基于路径无关语言的等价模式演化策略*

董传良，陆嘉恒，杨 虹，董玮文

(上海交通大学 信息统计中心，上海 200030)

E-mail: cldong@mail1.sjtu.edu.cn

http://www.sjtu.edu.cn

摘要：面向对象数据库的许多应用环境需要频繁的模式演化，但模式演化以后，基于先前模式的应用程序因此而不得不修改或重编，这就造成了巨大的软件浪费.提出了基于路径无关语言的等价模式演化方案来解决这个问题.首先，路径无关语言是一种面向对象数据库的编程语言，它能使程序脱离对细节数据模式的导航，对模式演化具有较强的适应性.而等价模式演化是一种新的模式演化方案，它能保证用路径无关语言编写的应用程序在模式演化以后无须修改而完全重用.此外，在实现等价模式演化的系统中，为了减少演化开销以及不增加用户的额外编程负担，提出了虚拟关系机制和对象演化技术.

关键词：可适应性软件的设计与发展；模式演化；面向对象数据库；软件重用；虚拟关系

中图法分类号：TP311 **文献标识码**：A

模式演化历来是面向对象数据库领域研究的一个重要课题.在工程设计应用中，随着设计过程的进展，需要动态地改变数据模式，这就向面向对象数据库提出了模式演化的需求.但模式改变之后，以前编写的大量程序由于依赖于具体的数据模式而不得不进行全面修改，这就造成了极大的软件浪费.为了解决这个矛盾，许多研究人员投入到OODB(object-oriented database)模式演化后软件重用的研究中去，提出了许多创新性的方案.归纳起来，至今为止主要有两种解决途径.一种解决途径是使模式演化改变考虑到现有的应用程序，使两者相互适应，相互集成.例如，文献[1~3]采用视图技术能够实现模式演化后的软件重用，但一个简单的模式演化需求往往需要生成一连串的子类，系统开销较大，且众多视图的合并也略嫌繁杂.另一种解决途径是找到一种非导航式的数据库编程语言，使之能够尽可能不依赖于具体的数据模式，而使得在大多数情况下，模式演化与应用程序无关.文献[4,5]提出了传播模式(propagation pattern)语言.该语言在书写上与模式图中具体的导航路径无关，因此对模式演化具有一定的适应性，体现出软件的多态重用机制(software polymorphicy reuse mechanism).但是，这种语言能否重用同模式演化的类型有关，当模式演化破坏了propagation pattern所需的最少信息时，在模式演化后，用这种语言编写的程序也不能够重用.Liu Ling[4]提出了模式重制(pattern refinement)来对已存在的传播模式程序进行修正，但这种修正在面对诸如删除一条关键边或者关键结点之类的模式演化需求时是失效的.为了很好地解决此类问题，本文提出了基于路径无关语言(path-independence program)的等价模式演化策略.路径无关语言(简称PI语言)也是一种非导航的面向对象数据库编程语言，而等价模式演化机制能简洁、有效地保证在模式演化之后，以往的PI语言完全重用，更好地体现了PI语言的多态性与可重用性.

* 收稿日期：1999-08-31；修改日期：2000-06-19

基金项目：国家863高科技发展计划资助项目(863-306-ZD02-02-9)

作者简介：董传良(1950-)，男，上海人，高级工程师，主要研究领域为面向对象数据库，分布对象计算；陆嘉恒(1975-)，男，上海人，工程师，主要研究领域为面向对象数据库；杨虹(1968-)，女，安徽芜湖人，工程师，主要研究领域为面向对象数据库；董玮文(1946-)，女，上海人，副教授，主要研究领域为数据库系统.

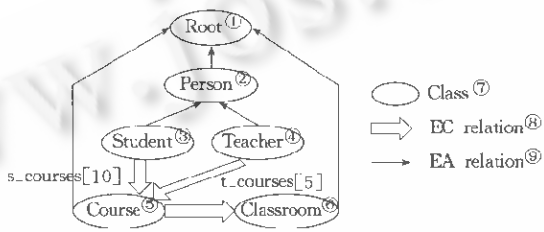
1 路径无关语言

1.1 基本概念

一个面向对象数据库的模式可用有向图来表示.下面我们给出它的形式化定义.

定义 1. 一个模式有向图是一个三元组 $G=(V,E,L)$.其中 V 是顶点集合,每个顶点表示一个类;E 是边集,$E=EA\cup EC$,EA 是一个二元关系 $V\times V$,表示类的继承关系,EC 是一个三元关系 $V\times L\times V$,表示类之间的引用关系;L 是一个字符串集合,表示该类的一个变量.

例 1:如图 1 所示,$V=${Root,Person,Student,Teacher,Course,Classroom},$EA=${(Person,Root),(Course,Root),(Classroom,Root),(Student,Person),(Teacher,Person)},$EC=$ {(Student,s_courses[10],Course),(Teacher,t_courses[5],Course),(Course,classroom,Classroom)}.



①根,②人,③学生,④教师,⑤课程,⑥教室,⑦类,⑧EC边,⑨EA边.

Fig.1 A schema digraph

图1 一个模式有向图

定义 2. 在一个模式有向图中,若某结点 u 可到达结点 v,记为 $u\rightarrow v$,当且仅当满足如下任意一个条件:(1) $u=v$;(2) $\exists\, l\in L$ 满足 $(u,l,v)\in EC$;(3) $\exists\, u'\in V$,$u'\neq u$ 且 $u'\neq v$,满足 $u\rightarrow u'$,$u'\rightarrow v$. "$\rightarrow$"关系是边集 EC 上的一个自反与传递关系.在图 1 中,Student→Course,Course→Classroom,所以 Student→Classroom.

定义 3. 一个模式有向图中,若结点 u 可唯一到达结点 v,记为 $u\rightarrow^1 v$,当且仅当同时满足以下两个条件:(1) $u\rightarrow v$;(2) 从 u 到达 v 的路径唯一,即 $(ul_1v_1)(v_1l_2v_2)\ldots(v_nlv)$ 是唯一的.

在图 1 中,Student$\rightarrow^1$Classroom.如果 u 不是唯一到达 v,则记为 $u!\rightarrow^1 v$.

定义 4. 在一个模式有向图中,传播路径 D 是一个三元组 $D=(F,PC,T)$,$F,T\in V$ 且 $F,T\neq\varnothing$,F 表示传播的起点,T 表示传播的终点,PC 是传播约束.$PC=(TH,B)$,其中

- TH(though)表示传播中必须通过的 EC 边.
- B(bypass)表示传播中禁止通过的 EC 边.

例 2:在图 1 的模式中,我们定义一个传播路径 D Source Student Through(Student,s_courses[10],Course)Destination Classroom,表示了 Student→Course→Classroom.其中起点 F 是 Student,终点 T 是 Classroom.在 PC 约束中,TH 是 EC 边(Student,s_courses[10],Course),B 为空.

1.2 一个实例

例 3:在图 1 中,假设我们需要编写一段程序去打印学生的上课教室,用路径无关语言书写如下:

PI print_classroomNo

D Source Student Destination Classroom

MA Classroom(print(classroomNo))

第1行PI print_classroomNo表示方法名,*D* Source Student Destination Classroom是传播路径,而MA Classroom(print(classromNo))表示具体的方法实现,Classroom是类名,(print(classroomNo)是在该类上的具体的C++语句.上述方法可用一个编译器编译成C++语言,然后嵌入数据库中执行.我们开发了一个JTESE/C++编译器.如图2所示为上述方法用编译器执行的结果.从图2可见,PI语言是一种非导航式的查询语言,在一般的OODB中,我们为完成以上查询,必须书写具体路径s_courses[i].classroom.print(classroomNo).这就束缚了应用程序对模式演化的适应性,见例4.

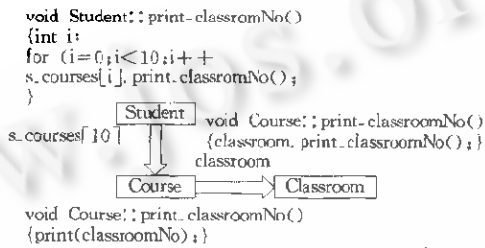


Fig. 2 Compiling results of Fig. 1 for method 'print_classroomNo'
图2 "print_classroomNo"方法在图1上的编译结果

例4:假定我们在图1中添加Tutor作为Teacher的子类,Student上课的课程由他(她)的Tutor所上课程决定.由此图1演化为图3,用编译器执行的结果如图4所示.在图3中,例3的路径无关语言仍然可以重用,因为在图3中,实际的传播路径变成Student→Tutor→Course→Classroom,但例3中的传播路径D由于没有指明具体的导航路径,在图3的模式中仍然可以使用.但上面传统的书写方法却不能复用了,因为针对图3的查询,按传统的书写方法应改为

tutor.t_courses[i].classroom.print(classroomNo).

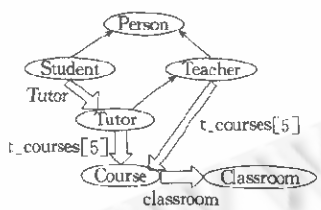


Fig. 3 An evolved schema of Fig. 1
图3 图1的一个演化模式

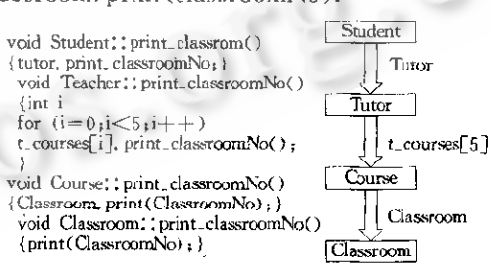


Fig. 4 Compiling results of Fig. 3 for method 'print_classroomNo'
图4 方法"print_classroomNo"在图3中的编译结果

由此可见,路径无关语言与传统的编程语言相比,具有明显的优越性,对模式演化有较强的适应性,从而可以最大限度地避免模式演化后的软件重编.

1.3 形式化定义

定义5(路径无关语言的形式化定义).在一个模式有向图$G=(V,E,L)$中,路径无关语言α是一个三元组(M,D,MA).

- M是一个方法名.
- D是一个传播路径.

· MA 是一个方法的说明集合,可表示为(w,C). 其中 w 是一个结点,$w\in V$;C 是实际的面向对象编程语言,定义了在 w 结点上方法的实现.

定义 6(路径无关语言的重用性). 如果 $\alpha=(M,D,MA)$是定义在模式有向图 G 中的一个路径无关语言,则在 α 的传播路径 D 中,若源结点 F 唯一到达目标结点 T,则我们称 α 是对 G 兼容的,记为 $G:F(\alpha)\rightarrow^{1}T(\alpha)$.

由上述定义可以得到,"print_classroomNo"方法对图 1 和图 3 的模式都是兼容的.

定义 7(路径无关语言的功能不变性). 若 $G1$ 与 $G2$ 是两个模式有向图,$G2$ 由 $G1$ 演化而来,α 是一个路径无关方法. 我们定义 α 对于模式 $G1$ 与 $G2$ 是功能不变的,当且仅当同时满足以下条件:

(1) $\forall w$,其中$(w,C)\in MA(\alpha)$ 满足 $w\in G1(V)$而且 $w\in G2(V)$;

(2) $\forall C$,其中$(w,C)\in MA(\alpha)$,满足 C 在 $G1$,$G2$ 模式中有相同的功能.

以上两个条件保证 $G1$ 演化成 $G2$ 后,同一个路径无关方法的 MA 部分在两个模式有向图能实现相同的功能. 其中条件(1)表明方法的说明集合中的类必须在模式有向图中存在,如"print_classroom"方法中的 Classroom 类必须在图 1 和图 3 中存在. 条件(2)表明,具体在类上执行的 C++语句应该完成同一种功能. 如 print(classroomNo)在图 1 与图 3 模式中都会打印教室号. 在下面的讨论中,我们将证明上述的定义 6 和定义 7 是等价模式演化的两个必要条件.

2 等价模式演化

虽然 PI 语言对模式演化具有很强的适应性,但遗憾的是,并不是对所有的模式变更要求,路径无关语言都能够重用. 请见下例:

例 5:假设在图 1 的模式中,我们为 Student 类加上属性 class_advisor:Teacher 变量. 见图 5 模式. 同图 1 相比,对于例 3 的 PI 语言,图 5 存在两条从 Stuent 到 Classroom 的路径,一条是 Student→Course→Classroom,另一条是 Student→Teacher→Course→Classroom,编译器无法决定该沿哪一条路径去传播. 因此,图 5 的模式演化导致了例 3 的程序不能重用.

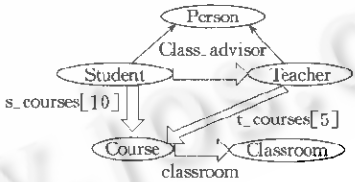


Fig. 5 An unequivalent schema of Fig. 1 for method 'print_classroomNo'
图5 与图1模式关于"print_classroomNo"方法的非等价模式

为了解决该问题,本文提出了等价模式演化算法. 即不直接在 Student 类中加上这一条边 class_advisor,而是采用等价模式演化算法,计算出一个新的演化方案,使之既满足原来的演化要求,又能保证所有的应用程序重用. 下面我们来详细阐述这一思想.

定义 8(等价模式演化). $G1$ 是一个模式有向图,$G2$ 是由 $G1$ 演化而来. α 是路径无关语言,若 α 对于 $G1$ 和 $G2$ 得到相同的查询结果或实现相同的功能,则称 $G2$ 是 $G1$ 的关于 α 的等价模式演化,记为 $G1(\alpha)\cong G2(\alpha)$.

由定义 5 可得,图 3 是图 1 关于"print_classroomNo"方法的等价模式演化.

下面给出等价模式演化的两个必要条件.

定理 1. $G2$ 是 $G1$ 关于路径无关语言 α 的等价模式演化,仅当同时满足以下条件:

(1) α 同时兼容于 $G1$ 与 $G2$,即 $G1:F(\alpha)\rightarrow{}^{1}T(\alpha)$,$G2:F(\alpha)\rightarrow{}^{1}T(\alpha)$.

(2) α 对 $G2$ 是 $G1$ 的功能不变.

证明:$G2$ 是 $G1$ 关于路径无关语言 α 的等价模式演化,则由定义 8 可得:对于 $G1$ 和 $G2$ 应得到相同的查询结果或者实现相同的功能.这样,首先 α 必须能够适用于 $G1$ 与 $G2$,即 α 同时兼容于 $G1$ 与 $G2$,即在 $G1$ 与 $G2$ 中,F 均唯一到达 T.这样,必要条件(1)就必须被满足.其次,若要路径无关语言 α 在 $G1$ 和 $G2$ 上都能得到相同的查询结果或者完成相同的功能,则由定义 5 可知:路径无关语言 α 是一个三元组 (M,D,MA),其中 MA 是唯一决定 α 的具体功能的集合,而集合 MA 又由 w,C 共同构成,若要 MA 相同,则 w,C 都要相同,因此,有关定义 7 中的两个条件都要满足,于是 α 对于 $G1$ 与 $G2$ 就是功能不变的.这样,必要条件(2)也要被满足.故定理 1 得证. □

图 3 是图 1 关于"print_classroomNo"方法的等价模式演化.不难验证,上述定理 1 中的两个必要条件在上述两个模式有向图中都是满足的.进一步值得指出的是,之所以上述两个条件仅是必要的,而非充分的,是因为存在诸如例 6 的特例.

例 6:在图 6(a)中,student 类的 like_course 变量表示学生喜欢上的课程,而图 6(b)中的 dislike_course 变量表示学生不喜欢上的课程.如果把路径无关语言"print classroomNo"应用到这两个模式中,则图 6(a)图会打印出学生喜欢上的课程的教室,而图 6(b)会打印出学生不喜欢上的课程的教室号.这里显然就不是等价模式演化,但以上演化过程是符合定理(1)的两个必要条件的(因为首先,在这两个模式图中,student 均唯一到达 Classroom,其次,Classroom(print(classroomNo))一句在图 6(a)与(b)中均执行打印教室号的功能).因此,我们说定理(1)的两个条件是非充分的.造成这种结果是因为在定理(1)中仅规定了传播路径上的唯一性而没有规定传播语义上的一致性.如在图 6 中,由 Student 类传播到 Course 类,虽然路径都唯一,但在(a)(b)两个模式中,语义是绝对不同的.因此,要想保证等价模式演化,就必须考虑到传播语义的一致性.

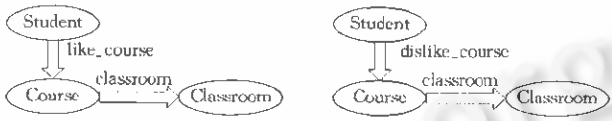


Fig. 6 An unequivalent schema evolution which is satisfied with the prerequisite of theorem 1
图6 满足定理1的必要条件而非等价模式演化的例子

3 形式化的等价模式演化算法

面向对象数据库的模式演化分类是由 Banerjee 等人在 Orion[6]数据模型上提出来的较为经典.本文以此作为模式演化的分类标准.

3.1 模式演化的公理模型

本文采用了 Peters[7]的形式化模式演化公理模式.该模式已被证明具有完整性和一致性.本文的模型与文献[7]相比,有以下区别:(1) 我们没有采用文献[7]中的基类(base class),因为在模式有向图中不存在是所有结点的子类这样一个结点.(2) 本模型没有使用标记 $Ne(t)$ 与 $Pe(T)$,因此,适当地修正了文献[7]中表 2 的公理描述,但这并不影响该公理模型的一致性和完整性.有关证明详见文献[7],本文略.

在表 1 中,$H(t)\cap N(t)=\varnothing$,$I(t)=H(t)\cup N(t)$.

表 2 描述了如何利用 $P(t)$ 与 $N(t)$ 计算出 $PL(t)$,$I(t)$,$H(t)$.

Table 1 Notation of the axiomatic model

表 1 模型使用的记号说明

G	Whole schema digraph①
Root	Root②
s	s is a class in G③
$P(t)$	Immediate superclass of class t④
$PL(t)$	All superclassses of class t, including t⑤
$N(t)$	Native variables and methods of class t⑥
$H(t)$	Inherited variables and methods of class t⑦
$I(t)$	All variables and methods of class t, including $N(t)$ and $H(t)$⑧
$\alpha_x(f(x),T)$	Results applying the unary function f to $x(\forall\ x\in T)$⑨

①整个模式有向图,②根结点,③G 中的一个类,s 是类名,④类 t 的所有直接超类,⑤类 t 的所有超类,包括类 t 本身,⑥类 t 自身定义的所有变量和方法,⑦类 t 的继承变量和方法,⑧类 t 的所有变量和方法,包括 $N(t)$ 与 $H(t)$,⑨用一元函数 f 作用于 $x(\forall\ x\subset T)$ 后产生的结果.

Table 2

表 2

Axiom①	Description②
Closure③	$\forall\ t\in G, P(t)\subseteq G$
Acyclicity④	$\forall\ t\in G, t\notin \bigcup\alpha_x(PL(x),P(t))$
Rootedness⑤	$\exists\ Root\in G, \forall\ t\in G\mid((Root\in PL(t))\wedge(P(Root)=\varnothing))$
Superclass⑥$(PL(t))$	$\forall\ t\in G, PL(t)=\bigcup\alpha_x(PL(x),P(t))\cup\{t\}$
Interface⑦$(I(t))$	$\forall\ t\in G, I(t)=N(t)\cup H(t)$
Inheritance⑧$(H(t))$	$\forall\ t\in G, H(t)=\bigcup\alpha_x(I(x),P(t))$

①公理,②描述,③闭包,④无环,⑤有根,⑥超类,⑦接口,⑧继承.

3.2 等价模式演化算法

3.2.1 为类 t 加上变量 x

3.2.1.1 算 法

$N(t)=N(t)+\{x\}$;

$(\forall\ s\in G)\wedge(t\in PL(s))$ recompute $H(s)$,$I(s)$; //根据表 2 中的公理 5 和公理 6 重新计算 $H(s)$和 $I(s)$

while $(\exists\ tm, t\in PL(tm))\wedge(\forall\ a\in$ PI program$)\wedge$(tm causes $G:F(\alpha)!\xrightarrow{1}T(\alpha)$)

do

{create new_tm, $N(new\_tm)=N(tm)$;

$(\forall\ s\in G)\wedge(tm\in P(s))$ $\{P(s)=P(s)+\{new\_tm\};\ P(s)=P(s)-\{tm\};\}$ //把所有 tm 的直接超类转给 New_tm

$N(tm)=N(tm)+H(tm)$; //把 tm 的继承属性和方法改为自身定义的属性与方法,即 $N(tm)=I(tm)$

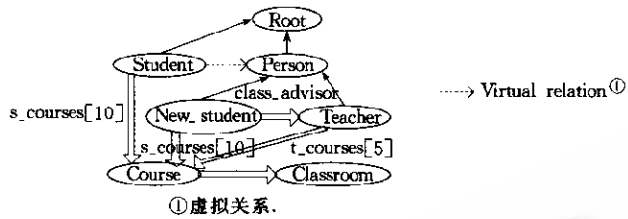
$N(tm)=N(tm)-\{x\}$; //tm 不具有新的变量 x

$P(tm)=\{Root\}$; //tm 只有"根"这一个超类

}

首先,直接将变量 x 加在类 t 上,t 的子类被通知修正它们的数据成员.这有可能导致在某些 PI 方法中形成了第 2 条 F 到 T 的路径.例如在图 5 中,直接将 *class_advisor* 加在图 1 上,形成了两条从 Student 到 Classroom 的路径.我们称导致路径无关语言不能兼容于演化后模式图的类为 Troublemaker,简称 tm.例如,图 5 中的 Student 类就是一个 Troublemaker.在上述算法中我们创建一个新类 New_tm 来替代 tm 接受变量 x,并把 tm 从原来的模式中分离开来,使之只有一个超类 Root,如图 7 所示(有关图中虚拟关系解释见第 4 节).但使 tm 保留所有变量与方法 $I(tm)$,包括 $H(tm)$与 $N(tm)$.(这样做是为了不影响原先使用 tm 继承变量与方法的 PI 程序).我们形象地称 tm 为一个悬挂类.在图 7 中,New_student 替代了 Student 拥有变量 class_advisor,Student 类

被悬挂起来.如果模式中存在多个 Troublemaker 类,则按该路径无关语言在模式有向图中的传播顺序,把它们一一悬挂,代之以新类.



①虚拟关系.

Fig. 7 An equivalent schema digraph with Fig. 1 for method 'print_classroomNo'
图7 一个与图1关于"print_classroomNo"方法等价的模式有向图

下面证明对于任何已存在的路径无关语言 α,根据以上算法所得到的模式一定与原模式等价.

证明:分两种情况加以讨论:

(1) 若在类 t 中直接加上变量 x 之后,没有导致任何已存在的路径无关语言 α 的复用性问题,则直接将变量 x 加在类 t 上,显然所得到的新模式与原模式等价.

(2) 若直接将变量 x 加在类 t 之上,导致了若干条路径无关语言 α 的复用性问题,则因为 α 是一个三元组 (M,D,MA),M 仅是一个方法接口说明,所以 α 的功能主要由 D 与 MA 决定.对于增加一个新变量的模式演化种类,MA 的功能不可能因此而改变(因为 MA 由 $w.C$ 决定,增加一个新变量与 w,C 无关),只能影响传播方法 D.在 D 中,肯定会形成第 2 条从 F 到 T 的传播路径,在这种情况下,上述算法将导致问题产生的 Troublemaker 类(Troublemaker 类有可能是类 t 或类 t 的子类),从原模式中分离出来,变成一个悬挂类.从而保证了 F 到 T 的路径唯一性,使得原来已有的路径无关语言的重用性得到了保证.因为模式演化以后,F 到 T 的传播路径与演化前完全一样,故已存在的路径无关语言 α 的查询结果或执行功能不会受到影响,则新模式与原模式等价. □

3.2.1.2 ESE 系统的处理流程

图 8 描述了在我们自行研制的 ESE 系统进行等价模式演化的处理过程.模式演化需求如例 5.首先,模式预处理模块在原有模式的一份拷贝上,把 class_advisor 直接加入类 Student 中,形成了一个伪模式,如图 8(b)所示.PI 程序检查器发现在伪模式中,从 Student 到 Classroom 有两条路径.ESE 处理器利用上述算法计算出一种新的模式演化方案.ESE 执行器执行该方案,最终图 8(a)等价演化为图 8(c).如果 PI 程序检查器发现对于所有已存在的 PI 方法,伪模式都不影响其重用,则该伪模式就是最终的等价模式.由于路径无关语言本身具有较强的可适应性,这种情况反而是相当常见的.

3.2.2 从类 t 中删除变量 x

$N(t)=N(t)-\{x\}$;

$(\forall\ s\in G)\wedge(t\in PL(s))$ recompute $H(s),I(s)$;

while $(\exists\ tm\in G)\wedge(\forall\ \alpha\in$ PI program$)\wedge(t\in PL(tm))\wedge$

$(tm$ causes$(G;F(\alpha(!\ \rightarrow T(\alpha))\vee(\alpha$ is not functional in variant in evolved $G))$

do

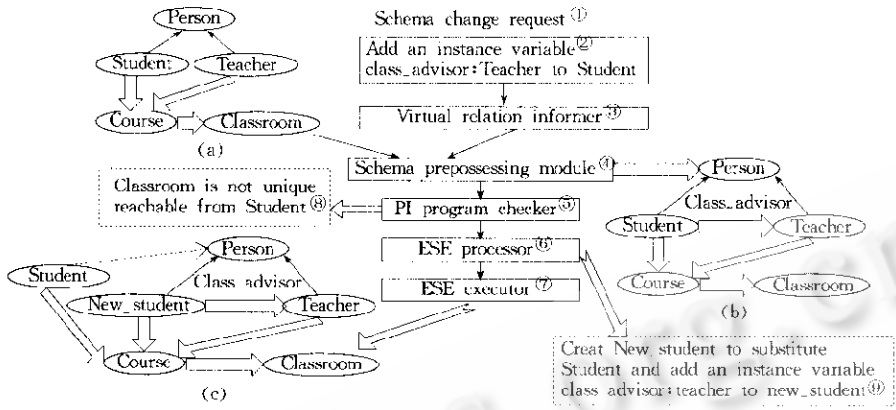
$\wedge\{$ creat new_tm,$N(new\_tm)=N(tm)$;

$(\forall\ s\in G)\wedge(tm\in P(s))$ $\{P(s)=P(s)+\{new\_tm\};P(s)=P(s)-\{tm\};\}$

$P(new\_tm)=P(tm);N(tm)=N(tm)+H(tm);N(tm)=N(tm)+\{x\}$;

$P(tm)=\{Root\}$;

$\}$



①模式演化需要,②增加一个实例变量,③虚拟关系通信器,④模式预处理器,⑤PI程序检查器,⑥ESE处理器,⑦ESE执行器,⑧Student类不是唯一到达Classroom类,⑨创建New_student类代替,为New_student类添加变量.

Fig. 8 Complete ESE procedure for an equivalent schema evolution
图8 等价模式演化的完整ESE系统处理流程

类 t 及类 t 的子类 s 被通知这一个变化,重新计算它们的 $H(s)$,$I(s)$. 如果 PI 方法仍在使用这一删除的属性,则可能导致重编程. 因此,上述算法创建 New_tm 类来替代 tm 类,变量 x 从 New_tm 删除,tm 被悬挂. 有关该算法的正确性证明类似算法 3.2.1.1 的证明,此处略.

利用上述两个算法,我们再回顾一下例 6. 例 6 的模式演化要求可描述为将类 Student 上的变量 like_course:Course 改为另一个新变量 dislike_course:Course. 这一演化要求可看成是删除一个旧变量 like_course:Course 与增加一个新变量 dislike_course:Course 的合并操作. 先后根据算法 2 与算法 1,最后由图 6(a)的模式可演化成图 9 的等价模式有向图. 因为篇幅有限,本文只能示意性地给出两个算法,至于 Orion 支持的其余几种模式改变,我们将在另文中给出(文献[8]中也有相关算法).

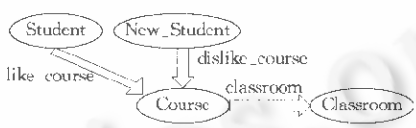


Fig. 9 An equivalent schema evolution whth Fig. 6(a)
图9 与图6(a)等价的模式演化

4 虚拟关系机制与透明模式演化

在前述的算法中,tm 类由于影响 PI 语言的重用而被"悬挂"起来. 这样,虽然软件的重用性得到了保证,但却导致了另一个问题. 如在图 7 中,如果我们加一个身份证号(ID)变量给 Person 类,已存在的 Student 的对象就无法利用这一个新的属性,因为此时 Student 已不是 Person 的子类. 这种情况显然是用户所不希望的,为了解决这个问题,我们提出了虚拟关系机制. 它使 Student 与 Person 之间建立了一个虚拟子类的关系,使 Student 也能得到相关的模式演化消息. 具体来说,我们采用了以下两个措施:

(1) 当 New-tm 被它的超类通知进行模式演化时,它就把这个消息转发给已被悬挂的 tm 类,使其也采取相应的动作.

(2) 当用户要求已悬挂的 tm 类进行模式演化时,tm 类也把这个消息告知 New-tm 类,由 New-tm 通知它的子类这一变化,同时 New-tm 本身也要执行这一演化操作.

通过 New-tm 与 tm 之间建立一个通信机制,虽然在 ESE 演化过程中,tm 类已被悬空,但它仍与原来的模式保持联系. New-tm 可以看做是 tm 类在模式演化后的一个新版本. 虚拟关系机制本质上是新老版本之间的一个相互通信的机制. 在图 8 的等价模式处理流程中,我们利用虚拟关系通信器模块实现了这个机制.

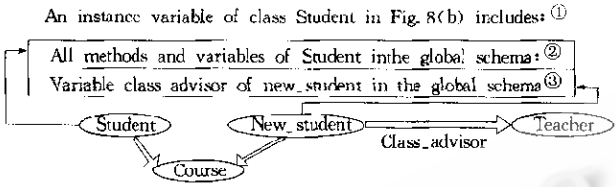
在 ESE 系统中,我们利用虚拟关系机制实现了透明模式演化. 例如,在图 8 中,一部分用户希望为 Student 类加上 class_advisor:Teacher 变量,而另一部分用户则没有这种模式演化要求,最后,利用等价模式演化算法,得到了图 8(c)的最终全局模式. 对于前一种希望模式演化的用户,系统可以为他们提供图 8(b)的视图(只要把图 8(c)中的 New_student 类映射为 Student 类,而图中的 Student 类对此种用户是不可见的),而对于后一类不希望模式改变的用户,可以提供图(a)的模式(这里,图(c)中的 New_student 类对此类用户是不可见的,利用虚拟关系机制,可使已悬空的 Student 类仍旧成为 Person 类的子类). 这样,系统内部的全局模式演化对于用户可以是完全透明,用户不必理会别人所提出的模式演化要求,甚至当全局模式经过多次复杂的修改之后,仍然可以按照自己原来所知道的旧模式的语义去编程访问数据库,而不必担心其适用性. 由于支持透明模式演化,用户就不会对不断变化的新模式语义感到难以理解,也不会增加新的编程困难. 这种机制的优越性是显而易见的.

5 演化对象机制

在等价模式演化算法中,我们将影响原有的路径无关语言重用的"Troublemaker"类分离出来,代之以一个符合模式演化要求的新类. 这样可以保证在已有类的路径上编制的有关 PI 程序的重用. 但产生的新问题是,在 ESE 系统中,当生成新类后,对已有的类的对象实例如何处理? 具体地说,在图 8(a)中,假如一位用户提出了为 Student 类加上 class_advisor:Teacher 变量,则系统根据等价模式演化算法,得到了图 8(c)中的等价模式,但由于 ESE 系统支持透明模式演化,这位提出模式演化要求的用户看到的却是如图 8(b)中的用户视图,那么,由于全局模式演化对用户的透明性,他肯定"误以为"全局模式也就是如此,因此,对他来说,在演化前已存在的如图 8(a)中属于 Student 类的对象实例也就顺理成章地是现在属于如图 8(b)中的 Student 类. 但事实上,由于该用户在图 8(b)中看到的 Student 类是图 8(c)中的 New_student 类,因此这里就存在语义误区. 我们必须在模式演化以后对原有类上的对象实例进行处理.

在国内外对面向对象数据库的模式演化研究过程中,许多支持模式版本化的系统都会遇到这种前后模式版本中的对象实例需要迁徙的问题. 在过去的系统中[7,9],多采用将旧版本的实例对象原封不动地拷贝一份到新版本中的方法. 这样,任何一个版本的实例对象的更改都需要传播到该对象的其他所有版本中. 这种做法给数据库的一致性维护带来了很大的麻烦. 本文提出了一种称为"演化对象"机制的解决方案. 数据库中的模式可以随着用户的要求而进行相应的演化,那么对已存在的对象实例为什么不能进行演化呢? 例如在图 8 中,可否让原来属于图 8(a)中的 Student 类的对象实例随着模式变更的进行而自动演化成为图 8(b)中的 Student 类中的对象呢? 我们认为这是可以的,其具体的解决思路是:由于在图 8(b)中的 Student 类实际上就是图 8(c)中的 New_student 类,这样就需要让原来属于 student 类的对象现在同时也是 New_student 类的对象实例,但 New_student 类的对象要比 Student 类的对象多一个变量 class_advisor. 因此,我们让这种由 Student 类演化而成的 New_student 类的对象由两部分构成,其中一部分是原来 Student 类已存在的对象实例,另一部分是现在的只有 class_advisor 这个新变量的 New_student 类的对象实例(这种对象在

原来的模式中不存在,是由系统自动为用户创建的,创建过程对用户透明).这样,图 8(a)中的 Student 类的对象通过加入一个变量 class_advisor 就演化成了图 8(b)中的 Student 类的对象实例,如图 10 所示.



①图8(b)中的Student对象实例包括,②全局模式中的Student类的所有变量与方法,③全局模式中的New_student类的class_advisor变量.

Fig. 10 Evolved object sketch map
图10 演化对象示意图

在实践中,我们的 ESE 系统采用如下的方法来实现演化对象机制.当在图 8(a)中,系统按用户的模式演化要求进行了等价模式演化得到了图 8(c)中的全局模式,然后将 New_student 类改名为 Student 以后,形成如图 8(b)中的用户视图,提供给提出模式演化要求的用户,并为该用户在全局模式中执行以下一条语句:

For each existing instance of Student Get_type class_advisor of New_student,

为每一个已存在的 Student 类的实例对象增加一个 class_advisor 变量,该变量属于 New_student 类.这样,原有的 Student 类的对象就"演化"成为新的视图中的 Student 类的对象了.当用户以后要使用新增加了的 New_student 类中的 class_advisor 这个变量时,系统就会创建一个新对象属于 New-student,用户只能使用该对象的 class_advisor 这一属性,其他变量与方法对此用户是不可见的.

从本质上来说,演化对象机制就是重新组织了已存在的对象实例的外在表示.在传统的概念中,每个存在的对象只能拥有一套只属于一个类的变量与方法实例,这些变量值存放在一个连续的存储单元中.而在我们的演化对象概念中,一个对象可以拥有不同类的变量与方法,这些变量值可以存放在离散的存储单元中,而且系统可以根据用户模式演化的要求,使用"Get_type"、"Lose_type"、"Get_method"、"Lose_method"等控制语句,随意组织一个对象的外在表示.演化对象机制与传统的拷贝对象方法相比,显然大大减少了数据库的开销和一致性维护的复杂度.

6 结 论

本文提出了路径无关语言以及在此基础上发展起来的等价模式演化策略.

(1) 路径无关语言是一种非导航式的面向对象语言,它能大大减少由于模式演化后造成的软件重编程.

(2) 等价模式演化机制既能保证不影响任何已存在的 PI 程序的重用性,又能表达所要求进行的模式演化的语义.

(3) 利用虚拟关系机制和多视图技术,我们实现了透明模式演化.

(4) 我们还提出了演化对象机制.

References:

[1] Ra, Young-Gook, Rundensteiner, E. A. A transparent schema-evolution system based on object-oriented view technology. IEEE Transactions on Knowledge and Data Engineering, 1997,9(4):600~624.

[2] Osborn, S. L. The role of polymorphism in schema evolution in object-oriented database. IEEE Transactions on knowledge and Data Engineering, 1989,1(3):310~317.

[3] Roddick, J. F. A survey of schema versioning issues for database systems. Information and Software Technology, 1995,37(7):383~393.

[4] Liu, Ling, Roberto, Zicari. The role of polymorphic reuse mechanism in schema evolution in an object-oriented database. IEEE Transactions on Knowledge and Data Engineering, 1997,9(1):50~67.

[5] Palsberg J., Lieberheer, K., Xiao, C. Efficient implementation of adaptive software. ACM Transactions on Programming Languages and Systems, 1995,17(2):264~292.

[6] Kim, W., Garza, J. F., Ballon, N., *et al*. Architecture of the ORION next-generation database system. IEEE Transactions on Knowledge and Data Engineering, 1990,2(1):109~124.

[7] Peters, R. J., Ozsu, M. T. An axiomatization model of dynamic schema evolution in objectbase system. ACM Transactions on Database Systems, 1997,22(1):75~114.

[8] Lu, J. H., Dong, C. L., Dong, W. W. An equivalent object-oriented schema evolution using path independence language. In: Chen, Jian, ed. Technology of Object-Oriented Language and System 31th. New York: IEEE Computer Society Press, 1999. 212~217.

[9] Ozeu, M. T., Peters, R. J., Szafron, D. TIGUKAT: a uniform behavioral objectbase management system. Vast Large Database Journal, 1995,4(3):445~492.

An Equivalent Schema Evolution Policy Based on Path-Independence Language*

DONG Chuan-liang, LU Jia-heng, YANG Hong, DONG Wen-wei

(*Information and Statistics Center, Shanghai Jiao Tong University, Shanghai* 200030, *China*)

E-mail: cldong@mail1.sjtu.edu.cn

http://www.sjtu.edu.cn

Abstract: The schema in object-oriented database (OODB) often experiences considerable changes during the development for typical application areas. After the update, the existing application programs based on the formerly schema have to be modified or rewritten, which makes a great deal of application programs obsolete. This paper addresses the problem by providing equivalent schema evolution (ESE) mechanism based on path-independence (PI) program. Path-Independence program is employed as OODB's specification formalism for enhancing the adaptability of database program against the schema evolution. Equivalent schema evolution can make the existing path-independence language reuse without any modification after the schema has been updated. In addition, a solution approach of remaining the equivalent schema evolution mechanism is described in this paper as a effective working system, which as its essential feature requires the system to support virtual relation mechanism and object-evolving technology.

Key words: adaptive software specification and development; schema evolution; object-oriented database; software reuse; virtual relation

* Received August 31, 1999; accepted June 19, 2000

Supported by the National High Technology Development 863 Program of China under Grant No. 863-306-ZD02-02-9