Generic Relationships in Information Modeling *

Mohamed Dahchour† Manuel Kolp‡ Alain Pirotte§ Esteban Zimányi¶

Abstract

Generic relationships are powerful abstraction mechanisms that help in better organizing information during the analysis phase of problem resolution. Recent research in conceptual modeling has studied new generic relationships that naturally model complex situations whose semantics escapes direct representation with classical relationships (generalization, classification, and aggregation). For full benefit, these relationships should be made available in object languages and systems as primitives for developing conceptual models of applications. Unfortunately, only classical relationships are supported by popular object models and existing implementations fail to accurately capture the semantics of other generic relationships. This paper characterizes the common semantics of generic relationships, presents some important generic relationships, and reviews several mechanisms for implementing generic relationships, with a particular emphasis on metaclasses.

Keywords: Object Orientation, Generic Relationships, Metaclass, Conceptual Modeling

1 Introduction

Conceptual modeling is the activity of creating abstract representations of some aspects of physical and social systems and their environment. Conceptual models are typically built in the early stages of system development, preceding design and implementation. But conceptual models can also be useful even if no system is contemplated: they then serve to clarify ideas about structure and functions in a perception of a part of the world.

Advances in conceptual modeling involve narrowing the gap between concepts in the real world and their representation in conceptual models by identifying powerful abstractions allowing to better represent application semantics (see, e.g., [KR94, Mat88, MPM92, Myl98, PMD95, PT88, YM94]). Thus, more powerful conceptual models help improve the mastering of the software development process and the quality of the final products.

Generic relationships in object and semantic models are such powerful abstraction mechanisms. They are high-level templates for relating real-world entities. Well-known generic relationships include generalization, classification, and aggregation. Recent research on conceptual modeling has studied other generic relationships like materialization, ownership, role, grouping, viewpoint, and generation. These generic relationships naturally model phenomena typical of complex application domains whose semantics escapes direct representation with classical relationships.

Over a 30-year period, the field of software development has witnessed a shift of emphasis from issues in programming (e.g., structured programming) to issues in design (e.g., specification) to issues in analysis (e.g., requirement engineering). The object paradigm has demonstrated promises of improvement through data abstraction, encapsulation, modularity, flexibility, extensibility, and reuse.

Object-oriented programming languages have dominated the early years of object technology and their influence still clearly permeates the area. Their treatment of relationships as little more than

*This work is part of the YEROOS (Yet another Evaluation and Research on Object-Oriented Strategies) project, principally based at the University of Louvain. See http://yeroos.qant.ucl.ac.be.

†University of Louvain, IAG School of Management, B-1348 Louvain-la-Neuve, Belgium, e-mail: dahchour@qant.ucl.ac.be

‡University of Toronto, Department of Computer Science, M5S 3G4 Toronto, Canada, e-mail: mkolp@cs.toronto.edu

§University of Louvain, IAG School of Management, B-1348 Louvain-la-Neuve, Belgium, e-mail: pirotte@info.ucl.ac.be

¶University of Brussels, Department of Informatics, CP 165/15, 1050 Brussels, Belgium, e-mail: ezimanyi@ulb.ac.be

pointer-valued attributes has confined relationships to a second-class status in most database management systems and, to a lesser extent, in software development methods. Thus, except for generalization, classification, and a version of aggregation, usual object models do not directly support generic relationships. Consequently, users are left with *ad hoc* implementation techniques, like pointers or references, with problems of dispersion and duplication of information among several participants. Still, it has been clear for some time (see, e.g., [Rum87]) that promoting relationships to a more independent status provides benefits concerning reusability, ease of modeling, independence from implementation choices of specific systems, control of redundancy, and facility of updates and maintenance.

The rest of the paper is organized as follows. Section 2 analyzes the concept of generic relationships and characterizes their common semantics. Section 3 is a review of some important generic relationships that illustrates how they contribute to enhance the expressiveness of information models. Section 4 discusses issues concerned with the identification and definition of new generic relationships. Section 5 reviews several mechanisms for implementing generic relationships, namely pointers or references, layers, parameterized classes, and mainly metaclasses. Metaclasses allow to capture structure and behavior associated with a generic relationship independently of specific application classes participating in it. The semantics is defined once and for all in a structure of metaclasses, that provide for defining and querying the relationship, creating and deleting instances of participating classes, and so on. The interaction of generic relationships is also discussed, that is, how to handle the participation of the same class in several relationships. Section 6 summarizes and concludes the paper.

2 Generic Relationships

2.1 Relationships

A relationship models an association between two or more real-world entities. The number of object classes involved in a relationship is the *degree* of the relationship. Most relationships have a small degree, i.e., they are binary or ternary. This reflects the concept formation strategy of human thinking.

Relationships at the conceptual level are not oriented, e.g., binary relationships are bidirectional. Roles are sometimes defined for privileging traversal paths: a binary relationship is thus viewed as two equivalent unidirectional relationships. Roles become essential when the same class participates several times in the same relationship.

Instances of relationships are also called *tuples* in the relational model and *links* in object models. A *link* thus represents an association among several real-world objects. Similar links are abstracted as *relationships* (or relationship types or classes) among classes in the same way that similar objects are abstracted into classes.

2.2 Generic relationships

Relationships can be classified as *generic* or *specific*. *Generic* relationships are high-level templates for relating classes. Well-known examples are generalization (of pattern Superclass ◁—Subclass), classification (of pattern Class <- - -Instance), and aggregation (of pattern Whole ◇—Part).

A *specific* relationship is a realization of a generic relationship in a particular application, like Vehicle◁—Car (a specific generalization) or Car◇—Body (a specific aggregation). Thus, a generic relationship abstracts all specific relationships of a given kind and it can be viewed as a metarelationship.

A number of other generic relationships have been studied. The following ones are reviewed in more detail in Section 3:

- *materialization* relates a class of categories (e.g., models of cars) and a class of more concrete objects (e.g., individual cars);
- *ownership* relates an owner class (e.g., persons) and a property owned (e.g., cars);
- *aggregation* forms composite objects (e.g., car) from component objects (e.g., body and engine);
- *role* relates an object class (e.g., persons) and a role class (e.g., employees), describing dynamic states for the object class;
- *grouping* relates a member class (e.g., players in a team) and a grouping class (e.g., teams);
- *viewpoint* represents partial information about a class viewed from a particular standpoint;
- *generation* represents processes that lead to the emergence of new output entities from input entities.

This list is not exhaustive. For example, the following relationships have not been included, for lack of space: versioning [AHLP95, AHPZ96, Kat90], that relates an object class and its time-varying versions, modeling various states of the object class; realization [LM96], a variant of classification that allows an object to add structure to that defined by its class; transition [HG91], that forms (target) object classes from other (source) object classes by changing instances of the source class into instances of the target class.

Notations. We use UML notations to specify classes, objects/instances, associations, generalization, instantiation, and aggregation. We add the notations " $\cdots\succ$", " $\rightarrow\circ$", " $\twoheadrightarrow$", " $\text{—}\!\!-\!\!\odot$", and " $\Longrightarrow$" to specify, respectively, the *ownership*, *role*, *grouping*, *viewpoint*, and *generation* relationships. Also, we find more intuitive and expressive to the traditional (min,max) notations for cardinalities (e.g., (0,n), (1,1)) than those of UML.

2.3 Semantics of generic relationships

Generic relationships have several important characteristics:

- **Class and instance semantics**. The semantics of generic relationships concerns both classes and instances of these classes. Consequently, comprehensive semantics must deal with both the *class level* and the *instance level* in a coordinated manner.
  As an example, the semantics of generalization at the class level states that:
  - a class can have several superclasses and several subclasses;
  - subclasses are mutually exclusive;
  - each class inherits all properties and methods of its superclasses;
  - conflicts induced by multiple inheritance are avoided with a specified strategy;
  - each class has a (1,1) cardinality regarding each of its superclasses and a (0,1) cardinality regarding each of its subclasses.

  At the instance level, the generalization relationship expresses the following semantics:
  - an instance of a class is also an instance of all its superclasses;
  - an instance of a class A cannot be an instance of another class that is not direct or indirect superclass of A[1];
  - an instance cannot have additional properties than those of its class[2].
- **Cardinality** constrains the number of objects related by a relationship. Cardinalities are usually represented as pairs of integers associated with each class participating in the relationship; they indicate the minimum and the maximum number of instances of the relationship that each object of the class can participate in. For instance, for generalizations: SubClass (1,1)—▷SuperClass (0,1), for materializations: AbstractClass (0,n)—∗ConcreteClass (1,1), for roles: RoleClass (1,1)→∘ObjectClass (0,n). In the above examples, the cardinality (0,n), where n stands for an arbitrarily large integer, can be further constrained by application semantics. Various dependencies with special names (e.g., mandatory or optional participation) are based on special values of minimal or maximal cardinalities.
- **Composition**. Generic relationships can be involved in compositions, where a class plays several roles of the same generic relationship R in several specific relationships based on R, as schematized in Figure 1(a). An example of composition of generalizations is Person ◁—Student ◁—GraduateStudent, where Student is at the same time a superclass of GraduateStudent and a subclass of Person. Similarly, in the composition of aggregations Car ◇—Body ◇—Door, Body is at the same time a composite of Door and a component of Car.
- **Transitivity** can follow from composition. For example, generalization is transitive: the generalizations Person ◁—Student ◁—GraduateStudent imply Person ◁—GraduateStudent. However, not all generic relationships that can be composed are transitive. For example, aggregation is not transitive in general.

[1] This is possible in models allowing multiple classification, like Telos [MBJK90], where an object can be an instance of several classes not related, directly or indirectly, by the generalization link.

[2] This restriction is overcome with the realization relationship [LM96].

- **Multiplicity.**[3] A role in a generic relationship is said to be multiple if the same class can participate with that role in several realizations of the generic relationship. Figure 1(b) shows an example of multiplicity for role r_1. Most generic relationships allow multiplicity in each role. For example, with generalization, a class can have several superclasses and several subclasses. Also, with aggregation, a composite can have several components and a component can be part of several composites.

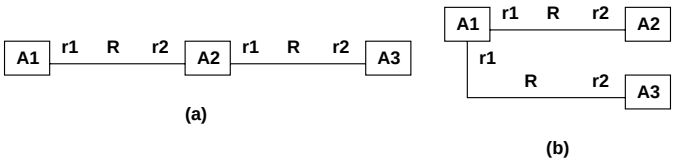

Figure 1: Composition (a) and multiplicity (b) of relationships.

- **Exclusiveness and sharing** concern generic relationships with the multiplicity property for a role. In Figure 1(b), role r_1 is said to be *exclusive* if relationship R enforces the restriction that an instance of A_1 can be related to only one instance of A_2 or A_3. Role r_1 is *shared* if relationship R puts no restrictions on the number of instances of A_2 and A_3 that a given instance of A_1 can be related to via R.
  For example, in the aggregations Proceedings◇—Article—◇Journal and Newspaper◇—Article—◇Compilation, Article plays an exclusive role in the former and a shared role in the latter, if the same article cannot appear both in conference proceedings and in a journal, while it can appear at the same time in a newspaper and in a compilation.
- **Existence dependency** characterizes whether or not an object can exist independently of related objects. Existence dependency reexpresses cardinality information by stating whether or not the existence of some objects depends on the presence of other objects. Existence dependency is often formulated in terms of propagation of deletion operations. Typical cases include the following:
  - the deletion of an object has no effect on related objects;
  - the deletion of an object implies the deletion of related objects even if they are involved in other relationships;
  - the deletion of an object implies the deletion of related objects only if they are not involved in other relationships;
  - the deletion of an object is prohibited if it is related to at least one object that is not explicitly deleted.
- **Attribute inheritance and value propagation.** Most generic relationships allow propagating structure and behavior from one participant to another. This is carried out by *inheritance* or by *delegation* [Lie86]. In some cases, propagation is unidirectional. For example, in generalization, subclasses inherit attributes and methods from superclasses. In aggregation, composites can access some properties of their parts (and vice versa) by delegation; for example, Car inherits the color attribute of its component Body.

3 A Review of Some Generic Relationships

This section presents some generic relationships. In the figures, classes are drawn as rectangular boxes and instances as rectangular boxes with rounded corners. Classification links appear as dashed arrows and generalization links as solid arrows.

3.1 Materialization

Materialization [GS94, PZMY94] is a binary relationship with pattern Abstract—∗Concrete relating a class of categories to a class of more concrete objects. In the example of Figure 2, CarModel is the abstract class of the materialization and Car is its concrete class. CarModel represents information typically displayed

[3]This definition of *multiplicity* should not be confused with its use, by some authors, as a synonym of *cardinality*.

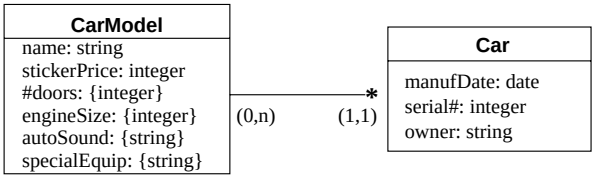

Figure 2: An example of materialization.

in the catalog of car dealers, while class Car represents information about individual cars. Figure 3 shows an instance FiatRetro of CarModel and an instance Nico's Fiat of Car, of model FiatRetro.

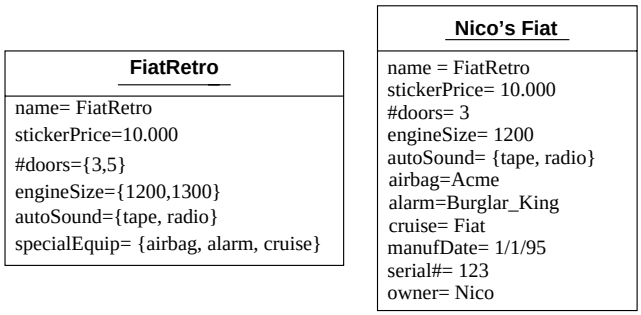

Figure 3: Instances of CarModel and Car classes from Figure 2.

Cardinality. Intuitively, the materialization CarModel—∗Car means that every concrete car (e.g., Nico's Fiat) has exactly one model (e.g., FiatRetro), while there can be any number of cars of a given model. Most real-world examples of materialization have cardinality (1,1) on the side of the concrete class and cardinality (0,n) on the side of the abstract class, although the latter cardinality can be further constrained.

Attribute propagation. Some information in a concrete instance is naturally viewed as obtained from its associated abstract instance. As illustrated in Figure 3, attributes of instances of CarModel and Car can be related in three different ways.

- Nico's Fiat directly inherits the name and stickerPrice of its model FiatRetro; this is called Type1 attribute propagation.
- Nico's Fiat has attributes #doors, engineSize, and autoSound whose values are selections among the options offered by multivalued attributes with the same name in FiatRetro; this is called Type2 attribute propagation. For example, the value {1200,1300} of engineSize for FiatRetro indicates that each FiatRetro car comes with either engineSize = 1200 or engineSize = 1300 (e.g., 1200 for Nico's Fiat).
- The value {airbag, alarm, cruise} of attribute specialEquip for FiatRetro means that each car of model FiatRetro comes with three pieces of special equipment: an airbag, an alarm system, and a cruise control system. Thus, Nico's Fiat has three new attributes named airbag, alarm, and cruise, whose suppliers are, respectively, Acme, BurglarKing, and Fiat. Other FiatRetro cars might have different suppliers for their special equipment. This mechanism is called Type3 attribute propagation.

In addition to those attributes propagated from FiatRetro, Nico's Fiat has a value for attributes manufDate, serial#, and owner of Car.

Composition. Materializations can be composed in hierarchies, where the concrete class of one materialization is also the abstract class of another materialization. The example of Figure 4 deals with theater Plays written by an author, with a title and a set of main roles. Plays materialize as Settings

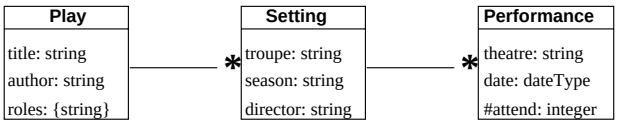

Figure 4: Composition of materializations.

that embody the production decisions for a theatrical season: a troupe, a director, and a set of actors for each role of each play at the repertoire of that season. Settings materialize in turn as Performances, at a particular date, with each role of Play assigned to a specific actor for each Performance. Such compositions of materializations are transitive. Thus, in the example, Play —∗Setting —∗Performance implies Play —∗Performance.

Abstract classes can materialize into several concrete classes. For example, in a movie rental application, class Movie materializes independently into classes VideoTape and VideoDisc, i.e., VideoTape∗—Movie—∗VideoDisc. Also, concrete classes can be materializations of several abstract classes, as in Play—∗Movie∗—Novel.

Dependency. In a materialization, the deletion of an abstract instance induces the deletion of its associated concrete instances. In the materialization of Figure 2, when model FiatRetro is deleted, all instances of Car associated to that model (e.g., Nico's Fiat) are also deleted.

On the other hand, the deletion of a concrete instance induces the deletion of its associated abstract instance only if the minimal cardinality of the abstract side of the materialization is 1. For example, if, in CarModel—∗Car, the minimal cardinality of CarModel is 1, meaning that a car model has at least one concrete car associated to it, then the deletion of the last car of a model implies the deletion of that model.

Materialization semantics. The semantics of materialization is defined as a combination of generalization, classification, and of a class/metaclass correspondence. This is expressed as a collection of *two-faceted constructs*, each one being a composite structure comprising an object, called the *object facet*, and an associated class, called the *class facet*.

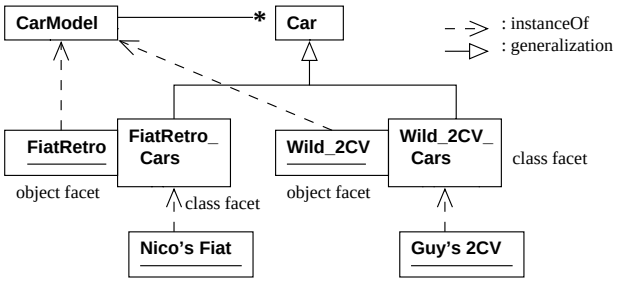

Figure 5: Semantics of the materialization in Figure 2.

As shown in Figure 5, FiatRetro, an instance of CarModel, is an object facet whose associated class facet FiatRetro_Cars, a subclass of Car, describes all instances of Car with model FiatRetro. Wild2CV is another instance of CarModel whose associated class facet is Wild2CV_Cars.

Thus, the semantics of materialization induces a partition in the instances of the concrete class Car into a family of subclasses (such as FiatRetro_Cars and Wild2CV_Cars), each corresponding to an instance of the abstract class. These subclasses inherit attributes from the concrete class through the classical inheritance mechanism of generalization. In addition, attributes of the object facet are propagated to the associated class facet in three different ways as explained above. Concrete instances, such as Nico's Fiat and Guy's 2CV, with attribute values propagated from their corresponding instance of CarModel, are ordinary instances of the subclasses of Car.

Of course, only application classes, like CarModel and Car, appear in conceptual schemas. The two-faceted construct machinery, making explicit the semantics of materialization and the propagation of

attributes, is invisible to external users. For users, Nico's Fiat is an instance of Car, with an instantiation mechanism that integrates attribute propagation.

3.2 Ownership

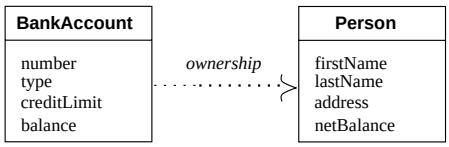

Figure 6: An example of ownership.

Ownership [HPYG95, YHGP94] is a binary relationship with pattern Property$\cdots\succ$Owner relating an owner and a property that is owned. In the example of Figure 6, Person is the owner class and BankAccount is the property class.

Intuitively, ownership means that the owner of a property has certain rights on the property. Various shades of ownership express the intuitive semantics of the relationship. Thus, the owner can be a person or a legal entity (e.g., a corporation or an organization). Property ownership can be temporary or permanent. A property can be *real* (e.g., a piece of land), *intellectual* (e.g., an idea, a creative work, a patent), or *personal*.

Cardinality. In general, ownership constrains a property to have at least one member whereas an owner may have 0 or several properties. In the example of Figure 6, a person can have (0,n) bank accounts and a bank account can have (1,n) owners.

Value propagation. Some features of a property are naturally viewed as features of its owner or vice versa. For example, the address of persons may be modeled as the address of their house rather than as an attribute of persons. Likewise, the name on a passport can be modeled as the name of the passport owner. In the former case, the value of address is propagated *upwards* from the property to the owner. In the latter case, the value of name is propagated *downwards* from the owner to the property. In a *transformational* value propagation, the value of the propagated attribute results from a transformation of values from several objects linked through ownership. For example, in Figure 6, attribute netBalance of a person is computed as the sum of the balance of the person's bank accounts.

Composition. Ownership can be composed where the property of one ownership is also the owner of another ownership, as in Corporation $\prec\cdots$Division $\prec\cdots$Factory.

Ownership allows multiplicity for both the property and the owner roles. Thus, owner classes can own several properties as in Vehicle$\cdots\succ$Person$\prec\cdots$House. Also, a property class can be owned by several owners as in Person$\prec\cdots$Stock$\cdots\succ$Company. Ownership can be recursive: for example, a company can own other companies.

Exclusiveness. Ownership can be *exclusive* or *joint*, i.e., a property may be owned by one owner or shared by several owners. Person$\prec\cdots$Retirement_Portfolio is an example of exclusive ownership.

There are two types joint ownerships. *Free joint* ownership states no explicit partition of the rights of the joint owners in the property. For example, a joint bank account is freely shared by a couple. In *percentage joint* ownership, each owner takes a percentage of the ownership, e.g., when husband and wife each own 50% of their house. An *equal joint* is when all owners have the same percentage. As noted in [HPYG95], percentage joint is unique to ownership, while exclusiveness also concerns other generic relationships,

Dependency. The deletion of a property can cause the deletion of its owner. For example, suppose that an insurance company distinguishes people who own cars from people who do not. This can be modeled by a class Person with a subclass CarOwner and an ownership CarOwner$\prec\cdots$Car. In this case,

the car owner is dependent on the car, i.e., if a car owner owns only one car and this car is deleted, then the car owner should be deleted from class CarOwner (but not from class Person).

An example of dependency of the property on the owner is an ownership Employee$\prec\cdot\cdot$Car with an additional constraint stating that, when employees stop working for the company, the information about their cars is no longer needed.

3.3 Aggregation

Aggregation (e.g., [HGP98, KBG89, KP97, MP96, MPK96, WCH87]) is a binary relationship with pattern Composite$\diamond$—Component by which a relationship between objects is considered a higher-level (aggregate) object. Other names for aggregation are part-whole or simply part relationship. For example, Figure 7 shows two aggregations between composite Newspaper and components Editorial and Article.

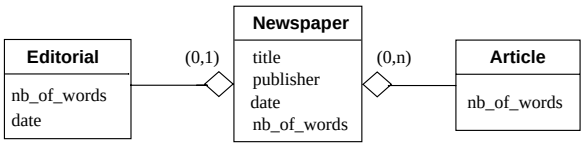

Figure 7: Examples of aggregation.

Cardinality. For the composite role, cardinality determines how many components can be grouped together to form a composite. For example in Figure 7, a newspaper is composed of (1,n) articles. For the component role it specifies the number of composites that a component can be part of. This is related to exclusiveness (see below).

Value propagation. Some features of a whole are viewed as features of its parts and vice versa. Thus, there are two kinds of value propagation: *upward* propagation from the part class to the whole class and *downward* propagation from the whole class to the part class. For example, in Car$\diamond$—Body, the color of a car can be propagated upwards from its corresponding body. Similarly, in Newspaper$\diamond$—Article, the date of articles is propagated downwards from their corresponding newspaper. Furthermore, the value of a propagated attribute can be obtained as a combination of values from several source objects. For instance, the color of a car's body can be defined as some combination of the colors of its panels.

Composition. Aggregations can be composed in hierarchies, where the component class of one aggregation is also the composite class of another aggregation as in Building$\diamond$—Room$\diamond$—Wall.

Aggregation allows multiplicity for both the composite and the component roles. Figure 7 shows an example of multiplicity for the composite role. An example of multiplicity for the component role is Journal$\diamond$—Article—$\diamond$Compilation where an article can be included in a journal or in a compilation. Also, aggregations can be recursive: a typical example is that of part-subpart in assemblies, where parts are composed of other parts.

Aggregation is not transitive in general. A counterexample is that Hand —$\diamond$Musician —$\diamond$Orchestra does not imply Hand —$\diamond$Orchestra.

Exclusiveness. Aggregations can be exclusive and shared. A *shared* aggregation puts no restrictions on the number of composites that a given component can be part of, allowing the component to be shared. An example is Compilation$\diamond$—Article if the same article can be included in any number of compilations.

An *exclusive* aggregation enforces the restriction that a given component can be part of only a single composite. Exclusiveness is quite natural with physical assemblies. Thus, in Car$\diamond$—Engine, two cars cannot share the same engine. This kind of exclusiveness is called *class exclusiveness* as it enforces the exclusive reference constraint within a single class. It is also the case that a car and, say, an airplane cannot share the same engine. This type of exclusiveness is called *global exclusiveness* since it bears on the entire database.

Dependency. The lifetime of parts sometimes depends on that of their wholes and conversely. The *part-to-whole* dependency means that the existence of a part depends on the existence of the corresponding whole, that is, that the deletion of the whole implies the deletion of the part. As an example, Journal◇—Article is part-to-whole dependent if the deletion of a journal implies the deletion of its articles. The *whole-to-part* dependency means that the existence of a whole depends on the existence of the corresponding part, that is, the deletion of the part implies the deletion of the whole.

3.4 Role Relationship

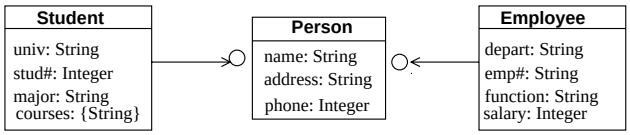

Figure 8: Examples of roles.

Role [ABGO93, CZ97, KZ98, EPdOVdC+94, GSR96, RHS95, WDJS95, WCL97] is a binary relationship with pattern ObjectClass○←RoleClass relating a class of *evolving* objects and a class of roles that those objects can play. Figure 8 shows two role relationships relating an object class Person, and role classes Student and Employee. In a role relationship, the object class defines *permanent* properties of an object while each role class defines a set of properties characterizing a particular aspect in which an object can be viewed during its lifetime. The idea is that the role relationship captures the temporal and evolutionary aspects of real-world objects, while the usual generalization relationship deals with their static aspects. Thus, while classes Male and Female may be linked to Person via generalization links, Student and Employee would rather be linked to Person via role links.

Figure 9 shows, on some instances of the schema of Figure 8, how persons may gain and/or lose roles. In Figure 9(a), John_p is created as instance of Person. Figure 9(b) shows an instance John_s of Student, related to John_p by the role relationship, expressing that John_p has become a student. Both instances John_p and John_s coexist with different identifiers. If John leaves the university, the instance John_s will just be removed. In Figure 9(c), John_p gained an additional role John_e, i.e., he became an employee.

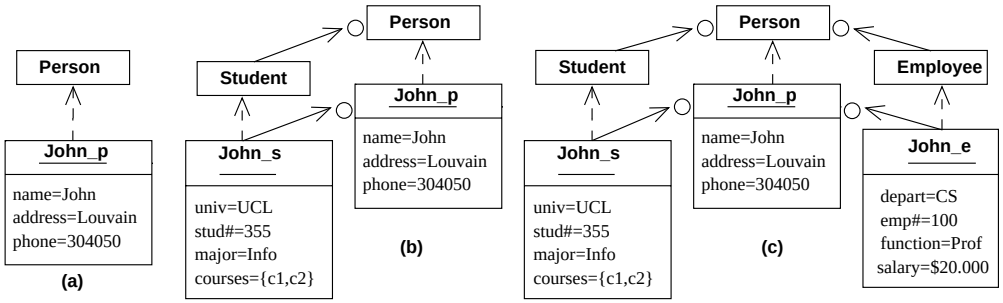

Figure 9: Various roles of an object.

Cardinality. Each instance of a role class (e.g., Student) is related to exactly one instance of its object class (e.g., Person) but, unlike generalization, each instance of the object class can be related to any number of instances of the role class, depending on the maximal cardinality at the side of the object class. For example, John can be at the same time a student in more than one university and an employee in more than one department.

Value propagation. Role classes are not introduced for sharing information. This should rather be the responsibility of subclassing. If Student is viewed as a subclass of Person, it inherits all properties and methods from Person. Viewed as a role class of Person, Student does not inherit properties and methods

of Person. Instead, instances of role classes access properties and methods of their corresponding objects by delegation.

Composition. Role relationships can be composed in hierarchies, where the role class of one role is also the object class of another role. For example, Figure 10 shows a role class Employee with two role classes Professor and UnitHead.

The role relationship allows multiplicity for both the object and the role classes. Figure 8 shows an example of multiplicity for the object class. An example of multiplicity for the role class is Student$\circ\leftarrow$Councilor$\rightarrow\circ$Faculty where both students and faculty can play the role of councilors in the university council. However, the role relationship cannot be recursive.

The role relationship is transitive: for role classes R_1, R_2, and object class O, $R_1\rightarrow\circ R_2\rightarrow\circ$O implies $R_1\rightarrow\circ$O.

Dependency. The lifetime of roles depends on that of objects playing those roles. Thus, the deletion of an object induces the deletion of its associated roles. In Figure 9(c), if object John_p is deleted, its associated roles John_s and John_e are also deleted. Also, the deletion of a role can induce the deletion of its associated object if the minimal cardinality of the object class is 1. For example, if Person (1,n)$\circ\leftarrow$(1,1)Employee, meaning that a person plays at least once the employee role, then the deletion of the last employee role implies the deletion of that person.

Modeling with roles. Roles address three kinds of issues arising when modeling evolving entities with traditional object models:

(1) *Object migration*, i.e., objects dynamically changing their classification. For example, upon graduation, a person ceases to be a student and becomes an alumnus. An *evolution* is the case where the transition object ceases to be an instance of the source class, while in an *extension*, the object remains an instance of the source class.
(2) *Multiple instantiation of the same class*, i.e., an object becoming an instance more than once of the same class, while losing or retaining its previous membership. For example, a student may be registered in two different universities.
(3) *Context-dependent access*, i.e., the ability to view a multi-faceted object in particular perspectives. For example, person John can be viewed as an employee or as a member of the golf club.

Interaction between generalization and role. The same class C can be at the same time a role class for another class O_1 (i.e., C$\rightarrow\circ O_1$) and a superclass for class O_2 (i.e., C$\triangleleft$—O_2).

Figure 10 illustrates interactions between generalization and role relationships. Class Person is subclassed by classes Male and Female since, e.g., an instance of Male cannot migrate to class Female. Class Male has a role class Draftee accouting for males serving on military duty. Class Person is refined by role classes Student and Employee. Student is subclassed by ForeignStudent and CountryStudent, since an instance of ForeignStudent cannot become an instance of CountryStudent nor conversely. Class CountryStudent gathers together those students whose original nationality is that of the host country. Class Employee has two role classes, Professor and UnitHead. This is an example of compositions of roles.

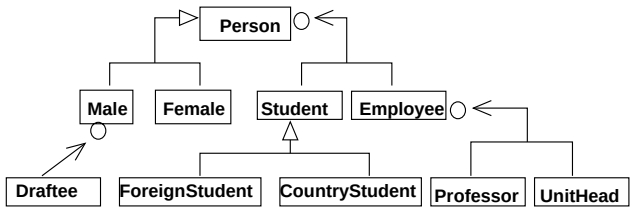

Figure 10: Generalization and role hierarchies.

Role control specification concerns how objects may evolve by gaining and/or losing roles. The notion of *meaningful combination* of role classes, inspired from the migration control of [LD94], allows to constrain the legal combinations of roles for an object. For example, if Person has roles Employee and Retired, employees can acquire the role of retirees, but not the converse. Several styles of such object life cycle diagrams are discussed in [WDJS95].

Transition predicates (similar to the predicates associated with category classes [Odb94]) describe necessary and/or sufficient conditions on how objects playing roles of R_1 may explicitly or automatically acquire roles within R_2, if (R_1, R_2) is a meaningful combination of roles. For example, for Teenager→∘Person∘←Adult, a transition predicate age≥18 associated with Teenager means that when an instance of Teenager satisfies the predicate, it becomes an instance of Adult.

3.5 Grouping

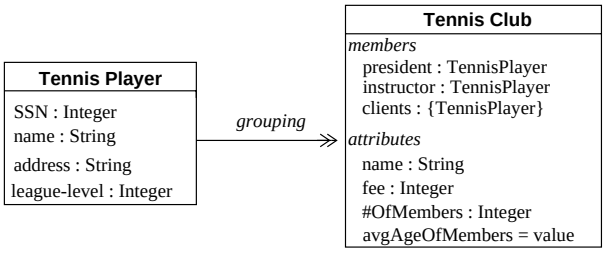

Figure 11: An example of grouping.

Grouping [Bro81, MPS95], with pattern MemberClass→↠SetClass, is a generic relationship by which a collection of set members is considered as a higher-level set object. Figure 11 shows an example of grouping between a member class TennisPlayer and a set class TennisClub.

The SetClass in a grouping has a *set-determining* attribute, whose value is the set of members. It has a number of *set-describing* attributes and/or constraints.

In Figure 11, SetClass TennisClub gathers three attributes grouped under label members. Label attributes gathers two ordinary attributes, name and fee, and two set-describing attributes, #OfMembers holding the current number of club members and avgAgeOfMembers holding the average age of club members.

The concepts of grouping and the set concepts differ in that a grouping explicitly deals with properties and constraints of members as a whole in addition to set membership. Thus, whereas two sets are equal if and only if they have the same members, two groupings with the same members (e.g., two tennis clubs) may differ in the value of some property associated with the grouping, such as the minimum age required to be a member of the club.

Cardinality. In general, the grouping relationship constrains a set class to have at least one member whereas the participation of a member in the grouping may be optional or mandatory. For example, the cardinalities in Person(0,1)→↠(1,n)PoliticalParty mean that a person may be a member of at most one political party and that a political party has at least one member.

Value propagation. Grouping can be seen as a kind of aggregation. Hence, both upwards and downwards propagation are possible, although this is not clear from [Bro81] nor from [MPS95].

Composition. Grouping can be composed in hierarchies, where the member class of one grouping is also the set class of another grouping. An example is TennisPlayer →↠TennisClub →↠TennisFederation.

Grouping allows multiplicity for both the member and the set classes. Examples are, respectively, TennisClub↞←Employee→↠TradeUnion and Person→↠Sponsors↞←Organization.

Grouping is normally not transitive. For example, Book →↠Library →↠AcademicInstitutions does not imply Book →↠AcademicInstitutions.

Exclusiveness. Members can be exclusive and shared. A *shared* member can participate in any number of groupings at the same time. Thus, for example, members in TennisClub ↞TennisPlayer and in TennisClub ↞Employee ↠Department are shared if an employee can be a member of several tennis clubs, and a member of department and a tennis club at the same time, respectively.

An *exclusive* member can only participate in a single grouping at a time. This is the case in Person ↠PoliticalParty, if two political parties cannot simultaneously "share" the same member. and in AcademicStaff ↞UnivStaff ↠TechnicalStaff, if a staff member cannot be a member of both the technical staff and the academic staff.

Dependency. In general, the lifetime of members does not depend on that of their groupings and conversely. However, a dependency may be implied by the cardinality constraints. For example, in Employee(1,1)↠(1,20)Department, due to the cardinality (1,1), the deletion of a department implies the deletion of its employees.

Member covering specifies whether or not all instances of the member class are necessarily related to an instance of the grouping class.

Partial covering means that there is at least one member that does not belong to any grouping. For example, in Employee↠TennisClub, not all employees must be members of the tennis club.

Complete covering means that the grouping provides a complete covering of the member class. For example, if TennisPlayer is a subclass of Employee, then in TennisPlayer↠TennisClub the grouping class TennisClub covers all instances of the member class TennisPlayer.

3.6 Viewpoints

Viewpoint [Ber92, FKN+92, MPM96] is a binary relationship of pattern Class—⊙View relating a class and a particular view on it according to a specific perspective. The viewpoint mechanism appears in various fields of computer science under several terms [Myl98]. In software development, it appears under the guises of *workspaces*, *versions*, and *configurations* [Kat90] for the support of cooperative work. In the database field [AB91, Ber92], the *view* mechanism is used to present partial data to different user groups. In hypertext bases [PT93] and more general information bases [MPM96, MMP95], the notions of *perspectives* and *contexts* have been used to decompose an information base in different sub-bases. In requirement engineering [FKN+92], viewpoints are used for structuring, organizing, and managing system specifications and the development process. When viewpoints are used to decompose a base system according to a given criterion, as in [FKN+92, MPM96, MMP95], the viewpoint mechanism has the pattern BaseSystem $\overset{criterion}{—\!\odot}$ SubSystem.

In the sequel, we focus only on the viewpoint mechanism in object database systems as proposed in [GBCGM97]. According to [GBCGM97], a database schema is composed of:

- a *base schema*, that is, a collection of classes;
- a set of *schema views* derived from the base schema, to provide applications with a customized view of data, one schema view for each application. A schema view is thus a collection of special classes called *views*;
- *derivation relationships*, both at the class and the schema level, relating each view to the classes (or views) it is derived from and relating each schema view to the base schema (or schema view) it is derived from.

The rest of this section concentrates on the derivation relationship between a class and its view, that is, the viewpoint mechanism of pattern Class —⊙View. Views are subcategorized as *object-generating* or *object-preserving*. The evaluation of a query associated with an object-generating view always returns new objects, with new oids (for example if the view query contains a join operation). By contrast, the evaluation of a query associated with an object-preserving view populates the view by extracting existing objects from a class or a view, possibly, modifying their structure and behavior (for example if the view query is a selection or projection operation). The view instances preserve the identifiers of base objects, instead of generating new identifiers as required for object-generating views.

As an example, Figure 12 shows an object-preserving view Employee derived from class Professor and an object-generating view Planning derived from classes Professor and Room. Note that an employee John

who works as a professor is an instance of Professor, while an instance of Planning is a new object derived from an instance of Professor and an instance of Room.

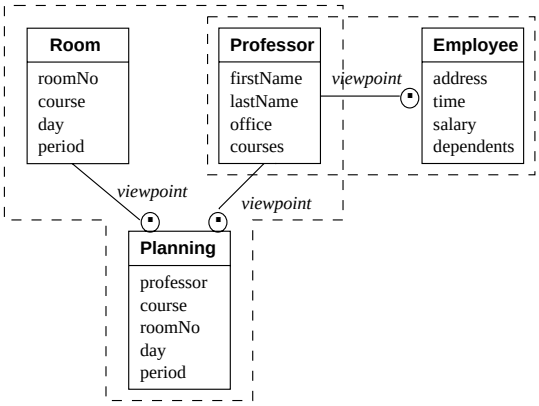

Figure 12: An example of object-preserving view and object-generating view.

Cardinality. In general, the cardinality of a class with respect to its views is (0,n) for object-generating views and (0,1) for object-preserving views. For example, in Professor—⊙Employee a professor John may appear as employee at most once. By contrast, the cardinality of a view regarding each of its corresponding classes is always (1,1). For example, an employee John corresponds to exactly one instance of Professor.

Attribute propagation. A view, be it object-preserving or object-generating, inherits structure and methods from its defining classes or views. However, a derived view could be considerably different from the base class (or view). Thus, new attributes/methods can be added, existing attributes/methods can be hidden, the name of an attribute/method of a class can be changed, the domain of an attribute of a class can be changed.

Composition. Viewpoints can be composed in hierarchies, where the view of one derivation relationship is also the base class of another derivation. As an example the Employee view in Professor—⊙Employee can be refined into the view MarriedEmployee. Furthermore, classes or views can be refined into several views and conversely a view can be derived from more than one class or view. For example, the Employee view can be refined into MarriedEmployee and SingleEmployee, and the view MarriedEmployee can be derived from both Employee and Couple.

Exclusiveness. For both object-preserving and object-generating views, two distinct instances of the same class (or view) cannot share the same instance of a derived view. In particular, for object-generating views that can be derived from more than one class (or view) such as Room—⊙Planning⊙—Professor of Figure 12, an instance of the derived view Planning is necessarily shared by corresponding instances of the base classes Professor and Room.

Dependency. The lifetime of view instances necessarily depends on that of their defining objects. Thus, the deletion/modification of an object, instance of a base class induces the deletion/modification of its derived objects, that are instances of the views derived from that base class. In Figure 12, when a room is deleted from class Room, all the plannings referring to it should be deleted from class Planning.

Generalization versus Viewpoint. The view derivation relationship is orthogonal to generalization. Views can be generalized in superviews in the same way that classes can be generalized in superclasses. The main difference between the generalization hierarchies on ordinary classes and on views is that the latter must take into account the query component of the view. This is due to the fact that, unlike a class, a view is not explicitly instantiated; rather its population is derived from the population of its root

classes. Thus, the restriction must be imposed that the query of a view must be subsumed by the queries of its superviews.

3.7 Generation

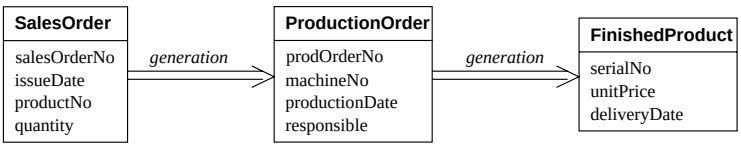

Figure 13: Examples of generation.

Generation [GH92] is a binary relationship of pattern InputClass$\Longrightarrow$OutputClass with the meaning that an instance (or a set of instances) of the input class generate(s) an instance (or a set of instances) of the output class. In the example of Figure 13, sales orders generate production orders which in turn generate finished products.

In a generation, the input objects can be preserved or consumed. A *transformation*, denoted by InputClass$\stackrel{\tau}{\Longrightarrow}$OutputClass, occurs when all the input instances are consumed in the generation process.

A *production*, denoted by InputClass$\stackrel{\rho}{\Longrightarrow}$OutputClass, takes place when all the input instances survive the generation process. The example of Figure 13 shows two productions. Notice that it is also possible that in a generation some objects are preserved while others are consumed.

The generation process, under both its transformation and production aspects, frequently appears in chemical application domains where products are consumed to generate new ones or are only used as a catalyst in a chemical reaction. It appears frequently also in production planning domains.

Cardinality. In general, generation requires that instances of the output classes be related to at least one instance of the input classes. For instance, the generation ProductionOrder(0,n) $\Longrightarrow$(1,n)FinishedProduct of Figure 13 means that every finished product results from at least one production order while a given production order may give rise to zero (when an order production fails) or several finished products.

Attribute propagation. Values of some attributes of output entities depend on those of input entities. For instance, in chemical processes, the quantity of generated products is determined from that of input products. Also, in the generation of Figure 13, the date of execution of a production order could determine the date of delivery of a finished product.

Composition. Generations can be composed in hierarchies, where the output class of one generation is also the input class of another generation, as shown in Figure 13. Generations can be recursive. For example, in a cadastral application, a parcel may be split generating new smaller parcels or, on the contrary, several parcels may be merged and generate a larger one. This can be modeled by a class Parcel with a recursive generation.

Dependency. Unlike other generic relationships, for generation process the dependency of output objects on their input objects and conversely does not hold, because output objects are *new* autonomous products.

M:N generations. Most real-world reactions and production planning applications involve M objects which interact to breed N other objects. An *M:N* generation, denoted $\{i_1,\ldots,i_m\}\stackrel{\tau}{\Longrightarrow}\{o_1,\ldots,o_n\}$, is viewed as a 1:1 generation G, with a complex input object, I, aggregated from the objects which are input into the production process, and a complex output object, O, aggregated from the objects which emerge from the generation (i.e., $G \equiv I\stackrel{\tau}{\Longrightarrow}O$). In general, G, may be any kind of generation, although it is usually a transformation. As an example, in the transformation process {woodA, woodB, catalyst}$\stackrel{\tau}{\Longrightarrow}${paper, pulp, catalyst}, both pulp and paper are generated from three input products: two different types of wood chips woodA and woodB, and a catalyst which survives the reaction.

Reversibility. Reversible generations require that no changes take place in the future which would prevent the reversal of the generation. Irreversible generations, however, mean that the input objects cannot be reached from the output objects. For example, if a mill order cannot be completed at one mill due to machine breakdown, the conversion of the original sales order to the mill order can be *reversed*, the current mill order invalidated, and a new mill order created to be executed by a different mill. However, once salt is produced, the process cannot be reversed to yield hydrochloric acid.

4 New Generic Relationships

4.1 Identification

Conceptual modeling focuses on capturing and representing certain aspects of the real world relevant to the functions of an information system. The central constructs in the building process of conceptual models are *entities* (or types, classes) representing important things of the application domain and *relationships* representing associations among those things.

When building a conceptual model, it is relatively easy to identify adequate entities to capture real-world objets: they directly correspond to the important concepts naturally manipulated by stakeholders in the application domain. The research reviewed in this paper advocates the use of a rich repertoire of relationships for modeling associations between entities. Thus, for the conceptual modeler, the choice of appropriate relationships to associate objects is comparatively more difficult. Various possible choices of relationships correspond to sometimes subtle differences in the shades of real-world semantics captured in a conceptual model. For example, it can be argued that the association between students and employees on the one hand and persons on the other hand is more adequately modeled as a role relationship than as a generalization. As another example, the association between books and their book copies is better modeled as a materialization than as an ad hoc relationship.

The generic relationships reviewed in Section 3 are largely application independent. They are general abstraction patterns for structuring information across application domains. Intermediate between these general patterns and application-specific relationships, generic relationships have also been defined to fit specific application domains. For example, in hypermedia document management [KNS90, WA95], a relationship *BinaryDirectedLink* relates two nodes (or a node and a link) of the hypermedia graph; *copyOf* that relates a blueprint to its original document. Another interesting example deals with the domain of architectural design [PMD95]. There, *hasGeometry* is a relationship that relates a symbolic object and a geometric object; *adjacentTo* relates two objects not related by aggregation and whose distance is less than a specific threshold; *connectedTo* is similar to *adjacentTo*, but the related objects have overlapping volumes.

In practice, when deciding on which relationship best models an association, the modeler has to choose between: (1) a new user-defined ad hoc relationship (like relationship worksFor in Employee–worksFor–Employer), (2) a specific relationship derived from a generic relationship within the repertoire of available generic relationships (such as the relationship Employee→∘Person derived from RoleClass→∘ObjectClass), and (3) a specific relationship derived from a *new* generic relationship identified in the application domain.

The research reviewed in this paper argues that option (2) should be preferred as much as possible. Ad hoc relationships carry little built-in semantics and thus application semantics has to be largely expressed in application programs. New relationships can be identified when the same pattern is repeatedly encountered and it does not fit well the available generic relationships, but the decision to define a new generic relationship should be made with care. For example, some candidate generic relationships can be best described as subcategories of already identified relationships (like the subcategories of aggregation discussed in, e.g., [KP97, Sto93, WCH87]).

4.2 Definition

When a new generic relationship has been tentatively identified, it must be well defined, that is, be intuitively well understood, it should correspond to a significant number of specific instances validated in application domains, its semantics should be formalized, and it should be associated with an appropriate graphical notation.

The *intuitive* semantics of a generic relationship is a broad intent about the duties of the relationship in application domains, in the style of the short descriptions of Section 2.2. The *formal* semantics fits in two categories: the first one positions the relationship along the various characteristics reviewed in Section 2.3 while the second characterizes the inherent semantics of the relationship, in particular in terms of the semantics of creation, update, and deletion of objects involved in the relationship [KR94].

A graphical notation for a generic relationship includes a notation for participating classes and for the relationship itself. Notations of a varying degree of detail can be defined for participation constraints linked with cardinality (see, e.g., [KP97]).

4.3 Close Generic Relationships

The identification and definition of new relationships should carefully explore and characterize their similarities and interactions with existing relationships. This section illustrates the issue with two neighbor relationships, generalization and the comparatively new role relationship.

Similarities and differences, illustrated by the example of Figure 14, include the following:

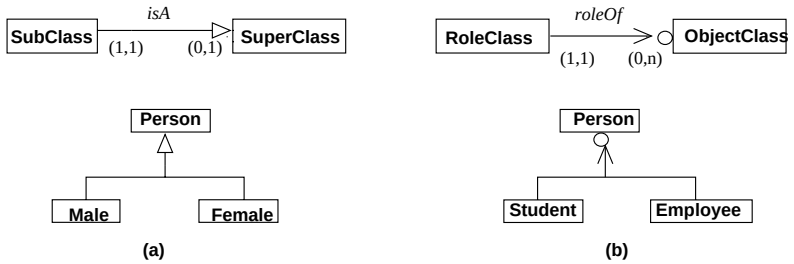

Figure 14: Generalization (a) versus role (b).

- **Cardinalities**. Each instance of a subclass (e.g., Male) is related to exactly one instance of the superclass (e.g., Person) and each instance of the superclass is related to at most one instance of the subclass.
  On the other hand, each instance of the role class (e.g., Student) is related to exactly one instance of the object class (e.g., Person) but, unlike generalization, each instance of the object class can be related to any number of instances of the role class (depending on the maximal cardinality N at the side of the object class).
- **Object identity.** An instance of the subclass has the same object identifier oid as an instance of the superclass. For example, the identity of John as Male is the same as that of John as Person.
  Unlike generalization, an instance of the role class has its own role identifier rid, different from that of all other instances of the role class and from the identifiers of the instances of the object class. For example, the identity of student John is different from that of person John. Furthermore, if John is registered in two universities, there will be one person with a given oid and two students with two different rids.
- **Change of classes.** In most models and systems, an instance of superclass A that is not an instance of subclass B cannot become an instance of B. For example, in the partition of Figure 14(a), an instance of Person that is not a Male cannot become a Male.
  Contrary to generalization, an instance of object class A that is not an instance of role class B can become an instance of B. For example, in the partition of Figure 14(b), an instance of Person that is not a Student can become a Student and an instance of Person that is not an Employee can become an Employee.
- **Change of subclasses.** An instance of a subclass in a partition of the superclass cannot become an instance of another subclass of the partition. For example, in the partition of Figure 14(a), an instance of Male cannot become an instance of Female. Instead, an instance of a given role class in the partition of the object class can become an instance of another role class of the partition. For example, in the partition of Figure 14(b), an instance of Student can become an instance of Employee.

- **Set of instances.** When a subclass changes the set of its instances by creating new objects, then its superclass also changes its instances. For example, the creation of a Male also creates a Person. On the contrary, when a role class creates a new role, then the related object class does not change its instances. For example, the creation of a new role for person John (e.g., John becomes an employee or registers in a university) does not affect the instances of Person.
- **Direct versus indirect instances.** In some generalizations, a superclass is not introduced with the purpose to be directly instantiated, but rather to be specialized by more specific subclasses. For instance, John is not created as a direct instance of Person, but as an instance of Male, that becomes an indirect instance of Person.
  With roles, object classes can be directly instantiated by objects expected to play roles within the specified role classes. For instance, John is created as a direct instance of Person that can play roles of Student and/or Employee.
- **Inheritance.** While subclasses inherit all properties and methods of their superclass, role classes do not inherit properties and methods of their object class. Instead, instances of role classes access properties and methods of their corresponding objects by delegation. Thus, generalization defines the inheritance mechanism at the class level while role defines it at the instance level. We agree with [GSR96] that, when objects change their type dynamically, it is more appropriate to apply specialization and inheritance at the instance level.

5 Implementation of Generic Relationships

In conventional object models, relationships are implemented by class attributes whose domain is a related class. Work on implementing specific relationships (see e.g., [AGO91, Bos96, DP94, DPW93, ER97, HL90, Rum87]) has demonstrated problems and limitations of such pointer or reference technique for representing relationships. Richer implementations essentially suggest, with varying degrees of precision and completeness, to represent generic relationships as independent units that can be reused to declare different user-defined specific relationships.

Only a few implementations of generic relationships have been reported: parameterized class implementations of role [RHS95] and aggregation [KP97], and metaclass implementations of aggregation [HGP98, HGPK94], role [GSR96, KZ98], and materialization [DPZ99]. Materialization has been also implemented in Logtalk [Zim97], and materialization and grouping in Telos [Dah98, MPS95].

This section reviews several mechanisms for implementing generic relationships: pointers or references, layers, parameterized classes (genericity), and, with more detail, metaclasses.

5.1 Pointers

A simple implementation of a generic relationship consists in incorporating the relationship semantics into its participating classes. For instance, to implement the aggregation Car◇—Body, class Body is extended with an attribute referring to Car objects and class Car is extended with an attribute referring to Body objects.

Both classes should also be extended with all necessary methods for supporting the procedural semantics and the integrity constraints of aggregation. Suppose that Car◇—Body is an exclusive and dependent aggregation, expressing that a body always belongs to exactly one car, and that, when deleting a car from the database, the related body must also be deleted. Methods in both classes Car and Body should enforce those constraints; this concerns constructor and destructor methods of both classes, as well as methods updating the pointers implementing the relationship.

Thus, the pointer approach for implementing relationships presents at least two drawbacks: (1) application classes will be more complex, as each class comprises properties for describing the class objects and other properties to account for the semantics of all relationships in which the class participates; (2) the semantics of a generic relationship is repeated for each of its concrete realizations.

5.2 Layers

The layer concept LAYOM [Bos96] is an extended object model that uses a concept of *layer* to deal with relationships between objects. An object can be viewed as consisting of two parts, the kernel

part and the layer part encapsulating the kernel part. Layers can represent several aspects of an object specification. For example, a layer can be used to model constraints on the behavior of the object or to extend the interface of the object without defining a new class for it. A layer has a type and a configuration part, and the behavior of the layer depends on their combination. Each layer encapsulates its object such that each message sent to or by the object has to pass the layer. The layer can handle the message various ways. It can delay, change, redirect or reject the message or it can decide to execute the request by itself and to return a response.

Relationships as layers In LAYOM, generic and specific relationships are implemented by means of layers. A relationship in LAYOM is any connection from one object (or class) O_r to another object (or class) O_s that *influences* the behavior of O_r in some way. Note that this definition explicitly requires the behavior of the object to be influenced by the relation. Relationships that do not influence the behavior of the object are ignored by the layer technique. Also, bidirectional relationships have to be modeled as two unidirectional relationships.

Relationships should be part of the object whose behavior is influenced by the relationship. A relationship is not a separate object, but it should be modeled as an identifiable unit. Therefore, as for the pointer implementation, all relationships of a class are implemented as a part of the class definition.

For example, according to [Bos96], part relationships are directed from the whole to the part, because the behavior of wholes is viewed as extended by its parts and not vice versa. Therefore, the aggregation between ⟨theWholeClass⟩ and ⟨thePartClass⟩ is represented by the layer type "⟨identifier⟩: partOf (⟨thePartClass⟩)" which appears, under the layers clause, as a component of ⟨theWholeClass⟩ definition. The name ⟨identifier⟩ distinguishes a layer among the other layers of the same class. To represent the part relationship Car◇—Body, the layer bPartc: partOf(Body) is associated with Car, meaning that the composite Car has a part relationship with class Body. Upon instantiation of Car, layer bPartc is also instantiated and the layer creates an instance of Body as a corresponding component, thereby reusing the generic semantics. When a message is sent to an instance of class Car, say car#1, it has to pass the layer bPartc. The layer reifies the message and checks whether the selector matches one of the interface elements of Body. If the message matches, it is forwarded to the instance of class Body, say body#1, that is, the part of car#1, otherwise it is passed on to the encapsulated object, i.e., car#1.

Layers concentrate the semantics of a generic relationship in a single structure that is reused in applications by instantiation. However, generic relationships are made available as primitives only by system administrators. Users can only reuse the generic semantics of predefined relationships and define application domain relationships. They cannot define new relationships as primitives. Also relationships are assumed to influence the behavior of at least one class or object involved so that they can be represented by layers as parts of classes influenced by those relationships. This constraint excludes the possibility of dealing with relationships independently of the classes involved.

5.3 Parameterized classes (genericity)

This approach has been investigated in our previous work [KP97] to implement in C++ the semantics of a part relationship model.

For such an implementation, the genericity approach consists in defining one template[4] class PartRel<CmpsClass, CmpnClass, PropClass> abstracting all specific part relationships. The formal generic parameters CmpsClass, CmpnClass and PropClass are types representing, respectively, the composite, the component, and the set of properties characterizing the part relationship between CmpsClass and CmpnClass. These types should be provided for specific realizations of the relationship. For example, the part relationship between Car and its component Body is declared as being the class PartRel<Car,Body,Car_BodyPropClass> where the generic parameters, CmpsClass, CmpnClass, and PropClass, are substituted by Car, Body, and Car_BodyPropClass. Car_BodyPropClass is a set of properties characterizing the part relationship between Car and Body. Similarily, the part relationship Article—◇Journal is represented by the class PartRel<Journal,Article,Journal_ArticlePropClass>.

So, the generic class PartRel<CmpsClass, CmpnClass, PropClass> is, rather than a class, a class pattern covering an infinite set of possible classes; we can obtain any one of these by providing specific parameters corresponding to the formal generic parameters.

[4]Parameterized classes are called *template* classes in C++ and *generic* classes in Eiffel.

As demonstrated in [KP97], genericity enhances reusability since only one specification is used, i.e., PartRel<CmpsClass,CmpnClass,PropClass>, for all realizations of the part relationship model. The genericity approach has, however, at least one limitation. Methods defined in PartRel<CmpsClass, CmpnClass, PropClass> can be applied only to terminal objects at the instance level. Genericity does not provide facilities to manage data or answer questions relevant to the class level such as "what are all parts of class Journal ?", "what are all composites of class Article ?", "what is the cardinality of Journal regarding its componenet Article"., "what are all characteristics of class Article regarding its composite class Journal?". To be able to ensure such requirements, classes must be treated as objects, which is possible only in systems and languages supporting metaclasses.

5.4 Metaclass implementation of relationships

5.4.1 Metaclasses

In object models, classes describe the structure and behavior of their instances with (instance) attributes and methods, respectively. Metaclasses [Coi87, KS95, MPM92] play the same role for classes as classes do for their instances: they describe the structure and behavior of classes with class attributes and methods.

Metaclasses can be used to make a model extensible and flexible, so that it can be tailored for particular uses. In particular, they can be used for defining generic relationships. The semantics of a generic relationship is defined by a structure of metaclasses that provides for defining and querying the relationship, creating and deleting instances of participating classes, and so on. The metaclasses also impose constraints on the transactions and interactions between classes and objects participating in the relationship.

Application classes involved in a generic relationship are then created as instances of the metaclasses representing the relationship. An application designer declares the desired semantics by choosing the appropriate values for the different characteristics of the relationship and for the parameters of the methods supplied by the metaclass. The system then automatically enforces the chosen specification. The designer is thus alleviated of the burden of having to "hand code" the relationship semantics into the methods of the participating classes. Section 5.4.4 illustrates these ideas by surveying a metaclass implementation of materialization.

Metaclass systems do not have a common definition of the metaclass concept. In [DPZ99], we analyzed a set of criteria accounting for the differences.

5.4.2 The data model of VODAK

This section gives an overview of the VODAK [Kla95] data model that will be used in the subsequent sections to illustrate our metaclass implementation ideas.

VODAK Modeling Language (VML) separates the notions of *class* and *object type*. each class in a VML schema is associated with exactly one object type, called its instance type, which defines the structure and behavior of the class's instances.

In VML, classes are themselves objects that are instances of other classes referred to as *metaclasses*. As with an ordinary class, a metaclass has an object type, called *instance type*, that describes its instances, which are classes. Furthermore, a second object type, called *instance-instance type*, is associated with a metaclass to specify the common structure and behavior for the instances of the instances of the metaclass. Therefore, through its two associated object types, a VML metaclass influences both its own instances, which are classes, and the instances of those classes.

Usefulness of VML metaclasses. VML metaclasses are especially useful to integrate generic relationships. In fact, the full semantics of a generic relationship R concern both classes participating in R and instances of those classes. Thus, it is more convenient to represent the semantics of generic relationship R by means of a metaclass to which are attached two types: an instance-type, that provides structure and behavior for the participating classes, and an instance-instance-type, that provides structure and behavior for instances of instances of the metaclass.

Figure 15 shows a VML metaclass PartWholeClass that represents the semantics of a part relationship. The two associated types are: PWInst-Type that describes the structure and behavior for the instances of

the metaclass (i.e., part and whole classes) and PWInstInst-Type that describes the structure and behavior for the instances of instances of the metaclass (i.e., individual part and whole objects).

Figure 15 employes shading to illustrate the effect of PartWholeClass on its instances (e.g., Car and Body) and on instances of its instances (e.g., myFiat and Body#1) are shown in Figure 15. Note that instances of the class Car (resp. Body) are influenced by both the instance type of Car (i.e., CarType) (resp. BodyType) and the instance-instance type of the metaclass PartWholeClass (i.e., PWInst-Type).

This philosophy of modeling assumes that each class A starts with a minimal set of properties and methods representing its structure and behavior as a neutral class. Then, if it is released later on, that A and its instances are also concerned by a concept C whose semantics is captured by a metaclass MC, then at that time A is just declared as instance of MC. As a result, the own properties and methods of A will be augmented by properties and methods of MC's instance type and the structure and behavior of A's instances will be augmented by properties and methods of MC's instance-instance type.

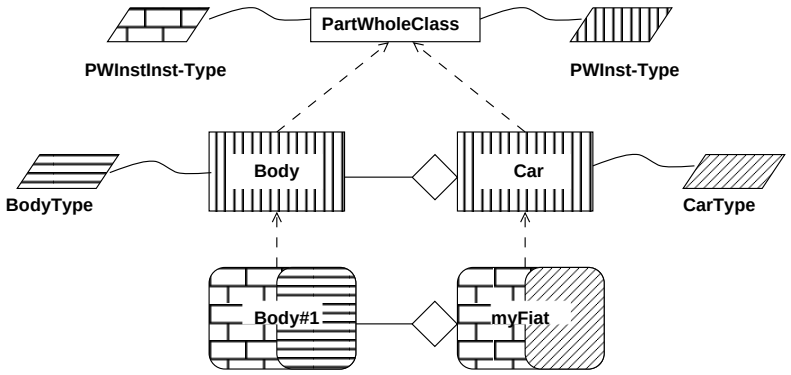

Figure 15: Effects of PartWholeClass on its instances (classes) and on instances of its instances.

5.4.3 Three metaclass approaches

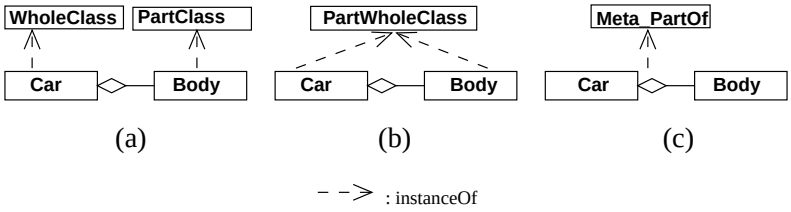

Figure 16: (a) Two metaclasses (b) Single metaclass (c) Relationship metaclass.

Three metaclass approaches for implementing generic relationships are discussed and compared to each other in [DPZ97]. They are illustrated in Figure 16 for the aggregation relationship.

The two-metaclass approach (Figure 16 (a)) consists in defining one metaclass for each role of the generic relationship. Thus, aggregation is represented by two metaclasses WholeClass and PartClass representing the composite and the component roles, respectively. For a specific relationship Car◇—Body, Car is created as instance of WholeClass and Body as instance of PartClass.

Two types are associated to each metaclass accounting, respectively, for the class- and instance-level semantics of the role. Thus, through the instantiation link, Car inherits from its metaclass WholeClass attributes and methods expressing the class- and instance-level semantics of the composite role. Body inherits from PartClass in the same way.

The single-metaclass approach (Figure 16 (b)) consists in defining one metaclass for both roles of the relationship. For example, aggregation is represented by a metaclass PartWholeClass. For a specific

relationship Car◇—Body, both Car and Body are created as instances of PartWholeClass. Two types are associated to the metaclass accounting, respectively, for the class- and instance-level semantics of both roles of the relationship.

This approach has been used to implement aggregation in VODAK [HGPK94], to implement materialization [DPZ99], and it has been motivated in [KS95] to implement all generic relationships.

The relationship-metaclass approach (Figure 16 (c)) consists in defining one metaclass representing the links between objects participating in the relationship. For example, Car and Body are instances of a general system metaclass, while the aggregation Car◇—Body is an instance of metaclass Meta_PartOf, that carries the semantics of aggregation. As in the other approaches, two types are associated with the metaclass to represent the class- and instance-level semantics of the relationship.

5.4.4 Case study: Metaclass implementation of materialization

This section illustrates the single-metaclass approach, for implementing the materialization relationship presented in Section 3.1. A more comprehensive presentation can be found in [DPZ99].

A metaclass AbstractConcreteClass is defined to capture the generic semantics of classes and objects participating in materialization relationships. Two abstract data types are attached to the metaclass, ACClass-InstType and ACClass-InstInstType, for the class- and the instance-level semantics, respectively. The structure of both types is shown, in Figures 17 and 19, respectively.

The AbstractConcreteClass metaclass is made available to applications, as an extension of the data definition mechanisms of the target system. An application can then invoke the generic template, by making application classes A (CarModel) and C (Car), that are to participate in a materialization A—∗C, instances of the metaclass, as shown in Figure 18. Classes like A and C are both referred to as AbstractConcrete (or AC) classes. Upon instantiation of the metaclass, specific information describing the characteristics of materialization A—∗C must also be specified by the schema designer.

```
Define type ACClass-InstType
  Attributes
    theMatRelshps: {matRelationshipType}
    theAbstractClass:OID
  Methods
    defConcreteRelshps(someRelshps: {matRelationshipType});
    defAbstractClass(aClass: OID);

    makeAbstractObject(): OID;
    makeConcreteObject(anObjectFacet: OID): OID;
    destroy(anObject: OID): BOOL;

    getMinCardinality (anAC_Class: OID): integer;
    getMaxCardinality (anAC_Class: OID): integer;
    isAbstractClassOf(anAC_Class: OID): BOOL;
    isConcreteClassOf(anAC_Class: OID): BOOL;
    getConcreteClasses(): {OID};
    getAbstractClass(): OID;
    getInhAttribType1(anAC_Class: OID): {Attribute1-Def};
    getInhAttribType2(anAC_Class: OID): {Attribute2-Def};
    getInhAttribType3(anAC_Class: OID): {Attribute3-Def};
END

DATATYPE matRelationshipType= [
    theConcreteClass: OID,
    cardinality: [min:integer, max:integer],
    inhAttribType1: {Attribute-1Def},
    inhAttribType2: {Attribute-2Def},
    inhAttribType3: {Attribute-3Def}
]
```

Figure 17: Interface of ACClass-InstType.

Class-level semantics. Type ACClass-InstType endows the instances of AbstractConcreteClass, i.e., application classes like CarModel and Car, with structure and behavior consistent with the semantics of materialization.

Figure 17 shows the interface of ACClass-InstType. Note that ACClass-InstType defines methods for both the abstract and the concrete classes. By making an AC class an instance of the metaclass, all methods are made available to the class, be it abstract or concrete. Appropriate error messages are issued when incorrect method invocations are attempted. ACClass-InstType has two attributes: theMatRelshps, meant for the abstract class, and theAbstractClass, for the concrete class. The first attribute is a set of matRelationshipType structures describing characteristics of a specific materialization: the concrete class, the cardinality at the abstract class side, and the propagation types for attributes of the abstract class.

For instance, if class A materializes in two classes B and C (i.e., B∗—A—∗C), then theMatRelshps associated with A will contain two structures describing, respectively, the characteristics of materialization A—∗B and those of A—∗C. The second attribute, theAbstractClass, is the abstract class corresponding to a given concrete class. The generic object type is noted OID: it acts like a formal parameter to be substituted by the name of an application class, when the metaclass is instantiated.

The methods of ACClass-InstType provide the following functions:

- define the specific characteristics of a materialization, and declare the abstract and concrete roles for AC classes with methods defConcreteRelshps and defAbstractClass;
- create abstract objects; when the constructor method makeAbstractObject creates an object, it also creates its associated class facet as a subclass of the concrete class, and it propagates the attributes of the object facet into the class facet (see Section 3.1);
- create concrete objects, with the constructor method makeConcreteObject, and relate them to their associated abstract object;
- delete objects of AC classes, with the destructor method destroy, consistently with the semantics of materialization;
- query AC classes about various aspects of their materialization relationship.

```
CLASS CarModel InstanceOf AbstractConcreteClass
    Attributes
        name: string;
        sticker_price: integer;
        #doors:{integer};
        eng_size:{integer};
        auto_sound: {string};
        special_equip: {string};
    Methods
        setName(string);
        getName(): string
        get#doors():{integer}
        add#doors(integer)
        remove#doors(integer)
        ...
END

CLASS Car InstanceOf AbstractConcreteClass
    Attributes
        manuf_date: date;
        serial# :integer;
        owner: string;
    Methods
        setSerial#(integer)
        setManufDate(date)
        ...
END

CarModel→defConcreteRelshps ({
    [theConcreteClass=Car,
    cardinality= [0,n],
    inhAttribType1= {name, sticker_price},
    inhAttribType2= {[#doors, mono], [eng_size, mono],
        [auto_sound, multi] },
    inhAttribType3= {[special_equip, string]} ]
})
Car→defAbstractClass (CarModel)
```

Figure 18: Materialization CarModel—∗Car.

As an example, Figure 18 shows how the materialization CarModel—∗Car is established. Both CarModel and Car are declared as instances of AbstractConcreteClass. Figure 18 also shows the initializations that give a value to attributes theMatRelshps of CarModel and theAbstractClass of Car. The argument of defConcreteRelshps specifies that: the concrete class related to CarModel is Car; the cardinality for CarModel is $(0, n)$; name and sticker_price propagate with Type1; #doors and eng_size both propagate with Type2 and each produces a monovalued instance attribute, while auto_sound produces, also with Type2, a multivalued instance attribute; special_equip generates, with Type3, new instance attributes of type string.

Querying a materialization hierarchy. Materialization relationships can be queried with the access methods shown in Figure 17.

Methods getMinCardinality and getMaxCardinality give, respectively, the minimal and the maximal cardinalities for the target abstract class with respect to its concrete class passed in the anAC_Class parameter.

Methods isAbstractClassOf and isConcreteClassOf test whether the target class is a *direct* abstract (resp., concrete) class corresponding to the concrete (resp., abstract) class denoted by the anAC_Class parameter. For example, CarModel—▷isAbstractClassOf(Car) returns True. Similarly, getAbstractClass and getConcreteClasses return the set of *direct* abstract (resp., concrete) classes related to the target class.

These methods query only one level of materialization at a time. For example, CarModel→getConcreteClasses() returns the concrete classes of CarModel, i.e., {Car}.

The last three methods (getInhAttribType1/Type2/Type3) are provided to access propagation types of attributes.

Instance-level semantics is provided by type ACClass-InstInstType, whose interface is shown in Figure 19.

```
Define type ACClass-InstInstType
   Attributes
      theConcreteObjects:{OID};
      theAbstractObject:OID;

   Methods
      addConcreteObject (aConcreteObject: OID) : BOOL;
      setAbstractObject (anAbstractObject: OID);
      removeConcreteObject (aConcreteObject: OID) : BOOL;
      removeAbstractObject (anAbstractObject: OID);
      getConcreteObjects () : {OID};
      getAbstractObject() : OID;
END
```

Figure 19: Interface for ACClass-InstInstType.

ACClass-InstInstType has two attributes, theConcreteObjects and theAbstractObject. As a concrete class has cardinality $(1,1)$ with respect to its abstract class, a concrete object is always associated with a single abstract object in theAbstractObject. Attribute theConcreteObjects contains all the concrete objects of an abstract object. They can be of different classes, when the abstract class materializes in more than one concrete class.

The Method part defines a set of methods for creating, suppressing, and querying materialization links. Note that methods addConcreteObject, setAbstractObject, removeConcreteObject, and removeAbstractObject are only invoked in the context of object creation and destruction by methods of ACClass-InstType.

Establishing materialization connections between instances. A connection between an abstract instance a and a concrete instance c is established by inserting c into theConcreteObjects set associated with a (a→addConcreteObject(c)) and assigning a to attribute theAbstractObject of c (c→setAbstractObject(a)). A Boolean value is returned by addConcreteObject to report success or failure. Failure occurs when the classes A of a and C of c are not linked by a materialization A—∗C or if attaching c to a would violate the maximal cardinality at the A side.

Deletion of materialization connections between instances. To break a connection between abstract object a and concrete object c, c is removed from theConcreteObjects set associated to a and attribute theAbstractObject attribute associated to c is set to null.

Like addConcreteObject, removeConcreteObject returns a Boolean value to indicate success or failure. Failure occurs when a and c are not linked by a materialization or if deletion would violate the minimal cardinality at the A side.

Querying materialization connections between instances. Links between abstract and concrete instances can be queried. Method getConcreteObjects returns all the concrete objects of an abstract object a (the contents of theConcreteObjects attribute). Method getAbstractObject returns the value of attribute theAbstractObject.

5.4.5 Interaction between Generic Relationships

Classes commonly participate in several generic relationships. As an example, Figure 20 shows class Car involved in four generic relationships: a Car is a specialization of Vehicle, a composite with a Body component, a materialization of a more abstract class CarModel, and a member of class Fleet describing sets of cars. This section examines how interacting generic relationships are implemented in the three

metaclass approaches. We take the example of class Car involved in an aggregation and a materialization relationships.

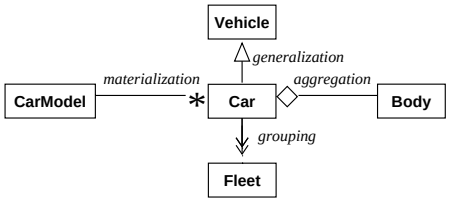

Figure 20: Interacting generic relationships.

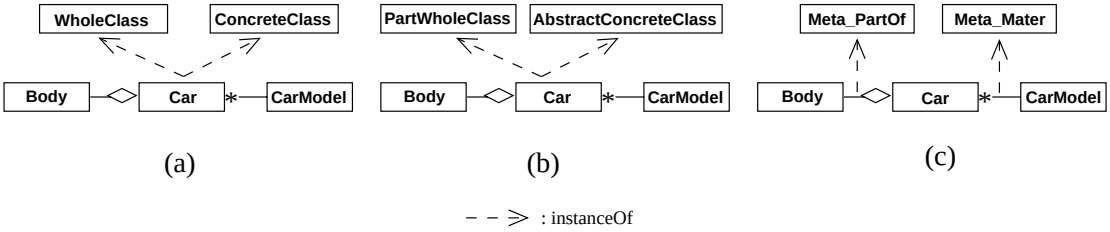

Figure 21: Metaclasses for interacting generic relationships.

The two-metaclass approach. The materialization relationship is represented by metaclasses AbstractClass and ConcreteClass corresponding to the abstract and concrete roles. Similarly, the aggregation relationship is represented by metaclasses PartClass and WholeClass. As a materialization of CarModel and a composite of Body, Car is created as an instance of both WholeClass and ConcreteClass, as in Figure 21 (a). Thus, such a solution is allowed only in systems supporting multiple classification [BG95].

The single-metaclass approach. As shown in Figure 21 (b), materialization is implemented by metaclass AbstractConcreteClass for both the abstract and concrete roles. Similarly, aggregation is represented by metaclass PartWholeClass. As a materialization of Vehicle and a composite of Body, Car is created as an instance of both AbstractConcreteClass and PartWholeClass. Thus, this approach also requires multiple classification for implementing interacting generic relationships.

The relationship-metaclass approach. As shown in Figure 21 (c), materialization and aggregation are represented, respectively, by metaclasses Meta_Mater and Meta_PartOf that abstract relationships instances. Thus, multiple classification is not needed with shared classes like Car, since CarModel—∗Car is created as an instance of Meta_Mater and Car⋄—Body is created, independently, as an instance of Meta_PartOf.

Multiple classification versus multiple inheritance. The usual approach to deal with interacting generic relationships is multiple inheritance [KS95]. As an example, for representing in the single-metaclass approach class Car involved in both a materialization and an aggregation, multiple inheritance leads to defining a new metaclass, say Mater&PartOf, as a subclass of both AbstractConcreteClass and PartWholeClass. Thus, Car is created as instance of Mater&PartOf avoiding multiple classification.

This solution presents, however, some drawbacks: (i) intersection classes like Mater&PartOf are complex, (ii) there is a combinatorial explosion of intersection classes when several generic relationships are subject to interaction, (iii) modifications in the set of relationships in which a class participates require substantial modifications in the structure of associated metaclasses.

6 Conclusion

Generic relationships play a central role for organizing data in the analysis phase of problem resolution. In addition to a few classical generic relationships (generalization, classification, and aggregation), research in conceptual modeling has studied new generic relationships (like materialization, role, generation, and ownership) that naturally model complex situations whose semantics escapes direct representation with classical relationships.

This paper has analyzed the concept of generic relationships and characterized their common semantics along several important dimensions like class and instance semantics, composition, multiplicity, exclusiveness, and sharing. Several generic relationships were reviewed to illustrate how they contribute to enhancing the expressiveness of information models.

The paper has also reviewed mechanisms to implement generic relationships, namely pointers or references, layers, parameterized classes (genericity), and, with more detail, metaclasses. Metaclasses allow to capture structure and behavior associated with a generic relationship independently of specific application classes participating in it. Broadly speaking, the metaclass implementation of generic relationships consists in defining the semantics of a given generic relationship once and for all in a structure of metaclasses, that provide for defining and querying the relationship, creating and deleting instances of participating classes, and so on. Application classes involved in a generic relationship are then created as instances of the metaclasses representing the relationship. Three metaclass approaches for implementing generic relationships have been sketched and one of them has been illustrated by the implementation of the materialization relationship.

References

[AB91] S. Abiteboul and A. Bonner. Objects and views. In J. Clifford and R. King, editors, *Proc. of the ACM SIGMOD Int. Conf. on Management of Data, SIGMOD'91*, pages 238–247, Denver, Colorado, 1991. SIGMOD Record 20(2).

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In R. Agrawal, S. Baker, and D. Bel, editors, *Proc. of the 19th Int. Conf. on Very Large Data Bases, VLDB'93*, pages 39–51, Dublin, Ireland, 1993. Morgan Kaufmann.

[AGO91] A. Albano, G Ghelli, and R. Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In G.M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. of the 17th Int. Conf. on Very Large Data Bases, VLDB'91*, pages 565–575, Barcelona, Catalonia, Spain, 1991. Morgan Kaufmann.

[AHLP95] E. Andonoff, G. Hubert, and A. Le Parc. Modeling inheritance, composition and relationship links between objects, object versions and class versions. In J. Iivari, K. Lyytinen, and M. Rossi, editors, *Proc. of the 7th Int. Conf. on Advanced Information Systems Engineering, CAiSE'95*, LNCS 932, pages 96–111, Jyvaskyla, Finland, 1995. Springer-Verlag.

[AHPZ96] E. Andonoff, G. Hubert, A. Parc, and G. Zurfluh. Integrating versions in the OMT models. In B. Thalheim, editor, *Proc. of the 15th Int. Conf. on Conceptual Modeling, ER'96*, LNCS 1157, pages 472–487, Cottbus, Germany, 1996. Springer-Verlag.

[Ber92] E. Bertino. A view mechanism for object-oriented databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Proc. of the 3rd Int. Conf. on Extending Database Technology, EDBT'92*, LNCS 779, pages 136–151, Vienna, Austria, 1992. Springer-Verlag.

[BG95] E. Bertino and G. Guerrini. Objects with multiple most specific classes. In W.G. Olthoff, editor, *Proc. of the 9th European Conf. on Object-Oriented Programming, ECOOP'95*, LNCS 952, pages 102–126, Aarhus, Denmark, 1995. Springer-Verlag.

[Bos96] J. Bosch. Relations as object model components. *Journal of Programming Languages*, 4:39–61, 1996.

[Bro81] M. Brodie. Association: A database abstraction. In P.P. Chen, editor, *Entity-Relationship Approach to Information Modeling and Analysis*, pages 583–608. North-Holland, 1981.

[CMV96] P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors. *Proc. of the 8th Int. Conf. on Advanced Information Systems Engineering, CAiSE'96*, LNCS 1080, Crete, Greece, 1996. Springer-Verlag.

[Coi87] P. Cointe. Metaclasses are first class: The ObjVlisp model. In Meyrowitz [Mey87], pages 156–167. ACM SIGPLAN Notices 22(12), 1987.

[CZ97] W.W. Chu and G. Zhang. Associations and roles in object-oriented modeling. In D.W. Embley and R.C. Goldstein, editors, *Proc. of the 16th Int. Conf. on Conceptual Modeling, ER'97*, LNCS 1331, pages 257–270, Los Angeles, California, 1997. Springer-Verlag.

[Dah98] M. Dahchour. Formalizing materialization using a metaclass approach. In B. Pernici and C. Thanos, editors, *Proc. of the 10th Int. Conf. on Advanced Information Systems Engineering, CAiSE'98*, LNCS 1413, pages 401–421, Pisa, Italy, June 1998. Springer-Verlag.

[DP94] O. Díaz and N.W. Paton. Extending ODBMSs using metaclasses. *IEEE Software*, pages 40–47, May 1994.

[DPW93] M. Doherty, J. Peckham, and V.F. Wolfe. Implementing relationships and constraints in an object-oriented database using a monitor construct. In N.W. Paton and M.H. Williams, editors, *Proc. of the Int. Workshop on Rules in Database Systems, RIDS'93*, LNCS 1312, pages 347–363, Edinburgh, Scotland, 1993. Springer-Verlag.

[DPZ97] M. Dahchour, A. Pirotte, and E. Zimányi. Metaclass implementation of generic relationships. Technical Report YEROOS TR-97/25, IAG-QANT, Université catholique de Louvain, Belgium, 1997. Submitted for publication.

[DPZ99] M. Dahchour, A. Pirotte, and E. Zimányi. Materialization and its metaclass implementation. Technical Report YEROOS TR-99/01, IAG-QANT, Université catholique de Louvain, Belgium, February 1999. To be published in IEEE Transactions on Knowledge and Data Engineering.

[EPdOVdC+94] N. Edelweiss, J. Palazzo de Oliveira, J. Volkmer de Castilho, E. Peressi, A. Montanari, and B. Pernici. T-ORM: Temporal aspects in objects and roles. In *Proc. of the 1st Int. Conf. on Object Role Modeling, ORM-1*, 1994.

[ER97] B.K. Ehlmann and G.A. Riccardi. An integrated and enhanced methodology for modeling and implementing object relationships. *Journal of Object-Oriented Programming*, 10(2):47–55, May 1997.

[FKN+92] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992.

[GBCGM97] G. Guerrini, E. Bertino, B. Catania, and J. García-Molina. A formal model of views for object-oriented database systems. *Theory and Practice of Object Systems*, 3(3):157–183, 1997.

[GH92] R. Gupta and G. Hall. An abstraction mechanism for modeling generation. In F. Golshani, editor, *Proc. of the 8th Int. Conf. on Data Engineering, ICDE'92*, pages 650–658, Tempe, Arizona, 1992. IEEE Computer Society.

[GS94] R.C. Goldstein and V.C. Storey. Materialization. *IEEE Trans. on Knowledge and Data Engineering*, 6(5):835–842, October 1994.

[GSR96] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM Trans. on Office Information Systems*, 14(3):268–296, 1996.

[HG91] G. Hall and R. Gupta. Modeling transition. In *Proc. of the 7th Int. Conf. on Data Engineering, ICDE'91*, pages 540–549, Kobe, Japan, 1991. IEEE Computer Society.

[HGP98] M. Halper, J. Geller, and Y. Perl. An OODB part-whole model: Semantics, notation, and implementation. *Data & Knowledge Engineering*, 27(1):59–95, May 1998.

[HGPK94] M. Halper, J. Geller, Y. Perl, and W. Klas. Integrating a part relationship into an open OODB system using metaclasses. In N.R. Adam, B.K. Bhargava, and Y. Yesha, editors, *Proc. of the 3rd Int. Conf. on Information and Knowledge Management, CIKM'94*, pages 10–17, Gaithersburg, Maryland, 1994. ACM Press.

[HL90] S. Hwang and S. Lee. An object-oriented approach to modelling relationships and constraints based on abstraction concept. In A.M. Tjoa and R. Wagner, editors, *Proc. of the Int. Conf. on Database and Expert Systems Applications, DEXA'90*, pages 30–34, Vienna, Austria, 1990. Springer-Verlag.

[HPYG95] M. Halper, Y. Perl, O. Yang, and J. Geller. Modeling business applications with the OODB ownership relationship. In R.S. Freedman, editor, *Proc. of the 3rd Int. Conf. on AI Applications on Wall Street*, pages 2–10, New York, June 1995.

[Kat90] R.H. Katz. Towards a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.

[KBG89] W. Kim, E. Bertino, and J. Garza. Composite objects revisited. In J. Clifford, B.G. Lindsay, and D. Maier, editors, *Proc. of the ACM SIGMOD Int. Conf. on Management of Data, SIGMOD'89*, pages 337–347, Portland, Oregon, 1989. SIGMOD Record 18(2).

[Kla95] W. Klas. VODAK V4.0 User Manual. Technical Report TR 910, Arbeitspapiere der GMD, April 1995.

[KNS90] W. Klas, E. Neuhold, and M. Schrefl. Using an object-oriented approach to model multimedia data. *Computer Communications*, 13(4):204–216, 1990.

[KP97] M. Kolp and A. Pirotte. An aggregation model and its C++ implementation. In M.E. Orlowska and R. Zicari, editors, *Proc. of the 4th Int. Conf. on Object-Oriented Information Systems, OOIS'97*, pages 211–224, Brisbane, Australia, November 1997.

[KR94] H. Kilov and J. Ross. *Information Modeling: An Object-Oriented Approach*. Prentice Hall, 1994.

[KS95] W. Klas and M. Schrefl. *Metaclasses and their application*. LNCS 943. Springer-Verlag, 1995.

[KZ98] M. Kolp and E. Zimányi. Prolog-based algorithms for database design. Technical Report YEROOS TR-98/06, IAG-QANT, Université catholique de Louvain, Belgium, 1998. Published as P-98/01.

[LD94] Q. Li and G. Dong. A framework for object migration in object-oriented databases. *Data & Knowledge Engineering*, 13(3):221–242, 1994.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In N.K. Meyrowitz, editor, *Proc. of the Conf. on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'86*, pages 214–223, Portland, Oregon, 1986. ACM SIGPLAN Notices 21(11), 1986.

[LM96] Y. Lahlou and N. Mouaddib. Relaxing the instantiation link: Towards a content-based data model for information retrieval. In Constantopoulos et al. [CMV96], pages 540–561.

[Mat88] N.M. Mattos. Abstraction concepts: The basis for data and knowledge modelling. In Carlo Batini, editor, *Proc. of the 7th Int. Conf. on the Entity-Relationship Approach, ER'88*, pages 473–492, Rome, Italy, 1988.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about informations systems. *ACM Trans. on Office Information Systems*, 8(4):325–362, 1990.

[Mey87] N.K. Meyrowitz, editor. *Proc. of the 2nd Conf. on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'87*, Orlando, Florida, 1987. ACM SIGPLAN Notices 22(12), 1987.

[MMP95] J. Mylopoulos and R. Motschnig-Pitrig. Partitioning information bases with contexts. In *Proc. 3rd Int. Conf. On Cooperative Information Systems*, Vienna, Austria, 1995.

[MP96] R. Motschnig-Pitrik. Analyzing the notions of attribute, aggregate, part and member in data/knowledge modeling. *Journal of Systems Software*, 33:113–122, 1996.

[MPK96] R. Motschnig-Pitrik and J. Kaasboll. Part-whole relationship categories and their application in object-oriented analysis. In *Proc. of the 5th Int. Conf. on Information System Development, ISD'96*, 1996.

[MPM92] R. Motschnig-Pitrik and J. Mylopoulos. Classes and instances. *International Journal of Intelligent and Cooperative Information Systems*, 1(1):61–92, 1992.

[MPM96] R. Motschnig-Pitrik and J. Mylopoulos. Semantics, features, and applications of the viewpoint abstraction. In Constantopoulos et al. [CMV96], pages 514–539.

[MPS95] R. Motschnig-Pitrik and V.C. Storey. Modelling of set membership: The notion and the issues. *Data & Knowledge Engineering*, 16(2):147–185, 1995.

[Myl98] J. Mylopoulos. Information modeling in the time of the revolution. *Information Systems*, 23(3–4):127–155, 1998.

[Odb94] E. Odberg. Category classes: Flexible classification and evolution in object-oriented databases. In G. Wijers, S. Brinkkemper, and T. Wasserman, editors, *Proc. of the 6th Int. Conf. on Advanced Information Systems Engineering, CAiSE'94*, LNCS 811, pages 406–420, Utrecht, The Netherlands, 1994. Springer-Verlag.

[PMD95] J. Peckham, B. MacKellar, and M. Doherty. Data model for extensible support of explicit relationships in design databases. *Very Large Data Bases Journal*, 4:157–191, 1995.

[PT88] W.D. Potter and R.P. Trueblood. Traditional, semantic, and hypersemantic approaches to data modeling. *IEEE Computer*, 21(6):53–62, June 1988.

[PT93] V. Prevelakis and D. Tsichritzis. Perspectives on software development environments. In C. Rolland, F. Bodart, and C. Cauvet, editors, *Proc. of the 5th Int. Conf. on Advanced Information Systems Engineering, CAiSE'93*, LNCS 685, pages 586–600, Paris, France, 1993. Springer-Verlag.

[PZMY94] A. Pirotte, E. Zimányi, D. Massart, and T. Yakusheva. Materialization: a powerful and ubiquitous abstraction pattern. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. of the 20th Int. Conf. on Very Large Data Bases, VLDB'94*, pages 630–641, Santiago, Chile, 1994. Morgan Kaufmann.

[RHS95] D.W. Renouf and B. Henderson-Sellers. Incorporating roles into MOSES. In C. Mingins and B. Meyer, editors, *Proc. of the 15th Conf. on Technology of Object-Oriented Languages and Systems, TOOLS 15*, pages 71–82, 1995.

[Rum87] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In Meyrowitz [Mey87], pages 466–481. ACM SIGPLAN Notices 22(12), 1987.

[Sto93] V.C. Storey. Understanding semantic relationships. *Very Large Data Bases Journal*, 2(4):455–488, 1993.

[WA95] J. Wäsch and K. Aberer. Flexible design and efficient implementation of a hypermedia document database system by tailoring semantic relationships. Technical Report TR 908, Arbeitspapiere der GMD, IPSI, Darmstadt, Germany, April 1995.

[WCH87] M.E. Winston, R. Chaffin, and D. Herrmann. A taxonomy of part-whole relations. *Cognitive Science*, 11:417–444, 1987.

[WCL97] R.K. Wong, H.L. Chau, and F.H. Lochovsky. A data model and semantics of objects with dynamic roles. In A. Gray and P.-A. Larson, editors, *Proc. of the 13th Int. Conf. on Data Engineering, ICDE'97*, pages 402–411, Birmingham, UK, 1997. IEEE Computer Society.

[WDJS95] R.J. Wieringa, W. De Jonge, and P. Spruit. Using dynamic classes and role classes to model object migration. *Theory and Practice of Object Systems*, 1(1):61–83, 1995.

[YHGP94] O. Yang, M. Halper, J. Geller, and Y. Perl. The OODB ownership relationship. In D. Patel, Y. Sun, and S. Patel, editors, *Proc. of the Int. Conf. on Object-Oriented Information Systems, OOIS'94*, pages 278–291, London, UK, 1994. Springer-Verlag.

[YM94] E. Yu and J. Mylopoulos. From E-R to A-R: Modelling strategic actor relationships for business process reengineering. In P. Loucopoulos, editor, *Business Modeling and Re-Engineering, Proc. of the 13th Int. Conf. on the Entity-Relationship Approach, ER'94*, LNCS 881, pages 548–565, Manchester, UK, 1994. Springer-Verlag.

[Zim97] E. Zimányi. Implementing materialization in Logtalk. Technical Report YEROOS TR-97/09, Laboratoire de Bases de Données, Département d'Informatique, Ecole Polytechnique Fédérale de Lausanne, Switzerland, April 1997.