A Data Model and Semantics of Objects with Dynamic Roles

Raymond K. Wong[1] H. Lewis Chau[1] Frederick H. Lochovsky
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{wongkk,lewis,fred}@cs.ust.hk

Abstract

Although the concept of roles is becoming a popular research issue in object-oriented databases and has been proven to be useful for dynamic and evolving applications, it has only been described conceptually in most of the previous work. Moreover, the important issues such as the semantics of roles (e.g., message passing) are seldom discussed. Furthermore, none of the previous work has investigated the idea of role player qualification, which models the fact that not every object is qualified to play a particular role. In this paper, we present a data model and the semantics of roles. We discuss each of the above issues and illustrate the ideas with examples. From these examples, we can easily see that the problems we discussed are fundamental and indeed exist in many complex applications.

1. Introduction

In class-based object-oriented systems, real-world entities are represented as instances of the most specific class [3] in which they can be classified. However, when modeling application domains, objects are often found that naturally belong to several most specific classes. Consider as an example, a class person, whose instances can be classified along orthogonal dimensions, such as students, nobles, males and females. According to the intuitive semantics, a person can be both a student, a noble and a female at the same time. Thus, the object representing this person does not have a unique most specific class, rather it has a set of most specific classes.

Although the above 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 lot of artificial subclasses. Thus, this approach can lead to a combinatorial explosion of sparsely populated classes, whose sole purpose is to allow an instance to have multiple most specific classes, without adding new state or behavior. Another problem with the multiple inheritance approach is that it only provides a single behavioral context for an object [12].

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

This paper presents the data model and semantics of a database programming language called DOOR (Dynamic Object-Oriented database programming language with Roles), which includes the important notions commonly associated with object-oriented programming languages and role modeling. In particular, it supports the operations for dynamic role playing and context-dependent behavioral modeling. Since an object can play multiple roles and acquire them at instance level, instantiation of multiple specific role classes are supported such that the association between an instance and role classes are neither exclusive nor permanent. Moreover, we also describe the semantic issues of roles, such as attribute and role access, role acquisition sequence, message passing and hence method lookup, etc., which have been seldom described in the previous work.

The organization of the rest of this paper is as fol-

[1] Part of this work was done when the authors were visiting the Computer Science Department at UCLA.

lows. Section 2 gives an informal overview on role modeling. Because of space limitation, Section 3 presents the core part of the data model for DOOR instead of the complete data model [18]. In particular, it defines the object consistency which is a crucial definition in the DOOR data model. In Section 4, the context-dependent access of an object, based on roles, is discussed. Section 5 describes the basic operations for roles which include role acquisition, role abandon, role migration, and temporary suspension of roles. Section 6 describes the semantics of message passing and hence method lookup for roles. Section 7 discusses the related work, and finally Section 8 concludes the paper.

2 Informal Overview of Role Modeling

In this section, we give an informal overview on role modeling based on the following notation. A *class* is a set of possible individuals, called *class instances*. If the instances are objects, the class is called an *object class*. If the instances are roles, then the class is called a *role class*. Let $inst(C)$ denote the set of all instances of class C. We assume there is a function called *played-by* in the model such that if OC is an object class and RC is a role class, we have

$$\textit{played-by: } inst(RC) \longrightarrow inst(OC) \cup inst(RC)$$

where *played-by(r)* ($r \in inst(RC)$) is called the *player* of r, and *played-by* has the following properties:

1. $r_1 \in inst(RC)$ and $r_2 = \textit{played-by}(r_1) \Longrightarrow r_2 \neq r_1$;
2. *played-by* is neither a surjective function nor injective function.

The codomain of *played-by* includes both the instances of object classes and role classes. Therefore, a role player can be an object, or even a role. However, by property (a) above, we eliminate the case that the player of a role instance is the role instance itself, although it is possible that both the player of a role instance and the role instance itself are of the same role class. For example, a person can be a *club-member* of a credit card club, and being a general club-member, he then further join as a *club-member* of the privilege club of the credit card. The second property provides more information about the role playing characteristics. It implies that an object (or a role) can play multiple roles (i.e., *played-by* is not injective), and also, an object (or a role) may not be a player of any roles at all (i.e., *played-by* is not surjective).

Furthermore, we define the automatic *delegation* from roles to players. For example, suppose we model an employee e as a role of a person p, and *sex* is an attribute of persons but not of employees. Then $sex(e)$ would be a type error. We can correct this error by delegating the evaluation of *sex* to *played-by*(e) [9]. This amounts to replacing *sex(e)* by *sex(played-by(e))*. Moreover, roles also provide data protection by partitioning the messages received by players. For example, suppose we model an employee e and a student s as two roles of a person p, and *studentid* is an attribute of students but not of persons or employees. Then *studentid(e)* would be a type error. Unless we know that person p is a student and access his/her information from the perspective of accessing student information (by *studentid(s)*), *studentid(p)* would also be a type error.

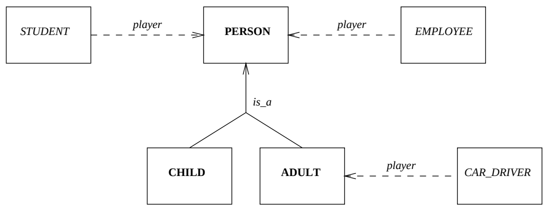

Figure 1. PERSON can at any moment be an EMPLOYEE and/or a STUDENT, but only an ADULT can at any moment be a CAR_DRIVER.

By introducing the role classs hierarchy into the object class hierarchy, our role model is formed. These two type of classes are orthogonal to each other, and each of them can be partitioned into subclasses. The difference between role classes and object classes lies in the fact that an instance of an object subclass is identical to (i.e., has the same identifier as) an instance of its superclass but an instance of a role class is different from any instance of its player class. Similar to the other properties of a class, the *player* relationships of a class will be inherited to all its subclasses. For example, consider a PERSON who can be an EMPLOYEE and a STUDENT as shown in Figure 1 [2]. The *player* relationship will be inherited to all the subclasses of PERSON, i.e., CHILD and ADULT. However, only an ADULT can be a CAR_DRIVER. Therefore, a CHILD cannot play the role CAR_DRIVER.

3 The Core DOOR Model

This section presents the definition of the core DOOR model (the detailed model is described in [18]).

[2] Class names in bold denote object classes, and class names in italic denote role classes.

3.1 Basics

Before we define the notion of objects formally, we assume we are given:

- A finite set $\mathbf{D}$ of pairwise-disjoint *domains* $D_1, \ldots, D_n$, $n \geq 1$. Currently, we have the following domains corresponding to the DOOR basic types: **integer**, **real**, **string**, **bool**.
- A countably infinite set **OID** of symbols called *object identifiers*, a finite set **O** of objects, a finite set **OC** of object classes, and a finite set **RC** of role classes.
- A countably infinite set **A** of symbols called *attributes*, and a finite set **M** whose elements are called methods and play the role of operations on our data structures. Informally, we can think of the elements of **M** as uninterpreted symbols (i.e., uninterpreted lambda expressions).

Besides, we assume the following set of type names:

- **Bnames** is the set of names for basic types, **OCnames** is the set of names for object types (object classes), **RCnames** is the set of names for role types (role classes).
- **Snames** = **OCnames** $\cup$ **RCnames** is the set of names for structured types that is countably infinite and disjoint with **Bnames**.
- **Tnames** = **Bnames** $\cup$ **Snames** is the set of all names for types.

3.2 Values

Let us now define the notion of *value*. There are three types of values in the DOOR data model described in this paper: *basic-value*, *set-value*, and *tuple-value*.

Definition 1

1. *Every element of* $\mathbf{D} \cup \{\perp_v\}$ *is called a* basic-value. *The special symbol* $\perp_v$ *represents the undefined (i.e., null) value.*
2. *Every finite subset of* **OID** *is called a* set-value. *Set-values are denoted in the usual way using brackets.*
3. *Every finite partial function from* **A** *into* **OID** *is a value, called a* tuple-value. *We denote by* $[a_1 : i_1, \ldots, a_p : i_p]$ *the partial function* t *defined on* $\{a_1, \ldots, a_p\}$ *such that* a_k *are distinct and* $t(a_k) = i_k$ *for* $1 \leq k \leq p$.

We denote by **V** *the set of all values.* □

3.3 Roles

In the context of OODBs, roles can be used to model the phenomenon of class migration whereby a person can become a student, i.e., the object gets student properties in addition to its person properties. In some implementations, this involves changing the identifier of the person object into that of a student object [8]. But changing identifiers creates practical implementation problems such as how to avoid dangling references and how to represent historical information. In other implementations (such as [17]), roles are identified by globally unique and unchangeable identifiers. This role representation scheme offers the following advantages: it distinguishes a role from other roles; it recognizes a role as the same role even if its state is changed; it models class migration by adding and deleting roles to an object (i.e., no need to change any identifiers).

However, this unique role identifier scheme still causes problems like dangling references and historial information representation if we allow a reference to a role. For example, if a manager of a company has been changed to another person, all references to the manager should change to the new manager (who is represented by a different role identifier). For this fundamental reason, we identify roles by the names of their role classes as well as their values, instead of using globally unique identifiers. The treatment for this will be further discussed in Section 4.2.

Definition 2 *A* role *is a quadruple* $r = (rcname, val, Rs, m)$ *where* $rcname \in$ **RCnames** *is the role class name of which* r *is an instance,* val *is a value (i.e.,* $val \in \mathbf{V}$*),* Rs *is the set of roles being played by* r *(i.e.,* $\forall\, rs \in Rs$, played-by$(rs) = r$*), and* $m \subseteq \mathbf{M}$ *is a set of methods. We denote by* **R** *the set of all roles; that is,* $\mathbf{R} = \mathbf{RCnames} \times \mathbf{V} \times \mathbf{2}^{\mathbf{R}} \times \mathbf{2}^{\mathbf{M}}$. □

Example: For example,

```
(student, [studentid: 9501231, ...],
  {(club-member, [...], {}, {})}, {})
```

represents a role instance which is of role class `student`, with attribute `studentid`, etc., and is playing a `club-member` role. For the sake of simplicity at this stage, we assume that there is no method defined for both `student` and `club-member` roles. This example can be used to show the relationship, between a role and its player, that all roles are destroyed if their players are destroyed, e.g., one is no longer a `club-member` if he/she is no longer a `student`.

3.4 Objects

We can now define the notion of an object.

Definition 3

1. *An* object *is a quintuple* $o = (oid, ocname, val, Rs, m)$*, where* $oid \in \mathbf{OID}$ *(an identifier),* $ocname \in \mathbf{OCnames}$ *is the object class name of which* o *is an instance,* val *is a value (i.e.,* $val \in \mathbf{V}$*),* Rs *is the set of roles played by* o *(i.e.,* $\forall\, rs \in Rs$*,* $\text{played-by}(rs) = o$*), and* $m \subseteq \mathbf{M}$ *is a set of methods.*

2. *We define the notion of* basic *objects,* set-structured *objects, and* tuple-structured *objects for objects with val being* basic-value, set-value, *and* tuple-value *respectively.*

3. $\mathbf{O}$ *is the set of all objects; that is,* $\mathbf{O} = \mathbf{OID} \times \mathbf{OCnames} \times \mathbf{V} \times \mathbf{2^R} \times \mathbf{2^M}$.

□

We also add some further notation about objects and roles to support the later definitions:

- Given an object $o = (i, n, v, r, m)$, $ident(o)$ denotes the identifier i, $class(o)$ denotes the object class name n, $value(o)$ denotes the value v, $roles(o)$ denotes the set of roles r played by o, and $allroles(o)$ denotes the set of roles played by o as well as any of its roles.

- Similarly, we extend some of these functions for roles. Given a role $r = (n, v, rs, m)$, $class(r)$ denotes the role class name n, $value(r)$ denotes the value v, $roles(r)$ denotes the set of roles rs played by r, and $allroles(r)$ denotes the set of roles played by r as well as any of its roles.

- We will denote by *ref* the function from $\mathbf{O}$ in $2^{\mathbf{OID}}$ which associates to an object the set of all the identifiers appearing in its value, i.e., those referenced by the object.

The following example illustrates the object and role representation in DOOR.

Example: Consider a student Peter who plays a role as a student-worker:

```
object1 = (oid1011, Person, [name: "Peter",
   sex: "male", birthday: 050871],
   { (student, [studentid: 9501253, dept: oid13],
     { (student-worker, [salary: 9500, ...], {},
     {method_a1...}) }, {}) }, {method_b1...} )
```

In this example, `Peter` (an object, with oid `oid1011`, of class `Person`) plays a role as `student`. Being a `student`, `Peter` is also a `student-worker`. Moreover,

$ident$(`object1`) = `oid1011`,
$class$(`object1`) = `Person`,
$roles$(`object1`) = {(`student, ...`)},
$allroles$(`object1`) = {(`student, ...`),
(`student-worker, ...`)}, and
$value$(`object1`) = [`name: "Peter", ...`].

3.5 Types

A type is an abstraction that allows the user to encapsulate in the same structure both data and operations. Users of a type only see its abstract part – that is, the interface of its methods – whereas the programmer of the type is concerned with the implementation. However, a type has only one implementation. In the next definition, we define object types and role types (object classes and role classes). Our definition of object types is similar to that in class-based OO languages that do not support roles. However, we introduce the notion of *player qualification* to role type definition to model the fact that not every object can play a particular role. Some trivial examples are: only a male can play a role such as a father, only a student can be a student-member, etc.

Definition 4 *(*Object Types and Role Types*) A* type *is either an object type or a role type. The set of all types, object types, and role types are denoted by* $\mathbf{T}$*,* $\mathbf{OT}$*, and* $\mathbf{RT}$*, respectively.*

An object type *is either a basic or a structured object type. A basic object type, or BOtype, is a pair (n, m) where* $n \in \mathbf{Bnames}$ *and* $m \subseteq \mathbf{M}$*. A structured object type, or SOtype, is one of the following:*

- *A triple* (s, t, m) *where* $s \in \mathbf{Snames}$*,* $t \in \mathbf{Tnames}$*, and* $m \subseteq \mathbf{M}$*. We denote such a type by* $s = (t, m)$*.*

- *A triple* (s, t, m) *where* $s \in \mathbf{Snames}$*, t is a finite partial function from* $\mathbf{A}$ *to* $\mathbf{Tnames}$*, and* $m \subseteq \mathbf{M}$*. We denote such a type by* $s = ([a_1 : s_1, \ldots, a_n : s_n], m)$*, where* $t(a_k) = s_k, 1 \leq k \leq n$*. We call it a tuple-structured object type.*

- *A triple* $(s, \{t\}, m)$ *where* $s \in \mathbf{Snames}$*,* $t \in \mathbf{Tnames}$*, and* $m \subseteq \mathbf{M}$*. We denote such a type by* $s = (\{t\}, m)$*. We call* s *a set-structured object type.*

Using the similar notation, a basic role type and a structured role type are defined as (n, m, oc) *and*

(s, t, m, oc) *respectively. oc is a subset of* **OCnames**, *which is called the* player qualification *and specifies the qualified players. To support later definitions, we also define an auxiliary function players, which returns the player qualification of a role type (i.e.,* $players(rt \in \mathbf{RT}) = oc$*).* □

Player qualification in DOOR models the fact that not every object qualifies to play a particular role. Moreover, it is costly, if not infeasible, to forecast all cases in which a subset of roles can be played by a particular class of objects (role qualification). Therefore, we choose to have player qualification as part of role class definition instead of having a role qualification in an object class. That is, to qualify as a player of a particular role, a player must be an element of the extension of a class which is included in the player qualification of the role classes. Furthermore, with player qualification instead of role qualification, an object class definition is similar to a class definition in traditional class-based object-oriented languages that do not support roles. Hence users can easily use DOOR as a traditional object-oriented language for simple applications which do not require roles.

Example: Assume the class hierarchies shown in Figure ??, i.e., `Child` $\le$ `Person`, `Adult` $\le$ `Person`, `Student-Worker` $\le$ `Student`, `Student-Worker` $\le$ `Employee`, we define the object types: `Person`, `Child`, `Adult`, and role types: `Student`, `Employee`, `Student-Worker` as follows:

```
(Person, [name: String, sex: String], _)
(Child, [name: String, sex: String], _)
(Adult, [name: String, sex: String, spouse:
  Adult], _)
(Student, [school: School, studentid: Integer],
  _, {Person})
(Employee, [employer: Company, position: String,
  salary: Integer], _, {Adult})
(Student-Worker, [school: School, studentid: Integer,
  employer: Company, position: String, salary:
  Integer], _, {Adult})
```

Suppose that `Peter` is of object class `Child`, and `Mary` is of object class `Adult`. `Peter` cannot play role(s) which is/are of role class `Employee` nor `Student-Worker`, otherwise, he is not in a consistent set of objects (i.e., violation of the player consistency condition). `Mary`, who is playing a role `Employee`, can be expressed by

```
(oid1022, Person, [name: "Mary", sex: "female",
  spouse: oid1031], {(Employee, [employer:
  oid1113, position: "typist", salary: 8000],
  {}, _)}, _)
```

3.6 Object Consistency

After the definitions of objects, roles, and types are defined, we can eventually have the consistency defined for all objects in DOOR as follows:

Definition 5 *(*Consistent Set of Objects*) A finite set* Θ *of objects is consistent iff all the following conditions hold:*

1. OID Uniqueness: *The* ident *function is injective on* Θ *(i.e., there is no pair of objects with the same identifier).*
2. Referential Integrity: *For all* $o \in \Theta$, $\mathrm{ref}(o) \subseteq \mathrm{ident}(\Theta)$ *(i.e., every referenced identifier corresponds to an object of* Θ*).*
3. Player Consistency: *For all* $o \in \Theta$ *and for all* $r \in allroles(o)$, $class(p) \le^* p'$ *and* $p' \in players(class(r))$ *where* $p = played\text{-}by(r)$ *and* $\le^*$ *denotes the transitive subclass relationship. This condition ensures that all roles are played by qualified players, according to the player qualification specified in their role types.*

□

4 The Context-Dependent Access

This section describes the different ways of accessing a particular role of an object. The access mechanism supports the notion of context-dependent access restriction and hence the context-dependent behavior of an object.

4.1 Basic Object, Role and Attribute Access

The basic notation to access a role r of object o is specified using the exclaimation mark '!', i.e., *object*!*role*. For example, `john!ClubMember` means accessing the `ClubMember` role of object `john` in Figure 2 Top. In other words, we consider `john` from his `ClubMember` perspective. Another example is when we access the attribute `id#` (identity number) of `john` from different perspectives. As shown in Figure 2, the different `id#`s of `john` mean different things depending on which context/perspective we consider. For example `john.id#` means its personal identity number assigned by government and `john!GradStudent.id#` means its student identity number given by university. Moreover, it is obvious that `john!GradStudent.id#` $\neq$ `john!TA.id#` $\neq$ `john!ClubMember.id#`.

However, is `john!TA.id#` $\neq$ `john!RA.id#` too? To resolve the attribute *name conflicts* such as this, we

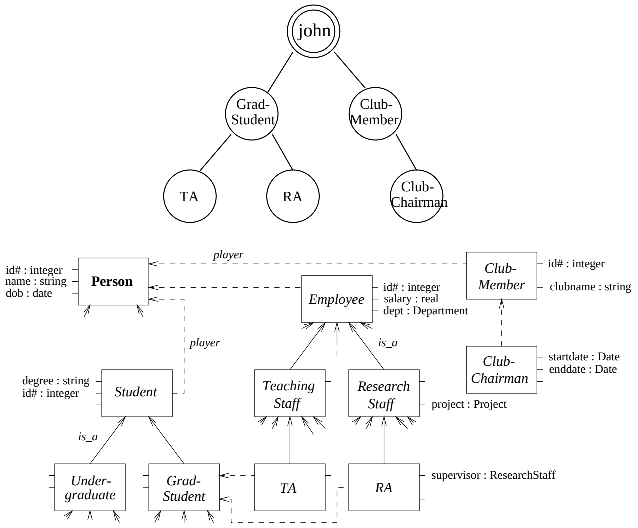

Figure 2. *Top*: **a person object** `john` **who is playing multiple roles:** `GradStudent` **(graduate student),** `TA` **(teaching assistant),** `RA` **(research assistant),** `ClubMember` **and** `ClubChairman`. *Bottom*: **its relevant object class and role classes.**

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, belonging 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 *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 by 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 in seperate ways. With roles this different information can be then accessed according to different contexts, while in [3], a renaming policy is defined to resolve the name conflicts.

In [3], this resolution scheme is default for all cases. However, this cannot tackle many situations that the attributes need to store different values even they come from the same source. Therefore, in DOOR, this resolution is used only for those attributes defined with a keyword `UNIQUE`. For all other attributes, the object may have two different values for the two attributes with the same name in different roles. This is necessary for many cases such as the attribute `dept` defined in the role class `Employee` in Figure 2. 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 real situation. That is, John may work as a teaching assistant in Computer Science Department and also as a research assistant in Electrical Engineering Department.

As a result, in Figure 2, 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 they are the same person.

4.2 Access with Multiple Roles of the Same Type

Users can of course specify the exact path for a role, 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 (as mentioned in Section 3.3), 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="UST").id#`.

Although users can specify a boolean criteria to control the result returned by the selection mechanism, non-determinism still exists because of multiple solutions. In case of multiple roles satisfying the expression, the system will only return one of them for the sake of simplicity and efficiency.

5 Basic Role Operations

There are a few basic operations for roles. They are the operations for role acquisition, role abandon, role migration, and temporary suspension of roles.

Before we discuss further about these key operations for roles, let us define some basic terminology. A role r is said to be played *indirectly* by a player x (or x is said to be an indirect player of r) if there exists some roles $r_0, \cdots, r_n$, where $n \leq 0$, such that $x!r_0!\cdots!r_n!r$ will not raise a *role_not_found* exception. Otherwise, r is said to be played *directly* by x (or x is said to be the direct player of r). In either case, x is said to be the direct or indirect root/object player of r if x is a single-faceted object[3], e.g., `john` is the indirect root player of `TA` in Figure 2. Note that here x can be an object or a particular perspective of an object (i.e., a role of an object). For example, in Figure 2, `john` is indirectly playing the role `TA` and `john!GradStudent` is directly playing the role `TA`, as `john!GradStudent!TA.id#` will not raise a *role_not_found* exception while `john!TA.id#` will.

5.1 Role Acquisition and the Acquisition Sequence

The function *role_play(x, r)* is used for an object x to acquire a role r for playing. If r is going to be played by another role r' being played by x, x is of the form $o! \cdots !r'$. For example, `role_play(john, GradStudent)` specifies that `john` acquires a `GradStudent` role, and then `role_play(john!GradStudent, TA)` specifies that being a `GradStudent`, `john` further plays a role as a `TA`.

As the role acquisition sequence is constrained by the player qualification specified in the class schema, such as Figure 2 Bottom, the possible sequence of role acquisition is restricted. For example, if `john` would like to play the roles as shown in Figure 2 Top starting from being a single-faceted object, all the possible sequences for acquiring these roles, as shown in Figure 3, can be generated according to the player qualification specified in the class schema. The arrows in Figure 3 denote the precedence relation. For example, if `john` would like to play only two roles – `ClubMember` and `GradStudent`, then the different paths from node 1 to node 6 represent the different possible sequences. In this example, `john` can acquire `GradStudent` before `ClubMember`, `ClubMember` before `GradStudent`, or acquire them at the same time, as there is no precedence relation between node 2 and node 3.

Although the player qualification can be generalized by including other constraints besides a list of qualified classes, the resultant schema is easier to be analyzed. For example, the all possible sequences for role aquisition, like the one shown in Figure 3, cannot be generated in static and automatically if we introduce the general pre-conditions/constraints (such as the events and conditions in the ECA rules for roles in [6]) to the player qualification, unless we consider the behavioral aspect of the objects by evaluting each expression of these conditions/constraints.

[3] A *single-faceted object* is defined to be an object which is not playing any roles.

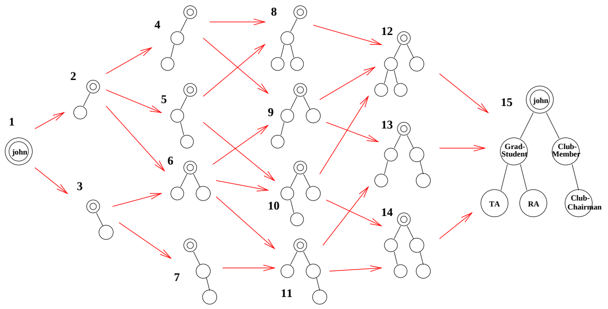

Figure 3. Possible role playing sequence for `john` (the arrows denote the precedence relation).

5.2 Role Abandon and the Abandon Scope

The operation *role_drop*$(x \cdots !r)$ is used to drop a role r from object x. If there are other roles being played by r no matter whether directly or indirectly, they will be dropped as well to keep the player consistency (*cf.* Definition 5). For example, assume `john` is playing the roles `GradStudent`, `TA`, and so on, as shown in Figure 2. The evaluation of expression `role_drop(john!GradStudent!TA)` will drop the role `TA` from `john`, while The expression `role_drop(john!GradStudent)` will result in dropping the roles `GradStudent`, `TA` and `RA` from `john`. This is reasonable and meaningful that `john` is no longer qualified to play the roles as `TA` and `RA` once he is not a `GradStudent`.

5.3 Role Migration and the Migration Scope

The function *role_move*$(x \cdots !r, y)$ is used to migrate a role r from object x to object y. If r is going to be played by another role r' being played by y, then y should be of the form $o \cdots !r'$. If there are other roles being played by r no matter whether directly or indirectly, they will be migrated to y as well to maintain the player consistency.

For example, assume `john` is playing the roles `GradStudent`, `TA`, and so on, as shown in Figure 2, and assume there is a single-faceted object `peter` which is of class `Person`. `john` can transfer the `ClubMember` and `ClubChairman` roles to `peter` by the operation `role_move(john!ClubMember, peter)`. Perhaps a more realistic situation will be that `peter` becomes a `ClubMember` first, and then `john` passes the chairmanship to him:

```
role_play(peter, ClubMember)
role_move(john!ClubMember!ClubChairman, peter)
```

The migration operation is indeed useful in many common cases that we need to transfer a role from one object to another. A typical example will be the transfer of a manager role from one person to another, or the transfer of a chairman role from one to another. Then all the properties (both structural and behavioral) associated with the manager role will be transferred to the new player automatically.

5.4 Temporary Suspension of Roles

Sometimes we may want to suspend a role temporarily and later resume it, as suggested by Pernici [11]. For example, a student may suspend his/her study for a few semesters and later resume it. In DOOR, three operations, *role_suspend*, *role_resume* and *role_suspended*, are provided for the temporary suspension of roles.

The function *role_suspend*($x \cdots !r$) is used to suspend a role r being played by object x. If there are other roles being played by r no matter whether directly or indirectly, they will be suspended as well. When a role is suspended, its behavior (both the states and methods) will be suspended, like it is not being played by its player. The function *role_resume*($x \cdots !r$) is used to resume a suspended role r being played by object x. If there are other suspended roles being played by r no matter whether directly or indirectly, they will be resumed as well. As usual, there will be no effect to suspend a suspended role and resume a non-suspended role. Finally, the function *role_suspended*(x) lists all the suspended roles being played (both directly and indirectly) by x. As before x can be a single-faceted object or of the form $o \cdots !r$.

One of the most interesting way to use these functions is to suspend a role r, and then resume a role r' which is indirectly played by r, and finally suspend a role which is indirectly played by r'. In this way, any set of roles being played by an object can be suspended.

6 Method Lookup as Delegation

This section describes the semantics of method dispatching among the roles and their players. The semantics is based on the idea of *delegation*, which is used to pass the messages between the prototype objects in classless object-oriented languages. We use the notation $\Leftarrow$ to represent the message send. For example, $o \Leftarrow m$ denotes the sending of message m to object o and $o!r \Leftarrow m$ denotes the sending of message m to the role r of object o. For simplicity, we eliminate the unrelated constructs and model an object as a record of methods together with a set of roles being played. Similarly, we model a role as a record of methods together with a role class identifier and a set of roles being played. The only difference between objects and roles here is the existential constraint that a role can only exist by being played by its player. For example, the representation of object o_1 is a record of methods together with a set of roles being played, and each role is another record of methods together with a role identifier together with a set of roles being played, as shown below:

$$
\begin{array}{lcl}
o_1 & = & (m_1 = e_1, \cdots, m_i = e_i, \{r_1 = d_1, \cdots, r_j = d_j\}) \\
r_1 & = & (m'_1 = e'_1, \cdots, m'_{i'} = e'_{i'}, \{r'_1 = d'_1, \cdots, r'_{j'} = d'_{j'}\})_{r_1} \\
r'_1 & = & (m''_1 = e''_1, \cdots, m''_{i''} = e''_{i''}, \{r''_1 = d''_1, \cdots, r''_{j''} = d''_{j''}\})_{r'_1} \\
\cdots & & \cdots \quad \cdots
\end{array}
$$

Suppose we send a message to a role r_n of an object o, i.e., $o!r_n \Leftarrow m$. The expression will then be evaluated to a form $(o, r_1, r_2, \cdots, r_{n-1}, r_n) \Leftarrow m$, where $(o, r_1, \cdots, r_n)$ is the list of roles and the object in which the method m is looked for. The order and the content of the list depend on the method lookup scheme (or called the delegation path) being used. In the current implementation of DOOR, seven method lookup schemes are supported, which are listed in Figure 4. These different schemes allow users to have the different choices for the tradeoff between encapsulation and information sharing, for the different requirements of different complex applications. For example, the most primitive scheme (with the best encapsulation but poorest data/resource sharing) is the one with no delegation among roles, as shown in Figure 4(g). The different method lookup schemes are implemented as meta-object protocols [7] so that users can easily change to different method lookup schemes for different specific applications. Under the most primitive scheme

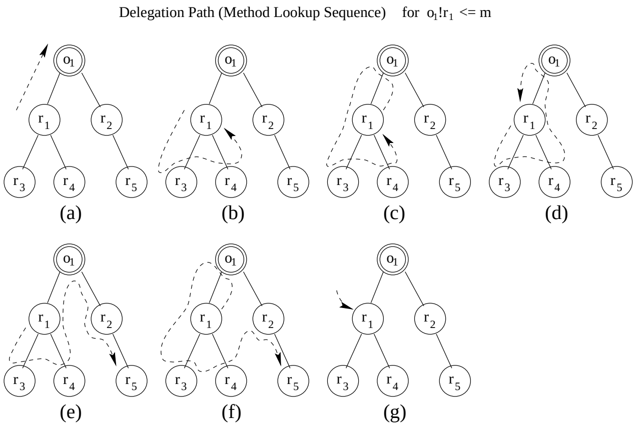

Figure 4. Different method lookup schemes for $o_1!r_1 \Leftarrow m$ in DOOR: (a) $r_1;\ o_1$. **(b)** $r_1;\ r_3;\ r_4$. **(c)** $r_1;\ o_1;\ r_3;\ r_4$. **(d)** $r_1;\ r_3;\ r_4;\ o_1$. **(e)** $r_1;\ r_3;\ r_4;\ o_1;\ r_2;\ r_5$. **(f)** $r_1;\ o_1;\ r_3;\ r_4;\ r_2;\ r_5$. **(g)** r_1.

(g), the semantics of sending a message m to object o_1 is simply as follows:

$$
\begin{array}{lll}
o_1 \Leftarrow_{(g)} m_\alpha & & \\
\leadsto & (m_1 = e_1, \cdots, m_\alpha = e_\alpha, \cdots, m_i = e_i, & \\
& \{r_1 = d_1, \cdots, r_j = d_j\}) \Leftarrow_{(g)} m_\alpha & \\
\leadsto & e_\alpha\,(m_1 = e_1, \cdots, m_\alpha = e_\alpha, \cdots, m_i = e_i, & \\
& \{r_1 = d_1, \cdots, r_j = d_j\}) &
\end{array}
$$

where the relation $\leadsto$ represents one evaluation step, not full evaluation of an expression, and we also denote the send operation under the scheme x of Figure 4 as $\Leftarrow_x$. In the above evaluation, if no such m_α is found in the record of methods, an exception *method_not_found* is raised, e.g.,

$$
\begin{array}{ll}
(m_1 = e_1, m_2 = e_2, \{\cdots\}) \Leftarrow_{(g)} m_\alpha & \\
\leadsto \quad \mathit{method\_not\_found} &
\end{array}
$$

Similarly, the expression $o_2!r_n \Leftarrow_{(g)} m_\beta$ can be evaluated by firstly finding out the corresponding r_n of object o according to the role access mechanism described in Section 4. Once the role r_n is found, the rest is similar as above, i.e.,

$$
\begin{array}{ll}
(m_1 = e_1, \cdots, m_\beta = e_\beta, \cdots, m_i = e_i, & \\
\quad \{r_1 = d_1, \cdots, r_j = d_j\})_{r_n} \Leftarrow_{(g)} m_\beta & \\
\leadsto \quad e_\beta\, o_2 &
\end{array}
$$

If no such m_β exists, *method_not_found* is raised, e.g.,

$$
\begin{array}{ll}
(m_1 = e_1, m_2 = e_2, m_3 = e_3, \{\cdots\})_{r_n} \Leftarrow_{(g)} m_\beta & \\
\leadsto \quad \mathit{method\_not\_found} &
\end{array}
$$

The default method lookup scheme used in DOOR is shown in Figure 4(a). Under the default scheme, the semantics of $o!r_n \Leftarrow_{(a)} m$ can be described as: DOOR tries r_n first; if r_n does not understand the message m, it will try its direct player r_{n-1}; if r_{n-1} does not understand, it will try its direct player r_{n-2} and so on, until either the root (object) is tried or someone along the listed understands m.

We can in fact describe method lookup semantics precisely with the following algorithm, by relabeling o as r_0 in the above list of delegation path, i.e., $(r_0, r_1, r_2 \ldots, r_{n-1}, r_n) \Leftarrow m$.

$i = n$
while $(r_i \Leftarrow m) \not\leadsto \mathit{method\_not\_found}$ and $(i > 0)$ do
    $i = i - 1$
endwhile
return $(r_i \Leftarrow m)$

where $e \not\leadsto \mathit{method\_not\_found}$ denotes that the evaluation of e does not raise a *method_not_found* exception.

7 Related Work

The concept of a role was already defined in 1977 by Bachman and Daya [2] in the context of the network data modeling approach. Afterwards, the restriction that an object be associated with a single, most-specific type in the database context was relaxed in Iris [4]. Iris allows an object to belong to several types, but it misses the possibility of role-specific / content-dependent behavior. Recently, the concept surfaced again and its significance was realised in work on office modeling [11], semantic modeling [12, 14], object-oriented modeling [10], and multimedia applications [19], etc. All of these efforts have utilized roles for building various dynamic functions (such as for object migration, multiple perceptions of objects, conceptual clustering, etc.).

The importance and support of multiple perspective/context-dependent behavior of objects were firstly described in *Clovers* [16], *multiple views* [15], *ORM* [11], and *Aspects* [12]. 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 [14] 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 [13], 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 programming language called Fibonacci [1] 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 multiple views, however, the roles attached to a class in both approaches can form their own *is-a* hierarchy.

In summary, most of these works described the concept of context-dependent access restriction and context-dependent behavior which are stressed in object data models supporting roles. In these approaches, it is not always enforced that (one of) the most specific behavior(s) of the object is exhibited. Moreover, among these works, only Fibonacci deals with the problem of method lookup for objects with multiple roles. However, none of the above work has described the idea of player qualification and hence the role acquisition and dropping orders.

8 Conclusions

This paper has presented a data model and its semantics for objects with dynamic role playing, which solve the problems in traditional class-based object-oriented systems caused by a permanent and exclusive link between an instance and a class. Role classes are defined to complement object classes. An object of an object class can play zero or more roles, and a role of a role class can only exist by being associated with an object. Moreover, a role can also play roles. The *played-by* relationship is defined between a role player and the roles being played by means of delegation. Through delegation, a unique object can be accessed through different roles (i.e., different perspectives), which have different types and can answer in different ways to a message. Some important issues, such as player consistency, attribute and role access, basic operations for roles which include acquisition sequence and abandon scope, message passing and hence method lookup, have been discussed. Furthermore, player qualification models the fact that not every object qualifies to play a particular role. That is, to qualify as a player of a particular role, a player must be an element of the extension of one of the classes specified in the player qualification of the role classes.

Our onging work on role modeling includes the investigation of multiple players for a role, generalizing the player qualification to allow general pre-conditions (and probably post-conditions) for role playing. For efficiency issues, we are currently working on the indexing scheme for objects with dynamic roles.

References

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

[2] C. Bachman and M. Daya. The role concept in data models. In *Proceedings of the Third International Conference on Very Large Databases*, pages 464–476, 1977.

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

[4] D. Fishman et al. Iris: An object-oriented database management system. *ACM Trans. on Office Information Systems*, 5(1):48–69, Jan. 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, Mar. 1995.

[7] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[8] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[9] 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, Oct. 1986.

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

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

[12] 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.

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

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

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

[16] L. Stein and S. Zdonik. Clovers: The dynamic behavior of type and instances. Technical Report CS-89-42, Brown University, Nov. 1989.

[17] R. Wieringa and W. d. 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.

[18] R. Wong, H. Chau, and F. Lochovsky. DOOR: A dynamic object-oriented data model with roles. Technical Report HKUST-CS96-12, Hong Kong University of Science & Technology, 1996.

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