The Role Players and Owners in DOOR

Raymond K. Wong†, H. Lewis Chau*, and Frederick H. Lochovsky*

Technical Report HKUST-CS97-03
May 1997

A Department of Computer Science
The Hong Kong University of Science & Technology
Clear Water Bay, Kowloon, Hong Kong

† School of MPCE Computing
Macquarie University
Sydney, NSW 2109, Australia
rwong@mpce.mq.edu.au

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. 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. 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. We show by examples that all these features are very useful for applications in which objects take on different and multiple roles over time.
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. 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 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.

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.

Recently, some researchers have proposed extending object models to incorporate the concept of roles to tackle these problems in various application domains. These include

---

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 $o$. 
office modeling [11], semantic modeling [12, 14], object-oriented modeling [10], manu-
facturing system modeling [21], and multimedia applications [19]. 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 [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 Schreifl and Neuhild in [13],
even though 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 Overview

We briefly review the object-role data model formally defined in [20, 18].

2.2 Basics

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

- A finite set $D$ 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.

2.3 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 $D \cup \{\bot_v\}$ is called a basic-value. The special symbol $\bot_v$ 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. □

2.4 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 [6]. 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 historical 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.

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

**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.
2.5 Objects

We can now define the notion of an object.

**Definition 3**

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

2. We define the notion of basic objects, set-structured objects, and tuple-structured objects for objects with $\text{val}$ being basic-value, set-value, and tuple-value respectively.

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

**Definition 4** A function $\text{played-by}: \text{RC} \rightarrow \text{OC} \cup \text{RC}$ is defined where $\text{RC}$ is role type and $\text{OC}$ is object type. $\text{played-by}(r)$ ($r \in \text{RC}$) is called the player of $r$, and $\text{played-by}$ has the following properties:

1. $r_1 \in \text{RC}$ and $r_2 = \text{played-by}(r_1) \Rightarrow r_2 \neq r_1$;

2. $\text{played-by}$ is neither a surjective function nor injective function.

The codomain of $\text{played-by}$ includes both the instances of object classes and role classes. Therefore, a role player can be an object, or even a role. However, by the first property 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., $\text{played-by}$ is not injective), and also, an object (or a role) may not be a player of any roles at all (i.e., $\text{played-by}$ 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 $\text{sex}$ is an attribute of persons but not of employees. Then $\text{sex}(e)$ would be a type error. We can correct this error by delegating the evaluation of $\text{sex}$ to $\text{played-by}(e)$ [9]. This amounts to replacing $\text{sex}(e)$ by $\text{sex}(\text{played-by}(e))$. Moreover, roles also provide data protection by partitioning the messages received by players.

We also add some further notation about objects and roles to support the later definitions:
Given an object \( o = (i, n, v, r, m) \), \( \text{ident}(o) \) denotes the identifier \( i \), \( \text{class}(o) \) denotes the object class name \( n \), \( \text{value}(o) \) denotes the value \( v \), \( \text{roles}(o) \) denotes the set of roles \( r \) played by \( o \), and \( \text{allroles}(o) \) 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 \( r = (n, v, rs, m) \), \( \text{class}(r) \) denotes the role class name \( n \), \( \text{value}(r) \) denotes the value \( v \), \( \text{roles}(r) \) denotes the set of roles \( rs \) played by \( r \), and \( \text{allroles}(r) \) denotes the set of roles played by \( r \) as well as any of its roles.

- We will denote by \( \text{ref} \) the function from \( O \) in \( 2^{\text{OID}} \) 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:

```plaintext
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,

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

### 2.6 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 5** (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 $T$, $OT$, and $RT$, 
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 $n \in Bnames$ and $m \subseteq M$. A structured object type, or 
$SOType$, is one of the following:

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

- A triple $(s, t, m)$ where $s \in Snames$, $t$ is a finite partial function from $A$ to $Tnames$, 
and $m \subseteq M$. 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 $s \in Snames$, $t \in Tnames$, and $m \subseteq M$. 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 $player$ which returns the player qualification of a role type 
(i.e., $players(rt \in RT) = oc$).

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 hierarchy: Child $\leq$ Person, Adult $\leq$ Person, 
Student-Worker $\leq$ Student, Student-Worker $\leq$ Employee, we define the object types: 
Person, Child, Adult, and role types: Student, Employee, Student-Worker as follows:
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

(o1d1022, Person, [name: "Mary", sex: "female", spouse: o1d1031], {Employee, [employer: o1d1031, position: "typist", salary: 8000], {}, {}}, {})

2.7 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 6 (Consistent Set of Objects) A finite set \( \Theta \) of objects is consistent iff all the following conditions hold:

1. **OID Uniqueness**: The ident function is injective on \( \Theta \) (i.e., there is no pair of objects with the same identifier).

2. **Referential Integrity**: For all \( o \in \Theta \), \( \text{ref}(o) \subseteq \text{ident}(\Theta) \) (i.e., every referenced identifier corresponds to an object of \( \Theta \)).

3. **Player Consistency**: For all \( o \in \Theta \) and for all \( r \in \text{allroles}(o) \), \( \text{class}(p) \leq^t p' \) and \( p' \in \text{players}(\text{class}(r)) \) where \( p = \text{played-by}(r) \) and \( \leq^t \) 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.

\( \square \)
2.8 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.

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 overridden. 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
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.

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 play-ership 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’ [11] 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 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 ad-hoc 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 [7]) 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 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# ≠ 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 [16] 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 privilege. 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 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 × Object-Class → 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 × Object-Class ∪ Role-Class → 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 × Role-Class → 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 → 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 o₁ and o₂ are identical, denoted by identical(o₁, o₂), if they are the same object.

**Shallow Equal:** Two objects o₁ and o₂ are shallow equal, denoted by shallow-equal(o₁, o₂), if their values are identical.

**Deep Equal:** Two objects o₁ and o₂ are deep equal, denoted by deep-equal(o₁, o₂), 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 peters from GradStudent's 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, 11]) 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 [11], role class hierarchies rooted in an object class [5], and the transition rules in the role classes in [21]), 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 [11], 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 [11], 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 additional 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 [8] is assumed between the class instance and the attribute’s value.

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.

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

18
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 the properties of being a *HS-Student* predefined 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 [19].

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 discussions. This research is partially supported by a research grant in Hong Kong RGC96/97.HKUST.757/96E.
References


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# 96112038;
   name "Andy";
   sex "male")

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

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

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

Figure 3: Example scripts for class, object, and role creation.
Figure 4: Example scripts to illustrate the `select` and `foreach` statements.

```sql
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")))
```

Figure 5: Illustration for the two method lookup modes.
Figure 6: Example scripts to illustrate the ownership of roles.
Figure 7: Object-role relationship for the computer science department.

![Diagram of object-role relationship for the computer science department]

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

1920 - Creation of object John who is a Person (and is-always a Person).
1931 - John becomes a high-school student of HK High School.
1938 - John becomes an undergraduate and also a club-member. Afterwards he becomes the chairman. After graduation, he continues his master degree in the same department.
1944 - John gets his first job, as an engineer in XYZ Inc. He buys a car at the same time. After a few years, he becomes a manager and changes his car to a better model.
1963 - John starts his own company and plays a role as the CEO. He retires when his son takes over the company.
1991 - Unfortunately, John has to be deleted from the database!