

# A Equivalent Object-Oriented Schema Evolution Approach

## Using the Path-Independence Language

Jiaheng Lu Chuanliang Dong Weiwen Dong  
Department of Computer Science and Engineering, Shanghai JiaoTong University  
Shanghai China 200030  
jiahenglu@263.net cldong@mail1.sjtu.edu.cn wwdong@mail.sjtu.edu.cn

### Abstract

*Software legacy problem caused by schema evolution in an Object-Oriented database is a very important research issue. This paper proposes a method of equivalent schema evolution based on a path-independence (PI) language. The PI language, which has been adopted in some systems, raises the level of abstraction for behavioral schema design and is almost independence with the specific schema digraph. We develop the PI language and advocate the equivalent schema evolution method which not only resolves software reuse problem in schema evolution, but supports schema version mechanism, which is an advanced feature in a database system.*

**Keywords:**object-oriented database, schema evolution, software reuse , schema version

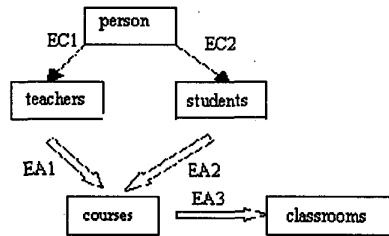
### 1. Introduction

Schema evolution is an important issue in object-oriented database(OODB) research, because typical OODB application areas such as CAD/CAM and multimedia information system require frequent schema changes to fit in with the needs of the designs. However, after the data models have been changed, existing application programs that rely on the previous data models have to be completely updated. The update process caused by a small schema change is likely to be long and tedious. To ease this problem, many researchers advocate a lot of novel methods .Up to now, there have been two main research directions for schema evolution in OODB system. One is to effectively preserve old schemas for continue support of existing programs running on the schema database.<sup>[1],[3],[5]</sup> The other direction is to improve the design of database method and query specification languages such that software programs written in these advanced language have higher adaptation and seamless to the schema evolution.<sup>[4],[6]</sup> The paper[3] falls into the first direction. It advocates the transparent schema-evolution (TSE) system, which integrating schema evolution with view facilities. So the system can avoid existing program's update. However, in their system, any tiny schema modification will lead to miscellaneous schema change and so many versions merging is still too complicate. The paper[4] belong to the second direction . It presents a polymorphic reuse mechanism in

schema evolution and the use of propagation patterns. Their main innovations are in making possible the derivation of operational semantics from structural specification. However, there are two problems which are not considered in the paper. First, in the light of the design requirement, how to modify the schema can guarantee the complete software reuse — the software without any change. Second, How to support the schema version mechanism in the propagation patterns

In contrast to those prior work, we integrate the schema evolution into the propagation pattern. Our schema evolution mechanism use the path-independence language and equivalent schema evolution(ESE) algorithm to support software complete reuse. In our system, unlike in the system[4], the propagation pattern or the propagation pattern refinement fits only passively in with the change of schema. Instead our ESE can consider the previous software and guarantee their reuse. Another advantage over the conventional approaches is our system can support the schema version mechanism.

The remainder of the paper is organized as follows. In section2, we introduce the path-independence language and its formal semantics. They are theoretical basis in the following discussion. We discuss the concept of the equivalent schema evolution in section 3. The definitions to decide two schemas are equivalent are formally studied as well. In section 4, we present an equivalent schema evolution algorithm and example for illustration and section 5 concludes this paper.



**Figure 1. An example of schema digraph**

element of E into the ordered pair of the element of V.

Example1. Consider Fig1  $U = \{person, teachers, students\}$ ;  $EC = \{EC1, EC2\}$ ,  $EA = \{EA1, EA2, EA3\}$   $\Psi(EC1) = \{(person, teachers)\}$   $\Psi(EA1) = \{(teachers, courses)\}$ ,

Definition 2 (reachable:  $\rightarrow$ ) In a schema digraph, a vertex  $u \in V$ , is reachable for vertex  $v (v \in V)$ , denoted by  $u \rightarrow v$ , if and only if (iff) one of the following conditions are satisfied,  
 (1)  $u = v$  (2)  $(u, v) \in EA$  (3)  $\exists u' \in v, u' \neq u$  and  $u' \neq v$ , s.t.  $u \rightarrow u', u' \rightarrow v$ .

Note that " $\rightarrow$ " is reflexive and transitive.

Definition 3 (propagation directive) In a schema digraph, the propagation directive D is a triple  $(F, PC, T)$ , where

- F denotes a noempty set of source vertex and  $F \subseteq V$ ;
- PC denotes propagation constraints and  $PC \subseteq EA$ , PC is a set of through edges called restriction constraints.

## 2. The basic concepts of path-independence (PI) language

Definition 1 (schema digraph) A schema digraph SD is a triple  $(V, E, \Psi)$ . V is a set of vertices, every vertex is a class. E is a set of edge,  $E = EC \cup EA$ , EC represents the object reference relationship. EA represents the inheritance relationship.  $\Psi$  maps each

- T denotes a noempty set of target vertex and  $T \subseteq V$ .

Example 2 In figure 1 we consider the following propagation directive D

From students Though (students ,courses),(courses,classrooms) To classrooms

Definition 4 (path-independence language) Let G be a schema digraph. A path-independence language  $\alpha$  over G is defined by a triple (M,PD,MA)

- M is a method interface defined by a triple(u,mn,L)

----- u  $\in \{V \cup \{\text{void}\}\}$  denotes the output type which is either a class vertex or the key "void", indicating an empty result type.

-----mn is the method name

-----L is a list of parameters.

- PD denotes a noempty set of propagation directive D over G

- MA is a set of method annotations. A method annotation consists of a set of prefix or suffix components denotes as  $(w,fg_{pre})$  or  $(w,fg_{suf})$ , where w is a class vertex in V,  $fg_{pre}$  and  $fg_{suf}$  each denotes code fragment; describing the user define method implementation. Its prefix fragments are executed before its suffix fragments.

Example 3 In figure 1,suppose we want to have a method which prints the classroom No. ,where a student have lessons. We may define the method by writing the path-independence (PI) language.

```

PI void print-classroom ()
MA classrooms
void students::print-classroom()
{int studenti;
for(studenti=1, studenti<=10; studenti++)
student_course[studenti].print-classroom()
}
void classrooms::print-classroom()
{print(classroomNO);}
void courses::print-classroom()
{course_room.print-classroom();}

```

Figure 2. C++codes for Figure 1

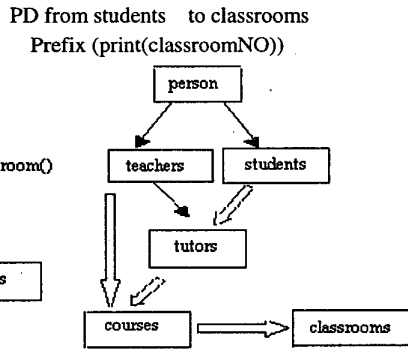


Figure 3. The modified schema digraph

Writing a PI language does not require knowledge of the detailed data structure. One obvious benefit of this feature is to allow reuse of PI language at hand for several similar data structures and thus to increase the adaptation of operation specification against future schema changes. For instance, for writing the PI method "print-classroom", the minimal knowledge we need to know is students, classrooms. They are critical information for defining this function. Suppose now we need to modify the schema of fig 1 by adding tutors class(see Fig 3)The schema modification as such needs no reprogramming of the method "print-classroom". Although the path from student to classroom is changed in the modified schema, because all the critical information for specifying "print-classroom" is unchanged and included in the modified schema. Thus the above PI language "print -classroom" can still be used as a valid

```

void students::print-classroom()
{ tutor.print-classroom();}

void tutors::print-classroom()
{ int tutori;
  for (tutori=1; tutori<=10; tutori++)
  tutor_course[tutori].print-classroom();
}

void courses::print-classroom()
{ course-room.print-classroom();}

void classrooms::print-classroom()
{ print(classroomNO);}

```

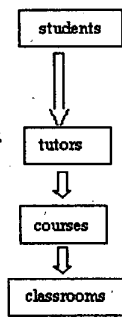


Figure 4. C++ code for Figure 3.

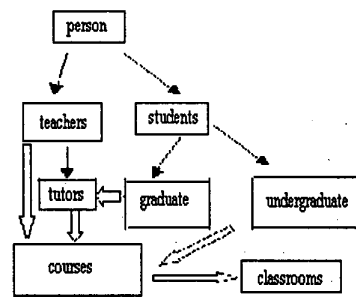


Figure 5. The schema digraph for example 4

and meaningful PI language to this modified schema. The PI program can automatically be translated into code written in object-oriented program language (for example C++) through so-called compiler. We develop a compiler called JTESE. The C++ codes attached to the classes in Fig 2, which we obtain by running the JTESE system, are the running result of Fig 1 and the code in Fig 4 is the running result of Fig 3.

### 3. Equivalent schema evolution

In some cases, when a schema modification incurs a change in the paths of existing methods or queries or when a schema modification changes the properties of objects, manually reprogramming of existing methods or queries can be avoided by structural derivation of the path-independence language. Unfortunately, when a schema modification changes the minimal knowledge required for specifying a method or query, the reprogramming can not be avoided completely.

**Example 4** Suppose in new term, students should be classified as graduate students and undergraduate students. A undergraduate student can choose his lessons by himself, but a graduate student's lessons must be decided by his(her) tutor. So the schema should be modified into Fig 5. In example 3 program  $\alpha$ , there are two paths in new schema: One is undergraduate\_students  $\rightarrow$  courses  $\rightarrow$  classrooms, the other is graduate\_students  $\rightarrow$  tutors  $\rightarrow$  courses  $\rightarrow$  classrooms. So the previous program cannot be reused. The reason that reprogramming cannot be avoided is that the schema evolution in Fig 5 is not equivalent schema evolution.

**Definition 5 (equivalent schema)**  $G1, G2$  are two schema digraphs.  $\alpha$  is a PI program. If  $\alpha$  over  $G1$  and  $G2$  accomplishes the same function then we call  $G1$  and  $G2$  concerning  $\alpha$  the equivalent schema digraphs, denoted by  $G1(\alpha) \cong G2(\alpha)$ .

Obviously, the schema of Fig 1 is equivalent schema with one of Fig3.

**Theorem 1.**  $G1$  and  $G2$  concerning  $\alpha$  are equivalent schema digraph, if and only if (iff) the following two conditions are satisfied

- (i) In  $\alpha$  over  $G1$  and  $G2$ ,  $F \rightarrow T$  and the path is alone.
- (ii) In  $\alpha$ , the code fragments of  $fg_{pre}$  and  $fg_{inf}$  over  $G1$  and  $G2$  both accomplish the same

function.(Intuitive Proof is omitted)

See example 4 again. According to the concept of equivalent schema evolution, we create a equivalent schema of Fig 3 which satisfies the demand of example 4. We create a new `all_student` class which becomes the superclass of `new_students` class and `students` class. See Figure 6.

According to theorem 1, it is easy to prove that the schema of Fig 1 concerning  $\alpha$  is the equivalent schema with one of Fig 6, that is:  $\text{Fig 6}(\alpha_{\text{example3}}) \cong \text{Fig 1}(\alpha_{\text{example3}})$ .

From the above discussion, we come to a conclusion that how to modify the schema is important issue for software reuse. The change of Fig 6 and Fig 5 both satisfy the demands of Example 4. But Fig 6 is a equivalent schema evolution with Fig 1 and effectively guarantees software reuse. On the contrary, Fig 5 is not a good schema change, because it needs to modify program  $\alpha$ .

In section 4, we represent the equivalent schema evolution algorithm.

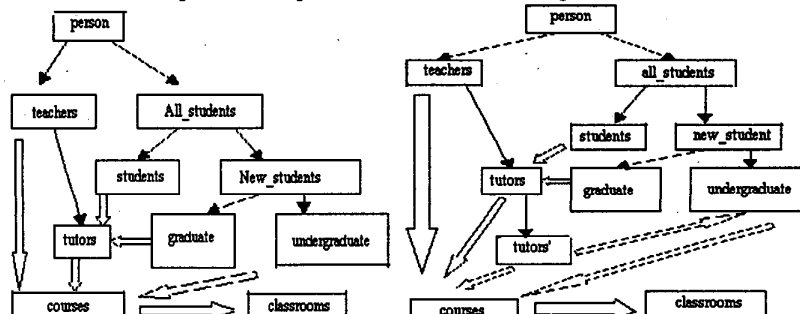


Figure 6. A new schema digraph for example 4

Figure 7. The schema digraph for example 5

#### 4. Equivalent schema evolution algorithms

One of the first object-oriented schema change taxonomies has been proposed by Banejee et al[2], for the Orion data model. To demonstrate the feasibility of the equivalent schema evolution, it is thus important to show that our approach can realize all the schema change operations supported by Orion. But due to space restriction, we cannot cover all the Orion taxonomy. In summary, we give an algorithm to realize adding a new attribute  $x$  to a class  $C$ . In fact, we have obtained all other algorithms that the Orion supports.

Algorithm 1: Add a new attribute  $x$  to a class  $C$ .

Input: the schema  $G$ , new attribute  $x$ , class  $C$ , the PI program  $\alpha$ .

Output: the schema  $G'$  which satisfy  $G'(\alpha) \cong G(\alpha)$  and there is a new attribute  $x$  in  $C$

Method: execute the following steps:

(1) If new attribute  $x$  is added to class  $C$  directly, then in program  $\alpha$  can it be satisfied that  $F$  is reachable for  $T$  and the path is alone?

If yes, then go to 3. If no, go to 2.

(2) Create a class  $C'$ , which is subclass of  $C$  and have the attribute  $x$ . Go to 4.

(3) Add the attribute  $x$  to class  $C$  directly.

(4) Output the schema  $G'$  constructed.

Example 5. Suppose now undergraduate students are going to fulfil their graduation projects. Each tutor should teach several students. So we add a attribute of undergraduate students to tutor class.

By algorithm 1. We generate Fig7 concerning a the equivalent schema with Fig 6.

Notice that the schema evolution in the example support not only software reuse, but also the schema version mechanism. System provides distinct view through distinct schema version. For example in Fig 7, students, tutors, courses, classrooms can be organized as the view of programmers who manage the old students class, while tutors', undergraduate students, courses and classrooms can be organized as the view of people who manage new undergraduate students class. Both old and new views are independence. A person who use the old view may well does not know the global schema has been changed. Because he can not perceive any change in his view and programs. Up to now, regardless of OODB or relation database, little ones can support schema version mechanism, while the algorithm we propose all can provide schema version mechanism.

## 5. Concluding Remarks

As shown by the examples in the previous section, the PI language are currently well-supported in a system called JTESE. In order to give a road map of possible implementation considerations of our approach, we would like to give a very brief illustration on the JTESE. The JTESE has two functions, one is translating the PI language into C++ code, the other is equivalent schema evolution system. The compiler use the schema digraph and the PI language to generate C++ code. The ESE system modifies the schema according to the ESE algorithms in accordance with the demand of the schema evolution.

**Acknowledgment:** Work has been supported by the natural 863 high science project under research grant 863-306-ZD02-02-9. We are grateful for some support from Leihong Jiang.

## Reference

- (1) Randal J.Peters et al "Axiomatization of Dynamic Schema evolution in objectbases" IEEE 11th International Conference on Data Engineering 1995 pages 156—164
- (2) Banerjee, W.Kim, H.J.Kim and H.F.Korth "Semantics and Implementation of Schema Evolution in Object-Oriented Databases" SIGMOD pages 311—322 `1987
- (3) Young-Gook Ra et al "A Transparent Schema-Evolution System Based on Object-Oriented View Technology" IEEE Transactions on knowledge and Data Engineering 1997 vol.9 N).4 pages 600—624
- (4) Ling liu et al "the Role of Polymorphic Reuse Mechanism in schema Evolution in an Object-Oriented Database" IEEE Transaction on knowledge and Data Engineering 1997 vol.9 NO.1 pages 50—67
- (5) M.Tresh and M.H.Scholl "Schema Transformation without Database Reorganization" SIGMOD Record pages 21—27 1993
- (6) J.Palsberg K.Lieberheer and C.Xiao "Efficient Implementation of Adaptive Software" ACM Trans Programming Languages and Systems Vol.17 NO.2 pages 264—292 Mar 1995