Representation of Generic Relationship Types in Conceptual Modeling

Antoni Olivé

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
08034 Barcelona (Catalonia)
olive@lsi.upc.es

Abstract. A generic relationship type is a relationship type that may have several realizations in a domain. Typical examples are *IsPartOf*, *IsMemberOf* or *Materializes*, but there are many others. The use of generic relationship types offers several important benefits. However, the achievement of these benefits requires an adequate representation method of the generic relationship types, and their realizations, in the conceptual schemas. In this paper, we propose two new alternative methods for this representation; we describe the contexts in which one or the other is more appropriate, and show their advantages over the current methods. We also explain the adaptation of the methods to the UML.

1 Introduction

A significant amount of research work, in the conceptual modeling field, has focussed on the identification, analysis, representation and implementation of generic relationship types. A generic relationship type is a relationship type that may have several realizations in a domain. A realization is a particular combination of the participant entity types, and can be seen as a specific relationship type [Dach98, PiZD98]. The most prominent generic relationship type is the *IsPartOf* [Mots93, WaSW99], which, in a particular domain, may have realizations such as *Division-Company*, *Company-Company* or *Office-Building*. All other combinations (like *Company-Division*) are not allowed. Generic relationship types are also known sometimes as *semantic* [Stor93] relationship types.

Among other generic relationship types that have been studied in detail there are: *MemberOf*, with example realizations *TennisPlayer-TennisClub*, *Person-ProjectTeam* or *Person-Committee* [MoSt95]; *Materializes*, with example realizations *Car-CarModel*, *Performance-Play* or *Volume-Book* [PZMY94]; and *Owns*, with example realizations *Person-Building*, *Bank-Mortgage* or *Person-Car* [YHGP94].

From a conceptual modeling point of view, generic relationship types offer the following benefits: (1) They ease the definition of realizations, because their common aspects are predefined. Generic relationship types foster the reuse of conceptual

A. Banks Pidduck et al. (Eds.): CAISE 2002, LNCS 2348, pp. 675-691, 2002.

schemas [PiZD98]; (2) They allow the definition of general knowledge, in the form of derived entity types, attributes or relationship types, and of integrity constraints, which can be included in libraries, and 'imported' by conceptual schemas when needed [Matt88]; and (3) They facilitate automated schema design tools that help the designers to develop, verify, validate and implement conceptual schemas [Stor93].

In broad domains, there are many generic relationship types. The top-level categories of general ontologies [Cyco97, Sowa00] include many of them. If the conceptual schema for a particular domain is embedded within a general ontology, then many domain specific relationship types are realizations of generic ones. Given the benefits, some authors recommend that all relationship types should be defined as realizations of generic ones [PiZD98].

However, the achievement of these benefits requires an adequate representation method of the generic relationship types, and their realizations, in the conceptual schemas. Surprisingly, the current methods do not represent the generic relationship type explicitly in the schema, but only its realizations. The consequence is that the benefits cannot be achieved in their full extent. Some methods, mainly based on Telos [MBJK90], define the common aspects of the realizations at the metalevel, which are then 'inherited' by the realizations, but again -with very few exceptions- they do not represent the generic relationship type explicitly in the schema.

In this paper, we argue that the achievement of the above benefits requires that generic relationship types be represented explicitly in the schema. We propose two new alternative methods for this representation, and describe the contexts in which one or the other is more appropriate. Our methods can be integrated with existing conceptual modeling languages. In particular, we show their adaptation to the UML.

The structure of the paper is as follows. Section 2 defines the terminology and the notation that will be used in the paper. In Section 3 we describe our first method for the representation of generic relationship types in conceptual schemas. We show that the method can be adapted to the UML. We analyze also the possibilities the method offers to define additional knowledge and comment some possible extensions. Section 4 does the same for the second method. Section 5 analyzes the properties of both methods, and compares them with the current ones. Finally, in Section 6 we give the conclusions.

2 Terminology and Notation

In this paper, we deal with entities, relationships, entity types, relationship types and meta entity types. We take a temporal view, and assume that entities, relationships and entity types are instance of their types at particular time points, which are expressed in a common base time unit such as second or day. We make this assumption for the sake of generality, but our work is also applicable when a temporal view is not needed.

We represent by $In(e_1,e^1_1,t)$ the fact that entity e_1 is instance of entity type e^1_1 at time t. We assume multiple classification, and allow an entity to be direct instance of several entity types. An entity type is also an entity, which is instance of a meta entity type. Therefore, we represent also by $In(e^1_1,e^2_1,t)$ the fact that entity type e^1_1 is

instance of meta entity type e^2_1 at time *t*. We use predicate *In* as a shorthand for *IsInstanceOf*. Also, we use the convention that constants are capitalized, variables start with a lowercase letter, variables denoting entity types have the superscript *1*, and those denoting meta entity types have the superscript *2*.

A relationship type has a name and a set of *n* participants, with $n \geq 2$. A *participant* is an entity type that plays a certain role in the relationship type. $R(p_1{:}E_1,...,p_n{:}E_n)$ denotes a relationship type named *R* with entity type participants $E_1,...,E_n$ playing the roles $p_1,...,p_n$, respectively. We say that $R(p_1{:}E_1,...,p_n{:}E_n)$ is the *schema* of the relationship type and that $p_1{:}E_1, ..., p_n{:}E_n$ are their participants. When the role name is omitted, it is assumed to be the same as the corresponding entity type. In this paper, attributes will be considered as ordinary binary relationship types.

We represent by $R(e_1,...,e_n,t)$ the fact that entities $e_1,...,e_n$ participate in an instance of *R* at time *t*. We also say that $(e_1,...,e_n)$ is an instance of *R* at *t*. The usual referential integrity constraint associated with *R* guarantees that $e_1, ..., e_n$ are instance of their corresponding entity types $E_1,...,E_n$:

$$R(e_1,...,e_n,t) \rightarrow In(e_1,E_1,t) \wedge ... \wedge In(e_n,E_n,t)$$

In the formulas, we assume that the free variables are universally quantified in the front.

A *class relationship type* is an ordinary relationship type but where some or all of its participants are meta entity types. We distinguish between class relationship type and meta relationship type. Instances of the former are facts, while instances of the latter are relationship types. In the UML, a class relationship type is an association having one or more association ends with the value *Classifier* for the attribute *targetScope* [OMG01, p. 2-23]. An example of class relationship type could be:

CanBeExpressedIn (QuantityType, Unit)

where *QuantityType* is a meta entity type, and *Unit* is an entity type.

In this paper, we only deal with class relationship types where *all* participants are meta entity types. An example could be:

InstancesAreFruitOf (FruitType, TreeType)

with the meaning that "The instances of FruitType <ft> are fruit of some instance of TreeType <tt>". Example instances are the facts:

InstancesAreFruitOf (Apple, AppleTree,_),

InstancesAreFruitOf (Orange, OrangeTree,_)

Note the difference with the relationship type:

IsFruitOf (Fruit, Tree)

whose instances are the relationships between particular fruits and their corresponding trees.

A generic relationship type R is an ordinary relationship type $R(p_1{:}E_1,...,p_n{:}E_n)$ with a set of *m* realizations, $m \geq 0$, and a realization constraint. A realization *i* is a set $\{p_1{:}E_{1,i},...,p_n{:}E_{n,i}\}$. We will assume that the entity types of the realizations $E_{j,i}$ are

direct or indirect subtypes of the corresponding participant E_j in the generic relationship type R. For example, the generic relationship type:

IsPartOf (part:Entity, whole:Entity)

where *Entity* is the direct or indirect supertype of all entity types in a schema, could have two realizations:

{part:Division, whole:Company}, {part:Company, whole:Company}

The *realization constraint* of a generic relationship R with m realizations ensures that:

- if entities $e_1,...,e_n$ participate in an instance of R, then
- they are instance of a set $E_{1,i},...,E_{n,i}$ of corresponding types, such that
- $\{p_1:E_{1,i},...,p_n:E_{n,i}\}$ is a realization.

The general form of the realization constraint of a generic relationship type R with m realizations $\{p_1:E_{1,i},...,p_n:E_{n,i}\}$, $i = 1...m$, is:

$$R(e_1,...,e_n,t) \rightarrow [(In(e_1,E_{1,1},t) \wedge ... \wedge In(e_n,E_{n,1},t)) \vee ... \vee (In(e_1,E_{1,m},t) \wedge ... \wedge In(e_n,E_{n,m},t))]$$

3 Realizations as Subtypes

In this section, we describe our first method for the representation of generic relationship types in conceptual schemas. For presentation purposes, we structure the description in two parts: the basic idea of the method and its possible extensions. We show that the method can be adapted to the UML. We analyze also the possibilities that this method offers to define additional knowledge.

3.1 The Method

The basic idea of our first method is very simple: A generic relationship type $R(p_1:E_1,...,p_n:E_n)$ with m realizations ($m \geq 0$) is represented in the conceptual schema by $m+1$ relationship types:

- The generic relationship type R itself.
- A relationship type $R_i(p_1:E_{1,i},...,p_n:E_{n,i})$ for each realization, $i = 1...m$, with the same number (n) and name (p_i) of roles as in R.

The relationship type R is defined as derived by union of the R_i. Therefore, its derivation rule has the general form:

$$R(e_1,...,e_n,t) \leftrightarrow R_1(e_1,...,e_n,t) \vee ... \vee R_m(e_1,...,e_n,t)$$

The R_i are then subtypes of R. We call them *realization subtypes* of R. The realization subtypes may be base or derived.

Note that, in this method, there is a relationship type R whose population comprises all the instances of the generic relationship type. The existence of this

relationship type allows us to centralize the definition of the knowledge common to all those instances. The centralization of that knowledge is one of the main differences of our methods with respect to the current ones. We call our first method *Realizations as Subtypes* (RS), because the realizations are defined as subtypes.

As an example, consider the generic relationship type:

MemberOf (member:Member, group:Group)

and the three realizations:

{member:Person, group:ProjectTeam}
{member:Person, group:Committee}
{member:Manager, group:BoardOfDirectors}

Their representation in the conceptual schema using the RS method would comprise the generic relationship type and three realization subtypes. The common relationship type is the *MemberOf* itself, that we would define with the schema:

MemberOf (member:Member, group:Group)

with the meaning:

Member <m> is a direct member of Group <g>

and with general integrity constraints such as, for example, "An entity cannot be member of itself". The instances of *MemberOf* give all the groups of all members. In the RS method, there is a *single* relationship type such that its instances are all the groups of all members. The other three relationship types are the realization subtypes:

PersonMemberOfTeam (member:Person, group:ProjectTeam)
PersonMemberOfComm (member:Person, group:Committee)
ManagerMemberOfBoard (member:Manager, group:BoardOfDirectors)

In general, the meaning and general descriptions of these relationship types are essentially the same as that of *MemberOf*, because they are the same concept.

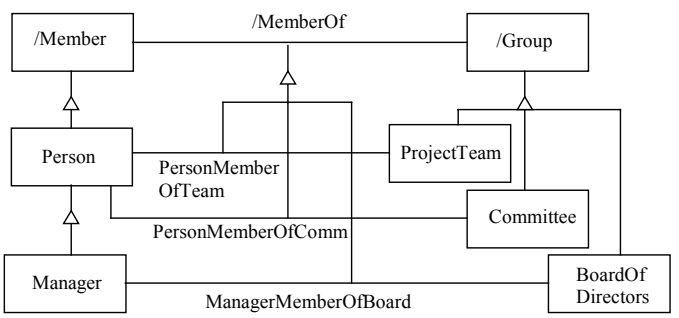

Figure 1. Representation in UML of the generic relationship type *MemberOf* in the RS method.

In the RS method, it is not necessary to define explicitly the realization constraint in the conceptual schema. The constraint is implicit in the fact that each realization

has its own relationship type, and that the generic relationship type is derived by union of the realization subtypes.

As we have explained in Section 2, we assume that the entity types of the realizations are direct or indirect subtypes of the corresponding participant entity types in the generic relationship type *R*. In the example, this means that *Person* and *Manager* are subtypes of *Member*, and that *ProjectTeam*, *Committee* and *BoardOfDirectors* are subtypes of *Group*. This fact ensures that the realizations conform to the referential integrity constraint of *MemberOf*.

On the other hand, it may be convenient to define the participant entity types of *R* as derived by union [Oliv01] of the corresponding entity types of the realizations. In the example, we could define that *Group* is derived by union of *ProjectTeam*, *Committee* and *BoardOfDirectors*. In this way, we prevent the occurrence of undesirable instances of *Group*. Moreover, we can then define easily constraints that must be satisfied for all instances of *Group*.

3.2 Adaptation to the UML

The RS method can be adapted easily to conceptual schemas in the UML. Figure 1 illustrates this adaptation with its application to the example. The generic relationship type (associations in the UML) *MemberOf* is defined as derived. Its derivation rule can be expressed formally using the OCL.

3.3 Extensions

The basic idea of the RS method can be extended in several ways. We describe briefly here one extension that we find useful: "generic relationship types with partial realizations".

A generic relationship type with partial realizations $R(p_1:E_1,...,p_n:E_n)$ is one such that only j of the participants are realized, with $j > 0$ and $j < n$; the other ones $(n - j)$ remain fixed. In the binary case, this means that one participant is fixed, and the realizations only give the other one. There are many examples of binary generic relationship types with partial realization; for instance[1]:

BasicPrice(Entity,price:Money)

with the meaning:

The entity <e> has the basic price of Money <m>

The realizations of *BasicPrice* concern only the participant *entity*; the other (*price*) remains fixed. Examples of realizations could be *Article*, *Machine* or *Travel*. In the RS method, there would be a relationship type for each of these realizations. The generic relationship type *BasicPrice* would be defined as derived by union of its subtypes. Once defined in the conceptual schema, *BasicPrice* can be combined with other elements to infer additional knowledge.

[1] This relationship type corresponds to predicate #$basicPrice : <#$Individual> <#$Money> of the CYC Ontology. See [Cyco97] for details.

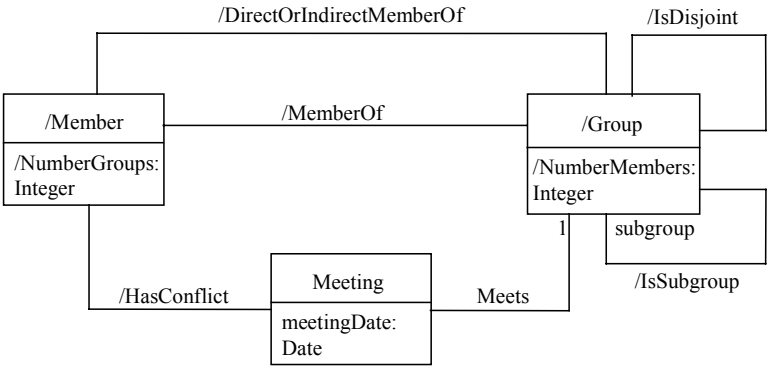

Figure 2. Example of additional knowledge at the level of the generic relationship type *MemberOf*.

3.4 Definition of Additional Knowledge

We have seen the representation of a generic relationship type *R* and its realizations, in the RS method. Now, we analyze how we can define additional knowledge related to *R* or to its realizations. This aspect is very important to evaluate the method, and to compare it with others. We distinguish among three kinds of additional knowledge:

- Related to a *particular* realization. This includes integrity constraints that must be satisfied only by the instances of *R* corresponding to that particular realization, new elements (entity types, attributes, relationship types) that can be defined related to the realization, and elements that can be derived from those instances. For example, we may need to define that, in a particular realization of *MemberOf*, a person cannot be member of more than three project teams.
- Related to *each* realization. This includes integrity constraints that must be satisfied by the instances of *R* corresponding to each realization, independently from the others, new elements that can be defined for each realization, and elements that can be derived from those instances. For example, for each realization of *MemberOf* we may wish to define a relationship type:
- MaximumNumberOfMembers (group:E_2, maximum:Natural) and the integrity constraint that the current number of members of the corresponding group is not greater than the maximum (a time-varying number given by the users). This knowledge can be predefined, and be part of a library of conceptual schema fragments, or of an ontology.
- Related to the *generic* relationship type. This includes integrity constraints whose evaluation requires the knowledge of all instances of the generic relationship type. For instance the constraint that a person cannot be member of more than five groups, which can be of any type (*ProjectTeam*, etc.). It includes also new elements that can be defined related to the generic

relationship type, and elements that can be derived from its instances. As in the previous case, this knowledge can be predefined.

Now, we analyze how each of the above kinds of knowledge can be defined in the RS method. It is easy to define additional knowledge related with a *particular* realization. The reason is that, in RS method, realizations are represented as explicit relationship types. For example, related with the realization *Person-ProjectTeam* we could define easily:

- The cardinality constraint that a project team must have between one and ten persons, and that a person cannot be member of more than two project teams.
- A reification of *PersonMemberOfTeam*, to represent the tasks assigned to a person as member of a team.

In general, the RS method does not allow a single definition of knowledge related to each realization. It must be defined explicitly in each realization.
Contrary to this, the RS method allows defining easily knowledge related with the generic relationship type. In our example, among the useful elements that we could relate to *MemberOf* there are:

- The derived relationship type *DirectOrIndirectMemberOf*.
- The integrity constraint that no entity can be direct or indirect member of itself.
- The derived attribute *NumberOfMembers* of a group.
- The derived attribute *NumberOfGroups* of a member.
- The derived relationship types *IsSubgroup* and *IsDisjoint* between groups.
- The relationship type *Meets*, between a group and a meeting.
- The derived relationship type *HasConflict* between a member and a meeting, stating that a member has another meeting with the same meeting date.
- and many others. Figure 2 shows the graphical definition of the above elements in the UML.

4 Metalevel-Governed Representation

In this section, we describe our second method for the representation of generic relationship types in conceptual schemas. For presentation purposes, we also structure here the description in two parts: the basic idea of the method and its possible extensions. We show that the method can be adapted to the UML. We analyze also the possibilities that this method offers to define additional knowledge.

4.1 The Method

The basic idea of our second method is also very simple: For each generic relationship type $R(p_1:E_1,...,p_n:E_n)$ we define two base relationship types in the conceptual schema:

- The generic relationship type R itself.
- A class relationship type $GovR(p_1:E_1,...,p_n:E_n)$, with the same number ($n$) and name ($p_i$) of roles as in R. The instances of $GovR$ are the realizations of R, and

therefore $E_1,...,E_n$ are metaentity types. We say that *GovR* is the *governing* relationship type of *R*, because the instances of *GovR* govern (constrain) the instances of *R* in the realization constraint. The name *GovR* is a mere notational convention. It is obtained by prefixing *R* with *Gov* (for governing).

As in the RS method, this method defines also in the conceptual schema a relationship type *R* whose population comprises all the instances of the generic relationship type. We call this method *Metalevel-Governed* (MG), because the instances of *R* are governed by instances of a class relationship type.

The basic idea of the MG method is similar to the one used in [Fowl97, p. 24+] to handle schema complexity, suggesting the introduction of two levels, called knowledge and operational, such that "instances in the knowledge level *govern* the configuration of instances in the operational level".

We illustrate the MG method with the same generic relationship example as before. Its representation in the conceptual schema will comprise the relationship types *MemberOf* and *GovMemberOf*. The instances of *MemberOf* (*member:Member, group:Group*) give all groups of all members. Note again that, in the MG method, there is also a *single* relationship type to represent the members of all groups. All characteristics of *MemberOf* are defined only once, in a single place.

The governing relationship type of *MemberOf* is:

GovMemberOf(member:MemberType, group:GroupType)

where *MemberType* and *GroupType* are metaentity types, whose instances are entity types of members and of groups, respectively. The meaning of *GovMemberOf* is:

Instances of MemberType <memType> may be member of
instances of GroupType <grType>

The realizations of *MemberOf* would then be defined as instances of *GovMemberOf* in the knowledge base. For example, the realizations (we omit the time arguments):

GovMemberOf(Person, ProjectTeam, _)
GovMemberOf (Person, Committee, _)
GovMemberOf (Manager, BoardOfDirectors, _)

Note that in the MG method, the realizations are explicit in the knowledge base. This allows the system to answer easily queries like "Who may be member of a ProjectTeam group?"

We define both *R* and *GovR* as base relationship types. In the section 4.4 we describe an extension in which *R* has a base and a derived part. The instances of *R* are constrained by the realization constraint. In the MG method, the general form of this constraint is:

$$R(e_1,...,e_n,t) \rightarrow \exists e^1_1,...,e^1_n\ (In(e_1,e^1_1,t) \wedge ... \wedge In(e_n,e^1_n,t) \wedge GovR(e^1_1,...,\ e^1_n,t))$$

The constraint requires that entities $e_1,...,e_n$ participating in an instance of *R* at *t*, must be instance of a combination of entity types $e^1_1,...,\ e^1_n$, respectively, such that the combination is one of the realizations of *R*. The adaptation of the general form of the realization constraint to *MemberOf* gives:

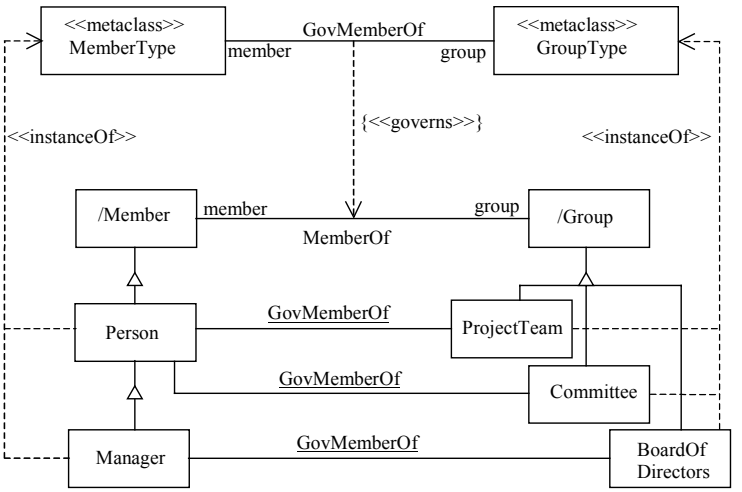

Figure 3. Representation in UML of the generic relationship type *MemberOf* in the MG method.

$$\text{MemberOf}(\text{mem},\text{gr},\text{t}) \rightarrow \exists e^1{}_1, e^1{}_2\ (\text{In}(\text{mem}, e^1{}_1, \text{t}) \wedge \text{In}(\text{gr}, e^1{}_2, \text{t}) \wedge \text{GovMemberOf}(e^1{}_1, e^1{}_2, \text{t}))$$

The constraint ensures that the two participating instances (*mem,gr*) in *MemberOf* have a pair of types ($e^1{}_1, e^1{}_2$) defined as a realization in *GovMemberOf*. This constraint would be defined in the conceptual schema, as any other integrity constraint. Note that if, at some time *t*, the population of *GovMemberOf* is empty then *MemberOf* cannot have any instance at *t*. Note also that our realization constraint does not change when a realization is removed from (or added to) *GovR*.

4.2 Adaptation to the UML

We now show how the MG method can be adapted to conceptual schemas represented in the UML. Figure 3 illustrates this adaptation with its application to the example. The generic *R* and the governing *GovR* relationship types can be represented as ordinary associations in UML. See *MemberOf* and *GovMemberOf* in Figure 3. The participants of *GovR* are defined with the predefined stereotype <<metaclass>> [OMG01, p.2-29] because their instances are classes (types). Realizations of *R* are defined as instances (links) of *GovR*.

Figure 3 shows three instances of *GovMemberOf*, which correspond to the three realizations of *MemberOf*. (Note that in the UML the association name is underlined to indicate an instance).

The entity types Person, Manager (resp., ProjectTeam, Committee, BoardOfDirectors) are defined as:

- direct or indirect subtypes of *Member* (resp., *Group*), and as

- instance of *MemberType* (resp., *GroupType*), shown as a dashed arrow with its tail in the entity type and its head on the metaentity type. The arrow has the standard keyword <<instanceOf>> [OMG01, p.3-93].

Ideally, the realization constraint would be predefined, like the constraints *complete* or *xor*, but this would require extending the UML metamodel. A more practical alternative is to use the extension mechanisms provided by the UML, particularly the stereotypes. We define a stereotype of *Constraint*, called <<governs>>, that constraints two associations: *GovR* and *R*. Graphically, an instance of this constraint is shown as a dashed arrow from *GovR* to *R*, labeled by <<governs>> in braces [OMG01, p. 3-27]. See the example in Figure 3.

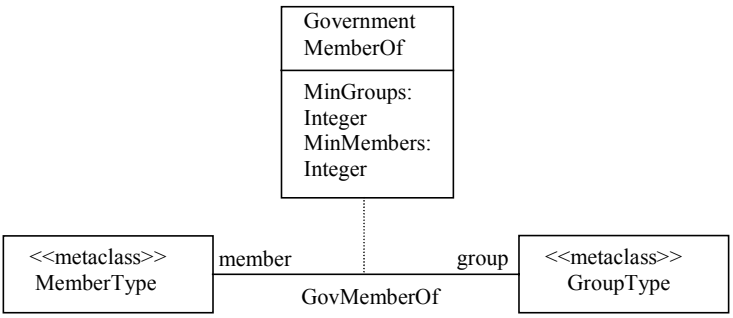

Figure 4. The reification of *GovMemberOf* allows an additional level of government of *MemberOf*.

4.3 Extensions

The MG method admits several extensions. One interesting extension is the reification [Oliv99] of the governing relationship type, *GovR*. This allows us to define details of realizations, which can be used to provide an additional level of automatic government of the generic relationship type.

We illustrate this extension with its application to the example. Figure 4 shows the reification of *GovMemberOf* into entity type *GovernmentMemberOf*. We have included two attributes in this entity type: *MinGroups* and *MinMembers*, which define the minimum cardinalities of the corresponding realizations. Thus, if we have *MinGroups = 0* and *MinMembers = 1* in the realization *Manager - BoardOfDirectors*, the meaning could be that not all managers must be members of a board, and that a board must have at least one manager as member. Many other attributes and relationship types could be included in *GovernmentMemberOf*.

The interest of the extension is that now we can define general integrity constraints on *MemberOf* to ensure that their instances conform to the 'details' given in the corresponding instance of *GovernmentMemberOf*. In the example of Figure 4, we would define two integrity constraints.

Another possibility this extension offers is to generate automatically specific integrity constraints and/or attributes and/or relationship types when a new realization

is defined (in the example, when *GovMemberOf* and *GovernmentMemberOf* are instantiated).

In some cases, this extension requires that realizations satisfy what we call the *realization exclusivity* constraint. This constraint may be needed to prevent that an instance of *MemberOf* has a member and a group such that they conform to two realizations. In some cases, this may not be acceptable because then there could be an ambiguity as to which realization applies.

The general form of the realization exclusivity constraint is:

$$(R(e_1,...,e_n,t) \wedge In(e_1,e^1{}_1,t) \wedge ... \wedge In(e_n,e^1{}_n,t) \wedge GovR(e^1{}_1,..., e^1{}_n,t) \rightarrow \neg\exists e'^1{}_1,..., e'^1{}_n ((e^1{}_1 \neq e'^1{}_1 \vee ... \vee e^1{}_n \neq e'^1{}_n) \wedge In(e_1,e'^1{}_1,t) \wedge ... \wedge In(e_n,e'^1{}_n,t) \wedge GovR(e'^1{}_1,..., e'^1{}_n,t))$$

The constraint requires that entities $e_1,...,e_n$ participating in an instance of R at t, cannot be instance of two different combinations of entity types $e^1{}_1,..., e^1{}_n$, and $e'^1{}_1,..., e'^1{}_n$ respectively, such that both combinations are realizations of R.

In the basic idea of the MG method, R is defined in the conceptual schema as base. This is overly restrictive. Some instances of R might be derived. For instance, in our example we could have that each project team has a manager and that, by definition, the manager must be considered member of the project team. To accommodate such cases, we extend the MG method as shown in Figure 5. The generic relationship type *MemberOf* is defined as derived by union of *BaseMemberOf* and *DerMemberOf*. As their name suggests, *BaseMemberOf* is base and *DerMemberOf* is derived. The designer would define the derivation rule of *DerMemberOf*.

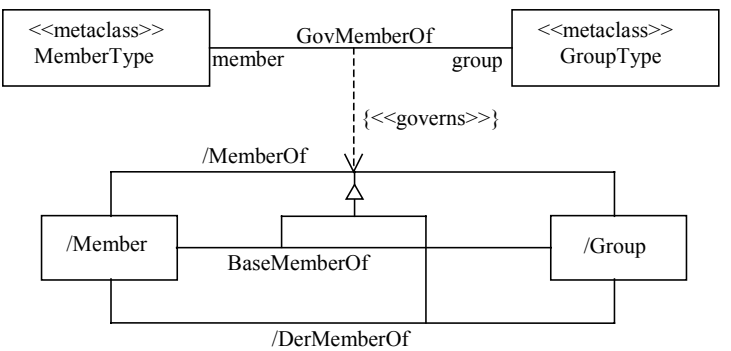

Figure 5. Extension of the MG method, in which the generic relationship type has a base and a derived part.

4.4 Definition of Additional Knowledge

Now we analyze how each of the three kinds of knowledge described in Section 3.4 can be defined in the MG method. It is not easy to define knowledge related with a *particular* realization. The reason is that, in the MG method, realizations are not represented as explicit relationship types. If needed, additional knowledge particular to a realization must be defined as refinement of the generic relationship type.

For example, assume that we want to define that *Committee* groups have at least two members. In the OCL this would be defined as:

context Group **inv** CommitteesHaveTwoOrMoreMembers:
self.oclIsKindOf(Committee) **implies** self.member ->size() > 1.

As we have seen, the reification of the governing relationship type eases the definition of knowledge related to *each* realization.

As in the RS method, the fact that we define only one relationship type for all instances of a generic relationship type *R*, allows us to define easily knowledge related to *R*.

5 Analysis and Evaluation

In this section, we analyze some of the properties of our two methods, and discuss the contexts in which their use may be appropriate. We do so by means of a comparison with the current methods. The comparison takes into account the following aspects:

- The definition of knowledge related to a generic relationship type, to each realization and to a particular realization.
- The knowledge of the existing realizations in a conceptual schema.
- The definition of new realizations.
- The simplicity of the representation.

5.1 RS vs. MG

The RS and the MG methods are equivalent with respect to the definition of knowledge related with a generic relationship type. The reason is that, as we have seen, both methods define explicitly the generic relationship type. The MG method, extended with the reification of the governing relationship type, allows an easy definition of knowledge related to each realization. In the RS method, that knowledge must be repeated in each realization. The methods differ also with respect to the definition of knowledge related with a particular realization. The RS method provides greater flexibility, since each realization has its own relationship type. In the MG method, that knowledge must be defined as refinement of the generic relationship type. Both methods allow querying the schema about the defined realizations.

The definition of new realizations is easier in the MG method: it is just a new instantiation of the governing relationship type. In the RS method, a new realization requires a new relationship type, and its definition as a subtype of the generic one.

With respect to the simplicity of the representations, we must distinguish between the structural and the behavioral parts of a schema. Structurally, we tend to consider both methods alike. For each realization relationship type in the RS method we have an instance of the governing relationship type in the MG method. Instead of the subtype relationships in the RS method, we have <<instanceOf>> relationships between entity and metaentity types in the MG method, and the {governs} constraint (compare Figures 1 and 3). Behaviorally, however, the MG method is simpler, because updates (insertions, deletions) and their effects are defined in a single place:

the generic relationship type. In the RS method, there must be a different definition for each realization, since in each case the updated relationship type is different.

Therefore, we conclude that both methods can be useful, depending on the generic relationship type and its realizations. In one extreme, with many realizations but without the need to define particular knowledge related to them, the MG method seems clearly better. In another extreme, with few realizations and with the need to define much specific knowledge in each of them, the RS method seems more appropriate.

5.2 One Relationship Type for Each Realization

We now analyze the current methods, starting with the simplest one, and the one most widely used in practice (with a few exceptions, presented below). The method consists in defining a different relationship type for each realization. The relationship types thus defined are not related each other, nor have a common root. The application of this method to our *MemberOf* example, with three realizations, would give only three relationship types:

- PersonMemberOfTeam (Person, ProjectTeam)
- PersonMemberOfComm (Person, Committee)
- ManagerMemberOfBoard (Manager, BoardOfDirectors)

The definition of a new realization is easy: just define a new relationship type. Note that these relationship types are like any other in the schema and, thus, there is no formal way to detect that they are realizations of some generic relationship type. Therefore, this representation method does not allow querying the schema about the defined realizations.

In this method, the knowledge related to a generic relationship type or to each realization must be repeated in each realization that requires it. The lack of an explicit generic relationship type makes the definition of that knowledge very complex. In contrast, the definition of knowledge related with a particular realization is very easy.

The representations obtained by this method are simple, provided that there are no too many realizations and that does not make extensive use of additional knowledge. Otherwise, the representations tend to be large and redundant.

In conclusion, the advantage of this method is its simplicity and the ease of definition of knowledge specific with a realization. This makes it very appropriate for conceptual schemas with only one realization of a generic relationship type, or when there are several realizations of a generic relationship type, but there is no need of additional knowledge related to it or to each realization, and there is no need to query the schema about the realizations of a generic relationship type.

5.3 One Marked Relationship Type for Each Realization

This is the most widely used method in practice for *some* generic relationship types. The idea is to define a relationship type for each realization, as in the previous method, but now all realizations of the same generic relationship type have a mark, which distinguishes them from the others. The real effect of such mark on the

conceptual schema is predefined, and may consist of some implicit static and/or dynamic integrity constraints.

The best example is the generic relationship type *PartOf*. The UML distinguishes two variants of it: aggregation and composition. Aggregation associations are marked graphically with a hollow diamond, attached to the whole entity type. The "only real semantics is that the chain of aggregate links may not form cycles" [RuJB99, p. 148]. Composition associations are marked graphically with a solid-filled diamond. Conceptually, a composition has "the additional constraint that an entity may be part of only one composite at any time ... " [RuJB99, p. 226]. Note that this knowledge is implicit and, therefore, it cannot be integrated with the rest of the knowledge defined in the conceptual schema. In the UML, the only predefined generic relationship types are the aggregation and the composition. Others may be added using the stereotyping extension mechanism.

The analysis of this method (one marked relationship type for each realization) reveals that it improves always the previous one in the following aspects:

- Some general knowledge, mainly in the form of integrity constraints, is attached automatically and implicitly to the realizations. However, the amount of knowledge that can be attached automatically is rather limited.
- The schema can be queried about the realizations of a generic relationship type that have been defined.

Therefore, our evaluation is that the advantage of this method is its simplicity, the ease of definition of knowledge specific with a realization and the availability of some level of predefined general knowledge. This makes it very appropriate for conceptual schemas with only one realization of a generic relationship type, or when there are several realizations of a generic relationship type, but there is no need of general knowledge related to it or to each realization, besides the implicit one.

5.4 Metaclass

This is the preferred method in research works and in advanced information systems. The idea is to define a meta relationship type for each generic relationship type, which is instantiated for each realization. Thus, we have, as in the previous methods, a relationship type for each realization, but now all realizations of the same generic relationship type are instance of the same meta relationship type.

In the literature, this representation method is usually associated with the Telos language, although other languages could serve as well (see, for instance [KaSc95]). [Mots93] describes the application of the method to the *PartOf*; and [MoSt95] does the same for *MemberOf*. [Dach98] describes an application of the same method for *Materializes*, which is, however, apparently similar to our MG method, but the overall idea is quite different.

From a conceptual modeling point of view, this method is always better than the previous one, because the knowledge related with a realization is now explicitly defined in the meta relationship type. However, as we have explained above, the amount of knowledge that can be defined at the level of the meta relationship type has its limits.

Therefore, our evaluation is that the advantage of this method is its simplicity of use, the ease of definition of knowledge specific to a realization, and of the knowledge related to each realization. This makes it very appropriate for conceptual schemas with only one realization of the generic relationship type, or when there are several realizations of the generic relationship type, but there is no need of additional knowledge related to it.

6 Conclusions

This paper has focused on the representation of generic relationship types in conceptual schemas. We have proposed two new methods for this representation, called the RS and the MG methods. We have seen that both methods can be useful, depending on the generic relationship type and its realizations. The RS method is appropriate when many realizations require the definition of particular knowledge related to them, different from one realization to the other, whereas the MG method is more appropriate when realizations are similar one another. We have explained in detail the adaptation of our methods to the UML. The adaptation to other conceptual modeling languages should also be possible.

We have compared the proposed methods with the current ones. We have seen that the main difference is that we define explicitly the generic relationship type in the schema, while the current methods define only its realizations. The current methods are then very appropriate when there is not the need to define knowledge at the generic level. When such a need do exist, then our methods could be more appropriate.

The need for knowledge related to a generic relationship type exists in large information systems, in generic information systems that can be adapted to several specific domains, or in information systems with advanced reasoning capabilities. This knowledge can be predefined and be part of a library of conceptual schema fragments, or part of a general ontology. On the other hand, in those contexts there are many generic relationship types, and many specific ones can be considered as realizations of some generic one. We hope then that our methods will be especially useful in those contexts.

Acknowledgements

The author wishes to thank Dolors Costal, Cristina Gómez, Juan Ramón López, Maria-Ribera Sancho, Ernest Teniente, Toni Urpí and the anonymous referees for their useful comments. This work has been partially supported by CICYT program project TIC99-1048-C02-01.

7 References

[Cyco97] Cycorp. " CYC® Ontology Guide", http://www.cyc.com/cyc-2-1/toc.html.

[Dach98] Dahchour, M. "Formalizing Materialization Using a Metaclass Approach", CAiSE 1998, LNCS 1413, pp. 401-421.

[Fowl97] Fowler, M. "Analysis Patterns: Reusable Object Models", Addison-Wesley, 357 p.

[KaSc95] Klass, W.; Schrefl, M. "Metaclasses and Their Application", LNCS 943.

[Matt88] Mattos, N.M. "Abstraction Concepts: The Basis for Data and Knowledge Modeling", ER 1988, pp. 473-492

[MBJK90] Mylopoulos, J.; Borgida, A.; Jarke, M.; Koubarakis, M. "Telos: Representing Knowledge About Information Systems", TOIS 8(4), pp. 325-362.

[MoSt95] Motschnig-Pitrik, R.; Storey, V.C. "Modelling of set Membership: The Notion and the Issues", DKE 16(2), pp. 147-185.

[Mots93] Motschnig-Pitrik, R. "The Semantics of Parts Versus Aggregates in Data/Knowledge Modelling", CAiSE 1993, LNCS 685, pp. 352-373.

[Oliv01] Olivé, A. "Taxonomies and Derivation Rules in Conceptual Modelling", CAiSE 2001, LNCS 2068, pp. 417-432.

[Oliv99] Olivé, A. "Relationship Reification: A Temporal View", CAiSE 1999, LNCS 1626, pp. 396-410.

[OMG01] OMG. "Unified Modeling Language Specification", Version 1.4, September 2001, http://www.omg.org/technology/documents/formal/uml.htm

[PiZD98] Pirotte, A.; Zimányi, E.; Dahchour, M. "Generic relationships in information modeling", Technical Report TR-98/09, IAG-QANT, Université catholique de Louvain, Belgium, December.

[PZMY94] Pirotte, A.; Zimányi, E.; Massart, D.; Yakusheva, T. "Materialization: A Powerful and Ubiquitous Abstraction Pattern", VLDB 1994, pp. 630-641.

[RuJB99] Rumbaugh, J.; Jacobson, I.; Booch, G. "The Unified Modeling Language Reference Manual", Addison-Wesley, 550 p.

[Sowa00] Sowa, J. "Knowledge Representation. Logical, Philosophical and Computational Foundations", Brooks/Cole, 594 p.

[Stor93] Storey, V.C. "Understanding Semantic Relationships", VLDB Journal 2(4), pp. 455-488.

[WaSW99] Wand, Y.; Storey, V.C.; Weber, R. "An Ontological Analysis of the Relationship Construct in Conceptual Modeling", ACM TODS, 24(4), pp. 494-528.

[YHGP94] Yang, O.; Halper, M.; Geller, J.; Perl, Y. "The OODB Ownership Relationship", OOIS 1994, pp. 278-291.