On the interpretation of binary associations in the Unified Modelling Language

Perdita Stevens*

Division of Informatics
University of Edinburgh

Abstract. Binary associations between classifiers are among the most fundamental of UML concepts. However, there is considerable room for disagreement concerning what an association is, semantically; it turns out that at least two different notions are called Associations. This confusion of concepts gives rise to unnecessary complication and ambiguity in the language, which have implications for the modeller because they can result in serious misunderstandings of static structure diagrams; similarly, they have implications for tool developers. In this paper we explore these issues, suggest improvements and clarifications, and demonstrate side-benefits that would accrue.

Keywords: UML, association, link

1 Introduction

The Unified Modelling Language has been widely adopted as a standard language for modelling the design of (software) systems. Nevertheless, certain aspects of UML are not yet defined precisely. This paper is concerned with one such aspect: associations.

The paper is structured as follows. The remainder of this section and the next give background information and introduce the main problems we address.

* Supported by an EPSRC Advanced Research Fellowship.

Email: Perdita.Stevens@dcs.ed.ac.uk. Fax: +44 131 667 7209

In Section 3, the main part of the paper, we carefully discuss the definition of Association in the latest version of UML and explore the implications of the ambiguities and contradictions it includes and of the different possible resolutions. Section 4 discusses interactions between associations and generalisation; Section 5 discusses association classes. Section 6 mentions some related work, and finally Section 7 concludes and summarises the implications for the UML standard.

This paper is based on UML1.4 [12], the latest version at the time of writing. It is hoped that this work may feed into the ongoing construction of UML2.0, one aim of which is to increase the precision of UML. A major aim of the paper is to be readable by anyone who can read the UML standard. At the end of the paper we will briefly discuss work that gives formal semantics for UML associations, but this paper will not add to their number: rather, we are concerned with making UML *formalisable*. A main purpose of formal semantics work with UML is often to find the ambiguities and contradictions in the informal specification, and such approaches can be very effective at fulfilling that aim. However, there is a serious drawback in that the formalisations are then not readable except by those familiar with the formalism used; so the vital process of gaining consensus for a modification is hindered. Perhaps more damagingly, the process of formalising UML in one's favourite formalism often suggests assumptions natural in that formalism, the following of which may obscure the very ambiguities we seek. In this paper, we take the view that once the meanings of UML's concepts are informally clear, the way to (perhaps various) formalisations will be clearer. This is our reason – some will say excuse – for not giving here such fundamentals as a programming language independent semantics for attributes of classes, which would be required to cover this material in a fully formal way.

Our concern with precision does, however, lead us naturally to consider the relationship between a UML model and the class of *systems that correctly implement it*, where to define what we mean by "correctly" is to define the relationship. This can be seen as an informally-described (denotational) semantics of UML: a UML model *denotes* the collection of systems which are deemed to implement

the model, and that collection defines the meaning of the UML model. If a community of people always agree on any question of the form "does this system implement this UML model?", they can be said to have agreed on a semantics for UML, albeit an informally defined one. Indeed, for designers and programmers, this is the most intuitively manageable form of UML semantics: disagreements are generally easy to concretise into a disagreement about some particular question of the given form, which can then be discussed. (Unfortunately, this form of semantics is hard to formalise and to use in formal ways, but as we have argued, this need not rule it out for informal use.) Readers with a business analysis background may find this mode of discussion initially less natural, but it is in fact pertinent to any discussion involving a model which will be implemented as a computer-based system[1]: to look at it another way, the model, whilst it does not completely determine the computer-based system, should enable us to be confident about *which systems the modeller intended to rule out.*

The reader is naturally assumed to be familiar with UML. We cite material from [12] by page; thus, [12] (3-90) indicates page 90 of section 3 of the UML standard, that is, of the Notation Guide.

We will use the usual convention of writing "an A" for "an instance of classifier A" etc., where no confusion results. We write $a : A$ etc. for "a is an instance of classifier A" etc..

A shorter, preliminary report of this work appeared in the Proceedings of UML2001 [17].

2 Background

Relationships between classes have long been considered; for example, relationships were a core concept of Entity-Relationship diagrams, and associations have been a part of each of the main ancestors of UML. However, it does not seem to have been widely realised that core problems remain in the interpretation

[1] A "computer-based", as opposed to a "computer", system may include the humans who interact with the computers and software, and their processes.

of binary associations in UML. We choose to focus on binary associations (the most common kind, associations that relate just two classifiers) because problems specific to associations relating three or more classifiers, multiplicity in particular, have received more attention [8, 6], and it seems to be widely assumed that binary associations present no interesting problems.

Why has the topic of making associations precise not attracted more attention in the past? One reason is that for purposes of informal design, an ambiguity about exactly which systems correctly implement which UML models – which is at the heart of this issue – is not fatal. Association information provided by a designer in a model can be treated by a programmer (who may or may not be the same person) as guidance, and may be interpreted in different ways for different associations even within the same model.

For other communities this may not suffice. In the reverse engineering community, for example, tools are required to produce UML models from existing code, as an aid to understanding legacy systems. If the tools are to be maximally useful it must be clear to the user what the presence or absence of an association means. Recently there have been several serious efforts to enable interoperability of reengineering tools; representatives of the major ones came together at a Dagstuhl workshop earlier this year, and detailed discussions of these issues took place. Careful consideration was paid to using UML as an interoperability metamodel[2]. However, to the disappointment, but with the eventual agreement, of the author it was concluded that this was impossible. The problems addressed in this paper were part of the reason, so those discussions motivate the paper.

2.1 Purposes of Association

Before proceeding to examples, we should collect together the purposes of associations, against which the concept in the language must be judged. They will overlap (several purposes may be served by the recording of one association),

[2] or as part of it, for recording the recovered design information; information about the code structure is also vital in such applications but is outside the scope of UML

and as we shall see they may (in a sense to be made precise) conflict. Not all modellers will use associations for all these purposes, of course.

1. We may wish to record *positive* decisions about the design, that is, any one of various forms of dependency between classifiers:
   (a) We may wish to record a conceptual relationship between two classes: that is, we may know that there is some connection between the concepts represented by the two classes at a stage before we have any clear understanding of how that connection will be manifested in the designed system.
   (b) We may wish to record that we expect the implementation of one class to contain a reference to an instance of another; this may sometimes apply even when we have not decided which of the pair is to contain the reference or whether each class must refer to the other.
   (c) We may wish to record that an instance of one class will need to invoke the services of an instance of another. Again, there may sometimes be a stage when we do not yet know in which direction services are invoked.
   (d) We may wish to write a constraint which jointly constrains the members of both classes; for example, a class invariant for one of the classes may need to mention the related object(s) of the other class. In this case, the presence of an association describes the ability to *navigate* from one object to another within a constraint or other form of discourse about the system. This normally occurs together with the intention that there should be some more concrete connection between the objects (a reference in one of them to the other, or the ability for one of them to send a message to the other), but this is not conceptually inevitable. We consider it (at least) bad style for the class invariant of $o : O$ to mention an object $p : P$ which will not in the implemented system be accessible from o, partly because such an invariant is hard to check at runtime and partly because it imposes a dependency of O on P which might not be obvious from the implemented system, and which might therefore damage maintainability.

2. We may wish to use the diagrams – which can be seen as a succinct description of the positive design decisions – to deduce negative information, namely the absence of certain kinds of dependency. For example, suppose that during maintenance we need to alter class A. We may expect the class diagram to be useful to us in determining which other classes we should examine to see whether they need to be altered as a consequence of A. Naturally, we will examine those classes which have a recorded dependency of any kind on A, but for the diagram to be useful we have to go further: we need to be confident that if such a dependency is not recorded, it does not exist. Specifically, we would like the absence of an association between classes A and B to tell us something useful about the absence of (certain kinds of) dependency between those classes.

Each of these purposes records some kind of dependency between the two classes. We use the term "dependency" here in the traditionally broad software engineering sense: A is dependent on B if a change to the implementation of B may force a change in the implementation of A; a fortiori, the removal of B may cause A not to work, so that we should not expect to be able to reuse A without B (until we have substituted something appropriate for B, or altered A). Similarly, in UML, a Dependency is a very general kind of relationship, and if we deem that a particular relationship between the classes is not appropriate to be represented by an Association, we will almost inevitably have to fall back on using a Dependency instead. Since there is no concept in UML of "an instance of a dependency", whereas there is the concept of a Link being an instance of an Association, our first, imprecise, attempt at distinguishing the concepts is that Association should be used when there is some "per instance" character to the relationship, and a Dependency otherwise (always assuming in the latter case that there is no appropriate more specific relationship, such as Generalization or Realization). That is, if for a given relationship R between classifiers A and B and instances a and b of those classifiers it makes sense to ask "does the relationship R relate instances a and b?" then we are looking at a relationship

with a "per instance" character, for which an Association may on the face of it be an appropriate model element.

2.2 Examples

More concretely, to introduce the problem, we present some questions which illustrate the legitimate disagreements that can arise as a result of readers of the current UML standard interpreting it differently. Readers are quite likely to have their own strongly preferred answers to these questions, and indeed, some of them are unequivocally answered by the UML specification. Others do lead, as we shall show, to exposing flat contradictions in the current UML specification. However, the main purpose of giving the examples is to demonstrate how easily modeller's expectations may be frustrated. Such arguments are dangerous of course in their informality, but vital where humans are using a large language like UML whose definition they inevitably do not carry in full detail in their heads. Therefore we will not be ashamed to talk about developers expecting something which a perfect reasoner from the UML specification would not expect.

For clarity we present trivial versions of the problems, but it is clear that non-trivial versions do arise in practice, and in particular, that a developer of certain tools would have to make decisions about the answers.

Consider first fragment of Java shown in Fig. 1. Of which class models is this a correct implementation? Specifically, what associations may or must appear between classes A, B and C? Everyone will agree that there must be an association between A and C, because C is declared to hold a reference to an A and there is a method of C which uses this reference to send a message to an A. Most people will argue that there must be an association between B and C because C is declared to hold a reference to a B. Some, however, will argue that there must not be, because no message ever passes between a B and a C. The opposite situation holds between A and B: most will say that there is no association, because the static structure of neither declares a reference to an instance of the

```
class A {
  public void doA(B b) {b.doB();}
}

class B {
  public void doB() {;}
}

class C {
  private A myA;
  private B myB;
  public void doC() {myA.doA(myB);}
}
```

Fig. 1. Java code

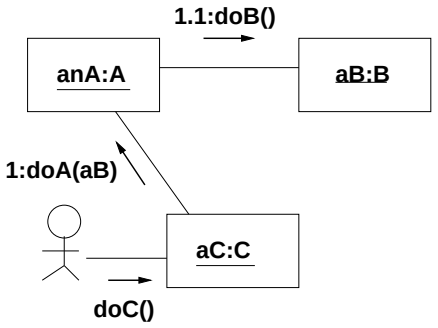

Fig. 2. A collaboration including an implicit ≪parameter≫ link

other; however, as instances of A may send messages to instances of B, it may also be argued that this should be recorded in an association.

Now let us go on to consider collaborations. Most people would expect C's reaction to message `doC()` to be shown in an (instance level) collaboration diagram like Fig. 2. That is, even those who do not expect an association between A and B certainly will expect a link between instances of those classifiers: the alternative is to show a stimulus with no link.

This pair of expectations contradicts another universal expectation, that any link is an instance of an association.

Lastly consider Fig. 3. In my experience most people will happily accept the class diagram as representing graphs, including the graph illustrated. They will draw the object diagram shown – or possibly, a variant with two links of `is incident on` both linking $e2$ to $n2$ – to represent the graph. In UML1.4, however, neither variant legally conforms to the class diagram. We will consider this example in Section 3.3.

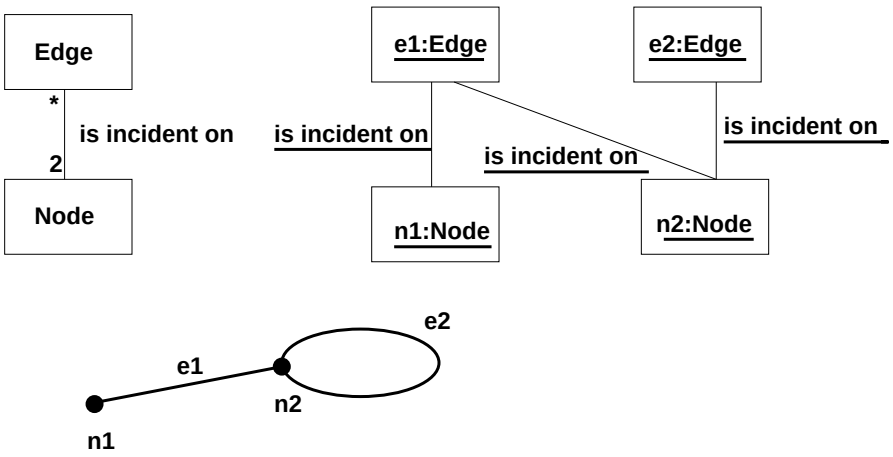

Fig. 3. An incorrect way to use UML1.4 multiplicities

3 Associations in the UML1.4 standard

In this section we address several issues that arise when we consider UML1.4's treatment of associations in detail.

3.1 Dynamic or static?

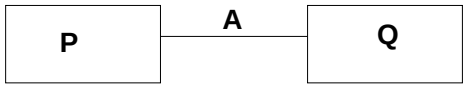

Fig. 4. An association

Suppose objects $p : P$ and $q : Q$ are linked within association A which associates classifier P with Q. There are two main ways of interpreting this. We refer to the first notion (labelled S) as *static association* and to the second (labelled D) as *dynamic association*. Because it is important to consider the consequences for both the association and the link, we give the alternatives

from both points of view. First, the fact that there is an association A between classifiers P and Q may be informally defined in either of the following ways:

S There is a structural relationship between the classifiers: for example, they should eventually be implemented by class definitions in which one of P, Q will include an attribute which is expected to contain a reference to an object of the other class. (We say "expected" because it is legitimate to use UML to model systems which are implemented in untyped languages, in which the expectation could not be verified from the code.)

D There is a behavioural relationship between the classifiers: that is, the system should be implemented such that it may happen (or perhaps, it is expected to happen: see below) that instances of classifiers P, Q exchange a message.

From the point of view of instances, these views become:

S One of these objects contains data which references the other: for example, p has an attribute whose value *at the moment* is (a reference to) the object q;

D These objects may at some stage exchange a message.

To explain the choice of terminology, notice that under the "static" interpretation, given the code of a system that is claimed to implement a model, we can trivially check whether the associations in the model are reflected in the code and vice versa by inspecting the code; there is no need to run the code. The "dynamic" notion, by contrast, permits associations that may be observable only in the behaviour of the system, not from static analysis of the code alone.

There are, of course, other possible relationships between P and Q, not covered by either of these options. For example, perhaps an object of class P may receive an object $q : Q$ as argument to a message, and may send q on to a third object without either storing it in an attribute or sending it any messages. It would be possible to include this kind of relationship within the definition of dynamic association, or alternatively to define a third kind of association. In the present work we prefer to disregard such relationships as being less interesting than the relationships we do consider. Thus we would recommend that modellers

continue to use Dependencies in the (rare) cases where such relationships need to be recorded.

Notice the cautious wording of the dynamic notion of association; the fact that we need to be so cautious is in fact the strongest argument against this notion. Granted that our aims are pragmatic – we want to be able to manipulate UML models reliably and check code against them – we could not usefully define instances $p : P$ and $q : Q$ to be linked exactly if they *do* exchange a message, or define P and Q to be associated if and only if some instances of those classifiers do exchange a message. Quite apart from the fact that for real systems such things will be affected by the environment of the system, including the behaviour of its actors, the notions will be undecidable (that is, it is impossible even in principle to write a program for this family of questions that could always terminate and answer the given question correctly). This is because to decide, given the code of a system, whether any instances of two given classes exchange a message, is at least as hard as the Halting Problem[3]. For this reason we have to settle for an approximation; under suitable assumptions it is possible to provide syntactic conditions sufficient to guarantee that instances of the classes *cannot* exchange a message, although the failure of these conditions will not guarantee that they *do*.

Oddly, therefore, from the point of view of the *links* (as opposed to the associations) the "static" view is in some sense more dynamic: the existence of a link between a given pair of objects is a property of one certain point in an execution. (As in the dynamic case, so in the static case we will not be able to decide in general whether two given instances will ever be linked. But this may not concern us, given that we can decide whether their classifiers are associated.)

[3] To be explicit: consider the family of systems in which the n^{th} system simply invokes method f of P with argument n, where f's behaviour given n is that it simulates the behaviour of the n^{th} Turing machine, sending a message to an instance of Q after the Turing machine terminates, if it ever does. Then if we could decide for each system whether an instance of P ever sent a message to an instance of Q, we could also decide whether the corresponding Turing machine halted; which is impossible.

Notice that in the dynamic view the *non*-existence of an association between P and Q implies by definition that instances of P and Q will never exchange a message; this can be useful in modelling as it provides a way to understand how far the effect of changing an object's interface can spread. In the static view, there is no such guarantee: objects may exchange messages without their classifiers being associated, as with A and B in Fig. 2.

Directionality and Ends So far, we have focused on Associations and Links, not on AssociationEnds and LinkEnds. The astute reader may be concerned about code such as:

```
class P {
  private Q myQ;
  public void foo() {myQ.mung(this);}
  public void bar() {...}
}

class Q {
  public void mung(P p) {p.bar();}
}
```

Clearly, there is an association between P and Q, but is it dynamic or static? From the point of view of P, it appears to be static, because of the attribute myQ of P. From the point of view of Q, however, which does send a message to an object of class P but does not have any corresponding attribute, it appears dynamic. Whilst the example may appear contrived, similar "callback" situations are common, for example in several well-known design patterns.

In response to such examples we can decide to classify, not Associations and Links, but AssociationEnds and LinkEnds. According to this view, the AssociationEnd which describes from P's point of view how Q is accessed will be

static, whilst the other AssociationEnd is dynamic. Figure 5 illustrates, using stereotypes.

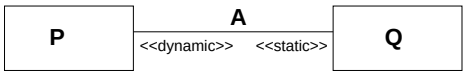

Fig. 5. Association with stereotyped AssociationEnds

The disadvantage of this approach is that it is notationally clumsy, and makes it nonsensical to say anything about whether an association is to be used statically or dynamically until its navigability has been decided. When we go on to consider the relationship between the class diagram and collaboration diagrams, we see that this approach forces us always to think of pairs of association ends. Following [7] we may actually consider this an improvement in modelling practice; but it is some way removed from the habits of many modellers today. A compromise solution consistent with our wording so far is to consider ends to be the things which are primarily static or dynamic, but to add the derived notion that an association is static whenever at least one of its ends is static; thus if there is some structural connection between the classes the association is static. The modeller can then choose whether to think in terms of association ends from the beginning, or initially only in terms of associations, according to their beliefs about the right order in which to make design decisions.

Law of Demeter It is interesting to observe that the effect of obeying the Law of Demeter[11] is to lessen the observable difference between static and dynamic association. Indeed, one way to describe the spirit of the Law would be to say that when objects of two classes have the potential to exchange a message, that fact should be obvious from examining the code of the two classes, in order that the dependencies in the design can be easily understood and so minimised. In other words, by limiting the design space we can get something approaching the best of both worlds.

The Law of Demeter states that, in responding to a message $m(a_1, \ldots a_n)$, an object o should send messages only to the following objects:

1. o itself;
2. any objects received as arguments to the message m, i.e., any of $a_1 \ldots a_n$ which are objects rather than values of basic types;
3. objects linked directly from o, i.e., objects which are (or, to which references are) the current values of attributes of o;
4. objects created by o in the course of responding to m.

Most importantly, this implies that o should not send messages to objects which are the return values of messages sent by o in the course of responding to m. This prohibition does not apply, of course, if such messages are permitted according to the clauses listed – for example, if o receives p *and then stores it in an instance variable*, it then becomes legal for o to send a message to p, by virtue of clause 3. What we do forbid is code like

```
(someObject.m1(...)).m2(...)
```

where message $m2$ is being sent to the object which is the return value of message $m1$, whose class is not mentioned in this code.[4]

To see the relevance of this, suppose that in the course of responding to m, $o : O$ sends a message and receives object $p : P$ (or a reference to p) in return; suppose for clarity that this is O's only relationship with P, so that, for example, o does not store p in one of its instance variables. Therefore in our terminology there is no static association between classes O and P. Now, according to the Law of Demeter, o must not send p a message. If o *did* send p a message, this would in our terminology mean that there was a dynamic association between classes O and P, although there was no static association between them. That is, the Law of Demeter forbids a large class of the cases in which static and

[4] Always accepting, however, that it is likely that a `#include` directive or similar will be required to make the code compile – the dependency is not completely invisible, but it is harder to localise than it need be.

dynamic associations fail to agree, and thus eliminates a large potential source of confusion.

We may ask whether following the Law of Demeter completely eliminates the distinction between static and dynamic association. Unfortunately perhaps, it does not. For example, when o receives message p as an argument and then sends a message to p (clause 2 above), there will in general be a dynamic association but no static association, even though the situation does not violate the Law of Demeter.

Our next step might be to ask whether there might be other reasonable rules of thumb which would eliminate the other cases where static and dynamic notions differ; can we claim that in a well-designed program, the notions will always coincide? In fact it seems that we cannot. For one thing, there seems to be no good reason to forbid the use of clause 2 just described. On the other hand, it is in any case important to realise that the Law of Demeter must not be followed blindly. When the return value from a message is an object of a standard library class, such as a collection, it often happens that the returned object can only be used by sending it messages, for example to extract the relevant object(s) from a collection. This violates the letter of the Law of Demeter. It does not, however, necessarily violate its spirit, which is to limit indirect dependencies which are obscure in the code. The reason for this is simply that it is often reasonable to assume, conservatively, that all code in a system is dependent on all parts of a standard library – rather than attempting to limit such dependencies, we regard the library as part of the programming language. Whether this is really reasonable depends on the circumstances; for example, the more stable and standard the library concerned, the more likely it is to be reasonable. Judgement is required in each case.

Derived associations Clarifying the distinction between static and dynamic association also helps us to put derived associations on a firmer footing. Typically, when a derived association is used in a diagram it is to indicate that two static associations may be composed to yield a derived association, which will be dy-

namic. (For example, in an implementation of Figure 6, $a : A$ might have a reference (implementing f) to $b : B$ which might have a reference (implementing g) to $c : C$, and a might send a message to c by first requesting from a the reference to c (implementing the derived association h.)

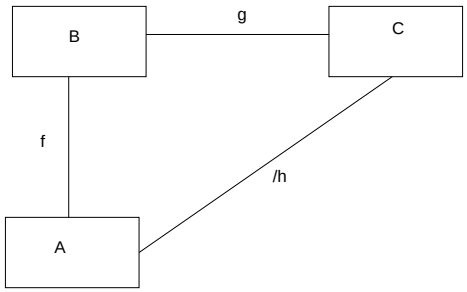

Fig. 6. A derived association

If we were to insist that all associations were static, we would be in the unfortunate position that such derived associations would not be associations at all.

The discussion in this subsection shows that the two notions of association, static and dynamic, each have their own advantages to the designer. Returning to our classification of the uses to which modellers may put associations, we may summarise by saying that the static notion better supports aim 1(b), whereas the dynamic notion better supports aims 1(c) and 2. For aim 1(a), either version will suffice. Aim 1(d) is more problematic: although undoubtedly the static view is easier to reason with, the main requirement is for all associations in the model, of whatever kind, to be usable in constraints. At this stage the case is strong for allowing both versions in UML, though of course any modeller must be aware of which they are using. In order to explore the implications of the choice further, we need to settle another question about what an association is.

3.2 Tuple or model element?

We come now to the issue which, though important and exposing contradictions in the UML standard, is more easily resolved: is a link simply a tuple, or is it more than that?

Consulting the definition of Association first, [12] 2-19 says:

> The instances of an association are a set of tuples relating instances of the classifiers. Each tuple value may appear at most once. [...] An instance of an Association is a Link, which is a tuple of Instances drawn from the corresponding Classifiers.

That is, a link is completely determined by the instances it links. Moreover, and more controversially, 3-84 says: Links do not have instance names, they take their identity from the instances that they relate.

If we take literally the statement that a link "is" a tuple of instances, then a link is a purely mathematical object, with no identity of its own: [5] and cannot have any kind of data beyond the literal tuple.

In particular, it cannot be possible to navigate from a Link to any other ModelElement (intuitively, there is "nowhere to store the reference" which would be used for such navigation). Thus in this interpretation we do not need to impose, as a constraint, the fact that the set of links which are instances of a given association cannot include two different links of an association between the same two instances; it is automatic from the fact that a link is a tuple (and the definition of "set").

[5] We do not wish to get into a philosophical discussion of what identity means. For the purposes of this paper, a class of things informally *has (the property) identity* if two elements of that class may own the same data (where "data" of course does not include "the identity of the element") and may nevertheless be distinguishable. It will suffice to say that if it is *meaningful* to ask "Is this a perfect copy of that, or is it actually the same thing?" – whether or not the question can be answered in a given computation framework – then the things concerned have identity.

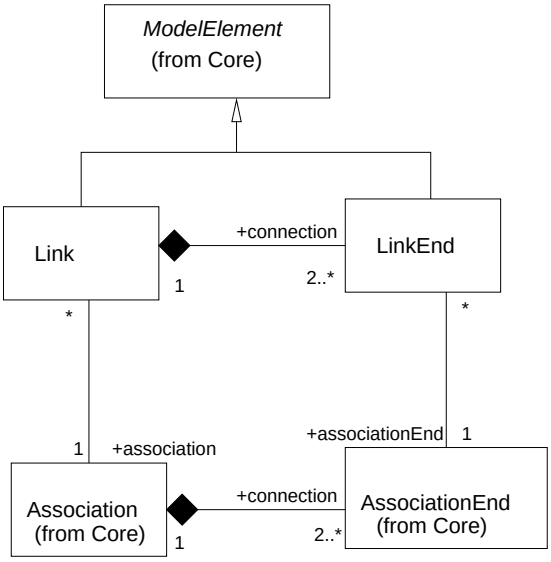

Fig. 7. Extract from UML1.4 metamodel

However, Link does in fact have navigable associations, as described in [12] (for example, see the extract from the metamodel given in Fig. /refmetamodel). Nor would altering the specification of Link specifically help to resolve this, since a Link is also a ModelElement. A ModelElement is *not* just a tuple: for example, any model element has a name and an identity of its own. These are automatically inherited by Link[6], so we have an inescapable contradiction: from the fact that a Link is a tuple we deduce that it cannot have data beyond the tuple, from the fact that it is a ModelElement we deduce that it must.

The easiest way to resolve this – in the sense that it involves a minimal change to the UML1.4 specification – is to decide that the wording of 2-19 and 3-84 is

[6] UML frequently specifies that certain things are *not* inherited; this is type-theoretically problematic. Pragmatically, it seems like nonsense that two instances could be indistinguishable when viewed as elements of a subclassifier – which by UML's intention carries more information than (is more specialised than, is a sub-type of) its parent – but distinct viewed as elements of the parent.

misleading; rather than literally *being* a tuple, a Link simply *determines and is determined by* a tuple: it determines a tuple via its association (connection) with LinkEnd, and to ensure that it is determined by a tuple we do need to add a constraint which forbids that two Links of a given association determine the same tuple. Indeed, [12] has such a constraint on page 2-110, further supporting the view that this is what is intended.

Henceforth we will reject the view that a link is (simply) a tuple: instead, we view a Link as a ModelElement in its own right. However, as a link *determines* a tuple, an Association *determines* a set of tuples. When it is convenient to talk about the set of tuples – that is, the relation – determined by an association A we will denote this A_R.

3.3 Multiplicity and multiple links

Next we address the implications of our choices for the definition of multiplicity of associations, which leads us into a consideration of whether multiple links between the same pair of instances should be permitted. [12] 2-66 specifies clearly enough:

> The multiplicity property of an association end specifies how many instances of the classifier at a given end (the one bearing the multiplicity value) may be associated with a single instance of the classifier at the other end. A multiplicity is a range of nonnegative integers.

This is a natural definition *which is based solely on the relation A_R determined by an association A*, and therefore makes sense for both static and dynamic associations. It is often what the modeller wants; but as usual, there are tradeoffs. To see one pragmatic disadvantage of this definition, consider the example given above in Fig. 3. Because the edge $e2$ is linked by "is incident on" to just one Node, $n2$, the multiplicity constraint is violated; although the modeller probably intended it simply to represent the domain-level constraint that any

edge has two ends. Replacing the multiplicity with 1,2 makes this object diagram legal, but also allows some unintended configurations.[7]

This is a possibly unintended feature rather than a bug: there is nothing wrong with leaving the multiplicity definition as it is, whichever interpretation of association we are using; the worst that will happen is that a modeller will be inconvenienced.

Notice that because of the constraint on page 2-110 which says that there may be at most one link of an association which links a given pair of objects – which we refer to for short as C – it would be illegal to show two links between $e2$ and $n2$. Pragmatically, however, it is tempting for the modeller to do so and to consider that the resulting instance diagram should conform to the class diagram, and it is not obvious why this should be forbidden. Indeed, it has been argued, for example by Genilloud [5], that it is useful to allow multiple links between the same instances in a given association, exactly because it would make it possible to model the graph shown in Fig. 3 more naturally. Of course, a *set* of tuples of instances by definition does not include the same tuple more than once, so someone who takes the view that an association *is* a set of tuples does not have this option. Given, however, that we have here rejected the "pure tuple" view, it would be a simple matter to remove the constraint C from the UML standard, thus permitting multiple links of an association A between the same instances. (A_R, being a set of tuples, would of course still include the tuple of instances just once. One could consider *defining* an association as a *bag* of tuples, but this gives something close to the worst of all worlds.)

Would dropping constraint C so as to permit multiple links between the same instances be an improvement, perhaps with some appropriate adjustment to the definition of multiplicity? The answer depends crucially on whether we take the dynamic or static view of associations.

[7] In fact, the most practical solution might be to interpose a "fictional" (in the sense that it does not model an obvious real-world entity) EdgeEnd, so that an edge connecting a node to itself would have two distinct EdgeEnds, which just happened to refer to the same Node. C.f. AssociationEnd etc. in the UML standard!

S In the static view, removing C can be seen as an attempt to help the modeller trying to model a graph with classes Edge and Node and a single association "is incident on", discussed above. We could replace the definition of multiplicity in the current standard by one that specified that an association end of association A having multiplicity m is not a statement about the tuples in A_R, but instead a structural statement about the classifiers associated by A; for example, if the implementation of class P will have a member which is an array of 3 Qs, then the corresponding association of P and Q will have multiplicity 3, and the correctness of the implementation will not be affected by the possibility that two or more of the array's elements are the same Q. Whether this would be an improvement is a matter of modellers' taste. In its favour, it can be argued that, given this suggested definition, it is easy to impose an additional constraint on the number of *distinct* instances, at any level (for one association, for one model, in a profile). That is, the current definition is easily recoverable by modellers who want it. The graph example demonstrated that the reverse adaptation is less easy, so we could argue that this suggested definition provides modellers with more flexibility than UML's current definition. Whereas UML's current definition of multiplicity is independent of the choice of a static or dynamic notion of associations, however, this proposed replacement only seems sensible with a static notion. UML could permit both variants of association and both variants of multiplicity, but would have to ensure that nonsensical combinations were excluded: this is likely to be a recurring problem when resolving disagreements about intended interpretations of parts of UML by allowing a choice between the interpretations.

D In the dynamic view, removing C could be seen as an attempt to model the fact that a single association may be used by the same tuple of instances to exchange more than one message; for example, if Edge's code at some point requires that both Nodes linked to the Edge be notified of some change, we might allow two links determining the tuple $(e2, n2)$ to represent the fact that $n2$ will receive the message twice. But this is an unfruitful view:

not only is there a difficulty about defining which messages "count" for a given association, but also, for any reasonable definition the question of whether given code correctly implemented a given UML class diagram would be undecidable.[8]

3.4 Links, associations and collaborations

Up to this point, it has been defensible to view the static version of association as correct and the dynamic version as a mistake. This view becomes harder to defend when we consider the behaviour (the dynamics!) of the system.

The metamodel classes AssociationEnd and LinkEnd are provided with stereotypes ≪association≫, ≪global≫, ≪local≫, ≪parameter≫, ≪self≫ corresponding to the different reasons why the corresponding instance is visible [12](2-103). However, any LinkEnd occurs in exactly one Association (because of the two relevant multiplicities of 1 in Fig. 2.17 on 2-98, an extract from which was shown in Fig. 7). If the corresponding instance is visible for any reason other than because of the existence of an association, there will in general *be* no Association with which the LinkEnd can be linked; contradicting the multiplicity constraint. (Note that of ≪association≫ it is said that it "Specifies a real association: default and redundant, but may be included for emphasis" – this suggests that the author of the section realised that the other stereotypes were in some sense unreal, but perhaps did not follow through the implications to the point of realising that the standard as written was genuinely inconsistent.)

This appears to be a serious problem, not resolvable by interpretation – at least, not in the presence of the popular static interpretation of associations – but definitely requiring a modification of the standard. There are two obvious approaches to fixing what we may call the *Baseless Link Problem*. We could

[8] We emphasise: class diagram. Of course, as soon as a UML model includes behavioural information, it is to be expected that such questions will be undecidable, but we may hope to avoid this at the level where we consider only the information contained in a static structure diagram.

change the specification so that some Links are allowed not to be instances of any Association; but this seems inelegant. Alternatively, we could allow some Associations not to be static. For example, we could add submetaclasses StaticAssociation and DynamicAssociation, and permit Links to be instances of either. Those favouring the static view of Association would then choose to record only the StaticAssociations in their class models, but could still use Links that were instances of DynamicAssociations in their collaborations. If desired, tools could identify links in collaboration diagrams that could not be instances of any static association, and bring them to the attention of the user to ensure that the user does know how the communication is to be achieved.

(We might also attempt to resolve this by supposing that the stereotyped versions of LinkEnd, corresponding to sub-metaclasses of LinkEnd, might relax the multiplicity constraints in Fig. 2.17. But this would be odd, given that subclassifiers in a model are supposed to inherit participation in associations "subject to all the same properties": why would we have one rule for models and another for metamodels?)

Remark A separate, less serious problem arises from the fact that the same stereotypes are defined both for AssociationEnd and for LinkEnd. A coherence condition must be intended, namely that if a given LinkEnd is stereotyped then the corresponding AssociationEnd should have the same stereotype. This would be easily remedied, but points up a general drawback to the stereotype mechanism: because it applies to one metamodel class at a time, it does not lend itself very well to situations where several related metamodel classes need to be specialised coherently to achieve the specialised aim of the modeller.

We now leave the basic definition of associations and move on to slightly more esoteric matters.

4 Associations and Generalisation

Generalisation and associations interact in two related ways: first, we need to understand the implications of a classifier being involved both in an association

and in a generalisation, and second, we need to understand the implications of the fact that Association is itself a GeneralizableElement.

4.1 Associations between generalised/specialised classifiers

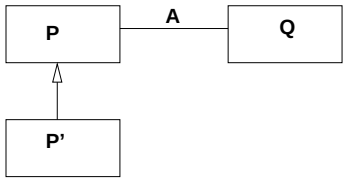

Fig. 8. A situation in which we expect an "induced" association between P' and Q

If classifier P' is a specialisation of P, and P and Q are associated by A, then because of substitutability considerations we expect P' and Q to be associated. Specifically, the restricted relation $\{(p,q) : p \in P', q \in Q, (p,q) \in A_R\}$ (in which an instance p of P' is linked to an instance q of Q iff p is linked to q when p is regarded as an instance of P) will often be an interesting design object. How is it reflected in the UML metamodel? On page 2-29 we read that any Classifier, being a GeneralizableElement, inherits associationEnd from its parent, meaning that The child class inherits participation in all associations of its parent, subject to all the same properties. That is, P' and Q are intended to be associated *by the same association A as P and Q*. By applying this notion twice, we get also that if A relates P and Q, and if P', Q' are specialisations of P, Q respectively, then A is to be regarded as also associating P' and Q'.

This cannot quite be taken literally; the Association "knows" what classifiers it relates, and in our example, that will include P, not P'. We *could* be completely formal by inventing the notion of a Generalization inducing, for every association A participated in by the parent GeneralizableElement, a new association participated in by the child GeneralizableElement. The determined relation of this induced association would be the restriction of A_R to instances

of the child. However, it is not immediately clear that there is any advantage to modellers to be gained by doing this, so to talk about "the same association" relating different classifiers probably falls within the "by abuse of notation" family of pseudo-errors dear to mathematicians. We might, indeed, feel that it was an advantage to have a distinct Association in the model for every pair of associated classifiers. Note that this need not have any impact on the diagrams. When an association is "simply" inherited in this way, modellers would not normally redraw the association at the P', Q' level, but we could if we chose regard this as mere notational convention: it does not have to determine whether there is or is not a new Association in the model. An alternative means to the same end is to regard this as a first example of generalisation of associations, which we consider next.

4.2 Association as GeneralizableElement

In the UML1.4 metamodel Association inherits from GeneralizableElement; that is, it is permitted for a model to contain an association which is said to be a generalisation of another. What should this most usefully mean? Note first that the "segment descriptor" style of definition, see e.g. [12] 2-70, is quite unhelpful in this context. [12] 2-21 makes a start[9]:

> The child must have the same number of ends as the parent. Each participant class must be a descendant of the participant class in the same position in the parent. If the Association is an AssociationClass, its class properties (attributes, operations, etc.) are inherited. Various other properties are subject to change in the child. This specification is likely to be further clarified in UML 2.0.

When we regard one association as a more exacting, or more specialised, version of another, it is natural to expect that a link of the more specialised

[9] There is also the non-sentence on 2-66: "Moreover, the classifier, or (a child of) the classifier itself." which may be connected

association may be regarded as a link of the more general association, but not the other way round. Going further than this is problematic: notice that it is not completely clear what else subtyping for associations should really mean, or in what contexts substitutability should apply. We do not attempt to solve the problem here, partly because it seems futile given that the issue is already highly complex and controversial for classes (although the controversy there is so old that it is often possible to overlook it). On a more technical UML note, notice that an AssociationEnd is *not* a GeneralizableElement, so the option of interpreting a subassociation between subclassifiers by subclassing all of the model classes involved is not available to us.

This will tend to lead us to a notion of generalisation of association which is merely a dressed-up version of subset inclusion of the links. In the absence of a clearly useful notion of subtyping for Associations, one could argue that it would be better to use a new name for these subset relationships, rather than allowing Association to participate in Generalizations. (The counter argument is that this would interfere with AssociationClass, which we consider in the next section.) This simple notion may well be sufficient for modelling purposes, however. A particularly simple case is when there are two different associations, say A and B, between the *same* classifiers, with the property that instances i and j are associated by A if they are associated by B, but may also be associated by A without being associated by B: that is, $B_R \subseteq A_R$. This situation rather frequently arises in modelling; consider, for example, relationships "is a member of" and "is captain of" between TeamMember and Team.

The other context in which generalisation of associations is often mentioned is the "animals eat food, cows eat grass" problem. For the benefit of any reader who has not met this classic before: suppose that classifier Cow is a specialisation of Animal whilst Grass is a specialisation of FoodStuff. The two specialisations of classifiers are quite independent: the inheritance of association "eats" does not ensure that Cows eat Grass, that is, that a Cow is linked by "eats" *only* to Grasses. (Nor should it, of course: the designer might have intended the spe-

cialisations to be independent.) However, we have already pointed out that the fact that subclassifiers inherit AssociationEnds means that there is an induced association between Cow and Grass whose links are exactly those that link instances of Cow to instances of Grass, and we have mentioned the possibility of modelling this, and the other associations induced by generalisations of classifiers, as a specialisation of the original "eat" association. Indeed, the confusion which may result from regarding all the various associations between FoodStuff, Grass, Cow and Animal as the same association "eats" in different guises argues for that approach.

5 Relationship with AssociationClass

In UML, an AssociationClass is both an Association and a Class. The idea is that this allows something which, conceptually, is a link between two objects also to own data and even provide behaviour in its own right. The classic example is the Account AssociationClass between two classes Bank and Customer.[10]

In the metamodel, AssociationClass and Association are significantly different; it is not the case, formally, that an AssociationClass which does not happen to have any Features is the same as an Association. This is contradicted in the Notation Guide (3-86, my italics):

> Generalization may be applied to associations as well as to classes. To notate generalization between associations, a generalization arrow may be drawn from a child association path to a parent association path. This notation may be confusing because lines connect other lines. An alternative notation is to represent each association as an association class and to draw the generalization arrow between the rectangles for the association classes, as with other classifiers. This approach can be used even if an association does

[10] Notice, however, that it is *not* permitted in UML1.4 for the same customer to have two or more accounts with the same bank, because the constraint we called C still applies! This is another argument for removing it.

not have any additional attributes, because *a degenerate association class is a legal association*.

It would arguably be sensible to make the quotation from the Notation Guide above true: make Association a Classifier (so that it is automatically a GeneralizableElement, automatically has Instances, etc., rather than having these properties specifically added). We could then use the terms "association class" and "plain association" where it was necessary to distinguish between associations that do and do not have added features. It is not quite clear why this has not been the approach taken by the UML standard-writers. Perhaps the most likely explanation was that there was a desire to keep associations simple, reflected in the statement that an association is (simply) a set of tuples. However, we have argued against the idea that this view is tenable, and it is not clear what the remaining objections might be. The change would simplify the UML definition, in which there are currently many concepts which are defined twice, once for Classifiers and once for Associations (the family of XRoles, for example).

Qualified associations Making Association be a specialisation of Classifier would regularise the position of qualified associations, which are currently rather confusing. The qualifiers are variously described ([12] 3-76, 3-77) as being attributes of the Association, the AssociationRole(!) or the AssociationEnd; this necessitates a clumsy description of Attribute, since an attribute can now be either part of a Classifier or a qualifier on an Association. If an Association was a Classifier, then qualifiers would not require such special treatment: they would simply be attributes of the Association, considered as Classifier.

6 Related work

As mentioned before, associations arise from a long history of modelling using relationships, including in database theory, in ER diagrams and in earlier object oriented modelling languages, notable OMT. This paper does not attempt to give a history of this work: suffice it to say that almost without exception,

the view taken of (what we now call) association is to identify the association with the relation it defines, and to take the static viewpoint. A representative example is [1]. A prominent strand is to discuss different ways of presenting the mathematical relation: for example, as a constrained pair of maps between the related sets, or in relational tables. Representative examples are [9, 2].

Rumbaugh's paper [15] contains an interesting discussion of relationships in UML; his "contextual associations" are ancestors of my "dynamic associations".

Övergaard has considered the formalisation of associations as part of his remarkably complete operational semantics for UML [14]; see also [13]. His approach, however, is very complex, and it is not clear how to relate it to other approaches; at the least, neither of the cited references address the issues considered here explicitly.

Many people have formalised parts of UML's class diagrams, including associations. Almost all model associations as relations between sets of instances, discarding the conflicts principally considered here at an early stage: we may mention particularly [16], [10] (using Z and B respectively). Most interesting for our purposes is [4], which (whilst also settling for modelling associations as relations) addresses some issues which have (deliberately) not been considered here, such as the changeability property of AssociationEnd. There is also a considerable amount of work concerning the semantics of aggregation and composition, for example [3].

7 Conclusions and implications

In this paper we have attempted to shed light on the root causes of some persistent disagreements about the use of associations in modelling and their definition in UML.

The inclusive spirit of UML leads us initially to hope that it is possible to allow all of the variants described here, leaving the choice of which is appropriate to the designer. To a certain extent this is possible. The problem described in Section 3.4 needs to be fixed. However, we may certainly allow the designer to

choose whether a dynamic or a static association is what they wish to represent, and it is straightforward, if we wish, to define stereotypes ≪static≫ and ≪dynamic≫ (on all appropriate metamodel classes) for the variants. Even for multiplicities, there seems to be no good reason why both of the definitions considered here should not be available. Thus particular profiles for UML could choose the appropriate variant.

We must leave open the question of whether Associations should be Classifiers: the idea is attractive, but this paper has not proposed the details of the considerable modification that this would require to the UML standard. We do recommend, however, that it should be seriously considered, and are pleased to note that the submissions UML2.0 do consider the issue, and that one of the submissions does indeed make Associations Classifiers. Since the documents available at the time of writing are work in progress, in which many important details have not been worked out, it is not appropriate to discuss the proposals here.

However, the question about whether a link is just a tuple or is a model element in its own right is more fundamental. There is no sensible way to make a notion of association that is just a set of tuples have a common ancestor with a notion in which instances of associations have identity. The former would have to avoid allowing a tuple to be a model element at all, because model elements have identity by definition. In principle, we could make the core of UML define Association (and AssociationEnd etc.) but not Link; then another member of the UML family of languages (which for technical reasons would have to be a Preface, not a Profile as currently considered) could specify that its Associations were composed of Links. However, this would mean that core UML would not be able to make any use of Links, which would be an intolerably large change.

Fortunately, we hope to have convinced the reader that there is no need to insist on links being simply tuples: all the same benefits can accrue from having links which determine tuples, possibly with the constraint that no distinct links in the same association determine the same tuple.

Finally, it is important to point out one non-consequence of this decision. A major, perhaps the major, reason for wanting associations to be just relations is that relations are familiar mathematical objects which are easy to manipulate, and giving a UML diagram a semantics in such terms permits the use of such mathematical manipulations. The reader should not imagine that a refusal to define associations as being relations in the UML metamodel implies that this avenue is closed. The UML metamodel is not the semantics of UML, or at the very least, is not the only semantics of UML; semantic functions can and do define mappings from the metamodel to convenient mathematical domains. There is no reason why such a mapping should not map an Association to a relation, if the other information carried by the Association is not relevant for the purposes of the semantics. Indeed, the power of semantics is precisely this power to discard what is irrelevant *for some particular purpose*.

Acknowledgements I would like to thank all the members of the pUML mailing list who participated in the discussion that provoked this work and in some private email. Especially, I thank Guy Genilloud for pointing out several problems with my early thoughts on the subject. Remaining mistakes are of course my own. I would also like to thank the participants in Dagstuhl Seminar 01041, especially the DMM group, for illuminating discussions. Gonzalo Génova and the anonymous referees of [17] and of this paper also made helpful suggestions.

References

1. R. Bourdeau and B. Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, 1995.
2. S. Cook and J. Daniels. *Designing Object Systems, Object-Oriented Modelling with SYNTROPY*. Prentice Hall, 1994.
3. D.Firesmith and B.Henderson-Sellers. Clarifying specialized forms of association in uml and oml. *Journal of Object-Oriented Programming*, 11(2):47–50, 1998.
4. Robert France. A problem-oriented analysis of basic UML static modeling concepts. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Confer-*

ence on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99), volume 34.10 of *ACM Sigplan Notices*, pages 57–69, N. Y., November 1–5 1999. ACM Press.

5. Guy Genilloud. Informal UML 1.3 - remarks, questions and some answers. Contributed to the ECOOP Workshop on UML Semantics, Lisbon, Portugal, June 1999. Available from http://icawww.epfl.ch/genilloud/.
6. Gonzalo Génova, Juan Llorens, and Paloma Martínez. Semantics of the minimum multiplicity in ternary associations in UML. In Martin Gogolla and Cris Kobryn, editors, *Proceedings of UML2001*, number 2185 in LNCS. Springer, 2001.
7. Ian Graham, Julia Bischof, and Brian Henderson-Sellars. Associations considered a bad thing. *Journal of Object Oriented Programming*, 9(9), 1997.
8. Terry Halpin. Augmenting UML with fact-orientation. In *Proceedings of HICCS-34*, 2000.
9. William Kent. *Data and Reality*. 1stBooks Library, 2000.
10. Régine Laleau and Fiona Polack. Metamodels for static conceptual modelling of information systems. Workshop on Defining Precise Semantics of UML, ECOOP, 2000.
11. Karl J. Lieberherr and Ian Holland. Formulations and benefits of the Law of Demeter. *ACM SIGPLAN Notices*, 24(3):67–78, March 1989.
12. OMG. *Unified Modeling Language Specification version 1.4*, February 2001. OMG document formal/01-09-67 available from http://www.omg.org/technology/documents/formal/uml.htm.
13. Gunnar Övergaard. A formal approach to relationships in the Unified Modeling Language. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.
14. Gunnar Övergaard. *Formal Specification of Object-Oriented Modelling Concepts*. PhD thesis, Department of Teleinformatics, Royal Institute of Technology, Stockholm, Sweden, November 2000.
15. James Rumbaugh. Modeling and design. *Journal of Object Oriented Programming*, 11(4), 1998.
16. M. Shroff and R. B. France. Towards a formalization of UML class structures in Z. In *Proceedings of COMPSAC'97*, 1997.

17. Perdita Stevens. On associations in the Unified Modelling Language. In Martin Gogolla and Cris Kobryn, editors, *Proceedings of UML2001*, number 2185 in LNCS. Springer, 2001.