DOOR: A Dynamic
Object-Oriented Data Model
with Roles

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

Technical Report HKUST-CS96-12
May 8, 1996

*Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
Abstract

Traditional object-oriented programming languages do not support the dynamic type change of an object so as to model the behavior of real world entities which change their status over time. This is a severe limitation in the context of a database programming language. Moreover, traditional object-oriented languages do not model the fact that the behavior of real world entities may depend on the role that they play. This paper describes the data model for DOOR, a dynamic object-oriented database programming language with role extension, which addresses these problems. We distinguish object classes from role classes by defining roles (instances of role classes) as parts of objects (instances of object classes). Objects are represented by globally unique and unchangeable identifiers while roles are represented by the names of their role classes as well as their values. The data model described in this paper emphasizes the role representation as well as the player qualification for the role players.
1 Introduction

As described by Richardson and Schwartz [13], most object-oriented database (OODB) systems display serious shortcomings in their ability to model both the dynamic nature and the many-faceted nature of common, real-world entities. A commonly used example of this kind of entity is a person. While existing OODBs may capture the notion that a student is a person, they do not support the notions that a given person may become a student, that after graduation, that person ceases to be a student, and becomes an alumnus; and that he or she may also be an employee, a car-owner, a club member, etc. Throughout his or her life, a person gains and loses many roles. This issue has received attention under the term roles in database modeling at least since Bachman and Daya wrote about it in the context of the network data modeling approach in 1977 [3].

In the context of an object-oriented database model, roles can be used to gracefully partition messages for objects so that objects can receive and send different messages at different stages of their evolution/life-cycle [12]. The partitioning of messages for an object according to different roles has the advantage of allowing the designer (and possibly the implementer of the application) to concentrate on the life-cycle of an object in one role at a time. By adopting the ideas of roles to model a complex, evolving object, we can partition the objects into different roles such that objects can perform different actions at different stages in their life-cycle. Similarly we can specify the interactions between activities in terms of the dependencies among the roles of objects.

This paper describes the data model for DOOR, a dynamic object-oriented database programming language with role extension, being developed at the Hong Kong University of
Science and Technology. Besides traditional business applications, the target applications of DOOR are manufacturing applications (including CAD/CAM and robotics), multimedia applications, geographic information systems, and other complex applications which consist of large amounts of complex and dynamic data. The data model described in this paper focuses on the representation as well as the semantics of roles.

In our model, object class hierarchies are complemented by role class hierarchies whose nodes represent role classes (similar to the approach described in [8]). At any point in time, an entity is represented by an instance of an object class and instances of any role classes whose roles it currently plays. An object can play zero or more roles, and a role can only exist by being part of an object. Certainly, not every class’s instances 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. Therefore, we introduce player qualification as a component of a role class to specify a set of classes whose instances are qualified to play the roles.

Another problem typical to role modeling concerns referencing a role when there is an update on its player. For example, how do we treat a reference to the manager of a department (that should be of type Employee) when he or she leaves the company and is turned into a ‘normal’ person? Should we treat this problem similar to that of referencing a deleted object instead of a deleted role (and using techniques from object/class migration to solve this)? In our model, we identify a role by its role class name instead of a system-unique identifier which facilitates a flexible reference to a role. For example, objects reference a role of a role class named Manager instead of reference a role with role identifier say rid1103.
The organization of this paper is as follows. In section 2, an overview of roles will be presented. Section 3 defines the formal DOOR data model, which includes the definitions for values, roles, objects, role and object types, type hierarchies, methods, schema, and database in DOOR. Finally, section 4 compares DOOR with other related work and section 5 concludes this paper.

2 Overview of Role Modeling in DOOR

In DOOR, a role is conceptually like an object, except that it has a special relationship to other objects (or roles) which are said to play the role. A role can be played by an object or by another role. We now define a role more formally with 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 \( \text{inst}_\alpha(C) \) denote the set of all possible instances of class \( C \) with the state of the world\(^1 \) being \( \alpha \).

Then we assume there is a function called played-by in the model such that if \( OC \) is an object class and \( RC_1, RC_2 \) are role classes, then in each state \( \alpha \) of the world, we have

\[
\text{played-by}: \text{inst}_\alpha(RC_1) \rightarrow \text{inst}_\alpha(OC) \cup \text{inst}_\alpha(RC_2)
\]

where \( \text{played-by}(r) \ (r \in \text{inst}_\alpha(RC_1)) \) is called the player of \( r \), and \( \text{played-by} \) has the following properties:

\(^1\)Here we use the notation state of the world \([20]\) as a reference for time, to denote the time variant properties of dynamic/evolving objects.
(a) Let $R$ be a role class. For any state $\alpha$ of the world, $r_1 \in inst_\alpha(R)$ and $r_2 = played-by(r_1) \Rightarrow r_2 \neq r_1$.

(b) $played-by$ 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 $\text{club-member}$ of a credit card club, and being a general club-member, he then further join as a $\text{club-member}$ 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, it is possible to define 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 $played-by(e)$ [10]. 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 (including CHILD and ADULT) can at any moment be an EMPLOYEE and/or a STUDENT, but only an ADULT can at any moment be a CAR_OWNER.

By introducing the role class 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. This formalizes the difference with respect to the counting problem mentioned in [19].

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. The player relationship will be inherited to all the subclasses of PERSON, i.e., CHILD and ADULT in this case. However, only an ADULT can be a CAR_OWNER. Therefore, a CHILD cannot play the role CAR_OWNER.

Note that this inheritance property of player relationships also holds for role classes, not just object classes. However, since DOOR allows multiple inheritance, there are some

---

2 Class names in bold denote object classes, and class names in italic denote role classes.
cases where we need to conjunct the player constraints of the superclasses of a multiple inherited role class. Consider an interesting example shown in Figure 2, where a PERSON (including a CHILD or an ADULT) can be a STUDENT, but only an ADULT can be an EMPLOYEE. Hence a CHILD cannot play the role EMPLOYEE. The role class STUDENT-WORKER is formed by multiple inheritance from both EMPLOYEE and STUDENT classes. The player constraint of this STUDENT-WORKER is computed simply by the conjunction of its superclasses’ player constraints, i.e., a STUDENT-WORKER has to be a PERSON (since both an EMPLOYEE and a STUDENT have to be a PERSON) as well as an ADULT (since an EMPLOYEE has to be an ADULT) and ADULT is a PERSON, so the player constraint for STUDENT-WORKER is \( \text{PERSON} \land \text{ADULT} = \text{ADULT} \). This means that ADULT can play the role STUDENT-WORKER and CHILD cannot.

![Diagram](image)

Figure 2: PERSON (including CHILD and ADULT) can at any moment be a STUDENT, but only an ADULT can at any moment be an EMPLOYEE. Therefore, only an ADULT can be a STUDENT-WORKER.

Multiple inheritance in the object class hierarchy does not cause the conjunction of player constraints in the role hierarchy since a multiple inherited object class, which possesses the properties of all its superclasses, can play any roles which can be played by any one of its superclasses.
3 Formal Definition of the DOOR Model

This section presents the formal definition of the DOOR model, which is strongly influenced by the $O_2$ data model [4]. While presenting the model, we also discuss the motivations for the choices made in defining the model.

3.1 Basics

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

- A finite set 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, unit. We denote by $D$ the union of all domains $D_1, \ldots, D_n$.

- A countably infinite set $A$ of symbols called attributes.

- A countably infinite set $ID$ of symbols called identifiers, and a set $OID$ which is defined to be $OID = ID \cup \{\bot_o\}$. The elements of $ID$ will be used as identifiers for objects, and the special symbol $\bot_o$ represents the undefined identifier which is used for a tombstone object, i.e., an undefined object / a deleted object. The idea of tombstone object is useful in handling the object deletion problem (such as dangling reference) [25].

- A finite set $M$ whose elements are called methods and play the role of operations on our data structures. We will define methods more formally later. For the moment, 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 (to be defined later).
- **OCnames** is the set of names for object types (object classes).
- **RCnames** is the set of names for role types (role classes).
- **Snames = OCnames ∪ RCnames** is the set of names for structured types that is countably infinite and disjoint with **Bnames**.
- **Tnames = Bnames ∪ 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**

(a) Every element of \( D \cup \{\bot\} \) is called a basic-value. The special symbol \( \bot \) represents the undefined (i.e., null) value.

(b) Every finite subset of \( \text{OID} \) is called a set-value. Set-values are denoted in the usual way using brackets.

(c) Every finite partial function from \( A \) into \( \text{OID} \) 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. \( \square \)
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 [7]. 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 [19]), 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 represent roles by the names of their role classes as well as their values, instead of using globally unique identifiers.

Moreover, in DOOR, an object or a role can play zero or multiple roles at any point in time. Therefore, we define a role (instance) in a recursive manner as follows:

**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., \( val \in V \)), \( Rs \) is the

\footnote{Although the motivations are different, this representation is similar to the aspect approach by [13].}
set of roles being played by r (i.e., ∀ rs ∈ Rs, played-by(rs) = r), and m ⊆ M is a set of methods. We denote by R the set of all roles; that is, $R = \mathcal{R}\text{names} \times \mathcal{V} \times 2^R \times 2^M$. □

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.

3.4 Objects

We can now define the notion of an object.

Definition 3

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

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

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

- Given an object \(o = (i, n, v, r, m)\), \(\text{ident}(o)\) 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", age: 25],
           { (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, \textit{ident(object1)} = oid1011, \textit{class(object1)} = Person, \textit{roles(object1)} = \{(student, \ldots )\}, \textit{all-roles(object1)} = \{(student, \ldots ), (student-worker, \ldots )\}, and \textit{value(object1)} = [name: "Peter", \ldots ].

\textbf{Definition 4} (Consistent Objects) A set \(\Theta\) of objects is consistent iff:

(a) \(\Theta\) is finite.

(b) The ident function is injective on \(\Theta\) (i.e., there is no pair of objects with the same identifier).

(c) For all \(o \in \Theta\), \textit{ref}(o) \subseteq \textit{ident}(\Theta)\) (i.e., every referenced identifier corresponds to an object of \(\Theta\)).

\[\square\]

3.5 Types

A type is an abstraction that allows the user to encapsulate in the same structure both data and operations. In our model, the static component of a type is called a type structure. Our notion of type bears some similarity to abstract data types. 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 \( B\text{O}t\)ype, is a pair \((n, m)\) where \( n \in \text{Bnames} \) and \( m \subseteq \mathbb{M} \). A structured object type, or \( S\text{O}t\)ype, is one of the following:

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

- A triple \((s, t, m)\) where \( s \in \text{Snames} \), \( t \) is a finite partial function from \( \mathbb{A} \) to \( \text{Tnames} \), and \( m \subseteq \mathbb{M} \). 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 \( s \in \text{Snames} \), \( t \in \text{Tnames} \), and \( m \subseteq \mathbb{M} \). 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 \((n, m, \text{o}c)\) and \((s, t, m, \text{o}c)\) respectively. \text{o}c is a subset of \( \text{OCnames} \), which is called the player qualification and specifies the qualified players. To support later definitions, we also define an auxiliary function \( \text{players} \) which returns the player qualification (i.e., \( \text{o}c \)) of a role type.
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 one of the classes specified 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.

3.5.1 Type Structures

In this subsection we are interested in the static part of a type, its structure. Given a type $t$, its structure part will be denoted by $\text{struct}(t)$ and its methods part by $\text{methods}(t)$. Intuitively, a type structure is a type in which the methods part is hidden; that is, it is the data part of the type. Note that recursion (or transitive recursion) is allowed in type definitions; that is, one of the $s_i$ may be $s$.

We need a notion of consistency for a set of expressions defining type structures. In order to define it formally, we need some technical notation.

- If $t$ is a type, then $\text{name}(t)$ is the name of the type, that is, the first component in its definition.

- If $st$ is a type structure associated to the type $t$, the name of the type structure $st$ is the name of $t$, i.e., $\text{name}(st) = \text{name}(t)$. 

14
If $st$ is a type structure associated to a type $t$, the set of all type names appearing in the type structure $st$ is called the set of type references for $st$. We denote this set by $\text{ref}(st)$.

**Definition 6** A set $\mathcal{S}$ of structured-type structures is a schema iff:

(a) $\mathcal{S}$ is a finite set.

(b) $\text{name}$ is injective on $\mathcal{S}$ (only one type structure for a given name).

(c) $\forall st \in \mathcal{S}, \text{ref}(st) \cap \text{Snames} \subseteq \text{name}(\mathcal{S})$; that is, there are no dangling identifiers.

**Definition 7** Let $\mathcal{S}$ be a schema, and let $\Theta$ be a consistent subset of the universe of objects $O$. An interpretation $I$ of $\mathcal{S}$ in $\Theta$ is a function from $\text{Tnames}$ to $2^{\text{id}(\Theta)}$ that satisfies the following properties.

(a) For every basic type, take its corresponding natural domain.

(b) For $t \in OT$, $I(t) = \{\bot_o\} \cup \{i \mid (i, \text{name}(t), v, r, m) \in \Theta\}$.

(c) For $t \in RT$, $I(t) = \{\bot_o\} \cup \{i \mid \text{name}(t) \in \text{name}(\text{allroles}((i, n, v, r, m)))$ where $(i, n, v, r, m) \in \Theta\}$.

(d) $I([a_1 : t_1, \ldots, a_n : t_n]) = \{[a_1 : v_1, \ldots, a_n : v_n, a_{n+1} : v_{n+1}, \ldots, a_{n+m} : v_{n+m}] \mid v_i \in I(t_i), i = 0, 1, \ldots, n\}$.

(e) $I(\{t\}) = \{\{v_1, \ldots, v_n\} \mid v_i \in I(t), i = 0, 1, \ldots, n\}$.
Definition 8

(a) An interpretation $I$ is smaller than an interpretation $I'$ iff $\forall s \in \text{Tnames}, I(s) \subseteq I'(s)$.

(b) Let $\mathcal{S}$ be a schema and let $\Theta$ be a consistent set of objects. The model $\mathcal{M}$ of $\mathcal{S}$ in $\Theta$ is the greatest interpretation of $\mathcal{S}$ in $\Theta$.

Intuitively, the model $\mathcal{M}(s)$ of a structured type structure of name $s$ is the set consisting of all objects (or identifiers of objects) having this structure.

3.6 Methods

We assume a countable set $\text{Mnames}$ of symbols which is used as names for methods.

Definition 9 Let $\mathcal{S}$ be a schema. A signature over $\mathcal{S}$ is an expression of the form

$$s_1 \times s_2 \times \ldots \times s_n \rightarrow s$$

where $s_1, s_2, \ldots, s_n,$ and $s$ are type names corresponding to type structures in $\mathcal{S}$, or basic type names.

A method $m$ is a pair $(n, \sigma)$ where $n$ is a method name (i.e., $n \in \text{Mnames}$) and $\sigma$ is a signature. We denote by $\text{name}(m)$ the name of the method $m$, and by $\text{sig}(m)$ the signature of the method $m$.

Definition 10 Let $m = (n, s_1 \times s_2 \times \ldots \times s_n \rightarrow s)$ be a method. We say that $m$ is defined on $s_1$. That is, given an object $o = (i, n, v, r, m)$, the first component of the
signature of every method of \( m \) is a type structure whose interpretation contains \( o \). □

**Definition 11** (Interpretation) Let \( S \) be a schema and \( \sigma \) a signature over \( S \) (\( \sigma = s_1 \times s_2 \times \ldots \times s_n \rightarrow s \)). If \( \Theta \) is a consistent set of objects, then the model of \( \sigma \) in \( \Theta \) is the set of all partial functions from \( M(s_1) \times \ldots \times M(s_n) \) into \( M(s) \) where \( M(s_k) \) is the model in \( \Theta \) of the structure of \( S \) identified by \( s_k \). □

### 3.7 Type Systems

**Definition 12** A set of types \( \Pi \) is a type system iff:

(a) The set of signatures associated to \( \Pi \) is a schema.

(b) For all types \( t \in \Pi \), and for all methods \( m \in \text{methods}(t) \), \( m \) is defined on \( \text{struct}(t) \). (where methods\((t)\) denotes the set of methods of the type \( t \).)

□

**Definition 13** Let \( \Pi \) be a type system, and let \( t \) and \( t' \) be two types of \( \Pi \). We say that \( t \) is a subtype of \( t' \) (denoted as \( t \leq t' \)) iff:

(a) \( M(\text{struct}(t)) \subseteq M(\text{struct}(t')) \) for all consistent sets \( \Theta \)

(b) \( \forall m \in \text{methods}(t), \exists m' \in \text{methods}(t') \) such that \( \text{name}(m) = \text{name}(m') \) and \( M(\text{sig}(m)) \subseteq M(\text{sig}(m')) \) for all consistent sets \( S \).

□

We denote \( t \leq \cdots \leq t' \) by \( t \leq^* t' \). After defining the notion of subtype, we extend the definition of consistent objects for DOOR database.
**Definition 14** (Consistent Objects Revisited) A set \( \Theta \) of objects is consistent iff:

(a) \( \Theta \) is finite.

(b) The ident function is injective on \( \Theta \) (i.e., there is no pair of objects with the same identifier).

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

(d) For all \( o \in \Theta \) and for all \( r \in \text{allroles}(o) \), \( \text{class}(p) \preceq p' \) and \( p' \in \text{players}(\text{class}(r)) \) where \( p = \text{played-by}(r) \).

\( \square \)

The condition (d) ensures that all roles are played by qualified players, according to the player qualification specified in their role types.

### 3.8 Databases

A database is a type system together with a consistent set of objects that represent the instances of the types at a given moment.

**Definition 15** A database is a tuple \((\Pi, \Theta, \leq_{\text{door}}, \text{ext}, \text{impl})\), where

- \( \Pi \) is a type system, and \( S \) is the associated schema.

- \( \Theta \) is a consistent set of objects.

- \( \leq_{\text{door}} \) is a partial order among \( \Pi \).
\* ext is an interpretation of S in Θ.

\* impl is a function assigning a function implementation (i.e., a lambda expression in DOOR) to every method m of a type t.

\[ \square \]

4 Examples

We describe two examples in this section to further illustrate the DOOR data model presented in last section. The first example is based on the example shown in Figure 2 with particular emphasis on player qualification. The second one is about how the object reference problems (such as the dangling reference problem mentioned in [25]) are handled by DOOR. For simplicity and clarity, the method component is omitted and denoted by an underscore.

4.1 Illustration of Player Qualification

Assume the class hierarchies shown in Figure 2, i.e., Child ≤ Person, Adult ≤ Person, Student-Worker ≤ Student, Student-Worker ≤ 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], _)
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 in DOOR database. Mary, who is playing a role Employee, can be represented by

\[(oid1022, \text{Person}, \{\text{name: "Mary"}, \text{sex: "female"}, \text{spouse: oid1031}, \})\]

\{(Employee, [employer: oid1113, position: "typist", salary: 8000], {}, _), _\}

### 4.2 Object Reference Problems

DOOR in fact supports many kinds of dynamic updates and type evolution. However, because of space limitation here, we can only briefly discuss the object reference problems caused by object deletion or role change.

Assume that John is working in a company with object identifier oid1088, and his supervisor is the manager of the company, i.e., a Person called Peter with object identifier oid1023.

\[(oid1023, \text{Person}, \{\text{name: "Peter Lee"}, \text{sex: "male"}], \}

\{(Manager, [company: oid1088, salary: 32000], {}, _), _\}\]

\[(oid1027, \text{Person}, \{\text{name: "John Ng"}, \text{sex: "male"}], \}

\{(Clerk, [company: oid1088, salary: 9000, supervisor: oid1023], {}, _), _\}\]
If for some reasons, the object oid1023 is deleted from the database, John’s supervisor will have a dangling pointer. This problem can be solved by replacing each reference to oid1023 with a tombstone which denotes a deleted object. However, with this approach, we need to update all these references again if we later have another person (another object of object class Person) being a manager of company oid1088.

Moreover, if Peter is still in the database, but he might have resigned and turned back to a ‘normal’ person, then John’s supervisor will still point to oid1023.

(oid1023, Person, [name: "Peter Lee", sex: "male"],
  {}, _)  
(oid1027, Person, [name: "John Ng", sex: "male"],
  {(Clerk, [company: oid1088, salary: 9000, supervisor: oid1023], {}, _)}, _)

If we again replace each reference to oid1023 with a tombstone, we may overwrite wrong, valid references (such as those references to Peter from his Person perspective).

Worst still, if Peter has resigned and Linda (obviously with different object identifier) become the new manager of company oid1088,

(oid1032, Person, [name: "Linda Lau", sex: "female"],
  {(Manager, [company: oid1088, salary: 33000], {}, _)}, _)
(oid1027, Person, [name: "John Ng", sex: "male"],
  {(Clerk, [company: oid1088, salary: 9000, supervisor: oid1023], {}, _)}, _)

then we need an intelligent update which replaces all references to oid1023 with oid1032 if they ‘treated’ oid1023 as manager.

In DOOR, users can specify the role type definition for Clerk as follows:
For object deletion, all references to the deleted object will be replaced by a tombstone which denotes the ‘dead’ object. Moreover, in DOOR, there are two operations to drop a role from an object. One of them is to destroy a role which is currently being played by an object. Another operation is to release a role from an object and all values of the role still persist. In the above example, all references of type (or subtype of, if any) Manager to oid1023 (Peter) will be deleted and replaced by a tombstone if we destroyed the role Manager of oid1023. Alternatively, if the role Manager is released from oid1023, all references of type (or subtype of, if any) Manager to oid1023 (Peter) will be replaced by a tombstone with an undefined identifier (⊥o) and then the role Manager (with its values) is moved from oid1023 to the tombstone object. With this a new object can pick this role so that the role values are preserved.

5 Related Work

The concept of a role was already defined in 1977 by Bachman and Daya [3] in the context of the network data modeling approach. Because of the mixture of logical and implementation issues in their approach, the characteristics of role and role change were not captured in a purely logical, implementation-independent manner. Recently, the concept surfaced again and its significance was realised in work on office modeling [12, 18], semantic modeling [8, 13, 15], object-oriented modeling [11], manufacturing system and robotic applications [21, 23, 24], and spatial applications [9, 22], 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.).

**ORM, Views, and Aspects** The ORM work by Pernici [12] defined roles by encapsulating roles into classes. In contrast with ORM, Richardson and Schwarz in [13] have introduced the concept of *aspects* to support modeling of roles, by allowing different aspects (perspectives) to be *attached to* a class directly (where each aspect is a special type of implementation that extends the class). A related idea, based on the concept of view was proposed by Shilling and Sweeney [16]. In views, an object is equipped with multiple interfaces (views). Every interface has its own set of methods and the interfaces of an object are separate and independent of each other. The object behavior depends on the interface used to access it, and the object identity is preserved across the various views allowing one to model multiple and independent roles. Compared with DOOR, all these models have a similar limitation in the sense that inheritance between/among the roles (aspects) is not supported. However, aspects and views are similar to DOOR in that they all do not have unique identifiers for roles (aspects/views).

**IRIS, Clovers, and Galileo** IRIS [5] is an OODBMS equipped with explicit features to model behavioral evolution of entities. IRIS objects may acquire or lose types during their life, retaining their identity; however, IRIS does not support the observation of an object from different perspectives while DOOR does. In fact, despite type multiplicity, an object, in a fixed instant of its life, always exhibits a uniform behavior no matter the context from which it is observed. Clovers [17] is a strongly typed language which can model entities with multiple and independent roles. The object behavior depends
strictly on the type through which the object is observed and there is no late binding. The difference from DOOR is the impossibility in Clovers of explicitly referring to the types in which one is interested. Galileo [2] allows objects to be dynamically extended with new types but the role mechanism is not provided. Thus objects always exhibit a uniform behavior no matter what type they are accessed through.

Sciore’s approach and Schrefl-Neuhold’s approach Taking a dramatically different approach of consolidating a prototype-based and an object-oriented system, Sciore’s work, described in [15], allows classes to be viewed (and realized) as an individual object’s auxiliary roles or perspectives, and objects 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 approach was also proposed and used by Schrefl and Neuhold in [14], except that in this approach, possible object hierarchies must be predefined by role specialization classes at the type level, and this is not the case in DOOR.

Albano et al.’s work using Fibonacci and Gottlob et al.’s work using Smalltalk

The most recent work on roles includes Fibonacci [1] and a Smalltalk-based role extension to objects [6]. Albano, Bergamini, Ghelli and Orsini have developed a new database language called Fibonacci [1] that can cope with multifaceted objects. In their language, an object has two internal components: a set of blocks where data and methods are stored, and a dispatcher in charge of directing the messages to the right method. Each block represents a role of the object, and the dispatcher helps bind methods to messages. Gottlob, Schrefl and Rock [6] also took a combined approach of consolidating the class-based and
the prototype-based paradigms. The differences from Albano et al.’s approach include the support of multiple instantiation of roles, and combining class and role hierarchies. To some extent, both Albano et al’s work and Gottlob’s work are similar to ORM in the sense that roles are also rooted in (though not encapsulated into) a class, and these roles can be inherited from the class to its subclasses, while in DOOR, a player qualification approach is used. The latter approach is more flexible in the sense that any object can be a player of a particular role as long as they qualify. Different from ORM, aspects, and views (but similar to the models in [1, 24] and DOOR), however, the roles attached to a class in both approaches can form their own “is-a” hierarchy.

Summary In summary, all of these works do not provide detailed discussion on role identification (except [19]). They do not consider problems such as dangling references, etc. caused by the object updates and class migration. Furthermore, the most important difference with DOOR (from the semantic point of view) is that the ideas of having qualification between roles and role players (and hence the role acquisition sequence) as well as allowing roles as role players were omitted.

6 Conclusion

The main contribution of this paper is to propose a data model for DOOR, a dynamic object-oriented database programming language with role extension. In DOOR, 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, which have different types and can
answer in different ways to a message.

Furthermore, player qualification in DOOR 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.

Instead of being represented by globally unique identifiers, roles are identified by the
name of their role classes together with the role values. This representation solves the
practical implementation problems (including dangling references and the representation
of historical information) of object updates as well as class migration.

Our ongoing work is on the full implementation of DOOR. The prototype based on meta-
object protocol (MOP) on the language Scheme, and using a file based associative string
database to support persistence, is under development. The role playing mechanism,
which is similar to the implementation of delegation in Clovers [17], has been implemented
as a MOP.

References

  R. Agrawal, S. Baker, and D. Bell, editors, Proceedings of the 18th International Conference
  on Very Large Databases, pages 39–51, Dublin, Ireland, August 1993.


28


