Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Information Systems 29 (2004) 235–270

www.elsevier.com/locate/infosys

A role model and its metaclass implementation ☆

Mohamed Dahchour[a,*], Alain Pirotte[b], Esteban Zimányi[c]

[a] *Institut National des Postes et Télécommunications, Department of Informatics, 2 Av. Allal El Fassi, Madinat Al Irfane, Rabat Instituts, Maroc*

[b] *University of Louvain, IAG School of Management, 1 Place des Doyens, 1348 Louvain-la-Neuve, Belgium*

[c] *Université Libre de Bruxelles, Department of Informatics C.P. 165/15, 50 Av. F. Roosevelt, 1050 Brussels, Belgium*

Received 2 October 2001; received in revised form 21 March 2002; accepted 21 February 2003

Abstract

The *role* generic relationship for conceptual modeling relates a class of objects (e.g., persons) and classes of roles (e.g., students, employees) for those objects. The role relationship is meant to capture dynamic aspects of real-world objects while the usual generalization relationship deals with their more static aspects. Therefore, to take into account both static and dynamic aspects, object languages and systems must somehow support both relationships. This paper presents a generic role model where the semantics of roles is defined at both the class and the instance levels. It discusses the interaction between the role relationship and generalization, and it attempts to clarify some of their similarities and differences. The introduction of roles as an abstraction mechanism in the overall software development lifecycle is reviewed. The paper then proposes a comprehensive implementation for the role relationship, with the help of a metaclass mechanism. Our implementation is illustrated along the lines of the VODAK modeling language. Thus, the semantics of our role model is implemented in a metaclass that is a template to be instantiated in applications. Application classes are then created as instances of the metaclass and they are thereby endowed with structure and behavior consistent with the semantics of roles.

Keywords: Conceptual modeling; Role model; Object models; Metaclass mechanisms

1. Introduction

Object models represent real-world applications as a collection of objects and classes. Objects represent real-world entities while classes represent sets of similar objects. The *classification/instantiation* relationship relates a class to its instances. Classes are organized in *generalization/specialization* hierarchies where subclasses inherit structure and behavior from superclasses.

Most object models assume that an object cannot be a direct instance of more than one class at the same time (in the presence of generalization, the object is also an indirect instance of all the superclasses of its class) and that an object cannot change its class. Those assumptions are not well suited to modeling dynamic situations from the real world. Consider class Person specialized into

☆ Recommended by Prof. P. Loucopoulos.

*Corresponding author.

E-mail addresses: dahchour@inpt.ac.ma (M. Dahchour), pirotte@info.ucl.ac.be (A. Pirotte), ezimanyi@ulb.ac.be (E. Zimányi).

0306-4379/04/$ - see front matter
doi:10.1016/S0306-4379(03)00029-2

classes Student and Employee. If John is created as an instance of Student, it also becomes an instance of Person. But John cannot be created as an instance of Person and later become in addition an instance of Student. Neither can student John change its class to become, say, an employee.

To represent persons that are both students and employees, some models allow *multiple inheritance* with intersection classes. For our example above, an intersection class StudentEmployee would be created as a subclass of both Student and Employee. Multiple inheritance may lead to combinatorial explosion in the number of classes. In fact, in the specialization of Person into Student (S), Alumnus (A), and Employee (E), it would be necessary to create four subclasses (S-A, S-E, A-E, S-A-E) in order to cover all possible cases. Obviously, this can lead to undesirable complexity. In addition, an instance of StudentEmployee cannot be viewed as only a student or only an employee, as all properties of both Student and Employee are inherited in StudentEmployee through generalization. Such a context-dependent access is not supported by generalization [1].

An alternative solution in some models is to allow overlapping generalizations where subclasses Student and Employee may share instances, leading to *multiple classification* [2], the possibility for an object to be a direct instance of more than one class. Although multiple classification avoids the combinatorial explosion caused by multiple inheritance, it does not allow a context-dependent access to objects: an object gathers all the properties of the classes to which it belongs.

Another problem pointed out in, e.g., [1,3–5] is the impossibility of representing the same real-world object as more than one instance of the same class. For example, assume John is a student at two universities, say ULB and UCL, with, in each of them, a student number and a registration in a number of courses. A possible model of such a situation requires three classes: two subclasses ULB_Students and UCL_Students of Student, and an intersection class ULB_UCL_Students as a subclass of both ULB_Students and UCL_Students. Such a solution is heavy and impractical.

The role relationship [1,4–9] was proposed as a way to overcome the above limitations of classical object models. It captures evolutionary aspects of real-world objects that cannot be easily modeled by the generalization relationship nor by time-varying attribute values.

This paper first presents a role model that builds upon existing models and proposes a comprehensive implementation, based on the metaclass mechanism. The metaclass approach has already been used to formalize [10–12] and to implement [1,13–15] other generic relationships such as *aggregation* and *materialization*.

In an object model, classes describe the structure and behavior of their instances with attributes (or instance variables) and methods, respectively. The link between metaclasses and classes is similar to that between classes and their instances: metaclasses describe the structure and behavior of classes with class variables and methods.

Metaclasses allow to capture the structure and behavior associated with a generic relationship R independently of specific application classes participating in the relationship [16]. One possibility is to define the semantics of R once and for all in a metaclass R-Metaclass and, for any specific realization of R between application classes C1 and C2, to define both C1 and C2 as instances of R-Metaclass. Methods inherited from R-Metaclass allow for defining and querying relationship R between C1 and C2, for creating and deleting instances of C1 and C2, and so on. From a modeling point of view, the metaclass mechanism thus extends the object model with a new relationship R, allowing users to better represent application semantics.

The semantics of most relationships R concerns both classes and instances of these classes. Consequently, a single metaclass representing the comprehensive semantics of R must be able to deal with both the *class* and the *instance* levels in a coordinated manner. Thus, it is natural to describe the semantics of R at both levels [14,17]: an *instance type* provides structure and behavior for the instances of the metaclass (which are classes)

and an *instance-instance type* provides structure and behavior for the instances of the instances of the metaclass.

This paper follows such an approach for implementing the role relationship. A metaclass ObjectRoleClass captures the generic semantics of object classes and role classes participating in role relationships. At the class level, the metaclass endows application classes (e.g., Person, Student, and Employee) with the means for defining and querying the role relationships existing among them; it also allows them to create and delete instances according to the semantics of the role relationship. At the instance level, the ObjectRoleClass metaclass provides the instances of its instances with structure and methods for establishing, deleting, and querying role connections.

The objectives of the research reported in this paper can be summarized as follows:

- survey models supporting roles;
- define a more general role model taking advantage of existing models and supporting new features;
- clarify similarities and differences about the semantics of roles and generalizations;
- describe the semantics of roles at both class and instance levels;
- specify legal combinations of roles for an object;
- specify conditions on how objects may or must evolve by gaining and/or losing roles;
- illustrate the relevance of the concepts introduced by a metaclass implementation.

The rest of the paper is organized as follows. Section 2 presents a detailed semantics for our role model. Section 3 presents roles in the context of the software development lifecycle. Section 4 presents the metaclass mechanism and introduces the facilities required of the target system for supporting our implementation of roles. Section 5 presents in detail the metaclass implementation of our role model along the lines of the VODAK modeling language and compares it to other implementation approaches. Section 6 summarizes and concludes the paper.

2. Our role model

This section presents our role model. It is based upon several models proposed in the literature. Section 2.1 presents the general structure of the role relationship. Section 2.2 addresses some issues about object identity and role identity. Section 2.3 explores similarities and differences between the role and generalization relationships. Section 2.4 introduces two concepts for role-control specification: *meaningful combinations* of role classes, used for monitoring membership in various role classes, and *transition predicates*, used to specify when objects may evolve by gaining and/or losing roles. Section 2.5 summarizes the class- and instance-level semantics of our role relationship. Section 2.6 discusses related work and suggests criteria to account for the variety of approaches for role modeling.

2.1. General structure

The role relationship (of pattern Object Class$\circ\leftarrow$RoleClass) relates a class, called *object class*,[1] and another class, called *role class*, that describes dynamic roles for the object class. Fig. 1 shows two role relationships relating the object class Person to role classes Student and Employee. Class Person has three attributes: name, address, and phone. Class Student has four role-specific attributes: univ, stud#, major, and courses, a multivalued attribute. Similarly, class Employee has four attributes: depart, emp#, function, and salary.

As for notations, we draw classes as rectangular boxes and instances as boxes with rounded corners. Classification links appear as dashed arrows and generalization links as solid arrows. Role relationships at the class level are drawn as solid oriented links with a circle on the side of the object class. Role links at the instance level are drawn as dashed oriented links with a circle on the side of the object instance.

We say that instances of the object class gain roles (also that they *play* roles), which are

[1] The *object class* of a role relationship has been called in the literature *root type*, *base class*, *player class*, *base role*, *natural type*, or *role model*.

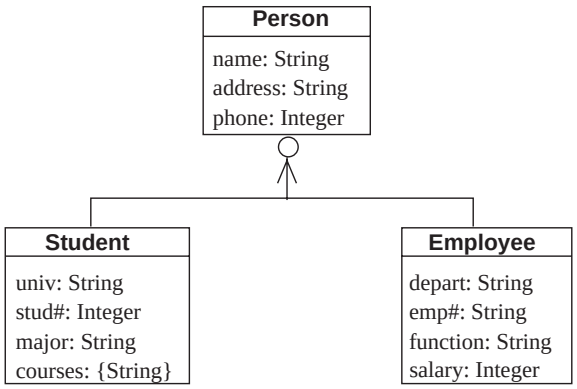


Fig. 1. Examples of role relationship.

instances of the role class. The object class defines permanent properties of objects over their lifetime, while each role class defines some of their transient properties. When the context is clear, instances of object classes will be called *objects* while instances of role classes will be called *roles*.

The role concept addresses three main issues arising when modeling evolving entities with traditional object models:

(1) *Dynamic change of class*, i.e., objects changing their classification. For example, upon graduation, a person ceases to be a student and becomes an alumnus. Two different cases arise depending on whether or not the object undergoing the transition is preserved as an instance of the source class. An *extension* describes the case where the object remains an instance of the source class. Otherwise, the transition is called an *evolution*.
(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 a particular perspective. For example, person John can be viewed separately either as an employee or as a student.

A role can be viewed as a possible state for an object. However, new states of an object that result from changes of its attribute values should not necessarily be modeled as roles. The basic idea is that roles concern new responsibilities, facets, or aspects. For example, a salary raise could be naturally modeled as the change of an attribute value, while an employee becoming member of a golf club could be naturally modeled as a new role for the employee.

Notice that, in the entity-relationship (ER) model, a notion of role is associated with relationships: an entity involved in a relationship is said to play a role. Most often those roles are little more than an alternative name for the relationship, emphasizing the viewpoint of one participating entity. Some differences between the concept of role in the ER model and that of a generic role relationship are discussed in [9]. A role class $\mathcal{R}$ related to an object class $\mathcal{O}$ (i.e., $\mathcal{R} \rightarrow \circ \mathcal{O}$) is expressed in the ER model by making the object class $\mathcal{O}$ participate in a specific relationship representing the role class $\mathcal{R}$. For example, the intuition behind the roles of Fig. 1 is that persons may play roles of students *registered* in some universities and may play roles of employees *working* in some departments. Thus, in the ER model, this can be modeled as classes Student and Employee participating in relationships Student $\overset{registers}{\text{———}}$ University and Employee $\overset{works}{\text{———}}$ Department, respectively. Of course, the semantics underlying the role relationship cannot be captured by the role concept of the ER model.

As already said, object classes are used to represent static properties of objects while role classes are used for modeling their evolutionary aspects. A nonevolving entity is represented as a permanent and exclusive instance of a class (e.g., John as instance of Male if persons are classified according to their sex). An evolving entity, in addition to being a permanent and exclusive instance of an object class, is represented by a set of instances of role classes. When an entity acquires a new role, a new instance of the appropriate role class is created; if it abandons a role, the corresponding role instance is deleted.

Fig. 2 shows some instances of the schema of Fig. 1. It shows how persons may evolve by gaining and/or losing roles. In Fig. 2(a), John_p is created as instance of Person. Fig. 2(b) shows

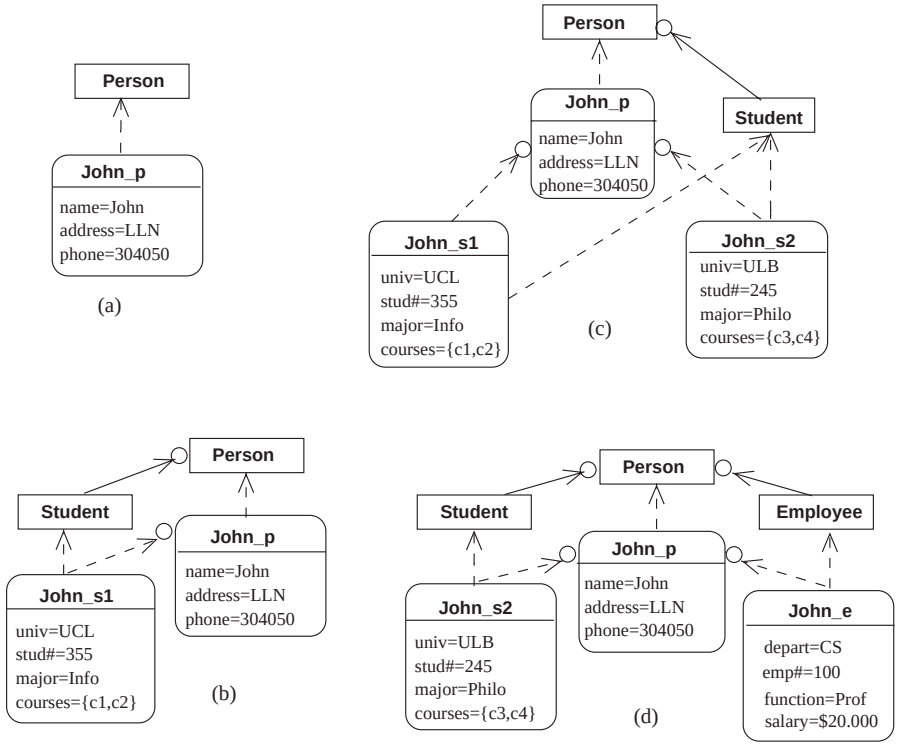


Fig. 2. Various roles of an object.

an instance John_s1 of Student, related to John_p by the role relationship, expressing that John_p has become a student of the UCL university. Both instances John_p and John_s1 coexist with different identifiers (see Section 2.2). If John loses his student role, the instance John_s1 will just be removed. Fig. 2(c) shows another instance of Student, John_s2, reflecting the fact that John is simultaneously registered at another university (ULB). In Fig. 2(d), John has lost his role John_s1 (he left the UCL university) and become an employee (John_e), while still being a student at the ULB university.

Role relationships can be composed in hierarchies, where the role class of one role is also the object class of another role, and so on. In a composition of roles, say $\mathcal{R}_1 \rightarrow \circ\mathcal{R}_2 \rightarrow \circ\mathcal{O}$, an object, instance of $\mathcal{O}$, cannot play a role of $\mathcal{R}_1$ unless it is already playing a role of the intermediate role class $\mathcal{R}_2$.

The example of Fig. 3 deals with a role class Employee having two role classes, ProjectManager and UnitHead. Employee is thus a role class for Person and an object class for ProjectManager and UnitHead. Note that persons can become ProjectManager only if they play the role of Employee. The figure also shows John_e of Fig. 2(d) as an object with a new role John_pm, instance of ProjectManager. If John is promoted as head of the software engineering unit at the CS department, this role is represented as an instance John_uh of UnitHead. John_e is thus a role for John_p and an object for John_pm and John_uh.

2.2. Object identity versus role identity

The concept of object identity has a long history in programming languages [18]. It is more recent in database management [19]. In a model with object identity, an object has an existence that is independent of its attribute values. Hence identity (twice the same object) is not the same thing as equality (objects with the same attribute values) [20].

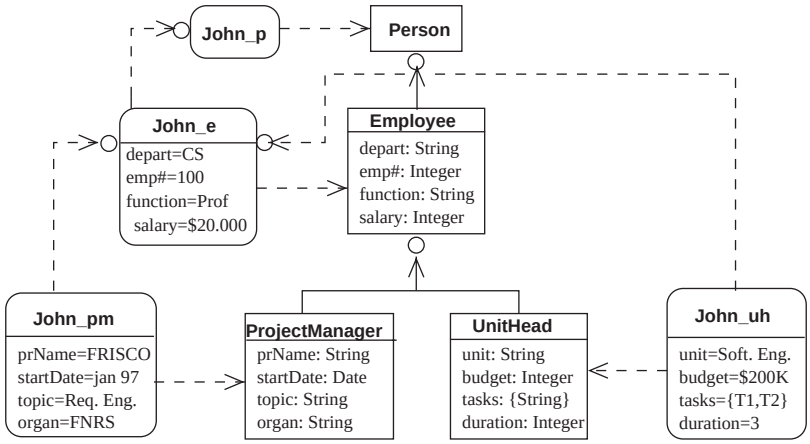


Fig. 3. Composition of roles.

A related concept called *role identity* (rid) was compared to object identity in [21]. The need for both oid and rid was motivated in [4] as follows: assume that Passenger is a subclass of Person (in the sense of generalization) and consider persons who migrate to the Passenger subclass, say, by boarding a bus. A bus may carry 500 passengers during one day, but these passengers may actually be only 100 different persons, if a person may board the bus several times. Hence the necessity of two different ways for counting persons and counting passengers. However, if Passenger is a subclass of Person, counting persons would give the same result as counting passengers.

The migration of an object from a source class to target class(es), with the object losing or retaining its previous membership, introduces the issue of object-identity conservation, i.e., whether the object changes its identifier in the case that it loses its previous membership and whether it retains its identifier in the case that it remains an object of the source class.

Two orthogonal approaches were proposed in the literature: the first one opts for oid conservation (e.g., [22,23]) while the second identifies roles independently of their associated objects, using the notion of role identifiers (e.g., [21,1,7]). For example, consider person John who becomes a student and then an employee while retaining his student status. In the first approach, John is represented by a single oid, while in the second, John is represented by one oid, that of the first state as a Person, and two independent rids corresponding to John as a student and as an employee, respectively.

The first approach presents several disadvantages. First, it fails to capture objects that can have several role objects of the same role class, like John as a student at two different universities. The second approach easily handles that case by creating several instances of the same role class, each with a different rid. Second, changing the oid of John into that of a student object and in turn into that of an employee object raises the issue of dangling references, unlike the second approach, which represents each new role instance with its own rid. Thirdly, the first approach does not permit to represent historical information as the history of oids is lost. For those reasons, our model follows the second approach.

2.3. Roles versus generalization

This section compares the role and the generalization relationships. Section 2.3.1 examines how generalization alone approaches object dynamics. Section 2.3.2 discusses some differences between the relationships, and Section 2.3.3 presents various cases where a class may be involved in both generalization and role relationships.

2.3.1. Dealing with object dynamics using generalization alone

Dealing with object dynamics using generalization is heavy and impractical [24], as illustrated by the following examples, where the role relationships Student→◦Person◦←Employee of Fig. 4(b) are modeled as generalization Student—▷ Person ◁—Employee:

- *How do persons become students?*
  If John is an instance of Person, establishing it as an instance of Student requires the following actions to be performed in order:
  - create an instance, say John_s, of Student;
  - copy values of the instance variables of John into the corresponding instances variables of John_s;
  - reset the references (if any) to John to the new instance John_s;
  - delete instance John.

  The main problem with that solution concerns the identification of all references to the old object John and their update so that they point to the new one John_s. References that are not updated will lead to dangling pointers.
- *How can persons be both students and employees at the same time?*
  A class StudentEmployee, subclass of both Student and Employee, is needed (with multiple inheritance). Anticipating each meaningful combination of classes is impractical.
- *How can persons be employees at more than one department?* To represent John as an employee at two departments, three subclasses of Employee are needed: one for each of the departments and a common intersection class.

To summarize, generalization and role relationships complement each other to deal with the semantics of real-world objects: the role relationship is suited to capture dynamic and evolutionary aspects while generalization best deals with static aspects.

2.3.2. Differences between generalization and the role relationship

As already stated, object languages and systems must somehow support both the generalization and the role relationships in order to capture the static and the dynamic aspects of real-world situations. This section points out some differences between the generalization and the role relationships, illustrated by the example in Fig. 4. Some of those differences are discussed in [4].

- *Cardinalities.* An instance of a subclass (e.g., Male) is related to exactly one instance of its superclass (e.g., Person) and each instance of the superclass is related to at most one instance of its subclasses.
  An instance of the 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.
- *Object identity.* An instance of a subclass has the same object identifier oid as the associated instance of the superclass (it is the same object viewed differently).
  An instance of a 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

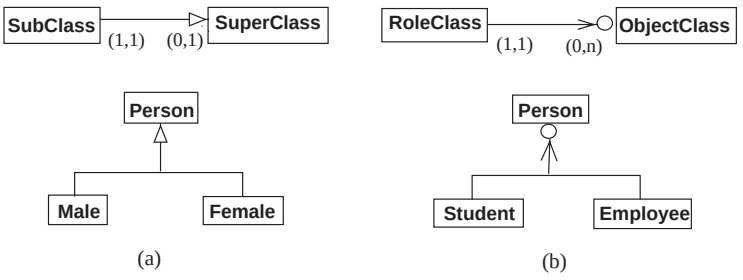


Fig. 4. Generalization (a) versus role (b).

different from that of person John. If John is registered at two universities, there is one person instance with its oid and two students instances each with its rid.

- *Change of classes.* In most object models, an instance of superclass A that is not an instance of subclass B cannot later become an instance of B. For example, in the partition of Fig. 4(a), an instance of Person that is not a Male cannot become a Male.

  An instance of object class A can later become or cease to be an instance of role class B. For example, in the hierarchy of Fig. 4(b), an instance of Person that is not a Student can become a Student.
- *Change of subclasses.* An instance of a subclass in a partition of the superclass cannot later become an instance of another subclass of the same partition. For example, in Fig. 4(a), an instance of Male cannot change to 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 Fig. 4(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.

  Instead, 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 at a university) does not affect the instances of Person.
- *Direct versus indirect instances.* In some generalizations, superclasses may not have direct instances, only their subclasses do. 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.

  There is no analog to those "abstract classes" with the role relationship. Object classes must be instantiated for their objects to be able to play roles of the specified role classes. For instance, John must be created as a direct instance of Person to be able to 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 with a delegation mechanism.

  For example, consider the role relationship Person$\circ\leftarrow$Student and the two instances John_p and John_s1 of Fig. 2(b). Since class Student does not inherit properties of Person, a role instance such as John_s1 accesses properties of John_p by delegation. Thus, if method getPhone() is defined in class Person to read the value of phone attribute, then with usual message handling, the call John_s1$\rightarrow$getPhone() fails since getPhone is not available in class Student. In systems allowing inheritance by delegation, the above message will be delegated to the object John_p (i.e., John_p$\rightarrow$getPhone()) which will successfully answer the request.

2.3.3. Combination of generalization and role relationships

This section summarizes various cases where classes may be involved in both generalization and role hierarchies, and gives some transitive rules linked to that interaction.

A role class $\mathcal{R}_1$ can be subclassed by other role classes $\mathcal{R}_2$ and $\mathcal{R}_3$ (i.e., $\mathcal{R}_2 \text{—}\triangleright \mathcal{R}_1 \triangleleft\text{—} \mathcal{R}_3$), with the usual semantics of generalization. Thus, an instance of $\mathcal{R}_2/\mathcal{R}_3$ is also an instance of $\mathcal{R}_1$, and an instance of $\mathcal{R}_2$ cannot become an instance of $\mathcal{R}_3$. Instead, for a hierarchy of roles $\mathcal{R}_2 \rightarrow \circ \mathcal{R}_1 \circ \leftarrow \mathcal{R}_3$, an instance of $\mathcal{R}_2$ is not an instance of $\mathcal{R}_1$ and an instance of $\mathcal{R}_2$ can become an instance of $\mathcal{R}_3$.

Two derivation rules are associated with the combination of generalization and role relationships:

(1) if $\mathcal{R}_1$ is a role class of $\mathcal{O}$ and $\mathcal{R}_2$ is a role subclass of $\mathcal{R}_1$, then $\mathcal{R}_2$ is also a role class of $\mathcal{O}$ (i.e., $\mathcal{R}_2 \text{—}\triangleright \mathcal{R}_1 \rightarrow \circ \mathcal{O} \Rightarrow \mathcal{R}_2 \rightarrow \circ \mathcal{O}$);

(2) if $\mathcal{R}_1$ is a role class of $\mathcal{O}_1$ and $\mathcal{O}_1$ is a subclass of $\mathcal{O}_2$, then $\mathcal{R}_1$ is a role class of $\mathcal{O}_2$ (i.e., $\mathcal{R}_1 \rightarrow \circ\mathcal{O}_1 \longrightarrow \mathcal{O}_2 \Rightarrow \mathcal{R}_1 \rightarrow \circ\mathcal{O}_2$).

Fig. 5 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 nor vice-versa. Class Male has a role class Draftee accounting for males serving on military duties. According to rule (2) above, Draftee is also a role class of Person. Class Person is refined by two role classes Student and Employee. Student is subclassed by two role classes 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. According to rule (1) above, ForeignStudent and CountryStudent in turn become role classes of class Person. Class Employee has two role classes, Professor and UnitHead. This is an example of compositions of roles.

2.4. Role-control specification

The notions of object class and role class have been used to model situations where objects can play several roles. This section introduces two concepts for role control specification: *meaningful combinations* of role classes, for monitoring membership in various role classes, and *transition predicates*, for controlling how objects may gain and/or lose roles.

2.4.1. Meaningful combinations of roles classes

When an object class $\mathcal{O}$ is related to role classes $\mathcal{R}_1, \ldots, \mathcal{R}_n$, not all combinations of roles are permitted for $\mathcal{O}$ objects. The specification of meaningful combinations of role classes can rule out undesired combinations of roles. For example, if Person has roles Employee and Retired, only employees can acquire the role of retirees, but not the converse.

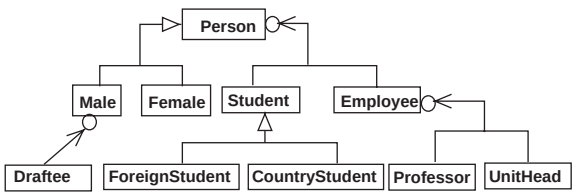


Fig. 5. Generalization and role hierarchies.

We define four types of *role combinations*:

- *Evolution*, noted $\mathcal{R}_1 \overset{ev}{\rightarrow} \mathcal{R}_2$ for role classes $\mathcal{R}_1$ and $\mathcal{R}_2$, with $\mathcal{R}_1 \neq \mathcal{R}_2$, states that an object with a role r_1 of $\mathcal{R}_1$ may lose r_1 and gain a role r_2 of $\mathcal{R}_2$. Further, roles of $\mathcal{R}_2$ cannot be created independently of roles of $\mathcal{R}_1$: an instance of $\mathcal{R}_2$ is necessarily created from an evolving instance of $\mathcal{R}_1$.

  Evolutions may be bidirectional or unidirectional, depending on whether the lost role of $\mathcal{R}_1$ may be recovered later. Examples of bidirectional evolutions are Single $\overset{ev}{\leftrightarrow}$ Married and Employee $\overset{ev}{\leftrightarrow}$ Unemployed. Examples of unidirectional evolutions are Employee $\overset{ev}{\rightarrow}$ Retired and Student $\overset{ev}{\rightarrow}$ Alumnus.
- *Extension*, noted $\mathcal{R}_1 \overset{ext}{\rightarrow} \mathcal{R}_2$, with $\mathcal{R}_1$ not necessarily different from $\mathcal{R}_2$, states that an object playing a role r_1 of $\mathcal{R}_1$ may gain a new role r_2 of $\mathcal{R}_2$, while retaining r_1. Examples include Student $\overset{ext}{\rightarrow}$ Employee and Professor $\overset{ext}{\rightarrow}$ DepartHead.

  Notice that when the source class and the target class of the extension are the same class $\mathcal{R}$, this allows to represent that an object playing a role r_1 of $\mathcal{R}$ may gain new role r_2 of the same role class $\mathcal{R}$, while retaining r_1. Examples include Student $\overset{ext}{\rightarrow}$ Student, when, for example, a person can be simultaneously a student in more than one university, and Employee $\overset{ext}{\rightarrow}$ Employee, when a person can be simultaneously an employee in more than one department.
- *Acquisition*, noted $\rightarrow\mathcal{R}$, states that an object may freely acquire a role of $\mathcal{R}$, if it does not have one. For example, $\rightarrow$Student means that persons can become students. Notice that $\rightarrow$Student could be noted as Person $\overset{ext}{\rightarrow}$ Student, where Person is the object class of role class Student.
- *Loss*, noted $\mathcal{R}\rightarrow$, states that an object may freely lose roles of $\mathcal{R}$. For example, Student$\rightarrow$

means that students may cease to be students. Notice that Student $\rightarrow$ could be noted as Student $\overset{ev}{\rightarrow}$ Person if Person is the object class of role class Student.

2.4.2. Transition predicates

Given a meaningful combination of roles $\mathcal{R}_1 \overset{mode}{\rightarrow} \mathcal{R}_2$, a *transition predicate* is associated with $\mathcal{R}_1$ for describing necessary and/or sufficient conditions on how objects playing roles of $\mathcal{R}_1$ may explicitly or automatically acquire roles within $\mathcal{R}_2$. The source class $\mathcal{R}_1$ may be a role class (in an evolution or an extension) or it may be an object class (in an acquisition).

Transition predicates are specified using a formal declarative language close to the object constraint language (OCL) [25] of the unified modeling language (UML) [26]. Transition predicates are similar to Eiffel's [27] assertions (pre- and post-conditions, invariants). Since a complete description of a language for expressing predicates falls outside the scope of this paper, we just introduce its basic features.

Path expressions with the dot notation are used to denote attributes and methods as well as for traversing role, relationship, and generalization links, and complex attribute structures. Role links allow to pass from an object to a role (and vice versa), and from a role to another role. Relationship links allow to reach a relationship or to pass through a relationship to reach a linked object. Quantifiers ($\exists$ and $\forall$), operators ($\in$, $\subseteq$, etc.), and aggregation functions (count, avg, etc.) are provided for handling multivalued attributes, roles, and relationships. Generalization links allow to change the abstraction level at which an object is seen.

For example, the predicates citizenship = 'Belgium', 'PhD' $\in$ diplomas, address.city = 'Brussels', and age $\geqslant$ 18 associated to the Person class specify conditions on attributes citizenship (monovalued) and diplomas (multivalued), on a component city of the complex attribute address, and on method age, respectively. If Employee is a role, defining a salary attribute and with cardinality (0,1) to Person, then a predicate Person.Employee.salary > 100 concerns the salary of the persons while playing the role of employee (if any). If, on the contrary, the cardinality of the Employee role is (0,n), then the predicates $\exists$e $\in$ Employee, count(e | e $\in$ Employee) < 3, and sum(e.salary | e $\in$ Employee) < 10K defined in Person state conditions about the number of times that a person can play the employee role and the total salary that a person earns from those roles, respectively.

Consider now a relationship WorksIn with attribute fromDate linking the Employee and Department classes. If the cardinality of Employee in the relationship is (1,1), then the predicates WorksIn.fromDate $\geqslant$ 1/1/2002 and WorksIn.Department.name = 'Research' state conditions on the hiring date of the employee and on the name of the department where the employee works. Finally, if Manager is a subclass of Employee, then all attributes and roles defined in Employee are accessible for defining predicates in Manager.

Other examples of transition predicates are as follows:

- Employee $\overset{ev}{\rightarrow}$ Retired. The predicate age $\geqslant$ 55 $\wedge$ workDuration $\geqslant$ 20 associated to Employee could state a (necessary and/or sufficient) condition for employees to retire.
- Employee $\overset{ev}{\leftrightarrow}$ Unemployed. The predicate contractExpirDate $\leqslant$ currentDate associated to Employee specifies that employees whose contract has expired automatically lose their employment and become unemployed. Another predicate associated to Unemployed could state that unemployed persons with a new contract automatically become employees.
- MasterStudent $\overset{ext}{\rightarrow}$ PhDStudent. The predicate grade $\geqslant$ minGrade and funding = OK attached to MasterStudent specifies a condition to be satisfied for master students to be allowed to register as PhD students.
- Student $\overset{ext}{\rightarrow}$ Student. The predicate nbProgs $\leqslant$ max states that the number of registrations for a student is limited by some maximum.
- $\rightarrow$ Employee. The predicate age $\geqslant$ 18 attached to Person specifies that only adults can be employees.

2.5. Summary of role semantics

Like most generic relationships [14,16], the semantics of role relationship concerns both the class level and the instance level. In the literature, these levels have mostly been considered as a whole, which resulted in unwarranted complexity in the semantic definition and implementation of roles. This section summarizes the semantics of the role relationship presented above in terms of class- and instance-level properties.

2.5.1. Class-level semantics

- An object class can have several role classes.
- A role class has exactly one object class.
- An object class has an arbitrary cardinality (c_{min}, c_{max}) regarding each of its role classes.
- Each role class has cardinality (1,1) regarding its object class.
- A role class can have other role classes.
- Meaningful combinations of role classes can be associated with object classes to provide what we have called *role control*.
- A predicate can be associated with object/role classes to describe constraints on how objects may evolve, as well as how objects may or must automatically gain and lose roles.

2.5.2. Instance-level semantics

- A role r is not identical to an object o playing that role, but it is a state in which object o can be.
- A role inherits properties and methods from its associated object by delegation.
- An object can gain several roles of the same role class or of different role classes.
- For each role, there is only one object playing that role.
- Roles can be acquired and lost independently of each other.
- A role has its own identifier that is different from that of the object playing the role.
- Roles evolve independently from each other or under the control of specified transition predicates.
- If an object class $\mathcal{O}$ has two role classes $\mathcal{R}_1$ and $\mathcal{R}_2$ and o is an instance of $\mathcal{O}$ playing role r1 of $\mathcal{R}_1$, then r1 can only evolve as an instance of either $\mathcal{R}_1$ or $\mathcal{R}_2$.
- In a composition of roles, say $\mathcal{R}_1 \rightarrow \circ\mathcal{R}_2 \rightarrow \circ\mathcal{O}$, an object of $\mathcal{O}$ cannot acquire a role of $\mathcal{R}_1$ unless it already holds a role of $\mathcal{R}_2$. For example, in Fig. 3, persons cannot become ProjectManager unless they are already playing the role of Employee.

2.6. Related work

The interest for capturing the roles of an object goes back to the late 1970s. The role model introduced in [28] uses the term *entity* to denote the static part of objects and *role-class* to denote a dynamic type meant for capturing their behavior. Since then, various approaches for role modeling have been proposed. We suggest the following criteria to account for the variety of those approaches.

Generalization only. The specialization/generalization hierarchy is used to model both static and dynamic objects. As the basic semantics of generalization allows modeling static objects only, new concepts have been added to generalization to deal with evolving objects. This approach was adopted, for example, in [23,22,29,30]. In [23], each class of the generalization hierarchy is considered as a particular *perspective* of its instances that are called roles. Thus, each class can be thought of as a role class in our terminology. Classes are endowed with special methods to manage roles: add a new role for an existing object, remove an existing role, etc. In [22], the concepts of *base class* and *residential class* are used to denote, respectively, the class where an object is initially created and the class in which the object currently resides after migration; they correspond, respectively, to the object class and the role class of our model. The role model of [29] deals with evolving objects as follows. Classes are organized along the generalization hierarchy assumed to be static. If, later on, an instance of class C of that hierarchy may evolve to another class D, then D must necessarily be created dynamically as a subclass or superclass of C. D is

referred to as a *role-defining* class. As a result, the application schema will be composed of a static generalization hierarchy and a set of dynamic generalization hierarchies, each one corresponding to a particular application domain. In [30], a class represents a domain abstraction. The "intrinsic" properties of a class are modeled by usual properties while its behavior-oriented properties are modeled by a set of roles types according to which its instances may play roles. A *role type* describes the view an object holds of another object. At any given time, an object may act according to several role types. Thus, the union of all operations defined by the role types constitutes the class interface. Role types are related to each other using association and aggregation, but generalization between role types is not allowed.

Role hierarchy only: In this approach a role hierarchy is built under a root class and an object of this class can play any role in that hierarchy. Thus, all classes are viewed as potentially dynamic, their instances may evolve over time. The classes are organized along the subclassing mechanism. For example, the role model of [6] defines a single hierarchy of *role types* along the subtyping relation, where a role subtype can have several role supertypes. In that model an object is not manipulated directly but only through its roles. The answer of an object to a message depends on the role that receives it: an object does not exist on its own but incarnated in a particular role.

Both generalization and role hierarchies: In other approaches, both hierarchies coexist with orthogonal semantics. The generalization hierarchy is kept for nonevolving objects and the role hierarchy is introduced to model evolving objects. In the role hierarchy, classes are related by a new relationship like our role relationship. This approach was followed, for example, by [4,1,5,31]. In parallel to the generalization hierarchy, [4] introduces a role hierarchy composed of a root class (called object class) and a set of role classes. The relationship between a role class and the object class is called *played-by* (i.e., role in our model). Role classes in turn can be refined in other role classes, and so on. The model of [1] also supports both hierarchies under the names of *class hierarchy* and *role hierarchy*. A role type and its direct ancestor in the role hierarchy are linked by a relationship called *role-of*. In the DOOR model of [5], the role hierarchy is organized along the *player* relationship. In [31] an orthogonal approach is used where (1) a role class is associated to an object class (or to another role class), and (2) both the role and the object class can participate in aggregations, specializations, and associations.

Role identification. The issue here is whether the object that evolves over time by gaining new roles will maintain its identity (i.e., oid) or acquire new oids. As mentioned in Section 2.2, two approaches coexist. The first approach consists in preserving the single object identity regardless of how the objects evolve. This approach is mainly adopted by models that only consider either the *generalization hierarchy* or the *role hierarchy* to model object dynamics such as [6,22,23,29,30]. The second approach argues that the identification principles for "ordinary" objects and for the roles of those objects are different. As seen in Section 2.2, we may count one person where we count five passengers at different times of the day, which corresponds to various states of the same person. Similarly, we may count one person where we count three employees (resp., students) given that a given person may be employee (resp. student) more than once at the same time. This approach is adopted by most models that consider both generalization and role hierarchies such as [7,4,1]. The approach of [5,31] differs from [21,1] in that roles are identified by the names of their role classes as well as their values, instead of using globally unique object/role identifiers. Comparison of both approaches to deal with the identity of evolving objects was discussed in Section 2.2.

Inheritance versus delegation: In addition to the structure and behavior supplied by role classes, roles have to access the properties of their players. Here again, two approaches coexist: inheritance via generalization (class-based inheritance) and inheritance by delegation (instance-based inheritance). In the context of roles, delegation means that roles may delegate messages to their players. Approaches that only use either the *generalization hierarchy* or the *role hierarchy* retain inheritance via generalization (e.g., [4,22,23,29,30]), while those using both generalization and role

hierarchies (e.g., [7,4,1,31]) mainly use the delegation mechanism. Systems that follow a prototype-based paradigm (e.g., [32]) systematically use the delegation mechanism. Further, in [33,34] the concept of *property role* is introduced where a property (e.g., a method) of a role class and an associated property from its object class can be combined. A problem is that the actual combination is programming-language dependent. Instead a conceptual specification of the semantics of property roles should be given, and the implementations should try to map this semantics according to the mechanisms provided by the different programming languages.

Multiple roles within the same class per player: This means the ability for a player to possess more than one role within the same class at the same time. For example, a person may be registered in two different universities as two different students, or may be affiliated with two different departments as two different employees. This issue is easily treated in [4,1] thanks to the coexistence of both identifiers oid and rid. Models that use the unique oid mechanism to identify both objects and their roles (e.g., [6,22,23,29,30]) fail to model this situation. In fact, if John is two employees John_e1 and John_e2 (that are both instances of the Employee class) in two departments, it is clear that both employees must be treated as different although they correspond to the same person. The model of [5] fails to allow multiple instances of a role class per player, whereas models such as [35,3,31] provide this facility, although they do not explicitly use the notion of rid.

Context-dependent access: This refers to the ability of viewing a multi-faceted object in a particular perspective. For example, person John can be viewed as an employee or as a student. Most role models provide this ability. There is however at least one exception. Iris [36], one of the pioneer models for object dynamics, allows an object to belong to several types, and to gain or lose type membership over time, but it misses the possibility to view an entity in a particular aspect, since the structure and methods of all the types that the object belongs to are visible in every context. As a side effect, two roles cannot have different attributes/methods of the same names.

Vertical versus horizontal role possession: With vertical role possession, roles may be gained only along the generalization hierarchy; an instance of a class can play roles only within its direct or indirect subclasses or superclasses. For example, [29] only allows this kind of role possession. With horizontal role possession, objects are allowed to gain roles even within *sibling* (also called *cousin*) classes, between which there is no subclassing relationship. Models that allow horizontal role possession, like [23], also allow vertical role possession, but the converse does not hold. Generally all approaches that use generalization and role hierarchies support both vertical and horizontal role possession.

Composition of roles: All models (e.g., [4,1,5,31]) that use both generalization and role hierarchies allow roles to be played by other roles in the role-playing tree. On the other hand, models that only use either generalization or role hierarchies (e.g., [35,6,22,23,29,30]) do not provide this facility.

Role compatibility: This refers to the ability of explicitly specifying roles that are inherently incompatible and cannot be possessed simultaneously. For example, class Person may have two role classes Secretary and Professor, but persons cannot be secretaries and professors at the same time. Similarly, persons may move from employees to retirees but cannot be employees and retirees at the same time. Few models provide for this facility. The concept of *player qualification* is introduced in [5] as a component of a role class to specify a set of classes whose instances are qualified to play the roles. In [35], the concepts of *independent roles* and *coordinated roles* are used. The coordination between different instances of roles of an object is achieved through rules. In [23], the concept of *disjoint predicate* is introduced to specify that no object may be a member of more than one class in a specified collection of role classes at the same time. For example, the disjoint predicate disjoint(Child, Employee, Retired) can be attached to class Person. In [31] the dynamic organization of roles is represented using a graphical notation, based on an extension of regular expressions. The notation defines legal sequences of role names, including sequencing,

overlapping, iteration, and duplication of roles. For example, such notation allows to associate to a ConferenceAssociate class a graphical regular expression equivalent to (Participant | Author* | Reviewer*)$^+$ indicating that a conference associate may have at most one participant role, any number of author roles, and any number of reviewer roles. In [37] a *role relationship matrix* is used for defining for every pair (A,B) of role types one of the following constraints: (1) *role-dontcare*, stating that there are no constraints on an object playing the roles, (2) *role-implied*, stating that an object playing an A role must always be able to play a B role (but not necessarily the other way), (3) *role-equivalent*, stating that both roles are always available together, and (4) *role-prohibited*, stating that an object playing an A role never plays a B role and vice versa. In [22], migration control is specified by means of the so-called *migration permissions* such as *relocation* permission (e.g., ⟨Single, Married, relocation⟩) and *propagation* permission (e.g., ⟨Professor, Consultant, *propagation*⟩). Relocation and propagation permissions correspond to our two modes of role possession *evolution* and *extension*, respectively. Our role control mechanism was inspired from the migration control of [22] and enriched by the transition predicates (see below), not supported in [22].

Transition predicates: They are attached to classes to specify when objects may explicitly or automatically gain or lose roles within those classes.

In [35], specification of this transition is achieved by means of *rules*. Predicate classes of [38] are populated according to the satisfaction of a given predicate specifying conditions for an object to belong to the class. In [29], there is a possibility to create role classes by means of predicates called *role-generating* conditions, but there is no way to define transition predicates in the sense mentioned above. For example, two predicate-based classes HighlyPaidAcademic and ModeratelyPaidAcademic can be defined as subclasses of Academic class by associating the predicate "Academic.salary >= 100K" to the first class and "100K > Academic.salary > 50K" to the second class. However, there is no way to define a transition predicate $\mathcal{P}$ that would allow persons to migrate from class ModeratelyPaidAcademic to class HighlyPaidAcademic whenever those persons satisfy predicate $\mathcal{P}$.

The concept of *category class* is introduced in [23] to extend the ordinary class by an additional predicate stating the condition required for the *implicit* or *explicit* membership to the class. Our concept of transition predicates was inspired from [23]. However, our transition predicates have different semantics regarding the predicates associated with category classes. Given a meaningful combination of roles $\mathcal{R}_1 \overset{mode}{\rightarrow} \mathcal{R}_2$, a predicate is associated with $\mathcal{R}_1$ to specify *when* an object playing a role of $\mathcal{R}_1$ *may* or *must* automatically play a role of $\mathcal{R}_2$ according to *mode*. Consider the hierarchy of role relationships Teenager$\rightarrow\circ$ Person$\circ\leftarrow$Adult. If a predicate "age$\geqslant$18″ is associated with the role class Adult, Adult becomes a category class in the sense of [23], meaning that the membership in Adult depends on satisfaction of "age$\geqslant$18". Our transition predicate "age$\geqslant$18", is rather associated with the role class Teenager, meaning that when an instance of Teenager satisfies "age$\geqslant$18", it becomes an instance of Adult.

Our model: Our model is similar to the models of [4,1]. Thus, in terms of the criteria above, our model supports: both generalization and role hierarchies, with a clear distinction between them as well as the possible interaction between both hierarchies; object and role identifiers (oid and rid); class-based inheritance (via generalization) and instance-based inheritance (delegation); multiple instances of a role class per player; context-dependent access to multi-faceted objects; vertical and horizontal role possession; composition of roles; role compatibility, by means of the concept of meaningful combinations of roles, and transition predicates. Unlike our model, [4,1] do not support the two last criteria.

Our role control mechanism was inspired from the migration control of [22] and enriched by the transition predicates, not supported in [22]. Our concept of transition predicates was inspired from the predicate associated to the category classes of [23], although, as already stated, our transition predicates have a different semantics.

3. Roles in the software development lifecycle

The role concept was introduced mostly as an abstraction mechanism for representing advanced modeling features that are difficult to deal with in usual object-oriented models, such as multiple classification and dynamic classification. However, the role concept is increasingly used in broader modeling contexts.

Traditionally, object-oriented software development methods have considered the class as the main modeling abstraction. However, some methods such as object-oriented role analysis and modeling (OORAM) [39], Catalysis [40] and TIMe [41] consider the interaction between objects as the essential aspect of object-oriented modeling. Such methods include the concept of *collaboration* as the main abstraction mechanism for the modeler, relegating the class abstraction for the implementation stage. The role concept is considered to adequately represent collaborations between objects, since it allows to focus on the behavior that an object exhibits with respect to the other participating objects, without the need to know the class to which the object belongs.

This section discusses the use of roles along the software development lifecycle, from analysis through design to implementation. The transformation of the role concept along stages of the lifecycle, the relationship between roles and patterns as well as CASE tool support for roles are also addressed.

Roles in object-oriented modeling: When modeling a system, objects are interesting as far as they collaborate with other objects to realize some activities. A collaboration represents an interaction among objects whose objective is the realization of some activity, and where every participating object plays a particular role.

A class describes objects independently of the context in which they exist, without taking into account other objects with which they interact. Thus, an isolated class is not adequate for describing the activities in which its objects participate, since the global description of an activity is distributed among the classes of its participating objects. If an object participates in several activities, then the description of those activities must somehow be part of the class definition.

On the other hand, a role describes structural and behavioral properties of an object in a context. An object plays one or more roles throughout its existence and a particular role may be played by different objects. A role can be seen as a projection (or a partial view) of the class of an object, where the properties of the role are a subset of the properties of the object. A class thus implements one or several roles and a role is implemented by one or several classes.

As already said, several object-oriented development methods have migrated their main focus of interest from classes to collaborations among objects described by roles or types. We next describe OORAM and Catalysis, two methods that offer the concept of collaboration as basic modeling abstraction. We also describe how UML describes collaborations and how it copes with the role concept.

OORAM: OORAM [39] is a software development method that introduces the concept of role modeling as main abstraction mechanism: the activity of design consists in identifying roles that collaborate to achieve a goal. Role modeling was designed for coping with the problem of modeling large systems and their implementation in object-oriented programming languages. Thus, a system is decomposed into a set of subsystems or *interest areas* representing various activities realized by a structure of collaborating objects. Each such structure is described by a role model.

A *role model* describes the objects participating in an activity and their interaction; it contains a set of roles, each role representing all objects occupying the same position in the structure. A role describes the behavior of an object in the context of an activity. A role model is represented by two views. In the *scenario view*, roles are connected by lines if there exists an interaction between them. *Ports* are used to represent that an object playing a role sends *messages* to other objects playing other roles. Sending a message causes the execution of the operation or method of the object receiving the message. The *collaboration view* shows the various messages exchanged by the objects playing the roles ordered along the time line.

OORAM also includes a *state diagram view*, describing the states of roles and the transitions between them, a *process view*, explicitly showing the actions realized by roles and the information flow that they exchange, and a *semantic view*, isomorphic to the collaboration view, but with association relationships instead of ports and interaction paths between roles.

The main properties characterizing roles in OORAM are as follows:

- The role concept unifies the concepts of class and object. Roles have both a static and a dynamic nature, since they allow to describe the properties of the objects that they represent, and they can be used to show how objects collaborate among them.
- Like objects, roles have identity; they can send and receive messages.
- The extension of a role is the set of objects that may play the role. The extension of a role model is a set of object structures obtained by making the objects of the system play their roles; during an instantiation, a role may be played by at most one object.
- A class describes an object independently of the context where that object interacts with other objects; a role describes an object in the context of an activity.
- Roles are independent of classes, they allow to defer the choice of the classes that implement the objects and of the relationships between those classes.
- A role may be implemented by one or several classes and a class may implement one or several roles.

Catalysis: Catalysis [40] is a software development method for open and distributed component systems based on UML. The first activity in Catalysis is to analyze the role of an application in its business environment, defining system specification independently of the implementation details. Then components are specified by their responsibilities and interfaces. Thus, a system is defined by a collaboration of "pluggable parts" such as programming language classes or components. Finally, the internal architecture of components is implemented.

Modeling in Catalysis is based on three basic constructions: type, collaboration, and refining. A *collaboration* defines a set of actions realized by cooperating objects to fulfill some behavior, and each object play some roles (called types) with respect to the other objects of the collaboration. A *type* specifies the visible behavior of an object but not its implementation. A *refining* is the relationship between an abstraction and its realization. For example, a class or a component may be the realization of some interface. Collaborations, types, and refinings are used in the various phases along the development cycle.

Catalysis makes a clear distinction between type and class. A class implements a type; more precisely, one or more classes can implement one or more types. Classes introduce implementation choices, including data members, method bodies, and implementation inheritance. Java also makes such a distinction between type and class, while C++ and Eiffel tend to equate classes and types and rely on programming conventions, such as abstract classes, to distinguish them. This can unnecessarily cause implementation decisions and coupling to make their way into code.

The Catalysis method is based on 3 key principles.

- *Collaborations and mutual models*: Every interaction implies a mutual protocol, which necessarily imposes assumptions and guarantees on all participants. Expressing those conventions with precision requires a shared vocabulary. Catalysis thus defines a mutual model, where responsibilities are clearly specified, although not imposing implementation decisions.
- *Multiple views and view compositions*: Objects participate in many collaborations and hence play multiple roles. Each role can be independently modeled as a separate and simplified view, defining a specific pattern of collaborations with a supporting model. This allows each object to be characterized as more than one type and provides strong support for design patterns in the method itself. Catalysis provides mechanisms for composing views and for

precisely describing essential dependencies between them. This allows to achieve reuse by composing existing collaborations patterns, to divide and conquer complex problems more easily, and to exploit framework-style techniques from models to implementation.

- *Abstraction and refinements*: Interactions can be described at several levels of detail. Abstract descriptions define the interface of components, deferring internal collaborations. Catalysis defines a clear notion of refinement, whereby more detailed descriptions are built in a systematic way from more abstract ones. This decision facilitates the maintenance of traceability from problem domain to code, it enables an incremental approach to model construction and to a development with mixed levels of detail and completeness.

The designers of Catalysis acknowledge the influence of OORAM: "the OORAM approach pioneered role-centric and collaboration-based modeling, and developed ideas of composition and abstraction based on it." Still, there are some differences in how Catalysis and OORAM treat roles. Although the role models of OORAM are similar to the type models of Catalysis, a type has no identity, unlike a role in OORAM. Also a Catalysis type classifies a set of objects, but it provides no implementation.

UML: The UML [26] distinguishes between static and dynamic roles. The former characterize the participation of a class in an association, i.e., they are similar to roles of the ER model. Dynamic roles are those that participate in collaborations.

UML introduces *collaborations* to specify how classifiers (such as classes, interfaces, types, or components and associations between them) collaborate to specify some collective behavior. UML collaborations are used to specify the realization of an operation or a classifier (normally a use case) and to modeling reusable solutions such as design patterns and frameworks. A *classifier* is a mechanism describing structural and behavioral properties. A collaboration defines the context for defining the interactions between instances of classifiers. An *interaction* is a behavioral specification defining the messages that exchange a set of instances for realizing a task, and is always defined in the context of a collaboration.

To describe a collaboration, it is usually not necessary to know all the characteristics of the participating classifiers nor all associations existing among them. Instead of classifiers and associations, a collaboration comprises *classifier roles* and *association roles*, which are projections of classifiers and associations, respectively. For example, for a class, the classifier role represents a projection of a base class and is defined by the characteristics of the class that are necessary for a collaboration. Also, an association role is a projection of a base association and specifies a particular use of it in the collaboration; it is also possible to find associations between classifier roles that have meaning only in some collaboration. A collaboration does not own any of its structural elements, it only references or uses them. Thus, the same element may belong to more than one collaboration.

Although collaborations in UML are appropriate to model interactions between objects, like those of OORAM and Catalysis, it is not clear how to represent the role concept of OORAM with UML collaborations. For this reason, there have recently been proposals for introducing roles in UML (e.g., [42,43]).

Roles in design: The generalization relationship is supported throughout the whole software development lifecycle, from analysis to implementation. Software development methods, like UML, provide generalization as basic abstraction mechanism to be used in analysis and design specifications. Programming languages, like Java and C++, provide specific implementations of generalization, each with its particular semantics.

Ideally, the same situation should hold for the role relationship. In particular, the role concept should be supported by target implementation platforms. There would thus be no gap in progressing from design to implementation. Our implementation of the role relationship in the VODAK Modeling Language presented in Section 5 follows that philosophy.

However, in the same way that object-oriented analysis and design can be effective for a non-object implementation (i.e., using a programming

language that does not support generalization), object-oriented methods could support the role concept directly into the analysis and design phases and could provide different strategies to implement those specifications in a target programming language not supporting the role relationship.

Roles can be implemented into existing object-oriented programming languages using *design patterns* (e.g., [44]). However, as can be expected, such a mechanism does not provide the full expressive power that results from having roles as a first-class construct in target implementation platforms. In particular, roles cannot participate in generalizations, associations, and aggregations in an orthogonal way.

A pattern called *role object* is defined in [45]. It allows to implement roles into systems that do not provide adequate means to dynamically attach new functionality to classes. A role object is an object that represents one specific role of a *core object*. Core objects and role objects correspond, in our terminology, to instances of object and role classes, respectively. The core object allows for the dynamic creation and deletion of roles objects at configuration and runtime. An implementation in C++ is also given.

An implementation of roles using the above pattern has some drawbacks. In particular, roles have been handled through delegation. This either violates the information-hiding principle or results in complicated interfaces and multi-step communication for accessing a role. An alternative approach to role implementation based on aspect-oriented programming (e.g., [46]) has been investigated for example in [47]. An aspect-oriented extension to UML, called Theme/UML, has also been proposed (e.g., [48]).

As noted in [33], the decorator pattern of [44] consists of three classes that to some degree correspond to the role concept: the Component, encapsulating both the object and the role facet, the ConcreteComponent, corresponding to the object, and the Decorator, corresponding to the role. This pattern can be used if the target implementation language does not support roles directly. One drawback is that Decorator classes can be instantiated independently of ConcreteComponent classes, which goes against the semantics of the role relationship. Further, when interpreting decorators as roles, it is in general not possible to reference different but overlapping subsets of decorators.

In the context of framework design, a notion of *composite design pattern* is defined in [37]. That pattern is best described as the composition of other patterns whose integration exhibits a synergy that makes the composition more than just the sum of its parts. The paper presents examples of composite patterns and discusses a role-based analysis and composition technique.

Roles in implementation: When implementing roles with object programming languages, roles are eventually translated into classes, although a class contains the methods and properties of multiple roles. We next discuss how languages like Java and C++ can be used to represent roles.

Java provides native support for interfaces, where interface extension and multiple inheritance of interfaces are supported. Thus, new interfaces can be created by extending existing ones and it is possible to let one class implement more than one interface. This makes Java very suitable for supporting the role relationship since design-level roles can be naturally mapped into implementation-level interfaces.

As noted in [49], the advantage of that formulation of roles is that references to other classes can be typed using the interfaces. Many errors can be prevented through type checking during compilation. With Java, types can also be used at runtime; two different components implementing roles from a particular role model can be plugged together at runtime. The runtime environment will use the type information to allow only legal connections between components.

On the other hand, C++ does not have a language construct for interfaces. The interface of a class is typically defined in a header file containing preprocessor directives. Such files are mixed with the source code at compile time and thus the generalization implied between an interface and its implementation is not supported. Interfaces can be simulated in C++ by using abstract classes containing only virtual methods without implementation. Using multiple inheritance, those

abstract classes can be combined as in Java. However, unlike Java interfaces, the use of virtual methods has a performance impact which may make that implementation roles less attractive in some situations.

However Java interfaces and classes, and C++ virtual classes only allow to implement the instance-level semantics of roles. Their class-level semantics is implemented using metaclasses as illustrated in Section 4. Other implementations of roles are presented in Section 5.4.

CASE tool support for roles: The integration of the role relationship as an abstraction mechanism for software development will obviously benefit from CASE tools that fully support roles throughout the overall software lifecycle.

Current CASE tools support generalization in the various stages of the lifecycle. In the analysis phase, a semantic-rich version of generalization is provided (e.g., allowing multiple inheritance and parallel hierarchies) and semantic checking is realized (e.g., to avoid cyclic generalization hierarchies). In the design phase, additional implementation-oriented generalizations may be added to the analysis models (e.g., to increase code reuse or as a consequence of applying design patterns). Finally, in the implementation phase, the semantics of the generalization relationships of analysis and design models must be captured into the target implementation platform (e.g., using single inheritance for classes and multiple inheritance for interfaces in Java).

In the same way, CASE tools should provide support for the role relationship in analysis, design, and implementation, taking into account the interactions of both the role and the generalization relationships. Of course, this support depends on the particular software development method to be used (e.g., role support will be different in the Catalysis or the OORAM method). For example during analysis, the generalization and role hierarchies must be supported at a conceptual level and semantic checking must be realized among both hierarchies (e.g., to avoid that a class be both a subclass and a role of another class). Also, cross checks must be realized between the static (or class) diagrams and dynamic (or collaboration) diagrams with respect to role relationships. In design, different patterns can be used to translate the semantics of role relationships using the abstractions mechanisms provided by current object-oriented systems. Finally, during implementation, it will be established which roles will be implemented by which classes. Of course, if the target implementation platform supports role as first-class construct, there will be a seamless support of roles throughout the software lifecycle. The implementation of roles given in the next sections goes along that direction.

4. Metaclasses

This section introduces the metaclass concept and shows how it can implement the semantics of our role model.

In an object model, classes describe the structure and behavior of their instances with attributes (or instance variables) and methods, respectively. Metaclasses [14,50–52] play the same role for classes as classes for instances: they describe the structure and behavior of classes with class variables and methods. Treating classes as first-class objects allows to manipulate application classes and their instances in the same way: classes can be retrieved and manipulated as individual objects, and they can be created as instances of metaclasses at run time. For example, a method display can be sent to an individual object. Depending on its concrete implementation, if the method is sent to a instance of the class Person, the property values of the person could be displayed. On the other hand, if the method is sent to the class Person, the characteristic values of the collection of individual persons, like the class definition, the cardinality of the set of instances, or any access statistics could be retrieved.

Metaclasses are an essential mechanism for ensuring the openness of a language or a system. Metaclasses allow to tailor the data model to the specific needs of an application. This can be done by defining specific metaclasses conveying particular requirements of an application and by building up a class system meeting such requirements.

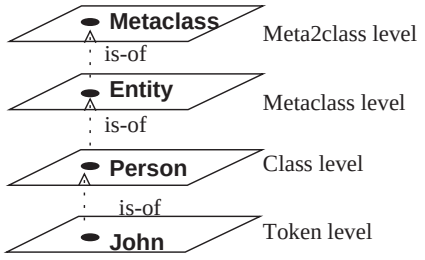


Fig. 6. Levels of systems with a metaclass concept.

Systems with metaclasses comprise at least three levels (Fig. 6): instance, class, and metaclass. In addition, one (or more) predefined metaclass, say Metaclass, can be provided as root describing the common structure and behavior of all metaclasses. The number of levels varies from one system to another. For example, while Smalltalk-80 [53] has a limited depth in its hierarchies of metaclasses, TELOS [54] allows for an arbitrary depth.

There is no common definition of the metaclass concept. A set of criteria accounting for the differences can be found in [52].

4.1. Facilities required of the target object system

Facilities required from the target system with metaclasses for which our implementation of role relationship is formulated are as follows.

- A typical object model with object classes and abstraction mechanisms like classification and generalization.
- The availability of a metaclass mechanism with the following properties:
  - The possibility to explicitly create a metaclass (the semantics of the role relationship will be defined in an ObjectRoleClass metaclass);
  - The ability for several classes to share the same user-defined metaclass (e.g., for a role Employee → ◦Person, both Person and Employee will be instances of ObjectRoleClass);
  - The possibility to define, in the same metaclass, an abstract data type for the class instances of the metaclass and another type for the instances of its instances. For systems that do not allow such mechanism, two data types are needed: a *metaclass* to describe the class-level semantics and a special *class* to represent the instance-level semantics. The coordination between the two data types must be explicitly established.
- The availability of a *generic object type*, as place holder for classes to be instantiated in applications. It acts like a formal parameter in the parameterized metaclass describing the generic semantics, and it is substituted by actual class names (e.g., Person and Employee) when application classes are created as instances of the metaclass.
- The possibility to propagate attribute values by means of a delegation mechanism. This is required so that instances of role classes may access properties and methods of their corresponding objects by delegation.
- The possibility to attach predicates to classes as ordinary properties.

4.2. The VODAK data model

This section gives an overview of the VODAK modeling language (VML) [55] that will be used to illustrate our metaclass implementation. VML is an open object-oriented data model that can be tailored to the needs of specific applications using the concept of metaclasses. VML fills all the features required for the target system described above, except for the support of predicate properties.[2]

VML separates the notions of *class* and *object type*. Each class is associated with exactly one object type, called its *instance type*, which defines the structure and behavior of instances of that class. An instance-type declaration consists of a public and a private part, thereby separating the interface from the implementation. The interface declaration consists of a set of properties and a set of methods. The public methods are the means by which objects communicate with each other. The implementation definition provides code for all

[2] As will be discussed in Section 5.3.5, in VML and in other systems that do not support assertions, such declarative predicates are translated into Boolean methods that are called when objects gain or lose roles.

methods of the interface definition and may introduce additional structural and behavioral definitions. A special primitive data type OID denotes a *generic* object type that can be used to refer an object whose class is not known.

In VML classes are themselves objects that are instances of other classes referred to as *metaclasses*. Like 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.

Fig. 7(a) shows the declaration of metaclass MyMetaclassMC. MyMetaclassMC is itself an instance of another metametaclass specified by the clause METACLASS classIdentifier in the same way as an ordinary class is specified as an instance of a metaclass. The instance type InstTypeMC of MyMetaclassMC is specified by the clause INSTTYPE, while its instance-instance type InstInstTypeMC is specified by the clause INSTINSTTYPE. Fig. 7(b) shows the declaration of MyClassC as an instance of metaclass MyMetaclassMC.

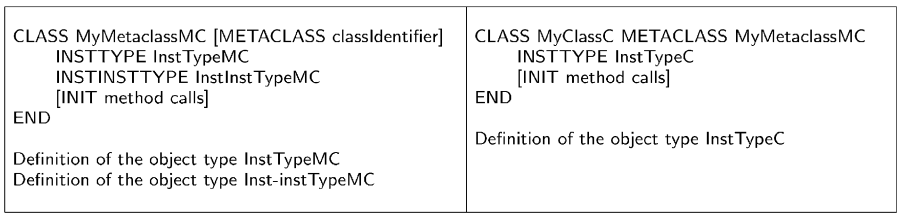
Fig. 8 employs shading patterns to illustrate the effect of the object types defined in Fig. 7 on a class and on the class' instances. Note that instances of class MyClassC are influenced by both the instance type of MyClassC (i.e., InstTypeC) and the instance-instance type of the metaclass MyMetaclassMC (i.e., InstInstTypeMC).

Metaclasses for generic relationships: VML metaclasses are especially useful to represent generic relationships like our role relationship. As seen in Section 2.5, the semantics of roles concerns both the class and the instance level. Thus, it is more convenient to represent the semantics of roles by means of a metaclass to which are attached two types: an instance-type for the class-level semantics and an instance-instance-type for its instance-level semantics.

Inheritance by delegation: VML supports both the usual inheritance by *subtyping* and inheritance by *delegation*. VML deals with inheritance by delegation using the NOMETHOD clause. NOMETHOD is a code segment specified in the instance-instance type of a given metaclass. The content of NOMETHOD is automatically executed by an object o when it receives a message o→m(argList) and is unable to answer it, because method m is not available in o. Two predefined identifiers currentMeth and arguments are available in the body of NOMETHOD. At run time, currentMeth is bound to the method name m and arguments is bound to argList. NOMETHOD uses those two primitives to delegate the failed request to appropriate objects o' that are related to o. This NOMETHOD facility will be used to implement the inheritance by delegation related to roles (see Section 2.3.2).

5. Metaclass implementation of our role model

This section surveys our implementation of the role model presented in Section 2. The implementation is carried out along the lines of VODAK presented in Section 4.2. The section is



(a) (b)

Fig. 7. Declaration of application metaclasses and their instances.

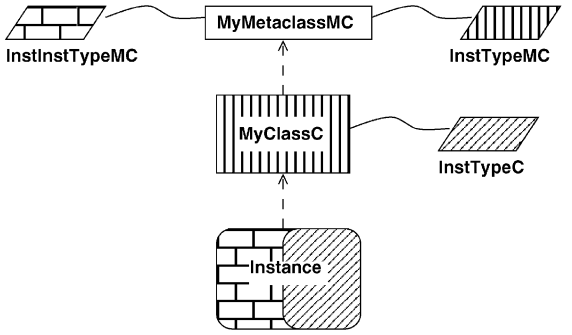


Fig. 8. Effect of the metaclass object types on classes and on individual instances.

organized as follows. Section 5.1 gives an overview of our implementation for roles. Sections 5.2 and 5.3 present the implementation of the class- and instance-level semantics of roles, respectively. Section 5.4 discusses existing work implementing roles and compares them to our approach.

5.1. General structure of our implementation

A metaclass ObjectRoleClass is defined to capture the generic semantics of classes and objects participating in role relationships through the definition of two abstract data types. At the class level, ORClass-InstType endows object classes and role classes (e.g., Person, Student, and Employee) with the means for defining and querying the role relationships existing among them; it also allows them to create and delete instances according to the semantics of roles. At the instance level, ORClass-InstInstType provides the instances of instances of ObjectRoleClass (i.e., objects and roles) with structure and methods for establishing, deleting, and querying role connections. Section 5.2 presents the structure of ORClass-InstType, while Section 5.3 is devoted to that of ORClass-InstInstType.

The effects of ObjectRoleClass on its instances (e.g., Person and Employee) and on instances of its instances (e.g., John_p and John_e) are shown in Fig. 9. Note in particular that instances John_p and John_e are influenced by both the instances types (i.e., PersonType and EmployeeType) of their corresponding classes and by the instance-instance type of ObjectRoleClass (i.e., ORClass-InstInstType).

The definition and use of roles involves the following stages. The first stage is the definition of the metaclass ObjectRoleClass, that embodies the generic semantics of the relationship. An application can then invoke the generic template, by creating application classes, like $\mathcal{O}$ and $\mathcal{R}$, that are to participate in a specific role relationship $\mathcal{R} \rightarrow \circ \mathcal{O}$, as instances of the metaclass (see Fig. 12). Classes like $\mathcal{O}$ and $\mathcal{R}$ are both referred to as $\mathcal{OR}$ classes.

Upon instantiation of the metaclass, specific information describing the characteristics of role relationships must also be specified by the schema designer. For each object class $\mathcal{O}$ related to a set of

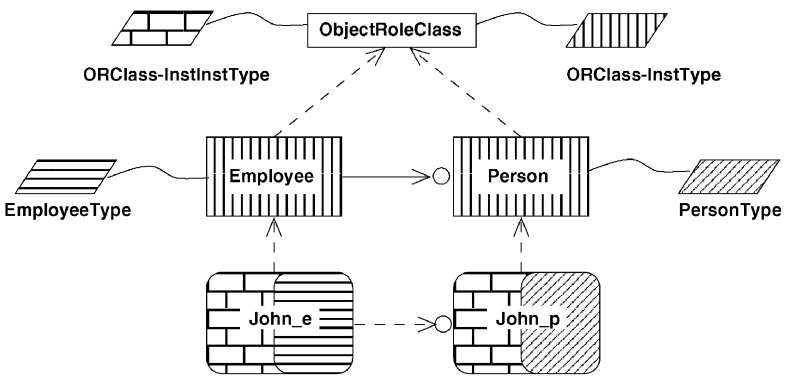


Fig. 9. Effects of ObjectRoleClass on its instances (classes) and on instances of its instances.

role classes $\mathcal{R}_i$, that information includes the cardinality of $\mathcal{O}$ regarding each $\mathcal{R}_i$, information about role control specifying all legal combinations of role classes associated with $\mathcal{O}$, and the transition predicates specifying necessary and/or sufficient conditions for objects to gain and lose roles.

Methods defRoleRelshps and defObjectClass (see Fig. 13) connect $\mathcal{O}$ and $\mathcal{R}$ and initialize the structures needed for instance creation and for querying $\mathcal{OR}$ classes about their role characteristics.

Method addCombination supplies the meaningful combinations of role classes and the corresponding transition predicate. When a combination is no longer valid, it is removed by method delCombination. The transition predicate of a given combination can be revised by method changePredicate.

Methods makeObject and makeRole of ORClass-InstType are available for creating instances of application classes. Method makeObject creates instances of object classes in arbitrary manner, while a role cannot exist without the object that plays it. For this reason, method makeRole, at the same time that it creates a new role, it also creates a role link between the new role and its player.

Method destroy of ORClass-InstType deletes objects and roles, after suppressing the role links in which they participate.

The last nine methods of ORClass-InstType are access methods. They allow to read the cardinality information; test whether a target class is a role/object class of the given parameter class; access all role classes of a given object class; access the meaningful combinations of role classes; get the transition predicate between an object class and a role class or between two role classes; and method canEvolveTo tests whether instances of the target class can evolve to the role class given as parameter.

Finally, methods of ORClass-InstInstType allow to establish, to delete, and to query the role connections between objects and their roles.

Notice that methods addRole, setObject, removeRole, and removeObject of ORClass-InstInstType (see Fig. 17) are never called independently of methods makeObject, makeRole, and destroy of ORClass-InstType. They are private methods hidden for users.

5.2. Class-level semantics

5.2.1. General structure

Type ORClass-InstType endows the instances of ObjectRoleClass (i.e., application classes like Person and Student) with structure and behavior consistent with the semantics of the role relationship. Fig. 10 shows the interface of ORClass-InstType as composed of two parts: Attributes and Methods.

ORClass-InstType comprises three attributes: theRoleRelshps, theObjectClass, and validCombinations. The first attribute is a set of RoleRelType structures describing characteristics of a specific role relationship (the role class, the cardinality at the side of the object class, and the transition predicate that specifies a necessary and/or sufficient condition for an object to play roles of the specified role class). For instance, if the object class $\mathcal{O}$ has two role classes $\mathcal{R}_1$ and $\mathcal{R}_2$ (i.e., $\mathcal{R}_1 \rightarrow \circ\mathcal{O}\circ \leftarrow \mathcal{R}_2$), then theRoleRelshps associated with $\mathcal{O}$ will contain two structures describing, respectively, the characteristics of $\mathcal{R}_1 \rightarrow \circ\mathcal{O}$ and those of $\mathcal{R}_2 \rightarrow \circ\mathcal{O}$. The second attribute, theObjectClass, is the object class related to a given role class. The third attribute, validCombinations, is a set of CombinationType structures describing meaningful combinations of role classes among different role classes specified in the attribute theRoleRelshps via the field theRoleClass of RoleRelType structure. More details about structures RoleRelType and CombinationType are given in Section 5.2.2.

The methods of ORClass-InstType provide the following functionalities:

- define the specific characteristics of the role relationship, and declare the object and the role classes with methods defRoleRelshps and defObjectClass;
- define a set of meaningful combinations of role classes and their associated transition predicate with method addCombinations;
- delete a given combination of role classes or revise the associated transition predicate with

```
Define type ORClass-InstType
  Attributes
    theRoleRelshps: {RoleRelType}
    theObjectClass: OID ;
    validCombinations:{CombinationType}

  Methods

    defRoleRelshps(someRelshps: {RoleRelType});
    defObjectClass(aClass: OID);

    addCombinations(validCombs: {CombinationType}): Boolean;
    delCombination(aValidComb: CombinationType): Boolean;
    changePredicate(aValidComb: CombinationType, aNewPredicate: Assertion): Boolean;

    makeObject(): OID;
    makeRole(anObject: OID): OID ;
    destroy(anObject: OID ): Boolean; % deletes an object or a role %

    getMinCardinality (anOR_Class: OID ): Integer;
    getMaxCardinality (anOR_Class: OID ): Integer;
    isObjectClassOf(anOR_Class: OID ): Boolean;
    isRoleClassOf(anOR_Class: OID ): Boolean;
    getRoleClasses(): {OID};
    getObjectClass(): OID ;
    getCombinations():{CombinationType};
    getPredicate(aR_Class: OID ): Assertion;
    canEvolveTo(aR_Class: OID ): Boolean
End
```

Fig. 10. Interface of ORClass-InstType.

methods delCombination and changePredicate;
- create objects with the constructor method makeObject;
- create roles, with the constructor method makeRole, and relate them to their associated object players;
- delete instances (i.e., objects and roles) of $\mathcal{OR}$ classes, with the destructor method destroy, consistently with the semantics of the role relationship;
- query $\mathcal{OR}$ classes about various aspects of their role relationship.

The structure ORClass-InstType defines methods for both the object classes and the role classes. By making an $\mathcal{OR}$ class an instance of the metaclass, all methods are made available to the class, be it object class or role class. This decision simplified our implementation. Appropriate error messages are issued when incorrect method invocations are attempted. The following sections examine in more detail the methods of ORClass-InstType.

5.2.2. Definition of role characteristics

Methods defRoleRelshps and defObjectClass initialize the structures storing the specific information of role relationships between $\mathcal{OR}$ classes. Method defRoleRelshps is applied to root object classes (i.e., classes without object class). Method defObjectClass is applied to role classes (which can also be object classes in a composition of roles) and specifies, in its aClass parameter, the object class of the target role class.

The parameter someRelshps of defRoleRelshps is a set of RoleRelType structures (see Fig. 11(a)) that specify the characteristics of all role relationships in which the target class participates.

The first component of RoleRelType (theRoleClass) is a place holder for the role class. It acts like a formal parameter to be substituted by the name of an application class, when the metaclass is instantiated (e.g., Student and PhDStudent in Fig. 13). The object class needs not explicitly appear since method defRoleRelshps is necessarily applied to an object class. The second component specifies the cardinality at the object class side. The cardinality at the role class is not declared explicitly, as it is always (1,1).

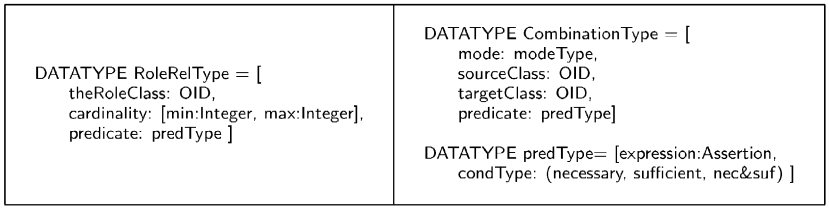


Fig. 11. Definition of the role characteristics.

This means that each instance of a given role class has one and only one associated object that plays it. The third component is a transition predicate that specifies a necessary and/or sufficient condition for an object to play roles of the specified role class.

Methods defRoleRelshps and defObjectClass define only the semantics related to object classes and their role classes. Method addCombinations is introduced to specify the semantics related to role classes between each other. The parameter validCombs of addCombinations is a set of CombinationType structures (see Fig. 11(b)). Each CombinationType structure specifies the mode of role possession (see Section 2.4.1); the source role class; the compatible target role class; and finally the predicate that specifies a necessary and/or sufficient condition for an object, playing a role of the source role class, to possess a role within the target role class.

As an example, consider the two role relationships Student→◦Person◦←PhDStudent and examine how they are established by invoking the generic semantics. Fig. 12 shows that classes Person, Student, and PhDStudent are declared as instances of ObjectRoleClass.

Fig. 13 defines the role relationships between class Person and its roles classes Student and PhDStudent. The argument of defRoleRelshps (see the left-hand side of Fig. 13) specifies the two role relationships Student→◦Person and PhDStudent→◦Person. The first structure of the argument specifies that: the role class related to Person is Student; the cardinality for Person regarding Student is (0,1), meaning that a person can be at most one university student; for a person to become a university student s/he must have obtained the Bacalaureat diploma. Similarly, the second element of the argument specifies that: the role class related to Person is PhDStudent; the cardinality for Person is (0,2), meaning that a person can follow at most two PhD programs in parallel; the associated predicate is false, meaning that a person cannot have its *first* role within PhDStudent. In fact, to be a PhD student, a person must play first the university-student role.

On the right-hand side of Fig. 13, we define one meaningful combination of roles by means of method addCombinations. The argument of addCombinations specifies that students can evolve to PhD students according to the evolution mode; and finally the associated predicate specifies that a student becomes a PhD student if the candidate *necessarily* obtains the Master diploma with grade greater than or equal to 16.

Note that method defObjectClass is not called for a class that is the root of a role hierarchy (i.e., a class without object class), like Person in the example. Similarly, method defRoleRelshps is not called for classes that are leaves of a role hierarchy (i.e., a class without more role classes), like Student and PhDStudent in the example.

Composition of role relationships: To define a composition of roles, say $\mathcal{R}_1 \rightarrow \circ \mathcal{R}_2 \rightarrow \circ \mathcal{O}$, both methods defRoleRelshps and defObjectClass must be invoked on class $\mathcal{R}_2$, as it is involved both as an object class and as a role class. Class $\mathcal{O}$, the root of the hierarchy, only requires defRoleRelshps, while class $\mathcal{R}_1$, the leaf, only requires defObjectClass.

Fig. 14 shows the corresponding method invocations for defining the composition of roles ProjectManager→◦Employee→◦Person of Fig. 3. The figure does not show classes Person, Employee, and ProjectManager as instances of the metaclass ObjectRoleClass.

```
DATATYPE Diploma = [
    type: String,
    major: String,
    grade: INT,
    institution: OID,
    year: DATE]
CLASS Person InstanceOf ObjectRoleClass
    Attributes
        name: String;
        address: String;
        phone: Integer;
        diplomas: {Diploma};
        ...
    Methods
        setName(String);
        getName(): String
        ...
END

CLASS Student InstanceOf ObjectRoleClass
    Attributes
        univ: String;
        stud#: String;
        major: String;
        ...
    Methods
        setUniv(String);
        getUniv(): String
        ...
END

CLASS PhDStudent InstanceOf ObjectRoleClass
    Attributes
        univ: String;
        thesisSubject: String;
        advisor: String;
    Methods
        setMajor(String);
        getMajor(): String
        ...
END
```

Fig. 12. Definition of classes Person, PhDStudent, and Student as instances of ObjectRoleClass.

```
Person→defRoleRelshps ({
    [theRoleClass=Student,
    cardinality= [0,1],
    predicate= [∃ d ∈ diplomas d.type = 'Bac',
        necessary] ],

    [theRoleClass=PhDStudent,
    cardinality= [0,2],
    predicate= [false, necessary] ] })

Student→defObjectClass (Person)
PhDStudent→defObjectClass (Person)

Person→addCombinations ({
    [mode=evolution,
    sourceRole= Student,
    targetRole= PhDStudent,
    predicate= [∃ d ∈ diplomas d.type='Master' ∧
    d.grade≥16, necessary] ] })
```

Fig. 13. Definition of role relationships Student→◦Person◦←PhDStudent.

```
Person→defRoleRelshps ({
    [theRoleClass=Employee,
    cardinality= (0,n),
    predicate= [age ≥ legalAge, necessary] ]
})
Person→addCombinations ({
    [mode=extension,
    sourceRole= Employee,
    targetRole= Employee,
    predicate= [count( e | e ∈ Employee ) ≤ 3 ∧
      sum( e.salary | e ∈ Employee ) ≤ $10.000, necessary] ] })
Employee→defObjectClass (Person)
```

(a)

```
Employee→defRoleRelshps ({
    [theRoleClass=ProjectManager,
    cardinality= (0,n),
    predicate= [∃ d ∈ diplomas d.type='Master' ∧
      d.major='IS', necessary] ]
})
ProjectManager→defObjectClass (Employee)
```

(b)

Fig. 14. Definition of role relationships ProjectManager→◦Employee→◦Person.

Fig. 14(a) shows characteristics of the role Employee→◦Person: Person has only one role class Employee; the cardinality of Person regarding Employee is (0,n), meaning that persons can play several employee roles; to be an employee, a person must necessarily have the legal age. Method addCombinations is also applied to class Person to specify the conditions for a person to simultaneously play several employee roles: the number of positions currently occupied by the employee is less than 3 and the total salary does not exceed $10.000.

Fig. 14(b) defines the characteristics of the role ProjectManager→◦Employee: Employee has only one role class ProjectManager; the cardinality of Employee regarding ProjectManager is (0,n) meaning that an employee can play more than one project manager roles; to be a project manager, an employee must have at least a master's degree in information systems.

5.2.3. Gaining and losing roles

Objects are created, without roles, by method makeObject while roles are created by method makeRole. Whereas objects can be freely created, role creation generally requires some constraints to be satisfied such as: only meaningful combinations of roles are allowed, some roles cannot be created independently, and the transition predicates must be satisfied.

For example, Fig. 15 shows class Person and its two role classes Student and PhDStudent. Person John_p is created as instance of Person (John_p : = Person→makeObject()). Its two roles John_s and John_PhD are created later as instances of Student and PhDStudent, respectively, using method makeRole. The corresponding method invocations are as follows:

- John_s := Student→makeRole(John_p)
  This call is performed when John becomes a student. During this invocation, method makeRole checks whether Student is a role class of John_p's class, i.e., Person, and whether the predicate Person→getPredicate(Student) (i.e., $\exists$ d$\in$diplomas d.type = 'Bac' according to Fig. 13) is satisfied by the object John_p. If all these tests are successful, the role John_s is created as instance of Student.
- John_PhD := PhDStudent→makeRole (John_p)
  This call is carried out when John becomes a PhD student, while losing the previous student role. During this invocation, method makeRole checks whether PhDStudent is a role class of John_p's class and performs Person→ getCombinations() that returns {(evolution,Student, PhDStudent, ($\mathcal{P}$, necessary))} where $\mathcal{P} = \exists$ d$\in$diplomas d.type = 'Master' $\wedge$ d.grade$\geqslant$16, meaning that only students satisfying predicate $\mathcal{P}$ may become PhD students. The method then verifies whether John_p has already a student role that satisfies $\mathcal{P}$. If so, role John_PhD is created as instance of PhDStudent. As the type of the combination above is "evolution", role John_s can be deleted by Student→destroy(John_s).

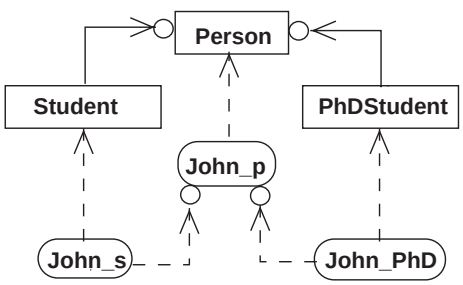


Fig. 15. Object and role creation.

Role connections are established between John_p and its two roles John_s and John_PhD by means of methods addRole and setObject of the instance-level semantics of roles given in Fig. 17.

Fig. 16 shows a general algorithm of method makeRole. For the sake of simplicity, we limit ourselves only to the case of simple role relationships. The algorithm can easily be extended to compositions of roles. Our example above corresponds to the case (1) of that algorithm. The algorithm uses two methods existsAs and viewedAs that are defined in Section 5.3.

```
R1→makeRole(o)
{   O:= o→Class();
    if O→isObjectClassOf(R1) = false then return(null);
    endif;
    combSet:= O→getCombinations();
    if R1 does not appear as a target class in any combination of combSet then
      P:= O→getPredicate(R1);
      if o satisfies P then return (R1→new())
      else return (null);
      endif;
    elsif ∃ R : R→isRoleClassOf(O) and
       case (1): (evolution,R,R1,P) ∈ combSet
         if (o→existsAs(R) = True) and
         the role o→viewedAs(R) satisfies the predicate R→getPredicate(R1) (i.e., P) then
            return (R1→new());
            R→destroy(o→viewedAs(R))
         else return (null);
         endif
       case (2): (extension,R,R1,P) ∈ combSet
         if (o→existsAs(R) = True) and
         the role o→viewedAs(R) satisfies the predicate R→getPredicate(R1) (i.e., P) then
            return (R1→new());
         else return (null);
         endif;
}
```

Fig. 16. Algorithm of method makeRole.

5.2.4. Querying a role hierarchy

Role relationships can be queried with the access methods shown in the lower part of Fig. 10.

Methods getMinCardinality and getMaxCardinality give, respectively, the minimal and the maximal cardinalities in the target class with respect to its role class passed in the anOR_Class parameter.

Methods isObjectClassOf and isRoleClassOf test whether the target class is a *direct* object (resp., role) class corresponding to the role (resp., object) class denoted by the anOR_Class parameter. Similarly, getObjectClass and getRoleClasses return the set of *direct* object (resp., role) classes related to the target class. These methods query only one level of the role relationship at a given time.

Methods getCombinations and getPredicate are provided to access information about meaningful combinations of role classes and the transition predicate between either an object class and a role class or between two role classes of a given meaningful combination of roles.

The last method canEvolveTo tests whether the target class is a role class from which roles can evolve to the role class passed in the aR_Class parameter.

For example, the following queries can be addressed to the roles Student→◦Person◦←PhDStudent:

- Person→getRoleClasses() returns the role classes related to Person (i.e., {Student, PhDStudent});
- Person→getPredicate(Student) returns the transition predicate that must be satisfied by persons to become students (i.e., ∃ d∈diplomas d.type = 'Bac', according to Fig. 13);
- Person→getMinCardinality(Student) returns 0;
- Person→isObjectClassOf(PhDStudent) returns True;
- Student→isRoleClassOf(Person) returns True;
- Person→getCombinations() returns [Student, PhDStudent, [∃ d∈diplomas d.type = 'Master' ∧ d.grade ⩾ 16, necessary]];
- Student→getPredicate(PhDStudent) returns ∃ d∈diplomas d.type = 'Master' ∧ d.grade ⩾ 16;
- PhDStudent→canEvolveTo(Student) returns False;

- Student → canEvolveTo(PhDStudent) returns True.

The following queries can be addressed to the composition ProjectManager → ∘Employee → ∘Person:

- Person → getRoleClasses() returns the direct role class Employee;
- Employee → getMaxCardinality-Employee → getMaxCardinality(ProjectManager)*returns n*;
- Employee → isObjectClassOf (ProjectManager) returns True;
- Person → isObjectClassOf(ProjectManager) returns False, as the method only takes into account the direct object class;
- ProjectManager → isRoleClassOf(Person) returns False, as the method only considers the direct role class.

5.3. Instance-level semantics

This section describes the methods of ORClass-InstInstType, whose interface is shown in Fig. 17.

Notice that methods for creating and suppressing roles links (i.e., addRole, setObject, removeRole, and removeObject) are only invoked in the context of object/role creation and destruction by methods of ORClass-InstType.

```
Define type ORClass-InstInstType
  Attributes
      theRoles:{OID};
      theObject: OID;

  Methods
      addRole(aRole: OID): Boolean;
      setObject(anObject: OID);
      removeRole(aRole: OID): Boolean;
      removeObject(anObject: OID);

      getRoles(): {OID};
      getObject(): OID;
      viewedAs(aRoleClass: OID): {OID};
      existsAs(aRoleClass: OID): Boolean;
      samePlayerAs(aRole: OID): Boolean;
End
```

Fig. 17. Interface for ORClass-InstInstType.

5.3.1. Establishing role connections

For role $\mathcal{R}_1 \rightarrow \circ\mathcal{O}$, a connection between an object o, instance of $\mathcal{O}$, and a role r_1, instance of $\mathcal{R}_1$, is established by inserting r_1 into theRoles set associated with o (i.e., o → addRole(r_1)) and assigning o to attribute theObject of r_1 (i.e., r_1 → setObject(o)). A Boolean value is returned by addRole to report success or failure.

Note that addRole and addObject only add connection information. They are private methods, called by methods makeObject and makeRole of ORClass-InstType, during object/role creation.

5.3.2. Deletion of role connections

To break a role connection between an object o and role r_1, r_1 is removed from theRoles set associated to o (i.e., o → removeRole(r_1)) and attribute theObject attribute associated to r_1 is set to null (i.e., r_1 → removeObject(o)). Like addRole, removeRole returns a Boolean value to indicate success or failure.

Again, these methods only delete the connections between instances. Methods removeObject and removeRole are never invoked independently of method destroy of ORClass-InstType, which deletes objects/roles of application classes, since roles must always be attached to an object.

5.3.3. Querying role connections

Role links between objects and roles can be queried with methods getRoles, getObject, viewedAs, existsAs, and samePlayerAs described next.

Method getRoles returns all the roles played by an object o (the contents of theRoles attribute). If o does not play any role, then the result is an empty set.

Method getObject is applied to a role and returns its associated object (i.e., the value of attribute theObject).

Method viewedAs is applied to an object and returns the role of that object in the context of the role class given in the parameter aRoleClass. It allows to view an object in a particular facet. For example, method invocations John_p → viewedAs(Student) and John_p → viewedAs (PhDStudent) return, respectively, roles John_s

and John_PhD. They allow thus to access the object John_p through two specific facets: John as a student and John as a PhD student.

The functionality of method viewedAs is given by the algorithm of Fig. 18. Note that method viewedAs allows to retrieve the role of an object in a composition of roles. For instance, referring to Fig. 3, both calls John_p→viewedAs(Employee) and John_p→viewedAs(ProjectManager) are allowed and return, respectively, John_e and John_pm. Also, when the object o plays simultaneously several roles of $\mathcal{R}_1$, the method invocation o→viewedAs($\mathcal{R}_1$) will return all roles of $\mathcal{R}_1$ played simultaneously by o. As an example, referring to Fig. 2(c), the call John_p→viewedAs(Student) returns the set {John_s1, John_s2}.

Method existsAs called in the algorithm of Fig. 18 is applied to an object and tests whether that object plays a role within the parameter aRoleClass. For example, method invocations John_p→existsAs(Student) and John_p→existsAs(PhDStudent) return True. A call o→existsAs($\mathcal{R}_1$) also returns True if o plays a role r_1 within $\mathcal{R}_1$ and $\mathcal{R}_1$ is an *indirect* role class of o's class. For instance, referring to Fig. 3, the call John_p→existsAs(ProjectManager) returns True as John_p plays a role John_pm within ProjectManager that is an *indirect* role class of the John_p's class (i.e., Person). The algorithm of method existsAs is very similar to that of method viewedAs given in Fig. 18.

Finally method samePlayerAs tests whether the target role and the role given in the parameter aRole are played by the same player object. For example, all calls John_s1→samePlayerAs(John_s2) (Fig. 2(c)), John_e→samePlayerAs(John_s2) (Fig. 2(d)), and John_e→samePlayerAs(John_pm) (Fig. 3) return True. Method samePlayerAs can be defined by means of method existsAs as shown in Fig. 19.

```
o→viewedAs(R1)
{
 case (1): [o→Class()]→isObjectClassOf(R1) = True
   if ∃r1 ∈ o→getRoles() : r1→Class()=R1 then
     R1Roles := { r | r ∈ o→getRoles() and r→Class()=R1};
     return(R1Roles);
   else return(null);
 case (2): [o→Class()]→isObjectClassOf(R1) = False and
                R1 is an indirect role class of o's class
   if ∃r ∈ o→getRoles() : r→existsAs(R1) = True then
     return(r→viewedAs(R1)) // recursive call
   else return(null)
   endif
}
```

Fig. 18. Algorithm of method viewedAs.

```
r1→samePlayerAs(r2)
{  if ∃ an object o: (o→existsAs(r1→class()) = True) and
   (o→existsAs(r2→class()) = True) then return(True)
   else return(False) }
```

Fig. 19. Algorithm of method samePlayerAs.

5.3.4. Role inheritance

As mentioned in Section 4.2, to deal with inheritance by delegation, VML provides a specific NOMETHOD clause that must be specified in a metaclass' instance-instance type. We use such mechanism for allowing inheritance by delegation between roles and their associated objects.

Thus, the body of the method NOMETHOD is specified in the instance-instance type of the ObjectRoleClass metaclass (i.e., ORClass-InstInstType). It will trap an invocation of a method in a role r (that can also be an object in a composition of roles) that fails and delegates recursively the request to the player of r (i.e., r→getObject()). Fig. 20 shows on its left-hand side the code of method NOMETHOD. The code shows that the message is delegated to the object playing the invoked role denoted by the predefined variable SELF.

On the right-hand side of the figure we illustrate the behavior of NOMETHOD by means of an example referring to Fig. 2(b). In line (1) a new object John_p is created as instance of Person, and in line (2) its phone is set to 304050. Line (3) assigns this phone to the variable phone_p through an invocation of method getPhone defined in class Person. Line (4) creates John_s1 as instance of Student and it is made role of John_p. Finally, in line (5), the variable phone_s1 gets assigned the value 304050, obtained from its object John_p. The initial call fails, since method getPhone is not available for class Student. However, upon this

```
NOMETHOD                                          (1) John_p:=Person→makeObject();
{ // SELF stands fot the actual invoked object    (2) John_p→setPhone(304050);
  [SELF→getObject()]→ currentMeth(arguments);     (3) phone_p:=John_p→getPhone();
}                                                 (4) John_s1:=Student→makeRole(John_p);
                                                  (5) phone_s1:=John_s1→getPhone();
```

Fig. 20. Implementation of role inheritance.

failure, the body of NOMETHOD will be executed in the context of the object John_s1. Thus, the predefined variable SELF will be the object John_s1; currentMeth will be substituted by getPhone and arguments by the empty set (since getPhone has no argument). Finally, as John_s1→getObject() is John_p, the result in line (5) is obtained by calling indirectly John_p→getPhone().

5.3.5. Implementation of predicate properties

We specified in Section 4.1 all the features required from the target metaclass system for our implementation of roles. VODAK provides all these features excepted the support of predicate properties. We discuss next how to cope with the problem of implementing such predicates.

As already stated, transition predicates are specified using a formal declarative language similar to UML's Object Constraint Language. Thus, the problem of supporting transition predicates in our implementation of roles is similar to the problem of supporting integrity constraints in software development methods such as UML. Depending on the target implementation system two different approaches are possible.

Programming languages such as Eiffel [27] or knowledge representation languages like TELOS [54] (supported by the ConceptBase system [56]) provide built-in support for declarative assertions. In this case, a more or less direct implementation of constraints or transition predicates is possible, where the predicates expressed in the conceptual constraint language must be translated into the assertion language provided by the target system. In [12] we used such approach for formalizing the semantics of the materialization relationship in TELOS.

If the target implementation system does not provide support for declarative assertions, then the declarative transition predicates must be translated into operational code by the user, possibly assisted by a CASE tool. Each transition predicate will be implemented by a Boolean method attached to the source class for testing whether or not the predicate is verified. Such Boolean methods are called when an object gains or looses a role.

The formal language used for expressing such predicates was designed for ensuring an easy translation into current object-oriented languages. The path expressions with the dot notation used for accessing monovalued attributes and methods as well as for traversing role, relationship, generalization links, and complex attribute structures are directly supported by current object languages. The quantifiers ($\exists$ and $\forall$), operators ($\in$, $\subseteq$, etc.), and aggregation functions (count, avg, etc.) used for handling multivalued attributes, roles, and relationships need to be translated into loop structures. However, their translation is straightforward.

5.4. Other implementations of roles

Existing work implementing roles can be roughly grouped in two categories. The first one implements roles by extending existing languages and systems using the available basic constructs (e.g., [1,33,8,22,45]). The second category starts from scratch by building suitable languages to support roles. This category includes [36,35,3,7,6,57,5,29]. As our approach follows the first category, we describe next related implementations in this category.

An implementation of roles in Smalltalk using its implicit metaclass concept is reported in [1]. Roles are introduced by adding a few classes without modifying the semantics of Smalltalk. However, because of the implicit status of the Smalltalk metaclasses, this implementation does not clearly distinguish the structure of applications classes involved in role relationships from the structure representing the semantics of roles. Consequently, it is difficult to establish whether

or not this implementation can be reused in other metaclass-based languages and/or systems not based on Smalltalk. Our implementation avoids including the semantics of the role relationship into application classes by defining the semantics of roles in a single metaclass.

The notion of *qualified role* is introduced in [1] to deal with role classes that may have several instances (roles) played simultaneously by an object. Each qualified role has one special attribute, the *qualifier*, according to which roles are created. For instance, according to [1], the role class Student of Fig. 2(c) is a qualified role whose qualifier is university. Roles John_s1 and John_s2 correspond to university = UCL and university = ULB, respectively. Qualified roles allow to distinguish among roles played simultaneously but they require additional methods to manage them. Our approach does not use an *explicit* qualifier as in [1] and thus, 'normal' and qualified role classes are treated uniformly.

In [33] is given an implementation of their role model in Smalltalk and in BETA. The Smalltalk implementation uses reflective constructs and metaclasses while an extension of the BETA compiler was needed since the language does not have reflexive capabilities. However, the BETA implementation has an important drawback from a conceptual perspective: roles may specialize other roles as well as ordinary object classes. In the Smalltalk implementation a Role metaclass is defined ensuring the class-based semantics of roles (e.g., for making accessible to the role instance the properties of the object instance through delegation), while the instance-based semantics is defined modifying the Object metaclass (e.g., for binding role instances to class instances). Our implementation uses the same metaclass for implementing both the class and the instance semantics. The article also discusses implementation issues in both programming languages with respect to referencing roles, binding and unbinding roles, and the dynamic behavior of roles.

An implementation of roles into MOSES with parameterizable classes (i.e., genericity) is reported in [8]. Two classes, Role_Model[R] and Role[M], represent the semantics of roles. Application classes reuse these definitions by inheritance, while specifying generic parameters. For example, to implement Student → ◦Person, Person is declared as a subclass of Role_Model[Student] and Student as a subclass of Role[Person].

Genericity allows a high degree of reusability since only two specifications, i.e., Role_ Model[R] and Role[M], are used for all realizations of the role relationship. The genericity approach has, however, at least one limitation. Methods defined in Role_Model[R] and Role[M] can be only applied to objects and roles 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 role classes of class Person ?", "what is the object class of role class Student ?", "what is the cardinality of Person in its link to its role class Employee?", and so on. To address those requirements, classes must be treated as objects, which is possible only in systems and languages supporting metaclasses.

A prototype based on C++ implementing a framework for object migration is described in [22]. The semantics of the object migration model is concentrated in a hierarchy of *abstract* classes connected by generalization links. Those abstract classes define a set of abstract (virtual) methods, modeling a common behavior for application classes involved in role hierarchies. Virtual methods are meant to be specified later by application classes that reuse abstract classes by subclassing. Again, the problem with this approach is that specializations of virtual methods can be only applied to objects, not to classes. A provision for migration control specification is introduced in [22], but without addressing its implementation.

6. Conclusion

This paper has first presented a generic model for roles that takes advantages from existing models and adds new features. The model enjoys the following characteristics:

- the ability for an object to change its classification, while remaining or ceasing to be an instance of its original class;

- class-based inheritance (via generalization) and instance-based inheritance (via delegation);
- the ability for an object to play more than one role within the same class;
- the ability for an object to play a role within any role class, in so far as it is declared a priori as a valid destination of the evolving object;
- the coexistence of two identifiers, oid for players and rid for their roles;
- the ability to view a multi-faceted object in a particular perspective;
- the possibility for roles themselves to play other roles.

In addition, two concepts were introduced: *meaningful combinations of role classes* to model the legal combinations of roles for an object and *transition predicates* to specify when objects may or must evolve by gaining and/or losing roles. We have also studied the interaction between the role relationship and specialization/generalization and clarified their similarities and differences, often confused.

The paper proceeds by analyzing the introduction of the role relationship in the overall software development lifecycle, from analysis through design to implementation. Software development methods such as OORAM and Catalysis incorporates the role concept as an important abstraction mechanism. The implementation of the role relationship in classical programming languages such as Java and C++ using patterns is also surveyed. However such implementations cannot capture all the semantics aspects of the role relationship.

The paper then proposed a metaclass implementation for roles in order to make them available in object languages and systems for designing applications involving evolving objects.

Our implementation relies upon a target system supposed to provide the following facilities: an object model with objects and classes, classification and generalization, a generic-object type as place holder for classes to be instantiated in applications, the ability to attach predicates as ordinary properties of classes, and a metaclass concept, which plays a key role in the implementation. To implement the role relationship with a metaclass, we also assume the possibility of defining two abstract data types in the metaclass: an *instance type*, for the structure and behavior of application classes involved in the relationship, and an *instance-instance type*, for the instances of these classes.

Our implementation is illustrated along the lines of the VODAK modeling language. VODAK is the closest to the ideal target system above, although it does not support predicate properties. A metaclass ObjectRoleClass was built as a template to capture the semantics of roles at both the class level and the instance level. The implementation is made simpler and more general, concentrating on essential mechanisms linked to a generic version of roles.

At the class level, the metaclass provides classes with the means for defining and querying the role links between them; it also allows them to create and delete their instances according to the semantics of roles. At the instance level, ObjectRoleClass provides the instances of its instances with structure and methods for establishing, deleting, and querying the role links between them. Furthermore, the metaclass provides for implementing value propagation between objects and their roles.

The main advantage of our implementation is that the metaclass approach avoids including the semantics of the relationship into the code of application classes, which would alter their original purpose. Also, code is not replicated every time a role is defined between application classes; instead the code is written once and for all, and reused through instantiations of the metaclass.

References

[1] G. Gottlob, M. Schrefl, B. Röck, Extending object-oriented systems with roles, ACM Trans. Office Inform. Systems 14 (3) (1996) 268–296.

[2] E. Bertino, G. Guerrini, Objects with multiple most specific classes, in: W.G. Olthoff (Ed.), Proceedings of the Ninth European Conference on Object-Oriented Programming, ECOOP'95, Aarhus, Denmark, Lecture Notes in Computer Science, Vol. 952, Springer, Berlin, 1995, pp. 102–126.

[3] J. Richardson, P. Schwarz, Aspects: extending objects to support multiple, independent roles, in: J. Clifford,

R. King (Eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD'91, Denver, Colorado, 1991, pp. 298–307 SIGMOD Record 20(2).

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

[5] R.K. Wong, H.L. Chau, F.H. Lochovsky, A data model and semantics of objects with dynamic roles, in: A. Gray, P.-A. Larson (Eds.), Proceedings of the 13th International Conference on Data Engineering, ICDE'97, Birmingham, UK, IEEE Computer Society, Silver Spring, MD, 1997, pp. 402–411.

[6] A. Albano, R. Bergamini, G. Ghelli, R. Orsini, An object data model with roles, in: R. Agrawal, S. Baker, D. Bel (Eds.), Proceedings of the 19th International Conference on Very Large Data Bases, VLDB'93, Dublin, Ireland, Morgan Kaufmann, Los Altos, CA, 1993, pp. 39–51.

[7] N. Edelweiss, J. Palazzo de Oliveira, J. Volkmer de Castilho, E. Peressi, A. Montanari, B. Pernici, T-ORM: temporal aspects in objects and roles, in: Proceedings of the First International Conference on Object Role Modeling, ORM-1, Magnetic Islands, Australia, 1994.

[8] D.W. Renouf, B. Henderson-Sellers, Incorporating roles into MOSES, in: C. Mingins, B. Meyer (Eds.), Proceedings of the 15th Conference on Technology of Object-Oriented Languages and Systems, TOOLS, Vol. 15, Melbourne, Australia, 1995, pp. 71–82.

[9] W.W. Chu, G. Zhang, Associations and roles in object-oriented modeling, in: D.W. Embley, R.C. Goldstein (Eds.), Proceedings of the 16th International Conference on Conceptual Modeling, ER'97, Los Angeles, California, Lecture Notes in Computer Science, Vol. 1331, Springer, Berlin, 1997, pp. 257–270.

[10] R. Motschnig-Pitrik, The semantics of parts versus aggregates in data/knowledge modelling, in: C. Rolland, F. Bodart, C. Cauvet (Eds.), Proceedings of the Fifth International Conference on Advanced Information Systems Engineering, CAiSE'93, Paris, France, Lecture Notes in Computer Science, Vol. 685, Springer, Berlin, 1993, pp. 352–373.

[11] R. Motschnig-Pitrik, J. Kaasboll, Part-whole relationship categories and their application in object-oriented analysis, in: IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 5, 1999, pp. 779–797.

[12] M. Dahchour, Formalizing materialization using a metaclass approach, in: B. Pernici, C. Thanos (Eds.), Proceedings of the Tenth International Conference on Advanced Information Systems Engineering, CAiSE'98, Pisa, Italy, Lecture Notes in Computer Science, Vol. 1413, Springer, Berlin, June 1998, pp. 401–421.

[13] M. Halper, J. Geller, Y. Perl, W. Klas, Integrating a part relationship into an open OODB system using metaclasses, in: N.R. Adam, B.K. Bhargava, Y. Yesha (Eds.), Proceedings of the Third International Conference on Information and Knowledge Management, CIKM'94, Gaithersburg, Maryland, ACM Press, New York, 1994, pp. 10–17.

[14] W. Klas, M. Schrefl, Metaclasses and their Application, in: Lecture Notes in Computer Science, Vol. 943, Springer, Berlin, 1995.

[15] M. Dahchour, A. Pirotte, E. Zimányi, Materialization and its metaclass implementation, IEEE Trans. Knowledge Data Eng. 14 (5) (2002) 1078–1094.

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

[17] M. Halper, J. Geller, Y. Perl, An OODB part-whole model: semantics, notation, and implementation, Data Knowledge Eng. 27 (1) (1998) 59–95.

[18] W. Kent, A rigorous model of object reference, identity, and existence, J. Object-Oriented Programming 4 (3) (1991) 28–36.

[19] S.N. Khoshafian, G.P. Copeland, Object identity, in: N.K. Meyrowitz (Ed.), Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'86, Portland, Oregon, 1986, pp. 406–416. ACM SIGPLAN Notices 21(11), 1986.

[20] M. Atkinson, F. Bancilhon, D. Dewitt, K. Dittrich, D. Maier, S. Zdonik, The object-oriented database system manifesto, in: W. Kim, J.-M. Nicolas, S. Nishi (Eds.), Proceedings of the First International Conference on Deductive and Object-Oriented Databases, DOOD'89, Kyoto, Japan, 1991, pp. 223–240. North-Holland, Amsterdam, Reprinted in the O2 Book, pp. 3–20.

[21] R.J. Wieringa, W. de Jonge, The identification of objects and roles: object identifiers revisited, Technical Report IR-267, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1991.

[22] Q. Li, G. Dong, A framework for object migration in object-oriented databases, Data Knowledge Eng. 13 (3) (1994) 221–242.

[23] E. Odberg, Category classes: flexible classification and evolution in object-oriented databases, in: G. Wijers, S. Brinkkemper, T. Wasserman (Eds.), Proceedings of the Sixth International Conference on Advanced Information Systems Engineering, CAiSE'94, Utrecht, The Netherlands, Lecture Notes in Computer Science, Vol. 811, Springer, Berlin, 1994, pp. 406–420.

[24] L. Al-Jadir, M. Léonard, If we refuse the inheritance..., in: T.J.M. Bench-Capon, G. Soda, A.M. Tjoa (Eds.), Proceedings of the Tenth International Conference on Database and Expert Systems Applications, DEXA'99, Florence, Italy, Lecture Notes in Computer Science, Vol. 1677, Springer, Berlin, 1999.

[25] T. Clark, J.B. Warmer (Eds.), Object Modeling With the OCL: The Rationale Behind the Object Constraint Language, Lecture Notes in Computer Science, Vol. 2263, Springer, Berlin, 2002.

[26] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language: User Guide, Addison-Wesley, Reading, MA, 1999.

[27] B. Meyer, Object-oriented Software Construction, 2nd Edition, Prentice-Hall, Englewood Cliffs, NJ, 1997.

[28] C.W. Bachman, M. Daya, The role concept in data models, in: Proceedings of the Third International Conference on Very Large Data Bases, VLDB'77, Tokyo, Japan, 1977, pp. 464–476. IEEE Computer Society and ACM SIGMOD Record 9(4).

[29] M.P. Papazoglou, B.J. Krämer, A database model for object dynamics, Very Large Data Bases J. 6 (1997) 73–96.

[30] D. Riehle, T. Gross, Role model based framework design and integration, in: Proceedings of the 13th Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98, Vancouver, British Columbia, Canada, 1998, pp. 117–133. ACM SIGPLAN Notices 33(10).

[31] B.B. Kristensen, Object-oriented modeling with roles, in J. Murphy, B. Stone (Eds.), Proceedings of the International Conference on Object-Oriented Information Systems, OOIS'95, Dublin, Ireland, Springer, Berlin, 1996, pp. 57–71.

[32] E. Sciore, Object specialization, ACM Trans. Office Inform. Systems 7 (2) (1989) 103–122.

[33] B.B. Kristensen, K. Østerbye, Roles: conceptual abstraction theory & practical language issues, Theory Practice Object Systems 2 (3) (1996) 143–160 Special Issue on Subjectivity in Object-Oriented Systems.

[34] B.B. Kristensen, J. Olsson, Roles & patterns in analysis, design and implementation, in: D. Patel, Y. Sun, S. Patel (Eds.), Proceedings of the International Conference on Object-Oriented Information Systems, OOIS'96, London, UK, Springer, Berlin, 1997, pp. 243–263.

[35] B. Pernici, Objects with roles, in: Proceedings of the Conference on Office Information Systems, Cambridge, MA, 1990, pp. 205–215.

[36] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbæk, B. Mahbod, M-A. Neimat, T.A. Ryan, M-C. Shan, IRIS: an object-oriented database management system, ACM Trans. Office Information Systems 5(1) (1987) 48–69. Also in Readings in Object-Oriented Database Systems, Morgan-Kaufmann, Los Altos, CA, 1990.

[37] D. Riehle, Composite design patterns, in: Proceedings of the 12th Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'97, San Jose, California, 1997, pp. 218–228. ACM SIGPLAN Notices 31(10).

[38] C. Chambers, Predicate classes, in: O. Nierstrasz (Ed.), Proceedings of the Seventh European Conference on Object-Oriented Programming, ECOOP'93, Kaiserslautern, Germany, Lecture Notes in Computer Science, Vol. 707, Springer, Berlin, 1993, pp. 268–296.

[39] T. Reenskaug, P. Wold, O.A. Lehne, Working with Objects: The OOram Software Engineering Method, Prentice-Hall, Englewood Cliffs, NJ, 1995.

[40] D.F. D'Souza, A.C. Wills, Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison-Wesley, Reading, MA, 1998.

[41] SINTEF Telecom and Informatics, TIMe: The Integrated Method, Version 4.0, 1999. Available at http://www.informatics.sintef.no/projects/time/-timecomplete.htm.

[42] M.J. Ortín Ibáñez, J. García Molina, Modelado basado en roles con UML, in: P. Botella, J. Hernández, F. Saltor (Eds.), Actas de las Jornadas de Ingeniería del Software y Bases de Datos, JISBD'99, Cáceres, España, 1999, pp. 295–306.

[43] T. Reenskaug, Modeling systems in UML 2.0. A proposal for a clarified collaboration, A proposal to UML 2.0 proposers and the UML Revision Task Force, at http://www.ifi.uio.no/~trygver/documents/2001/uml-20.pdf, 2001.

[44] E. Gamma, R. Helm, J. Johnson, J. Vlissides, Design Patterns Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[45] D. Bäumer, D. Riehle, W. Siberski, M. Wulf, Role object, in: N. Harrison, B. Foote, H. Rohnert (Eds.), Pattern Languages of Program Design, Vol. 4, Addison-Wesley, Reading, MA, 2000, pp. 15–32. (Chapter 2).

[46] T. Elrad, R.E. Filman, A. Bader, Aspect-oriented programming: introduction, Commun. ACM 44 (10) (2001).

[47] E.A. Kendall, Role model designs and implementations with aspect oriented programming, in: Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'99, Denver, Colorado, 1999, pp. 353–369. CM SIGPLAN Notices 34(10).

[48] S. Clarke, R.J. Walker, Composition patterns: an approach to designing reusable aspects, in: Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, Toronto, Canada, 2001, pp. 5–14.

[49] J. van Gurp, J. Bosch, Role-based component engineering, in: I. Crnkovic, M. Larsson (Eds.), Building Reliable Component-based Systems, Artech House Publishers, 2002 (Chapter 7).

[50] P. Cointe, Metaclasses are first class: the ObjVlisp model, in: N.K. Meyrowitz (Ed.), Proceedings of the Second Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'87, Orlando, Florida, 1987, pp. 156–167. ACM SIGPLAN Notices 22(12) 1987.

[51] R. Motschnig-Pitrik, J. Mylopoulos, Classes and instances, Int. J. Intelligent Cooperative Inform. Systems 1 (1) (1992) 61–92.

[52] M. Dahchour, A. Pirotte, E. Zimányi, Definition and application of metaclasses, in: H.C. Mayr, J. Lazansky, G. Quirchmayr, P. Vogel (Eds.), Proceedings of the 12th International Conference on Database and Expert Systems Applications, DEXA 2001, Munich, Germany, Lecture Notes in Computer Science, Vol. 2113, Springer, Berlin, 2001, pp. 32–41.

[53] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, MA, 1983.

[54] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis, Telos: representing knowledge about informations systems, ACM Trans. Office Inform. Systems 8 (4) (1990) 325–362.

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

[56] M. Jarke, M.A. Jeusfeld, C. Quix, ConceptBase V5.1 User Manual, Technical Report, RWTH Aachen, Germany, August 1999.

[57] A. Albano, G. Ghelli, R. Orsini, Fibonacci: a programming language for object databases, Very Large Data Bases J. 4 (3) (1995) 403–444.