Number: **165**

The Design of an Object-Role Database Management System

Raymond K. Wong* H. Lewis Chau* Frederick H. Lochovsky

Department of Computer Science

Hong Kong University of Science & Technology

Clear Water Bay, Hong Kong

{wongkk,lewis,fred}@cs.ust.hk

Abstract

In many class-based object-oriented database systems the association between an instance and a class is both exclusive and permanent. Therefore, these systems have serious difficulties in representing objects taking on different and multiple roles over time. Recently, some researchers have proposed the use of roles to tackle these problems. In their approaches, objects acquire additional properties by dynamically playing roles. However, relationships between objects and roles have not been addressed. Therefore, an object may evolve on its own by dynamic acquiring new roles, without coordination or cooperation by any other objects.

In this paper, a novel object-oriented database management system, called DOOR, which supports object evolution, dynamic role (context-dependent) modeling, objects of multiple specific classes, and object-role relationships, is described. In DOOR, a role is an entity with state and behavior, but does not have globally unique identity. Therefore, its existence has to be associated with an object. It acts as a special association between its owner and player, such that its owner can prescribes its state and its player gains its properties through dynamic role playing. In this way, an object can evolve dynamically and cooperatively according to its associating objects. Furthermore we discuss some interesting features of roles which have been seldom addressed. They include playing multiple roles of the same type, player change (or role migration), role ownership and playership, and player-class constraint, etc. We show by examples that all these features are very useful for applications in which objects take on different and multiple roles over time.

*Part of this work has been done when the authors are visiting the Computer Science Department at UCLA.

1. Introduction

Most object-oriented data models are based on the notion of class. In these models, real-world entities are represented as instances of the most specific class[1]. In reality, however, objects often belong to several most specific classes. For example, a person `John` might play multiple roles at time t_0. He may be a graduate student, a teaching assistant and research assistant, a club-member and chairman at the same time, as shown in Figure 1. Thus, the object representing this person does not have a

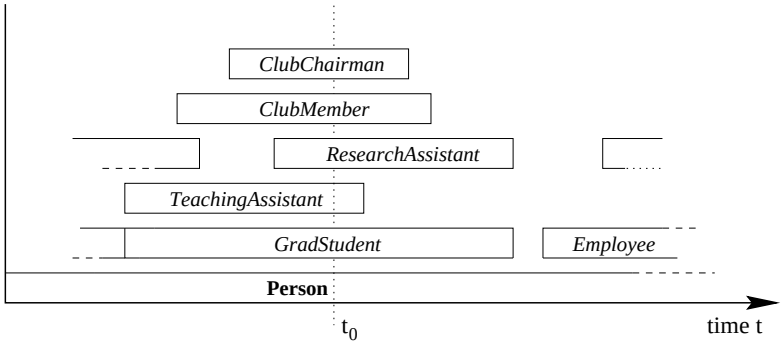

Figure 1: A possible evolution of object `John`.

unique most specific class, but rather has a set of most specific classes. Although this situation can be easily represented in a model with multiple inheritance by defining a subclass of all the involved classes, this solution may lead to a combinatorial explosion of artificial subclasses. Another shortcoming of the multiple inheritance approach is that it provides only a single behavioral context for an object [14].

Moreover, in the conventional class-based object-oriented approach, the association between an instance and a class is exclusive and permanent. Therefore, this approach is appropriate only if the entities to be modeled can be partitioned into a set of disjoint classes and never change their class. The problem is even more cumbersome if an entity can take on several roles simultaneously. However, many real-world applications are dynamic and encompass entities that evolve over time. Person entities give the most illustrative example. A person may take on different roles at different times. He/she may become a student, a club member, and then an alumni, an employee, and so forth. But a person is not the only kind of entity which evolves over time; so does an office document or a product in a production line, etc.

[1] An object o can belong to a set of classes S. We call an element of S that has no subclass in S a most specific class of object o.

Recently, some researchers have proposed extending object models to incorporate the concept of roles to tackle these problems in various application domains. These include office modeling [13], semantic modeling [14, 16], object-oriented modeling [12], manufacturing system modeling [22], and multimedia applications [21]. In these approaches, a role extends an existing object with additional state and behavior. An object may have many roles that come and go over time. Rather than being an instance of some unique subclass defined through multiple inheritance, an object simply is an instance of many types by virtue of having many roles. Every object reference is to a particular role, and the behavior of the object depends on which role is being referenced.

The restriction that an object be associated with a single, most-specific type in the database context was first relaxed in Iris [4]. Iris allows an object to belong to several types. But it misses the possibility of role-specific / content-dependent behavior, i.e., the entire set of types an object belongs to is visible in every context. Hence two roles of an object may not have different methods of the same name. Afterwards, the importance and support of multiple perspective/context-dependent behavior of objects were described in *multiple views* [17], *ORM* [13], and *Aspects* [14]. However, in these approaches, roles are not classified and encapsulated as classes and there is no inheritance or delegation defined between roles. Hence, role sharing among different classes is impossible. Moreover, no explicit operators for switching between roles were defined. Sciore's work [16] allows classes to be viewed as an individual object's auxiliary roles or perspectives, and objects to define their own inheritance paths. However, this approach is biased towards the prototyped-based approach and is more appropriate for experimental phases of system development, as opposed to database design. A similar idea was proposed by Schrefl and Neuhold in [15], except this approach towards class-based instead of prototype-based, that possible object hierarchies must be predefined by role specialization classes at the type level.

The most recent languages which support roles include a new, strongly-typed database language called Fibonacci [1, 2] and a Smalltalk-based role extension to objects [5]. In Fibonacci, an object simply consists of an identity and an acyclic graph of roles. Each role can be dynamically added or dropped. Objects are defined in classes and roles are defined separately and form a different hierarchy. Alternatively, Gottlob *et al.* [5] demonstrated the extension of Smalltalk for incorporating roles and emphasized the way to extend an existing language to support roles. Moreover, different from Fibonacci, they included multiple instantiation of roles, and the integration of class and role hierarchies. To some

extent, both Fibonacci and Gottlob's work are similar to ORM in the sense that roles are also rooted in (though not encapsulated in) a class. Different from ORM, aspects, and views, however, the roles attached to a class in both approaches can form their own *is-a* hierarchy. However, relationships between objects and roles have not been addressed in all these work. Therefore, an object may evolve on its own by dynamic acquiring new roles, without coordination or cooperation by any other objects. For example, a person object may gain a manager role and initialize its state on its own. Although a company object/name may be referenced by one of its attributes, the *ownership* information of the role is missing. That is, if that person later resigns the job, should the properties of the manager still persist and be ready for the new person who fills the vacancy? More naturally and reasonably, the detailed properties and initial state of a manager role should be prescribed by his/her company (the role *owner*) and gained by the one (the role *player*) who plays this role.

This paper presents an object-role database system called DOOR, whose goal is to provide generalized role support for dynamic and evolving applications. DOOR is based on an object-with-role model. All real-world entities are classified as either object classes or role classes. Their instances are called objects and roles respectively. A role extends an existing object with additional state and behavior while sharing the same object identity. Therefore, its existence has to be associated (by means of *played-by* and *owned-by* relationships to be described) with objects. In particular, DOOR supports the operations for dynamic role playing and context-dependent behavioral modeling. Since an object can play multiple roles and acquire them at the instance level, instantiation of multiple specific role classes are supported such that the association between an instance and its role classes are neither exclusive nor permanent. Most importantly, we emphasize the relationships between roles and objects, as well as the persistent and transient properties of objects. In DOOR, a role acts as a special association between its owner and player, such that its owner can prescribes its state and its player gains its properties through dynamic role playing. In this way, an object can evolve dynamically and cooperatively according to its associating objects. Furthermore we discuss some interesting features of roles which have been seldom addressed. They include playing multiple roles of the same type, player change (or role migration), role ownership and playership, and player-class constraint, etc. We show by examples that all these features are very useful for applications in which objects take on different and multiple roles over time.

The organization of the rest of this paper is as follows. Section 2 reviews the data model briefly and

presents a fragment of a university database as an example to be used throughout the paper. Section 3 introduces the basic programming and query constructs of DOOR. In Section 4, important issues such as an object with multiple most specific types, multiple roles of the same type, context-dependent behavior, and polymorphism of roles are discussed. These properties are supported by various facilities in DOOR which include path expressions, attribute name conflict resolution, different method lookup schemes, generic comparison operators, and different levels of constraints. Section 5 describes the different relationships (mainly playership and ownership) between objects and roles. Object evolution based on these relationships is presented. Finally Section 6 concludes the paper.

2. An Object-Role Data Model

2.1. Informal Overview

We briefly review the object-role data model formally defined in [19, 20]. The model extends a typical object-oriented data model (e.g. [7, 10]) with the notion of roles.

Object and role representation: Objects consist of both object state (in terms of the values of attributes), methods, and a set of dynamically changing roles. They are referred to via their logical object ids (oid) and any oid uniquely identifies an object. Oids may carry certain semantic information. For instance, we consider '20' to be the oid of the abstract object with the usual properties of the number 20. The object state is encapsulated and can only be queried and updated by sending messages to the object. An object is internally organized as an acyclic graph with the root being the object itself, and all the other nodes being roles. The parent node of any node A in the graph is called the *player* (or role player) of A, and A is said to be *played-by* its player. A role is also an entry to access the object it belongs to: an object can be accessed through itself (we consider the object itself as a base role) or one of its roles, and its behavior depends on this role. On the other hand, roles encapsulate both state and behavior, but do not have a persistent, globally unique identity. A role can be itself a player and include other roles being played.

Attributes and methods: Objects are described via attributes, and all our objects are tuple-objects, whose fields are the values of the object's attributes. If the attribute is single-valued, then the value is a single oid; if the attribute is set-valued, then the value is a set of oids. Since DOOR is strongly typed, a type signature needs to be assigned to each attribute in a class definition. If the

signature is an object type, the attribute value must be of that type or any subtype of that type. If the signature is a role type, then the value must be an object which is playing a role of that type or of a subtype of that type. Moreover, an object/a role X can *own* a role R such that R becomes part of the properties of X. Another object can share these properties by playing R. As a result, R bridges the relationships between its owner and its player. This issue will be discussed extensively in the section on object-role relationship modeling. A method, invoked in the scope of an object (or a role) on a tuple of arguments, returns an answer, and, possibly, changes the state of that object (e.g., by changing the value of an attribute). As a function, each method has an arity – the number of its arguments. An attribute is regarded as a 0-ary method.

Object class and role class: Object classes have the function of organizing the persistent properties of objects into sets of related entities, while role classes organize their transient properties. The instance-of relationship between objects (or roles) and classes determines which objects (or roles) belong to which classes. The IS-A or subclass relationship, is defined between classes and is acyclic. If a class C is a subclass of another class C', then all instances of C must also belong to C'. A player-class constraint can be optionally defined in the role class, to limit the possible player types of a role. If it is omitted, a player of any type is assumed. The player-class constraint is used to support the type-safe implementation of the methods in roles, as it may call methods in a role player. Besides the player-class constraint, other general constraints can be defined in the class-level and/or instance-level to model the fact that not every object is qualified to play a particular role. Similar to the other properties of a class, the player-class constraint of a class will be inherited by all its subclasses.

Types: The type of a class C is determined by the types of its methods, described as a signature of the form $\texttt{Meth} : \texttt{Arg}_1, \ldots, \texttt{Arg}_n \rightarrow \texttt{Result}$, or $\texttt{Meth} : \texttt{Arg}_1, \ldots, \texttt{Arg}_n \twoheadrightarrow \texttt{Result}$, for single-valued or set-valued methods, respectively. The signature is attached to the definition of class C, where $\texttt{Arg}_i$ and `Result` are class names, and means that when arguments that are instances of classes $\texttt{Arg}_1, \ldots, \texttt{Arg}_n$, respectively, are passed to the method `Meth`, the result is expected to be an instance, or a set of instances, of the class `Result`, depending on whether `Meth` is single- or set-valued, respectively. Note that there are actually $n + 1$ (rather than n) arguments, where the 0^{th} argument is not mentioned, because it is the object of class C for which the signature is defined.

A method can have several signatures, each constraining the behavior of the method on different sets of arguments. When this is the case, the method is said to have a polymorphic type. The signature of a method can include role types. If a role type is included in the method signature, the corresponding object must be playing such a role and will be treated context-dependently from that perspective. Otherwise, a type violation is caused. The type of an object is more complicated and its formal description is beyond the scope of this paper. Informally, an object type consists of a static component, i.e., the type of its object class, and a dynamic component, i.e., the types of the roles being played.

Inheritance and delegation: Methods, and the player-class constraints if there are any, defined in the scope of a class C are inherited by the subclasses of C through the is-a relationship. If there are different player-class constraints defined in a subclass, a most specific class will override a relatively more general one until all of them are disjoint. Inheritance is not defined for the played-by relationship. Instead, the automatic *delegation* between roles and their corresponding players is used. For example, suppose we model an employee e as a role of a person p, and *sex* is an attribute of person but not of employee. Then $sex(e)$ would be a type error. We can correct this error by delegating the evaluation of *sex* to *played-by*(e) [11]. This amounts to replacing *sex(e)* by *sex(played-by(e))*.

2.2. Example: A University Database

In this subsection, a schema for a university database is used to illustrate the above object-role data model.

About the university: There are several departments in the university. Each department has many undergraduates, graduate students, teaching assistants (TA), research assistants (RA), faculty, administrative staff (AdminStaff), and a department head (DeptHead). All TAs, RAs, faculty, and of course AdminStaff are regarded as university employees. A DeptHead may be employed directly from outside, or elected from the existing faculty, and he/she has to perform also the duties of a faculty member (which include teaching and research). Each faculty may be involved in more than one research project. They can hire graduate students, or some outstanding undergraduates, to work as RAs for the projects. Each project may have one or more than one project-leader(s).

A project-leader is himself/herself a faculty or a RA. Different from being a RA, only graduate students can be employed as TAs, to tutor the undergraduates. Moreover, there are some interest clubs for students, faculty and even off-campus people (but they need to pay a higher membership fee) to join. Each club has at least one chairman. As usual, in order to be a chairman, he/she needs to be a member beforehand.

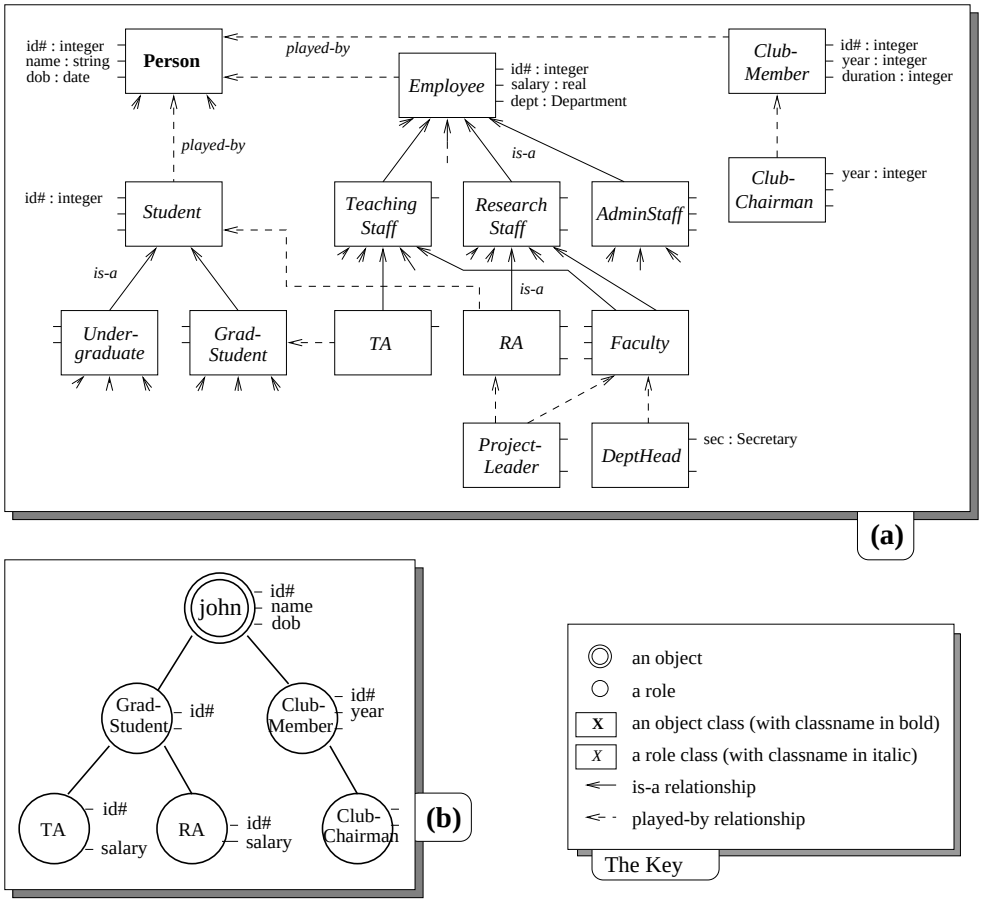

Figure 2: **(a)** An object-role database schema. **(b)** Internal organization of an object – a person `john` who is playing multiple roles: `GradStudent` (graduate student), `TA` (teaching assistant), `RA` (research assistant), `ClubMember` and `ClubChairman`.

About the schema: Figure 2(a) shows the corresponding database schema. For simplicity, attribution and composition are not shown. The schema is similar to an traditional object-oriented database schema extended with the played-by relationship which specifies the player-class constraint. Inheritance is defined along each is-a relationship, from a class to its subclasses. Apart from attributes and methods, the player-class constraints of a role class will also be inherited to all its subclasses.

Overriding is allowed. However, similar to the overriding of methods, the newly defined player-class constraint must be more specific than the one to be overriden. For example, the player-class constraint for *Employee* is **Person**, while the one for *TA* and *RA* are overwritten to *GradStudent* and *Student* respectively. Also the player-class constraint for *ProjectLeader* is the disjunction of *RA* and *Faculty*.

About the internal organization of an object: As mentioned previously, a role can be played by an object, or even by another role. As a result, an object can be represented as an acyclic graph with the root being an object itself and all the other nodes being roles. For example, consider a person object `john` playing multiple roles, with internal organization shown in Figure 2(b). `john` is said to be the root player of all the roles in the acyclic graph. In this graph, the `id#` associated with different nodes have different meanings. For example, the `id#` of the root `john` denotes his identity number given by the government, the `id#` of role `GradStudent` denotes `john`'s student number when he is considered as a graduate student, the `id#` of role `ClubMember` denotes `john`'s membership number when he is considered as a club member, and so on. Hence, the context-dependent modeling of objects is supported with this organization of roles. Moreover, when we ask for the membership number of `john` from his `ClubChairman` perspective, the message will be delegated to its role player, i.e., ClubMember, to get the `id#`. Dynamic, multiple role playing and hence object evolution (as the behavior of an object changes) are supported by dynamically inserting or deleting roles from the acyclic graph.

Comparisons with Fibonacci: Apart from the object-role relationships (i.e., the playership and ownership introduced previously and to be described in Section 5), DOOR shares similarities primarily with Fibonacci, as: both are strongly typed and support dynamic binding; both have separate hierarchies for object classes and role classes; both support dynamic object extension and contraction through dynamic role playing and role dropping respectively. However, they do have subtle differences, as described as follows.

As objects contain their own state and methods (while in Fibonacci, objects consist of only identity and roles), if role constructs are never used, DOOR objects are structurally and behaviorally exactly the same as classical objects, i.e., with only state and methods. Moreover, we can always assume that the creation of an object includes the creation of a 'base role' [13] such that every

object has a base role type (i.e., the static object type), which describes the initial characteristics of an object upon creation and the persistent global properties under its evolution. We must point out that persistent properties of an object can be as important as its dynamic behavior (by means of roles) in certain application domains. We claim that our approach is more general than the object representation in Fibonacci, because we can always define a dummy object with no state and behavior of its own but with different roles to play. Hence an object will simply be a collection of roles together with its identity. In Fibonacci, on the other hand, objects cannot be manipulated independently of their roles [2] and roles can be dynamically changing, so the global and persistent part of an object's characteristics are lost. Moreover, roles are identified by their class names and we have implemented an abstraction mechanism based on subtype polymorphism such that a role can be identified by any superclass of its class. However, only the behavior defined in that superclass can be accessed from the role.

3. Database Programming and Query Environment

This section describes the basic programming and query constructs that support object-role modeling in DOOR. These constructs include the creation of classes, objects, and roles. Moreover, we also illustrate the use of `select` and `foreach` statements for the objects extended with roles.

`Create` is a generic constructor in the DOOR programming and query environment. Specifically, `create object-class`, `create role-class`, `create method`, `create object` and `create role` are the constructors for a new object class, role class, method, object and role, respectively. As shown in Script 1 and Script 2 of Figure 3, the object classes `MAMMAL` and `PERSON`, and the role classes `STUDENT` and `GRADSTUDENT` are defined, respectively. The keyword **subclass-of** represents the *is-a* relationship in the schema and **played-by** represents the *played-by* relationship. The body of the class definition, which is similar to a traditional class definition, is self-explanatory. Similar to CLOS, methods are defined outside the classes, with an argument that specifies the class to which it belongs.

Similarly, objects are created with the constructor **create object**, as illustrated in Script 3. An optional global variable, *andy*, can be specified and will be bound to a particular object in the database. Further reference to this variable is equivalent to the reference to the bound object. Alternatively, if the global variable is not specified, as shown in Script 4, **create object** simply creates an object in the

```
Script 1:
(create object-class MAMMAL:
  STRING sex;
  DATE date-of-birth)
(create object-class PERSON subclass-of MAMMAL:
  INTEGER id#;
  STRING lastname, firstname, midname)
(create method PERSON age():
  return year(today() - date-of-birth))

Script 2:
(create role-class STUDENT played-by PERSON:
  INTEGER id#;
  DEPT dept)
(create role-class GRADSTUDENT subclass-of STUDENT:
  INTEGER office;
  FACULTY advisor)
```

```
Script 3:
(create object PERSON andy:
  id# is 96112038;
  name is "Andy";
  sex is "male")

Script 4:
(create object PERSON:
  id# is 96112038;
  name is "Andy";
  sex is "male")

Script 5:
(define object PERSON john:
  name is "John";
  sex is "male")
(insert john into University-Database)

Script 6:
(create role andy GRADSTUDENT:
  id# is 2069694;
  advisor is joe;...)
```

Figure 3: Example scripts for class, object, and role creation.

database and further retrieval of the object has to be done through the `select` or `foreach` statements.

Classes, objects and roles can be created in memory only through the generic constructor `define`, with usage the same as for `create`, and stored in the database only if needed. For example, in Script 5, an object *john* is created in memory that can be inserted into database explicitly if needed. This feature is useful for testing, or trial-and-error *adhoc* query construction or database prototyping. Similar to objects, roles are created and played with the constructor **create role** as shown in Script 6.

Script 7 shows a simple DOOR select statement (*cf.* OSQL and OQL in [8]) that selects all female names. Script 8 shows another example which selects *id#* and *name* from every object *s* playing a role as a `STUDENT` of either the `Computer Science` department or the `Electrical Engineering` department. To support batch creation, a **foreach** construct is provided. Its syntax is similar to the **select** statement mentioned above, with an additional action part after the keyword **do**, as shown in Script 9. In this example, each student who is not playing the `LIBRARY-CARD-OWNER` role is updated to play it, with the

```
Script 7:
(select o.name from o is-a PERSON
  where o.sex == "female")

Script 8:
(select s.id#, s.name
  from s is-a STUDENT
  where s.dept in (select d
    from DEPT
    where (d.name = "Computer Science")
      or (d.name = "Electrical Engineering")))
```

```
Script 9:
(foreach o is-a STUDENT
  where not(o is-a LIBRARY-CARD-OWNER)
  do
    (write :console "Name:" o.name "Lib-ID:");
    (create role o LIBRARY-CARD-OWNER:
      id# is (read :console);
      year is 1996))
```

Figure 4: Example scripts to illustrate the `select` and `foreach` statements.

initialization of attributes as specified. The **read** and **write** commands are used to attain the value externally (from the console interactively) with the creation of each role.

4. Objects with Multiple Role Playing

This section describes issues involved in supporting multiple role playing (of different types or of the same type), context-dependent behavior modeling, and polymorphism of roles. These issues include path expressions, attribute name conflicts, different method lookup schemes, various object and role comparison operators, and different constraints for roles.

4.1. Path Expression

The basic notation to access a role r of object o is specified using '!', i.e., *object*!*role*. For example, `john!ClubMember` means access the `ClubMember` role of object `john` in Figure 2(b). In other words, we consider `john` from his `ClubMember` perspective. Whenever a role cannot be found according to the expression, a *role_not_found* exception is raised. For example, `john!GradStudent!TA#` will not raise a *role_not_found* exception while `john!ClubMember!TA#` will. We can specify the attribute `id#` of `john` from the `TA` context simply by the exact path expression `john!GradStudent!TA.id#`.

However, in some cases, two roles may have exactly the same playing sequence (or acquisition sequence). For example, a person `peter` may play two roles of the same role class `ClubMember` with different values for the attribute `clubnames`. Since a role is identified by its role class name and its value, we need some role selection mechanism based on the role's value. To resolve this,

DOOR provides an optional role selection criteria based on the symbol '| <*boolean expression*>'. For example, we can express the attribute `id#` of a particular `ClubMember` of `peter` by writing `peter!(ClubMember|clubname="CS Club").id#`.

4.2. Attribute Name Conflicts

The semantics of attribute inheritance are crucial because attributes are the places that hold the state of an object. To resolve the ambiguity due to the name conflicts arising from the different roles being played, the keyword `UNIQUE` is used to specify if attributes with the same names actually denote the same state variable. Otherwise, name conflict is automatically solved by accessing object from different roles.

As we discussed in the previous section, the different `id#`s of `john` in Figure 2 mean different things depending on which context/perspective we consider. Name conflict, as in the one caused by multiple inheritance, is solved as attributes, with the same name, of different roles of the same object can be accessed independently. For example, we can access `john.id#` and `john!GradStudent.id#` independently although both attributes are named the same (i.e., `id#`), where `john.id#` denotes his personal identity number assigned by the government and `john!GradStudent.id#` denotes his student identity number given by the university. However, is `john!TA.id#` $\neq$ `john!RA.id#`? To resolve attribute *name conflicts* such as this, we employ a methodology similar to the one suggested in [3]. The idea of name conflict resolution is as follows. Informally, the state of an object with multiple playing roles, which belongs to its parent object class together with several most specific role classes, is the union of the attributes in those classes. However, the sets of attributes in those classes may not be disjoint, that is, name conflicts may arise. To handle these situations we introduce the notion of the *source* of an attribute. Intuitively, if an attribute belongs to the intersection of the attribute sets of two classes and it has in both classes the same source, that is, it is inherited from a common superclass, then the attribute is semantically unique, and thus the object must have a unique value for this attribute. If, by contrast, the attribute has different sources, then the two attributes in the two classes have accidentally the same name, but represent different information that must be kept separate. With roles this different information can be then accessed according to different contexts.

This approach is used in [3] for all cases. However, it cannot address many situations in which the

attributes of different roles need to store different values even when they come from the same source. A trivial example is that `john!TA.salary` is (in general) different from `john!RA.salary` although `salary` is defined in only a single source, i.e., *Employee*. Moreover, although the source of the attributes `john!TA.dept` and `john!RA.dept` is unique, i.e., the `Employee` role class, they may have different values in a real situation. That is, John may work as a teaching assistant in the Computer Science Department and also as a research assistant in the Electrical Engineering Department. Therefore, in DOOR, the resolution similar to the one in [3] is used only for those attributes defined with a keyword `UNIQUE`. For all other attributes, the object may have two different values for attributes with the same name in different roles even though they are defined in a single class.

Therefore, the attribute `id#` in role class `Employee` can be defined as `UNIQUE` so that `john!TA.id#` and `john!RA.id#` are not only equal but semantically the same attribute. Alternatively, we can also define it without the keyword `UNIQUE` if we want to have two different identity numbers for TA and RA even if they are for the same person.

4.3. Method Lookup

In traditional object-oriented languages, every message is dispatched only to the most specific class of an object, which either has a method for the message, or looks for a method in its superclasses. The idea of this message dispatching for the methods defined along the inheritance hierarchy (or is-a hierarchy) is still applied to each object, and each role, in DOOR. Indeed, in DOOR, this method lookup scheme is augmented with a similar idea called delegation [18] along the played-by relationship of the roles being played by an object. Two method lookup modes are supported:

Upward lookup: the method is looked up first in the receiving role and then in its ancestor players.

Double lookup: the method is first looked up in the receiving role, and then in all the descendant roles of the receiving role, and finally in its ancestor players.

They are illustrated in Figure 5 by assuming a message is sent to `john!GradStudent`. The default lookup mode is upward lookup. Upward lookup is used by the assumption that more general information about an object obtained from its particular role without the requirement of special privillege. Double lookup is used to access all information of a particular context, plus its general information can be accessed from that context. In fact, for these two lookup modes, DOOR insists that the method be first looked

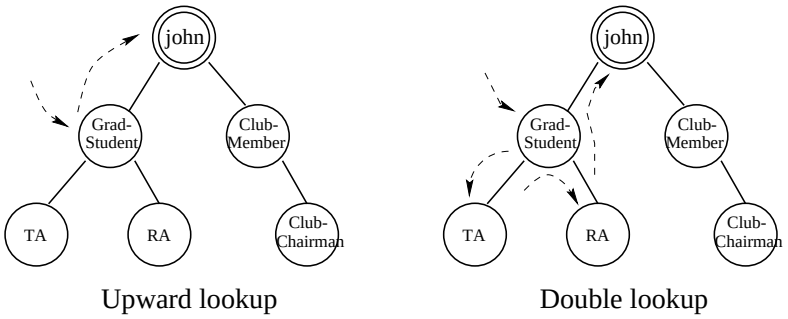

Figure 5: Illustration for the two method lookup modes.

up in the receiving role in order to achieve clean semantics of self recursion (i.e., a method which sends message(s) to itself). In Fibonacci [2], only two lookup modes are provided: upward lookup and double lookup. For its double lookup, the descendants of the receiving roles are looked up before the receiving role. Hence the semantics of self recursion is unclear.

4.4. Object and Role Comparisons

As an object includes a collection of roles being played, a set of type inquiry operators and comparison operators are provided. The type inquiry operators are used to query both the *persistent type* and *transient type* of an object. A type of an object is persistent if it does not change during the object lifetime, otherwise, it is transient. The set of type inquiry operators is defined as follows:

is-always: `Object` $\times$ `Object-Class` $\longrightarrow$ `Boolean` is used to query the persistent type of an object. *is-always(o, oc)* returns true if o is of object type oc. For example, `is-always(john, Mammal)` returns true if `Person` is a subclass of `Mammal`, and `is-always(john, Cat)` returns false.

is-a: `Object` $\times$ `Object-Class`$\cup$`Role-Class` $\longrightarrow$ `Boolean` is used to query the transient type (including persistent type) of an object. *is-a(o, c)* returns true if o is of object type c or it is playing a role (directly or indirectly) which is of role type c. For example, `is-a(john, Person)` returns true and `is-a(john, Student)` returns true.

can-play: `Object` $\times$ `Role-Class` $\longrightarrow$ `Boolean` is used to query about whether an object is qualify to play a role of a particular role class. *can-play(o, rc)* returns true if o is qualified to (directly or indirectly) play a role which is of role type rc.

roles: `Object` $\longrightarrow$ `bag-of( Role-Class )` is used to find out all the roles currently being played by an object. *roles(o)* returns a bag of roles being played by object o. Here a bag is used as a return

type instead of a set because an object may play multiple roles of the same role class. For example, `roles(john!GradStudent)` returns a bag of `TA, RA`, denoted by ⟨`TA,RA`⟩, and it is possible for a person `p` to play two RA roles from two different projects, i.e., `roles(p)` returns ⟨`RA,RA`⟩.

The following is a set of equality operators:

Object Identity: Two objects o_1 and o_2 are identical, denoted by *identical*(o_1, o_2), if they are the same object.

Shallow Equal: Two objects o_1 and o_2 are shallow equal, denoted by *shallow-equal*(o_1, o_2), if their values are identical.

Deep Equal: Two objects o_1 and o_2 are deep equal, denoted by *deep-equal*(o_1, o_2), if their values are the same.

The values of an object depend on the values of the attributes of an object, the values of the attributes of the roles being played, and the method lookup scheme being used.

All the type inquiry and comparison operators support the context-dependent characteristics of objects. For example, referring to Figure 2, `is-a(john, Student)` returns true, and `is-a(john!ClubMember, ClubChairman)` returns false. Similarly, `deep-equal(john!GradStudent, peter!GradStudent)` compares the values of objects `john` and `peter` from `GradStudent` perspective.

4.5. Player-Class Constraints versus General Role Constraints

In general, there are many cases where a role should not be rooted to a particular object class (such as [5, 13]) because objects of different disjointed classes may be qualified to play a particular role. Otherwise, an artificial superclass needs to be created for these disjointed classes such that the role class of that particular role can be rooted to it. For example, a library card holder must be either a student or a faculty (but not both a student and a faculty), a research project-leader can only be either a faculty or a RA, etc. With a non-exclusive link, by means of the player-class constraint, between an object class and a role class, the above problem is avoided. A player-class constraint can be specified in the role class definition using the keyword **played-by**, as demonstrated in Script 2. In some cases, an object of multiple specific classes (playing multiple roles) may be required in order to be qualified to play a particular role. However, discussion of this conjunctive player-class constraint is beyond the scope of this paper.

The constraint issue is not addressed in most of the related work on roles, including Fibonacci [1, 2]. Although the concept of role constraints has been mentioned in some work on roles (like constraints in the transition rules in [13], role class hierarchies rooted in an object class [5], and the transition rules in the role classes in [22]), they are too restrictive to be defined at the type/class level. For example, we may have a new role Project-Leader, which can only be played by either a RA or a faculty member in Figure 2(a). Gottlob *et al.*s work [5] cannot model this situation without creating another superclass for RA and faculty and rooting the Project-Leader under this newly created artificial class. On the other hand, in [13], all role constraints are defined at the class level. However, many real-world applications require different role constraints for different object instances. For example, although each department requires a TA to be a graduate student, it is possible and natural that the Mathematics Department might require their TAs to come from the same department, while students from the Mathematics Department, Electrical Engineering Department, and Computer Science Department all can be TAs of the Computer Science Department. Such constraints should be defined in the owner (object level) of the TA roles, i.e., the individual departments.

Moreover, the importance of player-class constraints has been previously overlooked. It would be useful for a role to access its player, e.g., by calling its method. However, type safety cannot be guaranteed if we cannot constrain the possible types of a role player. Obviously it will be a disaster if a method in TA calls a method defined in GradStudent but not Undergraduate, and, peter, being an undergraduate, tries to play a role as a TA. Therefore, the player class constraint in DOOR is for the sake of type safety rather than to increase modeling power.

Unlike most of the existing systems that have role constraints specified in object classes [13], we have player-class constraints that can optionally be specified in the role classes so that the specification of object class definitions is the same as that for traditional class definitions. Therefore, if roles are never used, the definition of classes and manipulation of objects are exactly the same as traditional class-based object-oriented systems. On the other hand, even when roles are to be used, users can never specify the player-class constraints in all the role classes such that the role definitions are the same as those in Fibonacci (i.e., without being concerned about whether a player is qualified to play a role), or just specify a single class as a player-class constraint to model a unified class hierarchy as described in [5]. In other words, our approach (based on player-class constraints) is more general and flexible than

the other existing approaches.

5. Relationships between Objects and Roles

Apart from the traditional associations (e.g., aggregation) between objects, we describe the modeling of relationships among roles, or between objects and roles in this section. These relationships model the dynamic relationships between entities as they evolve over time. Entities can easily gain additonal properties or give up part of their properties by establishing these links or dropping them respectively. Before we go on to the modeling aspect, let us describe another way to create a role. In addition to the **create role** construct mentioned previously, a role can be created with the creation of an object, or another role, by being owned by it. This can be done by specifying the keyword **own** before an attribute declaration of a class definition. If the attribute is of a role class, its value (a role) will be created automatically with an instantiation of the class. If the attribute is of an object class, an *exclusive* composition [9] is assumed between the class instance and the attribute's value.

```
Script 10:
(defclass object DEPARTMENT:
  STRING name;
  own DEPTHEAD head;
  own {FACULTY} facs;
  ...)
(defclass role DEPTHEAD played-by FACULTY:
  own SECRETARY sec;
  FACULTY associate;
  ...
  PRE:
    owner(self)=owner(player) ∧ ...)

Script 11:
(create DEPARTMENT csd:
  name is "Computer Science";
  head.sec is judy; ...);
  ...)
```

```
Script 12:
(update csd:   head is ray)
(update ray:   associate is joe)

Script 13:
(update csd:
  head is ray;
  head.associate is joe)

Script 14:
(update csd:
  head is vicki)
```

Figure 6: Example scripts to illustrate the ownership of roles.

Suppose a department and a department head are defined for the university database (Figure 2(a)) as in Figure 6. A department owns a DEPTHEAD role and a set of FACULTY roles. A department head

owns a SECRETARY role, and there is also an assistant (*associate* department head) for him/her. Then the object *csd*, computer science department, is created with the department head's secretary being *judy*. Up to now, the department head is still undefined, but the department can preset some properties for the *head* such that the one who picks this role up will possess these properties. In this case, *judy* will be the *head*'s secretary regardless of whom the *head* will be. So after this, *judy* is playing a role as a secretary of the department head. Then, in Script 12, *ray* becomes the *head* and he chooses *joe* as his assistant (the object-role relationship is visualized as in Figure 7). In this case, if *ray* steps down later and *vicki* becomes the new *head*, the secretary will be the same (i.e., *judy*) but the value of *associate* will be dropped and become undefined again. So *vicki* has to choose her own assistant from the faculty.

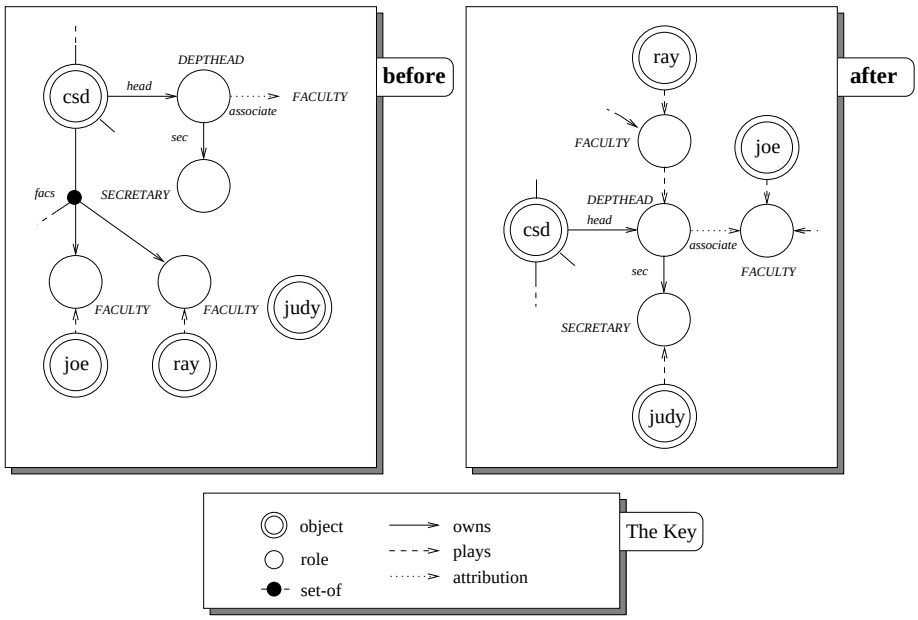

Figure 7: Object-role relationship for the computer science department.

Alternatively, the department can elect or assign *joe* as the *associate* department head by either initializing *associate* as the creation of *csd* (same as initializing *judy* as the secretary), or assigning *joe* as *associate* under the update of *csd* as shown in Script 13. In this case the value of *associate*, and also *sec*, will be retained if *ray* abandons the role as *head*. Therefore, if *vicki* later picks this role up and becomes the new *head*, she does not need to reassign these values. The department can update *associate*, and preserve its value even when the player of *head* is no longer defined. This is because the

attribute *associate* is of a role owned by (i.e., with the keyword **own**) the department. As described previously, the definition of **own** is transitive. Note the difference between an attribute with a keyword **own** and without. For example, *judy* starts playing a role as a `SECRETARY` after she fills the job *sec* of the department, but *joe* does not become a `FACULTY` member because of being assigned as *associate*. In fact, he needs to be a `FACULTY` member in order to be the *associate* department head.

A role acts as a bridge (for abstraction and information sharing) between its owner and player while the ownership of the information is clearly defined. For example, a university can revise the salary for department heads without knowing who the dept head of *csd* is, i.e., without accessing the actual object, say *joe*. It updates the salary by updating *csd.head.salary*. This is different from and better than having a reference pointer that points to *joe*, as the university can update the salary of the department head even if the position is open (i.e., if the university does not know which object the department head is). This models the fact that the role of a department head is actually defined by (or owned by) the department, not by the object who is going to be the department head.

As there is no globally unique identifier for a role, it can only be referenced through the played-by and/or owned-by links from an object. A role is identified through its role classname and value. Therefore, polymorphism is supported. Furthermore, to solve the reference problem caused by the object update, for example, one can refer to the department head of *csd* through *csd.head* (which is a role) instead of an object. Whenever a message is sent to it, it will be delegated to the object that is playing the role. If no such object is playing it, an exception is raised.

Object Evolution with the Changing Object-Role Relationship We have described the use of roles to link the relationships between objects dynamically, such that object X playing a role R owned by object Y (through the attribute r) can extend itself with the properties of R, and other objects can reference X through Y.r. As X keeps establishing links between different objects, and dropping some of its connected links, its behavior will be changed (extended and contracted from time to time). Moreover, these links (i.e., role playing) also represent the dynamic relationship between different entities. To further explain this, consider an object `John` which evolves during its lifetime, as shown in Figure 8. `John` first plays a role as one of the high-school students (*HS-Student*) of `HK High School`. At that time, the dynamic (or temporary) relationship between `John` and `HK High School` is built, and `John` gains

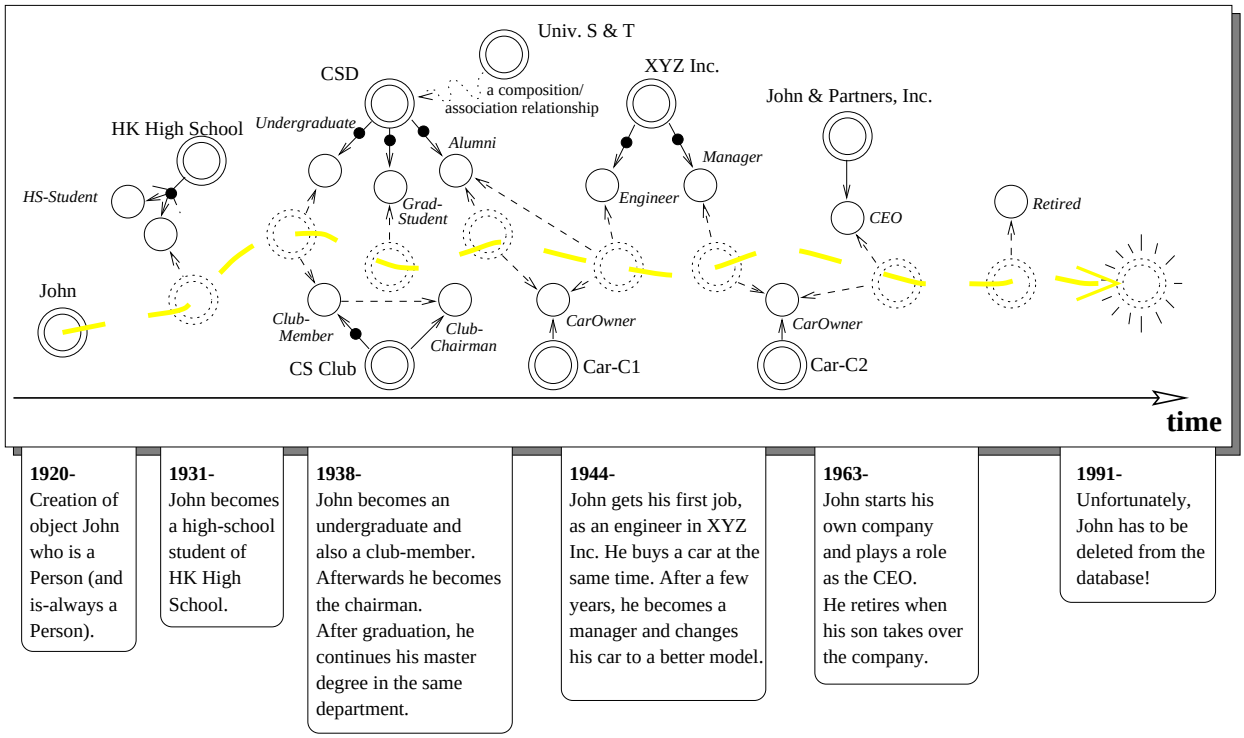

Figure 8: The evolution of object `John` during its lifetime.

the properties of being a *HS-Student* pre-defined by `HK High School`. Afterwards, `John` becomes an *Undergraduate*. It does not hold the properties of *HS-Student* anymore, as its role for `HK High School` is different. After leaving the university, `John` becomes an *Engineer* and then a *Manager* of `XYZ Inc`. For these two jobs, `John` owns the same *'kind'* of properties and maintains the same *'kind'* of relationship with `XYZ Inc`, i.e., being an *Employee* of `XYZ Inc`. This can be indicated by the fact that both *Manager* and *Engineer* are subclasses of *Employee*, and in this case they are owned by the same object. However, there are certainly some differences between being an *Engineer* and being a *Manager*. At the end, `John` retires and plays the role *Retired*, which defines the properties of a retired person. This role can be played by using the constructor **create role** described previously. Note that when `John` is deleted, all the roles being played by him and not owned by some other objects will be deleted automatically.

6. Summary

We have presented an overview of the data model, and outlined the modeling constructs and environment of DOOR, an object-role database system. This paper has presented several novel constructs, based on roles, to support object evolution, dynamic role (context-dependent) modeling, objects of multiple

specific classes, and object-role relationships in object-oriented databases. The most important of them include the player-class constraint, role playership and ownership. The player-class constraint allows any player to play a role type-safely if they satisfy the constraint. We have pointed out the significance of role playership and ownership. A role acts as a bridge (for abstraction and information sharing) between its owner and player while the ownership of the information is clearly defined. Different from other related work, objects can evolve and gain properties prescribed by the owners of roles. Moreover, we have discussed some interesting issues which include playing multiple roles of the same type, player change (or role migration), role ownership and playership, and player-class constraint, etc.

Our ongoing work includes the integration of the concept of composite objects with roles and further investigation of role constraints. Meanwhile, the efficient implementation of roles is under investigation. The first DOOR prototype is implemented using meta-object protocol in a lisp-like language, called Scheme. Most of the runtime efficiency issues are not addressed, except the mechanisms for method lookup and attribute name conflict resolution. We are also still testing DOOR by implementing some non-trivial applications such as multimedia systems [21].

Acknowledgments We thank Prof. Stott Parker at UCLA for his valuable comments and providing a stimulating work environment during this work. We also thanks M. Mira da Silva at University of Glasgow, Chih-Cheng Hsu at UCLA and Eric Lam at Hong Kong University of Science and Technology for useful comments. This research is partially supported by a research grant in Hong Kong RGC96/97.HKUST.757/96E.

References

[1] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In R. Agrawal, S. Baker, and D. Bell, editors, *Proceedings of the 18th International Conference on Very Large Databases*, pages 39–51, Dublin, Ireland, August 1993.

[2] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *VLDB Journal*, 4(3):403–444, 1995.

[3] E. Bertino and G. Guerrini. Objects with multiple most specific classes. In *ECOOP'95 - Object-Oriented Programming*. Springer LNCS952, 1995.

[4] D.H. Fishman et al. Iris: An object-oriented database management system. *ACM Trans. on Office Information Systems*, 5(1):48–69, January 1987.

[5] G. Gottlob, M. Schrefl, and B. Rock. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, July 1996.

[6] G. Kappel *et al.* Workflow management based on objects, rules, and roles. *Bulletin of the Technical Committee on Data Engineering*, 18(1):11–18, March 1995.

[7] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 393–402, 1992.

[8] W Kim, editor. *Modern Database Systems: The Object Model, Interoperability, and Beyond*. Addison-Wesley, 1995.

[9] W. Kim, E. Bertino, and J.F. Garza. Composite objects revisited. *SIGMOD Record*, 18(2):337–47, June 1989.

[10] W Kim and F.H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, 1989.

[11] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In N. Meyrowitz, editor, *Object-Oriented Programming: Systems, Languages and Applications*, pages 214–223, October 1986.

[12] M.P. Papazoglou. Roles: A methodology for representing multifaceted objects. In *Proceedings of the International Conference on Database and Expert Systems Applications*, pages 7–12, 1991.

[13] B. Pernici. Objects with roles. In *IEEE/ACM Conference on Office Information Systems*, Cambridge, Mass., 1990.

[14] J. Richardson and P. Schwartz. Aspects: Extending objects to support multiple, independent roles. In *ACM-SIGMOD International Conference on Management of Data*, pages 298–307, Denver, Colorado, May 1991. ACM SIGMOD Record, Vol. 20.

[15] M. Schrefl and E.J. Neuhold. Object class definition by generalization using upward inheritance. In *Proceedings of IEEE 4th International Conference on Data Engineering*, pages 4–13, 1988.

[16] E. Sciore. Object specialization. *ACM Transactions on Information Systems*, 7(2):103–122, April 1989.

[17] J.J. Shilling and P.F. Sweeney. Three steps to view: Extending the object-oriented paradigm. *OOPSLA '89, ACM SIGPLAN Notices,*, 24(10):353–361, October 1989.

[18] L.A. Stein. Delegation is inheritance. In *OOPSLA '87 Proceedings*, October 1987.

[19] R.K. Wong, H.L. Chau, and F.H. Lochovsky. A data model and semantics of objects with dynamic roles. *Submitted for publication.*

[20] R.K. Wong, H.L. Chau, and F.H. Lochovsky. DOOR: A dynamic object-oriented data model with roles. In *Technology of Object-Oriented Languages and Systems (TOOLS), The 21st International Conference.* Prentice-Hall, November 1996.

[21] R.K. Wong, H.L. Chau, and F.H. Lochovsky. The roles and views of multimedia objects. In *Proceedings of the 1996 International Conference on Multi-Media Modeling.* World Scientific Press, 1996.

[22] R.K. Wong and Q. Li. Manufacturing systems modeling with roles: A comprehensive approach. In *IFIP WG2.6 Sixth Working Conference on Database Semantics (DS-6), Atlanta, Georgia, USA*, May 1995.