Engineering Associations: From Models to Code and Back through Semantics

Zinovy Diskin[1], Steve Easterbrook[1], and Juergen Dingel[2]

[1] University of Toronto, Canada
{zdiskin,sme}@cs.toronto.edu
[2] Queen's University, Kingston, Canada
dingel@cs.queensu.ca

Abstract. Association between classes is a central construct in OO modeling. However, precise semantics of associations has not been defined, and only the most basic types are implemented in modern forward and reverse engineering tools. In this paper, we present a novel mathematical framework and build a precise semantics for several association constructs, whose implementation has been considered problematic. We also identify a number of patterns for using associations in practical applications, which cannot be modeled (reverse engineered) in UML.

1 Introduction

Modeling is a classical engineering instrument to manage complexity of system design. Its evolution in many branches of mechanical and electrical engineering, and practically everywhere in hardware engineering, has led to automated production. We do not see significant reasons to think that evolution of software engineering (SE) will be essentially different. The recent rapid advance of model-driven development (MDD) in many areas of SE shows potential of the process (cf.[21, 13]). Particularly, there has been a real explosion in the market of forward, reverse and roundtrip engineering tools (MDD-tools).

1.1 The problem. Among modeling notations used in OO analysis and design, Class Diagrams play a central role.[1] A basic type of class diagram is a graph whose nodes are classes and edges are associations between them. The latter present relationships between classes and can bear various adornments expressing properties of these relationships; Fig. 1 shows a few examples. The meaning of diagrams (a,b,d) should be clear from semantics of names. Diagram (c1) says that a pair of objects (c:Company, p:Position) determines zero or one Person object, and similarly (c2) says that a pair (p:Person, c:Company) determines zero or two Positions.

Note that if association ends are considered as directed mappings between classes then we name them by verbs to emphasize asymmetry (diagrams a,b,c).

[1] A survey reported in [10] claims that *all* experts participating in the study evaluated Class Diagrams as the most important and the most useful of the UML notations.

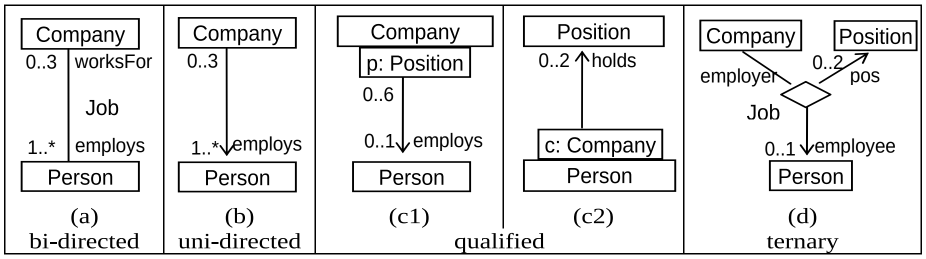


Fig. 1: A few sample associations

We use nouns when the ends are regarded as roles of the association considered as a symmetric relationship between classes (diagram d).

Implementation of these semantics is not quite straightforward, but everything looks manageable and an ordinary MDD-tool should accurately implement these constructs. However, the situation one finds in practice is in sharp contrast with these expectations. For example, the survey reported in [1] says that among ten major UML-based MDD-tools, none have implemented n-ary associations; only two – qualified associations; and only three – bidirectional associations (including those unidirectional, which have multiplicity constraints at both ends like, e.g., in Fig. 1b). In fact, the only (!) association construct well understood and implemented today is unidirectional binary association without multiplicity constraints at the source end (Fig. 1(b) with multiplicity 0..3 removed).

1.2 The causes. The root of the problems with associations is that their simple and compact syntax hides an integrated *system* of concepts, mutually related and dependant of each other. For example, implementation of a bidirectional association in Fig. 1(a) may seem to be as simple as declaring attributes *employs* of type Collection⟨Person⟩ in class *Company* and *worksFor* of type Collection⟨Company⟩ in class *Person*. However, these two attributes are not independent and represent the two ends of the *same* association: updating one of them means updating the entire association, which implies a corresponding update for the other end. Hence, the mutator methods (setters) for the attributes belonging to different classes must be synchronized. The problem is essentially complicated by the possibility to combine several properties for one end and several other properties for the other end. For example, UML does not prevent declaring one end to be bag-valued, qualified and read-only while the other end is set-valued, also qualified and writable. In [1], several such complex cases of feature interaction are identified.

Another big issue is the two-facet nature of associations, which are both extensional and navigational concepts. A common understanding of multi-ary associations sees them as sets of links (tuples) or relations. This view is basic for databases, but software modeling is more concerned with traversing relations in different directions; UML calls this aspect *navigation*. In the navigational view, relations re-appear as extensions (graphs) of the mappings involved, and thus are always implicitly on the stage. Implementation needs a clear and precise specification of the relationship between the two views but the issue is rarely discussed in the literature. Indirectly and informally it is addressed in the UML Standard but its specification in the metamodel is seriously flawed [5]. Since modern MDD-tools are based on metamodeling, an inconsistent metamodel of a modeling construct makes its implementation practically impossible.

Thus, the concept of association encompasses an integral system of navigational and extensional modeling constructs. Its straightforward implementation in modern OO languages necessarily leads to assigning these constructs to different classes. The integrity of the concept is thus corrupted and needs to be recovered by an additional implementation of “synchronization” services. In other words, the concept of association cannot be implemented directly in languages like Java or C++, and needs special *design patterns*. Creating the latter is an issue and may be non-trivial as demonstrated in [1, 16, 14]. In addition, these works differ in their understanding of what associations and their properties are, and how to implement them. A part of this diversity is just normal: the same specification can be implemented in different ways. Yet another part is caused by the absence of precise semantics for the constructs and their subsequent different interpretations by the implementers. This latter part is substantial.

1.3 The approach and results. The problems above show the necessity of unambiguous and transparent semantics for associations formulated in independent mathematical terms. A step in this direction was made in [5] with emphasis on formal definitions and metamodeling for the general case of n-ary associations. In the present paper we continue this work towards practical applications and implementation rather then metamodeling, and focus on binary associations, which are most often appear in practice. Our basic assumption is that precise semantics of a modeling construct, if it is formulated in clear and understandable terms close to programming concepts, makes implementation a technical issue that can be delegated to practitioners. We formulate the following requirements for a mathematical framework to be useful for the task: (i) be expressive enough to capture all aspects of semantics of associations needed in applications; (ii) be abstract enough to avoid the danger of offering only “pictures of code” but simultaneously be understandable and transparent; (iii) be aligned/coordinated with the modeling concepts to facilitate continuity and cross-references between the models and the formalism; (iv) be aligned/coordinated with the programming concepts to facilitate continuity and cross-references between the formalism and the code.

In section 3 we present a formal framework, which we believe is sufficiently satisfactory w.r.t. these requirements, and in sections 4,5 we apply it to formalizing basic types of UML associations. The results of this work are collected in Tables 1,2 and Fig. 4. The left columns of the tables present UML constructs, and middle and right ones show their formal semantics; Fig. 4 is structured similarly. Since our formalism is graph-based and our formal specifications are also diagrams, comparison of formal and modeling constructs is transparent and comprehensible (requirement iii). On the other hand, the main building blocks of our formalism are sets and mappings, which are naturally expressible in terms of, say, generic interfaces of the Java Utility Package [4]; hence, requirement (iv) is satisfied. Conditions (i,ii) will be addressed in section 3.

Some rows of the Tables have their UML cell blank, which means that the corresponding real world situation (formally described in the middle and right cells) cannot be modeled in UML. Owing to computationally completeness of programming languages, such situations can be coded but their adequate reverse engineering into UML is problematic. Moreover, as seen from the tables, these situations are quite natural for practical applications. The impossibility to model them adequately in UML contributes to the infamous phenomenon of *domain semantics hidden in the application code*. We propose a light modification of the UML toolbox for modeling associations, which nevertheless allows one to manage the issue. The new notational constructs are marked in the Tables by "Not UML!" tag. Thus, question or exclamation marks in our Tables mean problems of forward engineering – when they are in the middle or right columns, or reverse engineering – when they are in the left column. Forward and reverse engineering of qualified associations is thoroughly discussed in section 5; we also show that n-ary associations can be reduced to qualified ones.

2 Background and Relation to Other Works

Semantics for the concepts of relationship and aggregation and their database implementation is a well-known research issue that can be traced back to the pioneering works on data semantics by Abrial, Brodie, Chen and others in the 70s and early 80s (see [15] for a survey). Object-oriented modeling flourished a bit later and focused on navigation across relations, mainly binary ones; the corresponding construct was called association and had been widely used in OMT and other practical OOAD techniques [20]. The most significant of these practices were later catalogued, abstracted and standardized by OMG in the MOF/UML/OCL group of standards. The most essential contribution was made by UML2, in which a large system of constructs related to associations was defined [19, sect.7.3.44]. In addition to the main functionality of navigating between the classes, it comprises a lot of concepts like multiplicities and types of collections at the ends, ends' qualification and ownership, redefinition, subsetting, and more. These concepts may capture important aspects of semantics of the business/domain to be modeled, and in that case must be reflected in the design model and then accurately implemented. Accuracy in modeling and implementation of associations may be a critical issue, e.g., in designing real time or embedded systems, where using models is becoming increasingly popular [13, 1]. Unfortunately, the standards provided neither a precise semantics nor even a consistent metamodel for the constructs [5, 6].

In this paper we will use notation and terminology standardized by UML2, and when we write "the Standard", we mean the UML 2.1.1 Superstructure [19]. In UML2, ontological aspects of associations are specified by a special attribute "aggregationKind" and denoted by either a white diamond (proper aggregation) or a black diamond (composition). A detailed discussion can be found in [2]. These aspects of associations are beyond the goals of the present paper.

It has been noted and argued in [22, 12] that actually two different notions of association, *static* and *dynamic*, are used in OOAD. The former is used for modeling structural relationships between classes, which are expressed by instance variables (attributes). The latter are channels over which messages can be sent; in UML2 they are specified in collaboration diagrams and do not influence the Association part of the metamodel. In this paper, we focus only on the static associations and their basic properties as they are defined in UML.

Semantics of associations in UML1.* and their usage had been discussed in [11]; a few works had focused on implementation in the context of forward [14] or reverse [16] engineering. These works became outdated after the acceptance of UML2, in which the technical aspects of the construct are essentially reworked. So far, the analysis of associations in UML2 has not gained much attention in the literature. The metamodel is analyzed in [5]. Paper [17] addresses the problem of heterogeneous collections at different association ends; our formal semantics allows us to build a transparent model of the issue and discover a useful implementation pattern that cannot be reverse engineered into UML. A recent paper [1] presents a detailed and careful discussion of how UML2 associations can be implemented in Java. However, they do not consider the extensional aspects of the concept, which we will show are crucial for its proper understanding and implementation. Particularly, we propose an entirely different semantics for qualified associations. We also propose a new semantics for unidirectional association with a multiplicity constraint at the opposite end, which is precisely aligned with the Standard.

3 Formal Semantics Framework

In this section we build our semantic framework. We begin with class diagrams and their semantics in terms of run-time instances. Then we abstract this description in mathematical terms and derive from it our formal framework for semantic interpretation.

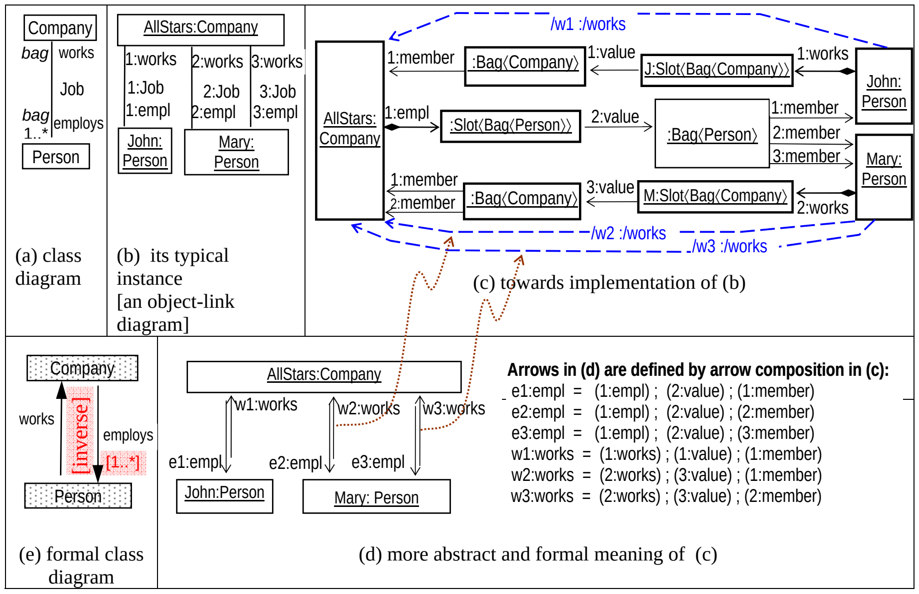
3.1 Class Diagrams, Informally

Figure 2(a) presents a simple class diagram with a bi-directional association between classes *Company* and *Person*. The diagram says that run-time *Person*-objects have references *works* to (collections of) *Company*-objects, and the latter have references *employs* to (collections of) the former. Adornments near the association edge ends specify these collections in more detail. They say that, if C is a *Company* object, then the collection $C.\mathit{employs}$ is a non-empty bag; and for a *Person*-object P, $P.\mathit{works}$ is an arbitrary bag.

Although collections are a technical issue, it is important for associations, and we need to consider briefly some details. For specifying types of collections, UML uses two Boolean attributes for association ends, *isUnique* and *isOrdered*.

They provide four possible combinations of Boolean values: 11,10,01,00, which specify the following types of collections: *ordered set, set, list* and *bag* respectively (cf. [19, p.128]). To ease readability, we will directly write *bag* or *set* in our diagrams. Since the default values for *isUnique* is True and for *isOrdered* is False, UML considers an association end to be set-valued by default while a bag-valued end needs an explicit declaration *isUnique*=False. In the formal semantics framework developed below, the situation is inverted: being a bag-valued is a default assumption while being set-valued is a constraint. Indeed, a set is a particular case of bag, not the other way round. In this sense, UML's convention is a special *concrete syntax* for specifying the constraint of being a set.

The situation with ordering is more complicated. In fact, declaring an association end to be list-valued means that we implicitly deal with a qualified association (discussed later in section 5). An ordered set can be treated as a list without repeated elements, i.e., satisfying the *constraint* of Uniqueness.



Arrows in (d) are defined by arrow composition in (c):

e1:empl = (1:empl) ; (2:value) ; (1:member)
e2:empl = (1:empl) ; (2:value) ; (2:member)
e3:empl = (1:empl) ; (2:value) ; (3:member)
w1:works = (1:works) ; (1:value) ; (1:member)
w2:works = (2:works) ; (3:value) ; (1:member)
w3:works = (2:works) ; (3:value) ; (2:member)

Fig. 2: Formal semantics for class diagrams: to be read consecutively (a)...(e)

3.2 Instances

A simple abstraction of a run-time configuration conforming to the model is presented in Fig. 2(b), where three objects are interrelated by three undirected edges or *links* (we write *empl* to save space). Objects are named "naturally" and typed by class names, ends of the links are named by numbers and typed by association end names. In the UML jargon, typing is often called *classifying* and classes and association ends are, respectively, *classifiers*; entities that are typed/classified are called *instances* of the respective classifiers.

In terms of information modeling, the diagram says that the company *AllStars* employs *John* and *Mary* with *Mary* counted twice (e.g., *Mary* can be employed at two different positions in the company). Correspondingly, *Mary* works for the company *AllStars* "two times". In a more implementation-oriented view, graph (b) shows objects holding references, but an object cannot hold multiple values in an instance variable slot. Rather, it holds a reference to a collection of values as shown by graph (c). For example, expression *AllStars*.(1:*empl*) refers to a memory cell, where a reference to a bag of *Person*-objects can be stored. In UML terms, *AllStars*.(1:*empl*) is a *slot*, whose contents is given by its *value*-attribute. Then expression *AllStars*.(1:*empl*).(2:*value*) denotes a bag of *Person*s. The latter can be implemented with Java generic collections [4]. Then setting a cursor (or Iterator, in Java terms) for this collection would list its three members with *Mary* counted twice. Similarly, attribute *works* of class *Person* is implemented by assigning slots of type $Bag\langle Company\rangle$ to *Person*-objects, for example, J and M in diagram (c). Slot assignment is done at compile time and does not change afterwards, and slots are not shared between objects, hence the black-diamond arrows. Value assignment is dynamic: a reasonable implementation is to consider that the same collection is changing its state.

Diagram (c) is built according to UML2 metamodel [19, Sect.7.2, Fig.7.8]. As is stressed by UML, it provides an abstract specification of the actual run-time configuration rather than its "mirror". For example, real addresses of slots depend on the order in which instance variables are defined inside classes (pointer arithmetic). Setting Iterator is also more complicated than it is shown in diagram (c). The latter thus presents a sufficiently abstract design model. However, for our further work it is useful to make it even more abstract and eliminate explicit presence of Collection objects in the model. To do this, we sequentially compose arrows in graph (c): (/w1:/*works*) = (1:*works*) ; (1:*value*) ; (1:*member*), (/w2:/*works*) = (2:*works*) ; (3:*value*) ; (1:*member*), and so on, where semicolon denotes composition and names of derived elements are prefixed by slash (UML's convention). That is, /w1,/w2,/w3 are derived directed links and /*works* is their common classifier – a virtual association end. What we really need for our further work is these derived links while the more detailed model and the way they are derived can be suppressed. The result is the graph in Fig. 2(d). The latter does not contain implementation details but it can be mapped to the implementation-oriented graph (c) augmented with derived elements: some elements of this map are shown in Fig. 2 by dotted "curly" lines (brown with a color display). We will term this and similar cases below by saying that model (d) is a *view* to model (c).

3.3 Formal Class Diagrams, I: Nodes, Arrows and Their Extension

Our next step is to collect the type/classifier names used in the instance graph (d) and organize them into a separate *type* graph as shown in Fig. 2(e) (ignore the label [**inverse**] for a moment). Nodes of this graph classify objects, and arrows classify directed links from the instance graph. Node rectangles are filled with dots to suggest that classes are populated with objects.

It is useful to invert our last passage from (d) to (e) and read it in the reverse direction from (e) to (d). We notice that each element E in graph (e) has an *extension* set $[\![E]\!]$: $[\![\mathit{Company}]\!] = \{\mathit{AllStars}\}$, $[\![\mathit{Person}]\!] = \{\mathit{Mary}, \mathit{John}\}$, $[\![\mathit{works}]\!] = \{\text{w1,w2,w3}\}$ and $[\![\mathit{employs}]\!] = \{\text{e1,e2,e3}\}$. Extensions of nodes are sets of objects, and extensions of arrows are sets of *labeled* directed pairs of objects. Labeling allows multiple occurrences of the same pair, i.e., extension of an arrow is a bag rather than a set of directed pairs. A collection of directed pairs is nothing but a (mathematical) *mapping* between the corresponding sets; more accurately, a *partially-defined multi-valued* mapping. We will designate such mappings with a black triangle head $f\colon X \twoheadrightarrow Y$, and keep the ordinary arrow head for a single-valued mapping $f\colon X \to Y$. By default, a multi-valued mappings is bag-valued because it corresponds to arbitrary instance graphs with multiple edges between the same nodes. Declaring a mapping to be set-valued is a constraint prohibiting duplication of edges in the instance graph. [2]

We will call graphs like (e) *formal class diagram* because their nodes correspond to classes and arrows to association ends, and at the same time have a formal meaning: nodes are sets and arrows are mappings. It can be specified by an *extension* (meta)map $[\![*]\!] : G \rightsquigarrow U$ with G the graph representing our formal diagram and U the graph specifying our semantic universe: its nodes are sets and arrows are partially-defined multi-valued mappings between them.

3.4 Formal Class Diagrams, II: Diagram Predicates

An important feature of the run-time instance graph (d) is that collections $[\![\mathit{employs}]\!]$ and $[\![\mathit{works}]\!]$ are isomorphic: they present the same set of object pairs traversed in the opposite directions. In this case we will say that the mappings are *co-extensional* or *inverse* (to each other). This feature is not a peculiarity of the particular instance (d) and must hold for *any* intended instance of the class diagram (e) as mappings *employs* and *works* present two opposite ends of the same association. Hence, we must add to our formal class diagram (e) a requirement that the arrows *employs* and *works* must have inverse extensions at any state of the system. In this case we call the arrows *inverse*.

Syntactically, we add to the graph (e) a formal expression, or *predicate declaration*, **[inv]**(*employs*, *works*), where **[inv]** is a predicate name (abbreviating 'inverse') and *employs*,*works* are arguments. The semantics for this declaration is that we consider to be legal only those extension maps $[\![*]\!]$ of graph (e), which make mappings $[\![\mathit{employs}]\!]$ and $[\![\mathit{works}]\!]$ inverse. To respect this constraint at any run-time moment, mutator methods (setters) must be synchronized and implemented with care, see [1] for details. Thus, the predicate declaration is an important part of the specification.

[2] In the Java Utility Package, what we call a mapping $X \twoheadrightarrow Y$ would be specified by a generic interface `Map<X,Collection<Y>>`, where objects of the class `X` are called *keys* and respective collections are their *values*. The values are accessed by the method `get(X key)` of the return type `Collection<Y>` [4]. Note that Java uses the term "mapping" for an individual pair $[x, f.get(x)]$ rather than for the set of all such pairs when x ranges over $[\![X]\!]$ like we do.

An important feature of predicates like **[inv]** is that their arguments must satisfy certain structural restrictions, e.g. for **[inv]**, the two argument arrows must go between the same nodes in the opposite directions. In the paper we will see other such predicates and call them *diagram predicates*. For example, association end's multiplicity is also a diagram predicate, whose arity shape consists of a single arrow. A bit more formally, we begin with a signature Σ of diagram predicates, each is assigned with its *arity shape* graph. Then we may form *Σ-graphs*, i.e., graphs in which some diagrams are marked with predicates symbols from Σ so that the shape of the marked diagram matches the arity shape of the predicate (see [7, Appendix] and [9] for details). If Σ is clear from the context, we call Σ-graph merely *dp-graphs* with "dp" standing for "diagram predicate".

It is mathematically proven that formal set theories can be interpreted in the language of dp-graphs and, hence, any specification possessing a formal semantics can be modeled by dp-graphs [3]. It has an important consequence that not only associations but many modeling constructs can be formalized in the same uniform way [8, 7].

4 Problems of Binary Associations

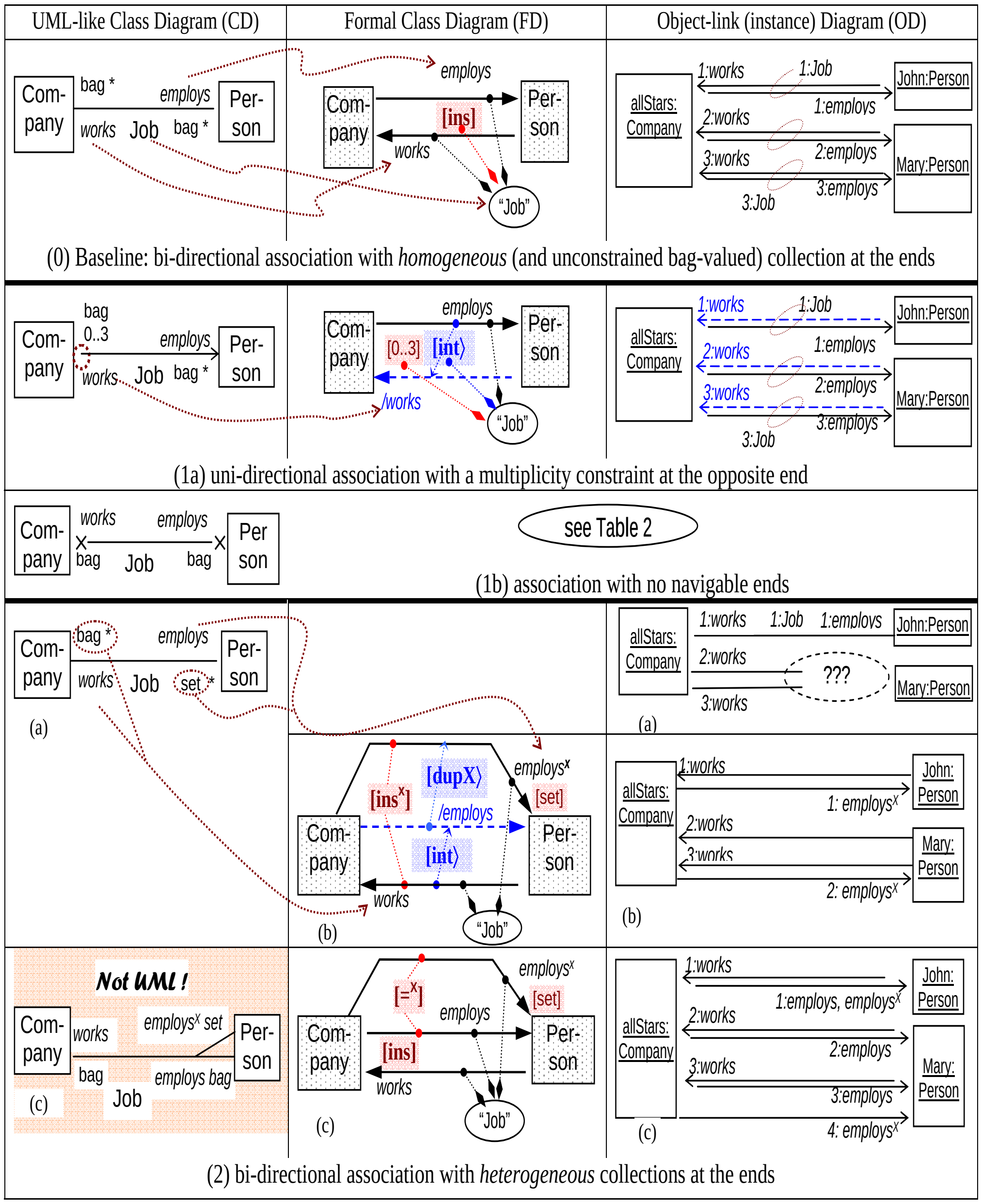
In this section we analyze semantics and propose implementation guidelines for the main use cases of binary associations. Results are presented in Table 1 for pure navigation and Table 2 for cases where extension is crucial. In both Tables, the left column presents typical UML class diagrams, the middle column shows their semantics in the framework of section 3 and the rightmost column presents typical instances of the diagrams. We will consecutively consider the rows in the tables.

4.1 Navigation: Is It That Simple? (Table 1)

Row 0: The baseline. The top row of Table 1 presents a very simple case: an unconstrained bi-directional association which we call the *baseline*. The class diagram shows the name Job of the association. Its counterpart in the formal diagram is an element "Job" framed with an oval. Formally, it is a name for the triple (*employs*, *works*, **inv**) and hence deletion of "Job" from the model implies deletion of its three components too. In UML jargon, "Job" is an object owning the elements of the triple and we use black-diamond ends of the corresponding meta-associations. Note that "Job" is not a classifier: its runtime "instances" are pairs of links, that is, a concept or a meta-construct rather than something really existing at run-time, see Fig. 2.

As discussed in the introduction, implementation of the baseline case requires accuracy to ensure synchronization of the ends' updates. It can be done in two different ways. One is to implement synchronized mutator methods for the attributes as suggested in [1]; care must be taken to avoid looping with mutual

Table 1: Back and forth between UML class diagrams and their semantics, I. (Color display would ease readability but a black-white version works too)


UML-like Class Diagram (CD)
Formal Class Diagram (FD)
Object-link (instance) Diagram (OD)
Com-pany
Per-son
bag *
employs
works
Job
"Job"
[ins]
allStars: Company
John:Person
Mary:Person
1:works
1:Job
1:employs
2:works
2:employs
3:works
3:employs
3:Job
(0) Baseline: bi-directional association with homogeneous (and unconstrained bag-valued) collection at the ends
bag 0..3
[0..3]
[int⟩
/works
(1a) uni-directional association with a multiplicity constraint at the opposite end
see Table 2
(1b) association with no navigable ends
set *
???
(a)
[insˣ]
[dupX⟩
employsˣ
[set]
/employs
(b)
1: employsˣ
2: employsˣ
Not UML !
employsˣ set
employs bag
[=ˣ]
1:employs, employsˣ
4: employsˣ
(c)
(2) bi-directional association with heterogeneous collections at the ends

Legend for Formal Diagrams. ***Classifiers*****:** Nodes are rectangles filled-in with dots. Arrows are a bit thicker than link-arrows. ***Diagram predicates*****:** Names are [,]-bracketed (shaded by dark red). The arguments are shown by thin dotted lines. ***Diagram operations*****:** Names are asymmetrically bracketed by [,⟩ (shaded by blue). The direction from the input to the output is shown by a dotted (blue) arrow. ***Tuples (Object-containers)*****:** Oval frame without filling and "quoted" names. The contents is shown by edges with black diamonds (composition); some of these edges are not shown to avoid clutter.

Cross references between UML and formal diagrams are shown with dotted "curly" (brown) lines. Elements in UML diagrams modifying these mappings are circled.

synchronization. Another way is to implement an object "Job", which will manage access and updates of the mappings via communication with classes. We will call this approach to implementation *objectifying the association.*

Row 1a: Unidirectional association with a multiplicity constraint at the opposite end. Unexpectedly, implementation of this case is problematic. Respecting the multiplicity at the *works*-end seems to require that the end should be implemented as an invisible attribute of class *Person*. It means that class *Person* knows about class *Company*, which destroys decoupling between classes – one of the main goals of having unidirectional associations. In [1, pp.21-23], the problem is discussed in detail but seemingly without a good solution.

Let us consider semantics of the case. The Standard [19, Sect.7.3.3,p.42] 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. This consideration is 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; accessing the latter requires querying and hence computation. Some types of queries can 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. The latter are immediately stored in the database and their access is always efficient. Thus, the distinction between navigable and non-navigable ends in UML is similar to the distinction between basic and derived data in SQL.

Formalization of this idea is shown in the middle diagram. By inverting the mapping *employs* (i.e., formally, by applying to it the operation **invert**), we come to a new derived mapping $/works = [\mathbf{inv}\rangle(employs)$ with $[\mathbf{inv}\rangle$ standing for $[\mathbf{invert}\rangle$. Note asymmetry in the brackets to distinguish operations from predicates. Since mapping */works* is derived from *employs*, the multiplicity constraint declared for it is, in fact, a constraint for the mapping *employs*.[3] Thus, what we need to do is to implement a specific constraint to mapping *employs*, which, in general, has nothing to do with attributes of class *Person*. The most immediate way to do this is to objectify the association by implementing an object "Job" that owns the mapping *employs*, computes its inverse and checks the multiplicity constraint. Class *Person* may know nothing about this, and the class coupling problem is thus resolved.

Row 1b: Both ends are non-navigable. This seems to be a meaningless idea (we are not aware of its discussion in the literature), yet the Standard explicitly allows such associations [19, Sect.7.3.3]. We analyze this case below in Sect. 4.2.

Row 2: Bi-directional association with heterogeneous collections at the ends. Suppose that one of the association ends, say, *employs*, is declared to be set-valued (*isUnique*=True in UML terms) while the opposite end is kept bag-valued, see UML diagram (a). It implies that in our UML-style instance

[3] As a simple analogy, consider the following situation. Let X be a set of natural numbers and $S(X)$ denotes the sum of all members of X. When we say that $S(X)$ must be less than 100, it is a constraint to X rather than to $S(X)$.

diagram (a) (the upper rightmost in the row), we need to glue together the duplicate ends 2:*employs* and 3:*employs*, but keep their opposite ends 2:*works* and 3:*works* separate. It cannot be done without destroying the structure of the graph (binary links cannot have three ends).

The problem generated a whole discussion in the UML community [18], which did not come to a certain conclusion. One solution is to consider associations with heterogenous collections at the ends illegal. However, it would prevent modeling many situations appearing in practice, see [17] for a detailed discussion. It appears that the problem is not in the heterogenous associations themselves but rather in an accurate formalization of their instance semantics.

A solution is again provided by considering operations and derived elements. The formal class diagram (b) shows three mappings: *works* and *employs* as before and, in addition, the duplicate eliminated version $\mathit{employs}^{\times}$ of mapping *employs* (the superscript reminds about duplicates crossed-out). This mapping is derived by applying operation **dupX** to *employs*. Now the inconsistent instance diagram (a) can be fixed as shown in diagram (b) below it. Mappings *works* and $\mathit{employs}^{\times}$ need not be co-extensional and the problem disappears.[4] We thus interpret the UML class diagram (a) as asserting that mapping $\mathit{employs}^{\times}$ rather than *employs* is to be implemented efficiently, and interpret the end *employs* by the formal mapping $\mathit{employs}^{\times}$. In other words, UML class diagram (a) amounts to a partial view to its fuller formal counterpart (b). This view is shown by "curly" dotted lines.

Note that in practical application we may need both mappings, *employs* and $\mathit{employs}^{\times}$, to be implemented efficiently. The corresponding formal diagram is shown in cell (c), where the predicate $=^{\times}$ declares mappings *employs* and $\mathit{employs}^{\times}$ to be equal up to duplicates. This situation (and code implementing it) cannot be modeled (reverse engineered) with UML. However, a slight addition to UML notation shown in the rightmost cell (c) fixes the problem.

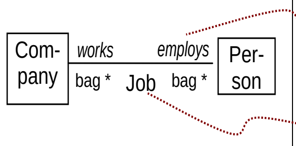
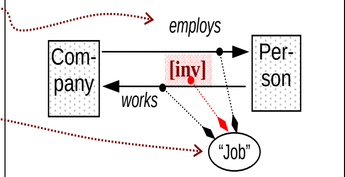
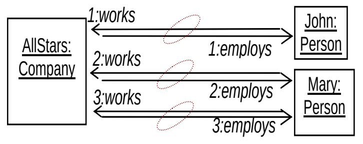
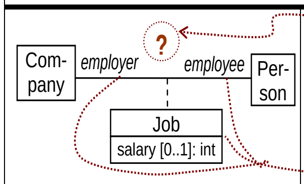
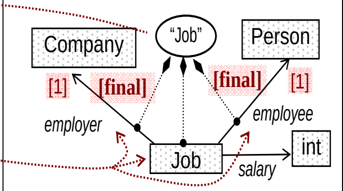
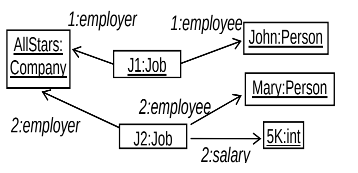
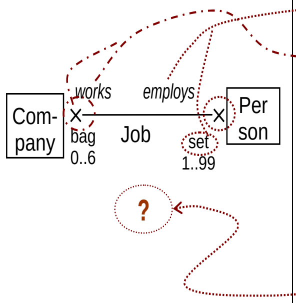
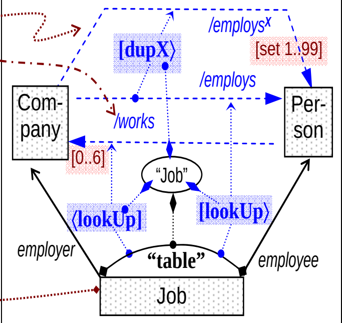
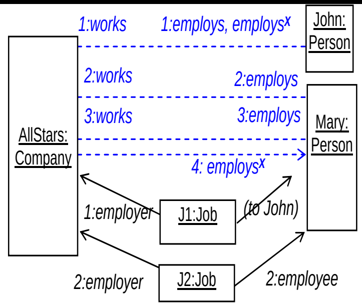
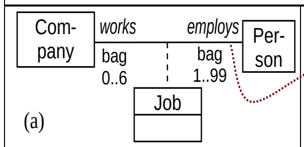
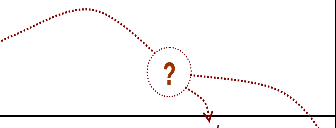
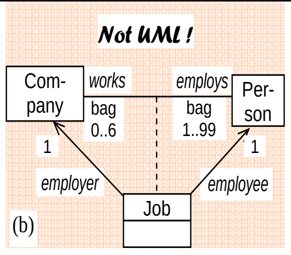
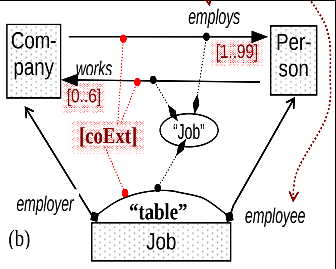
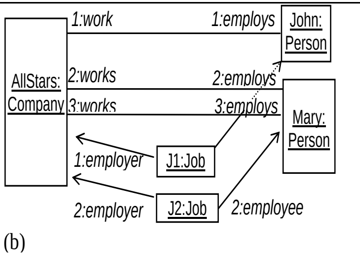
4.2 Navigation and Extension (Table 2).

Table 2 presents our study of cases that involve extensional aspects of associations. These aspects are especially important for n-ary associations with $n \geq 3$ [6]. However, for binary associations too, there are several semantic phenomena missed from the literature and mistakenly treated in the Standard. The top row of Table 2 repeats the baseline case to ease references.

Rows 1,2: Association classes and non-navigable associations. UML class diagram in Row (1) presents a major association construct called Association Class. The Standard says [19, Sect.7.3.4 p.49]: *An association may be refined to have its own set of features;[...]. The semantics of an association class is a combination of the semantics of an ordinary association and a class. [...]* the attributes of the association class and the ends of the association connected to the association class must all have distinct names. [...]. *It should be noted that in*

[4] Yet these mappings are still mutually inverse, hence label $\mathbf{inv}^{\times}$ in the formal diagram.

Table 2: Back and forth between UML class diagrams and their semantics, II

UML-like Class Diagram (CD)	Formal Diagram (FD)	Instance (object-link) Diagram (ID)
 Com-pany works employs Per-son bag * Job bag *		 1:works AllStars: Company 2:works 1:employs John: Person 3:works 2:employs Mary: Person 3:employs
(0) Baseline: bi-directional association with *homogeneous* (and unconstrained) collection at the ends		
 ? Com-pany employer employee Per-son Job salary [0..1]: int	 Company "Job" Person [1] [final] [final] [1] employer employee Job int salary	 1:employer 1:employee AllStars: Company John:Person J1:Job Mary:Person 2:employee 2:employer J2:Job 5K:int 2:salary
(1) association class (note how the ends change their meaning w.r.t. the row above!)		
 Com-pany works employs Per son bag 0..6 Job set 1..99 ?	 /employs^x [dupX⟩ [set 1..99] /employs Com-pany /works Per-son [0..6] "Job" ⟨lookUp] [lookUp⟩ employer "table" employee Job	 1:works 1:employs, employs^x John: Person 2:works 2:employs AllStars: Company 3:works 3:employs Mary: Person 4: employs^x 1:employer J1:Job (to John) 2:employer J2:Job 2:employee Undirected edges are abbreviations of pairs of directed edges like in Row (0)
(2) non-navigable association		
 Com-pany works employs Per-son bag 0..6 bag 1..99 Job (a)		
 Not UML ! Com-pany works employs Per-son bag 0..6 bag 1..99 1 1 employer employee Job (b)	 employs Com-pany Per-son [1..99] works [0..6] [coExt] "Job" employer "table" employee Job (b)	 1:work 1:employs John: Person AllStars: Company 2:works 2:employs 3:works 3:employs Mary: Person 1:employer J1:Job 2:employer J2:Job 2:employee (b)
(3) table and mappings together		

The legend is the same as in Table 1. The only new element is Semi-oval over class Job, which denotes the standard **"table"** container *an instance of an association class, there is only one instance of the associated classifiers at each end, i.e., from the instance viewpoint, the multiplicity of the association ends are '1'* For our case in Row 1, the italicized sentence in the quote says that the ends *employer* and *employee* are to be like attributes of class *Job*, which explains why duplication of names is prohibited. The next sentence says that a *Job*-instance is, in fact, a pair of instances, one from class *Company* and the other from class *Person*. Combination of these two requirements provides the formal diagram in the middle cell: class *Job* has two single-valued references *employer* and *employee* to classes *Company* and *Person*.

An important feature of the case is that defining attributes of the association class are immutable: if we have initialized object J:*Job* with values J.*employer* = $AllStars$, J.*employee* = $Mary$ and J.*salary* = $50K$, then only the value of *salary* can be changed later; changing J.*employee* from $Mary$ to, say, $John$ is impossible because it would change the link and hence the very object! Hence, we add to our formal diagram two more diagram predicate declarations **final**(*employer*) and **final**(*employee*) asserting immutability of the mappings.What we finally specified is a table *Job* with three columns (called also *projection mappings*), amongst which the pair (*employer*, *employee*) is considered as an immutable identifier. If an additional condition of disallowing duplicate values for pairs (J.*employer*, J.*employee*) holds, then the pair (*employer*, *employee*) would be exactly what is called the *primary key* to relation in the database theory.[5] Formally, we call a triple $T = (\mathit{Job}, \mathit{employer}, \mathit{employee})$ an *(association) table* if mappings *employer* and *employee* are totally defined, single-valued and immutable. From now on, we will designate such tables in our diagrams by a semi-oval with label "**table**".

Comparing Rows 0 and 1 shows that declaring an association to be a class changes the meaning of association ends. Making an association a class (in UML terms, *reifying* it) actually reifies its extension as a table and makes association ends projection mappings (columns) of this table rather than navigation mappings. However, interpreting association ends by projections does not mean that the old navigational mappings entirely vanished. By looking up the extension table in the two opposite directions, we can reconstruct the old mappings as it is shown by formal diagram in Row 2. Two labels **[lookUp⟩** denote two applications of the operation **[lookUp⟩** to the table, which produce mappings /*works* and /*employs*/ (we remind that slash-prefix denotes derived elements).Thus, our old association ends have reappeared but now as *derived* rather than *basic* elements. By our discussion of quote (Q1) p.11, it means that the old mappings are non-navigable as shown by UML diagram in Row 2. Now the multiplicity and collection-type constraints for non-navigable ends can be readily explained. For example, if a non-navigable end is declared to be set-valued (see UML diagram in Row 2), it means that we need to augment our formal diagram with one more derived element: the duplicate-eliminated version $\mathit{employs}^{\times}$ of the mapping *employs*. Thus, the UML diagram in Row 2 specifies a few constraints to a table, but the very table is not anyhow presented in the diagram! Besides conceptual

[5] In the earlier UML versions, this condition had indeed been assumed.

ambiguity, this notation does not define the names for the columns of the table and thus hinders cross-references between design and code.

Row 3: Navigation and extension together. The UML diagram (a) entirely conforms to the UML metamodel because any association can be declared to be a class. However, reifying an association as a class converts association ends into projection mappings (see Row 1) while having multiplicity constraint for the ends forces to interpret them navigationally. Semantics of the situation is clear and shown in formal and instance diagrams (b). However, mapping of the UML diagram (a) into the formal diagram (b) is ambiguous because we have two names for four mapppings.

On the other hand, it is an ordinary situation when both the extensional and the navigational components of association are required to be implemented efficiently, hence, be basic elements as specified by the formal diagram. We have a lot of redundancy in data representation, and the label **coExt** asserts that the following three sets must be equal: $\{(p, c) \mid p \in \mathit{Person}, c \in p.\mathit{works}\}$, $\{(p, c) \mid c \in \mathit{Company}, p \in c.\mathit{employs}\}$ and $\{(j.\mathit{employee}, j.\mathit{employer}) \mid j \in \mathit{Job}\}$. In other words, *Job* is the common graph of mappings *works* and *employs*. Though such situations are quite possible in practical applications and hence may be hidden in the application code, their reverse engineering into UML is problematic. To fix the gap, we propose the notational construct shown in row (3b) left. More generally, Figure 3 presents a modification of UML notation to manage the navigation-vs-extension issue.

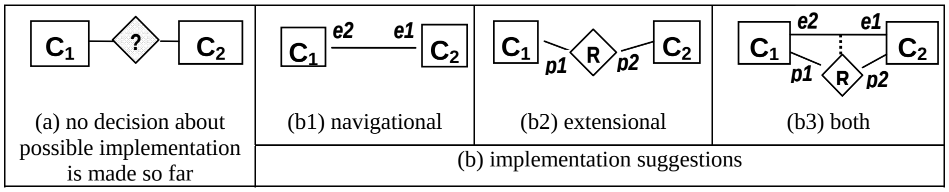

C1
?
C2
(a) no decision about possible implementation is made so far
C1
e2
e1
C2
(b1) navigational
C1
p1
R
p2
C2
(b2) extensional
C1
e2
e1
C2
p1
R
p2
(b3) both
(b) implementation suggestions

Fig. 3: Notational proposal

5 Qualified and N-ary Associations

Qualified associations are considered to be one of the most controversial constructs in the UML associations "package" [1]. Even simple unidirectional cases are rarely implemented in MDD-tools, let alone the bidirectional ones. The cause of the problems is that semantics of qualified associations is often misunderstood and their metamodel is essentially flawed [5, 6]. The latter is due mainly to misunderstanding that a qualified association is merely a particular traversal of the corresponding ternary association and conversely, any ternary association determines a collection of mutually inverse qualified associations. The example shown in Fig. 1(c1,c2,d) is quite generic in this sense. Precise formal definitions (including the general case of n-ary association and its qualified counterparts with

($n-2$)-qualifiers) and the metamodel can be found in [5]. In the present paper we are interested in semantics and design patterns rather than in metamodeling.

The Standard distinguishes two cases of using the construct: the *general* and the *common* [19, Sect.7.3.44,p.129], which we will consecutively consider.

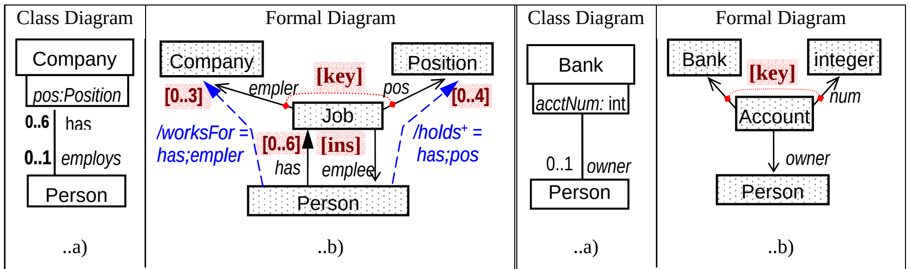
5.1 The general case. A typical situation is shown in Fig. 1(c2), which says that a person at a given company can hold not more than two positions. That is, class *Person* has a getter method *holds* with a parameter *c* of type *Company*. What makes the case general (rather than *common* to be considered below) is that invocation $p.holds(c)$ for an object $p{:}Person$ returns a collection rather than a single *Position*-object. We can present the case as a binary multi-valued mapping $holds^{*}: Person \times Company \twoheadrightarrow Position$. The passages from $holds^{*}$ to *holds* and conversely are well known in type theory and functional programming by names of *Currying* and *unCurrying* respectively. In its turn, the extension (graph) of mapping $holds^{*}$ is a ternary relation over the participating classes. By choosing suitable names for the roles of this relation, we come to diagram (d) in Fig. 1. Note that according to Standard [19, p.42], multiplicities at the ends are, by definition, exactly those specified in qualified association diagrams Fig. 1(c1),(c2).

The ternary relation in Fig. 1(d) can be traversed in six different ways grouped in three pairs. For example, methods $holds{:}Person \rightarrow [Company \twoheadrightarrow Position]$ at class *Person* Fig. 1(c2) and its counterpart $holds'{:}Company \rightarrow [Person \twoheadrightarrow Position]$ at class *Company*, give one such pair of traversals. Methods $employs{:}\ Company \rightarrow [Position \twoheadrightarrow Person]$ Fig. 1(c1) and its counterpart $employs'{:}\ Position \rightarrow [Company \twoheadrightarrow Person]$ give another pair. With JavaGenerics, implementation of these qualified mappings is not more complicated than in cases considered above. Our analysis of binary associations above can be immediately generalized for multi-ary ($n \geq 3$) associations with qualified instead of ordinary ends.

5.2 The *common* case. This is the case when the multiplicity at the target end is 0..1 like in diagram (c1) in Fig. 1; it is repeated below in (1a) Fig. 4. If we collect all those pairs ($c{:}Company, p{:}Position$), for which $c.employs(p)$ returns a single *Person*-value, into a set $X \subset Company \times Position$, then we will have a *totally defined single-valued* mapping $^{*}employs^{*}: X \rightarrow Person$. Usually it means that the set X has a certain semantic significance and it makes sense to model it as a special new class. In our situation, such pairs can be considered as jobs, and we come to a formal class diagram (1b) in Fig. 4 (ignore the dashed derived arrows for a while). Predicate **key** in this diagram states that *Job*-objects are uniquely determined by pairs of values ($J.empler, J.pos$) ([7] provides formal details). Note that association end *employs* specified in the class diagram (1a) is nothing but a Curried version of mapping *emplee* in diagram (1b).[6]

Introducing a new class into the model for the common case of qualified associations makes sense for many practical situations where semantics is to be explicated; diagrams (2a,b) in Fig. 4(2) present one more typical example. In

[6] The multiplicity changes from [1] to [0..1] because not any pair (c, p) determines a job.



(1.. Refinement of association Job from Tables 1,2 (names *empler, emplee* abbreviate *employer, employee* resp.)

(2.. Example from the Standard [23, sect.7.3.44]

Fig. 4: The *common* case of using qualified associations. Non-shown multiplicities are exactly [1] by default.

fact, the issue is well-known in database design: in the relational language, a qualified association with multiplicity 1 is nothing but a so called *functional dependency*: $(empler, pos) \rightarrow emplee$ for the relation *Job*. Then remodeling diagram (1a) into diagram (1b) appears as a typical case of normalizing relational schemas according to functional dependencies. Thus, the common case of qualified associations actually encourages to model associations in a non-normalized way and to hide a semantically important class. It is not necessarily a “bad’ design but the modeler should be aware of possible problems and recognize that diagram (1a) is only a partial view of semantics (1b).

5.3 Precise modeling with diagram predicates and operations. Formal diagrams like (1b,2b) in Fig. 4 not only accurately specify the mappings to which multiplicities in class diagram (1a,2a) refer, but allow specifying other important details of the situation. For example, to specify multiplicity of the end *worksFor* from diagram (a) in Fig. 1, we compose mappings *has* and *empler* and obtain a derived mapping $/worksFor(p.worksFor \overset{\text{def}}{=} \{J.empler \,|\, J \in p.has\}$ for a *Person*-object p), for which the required multiplicity can be declared. In addition, by inverting mapping */worksFor* we can augment our formal diagram with another derived mapping $/employs\colon Company \twoheadrightarrow Person$, and then map the entire diagram Fig. 1(a) to the augmented formal diagram. In this way ***all*** class diagrams in Fig. 1 can be presented as views to the formal diagram Fig. 4(1b) suitably augmented with derived elements (including Curried versions of binary mappings *emplee* and *pos*). This observation is crucial if we need to merge a few class diagrams without redundancy. None of the major MDD-tools addresses the issue.

Moreover, the same formal diagram can be used for specifying requirements beyond the class diagrams in Fig. 1. Suppose that despite the possibility of working for up to three companies, and of holding up to two positions at a company, a person is not allowed to have more than four positions in total. This multiplicity cannot be declared in either of diagrams in Fig. 1 yet it can be easily done with our formal diagram. To wit, by composing mappings *has* and *pos* we derive mapping $/holds^{+}$ and declare it to be [0..4]-valued. Thus, our formal diagrams appear as a very flexible means of modeling associations and precise specifying their semantics as well.

6 Discussion and Conclusions

Ownership, Reification and Objectification. Cases presented in Tables 1,2 show that an ordinary binary association comprises six mappings: two projections (see p.14) and two navigable versions of each of the ends (bag-valued and set-valued). The UML2 metamodel calls mappings *properties*, and says that a binary association has two properties called its *ends*. Thus, the metamodel provides two names for six objects and hence a controversy is inevitable. For qualified associations, the situation is even worse.

An important aspect of this controversy is related to the infamous issue of association end *ownership*. In the earlier versions of the Standard, including UML 2.0, navigable ends have been considered to be owned by the classes implementing them while non-navigable ends were owned by the very association. This treatment confuses ownership with interpretation of association ends (either by navigation mappings between the classes or by projection mappings of the extension table). This confusion has not been identified in the literature and the issue has been repeatedly debated in the community. The latest version of the Standard, UML 2.1.1, has changed its formulation once again. Now ownership and navigability are declared to be entirely orthogonal concepts [19, sect. 7.3.3]. Because the extension table of an association is still neither specified in the metamodel nor explicated in the semantics sections of the Standard, the new treatment did not clarify the issue. Rather, it made it even worse by increasing the number of possible yet meaningless combinations of ownership and navigability.

The problem disappears in our semantic framework. Association ends are always owned by the association itself: if it is deleted from the model, the ends are also deleted even though they are implemented as members of the corresponding classes. In our formal diagrams, this is denoted by black-diamond meta-associations coming from the oval "Job"; the latter can be thought of as an object containing the ends. If the association is reified by the class *Job*, then the container object "Job" will contain the class *Job*! (see formal diagram in Table 2 Row 1). A part of UML's controversy around association is caused by confusing these two distinct concepts: the class *Job reifying* the association and the object "Job" *objectifying* it.

OO Programming vs. OO Modeling. Another side of the problem is a conceptual mismatch between OO modeling languages and OO programming languages. The former are usually diagrammatic and this is not just a syntactic sugar. Rather, the diagrammatic syntax of modeling languages follows their *diagrammatic logic* in the sense that a basic modeling unit, e.g., an association, is often an integral systems of modeling elements, which consists of several nodes and edges; details and formal definitions of diagrammatic logic constructs can be found in [8, 7]. Implementation of diagrammatic modeling constructs in a OO framework requires their distribution over distinct classes. This causes synchronization problems, and the ownership controversy. We propose the following uniform and universal way of managing the issue. Irrespectively of the type of association, it can always be implemented by a special object, which (i) keeps track of all the components residing in distinct classes, (ii) manages their access and updates by communicating with the classes hosting (but not owning!) the components, and (iii) ensures synchronization and consistency w.r.t. the constraints declared in the model.

More technically, we propose to implement a generic (meta)class Association$\langle n \rangle$ with n the arity parameter. The members of this class are (i) mappings between the participating classes, (ii) the common extension table of these mappings (which is a class whose instance variables are projection mappings) and (iii) methods executing (diagram) operations discussed in the paper. Particularly, Association$\langle n \rangle$'s interface must include methods for Currying and unCurrying, projecting n-ary association to its m-ary, $m < n$, components and checking and maintaining the constraints. Implementation of this class as an Eclipse plug-in is planned for a future work.

Summary. Associations between classes are a major modeling concept in OOAD. Their essential feature is integrity: an association comprises a system of interrelated modeling elements. In contrast, implementation of associations in modern OO languages requires their elements to be distributed over distinct classes, which breaks the system into pieces. The integrity must then be recovered by implementing special synchronization means, which complicates the code. Implementation becomes even more intricate because of the interplay between basic (navigable) and derived (non-navigable) elements of associations.

A necessary prerequisite for addressing the problem is to have a clear semantic picture of what associations and their properties are. We have proposed a graph-based yet formal framework where these semantics can be built, and shown how naturally UML diagrams can be mapped into formal diagrams. This is the main contribution of the paper. It allowed us to explain semantics and implementation of a few controversial association constructs, e.g., unidirectional associations with multiplicities at the both ends, association without navigable ends, qualified associations. In addition, we have identified a number of patterns for using associations in practical applications, for which reverse engineering into UML is problematic. We have also suggested a universal pattern for implementing associations via their *objectification* and sketched the interface of the corresponding metaclass.

Acknowledgement. Thanks go to Bran Selic for many stimulating discussions of associations in general and their specification in UML2 in particular. Financial support was provided by Bell Canada through the Bell University Labs, NSERC, and the Ontario Centres of Excellence.

References

1. D. Akehurst, G. Howells, and K. Mcdonald-Maier. Implementing associations: UML 2.0 to Java 5. *Software and Systems Modeling*, 6:3–35, 2007.
2. F. Barbier, B. Henderson-Sellers, A.Le Parc, and J.Bruel. Formalization of the whole-part relationship in UML. *IEEE Trans. Software Eng.*, 29(5):459–470, 2003.
3. M. Barr and C. Wells. *Category theory for computing science*. PrenticeHall, 1995.
4. G. Bracha. *Generics in Java programming language*. Sun, `http://java.sun.com/j2se/1.5`, 2004.
5. Z. Diskin and J. Dingel. Mappings, maps and tables: Towards formal semantics for associations in UML2. In *ACM/IEEE 9th Int. Conf. Model Driven Engineering Languages and Systems*, LNCS, vol.4199. Springer, 2006.
6. Z. Diskin and J. Dingel. Mappings, maps, atlases and tables: A formal semantics for associations in UML2. Technical Report CSRG-566, University of Toronto, 2007. `ftp://ftp.cs.toronto.edu/pub/reports/csri/566`.
7. 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.
8. Z. Diskin, B. Kadish, F. Piessens, and M. Johnson. Universal arrow foundations for visual modeling. In *1st Int. Conf. on the Theory and Applications of Diagrams*, LNCS, vol.1889. Springer, 2000.
9. Z. Diskin and U. Wolter. A diagrammatic logic for object-oriented visual modeling. *ENTCS*, 2008. To appear.
10. J. Erickson and K.Siau. Theoretical and practical complexity of modeling methods. *Communications of the ACM*, 50:46–51, 2007.
11. R. France. A problem-orineted analysis of basic UML static modeling concepts. In *ACM/SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 1999.
12. G. Génova, J. Llorens, and J. Fuentes. UML associations: A structural and contextual view. *J.of Object Technology*, 3(7), 2004.
13. S. Graf, Ø.Haugen, I. Ober, and B. Selic. Specification and validation of real time systems in UML. *J.on Software Tools for Technology Transfer*, 8(2), 2006.
14. Y. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *ACM/SIGPLAN Conf.on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2004.
15. R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.
16. D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. *IEEE Trans. Software Eng.*, 27(2):156–169, 2001.
17. D. Milicev. On the semantics of associations and association ends in uml. *IEEE Trans. Software Eng.*, 33(4):238–251, 2007.
18. OMG, `uml2-superstructure-ftp@omg.org`. *E-Conference on UML2 Superstructure. Issue #5977*, 2003.
19. OMG, `http://www.omg.org/docs/formal`. *Unified Modeling Language: Superstructure. Version 2.1.1 Formal/2007-02-03*, 2007.
20. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design*. Prentice-Hall, 1991.
21. B. Selic. Model-driven development: Its essence and opportunities. In *9th IEEE Int.Symposium on Object-Oriented Real-Time Distributed Computing*, 2006.
22. P. Stevens. On the interpretation of binary associations in the unified modeling language. *Software and Systems Modeling*, 1(1), 2002.