Proceedings of International Conference on Distributed
Systems, Software Engineering and Database Systems

DOOR = 'Object with Roles' + 'Schema Evolution'

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

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong

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

Abstract

Roles and schema evolution are becoming popular research issues in object-oriented databases, and have been proven to be useful in dynamic and evolving applications. However, not much work has been done on discussing their differences. In particular, there is no programming language or database system which includes and integrates both mechanisms. Our objective is to integrate these two mechanisms such that a role is used to model evolving objects and schema evolution is used for the flexible design of dynamic and evolving systems. The resultant language is called DOOR, a dynamic object-oriented database programming language with role extension, which incorporates both roles and schema evolution constructs. Important issues which need to be addressed to integrate these two approaches are discussed in this paper.

1 Introduction

Roles and schema evolution are becoming popular research issues in object-oriented databases, and have been proven to be useful in dynamic and evolving applications. Recently, there has been a significant upsurge of interest in extending conventional object-oriented languages/systems (e.g., [3, 7, 10, 22, 23, 25, 26]) to facilitate the modeling of dynamic and evolving applications. Some of these approaches are based on the notion of *roles* (e.g., [3]), while others are based on the idea of schema evolution or type evolution (e.g., type evolution in [26], dynamic subclasses in [25], or dynamic inheritance in SELF [23]).

Roles $\neq$ type evolution, and both of them are useful in dynamic and evolving applications. As described by Richardson and Schwartz [17], 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 customer, a car owner, a club member, etc. 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 [5]. In particular, the role mechanism can be used to partition the messages for objects such that objects can receive and send different messages at different stages of their evolution/life-cycle [16]. 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. Moreover, objects can perform different actions at different stages in their life-cycle and can specify the interactions between activities in terms of the dependencies among the roles of objects. From the point of view that roles can provide multi-perspective access of an object, roles are similar but different (as we will present later in this paper) to the idea of *views* [18, 19] for objects.

Alternatively, schema evolution (including type evolution, etc.) is intended for the support and management of schema changes. The motivation of having schema change is that class hierarchy design is the main theme of schema design for object-oriented databases. The practical applications of object-oriented databases, such as CAD/CAM, AI, and multimedia office systems, require the ability to make a wide variety of changes to the database schema dynamically; this process is called *schema evolution.* The types of schema changes required include creation and deletion of a class, alteration of the IS-A relationship between classes, addition and deletion of instance variables and methods, and others. The users tend to arrive at the desired schema for objects through trial and error, and it would be useful if they can debug the schema on the spot (i.e., dynamically). Therefore, schema change operations are required for designing the desired schema.

According to these different motivations and advantages of having roles and schema evolution for modeling dynamic/evolving applications, we found that it would be desireable to have an object-role model and its corresponding language which supports both roles and schema evolution. Also, we totally agree with work done by Wieringa et al. [25] in which they discuss the difference (from the conceptual modeling point of view) between roles and type evolution.

(Type change is supported by dynamic subclassing in [25].)

However, there is no existing programming language or database system which supports both roles and schema evolution. Therefore, in this paper we propose a dynamic object-oriented database programming language with role extension called DOOR, being prototyped at the Hong Kong University of Science and Technology, which supports both role modeling and schema evolution. We describe the underlying object-role model and the schema evolution support in DOOR. We also discuss the issues which need to be addressed to integrate these two approaches.

The organization of this paper is as follows. In section 2, the object-role data model for DOOR is defined. We then present the schema evolution supported by DOOR in section 3. In particular, we discuss the integration issues arising from support of both the role mechanism and schema evolution. They include the reference problems caused by object deletion and role change, issues about objects which change type and objects which change role, the change of class definition, and the change of class lattice. Section 4 discusses other related work and section 5 describes the current status of the prototype and future work. Finally, section 6 concludes the paper.

2 The Object-Role Data Model for 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 the DOOR data model formally with the following notation and definitions.

Let P be an infinite set of property functions. Each $p \in P$ can be a value from a simple enumeration type, an object instance from some class, an arbitrarily complex function, or an object method. Each $p \in P$ has a name and signature (i.e., domain types). For simplicity, we assume that all $p \in P$ have a unique property identifier. Let T be the set of all types. For $t \in T$, **properties**(t) corresponds to the set of property functions of t and **domain**$_p(t)$ denotes the domain of p in t.

Let $C = OC \cup RC$ be the set of all classes. An **object class** $oc_i \in OC$ has a unique class name, a type description and a set membership, and a **role class** $rc_i \in RC$ has a unique class name (which is distinct from all object class names), a type description, a set membership and a *player qualification* (which is a set of class names that specify which classes' instances are qualified to be a role player of this role class). The type associated with a class corresponds to a common interface for all instances of the class. We refer to the name of the type associated with a class c (where $c \in C$) by **type**(c) and to the set of property functions defined for c by **properties(type**(c)**)**, or short, **properties**(c). If $p \in P$ is a property function defined for c, then we refer to the domain of p for c by **domain**$_p(c)$. In particular, if c is a role class (i.e., $c \in RC$), then we refer to the player qualification (a set of class names) by **players**(c). An **object class** is also a container for a set of objects, and **a role class** is a container for a set of roles.

Definition 1. For two classes $c_1, c_2 \in C'$ (where $C' \subseteq OC$ or $C' \subseteq RC$), c_1 is called a **subset** of c_2, denoted by $c_1 \subseteq c_2$, iff $i \in c_1 \Longrightarrow i \in c_2$. □

Definition 2. For two classes $c_1, c_2 \in C'$ (where $C' \subseteq OC$ or $C' \subseteq RC$), c_1 is called a **subtype** of c_2, denoted by $c_1 \preceq c_2$, iff (**properties**$(c_1) \supseteq$ **properties**(c_2)) and ($\forall p \in$ **properties**(c_2), **domain**$_p(c_1) \subseteq$ **domain**$_p(c_2)$). □

Definition 3. For two classes $c_1, c_2 \in C'$ (where $C' \subseteq OC$ or $C' \subseteq RC$), c_1 is called a **subclass** of c_2, denoted by c_1 *isa* c_2, iff $c_1 \preceq c_2$ and $c_1 \subseteq c_2$. □

Definition 4. Let $inst_\alpha(C)$ denote the set of all possible instances of class C with the state of the world being α. We define 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 α of the world, we have

$$\textit{played-by: } inst_\alpha(RC_1) \longrightarrow inst_\alpha(OC) \cup inst_\alpha(RC_2)$$

where *played-by(r)* ($r \in inst_\alpha(RC_1)$) is called the *player* of r, and *played-by* has the following properties:

1. For any state α of the world, $r_1 \in inst_\alpha(R)$ and $r_2 = \textit{played-by}(r_1) \Longrightarrow r_2 \neq r_1$;
2. *played-by* is neither a surjective function nor injective function.

□

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

Definition 5. An **object-role schema is** a directed acyclic graph $S = (V, E)$, where $V = V_o \cup V_r$ is a finite set of vertices and $E = E_{isa} \cup E_{played\text{-}by}$ is a finite set of directed edges. Each element in V_o corresponds to a class oc_i and each element in V_r corresponds to a class rc_j. E_{isa} corresponds to a binary relation on $V_o \times V_o$ or $V_r \times V_r$ that represents all direct *isa* relationships between all pairs of object classes in V_o and between all pairs of role classes in V_r, while $E_{played\text{-}by}$ corresponds to a binary relation on $V_r \times V_o$ or $V_r \times V_r$ that represents all direct *played-by* relationships between a role class in V_r and an object class in V_o, or between a pair of role classes in V_r. In particular, each directed edge e from c_1 to c_2, denoted by $e = < c_1, c_2 >$, represents the direct *isa* relationship between the two classes (c_1 *isa* c_2). There are two designated root nodes, one called **Object** which contains all object instances of the database, and another called **Role** which contains all role instances of the database. Both of their type descriptions are empty. □

Definition 6. An **object** o is a quadruple *(oid, class, properties, roles)* where *oid* is a globally unique and unchangeable identifier together with a tombstone symbol say $\perp$, *class* is the class name of which *properties* is defined, *properties* is the property functions inherited from the class, and *roles* is a set of roles being played by o. Let O be an infinite set of object instances. The collection of objects that belong to an object class oc is denoted by $\mathbf{inst_O}(oc) = \{o \mid o \in oc\}$. A **role** r is a triple (*class*, *properties*, *roles*) where *class* is the class name of which *properties* is defined, *properties* is the property functions inherited from the class, and *roles* is a set of roles being played by r. Similarly, let R be an infinite set of role instances. The collection of roles that belong to a role class rc is denoted by $\mathbf{inst_r}(rc) = \{r \mid r \in rc\}$. □

The tombstone symbol $\perp$, included as an object identifier, handles the reference problem for object deletion. This will be further illustrated later.

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) [14]. 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.

By introducing the role class hierarchy into the object class hierarchy, our role model is formed. These two types 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 [24].

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 the object-role schema in Figure 1[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.

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

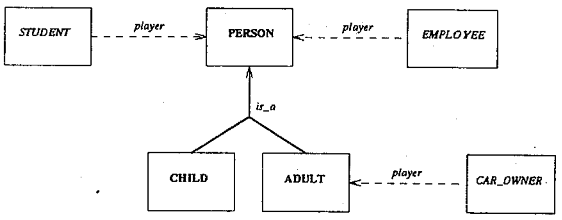

Figure 1: An object-role schema which shows that PERSON (including CHILD and ADULT) can at any moment be an EMPLOYEE and/or a STUDENT, but only an ADULT can at any moment be a CAR_OWNER.

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 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 PERSON $\wedge$ ADULT = ADULT. This means that ADULT can play the role STUDENT-WORKER and CHILD cannot.

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 Schema Evolution

Schema evolution may involve many different kinds of dynamic changes. For example, the class changes supported by DOOR include:

- update (add/delete/rename) an instance variable
- change the type of an instance variable
- update (add/drop) a superclass to/from a class's superclass list
- update the class hierarchy (add/delete/rename a class)

In addition, the following changes are supported if the class is a role class:

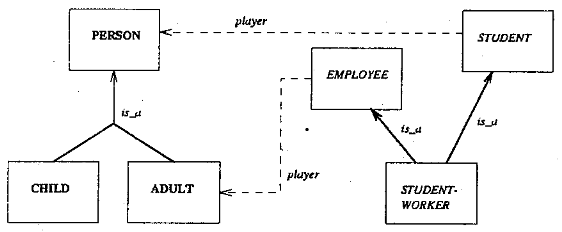

Figure 2: An object-role schema which shows that PERSON (including CHILD and ADULT) can at any moment be a STUDENT, but only an ADULT can at any moment be an EMPLOYEE. Therefore, only an ADULT can be a STUDENT-WORKER.

- adding a player to a role class's player qualification list
- deleting a player from a role class's player qualification list

Other changes to objects or roles (instances) include:

- update (create/delete) an object of an object class
- change the type of an object
- update the values of an object
- update (create/delete) a role for an object
- update the values of a particular role of an object
- transfer (copy/move) a role from one object to another object

Due to space limitations, we select a few interesting changes from the above lists and discuss their important issues in the following subsections.

3.1 Model Invariants

The following invariants are imposed for the DOOR data model. The invariants hold at every quiescent state of the schema, i.e., before and after a schema change operation. They provide a basis for the definition of the semantics of every meaningful schema change, by ensuring that the change does not leave the schema in an inconsistent state (one that violates any invariant).

Class Lattice Invariant The *isa* relationship forms two lattices, of which the predefined object class **Object** and the pre-defined role class **Role** are the only two roots.

Unique Name Invariant Each instance variable and method defined or inherited by a class must have a unique name. Each class (no matter whether an object class or a role class) must have a unique name.

Full Inheritance Invariant A class inherits the union of instance variables and methods from its superclasses, unless it defines an instance variable or method with the same name. If more than one superclass defines the same instance variable or method, the one inherited is the one defined by the superclass that appears earliest in the class's superclass list.

Type Compatibility Invariant If a class t defines an instance variable with the same name as an instance variable it would otherwise inherit from superclass s, the type of t's variable must be a subclass of the type of s's variable.

Typed Variable Invariant The type of each instance variable must have a corresponding class in the class lattice.

Role-Player Qualification Invariant A role of role class R has to be played by a player (an object or a role) which is of a class specified in the player qualification of R.

Besides these invariants, we would like to *keep as much information as possible* during the schema change process because there is no way to get back the information once it is lost during the process.

3.2 Reference problems caused by Object Deletion and Role Change

The dangling reference problem caused by object deletion indeed has been noticed and mentioned by many researchers (such as [1, 26]). In this subsection, we describe an approach to deal with the problem concerning the explicit deletion of objects or the role changes of objects.

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, Person, [name: "Peter Lee", sex:
  "male"], {(Manager, [company: oid1088,
  salary: 32000], {}, _)}, _)
(oid1027, Person, [name: "John Ng", sex:
  "male"], {(Clerk, [company: oid1088,
  salary: 9000, supervisor: oid1023], {},
  _)}, _)
```

If for some reason, 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) become the 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 wrongly overwrite valid references (such as those references to Peter from his Person perspective).

Worse still, if Peter has resigned and Linda (obviously with different object identifier) becomes 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. To deal with this problem, in DOOR, users can specify the role type definition for **Clerk** as follows:

```
(Clerk, [company: Company, salary: Integer,
  supervisor: Manager],_, {Person})
```

For object deletion, all references to a deleted object will be replaced by a tombstone which denotes the 'dead' object. For role change, adding a role to an object obviously does not cause any reference problem. However, this is relatively more complicated to delete a role from an object. 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 ($\perp$) and then the role **Manager** (with its values) is moved from **oid1023** to the tombstone object. With this approach, a new object can play this role so that the role values are preserved and reused.

To illustrate this further, consider an interesting example that a company has two managers (roles m_1, m_2) played by two different persons (objects p_1, p_2). If these two managers are referenced by other objects as shown in Figure 3(a), and then both the persons are deleted. The references to these deleted objects and their playing roles are held by two tombstone objects as shown in Figure 3(b). Since the original identifications of the objects (i.e., p_1, p_2) have been deleted and replaced by the tombstones, access and identification to the roles (i.e., m_1, m_2) has to pass

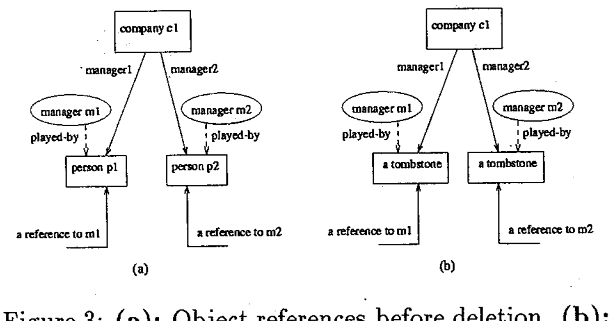

Figure 3: **(a):** Object references before deletion. **(b):** Object references after deletion.

through the company c_1. If no such access point exists, that is, there is no reference to the tombstones, these tombstones along with their roles will be destroyed by garbage collection in DOOR.

This approach is new in that it uses a tombstone object to hold a role (or multiple roles) such that its (or their) values can be preserved and passed to another object. Moreover, the use of a tombstone (to denote an object with undefined object identifier) can be used to hold a role such that it can be initialized by assigning values to the role attributes. For example, we can initialize the **company** attribute of role **Manager** to a company with object identifier **oid1088** such that someone who plays this role will automatically become a manager of *this* company.

3.3 Objects that change Type versus Objects that change Role

Objects that change type (also called object migration) cannot be simply implemented by creating a new object in class C', copying properties from object a and deleting a from C. This is because the object before and after the change should be referred to by the same object identifier. Therefore, a specific object migration function (or a similar function, e.g. the **become:** primitive in Smalltalk-80 [9]) has to be implemented in a level in which object identifiers can be accessed and manipulated. DOOR is equipped with a similar migration function to support the type change of objects. However, even with a specific migration function, problems with type checking (type violation) may still occur [26]. To illustrate these problems further, let us define object migration more formally.

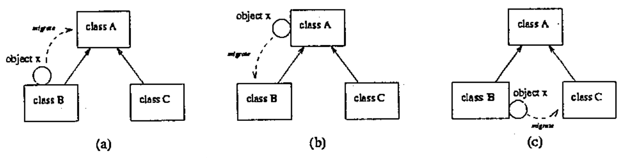

Figure 4: Assume the world will reach state α before α': **(a)** $x \in inst_\alpha(B) \subset inst_\alpha(A)$ and then $x \in inst_{\alpha'}(A)$ but $x \notin inst_{\alpha'}(B)$. **(b)** $x \in inst_\alpha(A)$ but $x \notin inst_\alpha(B)$ and then $x \in inst_{\alpha'}(B) \subset inst_{\alpha'}(A)$. **(c)** $x \in inst_\alpha(B)$ but $x \notin inst_\alpha(C)$ and then $x \notin inst_{\alpha'}(B)$ but $x \in inst_{\alpha'}(C)$.

Object c is said to migrate to class $C' \in OC$ from class $C \in OC$ if there exists states α, α' of the world where $\alpha \xrightarrow{*} \alpha'$ [2], then we have

$$c \notin inst_\alpha(C') \text{ and } c \in inst_{\alpha'}(C')\text{ , } \textit{and/or}$$

$$c \in inst_\alpha(C) \text{ and } c \notin inst_{\alpha'}(C)$$

Figure 4 shows the scenarios corresponding to this definition. Figure 4(a) corresponds to the second line of the above definition and Figure 4(b) corresponds to the first line. Figure 4(c) (which corresponds to the *and* case, i.e., both first and second lines, in the above

[2]To save space, we use $\alpha \xrightarrow{*} \alpha'$ to denote that the world will reach the state α before α'.

definition) describes a compound case which can be decomposed into the other two cases by considering x migrates to class A first and then migrates to class C.

The type violation (mismatch) problem will not occur in case (b) because an object changes its type A to A's subtype B so that all references to the object of type A will still be compatible with the object of type B. However, type violation may occur for case (a) and hence (c) (since (c) is a combination of case (a) and (b)). Similarly, the player qualification checking is needed for object migration processes like case (a) and hence (c).

If there are any references (of type X) to an object o (certainly of a subtype ($\preceq^*$) of X), and o changes to a new type Y which is not a subtype of X, then type violation occurs. In DOOR, we follow the approach mentioned in [26] that a tombstone will be used in place of the type-mismatched object for these references. Moreover, if there is a player qualification violation to the object of the new type, the roles which the object is no longer qualified to play will be moved to a tombstone object. These roles (including their states and values) can then be preserved for other qualified objects to play. However, if these roles are not referenced by any other objects/roles, it will be garbage collected (similar to the deletion case mentioned in the previous subsection). This is because in DOOR, a role can only exists by associating with an object (including a tombstone object) and a tombstone object will be garbage collected unless it is being referenced.

The difference between objects that change type and objects that change role, from the conceptual point of view, was pointed out by Wieringa et al. [25]. Furthermore, for languages like DOOR, which support the playing of multiple roles (in particular, multiple roles of the same role type), role change does not necessarily cause the same effect as an object's type change. This difference is also underlined by the difference between role class and object class being illustrated by the counting problem mentioned in [24].

3.4 Class Definition's Changes and Class Lattice's Changes

Changes of class definition which corresponds to changes of type definitions have been investigated by a number of researchers which include [6, 26]. With the support of object migration mentioned in last subsection, one simple way to support dynamic class definition's changes is: create a class (say class A') which is new version of the class (say class A) to be changed; then migrate all objects from the old class to the new class (i.e., from class A to class A'); afterwards, delete the old class; finally, repeat these steps to the subclasses of the old class A.

Class addition is a simple primitive which is in fact exactly the same as class creation which adds a new class to the class lattice. Class deletion of class C is implemented in a way that all objects of class C will be destroyed, and then its direct superclass(es) will become the direct superclass(es) of its direct subclass(es). Afterwards, the definitions of all its subclasses need to be modified (similar to the class definition's changes) to reflect this change.

Some systems allow users to delete a class only if no object exists for that class. In DOOR, we assume that useful objects will be moved to other classes (since object migration contructs are provided) before their class as well as its objects are deleted. Note that we cannot automate this process by automatically migrating all objects of the class to be deleted to its superclass because that class might have more than one superclasses, and we cannot decide which superclass should be migrated.

Addition and deletion of an *isa* edge in a class lattice is supported by two primitive functions: ***isa-add*** and *isa-del*, respectively. ***isa-add(C, C')*** is simply defined as $\textit{inst}(C) \subset \textit{inst}(C')$ in the world, and *isa-del(C, C')* is defined as $\textit{inst}(C) \not\subset \textit{inst}(C')$ in the world unless there exists a class C'' such that $\textit{inst}(C) \subset \textit{inst}(C'') \subset \textit{inst}(C')$. Addition and deletion of an edge to a pair of classes will trigger the need of modifying the definitions of all the subclasses of one of them. This modification process, and hence the effect, is similar to the one in class definition's changes described above.

4 Related Work

The concept of a role was already defined in 1977 by Bachman and Daya [5] in the context of the network data modeling approach. Various role models and implementations in the context of databases have been proposed afterwards. These include Vision [19], ORM [16], and aspects [17], etc., and the most recent systems include Fibonacci [2, 3], Gottlob *et al.*'s role extension of Smalltalk [10]. Fibonacci is a new, strongly typed database language. Its objects simply consist 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. Instead of implementing a new language, Gottlob *et al.* [10] demonstrated the extension of Smalltalk for incorporating roles. Different from Fibonacci, they included multiple instantiation of roles, and the integration of class and role hierarchies. To some extent, both Fibonacci and Gottlob *et al.*'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. Different from ORM, aspects, and views, however, the roles attached to a class in both approaches can form their own "is_a" hierarchy.

Besides the work (mainly based on the concept of roles) mentioned above, there is some other work which relies on schema evolution, type evolution, or dynamic inheritance. SELF is a prototype-based language with a simple and uniform model [23] which includes the concept that an object's parent slots, like other data slots, may be assigned new values at run-time. An assignment to a parent slot effectively changes an object's inheritance at runtime. Consequently, the object can inherit different methods and exhibit different behavior. This ***dynamic inheritance*** allows part of an object's implementation to change at runtime. The Garnet system [15] includes a similar mechanism, also called dynamic inheritance but implemented differently, to effect wholesale changes in the implementation of an object's behavior. This feature has been used in Garnet to capture the significant changes in a user-interface object's behavior when switching between build mode and test mode in an application builder tool. Predicate classes (in a language called Cecil) [8] can emulate some of the functionality of dynamic inheritance as found in SELF and Garnet. Where a SELF program would have an assignable parent slot and a group of parent objects that could be swapped in and out of the parent slot, Cecil with predicate objects would have an assignable field and a group of predicate objects whose predicates test the value of the field. A related mechanism is the `become:` primitive in Smalltalk-80 [9]. This operation allows the identities of two objects to be swapped, and so is more than powerful enough to change the representation and implementation of an object.

Alternatively, substantial work which includes [4, 7, 13, 12, 20, 21, 26] has been done in the context of schema evolution and type evolution, however, the concept of role modeling for evolving objects was excluded.

5 Current Status and Future Work

The first prototype of DOOR, based on the meta-object protocol (MOP) [11] of the language Scheme, has been developed. The role playing mechanism, which is similar to the implementation of delegation in Clovers [22], has been implemented as a MOP. The reasons for implementing DOOR as MOP are as follows:

User-defined schema evolution operators Existing approaches to schema evolution provide only a fixed set of evolution operations. With MOP implementation of schema evolution, users can compose complex schema evolution operations from a set of primitive operations which allow any schema modification.

Expanding the DOOR data model During the prototyping phrase, the entire data model may need to be expanded or changed frequently. This can be done by expanding or modifying the corresponding DOOR components (i.e., the corresponding MOPs).

Changes in consistency definition The schema consistency definitions and even the model invariants might need to be adjusted or relaxed both by users or developers. This also increases the flexibility for some special applications.

Furthermore, 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 historial information) of object updates as well as class migration.

For example, in [24], roles are identified by unique identifiers. Assume we have two persons **p1** and **p2** playing the roles of employee **e1** and **e2** respectively, that share information as follows:

```
(p1, [name: s, address: a], {(e1, [salary: n,
  company: c]...)}, _),
(p2, [name:...], {(e2, [manager: e1,...]...)},
  _).
```

The advantage of referring a manager of **e2** (or p2) to role **e1** instead of object **p1** is that the manager of **p2** refers to the right person even someone else (say **p1'**), instead of **p1**, is playing the manager role of **p2**.

```
(p1', [name: s, address: a], {(e1, [salary: n,
  company: c]...)}, _),
(p2, [name:...], {(e2, [manager: e1,...]...)},
  _).
```

However, transferring of a role identifier between two different objects is not possible in the identification scheme described in [24]: e.g., p is regarded as playing a different employee role only if p resigns and later is retired. Even this is possible, it causes problem when the value c is changed to another company, say c'. Then p1 is no longer the manager of p2. However, since the manager attribute of e2 refers to e1 and e1's player is still p1.

Currently, the prototype supports both transient objects and persistent objects. However, persistent objects are stored in flat files. This is no longer adequate and efficient for serious implementation. Therefore, our ongoing work is to implement a persistent object store for DOOR. The main difficulty is to design the persistent store such that it supports the dynamic updates well. As most of the work on persistent stores is focused on the recovery, scalability and access performance, not much previous work has been found on this particular issue.

6 Conclusions

We have presented in this paper a dynamic object-oriented database programming language with role extension, called DOOR, which incorporates both roles and schema evolution constructs. With DOOR, a role can be used to model evolving objects and schema evolution can be used to increase the flexibility and maintainability of the design of a dynamic and evolving system. Important issues which help to integrate these two approaches were discussed in detail. They include the reference problems caused by object deletion and role change, issues about objects which change type and objects which change role, the change of class definition, and the change of class lattice.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases.* Addison-Wesley, 1995.

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

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

[4] E. Amiel, M.-J. Bellosta, E. Dujardin, and E. Simon. Supporting exceptions to behavioral schema consistency to ease schema evolution in oodbms. In *Proceedings of the 20th VLDB Conference*, pages 108–119, Chile, 1994.

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

[6] J. Banerjee, H.J. Kim, W. Kim, and H.F. Korth. Schema evolution in object-oriented persistent databases. In *Proceedings of the 6th Advanced Database Symposium*, August 1986.

[7] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*, 1987.

[8] G. Chambers. Predicate classes. In *European Conference on Object-Oriented Programming (ECOOP93)*. Springer, 1993.

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

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

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

[12] W. Kim, N. Ballou, H.T. Chou, J.F. Garza, and D. Woelk. Features of the orion object-oriented database system. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 251–282. ACM Press, Addison-Wesley, 1989.

[13] W. Kim and H.-T. Chou. Versions of schema for object-oriented databases. In *Proceedings of VLDB Conference*, pages 148–159, Los Angeles, 1988.

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

[15] B.A. Myers, D.A. Giuse, and B.V. Zanden. Declarative programming in a prototype-instance system: Object-oriented programming without writing methods. In *OOPSLA'92 Conference Proceedings*. Published as SIGPLAN Notices 27(10), October 1992.

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

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

[18] E.A. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proceedings of the 18th VLDB Conference*, Vancouver, Canada, 1992.

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

[20] A.H. Skarra and S.B. Zdonik. The management of changing types in an object-oriented database. In *OOPSLA '86 Proceedings*, September 1986.

[21] B. Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In *ECOOP/OOPSLA '90 Proceedings*, October 1990.

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

[23] D. Ungar and R.B. Smith. Self: The power of simplicity. In *OOPSLA'87 Conference Proceedings*. Published as SIGPLAN Notices 22(12), October 1987. Also published in Lisp and Symbolic Computation 4(3), Kluwer Academic Publishers, Jun 1991.

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

[25] R.J. Wieringa, W. de Jonge, and P.A. Spruit. Roles and dynamic subclasses: a modal logic approach. In *ECOOP '94 Proceedings*, 1994.

[26] S.B. Zdonik. Object-oriented type evolution. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 277–288. ACM Press, Addison-Wesley, 1990.