객체지향 뷰 기술을 이용한 투명한 스키마 진화
(Transparent Schema Evolution using Object-Oriented View Technology)

나 영 국 †
(Young-Gook Ra)

요 약 중대한 산업용 소프트웨어가 동작하는 공유된 객체지향 데이타베이스를 안전하게 변경하기 위해서는, 그 공유 데이타베이스를 변경할 동안 데이타베이스 위에서 작동하는 기존의 응용 프로그램이 지속적으로 작동되어야 한다. 데이타베이스 변경을 필요로 하는 새로운 요구사항은 새 응용 프로그램의 추가, 기존 응용 프로그램의 기능 확장, 초기 디자인 에러 수정 등으로 인하여 발생할 수 있다.

우리는 한 사람의 사용자가 다른 사용자에게 악영향을 주지 않고 데이타베이스 스키마를 변경할 수 있게 하여 이 문제를 해결하는 투명한 스키마 진화 (TSE: Transparent Schema Evolution) 방법론을 소개한다. 이 방법론은 기존의 스키마를 직접 변경하는 대신 스키마 변경 연산의 의미를 반영하는 데이타베이스 뷰를 공유 객체지향 데이타베이스 상에 생성하여 투명한 진화를 달성한다. 데이타베이스의 용량을 증가시키지 못하는 뷰 메카니즘의 한계를 극복하기 위하여 이 방법론은 데이타베이스 용량 증가 연산에 대하여 다음의 세 단계로 정련된다. (1) 기저의 베이스 스키마는 데이타베이스 용량 증가를 위해 물리적으로 변화한다. (2) 데이타베이스 변경의 의미를 달성하는 목표 뷰가 위의 변화된 베이스 스키마로부터 생성된다. (3) 변화 이전의 베이스 스키마는 데이타베이스 뷰로서 재 구축된다. 이로써 기존의 다른 사용자가 정의한 데이타 인터페이스가 보존된다.

우리는 객체-지향 뷰 기술을 이용하여 스키마 변화 연산을 구현함으로써 TSE 방법론의 구현가능성 (feasibility)을 확인하였다. 표준적인 객체-지향 뷰 모델이 정의되고 상용 객체-지향 데이타베이스인 젬스톤 (GemStone) 위에 구현되었다. 그 뷰 모델은 갱신 의미 (semantic) 정의를 그 뷰가 베이스 스키마의 갱신 의미를 보존하도록 정의하였다. 그러한 뷰는 사용자가 그들이 실제로는 베이스 스키마가 아니라 뷰에서 작업하고 있다는 사실을 모르게 하기 위하여 TSE에서 필요하다.

Abstract To safely update industrially critical software interoperating on a shared Object-Oriented Database (OODB), it is important to keep the existing applications running while the shared OODB is modified to satisfy newly emerging data service requirements. The new requirements may be due to adding new applications, extending the functionality of existing applications, correcting initial design errors, etc.

In this paper, we present the transparent schema evolution methodology (TSE) that addresses the problem by allowing one user to change the database schema without adversely impacting other users. The methodology achieves transparent evolution by constructing a database view over the shared OODB that reflects the intention of the schema change operation instead of modifying the existing schema in-place. To overcome the inherent limitation of view mechanisms not being able to augment the capacity of the database, the methodology is further refined for capacity-augmenting changes into the following three steps. (1) The underlying base schema is physically restructured to augment the database capacity. (2) A target view, which achieves the semantics of the schema change, is generated from the modified base schema. (3) The original base schema is reconstructed as a view over the modified base schema to preserve the interface required by existing users.

base schemas.

---

† 정 회 원 : 한경대학교 컴퓨터공학과 교수
ygra@ce.hankyong.ac.kr

논문접수 : 2000년 4월 26일
심사완료 : 2000년 12월 11일

We have validated the feasibility of the TSE methodology by implementing schema change operations using object-oriented (OO) view technology. A standard OO view model has been defined and implemented on top of a commercial OODB GemStone. The view model defines the update semantics of views such that the views preserve the update semantics of the base schema. Such views are required in TSE to shield users from the fact that they are operating on views rather than base schemas.

1. 서 론

데이타베이스 설계자는 데이타베이스 시스템이 현실 환경을 정확히 반영하는 것을 목표로 스키마를 구축한다. 그러한 스키마는 환경에서 실체를 확인하고 그 실체들 사이의 논리적 관계를 관계형 데이타베이스에서는 속성과 릴레이션 (relation)으로 정의하고, 객체지향 데이터베이스에서는 속성, 메소드 (method), 그리고 클래스 (class)로 정의한다. 그러한 스키마는 대상 환경이 변화해도 충분히 안정적이라 가정된다. 그러나, 현실에서는, 데이타 모델은 데이타베이스 설계자들이 통상적으로 생각하는 만큼 안정적이지 못하다.

데이타 모델의 불안정성은 모델링 실수, 요구사항 분석 실수, 대상 환경의 변화에 따른 새로운 요구사항 등에 기인한다. Marche[1]는 데이타 모델의 안정성에 관한 주장들을 조사하였다. 그는 많은 주장들이 건실한 이유가 없다는 걸 발견하였다. 그는 데이타 모델이 소프트웨어 프로그램에 비하여 상대적으로 안정적이라 언급하였을 뿐이고, 그것도 기껏해야 논란의 여지가 많은 논리와 예를 가지고 이루어졌을 뿐이다. 그는 데이타 모델의 안정성을 측정하는 경험적 연구를 하였고 일곱 개의 관계형 스키마 중 단지 하나만이 2년 후에도 50% 이상의 주 속성이 그대로 남아 있다는 것을 발견하였다 (이 기간은 평균 기간이며, 실제 관찰은 조사 대상 모델에 따라 6~7 개월 정도이다). 다른 경험적 연구로 Sjø berg[2]도 한 거대한 산업용 데이타베이스 시스템에서 1990년 6월부터 1991년 12월까지의 짧은 기간에 변경된 모든 릴레이션을 지적하면서 이 문제를 확인하였다.

스키마 변환은 단순하게 릴레이션 (또는 객체지향 데이타베이스에서는 클래스)의 더함/제거 또는 속성의 더함/제거가 있고 복잡한 릴레이션 (클래스)을 여러 개의 간단한 릴레이션들 (클래스들)로의 분해가 있을 수 있다. 이러한 변화가 전문화된 소프트웨어 도구에 의해 자동적으로 수행된다면, 특히 이 도구가 무결성 제약조건을 확인하고 이전 스키마에 따른 저장 데이타베이스를 새 스키마에 맞춰 변이하는 기능을 가지고 있다면 데이타베이스 관리자의 부담을 많이 줄일 수 있을 것이다. 실제로 많은 그러한 도구들과 방법론이 관계형과[3,4] 네트워크 모델[5]을 위하여 제안되었으며, 더욱 많은 연구가 객체지향 데이타베이스에 대하여 이루어 졌다. 이러한 스키마 변화 연구는 네가지로 분류된다. 첫째는 통상적인 스키마 변화 개념으로 직접 베이스 스키마를 변화시키는 것이다[4]. 둘째는 통상의 버전 (version) 개념으로 스키마 버전 시스템에 해당한다[6,7,8]. 셋째는 프로시져-기반의 투명한 스키마 변화 시스템으로 변화된 스키마와 그 인스턴스와의 타입 (type) 불일치를 프로시져로 해결한다[9,10,11,12]. 넷째는 뷰-기반의 투명한 스키마 변화 시스템으로 각 스키마 버전이 뷰로 구현된다[5,13,14,15,16,17].

많은 객체지향 데이타베이스가 지원하는 전형적인 스키마 진화 연산은 속성의 더함/제거, 속성의 도메인/이름 변화, 클래스의 더함/제거, 클래스 이름 변화, 그리고 상속 (inheritance)의 더함/제거, 메소드 (method)의 더함/제거 등이 있다[15]. 이처럼 풍부한 스키마 변화 연산이 현재 객체지향 데이타베이스에서 지원되지만, 사용자 (데이타베이스 관리자 또는 응용 프로그램 개발자)는 여전히 스키마 변화가 그 스키마에 대하여 쓰여진 기존의 응용 프로그램에 영향을 끼치는 문제를 겪고 있다. 데이타 모델에 관계없이, 응용 프로그램과 질의 (query)의 재작성이 스키마 변화의 주요 병목현상 (bottleneck)으로 보고된다[2,3,4,16,17]. Shneiderman[4] 은 의료보험 응용 프로그램의 경우에, 평균적으로 1000 문장의 PL/I-IMS 응용 프로그램을 변환하고 테스트하는데 프로그램 당 2 맨-먼스 (man-month)가 요구된다고 보고하고 있다. 이 문제는 데이타베이스가 여러 응용 프로그램에 의해 공유되면, 한 사용자가 수행한 스키마 변화가 다른 사용자의 프로그램에 원하지 않은 영향을 줄 수 있기 때문에 더욱 심각해진다.

아래 <그림 1>은 이 스키마 변화에 따른 프로그램 재작성 문제를 잘 설명한다. <그림 1>의 왼쪽에 그려져 있는 데이타베이스의 스키마에 대하여 *프로그램1* 이 동작하고 있고 새로운 요구사항을 만족시키기 위해 *프로그램2*가 개발 계획중이라 가정하자. 이 경우, 현재 데이타베이스 스키마가 *프로그램2*가 요구하는 데이타 서비스를 제공하지 못할 수도 있다. <그림 1>에서 묘사되어 있듯이, 현재 스키마 진화 시스템은 일반적으로 새 데이

타 서비스를 만족시키도록 사용자가 스키마를 변화하게 허용한다. 그러나, <그림 1>의 오른쪽 편에 묘사되어 있듯이, 기존의 *프로그램1*은 변경된 스키마에 대하여 더 이상 동작하지 않을 수 있다. 이 문제에 대한 단순한 해결책으로 (1) 새 스키마에 맞도록 영향을 받는 모든 응용 프로그램을 재작성하든지 (2) 스키마 업그레이드를 허용하지 않든지 하는 두 가지를 들 수 있다. 그러나 두 가지다 바람직하지 않다 - 첫째는 [4]에서 지적됐듯이 실제로 가능하지도 않고 비용이 많이 들며, 둘째는 데이타베이스의 발전과 진화를 막는다. 그러므로, 현재의 데이타베이스 시스템에서 스키마 변경 능력은 스키마 변화 언어의 능력보다는 기존 프로그램에 대한 영향에 의해 더욱 제약을 받는다[18,19].

이 영향을 극소화하기 위하여 기존 응용 프로그램이 스키마 변화 이후에도 지속적으로 동작하여야 하며 그렇기 위하여 옛 스키마가 보존되어 계속하여 갱신과 질의가 가능하여야 하며 관련 데이타는 항상 최신 상태로 유지되어야 한다. 우리는 이를 스키마 변화의 투명성이라 부른다. 정확히 말하면, 모든 인스턴스 (instance)는, 어떤 스키마 버전에서 생성되었는지에 관계없이, 스키마의 범위 (scope) 안에 있다면 - 스키마의 제약 조건을 만족한다면 - 어떤 스키마 버전으로부터도 보여지고 갱신 가능하여야 한다. 더욱이, 하나의 스키마 버전으로부터의 인스턴스 갱신은 그 인스턴스가 보이는 모든 다른 스키마 버전에 전파된다.

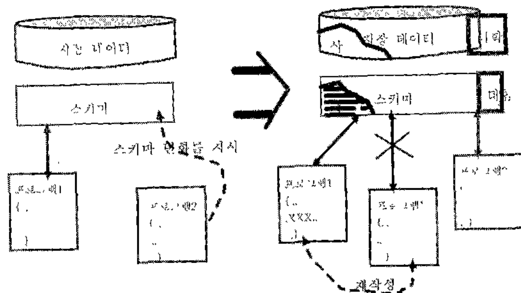

그림 1 현 스키마 변화 도구의 문제점

이 논문은 다음 장에서 객체 모델과 뷰 모델을 소개하고 뷰의 갱신도 함께 논의된다. 3장에서 스키마 변화 도구의 투명성을 엄밀하게 정의하고 투명하게 스키마를 변화시키는 TSE (Transparent Schema Evolution) 방법론을 소개한다. 4 장에서는 이 방법론으로 달성할 수 있는 스키마 변화 연산자들을 소개하고 그중 속성 제거와 속성첨가, 값 변이와 객체 변이, 그리고 에지 제거와 에지 첨가 연산을 자세히 설명한다. 5장은 TSE 구현 시스템의 구조를 간략히 소개한다. 6 장은 이 논문을 다른 관련 연구와 비교하고 7 장에서 결론을 맺는다.

2. 배경

이 논문은 기본 객체 모델을 가정한다. 이 기본 객체 모델은 매우 표준적인 객체지향 데이타 모델에 고유 갱신 (update) 연산들을 포함한다. 또한 우리의 TSE 방식에 필요한 뷰 기능을 지원하는 뷰 모델이 정의된다. 그리고 뷰 스키마가 사용자에게 베이스 스키마로 보여지도록 하기 위한 갱신 시멘틱 (semantic)이 조사된다.

2.1 객체 모델

이 연구에서 가정한 기본적인 객체 데이타 모델을 정의한다. 객체 모델은 고유의 갱신 연산을 포함하는 상당히 표준적인 객체지향 객체 모델이다. 아래에, 우리는 객체지향 데이타베이스 모델의 기본 개념을 소개한다. O가 객체 인스턴스 (instance)의 무한 집합이라 하자. O의 각 요소 $o \in O$는 추상적 데이타 타입 (ADT: Abstract Data Type)의 인스턴스 이다. 즉, 그것은 해당 ADT의 인터페이스를 통해서만 다루어질 수 있다.

클래스 C는 같은 인터페이스를 공유하는 인스턴스들의 모임이다. C는 유일한 클래스 이름을 갖고, 타입 묘사 (set description)와 소속 집합 (set membership)을 갖는다. 클래스의 타입은 그 클래스 인스턴스들의 공통 인터페이스에 해당한다. 속성은 미리 정의된 열거 (enumeration) 타입의 값 또는 어떤 클래스의 객체 인스턴스를 저장한다. 속성 $a[C]$ 는 그 이름 a와 속성을 정의한 클래스 C로 유일하게 확인된다. 메소드 (method)는 계산 함수이다. 메소드 $m[C]$는 역시 이름 m과 메소드를 정의한 클래스 C에 의해 유일하게 확인된다. 성질 (property) p는 속성 또는 메소드를 언급한다.

우리는 두 개의 클래스를 서로 연관시키는 특별한 목적의 속성을 관계속성 (relationship)이라 부르고 이를 r로 표시한다. 두 개의 클래스 (같을 수 있음) $C1$과 $C2$가 관계속성 r로 연결될 때, 그 연결을 $r(C1, C2)$로 표시한다. $C1$과 $C2$ 각각의 인스턴스 $o1$과 $o2$가 r로 연결될 때, 그 연결은 $r(o1, o2)$로 표시한다. 이는 $o2 \in o1 \rightarrow r$[1] 이고 $o1 \in o2 \rightarrow r$를 의미한다.

클래스는 객체 집합의 포함체 (container)이기도 하다. 클래스 C에 속하는 객체의 집합을 $외연(C) = \{o \mid o \in C\}$로 표현하고 이때 소속 조건 $\in$은 객체 인스턴스의 식별자 (identifier)에 의하여 결정된다. 객체 o, o

1) 표현 $o \rightarrow r$은 인스턴스 o의 속성 r 값을 언급한다.

$\in$ *외연(C)*에 정의된 성질의 집합을 *타입(C)*라 부른다. 어떤 성질이 클래스 C에 정의되어 있다면, $p[C]$는 *외연(C)*에 속하는 모든 객체 o에 대해 정의된다는 것을 주목한다. 그러므로, C에 정의된 모든 성질, $p[C]$,은 *타입(C)*에 속한다. 관계속성 r이 두 개의 대칭 속성으로 모델 된다는 것을 주목한다. 그러므로, 두 개의 클래스 $C1$과 $C2$가 r로 관련되면, $r \in$ *타입(C1)*과 $r \in$ *타입(C2)*이 성립한다.

정의 1 두 개의 클래스 $C1$과 $C2$에 관하여, $(\forall o \in O)((o \in C1) \Rightarrow (o \in C2))$일 경우만 $C1$을 $C2$의 부분집합이라 하고 $C1 \subset C2$로 표시한다.

정의 2 두 개의 클래스 $C1$과 $C2$에 관하여, *타입(C1)* $\supset$ *타입(C2)*일 경우에만 $C1$이 $C2$의 서브타입 (subtype)이라고 하며 $C1 \leqslant C2$로 표시한다.

정의 3 두 개의 클래스 $C1$과 $C2$에 관하여, $C1 \leqslant C2$이고 $C1 \subset C2$일 경우에만 $C1$을 $C2$의 서브클래스 (subclass)라 하며 $C1$ *isa* $C2$로 표시한다.

비공식적으로, (1) $C1$의 모든 멤버 (member)가 $C2$의 멤버 (부분집합 관계)이고 (2) $C2$에 정의된 모든 성질이 $C1$에도 역시 정의될 경우 (서브타입 관계)에 $C1$이 $C2$에 *isa* 관계에 있다고 한다.

스키마는 하나의 미리 정해진 뿌리 (root) 노드를 갖는데, 이 클래스는 뿌리라고 불리며, 스키마의 모든 클래스에 대한 슈퍼클래스이다.

이 클래스는 데이타베이스의 모든 객체 인스턴스를 포함하며 그 타입묘사는 공집합이다. 스키마의 모든 *isa* 에지는 잎으로부터 뿌리 노드로 향해 있다. 이는 스키마 그래프가 여러 개의 단절된 서브그래프 (subgraph)로 이루어지지 않고 하나의 DAG를 형성하게 한다.

우리는 데이타 모델에 모든 객체에 적용되는 고유 갱신을 포함하도록 하였다. 객체지향 데이타베이스에서 일반적으로 각 클래스에 특수한 갱신 메소드를 정의한다. 그러나, 우리는 다른 뷰 시스템 [10]에서 제안한 것처럼 타입-특수한 갱신을 확장하는 고유의 갱신 연산의 집합을 제공하길 원한다. 그 고유의 갱신 연산은 객체를 생성하고 소멸하는 *생성*과 *제거*를 포함하고 속성을 새로운 값으로 만드는 *변경*을 포함한다.

- *생성* 연산, (*<클래스> 생성*)으로 정의, 은 *<클래스>*의 인스턴스를 생성하여 그 클래스의 외연에 더한다.
- *제거* 연산, (*<객체> 제거*)로 정의, 은 그 객체를 소멸시킨다.
- *변경* 연산, (*<객체> 변경 [값 할당]*)으로 정의, 은 *<객체>*의 속성에 새로운 값을 할당한다. 예를 들면, (*<객체> 변경 [사람.월급 = 3,000]*)은 사람 클래스의 인스턴스 <객체>의 월급 속성 값을 3,000으로 만든다.

2.2 뷰 모델

우리는 베이스 클래스와 버츄얼 클래스를 구분한다. 베이스 클래스는 초기 스키마를 정의할 때 생성된다. 객체 인스턴스는 베이스 객체로 명시적으로 저장되는 베이스 클래스의 멤버들이다. 버츄얼 클래스는 객체지향 질의를 사용하여 데이타베이스 운용 중에 정의된다. 버츄얼 클래스는 데이타베이스의 상태에 기반 하여 정확한 회원 자격을 판별하는 회원 유도 함수가 관련되어 있다. 버츄얼 클래스의 외연은 일반적으로 명시적으로 저장되지 않고 필요시에 계산된다.

우리 뷰 모델의 객체 대수는 문헌에서[13,14,20] 발견되는 객체지향 뷰뿐만 아니라 SQL 뷰도 포함하기 충분할 정도로 일반적이다. 우리 뷰 모델에서 지원되는 객체 대수 연산자는 선택 (select), 숨김 (hide), 정련 (refine), 합집합 (union), 교집합 (intersection), 차집합 (difference) [14], 조인 (join)으로 아래에 자세히 정의되어 있다.

1. C = *선택(A, pred)*, A는 클래스이고 *pred*는 조건일 때, 는 클래스 C를 돌려준다. C의 외연은 A 외연의 부분집합이며, 조건 *pred*를 만족시키는 A의 모든 객체들로 구성되어 있고 C의 타입은 A의 것과 같다.
2. C = *숨김(A, attr)*, A는 클래스이고 *attr*은 숨겨지는 속성일 때, 은 클래스 C를 돌려준다. C의 타입은 *attr* 속성을 제외한 모든 A의 성질을 포함하며, C의 인스턴스는 A에 속한 인스턴스들이다.
3. C = *정련(A, m)*, A는 클래스이고 m은 새롭게 정의된 메소드일 때, 은 출력 클래스 C를 돌려준다. C의 타입은 A의 타입에 메소드 m을 더한다. A와 C의 클래스는 같은 객체 집합을 갖는다.
4. C = *합집합(A, B)*, A와 B는 클래스일 경우, 는 클래스 C를 돌려준다. C의 타입은 입력 타입의 최소 공통 슈퍼타입이며, 외연은 입력 클래스 외연들의 합집합이다.
5. C = *교집합(A, B)*, A와 B가 클래스일 경우, 는 클래스 C를 돌려준다. C의 타입은 입력 타입의 최대 공통 서브타입이며, 외연은 입력 클래스 외연들의 교집합이다.
6. C = *차집합(A, B)*, A와 B가 클래스일 경우, 는 클래스 C를 돌려준다. C의 타입은 첫 번째 변수 클래스 A의 것과 같으며, 외연은 A의 모든 인스턴스에서 B의 인스턴스를 뺀 것에 해당하며 A 외연의 부분집합이다.

이상이 멀티뷰[14]의 모델에서 제공하는 객체 대수 연산자이고 우리의 뷰 모델은 더 나아가 다음의 조인

연산자를 추가로 제공한다.

7. C = *조인(C1, C2, r),* *C1*과 *C2*는 클래스이고 *r*이 관계속성일 경우, 는 클래스 *C*를 반환한다. *C*의 타입은 관계 *r*을 제외하고 *C1*과 *C2*의 성질들을 포함한다. *C*의 외연은 $\{o|\ o = (o1, o2)$, *o1과 o2는 각각 C1과 C2에 속하고* $\exists r(o1,o2)\}$이다. 속성 *b*가 C에 정의되었다고 하면 *o*의 b 값은, $o \rightarrow b$로 표시, $o1 \rightarrow b$ (b가 C1에 정의) 또는 $o2 \rightarrow b$ (b가 C2에 정의) 와 같다. 만약 *b*가 *C1*과 *C2* 클래스 양쪽에 정의되어 있다면; $o \rightarrow b$는 $o1 \rightarrow b$와 같다, 즉, 조인 객체 *o*의 왼쪽 원천 객체 *o1*이 오른쪽 원천 객체 *o2*에 비해 높은 우선 순위를 가진다.

8. *C=동일조인(C1, C2, r), C1*과 *C2*는 클래스이고 *r*은 *C1*의 어떤 서브클래스 C_s와의 관계속성일 경우, 는 클래스 *C*를 반환한다. *C*의 타입은 *C1*과 같다. *C*의 외연은 $\{o|o = (o1, o2)$, $o1 \in C_s$ *이고* $o2 \in C2$, *또는* $o \in C1$ *이고* $o \notin C_s\}$이다.

정의 4 베이스 스키마 (BS)는 객체 스키마 *S=(V, E)*로, V의 모든 노드가, 유도되기보다는 저장된 객체 인스턴스를 가진 베이스 클래스에 해당한다.

정의 5 *V*는 버츄얼 클래스의 집합이다. 그러면 뷰 스키마 (VS), 또는 간단히 뷰, 는 다음과 같은 성질을 *가진 스키마 VS=(VV, VE)*로 정의된다:

1. VS는 <*VS*>로 표시되는 유일한 뷰 식별자를 가진다.
2. $VV \subset V$.

첫째 조건은 각 뷰 스키마가 유일하게 식별되어야 한다고 말하고 있다. 두 번째 조건은 VS의 모든 클래스는 버츄얼 클래스이어야 한다고 말하고 있다. 버츄얼 클래스간의 *isa* 관계는 베이스 클래스의 경우와 비슷하게 정의된다, 즉, 버츄얼 클래스 *VC1*과 *VC2*가 *VC1 isa VC2*라 하면 *외연(VC1)*$\subset$*외연(VC2)* 이고 *타입(VC1)*$\supset$*타입(VC2)*을 의미한다.

위의 정의들은 예를 들음으로써 더욱 잘 설명된다. <그림 2> (a)에서 보여지듯이 *사람, 학생, 피고용인, 조교*의 베이스 클래스를 포함하는 베이스 스키마 *BS*가

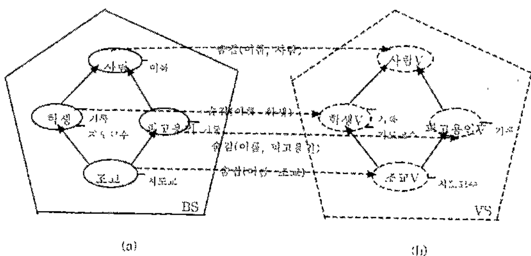

그림 2 뷰 스키마 생성 예

있다고 가정하자. 그러면 우리는 <그림 2> (b)에서 보듯이 *사람V, 학생V, 피고용인V, 조교V* 버츄얼 클래스로 구성된 뷰 스키마 *VS*를 가질 수 있다 - 이때 이들 버츄얼 클래스는 *사람, 학생, 피고용인, 조교* 베이스 클래스 각각으로부터 속성 *이름*을 숨기는 객체 대수 연산 *숨김(이름,<원천 클래스>)*을 적용하여 유도된다.

2.3 뷰 갱신 시맨틱

우리의 목표에 의하면, 뷰 스키마는 사용자에게 베이스 스키마로 보여져야 하기 때문에 베이스 스키마와 같은 성질 (property)을 가져야한다. 그렇지 않다면 베이스 스키마에서 개발된 응용 프로그램이 객체지향 뷰에서 동작할 때 비정상적인 행동을 보일 수 있다. 이는 뷰 스키마가 베이스라면 허용되어야 할 모든 질의와 갱신이 똑같이 뷰에서 허용되어야함을 의미한다. 그러나, 뷰에 대한 갱신이 베이스 스키마에 대한 갱신으로 변환될 때 모호한 시맨틱을 가진다는 것은 잘 알려진 사실이다. 이 문제는 뷰 갱신 문제 [8]로 알려져 있다. 이 문제를 해결하기 위해, 우리는 뷰 갱신 시맨틱 (semantic)을 다음과 같이 정의하여야 한다: (1) 뷰에 대한 갱신은 모호함 없이 기저의 베이스 스키마에 대한 갱신으로 번역된다; (2)기저의 베이스 스키마로부터 뷰에 대한 재-전파는 그 갱신이 뷰에 직접 수행되는 것과 동일한 효과를 가진다.

정의 6 *V*를 데이타베이스 뷰라 한다. 그러면, (1) *V*에 대한 갱신이 모호함 없이 아래의 베이스 스키마에 대한 갱신으로 번역될 수 있으며; (2) 베이스 스키마에 대한 갱신이 뷰에 다시 전파되는 효과가 마치 그 갱신을 뷰에 직접 했을 경우의 효과와 같을 경우; *V*를 *usp* (update semantic preserving) 뷰라 한다.

위의 정의에 따르면 뷰 *V*가 *usp*이면 사용자는 *V*가 뷰인지 베이스 스키마인지 구별할 수 없다. 그리고, *V*를 생성하는 함수를 *VD*라 할 때 *VD*를 *usp 뷰 생성*이라 한다.

3. 투명성 정의

이 장에서, 우리는 스키마 변화의 *투명성* 개념에 관하여 엄밀한 정의를 개발하고 이와 관련한 정의도 함께 소개한다.

정의 7 **A**를 스키마 집합으로 한다. 함수 $T{:}\mathbf{A} \rightarrow \mathbf{A}$가 $\mathbf{A}$의 모든 스키마 *S*, $S \in \mathbf{A}$, 에 대하여 정의되는 전체 함수 (total function)일 경우에만 함수 *T*는 스키마 변환이라 불린다.

스키마 *S*는 그것이 모델링하는 실세계에 대한 정보를 전달한다. 이 정보는 필연적으로 그 스키마 *S* 하에서 가능한 모든 데이타베이스로 담겨진다. 스키마 *S*에 의

해 모델되는 실세계의 모든 가능한 데이타베이스 상태의 집합을 *I(S)*로 표시한다.

예 1 스키마 S가 <그림 3> (a)에서 보여지듯이 단지 하나의 클래스 *사람*을 포함한다고 하자. 그러면 스키마 S의 데이타베이스는 <그림 3> (c)와 <그림 3> (d) 등을 포함한다. 이 데이타베이스는 *사람*의 OID들과 *이름* 속성들의 값들로 구성된다. 모든 그러한 데이타베이스의 집합은 S의 용량[2)]인 I(S)를 형성한다. S가 <그림 3> (b)에서처럼 또 하나의 클래스 주소를 포함하면, 새로운 S의 데이타베이스의 다른 예가 <그림 3> (e)이다. 그러면 I(S)의 저장 변수[3)]는 *사람*과 *주소*의 OID들, *이름*과 *시* 속성들, *addr* 관계속성 등이다.

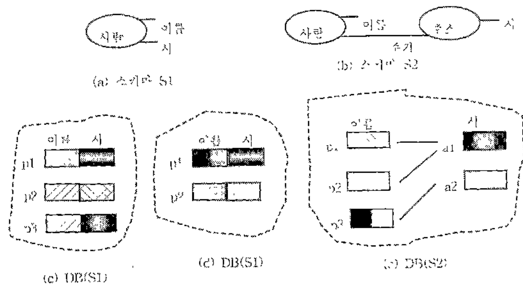

그림 3 스키마 용량의 예

정의 8 각 스키마 변환 T(S1) = S2, S1과 S2 $\in$**A**, 에 대하여, 함수 $\varnothing_T$:I(S1)→I(S2)를 연관시킨다. 이 함수 $\varnothing_T$는 T의 인스턴스 변환이라 불린다.

정의 9 스키마 변환 T는 $\forall$S1$\in$**A**에 대하여, 출력 스키마 S2 (= T(S1))가 입력 스키마보다 더 많은 용량을 가질 때만, 즉 I(S2)$\supset$I(S1)일 경우, 용량-증가라 불린다.

예 2 어떤 스키마 변환 T가 하나의 클래스로부터 몇 개의 속성을 추출하여 이들을 하나의 새로운 클래스로 묶는다고 (aggregation) 하자. 예를 들어, <그림 3> (a) 스키마에서 *사람*의 *시* 속성이 <그림 3> (b)에서 보여지는 새 클래스 주소로 변환되고 *사람* 클래스와 *주소* 클래스는 새로운 관계속성 addr로 연결된다. *주소* 클래스의 인스턴스들은 *사람* 클래스의 각 인스턴스에 대하여 하나씩 생성되며 *시* 속성 값들은 원래의 *사람* 인스턴스로부터 *주소* 인스턴스에 복사된다. 그러면 이 스키마 변환는 <그림 3> (b)의 새로운 스키마가 <그림 3> (a)의 원 스키마보다 두 개의 저장 변수, *주소* 클래스의 OID와 *주거* 관계속성을 더 가지고 있기 때문에 용량-증가이다.

우리는 또한 용량-보존과 용량-감소 스키마 변화를 아래처럼 정의한다.

정의 10 스키마 변화 T는 $\forall$S1$\in$**A**에 대하여 출력 스키마 S2 = T(S1)가 입력 스키마 S1과 같은 용량을 가질 때만, 즉 S2 스키마가 가질 수 있는 데이타베이스 상태의 수와 S1의 수와 같을 경우만, 용량-보존이라 불린다.

정의 11 스키마 변화 T는 $\forall$S1$\in$**A**에 대하여 출력 스키마 S2 = T(S1)이 입력 스키마 S1보다 더 적은 용량을 가질 때만, 즉 S2 스키마가 가질 수 있는 데이타베이스 상태 수가 S1의 수보다 작을 경우만, 용량-감소라 불린다.

정의 12 *T*를 *T: S1 → S2* 인 스키마 변화라 하자. 그러면, 어떤 usp 뷰 생성 VD가 존재하여 *VD(S1)*이 *S2*와 동일하거나 또는 어떤 usp 뷰 생성 VD′이 존재하여 VD′(S2)가 S1과 동일하면 T를 투명하다고 한다.

위의 정의에서 보듯이 투명한 스키마 변화를 통상적인 스키마 변화와 구별짓는 핵심은 옛 스키마와 새 스키마 각각의 데이타베이스를 사상하는 usp 뷰 생성 함수 VD (또는 VD′)의 존재 유무이다. 이 두 개의 데이타베이스간의 사상 관계는 두 스키마의 *데이타베이스 공유를* 가능하게 한다[4)]. 즉, T 가 투명하면, 입력과 출력 스키마 S1과 S2는 서로 *데이타베이스 공유 가능* 관계에 있으며, 그 역도 성립한다. 두 개의 스키마 S1과 S2 간의 *데이타베이스 공유는* 하나의 스키마에 대한 갱신이 다른 스키마의 데이타베이스에 영향을 주는 결과를 낳는다. 예를 들어, S1의 객체 o의 속성 b의 값을 변경하면 그 효과는 스키마 S2에서도 볼 수 있다 (객체 o가 스키마 S에 포함된다면). 즉, S1에 대한 갱신이 S2의 데이타베이스에 영향을 준다. 이 경우 객체 o가 S2에 포함되지 않는다면 이 갱신은 S2에 영향을 주지 않는다.

4. TSE 프레임워크 (Framework)

이제 우리는 통상의 스키마 변화 T를 *투명하게* 수행할 수 있게 하는 프레임워크 (framework)를 소개한다. 이, 소위 TSE라 불리는 프레임워크는 다음을 가정한다:

2) 스카마 S의 용량은 S가 표현하는 실세계의 모든 상태의 집합 I(S)로 정의된다.

3) 저장 변수는 데이타베이스의 속성 또는 OID로써 그 값이 물리적으로 저장되는 변수를 의미한다.

4) 두 개의 스키마 중 어느 하나가 다른 스키마로부터 뷰로써 유도될 수 있다면 이 두스키마는 하나의 저장 데이타베이스에 대한 각각 다른 시각을 제공하는 역할을 하고 물리적으로는 하나의 저장 데이타베이스만 필요하다. 즉, 두 개의 스키마가 하나의 저장 데이타베이스를 공유하는 것이다.

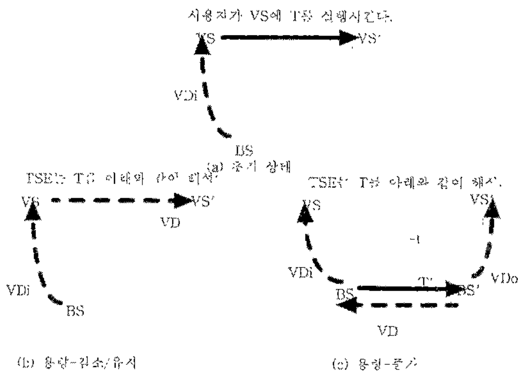

그림 4 TSE 프레임워크에서 스키마 변화

1. 모든 뷰 스키마가 기반으로 하는 베이스 스키마는 오직 하나 존재한다.

2. 모든 뷰 스키마는 베이스 스키마로 보이나 이는 실제로는 <그림 4> (a)에서 보듯이 베이스 스키마 *BS*로부터 유도된 usp 뷰이다.

3. 모든 데이타는 베이스 스키마 *BS*에만 관련되어 있다 (버츄얼 클래스의 가능한 머티릴라이제이션 (materi-alization)에 기인한 중복을 제외하면).

TSE 프레임워크에서, 직접 스키마 변화 T는 T의 실행을 모의 실험 (simulate)하는 usp 뷰의 생성으로 해석된다. 이는 T가 용량-증가이냐 아니냐에 따라 두 가지로 해석된다. *T*가 용량-감소/유지라면 <그림 4> (b)에 보여지듯이 *T*를 실현하는 뷰 생성 *VD*가 존재하여 이를 VS에 적용하여 *VS'*을 생성한다. *T*가 용량-증가이면, (1) <그림 4> (c)에서 보여지듯이, 출력 스키마 *VS'*의 용량을 포함하도록 베이스 스키마 *BS*의 용량을 충분히 증가시키는 직접 스키마 변화 *T'*을 생성한다; (2) *T'*이 적용되어 베이스 스키마를 *BS'*으로 변화시키고 목표 스키마 *VS'*은 *VDo*를 *BS'*에 적용하여 생성된다. 마지막으로, 원래의 베이스 스키마 BS는 usp 뷰 생성 함수 *T'*의 출력 스키마 *BS'*에 *T'*의 역에 해당하는 usp 뷰 생성 *VD*를 적용하여 복원된다.

다음은 이 TSE 프레임워크를 입력 스키마 VS가 하나의 베이스 클래스에서 *hide* 대수 (algebra) 연산자로 생성한 버츄얼 클래스 하나만 포함하는 단순한 예의 경우를 들어 설명한다. <그림 3>에서 위는 사용자 관점을 나타낸 것이고 아래는 TSE 시스템에 의해 행해지는 실제 변화를 나타낸다. 사용자가 <그림 5> (a)에서처럼 새로운 속성 *a*를 클래스 *C*에 첨가하는 ***속성-첨가****(a, C)* 스키마 변화 연산을 시행하였다고 하자. 그러면, <그림 5> (b)에서 보듯이 **속성-첨가** 연산이 용량-증가

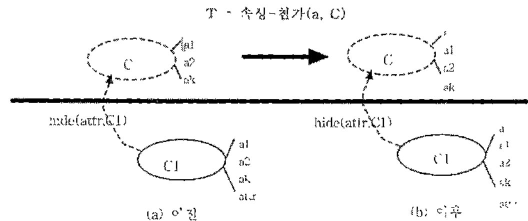

그림 5 TSE 스키마 변화 해석 실례

이기 때문에 *C*의 원천 베이스 클래스 *C1*에 속성 *a*가 첨가되어 *C1*은 직접적으로 변화한다. 다음은 변화된 <그림 5> (b)의 클래스 *C1*으로부터 목표 클래스 *C*가 유도된다.

5. 스키마 변화 연산자 (Operator)

우리의 TSE 방법론은 스키마 진화 연구들 [6,9,12, 15,16,17,21]에서 제안된 광범위한 스키마 변화 연산 집합을 포함한다. 이러한 연산은 기본적으로 (1) 속성 첨가 (2) 속성 제거 (3) 속성 이름 변화 (4) 속성 도메인 변화 (5) 클래스 이름 변화 (6)클래스간 상속 첨가 (7) 클래스간 상속 제거 (8) 클래스 제거 (9) 클래스 첨가 등이다. 이러한 기본적인 연산 이외에도, (10) 객체로 변이와 (11) 값으로 변이 두 개의 강력한 스키마 변화 연산 [17]을 포함한다. 이 연산들은 모두 TSE 방법론에 따라 실현 [22] 되었지만 이 논문에서는 지면 제약상 속성 제거, 속성 첨가, 값으로 변이, 객체로 변이, 클래스간 상속 첨가, 클래스간 상속 제거의 여섯 가지 연산만 소개한다. 이 중 속성 제거, 속성 첨가, 값으로 변이, 객체로 변이에 대해서만 TSE 구현을 설명한다.

5.1 속성 제거

이 연산자의 문법은 **속성-제거**(b, C)이다. C의 타입은 b 만큼 줄어든다. 외연은 동일하다. 타입 변화는 C의 서브클래스에 전파된다.

알고리즘 이 연산자가 용량-감소이므로, 입력 스키마 S에 **속성-제거**(b, C)가 적용된 출력 스키마 S는 다음의 뷰 스키마 유도 연산자 $VD_{속성-제거}$(b, C)에 의해 얻어진다:

1. S의 클래스 C와 그 모든 서브 클래스 $C_{서브}$에 대하여 $C_{서브}$'= **속성-제거**(b, $C_{서브}$)인 버츄얼 클래스들을 생성한다.

2. S의 모든 나머지 클래스 C_r 에 대해 C_r' = **동일**(C_r)인 버츄얼 클래스를 생성한다.

3. 위에서 생성된 버츄얼 클래스 $C_{서브}'$과 C_r'를 출력 스키마 S'에 포함시키고 그들의 원천 클래스

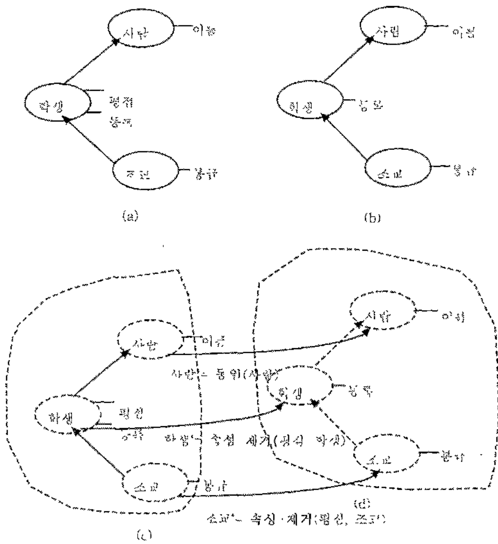

그림 6 속성-제거 뷰 생성 함수의 예

($C_{서브}$ 또는 C_r)가 서로 isa 관계가 있으면 해당 버츄얼 클래스 사이에도 같은 isa 관계를 구축한다.

예 3 <그림 6> (a)는 원래 스키마를 보여주고 (b)는 ***속성-제거**(평점, 학생)* 연산이 생성하는 새 스키마를 보여준다. 그러면, 이 연산은 <그림 6> (c)와 (d)에서 보듯이 세 버츄얼 클래스 ***동일**(사람)*, ***숨김**(평점, 학생)*, ***숨김**(평점, 조교)*을 생성함으로써 달성된다. 이 클래스들은 <그림 6> (d)의 목표 스키마를 생성한다.

5.2 속성 첨가

속성-첨가(x,C)로 정의되는 스키마 변화 연산은 클래스 C와 그 서브클래스 (subclass)의 타입에 속성 x를 더한다. 클래스의 외연 (extent)은 변하지 않는다 [23]. 만약 클래스 C에서 정의된 같은 이름의 속성 x가 존재하면 이 연산은 거절된다.

예 4 <그림 7> (a)와 (b)는 각각 ***속성-첨가**(등록, 학생)* 연산을 실행하기 전 스키마와 후 스키마이다.

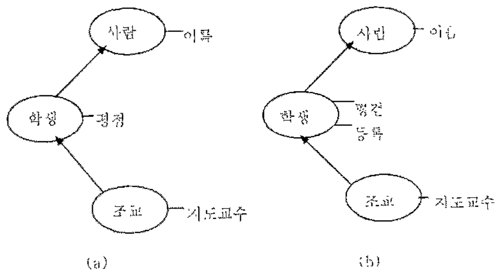

그림 7 첨가-속성 스키마 변화의 예

<그림 4> (c)에서 그려진 TSE 프레임워크에 따르면 이 **속성-첨가** 연산은 베이스 스키마 BS를 직접적으로 변화시키는 스키마 변화 연산 T'으로 번역된다. 이를 위하여 다음을 정의한다.

정의 13 **속성-첨가** 연산을 위하여 C의 주요 원천 클래스 pSrc(C)는 C의 생성 트리를 역추적하여 얻어지는 베이스 클래스를 지칭한다. 그 트리의 노드가 합 (union), 차 (difference), 조인 (join) 일 경우 첫 번째 매개 변수 (argument)에 대입되는 클래스가 역추적에서 선택된다.

그러면 사용자스키마 S에 적용되는 **속성-첨가**(r2, C) 연산 T는 직접 스키마 변화 연산 **속성-첨가**(r2, pSrc(C))로 번역된다.

정의 14 T를 사용자 스키마 S (베이스 스키마에 VD_i를 적용하여 유도된)에 적용되는 스키마 변화 연산이라 한다. 그러면 어떤 함수 f_T:T→T′와 f_{VD}: $VD_i \rightarrow VD_o$가 있어 베이스 스키마 BS에 T'을 적용하고 물리적으로 변화된 베이스 스키마에 VD_o를 적용한 한 것이 S가 T의 시맨틱에 맞게 변화한 결과를 낳으면 T를 VD_i에 대해 독립이라고 한다.

정리 1 T가 용량-증가 스키마 **변화**이고 VD_i가 $VD_1 \circ VD_2 \circ ... \circ VD_n$, 로 표시되고 VD_i, $1 \leq i \leq n$, 에 대해 베이스 스키마 변화 T_i', $1 \leq i \leq n$,이 있어 각 T_i가 VD_i에 대해 독립이면 T는 VD에 독립이다.

증명 이 정리의 증명은 [22]에서 찾을 수 있다.

위 정리는 어떤 사상 f:(T, VD_i)→(T′, VD_o)이 있어 각 뷰 스키마 유도 연산자에 대해 정확한 T′과

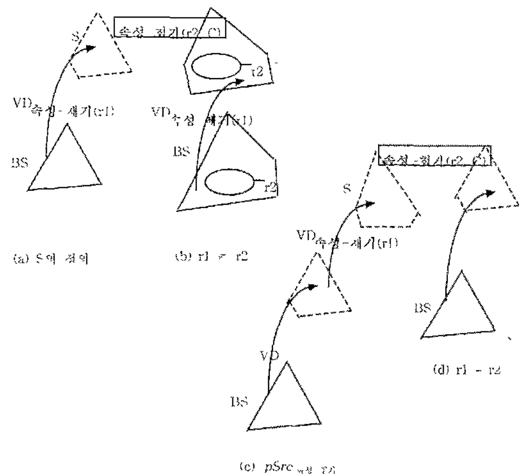

그림 8 속성-첨가 연산의 T′과 VD_0 사상

VD_o을 발견한다면 그 사상은 그 뷰 연산자의 임의의 결합에 대하여 성립한다는 것을 말한다. 이 정리를 이용하여 우리는 이제까지 발견한 뷰 스키마 유도 연산자 $VD_{속성-제거}$에 대해 **속성-첨가**를 달성하는 사상이 존재함을 보인다.

<그림 8> (a)에서 보듯이 사용자 스키마 S가 $VD_{속성-제거(r1)}$에 의해 정의된다고 하자. 그러면 T'은 C 클래스에 새로운 속성 $r2$를 첨가하는 연산이고 VD_{out}은 $VD_{속성-제거(r1)}$ 이다. 그러면 새 속성 $r2$는 <그림 8> (b)에서 보듯이 출력 스키마에서 나타날 것이다. 이는 왜냐하면 $VD_{속성-제거(r1)}$ 뷰 유도 함수가 $r2$가 $r1$과 같지 않고서는 $VD_{속성-제거(r1)}$이 $r2$를 숨길 수 없기 때문이다. $r1=r2$이면, 속성 $r2$를 숨기는 뷰 유도 함수 $VD_{속성-제거(r2)}$가 존재하고, 이 경우 <그림 8> (d)에서처럼 이 뷰 스키마 유도 연산자를 제거하면 ***속성-첨가*** *(r2, C)*가 달성된다.

BS가 **속성-첨가**(r2, pSrc(C)) 연산에 의해 직접 변경될 때, 옛 사용자 스키마 S에서 동작하는 기존의 응용 프로그램을 지속적으로 지원하기 위하여 사용자 스키마 S를 보존할 필요가 있고, 이를 위하여 원래의 베이스 스키마 BS가 복원될 필요가 있다. 이러한 S의 유지가 <그림 8> 에서 보여진다. 사용자 스키마 *S*는 <그림 9> (a)에서처럼 정의되어 있다. 그러면, 직접 스키마 변화가 있은 후에, <그림 9> (b)에서 보이듯이 ***속성-제거****(r2,C)* 연산은 ***속성-첨가****(r2,C)*의 역이므로 원래 베이스 스키마 BS는 ***속성-제거****(r2,C)*에 사용되는 뷰 스키마 유도 연산자 *VD(****속성-제거****(r2,C))*를 *BS'*에 적용하면 원래 베이스 스키마 *BS*가 얻어진다. 원래의 사용자 스키마 *S*는 <그림 9> (b)에서처럼 *S*를 발생한 뷰 스키마 연산자 VD_i를 변화된 스키마 *BS'*에 뷰 스키마 *BS*에 적용하여 얻어진다.

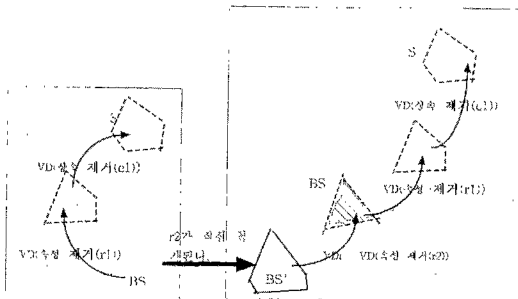

그림 9 속성-첨가 스키마 변화에서 원래 스키마 복원 단계

5.3 값으로 변이

이 연산자의 형식 (syntax)은 **값-변이**(C1,C2,r)이다. 이 연산자는 두 개의 클래스 C1과 C2를 하나의 클래스, 이름을 붙이자면 C로 합병한다. 이때 C1과 C2는 관계속성 r (정의 또는 상속)로 연결되어 있으며 C2는 말단 클래스이다. C1과 C2의 인스턴스 o1과 o2가 관계속성 r로 연결되어 있으면 이들은 하나의 새 객체 o12로 합병된다. C의 타입은 C1과 C2의 속성들의 합집합에 해당한다. 이 연산은 하위 클래스 $C_{서브}$에 전파되어 $C_{서브}$와 C2가 합병된다.

<그림 10> (a)의 스키마는 ***값-변이****(사람, 주소, 거주)* 연산에 의해 <그림 10> (b)의 스키마로 변화한다. <그림 10> (c)는 *사람*과 *주소*의 인스턴스들이 사람의 새 인스턴스들로 합병됨을 보여준다. 더 자세히 보면, *S1*과 *R1*, *S2*와 *R1*, *S3*와 *R3* 인스턴스들이 각각 *SR1*, *SR2*, *SR3* 인스턴스들로 합병된다. 이 합병은 *사람*의 하위 클래스, *학생*에 전파된다. 이 결과로 새로운 클래스 *(새)학생*이 나타난다.

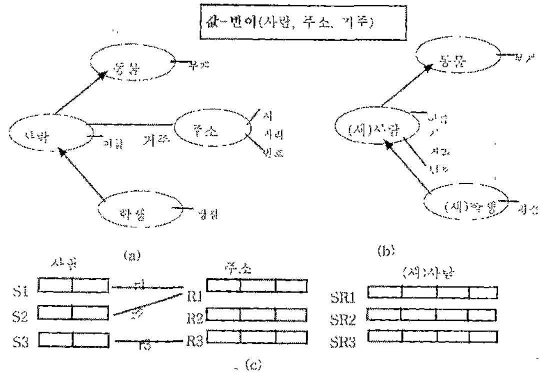

그림 10 값-변이의 예

뷰 스키마 유도 연산자 VD(**값-변이**(C1, C2, r))은 다음과 같이 정의된다.

1. C1의 모든 상위 클래스 w 에 대하여 동일조인(w, C2, r)으로 정의되는 버츄얼 클래스 w'을 생성한다.
2. C1 자체를 포함하여 C1의 하위 클래스에 대하여 조인(v, C2, r)으로 정의되는 버츄얼 클래스 v'을 생성한다.
3. C1과 C2의 상위나 하위 클래스가 아닌 모든 나머지 클래스 u에 대하여 동일(u)로 정의되는 버추얼 클래스 u'을 생성한다.

<그림 10> (a)의 *주소* 클래스를 *사람* 클래스의 속성으로 변환하려 한다고 하자. 위의 알고리즘에 따르면 동

물은 주소와 함께 동일조인되어 *동물'*을 생성한다, 즉 *동물' = 동일조인(동물, 주소, 거주)*. *사람*은 *주소*와 조인되어 *사람'*을 생성하고, 즉 *사람'= 동일조인(사람, 주소, 거주)*, *학생*은 *주소*와 조인되어 *학생'*을 생성한다, 즉 *학생'=조인(학생, 주소, 거주)*이다. **값-변이** 연산의 목표 뷰 스키마는 *동물', 사람', 학생' 버츄얼 클래스*로부터 만들어진다.

5.4 객체로 변이

이 연산자의 형식은 **객체-변이**(속성-리스트, C, D, r)이다. 이 연산자는 C로부터 속성-리스트의 지역 속성들을 추출하여 새 클래스 D로 변이한다. 그래서, 새 C 클래스의 타입은 이전 C 클래스의 타입에서 속성-리스트의 속성들을 뺀 것과 같다. 두 클래스들은 관계 r로 관련되어진다, 즉, C와 D의 인스턴스 o1과 o2는 그 둘이 예전의 C 클래스의 하나의 인스턴스로부터 분할된 것이면 r로 연결되어 있다. 이 연산은 C의 서브클래스 $C_{서브}$에 전파되고 그래서 새로운 $C_{서브}$의 타입은 원래 타입에서 속성-리스트의 속성들을 뺀 것과 같다. 새 $C_{서브}$ 클래스는 새로운 관계속성 r로 D와 연결되어 있다. 예전 $C_{서브}$의 인스턴스들은 새 $C_{서브}$과 D 클래스의 인스턴스들로 분할되고, 이들은 새 관계속성 r의 인스턴스로 연결된다.

예 5 <그림 11> (a)와 (b)는 각각 ***객체-변이**({시, 거리}, 사람, 주소, 주거)*가 수행되기 전과 후의 스키마를 보여준다.

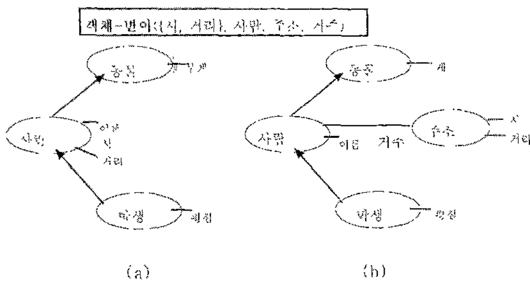

그림 11 객체-변이 스키마 변화의 예

객체-변이(속성-리스트, C, D, r)는 스키마 S (뷰 스키마 유도 연산자 VD_i로 정의되는)에 적용된다고 하자. 그러면 VD_i는 $VD_{속성-제거}$와 $VD_{값-변이}$의 임의의 결합이다. VD_i 안에 뷰 스키마 유도 연산자 **값-변이**(C1, C2, r)이 있고 C2의 속성들이 <그림 12> (a)에서 보듯이 속성-리스트와 같으면 VD_o는 VD_i에서 *값-변이(C1, C2, r)*를 제거함으로써 얻어진다. 그러면 VD_o를 베이스 스키마 BS에 적용함으로써 <그림 12> (b)에서 보듯이 출력 스키마 *S'*을 생성한다. 그렇지 않으면. 즉 VD_i 에 그러한 뷰 스키마 유도 연산자가 없으면, ***객체-변이**(속성-리스트, C, D, r)*는 물리적 스키마 변화 *T'*과 출력 뷰 스키마 유도 연산자 VD_o로 적절히 사상되어야 한다.

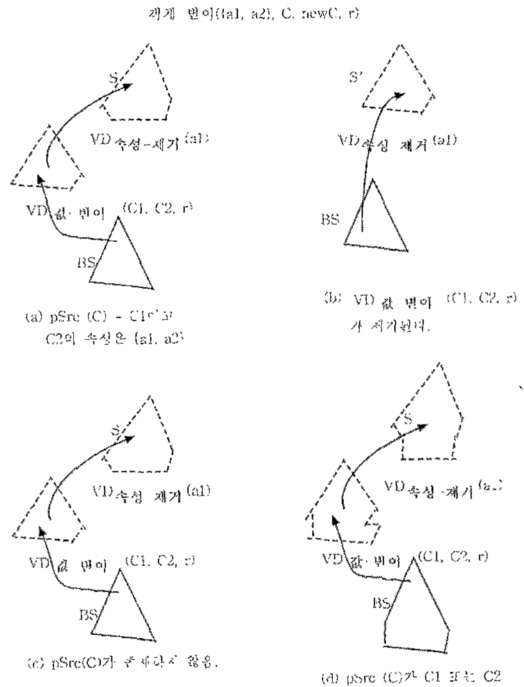

그림 12 객체-변이 연산의 T'과 VD_{out} 사상

정의 15 C의 주요 원천 클래스 C, $pSrc_{객체-변이}$(C),는 C의 유도 함수 트리를 역추적하여 얻어지는 베이스 클래스이다. 트리의 노드가 합집합 또는 차집합이면, 역추적에 첫째 변수 클래스가 선택된다. 노드가 조인이면, 원천 클래스는 속성-리스트의 모든 속성을 포함하는 변수 클래스가 선택된다.

위 정의에서, 우리는 역추적에서 조인 노드가 속성-리스트의 모든 속성들을 포함하는 변수 클래스를 가지고 있지 않을 때 $pSrc_{객체-변이}$(C)는 정의되지 않는다는 것을 유추할 수 있다. $pSrc_{객체-변이}$(C)가 존재하지 않으면 이는 달성될 수 없기 때문에 <그림 12> (c)에서처럼 ***객체-변이*** 연산은 거부된다 - 이는 [22]에서 증명된다. 그렇지 않으면, 즉 $pSrc_{객체-변이}$(C)가 존재하면, <그림 12> (d)에서 보듯이 스키마 S에 대한 ***객체-변***

이(속성-리스트, C, D, r) 연산은 ***객체-변이**(속성-리스트, $pSrc_{객체-변이}(C)$, D, r)*로 사상되고 VD_i와 동일한 VD_o가 변경된 베이스 스키마에 적용되어 목표로 하는 출력 스키마 S'을 생성한다.

5.5 에지 제거

이 연산자의 형식은 **에지-제거**(C1, C2) 이다. 이 스키마 변화 연산자는 C1에서 C2로의 isa 관계를 제거한다. 이 관계 제거로 인해 C1이 스키마로부터 분리되면, C1은 C2의 모든 슈퍼클래스의 직접 서브클래스로 만든다.

효과 측면에서 보면, C1과 C2 사이의 isa 관계 제거는 다른 isa 관계를 통해 여전히 상속되지 않으면 C1과 그 서브클래스로부터 C2로부터 상속받았던 모든 속성들을 숨긴다. 또한 C1의 외연을 다른 isa 관계를 통해 여전히 C2의 스코프 (scope)에 있지 않으면 C2와 그 슈퍼클래스로부터 제거한다. 이 연산자의 알고리즘은 [22]에서 참조한다.

예 6 <그림 13> (a)와 (b)는 각각 스키마 변화의 전후 스키마를 보여준다. *교원*과 *조교* 사이의 isa 관계를 제거하면 *강의* 성질이 더 이상 *조교* 클래스에 상속되지 않게 된다. 더욱이, 이는 *교원* 클래스로부터 *조교* 클래스의 외연 *{o4 o5}*를 숨기는 결과를 낳는다, 즉, 외연은 *{o2 o3 o4 o5}*에서 *{o2 o3}*로 줄어든다.

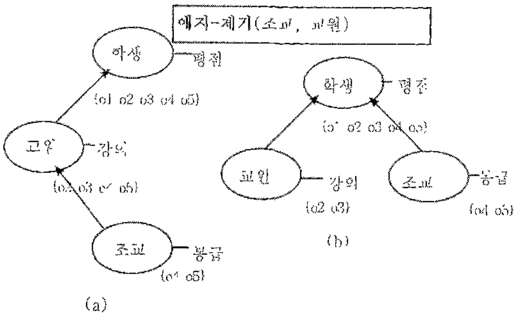

그림 13 에지-제거 스키마 변화의 예

5.6 에지 첨가

에지-첨가(C1, C2)로 정의되는 스키마 변화 연산자는 C2를 C1의 슈퍼클래스로 만듦으로써 두 클래스 사이에 isa 관계를 삽입한다. 의미적으로, 두 클래스 사이의 isa 관계 첨가는 C2 클래스의 모든 성질이 C1과 그 서브클래스로 상속됨을 의미한다. 첨가된 에지는 다중 상속 문제를 야기할 수 있다. 이에 대한 해결책은 같은 이름을 가진 두 개의 속성이 상속되고 사용자가 이들의 갈등을 해결토록 하는 것이다. isa 관계의 첨가는 C1 클래스의 외연을 C2와 그 슈퍼클래스의 외연에 더하는 결과를 낳는다.

예 7 <그림 14> (a)와 (b)에서, *행정직원* 클래스는 *조교* 클래스의 슈퍼클래스로 만든다. 결과적으로, *조교* 클래스와 그 서브클래스 *채점요원*은 성질 *상사*를 상속한다. 더욱이, *조교* 클래스의 외연은 *행정직원* 클래스의 외연에 합해진다. *행정직원* 클래스의 외연 *{o2 o3}*은 *{o2 o3 o4 o5 o6}*로 확대된다.

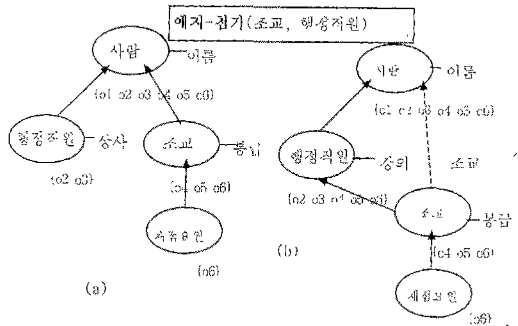

그림 14 에지-첨가 스키마 변화의 예

6. 구 현

<그림15>는 TSE 구현의 전체적인 구조를 보여준다. 전체 시스템은 4 개 층으로 이루어져 있다. TSE 시스템은 우리의 뷰 모델 (2.2 절 참조)과 객체 모델 (2.3 절 참조) 위에 구현된다. 그리고 이 모델들은 젬스톤 (GemStone) 데이타베이스 위에 구현되어 있다. 젬스톤은 우리의 객체 모델의 객체를 위하여 영속적인 저장소를 제공한다.

층 간의 화살표는 아래 층이 위 층에 어떤 기능을 제공한다는 것을 나타낸다. 우리의 뷰 모델은 TSE 시스

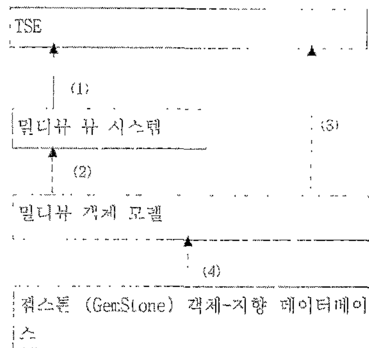

그림 15 TSE 시스템 구조

표 1 여러 가지 스키마 변화 도구의 비교

	투명성	투명성 공식화	카테고리	데이타 모델	용량증가/비 용량증가	기본/복잡	타입 불일치 해소	구현
Thomas&Shneiderman	no	no	4	네트워크	yes/no	yes/yes	하위 스키마	no
Shneiderman	no	no	1	관계형	yes/yes	yes/yes	불필요	no
Encore	yes	no	3	객체지향	yes/yes	yes/no	예외 처리	계획
Orion의 스키마 버전	no	no	2	객체지향	yes/yes	yes/no	불필요	yes
Goose	부분	no	2	객체지향	yes/yes	yes/no	불필요	yes
Bertino	yes	no	4	객체지향	yes/yes	yes/no	뷰	no
Bratsberg	yes	no	4	함수	yes/yes	yes/no	뷰	no
Clamen	yes	no	3	객체지향	yes/yes	yes/no	예외 처리	계획
CLOSQL	yes	no	3	객체지향	yes/yes	yes/no	갱신/백업 함수	미지
Rose	yes	no	3	객체지향	yes/yes	yes/no	지능형 인터페이스	계획
Tresh & Scholl	yes	no	4	객체지향	no/yes	yes/no	뷰	no
Breche	no	no	2	객체지향	yes/yes	yes/yes	불필요	yes
TSE-CAV	yes	no	4	객체지향	yes/yes	yes/no	뷰	부분
TSE	yes	yes	4	객체지향	yes/yes	yes/yes	뷰	yes

템을 구현하기 위하여 버츄얼 (Virtual) 클래스의 생성, 뷰 스키마의 생성/유지 [14]와 같은 필요한 객체-지향 뷰 기능을 제공한다. 우리의 객체 모델은 표준적인 객체 모델(섹션 2.3 참조) 로써 인캡슐레이션 (encapsulation), 하위 클래스 (subclassing), 다중 상속 (multiple inheritance), 폴리모르피즘 (polymorphism) 등의 특징이 있다. 객체 관리자는 베이스 스키마를 변화하는 통상의 스키마 진화 (evolution) 기능을 가지고 있다. 선 운영체제 (SunOs)와 스파크스테이션 (SPARCstation) 10 위에서 동작하는 젬스톤 4.0은 우리 객체 모델을 위한 영속적인 저장소를 제공한다. 그것은 또한 시스템 전체 구축을 위한 프로그래밍 언어로 사용되는 smalltalk 비슷한 인터페이스 언어인 OPAL을 제공한다.

7. 관련 연구

이 장에서 우리의 TSE 시스템처럼 투명한 스키마 변화에 가까운 기능을 제공하는 시스템들을 비교한다. 이는 다음의 기준으로 행해진다.

- 투명성은 스키마 변화가 다른 사용자에 영향을 주느냐 아니냐 하는 기준이다.
- 스키마 변화 시스템의 카테고리는 1,2,3,4로 나누어 진다. 이는 각각 직접 스키마 변경 시스템, 스키마 버전 시스템, 프로그램 구현 투명한 스키마 변화 시스템, 그리고 뷰-기반의 투명한 스키마 변화 시스템으로 불리운다.
- 스키마 변화 시스템이 기반으로 하는 데이타 모델을 분류한다.
- 허용되는 스키마 변화 연산의 범위를 나타낸다.
- 타입 불일치 해소는 스키마 버전간 타입의 불일치가 발생하였을 때 이를 해소하는 메소드가 제공되느냐 아니냐를 나타낸다.
- 구현은 제안된 시스템이 실제로 구축되었느냐 아니냐를 의미한다.

표 1은 우리의 TSE 시스템에 가까운 여러 가지 대표적인 시스템을 비교하여 놓은 표로써, 대상이 되는 시스템은 Thomas & Shneiderman [5], Shneiderman & Thomas [4], Encore [11], Orion's Schema Version System [6], Goose [24], Bertino [25], Bratsberg [26], Clamen [27], CLOSQL[9], Rose [12], Tresch & Scholl [15], Breche [28], TSE-CAV [16,29] -등이다. 직접 변화 시스템은 기존의 프로그램을 지속적인 지원이라는 우리의 목표에 너무 동 떨어져 Shneiderman & Thomas' [4]외에는 비교에서 제외됐다. Shneiderman & Thomas [4]는 프로그램 재작성 문제를 다루기 때문에 포함되었다.

8. 결론 및 미래연구

이 논문은 기존의 프로그램에 영향을 미치는 스키마 진화의 문제에 대한 해결을 제공한다. 어떤 스키마의 변화가 비록 같은 데이타베이스를 공유한다 해도 다른 스

키마에 영향을 주어서는 안 된다. 더욱이, 기존의 프로그램을 지속적으로 지원하기 위해서는 옛 스키마가 보존되고 데이타는 최신 상태로 유지되어야 한다. 우리는 이러한 조건을 만족시키는 스키마 변화 연산을 투명하다고 말한다.

이 논문의 주요한 기여는 이 투명한 스키마 변화라는 새로운 개념을 엄밀하게 정의했다는 데 있다. TSE 방법론에 따르면 사용자의 스키마는 베이스 스키마의 뷰로서 구현된다. 우리의 TSE 방법론의 유용성과 실제성을 입증하기 위하여 기본적인 스키마 변화와 강력한 변화를 망라하는 광범위한 스키마 변화 연산 집합을 사용하였다 [6,7,9,10,13,16]. 이 논문의 주요한 결과 중 하나는 이 집합의 연산 전부가 TSE 방법론의 프레임워크에서 투명하게 수행될 수 있다는 것을 보인데 있다.

사용자 스키마를 뷰로 구현함으로써 성능의 저하가 예상된다. 이의 보완을 위해서 두 가지 접근방식이 연구될 것이다. 하나는 사용자 스키마가 여러 개의 뷰 유도 함수의 결합으로 정의될 경우, 여러 뷰 유도 함수를 하나의 함수로 최적화 하여 뷰 유도 계산 시간을 줄이는 것이다. 다른 하나는 객체-지향 뷰에 적절한 뷰 구체화(materialization)를 적용하여 뷰에 대한 검색 시간을 단축하는 것이다. 이 두 방식은 서로 배타적이지 않기 때문에 함께 사용되어 시스템 성능을 향상시킬 것으로 기대된다.

참 고 문 헌

[1] S. Marche, "Measuring the stability of data models," European Journal of Information Systems, vol. 2, no. 1, pp.37-47, 1993.

[2] D. Sjø berg, "Quantifying Schema Evolution," Information and Software Technology, vol. 35, no. 1, pp.35-54, January 1993.

[3] V. M. Markowitz and J. A. Markowsky, "Incremental restructuring of relational schemas," in International Conference on Data Engineering, pp.276-284, 1988.

[4] Shneiderman and G. Thomas, "An architecture of automatic relational database system conversions," ACM Transactions on Database Systems, vol.7, no.2, pp.235-257, June 1982.

[5] G. Thomas and B. Shneiderman, 'Automatic database system conversion: A transformation language approach to sub-schema implementation," in IEEE Computer Software and Applications Conference, pp. 80-88, 1980.

[6] W. Kim and H. Chou, "Versions of Schema for OODBs," in Proc. 14th VLDB, pp.148-159, 1988.

[7] H. J. Kim, Issues in Object-Oriented Database Systems, Ph.D. thesis, University of Texas at Austin, May 1988.

[8] C. B. Medeiros and F. W. Tompa, "Understanding the implications of view update policies," in International Conference on Very Large Data Bases, pp.316-323, 1985.

[9] S. Monk, "A model for schema evolution in object-oriented database systems," in Ph.D. dissertation, Computing Department, Lancaster University, February 1993.

[10] M. H. Scholl, C. Laasch and M. Tresch, "Updatable views in object-oriented databases," in Proceedings of the Second DOOD Conference, December 1991.

[11] A. H. Skarra and S. B. Zdonik, "The management of changing types in an object-oriented databases," in Proc. 1st OOPSLA, pp.483-494, 1986.

[12] A. Mehta, D. L. Spooner, and M. Hardwick, "Resolution of type mismatches in an engineering persistent object system," in Tech. Report, Computer Science Dept., Rensselaer Polytechnic Institute, 1993.

[13] E. A. Rundensteiner, "Tools for view generation in OODBs," in International Conference on Information and Knowledge Management, pp.635-644, November 1993.

[14] E. A. Rundensteiner, "MultiView: A methodology for supporting multiple views in object-oriented databases," in 18th VLDB Conference, pp.187-198, 1992.

[15] M. Tresch and M. H. Scholl, "Schema transformation without database reorganization," in SIGMOD RECORD, vol.20, no.4, pp.16-20, 1991.

[16] Y. G. Ra and E. A. Rundensteiner, "A Transparent schema evolution system based on object-oriented view technology," IEEE Transactions on Knowledge and Data Engineering," 1997.

[17] Y. G. Ra and E. A. Rundensteiner, "Towards supporting hard schema changes in TSE," in International Conference on Information and Knowledge Management, 1995.

[18] W. Kent, "Solving domain mismatch and schema mismatch problems with an object-oriented database programming language," in International Conference on Very Large Data Bases, pp.147-160, 1991.

[19] V. Ventrone, "Semantic heterogeneity as a result of domain evolution," SIGMOD RECORD, vol.20, no.4, pp.16-20, 1991.

[20] S. Abiteboul and A. Bonner, "Objects and Views," SIGMOD, pp. 238-247, 1991.

[21] R. Zicari, "A framework for O_2 schema updates," in 7th IEEE International Conf. on Data Engineering, pp.146-182, April 1991.
[22] Y. G. Ra, "Transparent Schema Evolution (TSE) Using Object-Oriented View Technology: Transparency Theory, Methodology and System,", Ph.D. dissertation, Computing Science Division, EECS Dept., University of Michigan, December 1996.
[23] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth, "Semantics and implementation of schema evolution in object-oriented database," SIGMOD, pp.311-322, 1987.
[24] J. Andany, M. Leonard, and C. Palisser, "Management of schema evolution in databases," in VLDB, pp.161-170, September 1991.
[25] E. Bertino, "A view mechanism for object-oriented databases," in 3rd International Conference on Extending Database Technology, pp.136-151, March 1992.
[26] S. E. Bratsberg, "Unified class evolution by object-oriented views," in Proc. 12th Intl. Conf. on the Entity-Relationship Approach, pp.423-439, 1992.
[27] S. M. Clamen, "Type evolution and instance adaptation," Technical Report CMU-CS-92-133R, Carnegie Mellon University, School of Computer Science, 1992.
[28] P. Breche, F. Ferrandina, and M. Kuklok, "Simulation of schema change using views," in International Conference and Workshop on Database and Expert Systems Applications, 1995.
[29] Y. G. Ra and E. A. Rundensteiner, "A Transparent object-oriented schema change approach using view schema evolution," in IEEE Conference on Data Engineering, pp.165-172, March 1995.

나 영 국
1987년 서울대학교 전자공학과 졸업(학사). 1989년 The Penn. State Univ. 컴퓨터공학과 졸업(석사). 1996년 The Univ. of Michigan (Ann Arbor) Comp. Sci. 졸업(박사). 1997년 ~ 1999년 SDS 재직. 1999년 ~ 현재 국립한경대학교 전임강사. 관심분야는 데이타베이스 시스템, 데이터 모델링 방법론