the general modeling arsenal whatever sort of models we consider. The concept (ii) is more particular and aimed at specifying a particular concern of how to implement an association end. Actually the notion of a *holder* class is crucial for associations as soon as we think of them in a more operational/implementation-oriented way yet it is missing from the Standard. The result is that the role of being the holder of a navigable mapping/end is mistakenly confused with the ownership of the end and then ownership becomes indeed related to navigability. It is this second sense of ownership that is often considered in the folklore and was assumed in the older versions of UML2. We again have a Cylinder Syndrome case.

4.2 Navigable vs. non-navigable = Basic vs. derived.

We begin our analysis with the concept of navigability. The Standard says [1, Sect. 7.3.3,p.42]:

> (Q8) Navigability [of an end] means [that] instances participating in links at runtime (instances of an association) can be accessed efficiently from instances participating in links at the other ends of the association. The precise mechanism by which such access is achieved is implementation specific. If an end is not navigable, access from the other ends may or may not be possible, and if it is, it might not be efficient.

This definition says that for a given association, the set of its association ends is partitioned into navigable and non-navigable ends. The former are navigated efficiently while the latter are non-efficient or not navigable at all. Suppose, for instance, that the end *help** in our main example is declared to be navigable while the ends *lead** and *mod** are not. A natural way to implement this would be to index the extension table (a0) in a way providing an efficient navigability from the columns *lead* and *mod* to the column *help*, see table *Task*-to-*mod* in cell (b0) of Table 1. The next question is where to store this table, and which of the classes, *Model* or *Person*, or both, would use this indexed (prepared for navigation) table and host/store the method *help**.



Constraints for Association context in OCL
(to shorten expressions we write end for memberEnd):

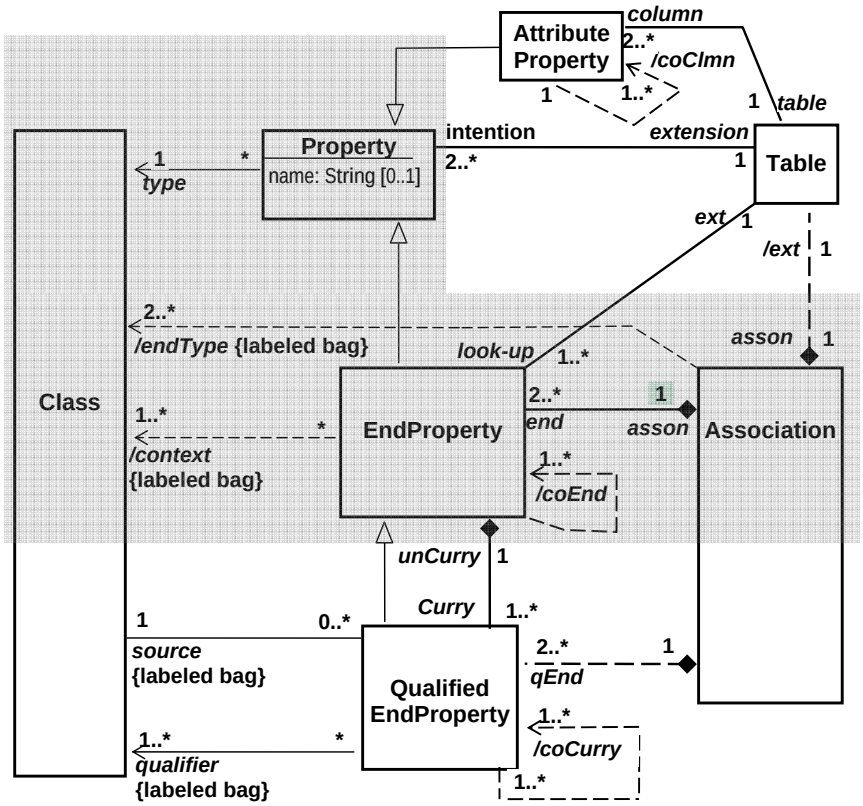
(5) self.end->includesAll(self.ownedEnd) ->includesAll(navigOwnedEnd)

(6) def: self.endType = self.end->collect(type)

(7) self.end->size() > 2 implies self.ownedEnd = self.end[a]

[a] this is the Constraint (5) in [1, sect.7.3.3, p.42],

Fig. 2. A piece of UML metamodel extracted from [1, Fig.7.12] with additions from Figures 7.5, 7.10 and 7.17. In more detail, according to the piece of the metamodel in Fig.7.10, metaclass StructuralFeature is a subclass of both TypedElement and MultiplicityElement metaclasses. From the former, it inherits meta-association *type* (Fig. 7.5), and from the latter, it inherits Boolean attributes *isOrdered* and *isUnique* (again Fig. 7.5). Meta-association *qualifier* is provided by Fig. 7.17. The constraints are written in OCL [15]



Definitions for EndProperty

(10) def: self.coEnd = self.asson.end − {self}

(11) def: self.context = self.coEnd->collect(pair(name,type))[a]

(12) def: self.ext = self.extension (just rename)

Constraints for Association

(13) self.end->forAll(f1,f2| f1.ext = f2.ext)[b]

Definitions for Association

(14) def: self.endType = self.end->collect(pair(name,type))[c]

(15) def: self.ext = self.end->any().ext[d]

Constraints for QualifiedProperty

(16) union(self.qualfier, {self.source}) = self.context

Definitions for QualifiedProperty[e]

(17) def: self.coCurry = self.unCurry.Curry−{self}

[a] this definition is missing from the metamodel but described in Semantics section of the Standard

[b] This is condition ((inverse)) in Definition 3.2. It is missing from the UML metamodel

[c] we correct definition (6) of the metamodel, see Fig. 2

[d] definition is correct owing to constraint (13).

[e] we add it to show similarity with coEnd

Fig. 3. Metamodel of the three structural aspects of Association (ends, qualified ends and extension) aligned (as far as possible) with UML metamodel (see Fig. 2). Derived meta-associations are dashed

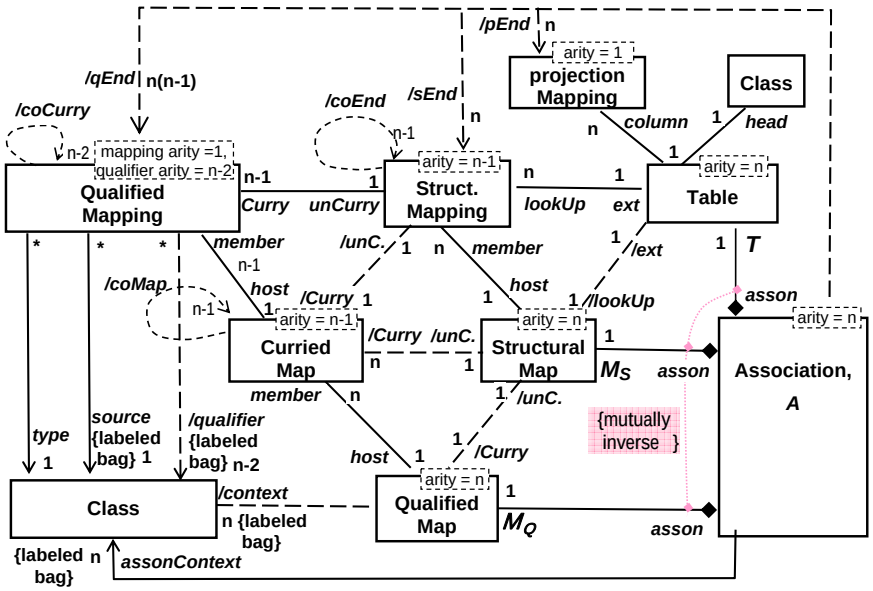


Fig. 4. Metamodel of Definition 3.5: Semantics of Association as a triple $A = (T, M_S, \boldsymbol{M_Q})$. Derived meta-associations are dashed. Label unC. is a shorthand for unCurry.

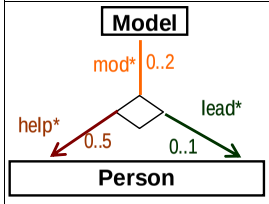
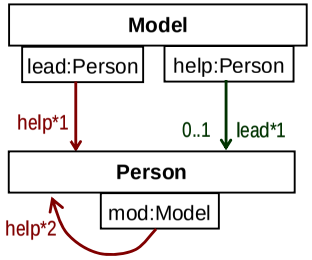
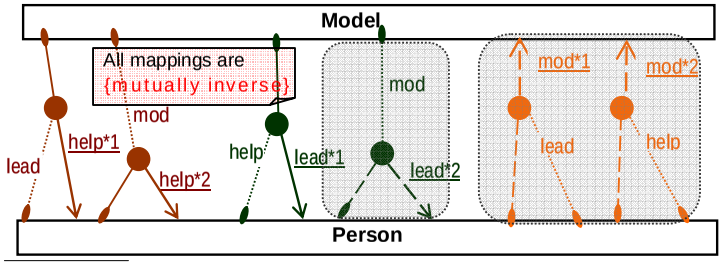
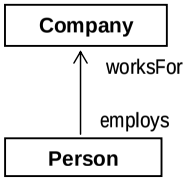
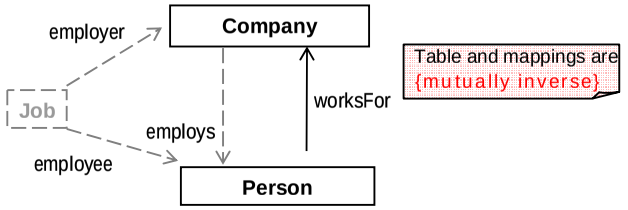
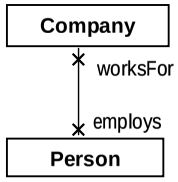
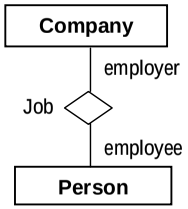
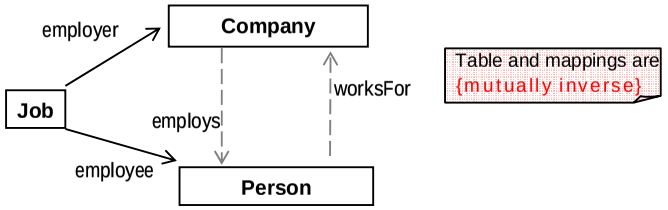
Underspecified UML model	Well-specified UML model	Precise set-and-mappings specification. Derived elements are either shaded or dashed or/and gray
		 *) One more derived item, the extension table, is not shown to save space.
N/A		
		
The Standard states that both UML notations (line and diamond) are synonyms. Our proposal is to separate their meaning interpreting diamonds as shown in the right cell.		

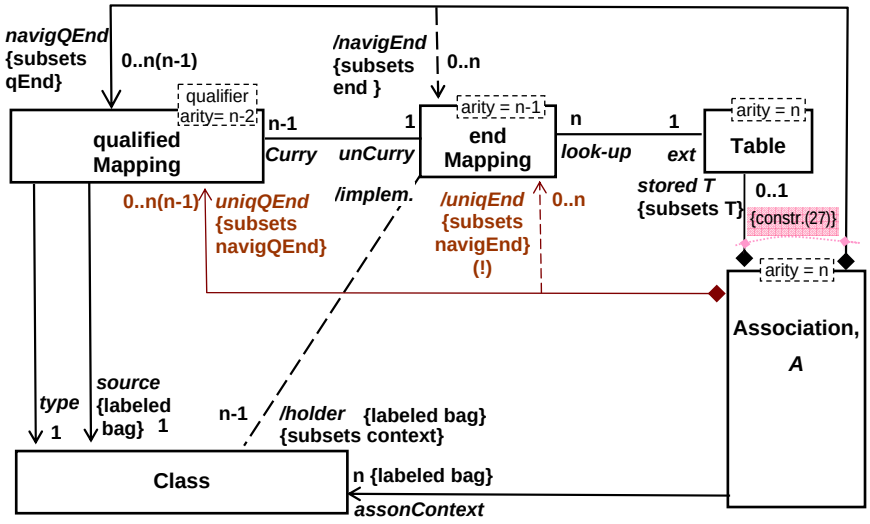
Table 2. Specifying navigability in associations

This consideration appears to be close to a well-known distinction between basic and derived data in databases. The former are stored in the database and hence are directly accessible while getting the latter requires asking queries. Some types of queries could be executed efficiently while others are not, yet data to be queried are not specified in the database schema and conceptually are quite distinct from basic data. On the other hand, basic data immediately stored in the database are also accessible by queries, but the latter are fairly trivial (and automatically efficient). Thus, the distinction between navigable and non-navigable ends in UML is exactly similar to the distinction between basic and derived data in SQL.

Examples presented in Table 2 illustrate the idea. In the left cell of the top row, we have a ternary association with two navigable ends. However, the model does not specify which of the classes should be responsible for implementation, that is, in our terminology, be the holder. The issue is fixed in the model on the right with qualified associations, and a precise sets-and-mappings formal specification is shown in the rightmost cell; note the constraint **{mutually inverse}** there. The example in the second row is clear. Particularly, the constraint **{mutually inverse}** says that the items *worksFor*, *employs* and (*Job*,*employee*,*employer*) are mutually derivable. Hence, since the mapping *worksFor* is assumed to be basic, the rest of the configuration can be derived from *worksFor* if needed.

Consider the third example. In the left model, both ends are declared non-navigable and hence the basic component of the association is its extension table (from which the ends/mappings can be derived). Here we assume that if an association appears in the model, then it should have at least one way of implementation prescribed. However, the model provides names for neither the table nor for its columns: a precise specification in the rightmost column demonstrates the problem. A reasonable question is whether it is possible to build a better UML model for the case. Our solution is proposed in the middle column, where, contrary to its use in UML, the diamond is intended to show that the data table (the set of links) is a basic component while the corresponding mappings are derived. We will return to this idea below in section 4.5.

The metamodel for the notion of navigability as discussed above is fairly simple and is presented in Fig. 5 (disregard the fragment related to Uniqueness for a while, it is brown with a color display). All that we need to do is to select a number of qualified ends as basic or navigable, and hence to be implemented as retrieval operations in their source classes. If a qualified end g is navigable, its unCurried version gives us a navigable association end $f = \mathsf{unCurry}(g)$, and we call the source class of g a *holder* of f. Of course, there may be other holders as well: $f.holder \stackrel{\text{def}}{=} \{g.source | \ g \in f.\mathsf{Curry}\}$; we remind that $g.source$ is a labeled class name and hence $f.holder$ is a labeled bag. Thus, a class $g.source$ will implement the navigable association end $g.\mathsf{unCurry}$ but it does not mean that this class owns the end. The same end may have a bag of holders but only one owner – its association. In fact, ownership of elements occurring into the metamodel of Association is not anyhow related to navigability nor to implementation. In the next section we will consider the issue in more detail.



Definitions for endMapping

(20) self.holder = self.Curry->collect(source)[a]

[a] the Standard mistakenly attributes this metaassociation to ownership

Fig. 5. Metamodel of the operational view of Association (see Definition (4.1) below). Derived meta-associations are dashed. See Fig. 4 for the supersetting metaassociations.

4.3 Navigability and ownership again

An association A owns all its ends because as soon as A is deleted from the model, all its ends must be also deleted even though they are implemented as retrieval operation in some classes. Conversely, if an association A with an end f is added to the model, it does not mean that the corresponding operation will appear in one of the classes. The latter is a question of (i) declaring the end navigable and (ii) selecting this class to be a holder for the end. For arity $n \geq 3$, it precisely corresponds to what is said in the Standard, [1, Constraint (5) in Sect.7.3.3,p.42]. However, the Standard entirely changes the treatment for the case $n = 2$. We can guess that the reasons for this are as follows.

When the arity of association is $n = 2$, we have a degenerate situation, for which two structural mappings/ends/Properties coincide with the corresponding qualified mappings. Then, if a (structural) end is declared navigable, its *automatically* becomes an attribute of the corresponding *uniquely determined* holder class. UML mistakenly qualifies this situation as ownership, invoking for implementation/operational concerns a constructs from the entirely different structural view. Thus, a proper treatment is to consider all ends of an association to be owned by the association irrespectively of its arity. Navigable ends come from an entirely different concern of implementation, and are correspondingly related to one or more holder classes, again irrespectively of the arity. The only special feature of arity 2 is that a navigable end has one and only one holder.

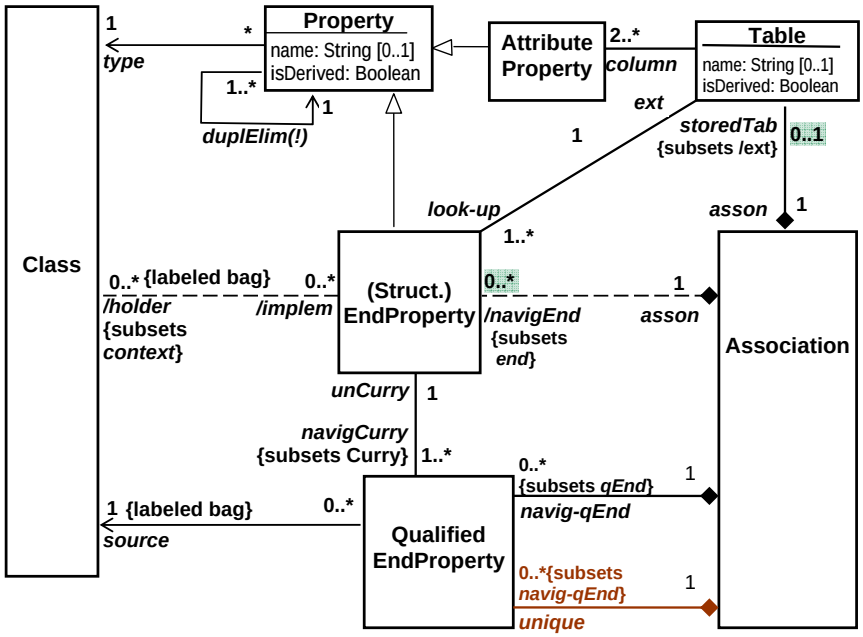
4.4 UML metamodel in the light of formalization (B)

The metamodel in Fig. 6 presents the same semantics as metamodel in Fig. 5 but in a way aligned with the UML-style of metamodeling, and can be immediately compared with the UML metamodel.

First of all, we note that in our metamodel, metaassociation *navig-qEnd* is basic while *navigableEnd* is derived contrary to the UML metamodel. The point is that while the latter can be indeed derived from the former by applying unCurry operation, derivation in the reverse direction does not work: operation Curry is multivalued and declaring an end navigable leaves unspecified the holder class or classes to implement it (see example in the top row of Table 2).

Metaassociation class–ownedAttribute in the UML metamodel (actually means and) must be replaced by metaassociation holder–implem. On the other hand, the Asson end of memberEnd metaassociation must be black-diamond while metaassociation ownedEnd–owingAsson must be removed.

Finally, if none of the ends is navigable, then the only way of representing the association in the software system is to store the corresponding data table in some class (preferably not occurring into the bag of the context classes to avoid name clashes between projections of the table and the structural and qualified mapping). This requirement is captured by an important constraint (26), which says that an association must be somehow implemented: either navigationally as a mapping (retrieval *get* method) or extensionally as a data table.



Definitions for Association:

(21) $\text{self.navigEnd} = \text{self.navigQEnd.unCurry}$

Definitions for EndProperty

(22) $\text{self.navigCurry} = \text{intersect (self.Curry,self.asson.navigQEnd)}$

(23) $\text{self.navigCurry->collect(source)}$

Constraints for Association:

(24) $\text{self.navigEnd.isDerived} = \text{False}$

(25) $\text{self.storedTab.isDerived} = \text{False}$

(26) $\text{self.navigEnd->size()} + \text{self.storedTab->size()} \geq 1$

Fig. 6. Metamodel for the *operational* view of associations aligned (as far as possible) with UML metamodel (see Fig. 2). Derived meta-associations are dashed. See Fig. 3 for the supersetting metaassociations.

4.5 Notation for the operational aspects: diamonds vs. lines

We return to the problem of the UML notation for associations exposed by our example in the third row in Table 2. The Standard states that the diamond and line-segment notations for binary associations are synonyms [1, Sect.7.3.3, Notation, p.43]:

(Q9) Any association may be drawn as a diamond [...] with a solid line for each association end connecting the diamond to the classifier that is the ends type. An association with more than two ends can only be drawn this way.
A binary association is normally drawn as a solid line connecting two classifiers, or a solid line connecting a single classifier to itself (the two ends are distinct).

Contrary to this convention, we suggest to reserve the diamond notation for the case when the extension table is considered to be a basic item in the model. This notation would make a perfect match with the use of diamond in famous ER-diagrams still extremely popular in the database world. However, if we use diamonds for tables, we need to invent another, non-diamond, notation for the case of navigable, i.e., basic, association ends. Since for the binary associations this case is denoted by a line segment, it is reasonable to extend the line-based notation for multi-ary associations too. A possible realization of this idea is presented in Table 3 in hopefully self-explained way.

Note that the meaning of association ends in the left and the right columns are different. In the former they denote structural mappings while in the latter they mean projection mappings. Note also that the models in the left columns are underspecified: they show navigable ends but say nothing about the classes which should implement them. A complete in this sense notation is shown in the middle column. In the case of binary associations both notations, the left and the middle, coincide. Finally, pay attention to the blank diamond notation that may be useful during early phases of design. Semantics for it is given by our structural notion of association, see Definition (3.5).

4.6 Uniqueness: a constraint or design decision?

Boolean attribute *isUnique* of an association end is intended to regulate the possibility of having duplicates in the collection retrieved by the end. The corresponding Semantics section says [1, Sect 7.3.3, p.42]

(Q10) When one or more ends of the association have isUnique=false, it is possible to have several links associating the same set of instances. In such a case, links carry an additional identifier apart from their end values.

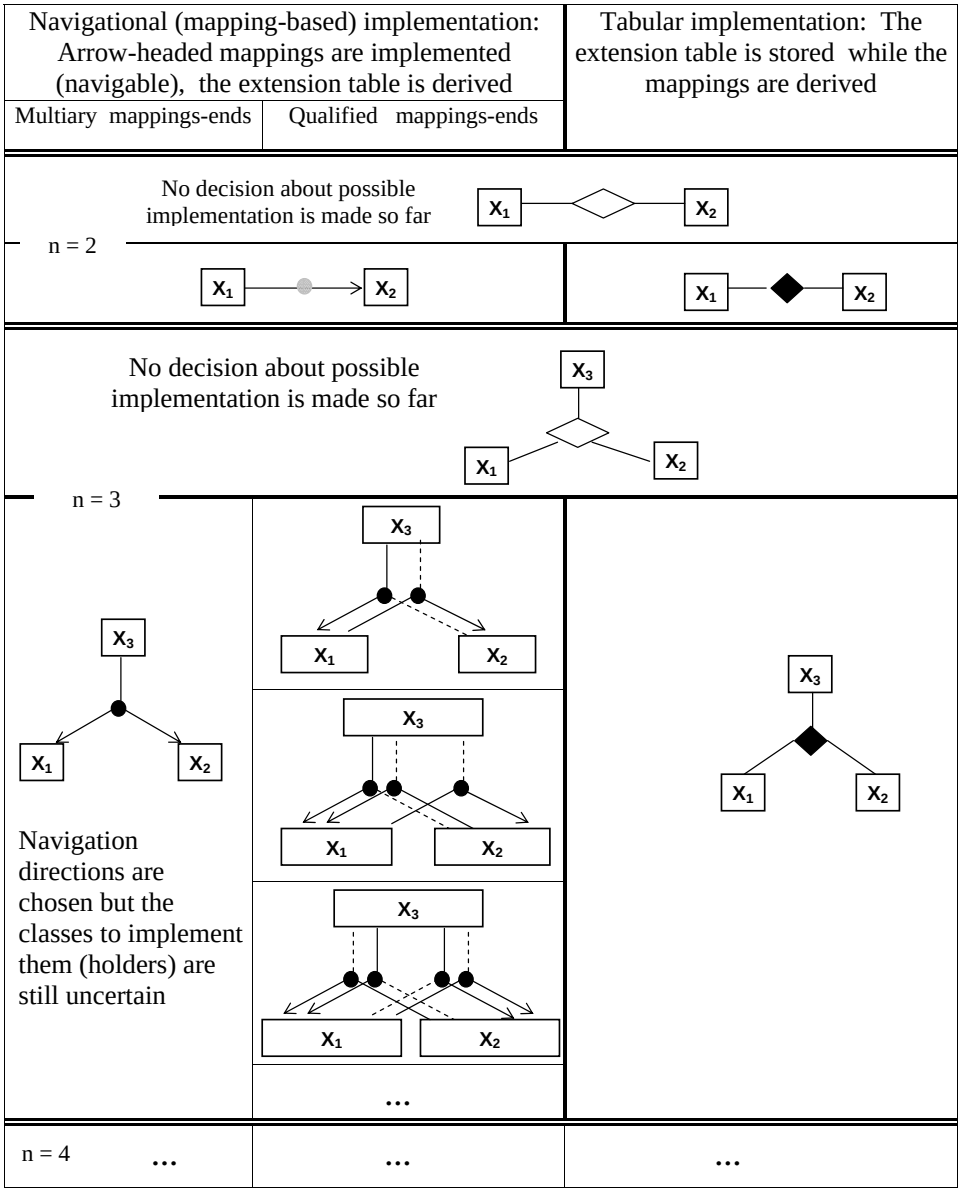


Table 3. Consistent notation for associations (a proposal)

At first glance, this is just a particular constraint saying whether duplicates in the association are allowed or not (it well corresponds to the second phrase of the description above). However, this evident interpretation turns out to be problematic under a more careful inspection. Consider again our main example in cell (a1), Table 1. If the constraint **{key}** is satisfied, then there are no duplicate links/rown in the extension table [cell (a0)] and hence there are no duplicates in any of the collections *lead**, *help**,*mod** retrieved by the ends. If the constraint **{key}** is not declared, then duplicate links/rows in table (a0) are possible and hence, each of the ends will retrieve a bag rather than a set. It follows then that all ends must have either *isUnique=True* or *isUnique=False* simultaneously; the situation when some of them have *true* and some *false* is inconsistent. However, the Standard never says about this requirement and it seems that the folklore well admits the case when only some of the ends are *Unique*. The problem generated a special discussion during preparation of UML 2 [**?**].

A reasonable way to treat the issue was proposed by Dragan Milicev. [12]. Suppose that the extension table of the association in question has duplicates and hence the ends/structural mappings (Definition 3.1) retrieve bags rather than sets. In this case, for each of the mappings f_i $(i = 1..n)$, there is its version $f!_i$ with the same extension but with duplicates eliminated. Thus, together with a structural map $M_S = (f_1..f_n)$ we have its duplicates-eliminated version $M!_S = (f!_1..f!_n)$. Now a navigable end of an association is an element of the set $M_S \cup M_S!$ rather than M_S and, hence, some of the navigable ends can be *Unique* while others are not. (Evidently, though playing with the *Unique*-*nonUnique*-attributes for non-navigable ends is formally possible, it really makes sense for navigable ends). This interpretation is considered by Milicev in detail in [12].

Note, however, that as soon as we consider Uniqueness as a design decision rather than a constraint, to be consistent we need to go further and admit the situation when some of the Curried realization of a navigable end are *Unique* while others are not. For instance, in our main example of *Task* association, suppose that the end *help** is navigable (see the top row in Table 2), duplicates are allowed and the qualified mapping $\underline{help}^*_1(lead{:}Person){:}\, Model \twoheadrightarrow Person$ is chosen to be bag-valued while its counterpart $\underline{help}^*_2(mod{:}Model){:}\, Person \twoheadrightarrow Person$ is chosen to be set-valued. In this case, the end *help** has neither *isUnique*=True nor *isUnique*=False. It follows then that the consistent realization of Milicev's idea leads to assignment of *Unique* or *nonUnique* attributes to qualified ends of associations rather than to (structural) ends.

Moreover, it may well happen for a navigable qualified end g that we need both its versions (with and without duplicates) to be efficiently implemented. It means that rather than declaring the end g to be unique or non-unique, we need to say which of the ends, g or $g!$, or both, occur into the set of navigable ends (that is, those ends that must be efficiently implemented).

4.1 **Definition: Operational view of association.** Let $A = (T, M_S, \boldsymbol{M_Q})$ be an association as defined in 3.5. Let further duplicates are allowed and $M!_S =$

$\{f!_1..f!_n\}$ and $M!_Q = M!_{C1} \cup ... \cup M!_{Cn}$ with $M!_{Ci} = \mathsf{Curry}(f!_i) = \{\underline{f!_{i1}}...\underline{f!_{i(n-1)}}\}$ denote, respectively, maps of duplicate-eliminated ends and qualified ends.

A *possible implementation* of A is merely a non-empty subset $B \subset (M_Q \cup M!_Q \cup \{T\})$ of *basic* elements. The set of *navigable qualified ends* is given by $N_q \stackrel{\text{def}}{=} B \cap M_Q$ and the set of *navigable (structural) ends* is $N \stackrel{\text{def}}{=} \{\mathsf{unCurry}(g) : g \in N_q\}$.

The qualified ends in the set $B \cap M!_Q$ can be considered as those with *isUnique*=True and those in $B \cap M_Q$ are with *isUnique*=False, not excluding the possibility of having both.

4.7 UML in the light of formalization (C)

If we want to be logically consistent with use of *isUnique* attribute/flag, then we have to choose from one of the disciplines (1) or (2) below.

1. If the flag is considered as a constraint to association, then either all ends are *Unique* or all ends are *nonUnique* simultaneously. In the former case duplicate links are *not* allowed, in the latter they are allowed.
2. If *isUnique* is considered as a design decision, then (2a) setting the flag for a non-navigable end is senseless, (2b) different navigable ends may have different values and moreover, (2c) some of them can have both values, *isUnique* and *nonUnique*, as well. In addition, (2d) we need a special notation for the case when duplicate links are not allowed. Indeed, if some of the ends are *nonUnique*, then duplicate links are certainly allowed. However, if all ends are declared *Unique*, it may mean either that there are no duplicate links at all or that duplicate links are allowed but we have chosen the duplicate-eliminated versions of the ends. To distinguish between these two cases, we may agree to declare the absence of duplicates by an explicit declaration like that one shown in cell (a1) Table 1.

In either of the disciplines, managing duplicate elimination/Uniqueness through a Boolean-valued attribute assigned to association ends is misleading.

Operationally, an association is a pair (A, B) with A a full collection (an atlas) of mappings as defined in 3.5 and 4.1, and B its non-empty subset. The Standard has chosen to represent A by its structural map M_S and hence, in UML, an association is a pair (M_S, B). As we have seen, such a definition leads to models underspecified in two directions: because the map M_Q is disregarded, we do not know which classes are to be the holders of the navigable ends, and which of the qualified versions are chosen to be duplicate eliminated. To summarize, the qualified ends are more fine-grained units of the association construct than usual (structural) ends and hence the operational view of Association should be based on the former rather than the latter. Only for the binary case n=2, the issue disappears because in this case $M_Q = M_S$. Yet even for the binary case, the possibility (2c) is missing from the Standard.

5 An accurate formal model for associations: Names, names, names....

Our goal in this section is to specify all the constructs we need, finishing at a precise definition of association, in an accurate way with explicit separation between syntax and semantics. The first step is to set a proper framework for working with names/labels in labeling bags and similar constructs.

5.1 Basic definitions and conventions.

5.1 **Definition: Roles and contexts.** Let $\mathcal{L} = \{\ell_1 ... \ell_n\}$ be a *base* set of n different labels/symbols called *role names* and $\mathcal{X}$ a disjoint set of symbols called *class names*.

(i) A *role* is a pair $\ell{:}X$ with $\ell \in \mathcal{L}$ a role name and $X \in \mathcal{X}$ a class name. A *context* is a finite set of roles $\mathbf{X}_{\mathcal{L}} = \{\ell_1{:}X_1, \ldots, \ell_n{:}X_n\}$ such that all role names are distinct (while the same class name may appear with different roles). We will also write $\mathbf{X}$ for $\mathbf{X}_{\mathcal{L}}$. In fact, a context is a mapping $\mathbf{X}\colon \mathcal{L} \to \mathcal{X}$ from a set of role names to a set of class names, and we will also write $X(\ell)$ or X_ℓ for the class name X in the pair $(\ell{:}X)$. Cardinality of the base set is denoted by $|\mathcal{L}|$ and called the *arity* of the context. For example, the sets $\{$*lead*:*Person*, *help*:*Person*, *mod*:*Model*$\}$ and $\{$*course*:*Subject*, *student*:*Person*, *professor*:*Person*$\}$ are ternary contexts.

(ii) Our definitions will be parameterized by some context $\mathbf{X}$. We will say that the notions are defined *over the context* $\mathbf{X}$. Given $\mathbf{X}$, any subset $\mathcal{K} \subset \mathcal{L}$ of role names uniquely determines a context $\mathbf{X}_{\mathcal{K}} = \{\ell{:}X_\ell \mid \ell \in \mathcal{K}\}$. We will often say that *a construct is over* $\mathcal{K}$ meaning that it is over $\mathbf{X}_{\mathcal{K}}$.

5.2 **Construction: classes and their states.** (i) For our goals in this section, classes are named sets of elements (called objects). Given a time moment t, each class name X is assigned with a set of its elements $[\![X]\!]^t$; we may also consider t as a reference to the state of the system at moment t and call $[\![X]\!]^t$ the *state* of class X at the system state t.

We are not going to consider dynamics of associations and participating classes, and all our semantic notions will be related to some arbitrary but fixed moment/state t. Hence, the superscript t can be omitted but it is useful to keep in mind that all our semantic notions are in fact synchronized. For example, when we write $[\![X_1]\!]$ and $[\![X_2]\!]$ we actually mean $[\![X_1]\!]^t$ and $[\![X_2]\!]^t$ for the same common t.

Below, by an abuse of terminology we will call class names just classes.

(ii) Given a subset $\mathcal{K}$ of $\mathcal{L}$, we write $\bigcup_{\mathcal{K}} \mathbf{X}$ for $\bigcup \{[\![X_\ell]\!] \mid \ell \in \mathcal{K} \subset \mathcal{L}\}$. We also write $\bigcup \mathbf{X}$ for $\bigcup_{\mathcal{L}} \mathbf{X}$. We also remind the reader our convention about distinguishing general and functional mappings described in Definition (3.1).

5.2 Schemas, their instances and states.

In what follows, some context $\mathbf{X}_{\mathcal{L}}$ or just $\mathbf{X}$ is assumed to be given.

5.3 **Definition: Table schemas, links and tables.** A *table schema* is a pair $T = (\mathbf{X}, R)$ with $\mathbf{X}$ a context and R a name disjoint from $\mathcal{L}$.

An *instance* of schema T, or else a *link* over T, is a functional mapping $r\colon \mathcal{L} \to \bigcup \mathbf{X}$ s.t. $r(\ell) \in X_\ell$ for all $\ell \in \mathcal{L}$. The set of all possible links over $\mathbf{X}$ is a labeled Cartesian product and will be denoted by $\prod_{\mathcal{L}} \mathbf{X}$ or just $\prod \mathbf{X}$. If $\{(\ell : X) \mid \ell \in \mathcal{K}\}$ is a sub-context of $\mathbf{X}$ for some $\mathcal{K} \subset \mathcal{L}$, we will write $\prod_{\mathcal{K}} \mathbf{X}$ for the set of the corresponding sub-links.

A *state* of T is a set of instances over T, that is, a set $[\![R]\!]$ of links or rows. Evidently, a state is nothing but a table with columns named by role names, in fact, roles because each column also has its domain X_ℓ specified. Given a state $[\![R]\!]$, each role label ℓ determines a *projection* function $p_\ell\colon [\![R]\!] \to [\![X_\ell]\!]$ by setting $p_\ell(r) \stackrel{\text{def}}{=} r(\ell)$. Taken together, these functions generate a function $p_{\mathcal{L}} \stackrel{\text{def}}{=} \prod_{\ell \in \mathcal{L}} p_\ell\colon [\![R]\!] \to \prod_{\mathcal{L}} \mathbf{X}$, which makes the set $[\![R]\!]$ a multi-relation over $\mathbf{X}$. When we consider the duplicate-eliminated version, we first form a new name $R!$, and then set the state $[\![R!]\!]$ to be the set $[\![R]\!]$ with duplicates eliminated, $[\![R!]\!] \subset \prod \mathbf{X}$.

5.4 **Definition: Mapping schemas, directed links and mappings.** A *mapping schema* over $\mathbf{X}_{\mathcal{L}}$ is a triple $F = (\mathcal{S}, \mathcal{Q}, \ell)$ with $\mathcal{S}$, $\mathcal{Q} \subset \mathcal{L}$ and $\ell \in \mathcal{L}$ such that the triple $(\mathcal{S}, \mathcal{Q}, \{\ell\})$ is a partition of $\mathcal{L}$. The set $\mathcal{Q}$, but not $\mathcal{S}$, is allowed to be empty.

An *instance* of schema F, or a *directed (qualified) link* over F, is a triple $r = (r_{\mathcal{S}}, r_{\mathcal{Q}}, r_\ell)$ with $r_{\mathcal{S}}\colon \mathcal{S} \to \bigcup_{\mathcal{S}} \mathbf{X}$ a link over $\mathcal{S}$, $r_{\mathcal{Q}}\colon \mathcal{Q} \to \bigcup_{\mathcal{Q}} \mathbf{X}$ a link over $\mathcal{Q}$, r_ℓ is an element (object, value) of $[\![X_\ell]\!]$.

A *state* of F is a set L_F of directed links over F, and it is easy to see that any state generates a mapping

$$\overrightarrow{L_F}\colon \textstyle\prod_{\mathcal{S}} \mathbf{X} \to [\prod_{\mathcal{Q}} \mathbf{X} \twoheadrightarrow [\![X_\ell]\!]]^{7},$$

which we will call a *mapping over schema* F. Correspondingly, we call the sub-contexts $\mathbf{X}_{\mathcal{S}}$ and $\mathbf{X}_{\mathcal{Q}}$ the *source* and the *qualifier* of F respectively, and the class X_ℓ the *type* of F.[8]

Of course, there can be different states/mappings over the same schema. However, for a fixed given state of a given association A, each mapping schema uniquely identified a mapping and hence can be considered as a name of this mapping. That is, if t denotes a time moment or the state of A at this moment, then given a schema F, the mapping $\overrightarrow{L_F}$ is uniquely determined. Hence, our notation for it is $[\![F]\!]^t$. For another moment u, we may have another mapping $[\![F]\!]^u$ but we consider these two mappings as two different states of the same mapping name/schema F.

In concrete syntax, using a triple $(\mathcal{S}, \mathcal{Q}, \ell)$ as a name is not convenient and we can employ various naming tips: sub- and super-indexes or/and underlying and

[7] we remind that expression $[A \twoheadrightarrow B]$ denotes the set of all mappings from set A to collections built from elements of B, see Definition 3.1

[8] If $\mathcal{Q} = \varnothing$, then $\prod_{\mathcal{Q}} \mathbf{X} = 1$ (a canonic singleton set) and hence $[\prod_{\mathcal{Q}} \mathbf{X} \twoheadrightarrow [\![X_\ell]\!]] \cong \mathsf{coll}([\![X_\ell]\!])]$

the like. For instance, the mapping name *help** in our *Task* example from section 2 is a shorthand for the mapping schema $(\{lead, mod\},\ \emptyset,\ help)$ (see Table 1. Similarly, the name $\underline{help^*{}_1}$ is a shorthand for the schema $(\{mod\},\ \{lead\},\ help)$ and the name $\underline{help^*{}_2}$ is a shorthand for $(\{lead\},\ \{mod\},\ help)$. When the source contexts are singletons, it is customary in concrete syntax to omit the role names near the source classes and write, for example,
$\underline{help^*}_2(mod : Model) : Person \twoheadrightarrow Person$. Given the mapping name and the parameter name, the source role name can be figured out if needed. All these stars,underlines and conventions live in the world of concrete syntax while in the abstract syntax we have mapping schemas.

5.5 **Construction: Currying.** Let $F = (\mathcal{S}, \mathcal{Q}, \ell)$ be a mapping schema and $[\![\, F \,]\!]$ its state. If $\mathcal{S}' \subset \mathcal{S}$, then we can apply Currying to $[\![\, F \,]\!]$ w.r.t. the arguments in $\mathcal{S}'$ and obtain a mapping

$$\mathsf{Curry}_{\mathcal{S}'}[\![\, F \,]\!] : \textstyle\prod_{\mathcal{S}\setminus\mathcal{S}'} \mathbf{X} \to [\prod_{\mathcal{Q}\cup\mathcal{S}'} \mathbf{X} \twoheadrightarrow [\![\, X_\ell \,]\!]]$$

with the same extension, $\mathsf{ext}(\mathsf{Curry}_{\mathcal{S}'}[\![\, F \,]\!]) = \mathsf{ext}([\![\, F \,]\!])$. The schema of this mapping is

$$Curry_{\mathcal{S}'}(F) \stackrel{\text{def}}{=} (\mathcal{S} \setminus \mathcal{S}', \mathcal{Q} \cup \mathcal{S}', \ell),$$

where *Curry* denotes the syntactical side of Curry operation. We thus have $[\![\, Curry_{\mathcal{S}'}(F) \,]\!] = \mathsf{Curry}_{\mathcal{S}'}[\![\, F \,]\!]$. Correspondingly, we have the inverse operations, $unCurry_{\mathcal{S}'}$ for schemas and $\mathsf{unCurry}_{\mathcal{S}'}$ for mappings, and $[\![\, unCurry_{\mathcal{S}'}(F) \,]\!] = \mathsf{unCurry}_{\mathcal{S}'}[\![\, F \,]\!]$. An algebraically minded reader can notice that it means that the semantics functor $[\![\, - \,]\!]$ is a homomorphism w.r.t. Curry operations.

5.6 **Construction: Duplicate elimination.** If $[\![\, F \,]\!]$ is a mapping as defined in 5.4, we can apply to its target collections the procedure of duplicate elimination and obtain another mapping

$$[\![\, F \,]\!]\mathbf{!} : \textstyle\prod_{\mathcal{S}} \mathbf{X} \to [\prod_{\mathcal{Q}} \mathbf{X} \twoheadrightarrow^{!} [\![\, X_\ell \,]\!]],$$

whose target collections are necessarily sets, not bags (note the !-superindex near the arrow head). We consider such a mapping as a state of a new mapping schema $F! \stackrel{\text{def}}{=} (F, !) = [(\mathcal{S}, \mathcal{Q}, \ell), !]$, where ! is some new symbol disjoint to any of the labels we used. The source, qualifier and type of schema $F!$ are the same as for schema F, the only difference is in the new symbol "!" attached to the schema. We may consider "!" as a two-valued flag/attribute for schemas so that $F!$ means that the value of the flag ! for F is set to True.

Now we define a state of !-valued schema $F!$ as a mapping

$$[\![\, F! \,]\!] \stackrel{\text{def}}{=} [\![\, F \,]\!]\mathbf{!} : \textstyle\prod_{\mathcal{S}} \mathbf{X} \to [\prod_{\mathcal{Q}} \mathbf{X} \twoheadrightarrow^{!} [\![\, X_\ell \,]\!]],$$

and we again have semantic functor $[\![\, - \,]\!]$ being a homomorphism w.r.t. the !-operation. (Note the difference in notation: in syntax we use a normal font for ! while the corresponding semantic operation is denoted by bold !).

Following UML, we could also treat formation of schema $F!$ in a slightly different way as introducing a Boolean-valued flag/attribute for the role names/labels

rather than for schemas. Then each role name appears in two versions: ℓ and $\ell!$. However, !-valued names are allowed to appear only in the type contexts and are disallowed in the sources and qualifiers. To avoid keeping this restriction, we prefer to consider ! as a flag for schemas (rather than role names) as it was defined above.

Our previous definitions of structural, Curried and qualified maps in sections 3,4 can be reformulated for the general $\mathbf{X}$-context situation in a quite straitforward way. Here is some details.

5.7 **Definition: Maps over a context.** What we called earlier a *structural mapping* over $\mathbf{X}$ is a mapping as above with the empty qualifier, $\mathcal{Q} = \emptyset$. What we called earlier a *qualified* mapping is a mapping as above with a source context being a singleton, $\mathcal{S} = \{\ell'\}$ for some $\ell' \in \mathcal{L}, \ell' \neq \ell$. We will sometimes refer to general schemas with "more than singleton" sources and non-empty qualifiers as *mixed*. Our earlier Curry operation is a particular case of 5.5 when $\mathcal{S} \setminus \mathcal{S}'$ is a singleton. We will sometimes call it *full Currying* and the result a *fully Curried/qualified* schema/mapping.

A given context $\mathbf{X}$ of arity $n = |\mathcal{L}|$ generates n structural mapping schemas

$$F_\ell = (\mathcal{L} \setminus \{\ell\},\ \emptyset,\ \ell),\ \ell \in \mathcal{L},$$

each one exists in the two variants, F_ℓ and $F!_\ell$, and we call the sets $M_S(\mathbf{X}) = \{F_\ell \mid \ell \in \mathcal{L}\}$ and $M!_S(\mathbf{X}) = \{F!_\ell \mid \ell \in \mathcal{L}\}$ the *structural maps* of $\mathbf{X}$.

The context $\mathbf{X}$ also generates $n(n\text{-}1)$ fully Curried/qualified mapping schemas

$$\underline{F_{\ell\ell'}} = (\{\ell'\},\ \mathcal{L} \setminus \{\ell, \ell'\},\ \ell),\ \ell, \ell' \in \mathcal{L},\ \ell' \neq \ell,$$

again in the two variants, and we call the sets

$$M_Q(\mathbf{X}) = \left\{\underline{F_{\ell\ell'}} \mid \ell, \ell' \in \mathcal{L},\ \ell' \neq \ell\right\}$$

and $M!_Q(\mathbf{X}) = \{\underline{F!_{\ell\ell'}}\}$ the *(fully) qualified maps* of $\mathbf{X}$.

Also, for a given ℓ, the set $M_{C\ell} = \{\underline{F_{\ell\ell'}}\}$, $\ell' \neq \ell$ is a *(fully) Curried map*, $\boldsymbol{M_Q} = \{M_{C\ell} \mid \ell \in \mathcal{L}\}$ and similarly $\boldsymbol{M!_Q}$ are the *(fully) qualified atlases* of $\mathbf{X}$. Note that M_Q, $M!_Q$ are the flattened version of $\boldsymbol{M_Q}$,$\boldsymbol{M!_Q}$ resp., $M_Q = \bigcup \boldsymbol{M_Q}$ and $M!_Q = \bigcup \boldsymbol{M!_Q}$.

5.3 What is Association: an informal discussion and a formal definition.

Let $\mathbf{X}$ be a context, which we assume to be thought of as the context of some association. As we discussed above, mapping schemas over $\mathbf{X}$ can simultaneously serve as names for the corresponding mappings (states) over $\mathbf{X}$, and hence the context $\mathbf{X}$ generates a pool of unique names for all the mapping involved. Because all these mappings are assumed to have the same extension table (for a given state of the association), each one of them generates all the other (mutual invertibility) and we indeed have a single state of the association. Particularly, this single state possesses

- an extension (data table) $[\![T]\!]$,
- n structural mappings/ends $[\![F]\!]$ for each schema $F \in M_S(\mathbf{X})$ plus n duplicate-eliminated versions $[\![F!]\!]$,
- $n(n\text{-}1)$ fully qualified mappings/ends $[\![G]\!]$ for each schema $G \in M_Q$ plus their duplicate-eliminated versions $[\![G!]\!]$ and, finally,
- a pool of non-fully qualified mappings of mixed schemas in-between the structural and fully qualified schemas.

Fortunately, the Standard does not consider the latter as an interesting component of associations, but even the first three components provide a rich arsenal of objects, which includes a set of rows/links and a set of (sets of) mappings (projections, structural and qualified ends, the latter two in the two versions). When arity $n \geq 3$, the total number of mappings is $n+2[n+n(n-1)] = 2n^2+n$; for n=2, because M_S and M_Q coincide, we have only $n + 2n = 3n$ mappings.

Since all these objects are mutually derivable from each other, for their implementation we do not need to implement/store each of them immediately. Rather, we may choose to directly implement some of the objects, which we need to have really efficient, and leave all the rest for retrieval (perhaps, non-efficient) only if needed. Following the database jargon, we can call the former objects *basic* and the latter *derived*. Moreover, operations lookUp, ext, Curry, unCurry and **!** (duplicate elimination) can be considered as possible queries against the basic elements to retrieve the derived ones.

A standard database way of implementing an association is to store the table and implement the mappings by the corresponding queries. If we need some of these queries executed efficiently, we can index the table in the corresponding directions. An indexed table is then can be considered as an implementation of the corresponding navigable end. A standard OO-programming way of implementing an association is to define efficient retrieval/*get* methods in the corresponding classes for navigational ends, and create a special table-object if we need to get the extension efficiently. In either case, a reasonable association schema should specify those objects that we need to implement efficiently (basic objects) and leave the rest unspecified for "general ad hoc querying" by means of operations lookUp, ext, Curry, unCurry and **!** if needed. In the OO-world, where mapping procedures have to be assigned to classes (the holders of the methods), a natural unit of specification is a (fully) qualified rather than a structural or mixed mapping schema. Thus, all that we need to do is to specify a subset of all possible such schemas to be implemented efficiently. Here is a precise definition.

5.8 **Definition. Associations operationally, I: Syntax.**

An *association schema* is a triple $A = (\mathbf{X}, R, B)$ with the following components.

1. $\mathbf{X} = \{(\ell{:}X_\ell) \mid \ell \in \mathcal{L}\}$ is a context. It determines two sets (maps) of schemas, $M_Q(\mathbf{X})$ and $M!_Q(\mathbf{X})$, as defined in 5.7.
2. $R \notin \mathcal{L}$ is a name to be thought of both the association name and the extension table name. It determines a table schema $T = T(A) = (R, \mathbf{X})$.
3. $B \subset (M_Q(\mathbf{X}) \cup M!_Q(\mathbf{X}) \cup \{T\})$ is a subset of *basic* schemas.

Schemas occurring in the set

$$N_q = B \cap [M_Q(\mathbf{X}) \cup M!_Q(\mathbf{X})]$$

are called *navigable qualified ends*. In addition, those from the set $N_q^0 = B \cap M_Q(\mathbf{X})$ are called *non-unique* and those from $N_q^! = B \cap M!_Q(\mathbf{X})$ are *unique*.

The set $B \cap \{T\}$ is a singleton, and if it is not empty, i.e., $T \in B$, we say that the association's extension is *to be stored* in a table with schema T. Thus, the set of basic elements consists of the following disjoint components: $B = N_q^0 \cup N_q^! \cup \{T\}$, which are all optional but at least one of them must be included into a valid association schema.

Note that although the sets N_q^0 and $N_q^!$ are disjoint, the situation when $F \in N_q^0$ and $F! \in N_q^!$ is not excluded (and then we may say that the end F is both unique and non-unique). It merely means that we need both versions of the end, with and without duplicates, to be implemented efficiently.

The set of *navigable (structural) ends* considered in UML is given by

$$N \stackrel{\text{def}}{=} \{unCurry(g) : g \in N_q\} \subset M_S(\mathbf{X}) \cup M!_S(\mathbf{X}).$$

We will generically refer to the elements of constructs from which an association schema A is built as to A's *elements*. Thus, roles, names, schemas are (modeling) elements.

5.9 **Definition. Associations operationally, II: Semantics** Let $A = (\mathbf{X}, R, B)$ be an association schema as defined above. A *state* of A is a set of objects

$$[\![\, A \,]\!] \stackrel{\text{def}}{=} \{[\![\, e \,]\!] \mid e \in B\}$$

such that

1. if $e = T \in B$ is the A's table schema, then $[\![\, e \,]\!]$ is a table $[\![\, T \,]\!]$ over this schema as defined in Definition 5.3,
2. if $e = F \in N_q^0$ is a navigable non-unique qualified end/schema, then $[\![\, e \,]\!]$ is a qualified mapping $[\![\, F \,]\!]$ of schema F as defined in 5.4,
3. if $e = F! \in N_q^!$ is a navigable unique qualified end/schema,then $[\![\, e \,]\!]$ is $[\![\, F \,]\!]\mathbf{!}$, that is, a mapping of schema F but with duplicates eliminated.
4. Moreover, all objects $[\![\, e \,]\!]$ are mutually invertible. In this sense the entire collection $[\![\, A \,]\!]$ is indeed a single state of the association.

We may call the total collection of association's ingredients,

$$\{[\![\, e \,]\!] \mid e \in M_S(\mathbf{X}) \cup M!_S(\mathbf{X}) \cup M_Q(\mathbf{X}) \cup M!_Q(\mathbf{X}) \cup \{T\}\}$$

the *virtual* state of A. The state defined above is just a subcollection of this virtual state to be implemented. In our geographical terms, an association is an atlas of maps of the same territory (the extension of the association), with each map presenting a net of all possible roads (mappings) of a specified sort. What an association schema does is selecting a few roads in the atlas to make them effective transportation routes (to be efficiently implemented).

A diagrammatic representation of the syntactical half of the definition above, that is, the metamodel, is given in Fig. 7. Nodes denote sets of syntactical objects and edges are meta-mappings between them. The constraints **{def}** hung on tuples of arrows with a common source mean that any object in the source is a tuple of elements, say, a Role object is a pair (name, className). Particularly, the label **{def}** hung on edge "content" says that a Context object consists of n Role-objects. Dashed nodes and edges denote derived sets of elements. For example, having a Context, we derive the corresponding map $M_Q(\mathbf{X})$ of qualified mapping schemas, the map $M_S(\mathbf{X})$ of structural mapping schemas and the table schema $(\mathbf{X}, R)$. Note that the presence of derived metaclasses QualifiedMappingSchema and TableSchema is necessary in the metamodel in order to show basic metaassociations *navig-qEnd* and *storedTab*. In contrast, metaclass StructMappingSchema is only shown for illustrative purposes and better comprehensibility.

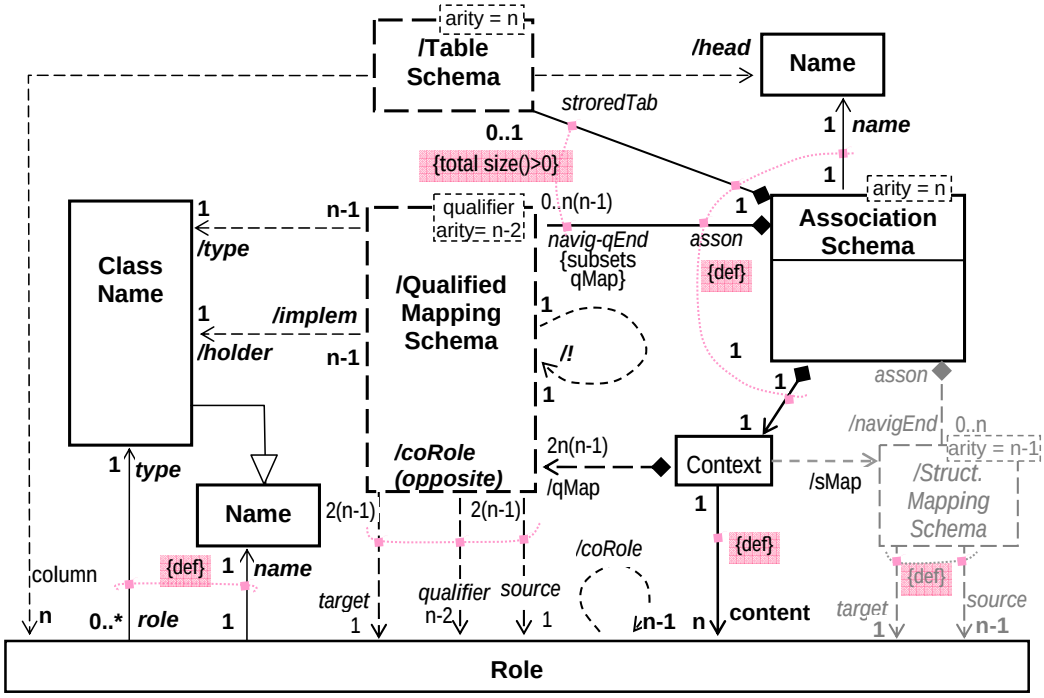


Fig. 7. Metamodel of Definition 5.8, mathematically

5.10 **Discussion: Is association schema a classifier?** Suppose that an association schema A consists of a table schema T and two mapping schemas F_1, F_2. A table schema is a classifier and its instances are links (Definition 5.3). A mapping schema is also a classifier and its instances are directed (qualified) links (Defi-

nition 5.4). Could we say that A is also a classifier whose instances are triples of links $r = (r_0, r_1, r_2)$ with $r_0 \in [\![T]\!]$ and $r_i \in [\![F_i]\!]$, $i = 1, 2$? It would be a nice picture but, unfortunately, it is not correct. The point is that all three links in question are not independent: $r_{1,2}$ are nothing but the same link r_0, which is navigated in two different directions. In fact, A's instances are A's extension table instances (irrespectively to whether T is included into the schema or not), and these instances are navigated in a few directions prescribed by A. Strictly speaking, an association schema is not a classifier but a typical classifier (schema $T(A)$) can be derived from it.

6 Discussion: UML's associations in the light of formal semantics and formal modeling

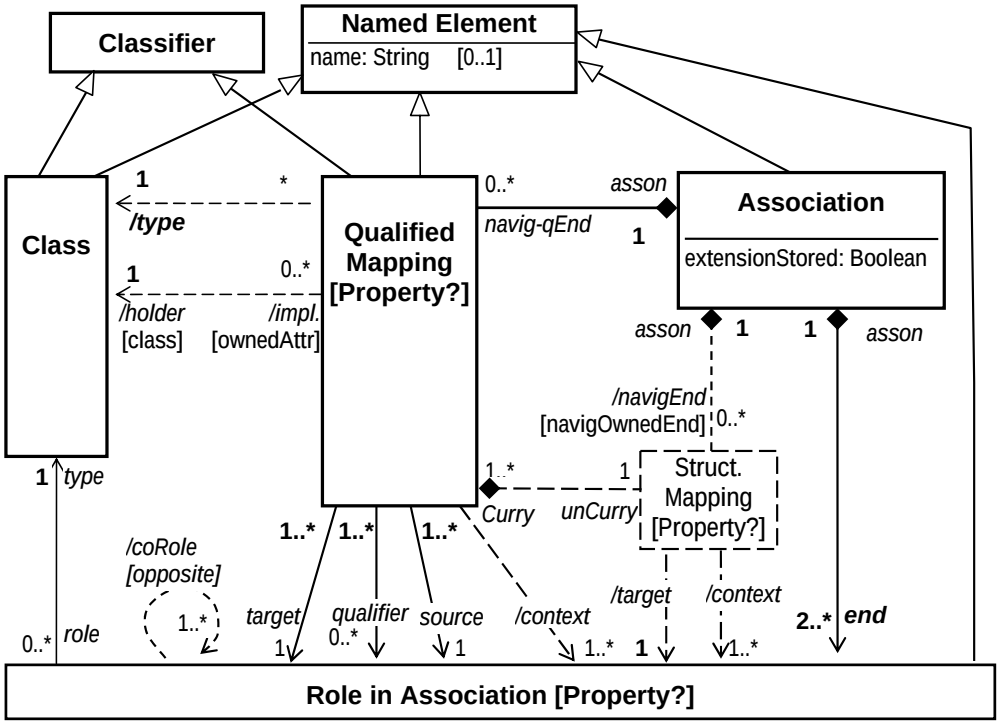
In this section, we arrange the metamodel of our formal definition in UML metamodeling style and compare it with the UML metamodel. This analysis summarizes and clarifies our previous comparative sections 3.4, 4.4, 4.7 and answers to some of the earlier questions but still reveals a few new flaws/deformations in the UML metamodel. Then we will try to figure out a plausible explanation for why the UML metamodel is deformed.

6.1 What is distorted in the UML metamodel

The metamodel in Fig. 7 is a diagrammatic specification of our definition 5.8. The metamodel in Fig. 8 is its arrangement in the UML metamodeling style with the arities of the constructs implicit; results of Discussion 5.10 are also added. This metamodel is immediately comparable with the UML metamodel in Fig. 2, and this comparison reveals the following results.

The Ubiquitous Property. The most striking observation is that UML metamodel glues together three different metaclasses specifying three different notions: Role, QualfiedMapping Schema and StructuralMapping Schema, into one metaclass Property (a typical case of the Cylinder Syndrome). Especially distorting is merging class Role with Mapping classes. Particularly, the metaassociation "qualifier" becomes a loop with incomprehensible meaning, and a principal (for the association construct) notion of qualified mapping is left unspecified. It seriously deforms the entire metamodel and, in fact, makes it hardly suitable to fulfill its primary function: to specify the concepts in a precise and unambiguous way.

Holdership vs. ownership. The metaassociation "holder" specifying the source class of a qualified mapping is mistakenly considered as ownership. In the previous version of the Standard, it resulted in a entirely artificial interference between ownership and navigability. This interference is deprecated in the newest version but the metamodel still keeps the metaassociation ownedEnd subsetting memberEnd (constraint (5) in Fig. 2). However, we have



Definitions for QualifiedMapping:

(27) $\mathsf{self.type} = \mathsf{self.target.type}$

(28) $\mathsf{self.holder} = \mathsf{self.source.type}$

(29) $\mathsf{self.context} = \mathsf{union(self.source, self.qualifier)}$

Constraints for QualifiedMapping:

(30) $\mathsf{disjoint(self.target, self.source, self.qualifier)}$

Definitions for Association:

(31) $\mathsf{self.navigEnd} = \mathsf{self.navig\text{-}qEnd.unCurry}$

Constraints for Association:

(32) $\mathsf{union(self.navig\text{-}qEnd.target, self.navig\text{-}qEnd.context)} = \mathsf{self.end}$

(33) $\mathsf{self.navig\text{-}qEnd\text{->}size()}{=}0 \text{ implies } \mathsf{extensionStored}{=}\mathsf{True}$

(34) $\mathsf{self.navig\text{-}qEnd\ \text{->}forAll(g1,g2|\ g1.unCurry.ext} = \mathsf{g2.unCurry.ext)}$

Definitions for Role:

(35) $\mathsf{self.coRole} = \mathsf{self.asson.end} - \{\mathsf{self}\}$

Fig. 8. Metamodel of Definition 5.8 rendered in the UML metamodeling style. Particularly, metaclasses are named "semantically" (compare with metamodel in Fig. 7) and arity size constraints are omitted

seen that *all* the ends of an association are owned by the association (section 4.3) and hence metaassociation ownedEnd is entirely redundant and misleading, and must be removed from the metamodel.

Constraints. Two principal semantic constraints (33), (34) are missing from the UML metamodel. A number of size constraints are missing too. Indeed, in the arity-parameterized form of the metamodel Fig. 7, essential size information is captured with metaassociations multiplicities also parameterized by the arity. In the UML-style metamodeling Fig. 8, this information is lost and must be specified with additional constraints. For example, in the QualifiedMapping Context, we have

$$\mathsf{self.qualfier\text{->}size()} = \mathsf{self.asson.end\text{->}size()} - 2$$

In fact, many arity-parameterized multiplicities in Fig. 7 give rise to size constraint like above and must be added to Fig. 8.

Classification. As it was discussed in 5.10, considering Association as a classifier is not justified (or at least, needs special reservations) and the UML metamodel is not quite correct here. Should Property be declared a classifier? Since Mapping schemas are classifiers while Roles are not, and UML's Property subsumes both, this question could not be answered at all. Thus, a fundamental for OO-modeling concept of Classifier is inconsistent with UML's notion of Property.

6.2 Why is the UML metamodel so distorted?

We see a few general issues, the importance of which for proper modeling (in general and associations in particular) is essentially underestimated in UML.

Separating syntax and semantics. Care and accuracy in treating this issue are a must for any modeling language pretending to be precise. In contrast, UML carelessly employs the same terms for similar syntactical and semantic notions, for example, the terms AssociationEnd and Property, or the very term Association, are used in both senses. As is usual in such cases, UML actually tends to understand them semantically, thus leaving many syntactical constructs nameless. We have fixed this problem by introducing a set of syntactical constructs called *schemas* and their semantic counterparts called *states*.

Of course, naming a schema and its state by the same term is a common engineering practice. For example, in the database world, the term Relation is often used to refer to both a relational schema and a relation populating it. Such practices are acceptable and need not be necessarily confusing if a precise formal model of the subject is developed and can be invoked in case of ambiguities (recall our Cylinder analogy). The situation with UML's associations has been essentially different, and systematic use of two-meaning terms could be quite ambiguous.[9]

[9] We hope that now, after we have built a formal model of UML associations, the situation changes and the two-meanings jargon can be used safely.

Setting a metadata operation (query) language explicitly. As we have seen, the very definition of association essentially involves operations (queries) and derived (meta)data. Thus, a proper language for specifying the metamodel has to combine metadata definition and metadata operation functionalities. This is a well-known classical idea in the database world. In the latter, a data definition language without querying mechanisms is considered to be useless, and the standard notion is a data definition *and* manipulation language. In contrast, the UML metamodeling toolbox has fairly weak operation functionalities: there are just few primitive operations like derived union and intersection, but even the metamodel of Association alone needs much more.

Working with labeled structures. UML inconsistently mixes two techniques of working with record-like structures: the labeled style and the (widespread though often irrelevant and confusing) "ordered" style where labels are replaced by natural numbers. In fact, an accurate syntactical mechanism for working with labels and labeled structures is missing from the Standard.

Lessons of data modeling. Metadata is (although specific yet) data and hence metamodeling, as a discipline of modeling metadata, could learn much from data modeling. Particularly, the three issues mentioned above are well known in the data modeling world, and could be adapted for OO metamodeling.

6.3 What we suggest.

Basically, our list of causes of disorder provides the basis for a possible set of corresponding solutions. We will make a few additional remarks.

Syntax vs. semantics: separated yet similar. Generally speaking, what is normally called a metamodel is a precise specification of syntax irrespectively of semantics. However, if semantics is structurally similar to syntax, then the metamodel is also semantically meaningful. This is in itself an important reason to build syntax and semantics in a coherent way as we did above for associations. In addition, there are other essential theoretical as well as practical reasons for having semantics structurally similar to syntax so that semantics of a syntactical construct can be considered as a homomorphism to the respective semantic universe. This idea is well known in algebraic, particularly, categorical logic [17], and in computer science as well, where it is usually termed as compositional semantics. We believe that accurately separating syntax and semantics yet keeping them structurally similar is a fundamental pattern that a reasonable modeling language should follow. However, following this pattern in the case of diagrammatic modeling and metamodeling is technically non-trivial and special mathematical means need to be developed. An essential step in this direction is presented in [6].

Algebra of (meta)data manipulation. The UML metamodel is specified in some core subset of UML consisting mainly of classes and associations. Our analysis of Association shows that even for this small part of the metamodel some data operation language should be added to the core. Moreover, it was

shown in [5] that derived (meta)data and their operation is an essential component of general metamodeling and model management. UML's companion responsible for the task is OCL, and hence UML metamodeling has to be based on some core fragment of UML plus some subset of OCL. The primary question is what this subset should be, and whether even the entire OCL is expressive enough to support the UML metamodel's needs. For example, it may turn out that the Currying operation, which is evidently of higher-order, is not expressible in OCL. Much research is needed here, and its value should not be underestimated.

Ubiquitous Mappings. A mathematical universe consists of sets and mappings between them. As soon as we try to understand semantics of UML constructs in mathematical terms, mappings inevitably appear on the scene (perhaps hidden in logical formulas of usual string-based formalisms). In a sense, this is nothing but a mathematical realization of the epigraph to the paper. An adequate and accurate specification of semantics leads to graph-based structures (sets are nodes and mappings are arrows), which are naturally represented by diagrams. Thus, there is much more to the graphical nature of UML diagrams than just concrete syntax: it is the graph-based semantics that makes UML diagrams an effective modeling tool (if, of course, their syntax follows and reveals the semantics). Thus, making mappings the first-class citizens of UML modeling and metamodeling would help to draw precise semantic foundations under UML.

7 Conclusion

We have found that semantics of the association construct can be uncovered in the explanatory sections of the Standard, where it is described in a piecemeal and informal yet sufficiently consistent way. We have built a mathematical framework, in which these multiple intuitive descriptions can be formally explicated, analyzed, and coherently integrated. The formal model allowed us to explain accurately many delicate aspects of the construct and build a new consistent metamodel for it.

We then compared this metamodel with the UML2 metamodel of association. Our comparative analysis revealed a few essential omissions and distortions in the UML2 metamodel, explained a number of the known problems with associations and detected a few new ones. It showed that the current UML2 metamodel of association is inaccurate, incomplete and inconsistent. We also demonstrated that the current UML notation for associations may lead to essentially incomplete specifications, and proposed a new consistent and unambiguous notation. Finally, we sketched a few general problems inherent to the UML metamodeling style, and suggested some measures to improve it.

Acknowledgements. We are grateful to Bran Selic and Dragan Milicev for a few fruitful discussions. Special thanks go to Bran for showing us many delicate issues in the subject. Our attitude towards listening to the UML spirit but

checking it with the metamodel was inspired and shaped in our numerous talks with him. Thanks also go to Jean-Marie Favre for a stimulating discussion of the linguistic aspects of modeling languages.

References

[1]
[2] D. Akehurst, G. Howells, and K. Mcdonald-Maier. Implementing associations: UML 2.0 to Java 5. *Software and Systems Modeling*, 6:3–35, 2007.
[3] Z. Diskin. Visualization vs. specification in diagrammatic notations: A case study with the UML. In *Diagrams'2002: 2nd Int. Conf. on the Theory and Applications of Diagrams*, Springer LNAI#2317, pages 112–115, 2002.
[4] Z. Diskin and B. Kadish. Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. *Data & Knowledge Engineering*, 47:1–59, 2003.
[5] Z. Diskin and B. Kadish. Generic model management. In Rivero, Doorn, and Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 258–265. Idea Group, 2005.
[6] Z. Diskin and U. Wolter. Generalized Sketches: A Universal Logic for Diagrammatic Modeling in Software Engineering. In *Applied and Computational Category Theory. Satellite Events Handouts, ETAPS'07*, 2007. Final revised version is to appear in ENTCS.
[7] Robert France. A problem-orineted analysis of basic UML static modeling concepts. In L. Meissner, editor, *OOPSLA*, pages 57–69. ACM Press, 1999.
[8] G. Génova, J. Lloréns, and J. Fuentes. Uml associations: A structural and contextual view. *Journal of Object Technology*, 3(7):83–100, 2004.
[9] G. Génova, J. Llorens, and P. Martínez. Semantics of the minimum multiplicity in ternary associations in UML. In M. Gogolla and C. Kobryn, editors, *UML'2001, 4th Int.Conference*, volume 2185 of *LNCS*, pages 329–341. Springer, 2001.
[10] C. Gunter. *Semantics of programming languages*. MIT Pres, 1992.
[11] R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.
[12] Dragan Milicev. On the semantics of associations and association ends in uml. *IEEE Trans. Software Eng.*, 33(4):238–251, 2007.
[13] Dragan Milicev, Bran Selic, and the Authors. Joint E-mail Discussion, Fall 2005.
[14] Object Management Group, `http://www.uml.org`. *Unified Modeling Language: Superstructure. version 2.0. Formal/05-07-04*, 2005.
[15] OMG, Object Management Group, `http://www.omg.org/technology/documents/`. *OCL Document Set*, 2004.
[16] OMG, Object Management Group, http://www.omg.org/docs/formal/07-02-03.pdf. *Unified Modeling Language: Superstructure. Version 2.1.1 Formal/2007-02-03*, 2007.
[17] A. Pitts. Categorical logic. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume VI*. Oxford University Press, 1996.
[18] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
[19] Bran Selic. Personal Communication, Fall 2005.
[20] P. Stevens. On the interpretation of binary associations in the unified modeling language. *Software and Systems Modeling*, (1), 2002.

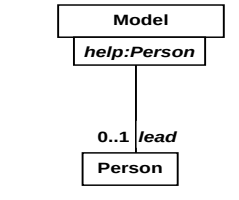
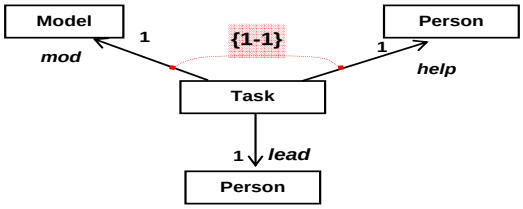
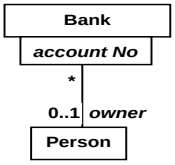
A Appendix. Qualified associations and normalization

In this section we analyze that use of qualified associations that the Standard considers "common" and "having semantic consequences", see quote (Q4) on page 12. Note, first of all, that multiplicity of a qualified mapping (say, $\underline{lead^*}_1$) is nothing but the multiplicity of the corresponding unCurried structural mapping ($lead^*$), and hence the multiplicity 0..1 means that the structural mapping in questions is single-valued (functional). It implies that each link/tuple in the extension table is uniquely determined by its components from the domain of the *-mapping, that is, in our case, by the pair (*help*,*mod*). As a rule, it means that the subtable consisting of these components has a clear semantic meaning and it makes sense to model it by an association class. For instance, in our example we can introduce a new association class *Task*, whose objects are binary links (*help*:*Person*, *mod*:*Model*), and the mapping $lead^*$ becomes an attribute of this class of type *Person*.

Note that the multiplicity of the attribute is 1 rather than 0..1 that the original mapping has. The point is that the source of mapping $lead^*$ is a (labeled) Cartesian product, (*help*:*Person*)×(*mod*:*Person*), and normally a *-mapping (particularly, $lead^*$) is not defined for all tuples from this set, hence the lower bound 0. However, class *Task* is a subset of the Cartesian product above consisting of all pairs appearing in the table, and for such pairs mapping $lead^*$ is defined. Thus, we come to modeling our ternary association *Task* by a binary association class *Task* endowed with a single-valued attribute $lead^*$ of type *Person* as shown in the top row of Table 4. The marker **{1-1}** says that objects of class *Task* are, in fact, pairs (*help*:*Person*, *mod*:*Model*), and for each such a pair one and only one *lead*(er) is defined. The second row in the table shows how this treatment works for an example of qualified association from the Standard.

The issue is well-known and well-elaborated for database modeling: in the relational language, a qualified association with multiplicity 1 means a functional dependency, say, (*help*, *mod*) → *lead*, and our remodeling procedure shown in Table 4 is nothing but the well-known procedure of normalizing relational schemas according to functional dependencies. Thus, here we have a variation of the Cylinder Syndrome, in which UML describes a known construct in its own terms. It would be just an interesting observation but, unfortunately, this "common version" of the construct actually encourages to model associations in a non-normalized way. In other words, "common" qualified association offered by UML is a design pattern the modeler should avoid rather than to follow. Another evidence for this can be found in the analysis of constraints for qualified associations [3].

Table 4. "Semantic consequences" of UML's qualifiers

UML diagram for qualified association	Its intended meaning (formally specified by sets and mappings)
 Our example from section 2	
 The Standard [12, sect.7.3.44, Examples, p.131]