객체지향 데이타베이스 스키마 버전 모델
(A Schema Versioning Model for Object-Oriented Databases)

이 상 원 † 김 형 주 ††
(Sang Won Lee) (Hyoung-Joo Kim)

요 약 본 논문에서는 객체지향 데이타베이스를 위한 스키마 버전 모델 RiBS를 제시한다. 본 모델은 풍부한 기본 스키마(Rich Base Schema: RiBS) 개념에 기반한다. 각 스키마 버전은 하나의 기본 스키마에 대한 클래스 계층구조 뷰이고, 이 기본 스키마는 모든 스키마 버전들에서 필요로 하는 스키마 정보를 모두 갖고 있다. 사용자는 스키마 버전을 통해서 데이타베이스를 접근하게 된다. 특히, 특정 스키마 버전을 대상으로 스키마 변경연산이 이루어질 경우, 이 연산 영향은 기본 스키마에 자동적으로 파급된다. 즉, 스키마 버전은 스키마 정보 갱신이 가능한 뷰이다.

본 논문에서는 우선 이 RiBS 모델의 세가지 구성, 즉 구조(structure), 무결성 조건(invariants) 그리고 스키마 변경 연산(operations)에 대해 설명한다. 그리고 RiBS 모델에 있어 중요한 작업인 스키마 버전 통합과 관련한 충돌의 종류에 대해 설명하고 이에 대한 해결책에 관해 언급한다.

Abstract In this paper, we propose a model of schema versions for object-oriented databases, called RiBS. At the heart of this model is the concept of Rich Base Schema(RiBS). In our model, each schema version is in the form of class hierarchy view over one base schema, called RiBS, which has richer schema information than any existing schema version in the database. Users are supposed to concern only with schema versions. Direct schema updates on schema versions are allowed, which are, if necessary, automatically propagated to RiBS.

We first describe the structural part of the model and then introduce a set of invariants which should be always satisfied by structural part. As the third element of our model, we give a set of schema update operations, of which semantics are defined so as to preserve all the invariants. Another contribution of this paper is the work on schema version merging within RiBS model. We identify several conflicts in schema version merging, and touch upon our solutions to these conflicts.

1. 서 론

객체지향 데이타베이스 시스템(Object Database Management Systems)과 관계형 데이타베이스 시스템의 가장 큰 차이점중의 하나는 데이타베이스 스키마 변경에 대한 지원이다. 객체지향 데이타모델이 80년대 중반이후 도입된 후, 객체지향 데이타베이스에서의 스키마 변경의 지원에 관한 많은 연구가 진행되었다[3,24,32]. 이는 CAD/CAM, CASE, 멀티미디어를포함한 객체지향 데이타베이스의 주요 응용분야에서 동적인 스키마 변경과 효율적인 스키마 관리를 필요로 하기 때문이다. 현재는 많은 상용 객체지향 데이타베이스 시스템들(예를 들어, GemStone[24], O2[32], ObjectStore[21], Objectivity[22])에서 기본적인 스키마 변경 연산들을 온라인으로 지원하고 있다. 또한, O2, Objectivity, ObjectStore 같은 제품들은, 기본적인 스키마 변경 연산들 외에, 사용자 정의 함수(user-defined functions)를 통한 스키마 변경을 지원한다.

그런데, 이들 시스템에서는 한 시점에 하나의 스키마만 데이타베이스 내에 존재한다. 즉, 한 스키마 변경 연산이 끝나면 이전 스키마는 더이상 존재하지 않는다. 이

· 본 연구는 통상부 지원 프로젝트 943-20-4, "객체지향 데이터베이스를 위한 설계도구 개발"과 학술진흥재단의 "신진 연구 인력 장려금"의 일부 지원에 의한 것임
† 학생회원 : 서울대학교 컴퓨터공학과
swlee@candy snu ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학과 교수
hjk@oopsla.snu.ac kr
논문접수 : 1997년 9월 8일
심사완료 : 1998년 9월 10일

와 같은 메카니즘은 [11]에서 지적한 것처럼 몇가지의 단점이 있다. 첫째, 스키마 변경 연산이 이전 스키마를 대상으로 작성된 응용 프로그램을 더 이상 동작하지 않게 한다. 둘째로, 모든 사용자가 하나의 스키마를 공유하기 때문에 한 사용자에 의한 스키마 변경은 다른 사용자들의 뷰를 변경시킨다. 이와 같은 문제를 해결하고자 제시된 것이 스키마 버전 기능이다[13,16].

특히, 최근에 들어 객체지향 데이타베이스의 새로운 응용분야인 Repository[5,28], PCTE(Portable Common Tool Environment)[8], WWW[2,31] 등에서 스키마 버전 기능의 필요성이 새로이 대두되고 있다. Atwood는 [2]에서 스키마에 대한 동적인 요구사항을 갖는 WWW 응용들은 스키마 버전 기능을 필요로 함을 지적하고 있다. 그리고, Silberschartz 등은 [28]에서 대부분의 복잡한 디자인 분야에서는 스키마 변경의 지원이 반드시 필요하고, 객체지향 데이타베이스의 주 응용분야의 하나로 주목받는 Repository[5]의 경우 스키마 버전의 관리가 필수적이라고 기술하고 있다. 또한, 스키마 진화와 스키마 버전 관리는 PCTE 환경에서 요구하는 객체지향 데이타베이스의 중요한 기능이다. 특히 Loomis가 [18]에서 지적한 것처럼, 계속 변화하는 PCTE 환경에서 각 도구들은 각기 서로 다른 스키마 버전을 대상으로 동작할 수 있어야 한다. 이들 사실로 알 수 있듯이, 앞으로 객체지향 데이타베이스가 대부분의 새로운 응용분야에 도입되고 효율적으로 사용되기 위해서는 스키마 진화와 스키마 버전의 관리가 필요하다.

객체지향 데이타베이스에서의 스키마 버전 지원에 관한 몇몇 연구들[6,13,20,26]이 있었지만 아직까지 만족할 만한 해결책이 없다. 본 논문에서는 객체지향 데이타베이스 뷰 개념에 기반한, 단순하면서도 강력한 스키마 버전 모델을 제시하고자 한다. 이 스키마 버전 모델은 객체지향 데이타베이스 시스템뿐 아니라 최근에 각광받는 객체/관계형 시스템에도 마찬가지로 적용가능하다

데이타베이스 스키마는 데이타베이스가 모델링하고자 하는 실세계의 엔티티(entity)들과 그들 사이의 관련성을 포함한 세만틱(semantics)을 표현한다. 본 논문에서는 객체지향 데이타베이스의 스키마 버전을 1) 계속 변하는 실세계하에서 특정 시점에서 의미있는 스키마의 스냅샷(snapshot) 혹은 2) 각기 다른 사용자의 관점을 재단(customization)을 위한 특정 스키마로 생각한다. 본 논문에서는 이와 같은 목적의 스키마 버전의 지원을 위한 *RiBS*(*Rich Base Schema*)모델을 제시한다. 이 모델의 아키텍처는 세 계층 - (1) 객체 베이스(object base) 계층, (2) 기본 스키마 *RiBS* 계층 (3) 스키마 버전 계층 -으로 이루진다. 사용자의 데이타베이스에 대한 접근은 단지 스키마 버전 계층을 통해서만 이루어진다.

이 논문의 구성은 다음과 같다. 2장에서 RiBS 모델의 개요를 설명하고, 3장에서는 본 논문에서 가정하는 객체 모델을 기술한다. 4, 5, 6장은 *RiBS* 모델의 세가지 구성요소 각각을 설명한다. 그리고, 7장에서는 객체 변환(object adaptation)과 특별한 스키마 관리 연산인 스키마 버전 통합(schema version merging)의 문제를 간단히 설명한다. 마지막으로, 8, 9장에서 각각 관련연구와 결론을 기술한다.

2. RiBS 모델의 기본 아이디어

이 장에서는 *RiBS* 모델의 기본 아이디어를 예를 통해서 간단히 설명하겠다. 모델의 각 구성요소별 자세한 설명은 4, 5, 6장을 참조하기 바란다. 설명의 편의상, 이 장에서는 다음의 간단한 객체지향 데이타베이스의 스키마에 대한 정의를 사용한다. 좀 더 자세한 스키마 모델에 대한 설명은 3장에서 기술하겠다. 데이타베이스의 스키마는 ISA 관계에 있는 클래스들의 계층구조로 구성된다. 각 클래스들은 애트리뷰트와 메소드를 포함하는 속성들로 이루어진다. 그리고, 각 클래스는 해당 클래스의 직접 속하는 인스턴스 객체들의 집합인 익스텐트(extent)를 갖는다. 익스텐트의 각 객체들을 해당 클래스의 직접 인스턴스 객체(direct instance object)라 한다.

2.1 풍부한 기본 스키마(Rich Base Schema)와 스키마 버전

예를 통해 아이디어를 설명하기에 앞서, 우리는 이 절에서 풍부한 기본 스키마(Rich Base Schema)의 개념을 우선 설명하고, 이 개념이 어떻게 스키마 버전을 지원하는데 사용될 수있는지를 알아보겠다.

우리는 다음의 모든 조건들이 만족되면, 하나의 스키마 S1이 스키마 정보에 있어 스키마 S2에 비해 풍부하다(rich)라고 말한다.

1. **스키마 S2의 각 클래스에 대해, 스키마 S1에 대응하는 클래스가 존재한다.**
2. **스키마 S2의 각 클래스의 속성에 대해, S1의 대응클래스에 대응 속성이 존재한다.**
3. **스키마 S2의 모든 직접 ISA 관계에 대해, S1에 대응하는 직/간접의 ISA 관계가 존재한다.**

스키마 S1이 S2에 비해 풍부하다는 의미는 직관적으로, S1이 S2보다 더 많은 스키마 정보를 가짐을 나타낸다. 이는 다시, 스키마 S2를 S1의 부분 스키마로 정의할 수 있음을 의미한다. 이 개념은 [9,19] 등의 연구에

서 도입된 *상대적 정보 용량(relative information capacity)*과 관련되어 있다. 이들 연구의 용어를 빌려쓰면, 위의 개념은 "*스키마 S1이 스키마 S2를 dominate (혹은 subsume)한다*"라고 표현할 수 있다.

RiBS 스키마 버전 모델은 풍부한 스키마 개념에 기반해서, 모든 스키마 버전들에 비해 스키마 정보면에서 풍부한 하나의 물리적 기본 스키마(이를 *Rich Base Schema(RiBS)*라 부른다)를 유지하고 모든 스키마 버전들은 이 *RiBS*의 뷰 형식으로 제공한다. 그리고, 특정 스키마 버전에서 발생하는 스키마 변경은, 변경 연산의 종류에 따라 그 영향이 *RiBS*에 전파되어, 항상 *RiBS*가 모든 스키마 버전들에 대해 풍부한 스키마를 갖도록 한다. 요약하자면, *RiBS* 모델에서 스키마 버전은 "갱신가능한 클래스 계층구조 뷰"(updatable class hierarchical view)[1]이다.

RiBS 모델에서 스키마 버전 계층은 *RiBS* 계층과 분리되어 있다. 이는, 뷰 클래스와 보통 클래스를 같은 계층에 두는 이전의 객체지향 데이타베이스 뷰에 관한 연구들[1,6]에서 나타나는 단점들[12]을 막기 위해서이다. 첫째, 보통의 클래스와 뷰 클래스가 하나의 계층구조내에 공존하기때문에, 사용자가 복잡한 클래스 계층구조를 이해하기가 힘들다. 다음으로, 각 클래스의 익스텐트가 겹치는 문제가 생긴다. 마지막으로, 뷰 클래스를 클래스 계층구조의 어디에 위치시킬 것인가를 결정하기가 힘들거나, 경우에 따라서는 불가능하다.

2.2 RiBS: 예제 스키마

이 절에서는 그림 1의 예를 통해 *RiBS* 모델의 전반적인 내용을 설명한다. 그림에서 보여지듯이, *RiBS*의 스키마 정보는 스키마 버전 SV1이나 SV2에서 필요로 하는 클래스 버전이나 속성 버전\footnote{스키마 버전에서의 클래스나 속성에 대해서는, 기본 스키마 *RiBS*의 클래스나 속성과 명확한 구분이 필요한 경우, 클래스 버전(class version), 속성 버전(property version)으로 기술하겠다.}과 관련한 정보를 모두 포함하고 있다. 즉, 스키마 버전 SV1과 SV2에 비해 스키마 정보가 풍부하다. 한 스키마 버전의 특정 클래스 버전에 대응하는 기본 스키마의 클래스를 해당 클래스 버전의 "기본 클래스"(direct base class)라 부른다. 예를 들어, *RiBS*의 BC1[2]은 스키마 버전 SV1의 클래스 C1의 기본 클래스

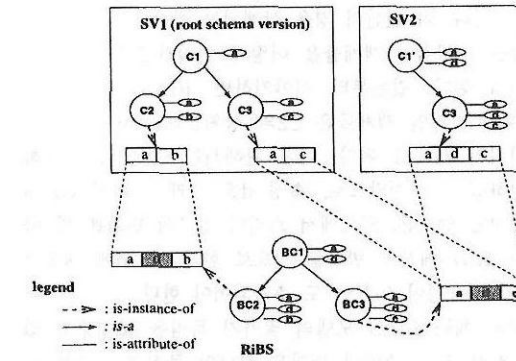

그림 1 RiBS와 스키마 버전 예

이다. 한 클래스 버전 C의 기본 클래스의 익스텐트에 포함된 모든 인스턴스 객체는 논리적으로 클래스 버전 C의 논리적인 객체가 된다. 실제로, *RiBS* 모델에서는 각 클래스 버전의 익스텐트가 명시적으로 관리되는 것이 아니라, *RiBS*로부터 유도된다.

그림 1에서 두 스키마 버전 SV1과 SV2은 *RiBS*의 인스턴스 객체들을 공유한다. 특정 스키마 버전하에서, 각 클래스 버전의 인스턴스 객체들은 자신의 기본 클래스에 속하는 물리적인 객체들에서부터 유도된다. 이는 [20]의 클래스 버전닝 연구와 밀접한 관련이 있는데, 그 연구에서는 여러 클래스 버전들에 의해 공유되는 인스턴스 객체가 해당 객체를 접근하는 환경(context)하에서 특정 클래스 버전의 타입으로 보여진다. 그림 1에서 우리는 다음의 가정을 한다. 우선, 스키마 버전 SV1이 먼저 생기고(이를 루트 스키마 버전(root schema version)이라 부른다), 스키마 버전 SV2는 SV1에서 유도되었다. 다음으로 SV2에서, 어떤 사용자가 다음 세가지의 스키마 변경 연산들 - 1) 클래스 버전 C1의 이름을 C1'로 변경, (2)클래스 버전 C2의 삭제, 그리고 (3) 클래스 버전 C1'에 속성 d를 추가 -을 차례로 수행한다. 이런 가정하에서 이들 스키마 변경 연산이 스키마 버전 SV2와 *RiBS*에 미치는 영향을 설명한다. *RiBS* 모델에서는 각 스키마 버전내에 소속 클래스 버전들에 대한 이름정보를 *RiBS*와 독립적으로 유지하고 있다. 따라서, 첫번째 스키마 변경 연산은 SV2내에서만 클래스 버전 C1의 이름을 C1'로 변경할 뿐, *RiBS*나 SV1에는 전혀 영향을 미치지 않는다. 두번째 변경 연산의 경우에도, 클래스 버전 C2에 대한 스키마 정보를 SV2내에서

1) 여기서 갱신가능하다는 것은 스키마 변경이 직접 스키마 버전을 대상으로 이루어진다는 의미임.

2) 본 논문에서는 기본 클래스들에 대해 BC1이나 BC2등의 이름을 붙이는데, 이는 단지 설명의 편의를 위한 것이다. 실제로 구현상에 있어서는 이와 같은 이름 대신에 기본 클래스 객체의 OID 등을 통해서 정보를 표현할 수 있다.

만 삭제할 뿐, *RiBS* 등에는 전혀 영향을 미치지 않는다. 그러나, 이 연산의 경우 삭제되는 클래스 버전 C2에 속하는 논리적인 객체들을 어떻게 처리하는가의 문제가 생기게 된다. 결론부터 이야기하면, *RiBS* 모델에서는 C2의 논리적인 객체들은 C2의 상위클래스 버전 C1′의 논리적인 객체로 된다. 이에 관해서는 6장에서 자세히 설명하겠다. 마지막으로, 속성 d를 클래스 버전 C1′에 추가하는 연산은, SV2에서 스키마 정보의 변화뿐 만 아니라, *RiBS*에서도 변화를 필요로 한다. 즉, d에 대응하는 기본 속성이 *RiBS*에도 추가되어야 한다.

위의 예는 *RiBS* 모델의 몇가지 특징을 보여주고 있다. 첫째, 모든 스키마 버전은 하나의 물리적인 기본 스키마 *RiBS*상의 논리적인 클래스 계층구조 뷰이다. 둘째, 기존의 객체지향 데이타베이스의 뷰 관련연구들[6]들과 달리, 각 스키마 버전은 갱신가능한 스키마들이다. 기존 뷰 연구에서는 뷰에서 발생하는 스키마 변경을 기본 스키마로 전달하는 메카니즘이 존재하지 않았다. 결국 뷰 스키마에 변화가 필요한 경우는 기존 뷰 스키마를 삭제한 후에 새로운 뷰 스키마를 생성하는 수 밖에 없었다. *RiBS* 모델에서는 사용자는 기본 스키마 *RiBS*가 아니라, 특정 스키마 버전을 대상으로 응용 프로그램이나 질의를 작성한다. 그리고, 이들 응용 프로그램이나 질의는 기본 스키마 *RiBS*에 대한 프로그램이나 질의로 변환되어 수행된다. 스키마 버전을 대상으로 작성된 질의는, 마치 관계형 데이타베이스에서 뷰를 대상으로 작성된 SQL 문장이 뷰의 기본테이블에 대한 질의로 바뀌는 뷰 변환 과정과 유사한 과정을 거쳐서 수행될 수 있다. 관계형 데이타베이스의 경우 뷰 변환 과정은 질의의 전체 수행 시간에 비해서는 상대적으로 짧은 시간이다. 따라서, *RiBS* 모델에서서도 스키마 버전을 대상으로 작성된 응용 프로그램이나 질의의 변환 과정의 오버헤드는 프로그램과 질의의 전체 실행시간에 비해 상대적으로 적다. 대신에 *RiBS* 모델은 스키마 관리의 효율성을 제고시키는 장점을 제공한다.

이 장에서 우리는 *RiBS* 모델에 대해 예를 통해 개략적으로 살펴보았다. *RiBS* 모델은 세가지의 기본 구성요소 - (1) 스키마 버전과 *RiBS*로 이루어진 구조(structure), (2) 스키마 버전과 *RiBS*가 항상 만족시켜야 하는 무결성 조건들(invariants), (3) 스키마 버전에 대한 스키마 변경 연산들(operations) - 을 갖고 있다. 이들 각각의 구성요소들에 대해 4, 5, 6장에서 자세히 설명하겠다.

2.3 논문의 기여 사항

기존의 스키마 버전 접근방법들[6,13,20,26]은 3가지 측면의 문제점, 즉, (1) 인스턴스 객체의 중복에 따른 기억공간의 오버헤드[13,26], (2) 제한된 스키마 변경 연산 및 스키마 변경과정의 복잡함[6], 그리고 (3) 일관성있는 스키마 관리의 부담[20]이 있다. 자세한 내용은 8장의 관련 연구를 참조하기 바란다. 본 논문에서 제안하는 RiBS 모델은 이들 문제점을 모두 극복함으로써 스키마 관리의 효율성과 유연성을 제공한다. 구체적으로 다음의 기여사항을 들 수 있다.

1. *RiBS* 모델은, 일종의 뷰 방식으로 정의된 스키마 버전에 직접적인 스키마 변경 연산을 가능하게 하는 첫 번째 시도로서, 스키마 버전에 대한 변경 연산을 스키마 버전 및 기본 스키마에 대한 변경연산으로 사상시키는 프레임워크를 제안하고 이에 필요한 구체적인 연산, 무결성 조건, 그리고 세만틱를 고안했다.
2. 위의 결과로서, *RiBS* 모델은 뷰 개념을 이용한 스키마 변경을 시뮬레이션하는 방법들의 단점인 제한된 스키마 변경 연산 및 변경의 복잡성 문제를 해결한다. 즉, 사용자는 기본 스키마 *RiBS*에 대해 정의된 일종의 뷰인 스키마 버전을 대상으로 모든 스키마 변경을 수행할 수 있고 이 변경의 효과는 사용자의 개입없이 자동적으로 기본 스키마에 반영된다.
3. 또한, 모든 물리적인 객체들은 *RiBS*라는 하나의 기본 스키마내에 존재하기 때문에 기억공간을 최소화한다. 아울러, 버전닝의 단위가 단위클래스가 아니라 클래스 계층구조 전체이기 때문에 사용자가 각각의 스키마 버전의 일관성 관리의 부담없이 자신이 행하고자 하는 스키마의 변경에만 관심을 가질수 있다.

3. 객체 모델

이 장에서는 *RiBS* 모델을 설명하기 앞서, 이 논문에서 사용하는 객체 모델를 정의한다. 이 객체 모델은 기본 스키마 *RiBS*와 각 스키마 버전에 공통으로 적용된다.

스키마에서 클래스 *C*는 자신의 인스턴스 객체들의 속성(property)들 -애트리뷰트(attribute)와 메소드(method) -을 정의하는 역할을 한다. 한 클래스에 직접 속하는 모든 인스턴스 객체들의 집합을 그 클래스의 익스텐트라 한다. 하나의 클래스 *C*는 하나 이상의 상위클래스를 가질수 있고, 이들 직접 상위클래스들의 집합을 $P(C)$로 표기한다. 클래스 *C*에 대해 이행적(transitive) 상위클래스 관계에 있는 모든 직/간접 상위클래스를 $P^*(C)$로 표기한다 상/하위클래스 관계에 있어, 상위클래스의 모든 속성은 하위클래스로 상속된다. 특정 클래스 *C*로 상속

된 속성들 $IP(C)$과 더불어, C에서 새로이 정의된 속성들 $LP(C)$은 C의 인터페이스(interface)를 이룬다. 클래스 C로 상속된 속성 p에 대해, $P^*(C)$내의 특정 클래스 C_p에 p의 원조 속성(origin property)이 반드시 존재하고 이를 $Org(p)$로 표기한다.

표 1은 본 논문의 객체 모델과 관련한 용어들을 정리한 것이다.

표 1 객체 모델을 위한 용어

용어	설명
C	클래스
S	*RiBS* 혹은 스키마 버전
$\tau(S)$	스키마 S내의 모든 클래스의 집합
$C.name$	클래스 이름
p	속성
$P(C)$	클래스 C의 직접 상위클래스 집합
$P^*(C)$	클래스 C의 직/간접 상위클래스 집합
$ISA(C_1,C_2)$	C_2가 C_1의 직접 상위클래스임
$ISA^*(S)$	스키마 S내의 모든 클래스간의 ISA관계
$I(C)$	클래스 C의 인터페이스
$IP(C)$	클래스 C의 상속 속성들의 집합
$LP(C)$	클래스 C의 지역 속성들의 집합
$E(C)$	클래스 C의 익스텐트
$Org(p)$	속성 p의 원조 속성

표 2는 위에서 기술한 객체 모델의 상속 세만틱과 관련한 axiom들을 보여주고 있다. [29]에서 지적한 것처럼, 상속의 세만틱과 관련한 많은 연구에도 불구하고, 아직까지 상속의 의미와 사용에 관해서 시스템이나 언어마다 차이를 보이고 있다. ODMG[7]와 같은 객체 지향 데이타베이스 표준안에서도 상속(특히 다중상속)과 관련한 정확한 의미를 제시하지 않고 있다. 본 논문에서는 전개할 내용들은 정확한 상속 세만틱을 필요로 하기 때문에, 표 2의 상속 세만틱을 정의했다.

Axioms 1부터 3까지는 보통의 객체지향 데이타 모델에서처럼 클래스 계층구조가 DAG(Direct Acyclic Graph)임을 요구한다. Axiom 4는, 앞에서 언급한 바와 같이, 클래스의 인터페이스는 상속 속성들과 지역 속성들로 이루어짐을 나타낸다. Axiom 5는 클래스 C의 상속 속성들 $IP(C)$은 상위클래스의 모든 인터페이스의 합집합임을 나타낸다. 마지막으로, Axioms 6은 클래스 C의 모든 상위클래스 집합을 나타내는 $P^*(C)$는 상위클래스 관계의 이행적 폐포(transitive closure)임을 의미한다. 이 axiom들에 따르면, 클래스 C는, 공통의 상위클래스를 갖는 C의 상위클래스들로부터는 오직 한번만 속성을 상속함을 알 수 있다 - 이는 객체지향 언어 C++의 가상 상속(virtual inheritance)의 의미와 동일하다. 또, 상속에 있어서 발생하는 이름 충돌 - 예를 들어, 서로 다른 두 상위클래스에서 같은 이름으로 선언된 지역 속성들 - 을 허용한다. 이의 대해서는 특정 속성을 필요로 하는 경우 사용자가 그 속성을 지정해야 한다. 이 점에서도 C++의 상속 세만틱과 유사하다.

표 2 Axioms for Inheritance

(1) Axiom of Closure	$\forall\ C \in \tau,\ P(C) \subset \tau$
(2) Axiom of Acyclicity	$\forall\ C \in \tau,\ C \notin P^*(C)$
(3) Axiom of Rootedness	$P(Object) = \{\} \wedge \forall C \in \tau - \{Object\},\ Object \in P^*(C)$
(4) Axiom of Interface	$I(C) = IP(C) \cup LP(C)$
(5) Axiom of Property Inheritance	$IP(C) = \cup_{C' \in P*(C)} I(C')$
(6) Axiom of Superclasses	$P^*(C) = \cup_{C' \in P(C)} P^*(C') \cup P(C)$

4. RiBS 모델: 구조

앞에서 기술한 바와 같이 *RiBS* 모델은 3 계층의 구조 - 즉, 물리적 객체가 존재하는 객체 베이스, 기본 스키마 *RiBS*, 그리고 스키마 버전 계층 -으로 이루져있다. 이 장에서는 이들 구조의 각 구성요소들과 이들 사이의 관계를 Definition 형식으로 설명하는데, 독자들의 이해를 돕기위해 2장에서 사용한 예를 이용하겠다.

정의 1 (기본 스키마, *RiBS*) 하나의 데이타베이스에는, *RiBS*(Rich Base Schema)라 불리는 하나의 기본 스키마가 존재하고, 이는 해당 데이타베이스내에 물리적으로 존재하는 객체들의 구조를 정의한다.

*RiBS*는 기본 클래스들의 집합이고 이들 사이의 상속 관계가 *RiBS*의 클래스 계층구조를 이룬다. 그림 1에서 *RiBS*로 표시된 부분에 해당한다.

정의 2 (스키마 버전, *SV*) 하나의 스키마 버전 *SV*는 기본 스키마 *RiBS* 위에 정의된 논리적인 클래스 계층구조로서, 계속적으로 변화하는 데이타베이스의 특정 시점에서의 스키마나 특정 사용자의 뷰를 나타낸다.

그림 1에서 SV1과 SV2는 각각 *RiBS* 위에 정의된 스키마 버전들의 예이다.

정의 3 (현재 스키마 버전, *CSV*) 특정시점에 사용자나 프로그램이 사용하고 있는 특정 스키마 버전을 현재 스키마 버전(Current Schema Version: *CSV*)이라 부른다.

사용자는, 데이타베이스내의 객체를 접근하기 위해서는, 반드시 하나의 특정 스키마 버전을 *CSV*로 지정해서(6장 참조), *CSV*를 통해서 객체를 접근해야 한다. 만일 사용자가 그림 1에서 SV2를 통해서 데이타베이스의 구조를 접근하는 경우에 SV2가 '현재 스키마 버전'이 된다.

정의 4 (스키마 버전 유도 그래프, *SVDG*) 스키마 버전 유도 그래프(Schema Version Derivation Graph: *SVDG*)는 DAG(Directed Acyclic Graph)형식의 그래프로써, 각 노드는 하나의 스키마 버전을, 각 간선(edge)는 스키마 버전사이의 유도관계를 나타낸다.

*SVDG*와 관련하여 데이타베이스의 생성초기에, 기본 스키마 *RiBS*와 더불어, "root schema version"이라 불리는 특정한 스키마 버전이 같이 생성된다. 그림 1의 상태에서는 SVDG는 SV1과 SV2의 두개의 스키마 버전으로 이루어진다. SV1에서 SV2가 유도되었기 때문에 *SVDG*는 선형(linear) 구조를 갖는다. 그러나, 두개 이상의 부모 스키마 버전에서 새로운 스키마 버전을 유도할 수 있기 때문에 *SVDG*는 DAG 구조를 가질 수 있다.

정의 5 (클래스 버전 *CV*과 기본 클래스 *B(CV)*) 특정 스키마 버전내의 클래스 버전 *CV*는 *RiBS*내의 특정 기본 클래스를 모델링하는 역할을 한다. 이 기본 클래스를 클래스 버전 *CV*의 기본 클래스라 부르고, *B(CV)*로 표기한다.

특정 스키마 버전내의 한 클래스 버전에 대해서 기본 스키마 *RiBS*에는 반드시 하나의 대응하는 기본 클래스가 존재하지만, 그 반대의 경우는 성립하지 않는다. 즉, 어떤 기본 클래스는 특정 스키마 버전에서 명시적으로 모델링할 필요가 없는 경우 대응하는 클래스 버전이 없을 수 있다. 예를 들어, 그림 1의 SV1의 C3의 기본 클래스는 *RiBS*의 BC3이다. 그러나, SV1에서 *RiBS*의 BC2를 기본 클래스로 갖는 클래스 버전은 없다.

앞에서 간단히 언급한 것처럼, *RiBS* 모델에서는 스키마 버전에 대한 스키마 변경연산의 효과가 *RiBS*로 전달될 경우가 있다. 기본 클래스의 중요한 역할 중의 하나는 *RiBS*로 이 변경연산을 전달할 대상 클래스의 역할을 한다는 점이다.

정의 6 (익스텐트 기본 클래스들, $B^+(CV)$) 특정 클래스 버전 *CV*의 익스텐트 기본 클래스들(Extent Base Classes)은, *CV*에 속하는 논리적인 객체들의 집합을 이루는 *RiBS*내의 기본클래스들의 집합이고, 이를 $B^+(CV)$로 표기한다.

예를 들어, 그림 1에서 SV1의 *C2*의 익스텐트 기본 클래스는 자신의 기본 클래스인 BC2로만 이루어진다. 그러나, SV2의 클래스 버전 C1′의 경우에는 자신의 기본 클래스인 BC1뿐만 아니라 BC2도 자신의 익스텐트 기본 클래스에 포함된다. 따라서, BC1과 BC2에 속하는 모든 인스턴스 객체들의 집합이 논리적으로 클래스 버전 C1′의 익스텐트가 된다.

다음 식에서처럼, $B^+(CV)$의 속하는 각 기본 클래스의 익스텐트들의 합집합이 클래스 버전 *CV*의 논리적인 익스텐트가 된다.

$$E(CV) = \cup_{C' \in B^+(C)} E(C')$$

정의 7 (속성 버전 *PV*과 기본 속성 *B(PV)*) 하나의 클래스 버전 *CV* 내의 특정 속성 버전 (Property Version) *PV*은 애트리뷰트나 메소드를 지칭한다. *CV*의 기본 클래스 *B(CV)*에는 *PV*에 대응하는 하나의 기본 속성을 갖고 있으며 이를 *B(PV)*로 표기한다

속성 버전에 대응하는 기본 속성을 유지하는 이유는 클래스 버전을 통해서 논리적인 객체를 만들때 속성 버전의 값을 어떤 기본 속성값에서 유도할 지는 결정하기 위해서이다.

표 3은 본 논문을 통해 자주 사용되는, 스키마 버전 관련용어를 정리한 것이다

표 3 스키마 버전 관련 기호

기 호	의 미
B(C)	클래스 버전 *C*의 기본 클래스
$B^+(C)$	클래스 버전 *C*의 익스텐트 기본 클래스
B(p)	속성 버전 *p*의 기본 속성

5. RiBS 모델: 무결성 조건

이 장에서는 *RiBS* 모델의 두번째 구성요소인 4 가지의 무결성 조건을 설명한다. 이 무결성 조건들은 *RiBS* 모델의 첫번째 요소인 구조가 항상 만족해야 하는 것들로, 다음 장의 스키마 변경 연산의 세만틱을 정의할 때 중요한 역할을 한다.

무결성 조건 1 (Phantom 객체 참조의 방지(No Phantom Reference)) 하나의 스키마 버전에서 객체식별자(object identitifier)를 통해서 접근되는 모든 객체는 해당 스키마 버전내의 특정 클래스 버전의 객체로써 존재해야 한다.

RiBS 모델에서 스키마 변경 연산 '클래스 삭제'의 의미를 잘못 정의하는 경우 특정 클래스 버전의 인스턴스 객체는 아니면서 다른 객체의 객체식별자를 통해서 접근이 가능한 객체가 존재할 수 있다 이와 같은 객체를 "phantom 객체"라 할 수 있다. 이 phantom 객체는, 모든 객체는 자신의 소속 클래스를 가져야 한다는 객체지향 데이타 모델의 원칙에 어긋난다. 따라서, 스키마 변경 연산의 결과로 이와 같은 phantom 객체 현상을 피할 수 있도록 해야 한다.

무결성 조건 2 (객체의 다중 클래스화 방지(No

Multiple Classification)) 스키마 버전내에서, 객체는 오직 한 클래스 버전의 인스턴스 객체로만 존재해야 한다.

무결성 조건 1에서와 같이, *RiBS* 모델에서 스키마 변경 '클래스 삭제' 연산의 의미를 잘못 정의하는 경우, 특정한 객체가 두개 이상의 클래스 버전의 인스턴스 객체로 보여질 수가 있다.마찬가지로 이 경우에도 객체지향 데이타 모델의 원칙에 위배되지 않도록 스키마 변경의 의미를 정의해야 한다.

무결성 조건 3 한 스키마 버전내의 각 클래스 버전 CV(그리고 각 속성 버전 pv)에 대해 기본 스키마에는 반드시 하나의 대응 기본 클래스 $B(CV)$(그리고, 대응 기본 속성 $B(pv)$)이 존재해야 한다.

무결성 조건 4 *RiBS*의 각 기본클래스는, 특정 스키마 버전내에서 반드시 하나의 클래스 버전 CV의 익스텐트 기본 클래스 $B^*(CV)$에 포함되어야 한다.

위의 무결성 규칙은 다시 말해 한 스키마 버전의 모든 클래스 버전들의 익스텐트 기본 클래스들의 합은 *RiBS*의 기본 클래스들의 집합과 동일하다는 의미이다.

$$\cup_{CVk \in B^*(C)} B^*(CV_k) = \tau(RiBS)$$

6. RiBS 모델: 스키마 버전 변경 연산

이 장에서는 *RiBS* 모델의 마지막 구성 요소인 스키마 버전 관리 연산들에 대해 설명한다. 이 연산들은 크게 두 개의 그룹 - 즉 *SVDG*와 관리를 위한 연산들과 스키마 버전내에서 스키마 변경을 위한 8가지의 기본 스키마 변경 연산들[3] - 으로 구분된다. 특히 스키마 변경 연산은 해당 연산의 *RiBS*와 다른 스키마 버전에 대한 영향의 유무에 따라 새롭게 구분한 것이다.

* *SVDG* 조작 연산

1. Derive *SV* from *parent-list*
2. Delete *SV*
3. Set *current schema version* to *SV*

* 스키마 변경 연산

1. 기본 스키마 *RiBS*에 영향을 주지 않는 스키마 변경 연산들(현재 스키마 버전 *CSV*에만 영향을 주는 스키마 변경 연산)
   (1) Change the name of a class version C
   (2) Change the name of a propery version v
   (3) Drop an existing class version C
   (4) Drop an existing property version v from a class version C
   (5) Drop an edge to remove class version S as a superclass of another class version C
   (6) Change the ordering of superclasses of a class version C
2. 기본 스키마 *RiBS*에 영향을 주는 스키마 변경 연산들
   (1) Add an edge to make class version S a superclass of class version C
   (2) Add a new property version v to a class version C
3. 기본 스키마 *RiBS*와 다른 스키마 버전에 영향을 주는 스키마 변경 연산들
   (1) Create a new class version C

6.1 SVDG조작 연산들

Derive *SV* from *parent-list*; 앞에서 언급한 바와 같이 이 연산은 parent-list에 있는 이미 존재하는 하나 이상의 스키마 버전(이를 부모 스키마 버전이라 한다)에서 새로운 스키마 버전(이를 자식 스키마 버전이라 한다)을 명시적으로 유도하는데 사용된다. 그런데, 둘 이상의 스키마 버전에서 새로운 스키마 버전을 유도할 때는 서로 다른 스키마 정보를 하나의 일관된 스키마 정보로 통합을 해야 한다. 이 "스키마 버전 통합" 과정에 대해서는 다음 장에서 설명하겠다.

Delete *SV*; 하나의 스키마 버전 *SV*가 더 이상 데이타베이스에서 더 이상 필요없을 경우, *SV*를 *SVDG*에서 제거하고, *SV*상에서 관리되는 모든 스키마 정보를 삭제한다. *SV*의 삭제 연산의 결과로 *SV*의 부모 스키마 버전은 SV의 자식 스키마 버전(if any)들의 새로운 부모 스키마 버전이 된다. 그리고, 루트 스키마 버전에 대한 삭제는 허용하지 않는다.

Set *CSV(current schema version)* to *SV*; 사용자나 프로그램이 데이타베이스를 접근하기 전에 자신이 사용할 현재 스키마 버전으로 *SV*를 지정하는데 사용되는 연산이다.

6.2 스키마 변경 연산

앞의 분류에도 나와 있듯이, *RiBS* 모델에서 제공하는 첫번째 스키마 변경 연산 그룹은 단지 현재 스키마 버전 *CSV*의 스키마 정보만을 변경하게 된다. 이런 측면에서 이들 연산은 객체지향 데이타베이스의 뷰 개념을 이용해서 스키마 변경을 시뮬레이션하는 연구들[6,12]과 유사하다. 그러나, 이와 같은 접근 방법에서는 *RiBS* 모델에서 지원하는 두번째, 세번째 그룹의 스키마 변경 연산들에 대해서는 효과적인 지원이 어렵다(관련 연구 부분 참조).

본 논문에서는, 각 그룹의 대표적인 연산 하나씩 - 즉, 세만틱을 정의함에 있어 복잡한 이슈를 야기하는 연산 -을 설명한다. 전체 스키마 변경 연산의 자세한 세만틱은 [17]을 참조하기 바란다.

6.2.1 클래스 버전의 삭제

이 연산은 현재 스키마 버전 *CSV*에서 클래스 버전 *C*를 삭제한다. 이는 클래스 버전 *C*의 직접 상위/하위클래스 버전들에서 *C*의 정보도 제거하게 된다. 이때 *C*가 직접 하위클래스 버전 C_{sub}의 유일한 상위클래스인 경우, *C*의 모든 직접 상위클래스 버전(들), *P(C)*이 C_{sub}의 새로운 직접 상위클래스 버전이 된다[3]. *C*에서 정의된 지역 속성들은 *C*의 모든 직/간접 하위클래스에서 제거된다.

RiBS 모델에서 모든 클래스 버전은 자신의 논리적인 익스텐트를 갖는다. 따라서, 하나의 클래스 버전을 삭제할 때 자신의 익스텐트에 속하는객체들을 어떻게 처리할 것인가의 문제가 생긴다. 이와 관련하여, 기존의 스키마 변경에 관한 연구들[3,21,32]에서는 크게 두가지 접근방법이 있었다. 첫째로, 삭제되는 클래스의 객체들을 데이터베이스에서 함께 삭제하는 방법이다[3,21]. 그렇지만 이 세만틱에 의하면, [3]에서 언급한 것처럼, 삭제된 객체들을 다른 객체에서 객체 식별자(object identity)를 갖고서 참조하는 "dangling reference" 문제가 야기된다. 상업용 객체지향 DBMS, ObjectStore[21]에서는 삭제되는 객체를 참조하고 있는 다른 모든 객체들에서 해당 객체 식별자를 nil로 세팅함으로써 이 문제를 해결하고 있다. 그렇지만, 이 방법 역시 스키마 변경에 소요되는 시간 오버헤드가 너무 심하다. 두번째 접근방법으로는, O2 시스템[32]에서 취하는 방법으로 클래스의 삭제는 해당 클래스의 익스텐트가 비었을 때만 허용하는 방법이다.

RiBS 모델에서는 이들 외에 가능한 또 하나의 세만틱이 존재한다. 즉, 클래스 버전 *C*와 *C*의 논리적 익스텐트 *Extent(C)*를 현재 스키마 버전 *CSV*에서 제거하는 것이다. 이 세만틱에 따르면, 이 스키마 변경 연산이 완료되었을 때, *Extent(C)*에 속하는 어떤 객체도 더이상 *CSV*의 클래스 버전을 통해 접근이 불가능하다. 이 세만틱은, *CSV*에서 특정 객체들의 집합을 제거하다는 의미에서, 마치 내용기반 보안 기능을 제공하는 관계형 데이타베이스의 뷰 기능과 비슷하다. 실제로 객체지향 데이타베이스 뷰에 관한 연구들[6,12]도 관계형 데이타 모델의 뷰의 장점을 이용하려고 노력했다.

그러나, 객체지향 데이타모델에서의 객체 식별자를 통한 객체 접근 패러다임은 데이타에 대한 접근의 단위가 테이블이나 뷰인 관계형 데이타모델의 접근방식과는 근본적으로 차이가 있다. 객체지향 데이타 모델에서는 하나의 클래스는 다른 클래스의 애트리뷰트의 도메인으로 사용가능하고, 따라서 객체의 특정 애트리뷰트의 값으로 다른 객체의 식별자를 가질 수 있다. 그런데, 객체지향 데이타 모델의 이런 특징은 클래스 버전 삭제와 관련한 세번째 세만틱하에서는 phantom 객체 참조 문제를 야기한다.

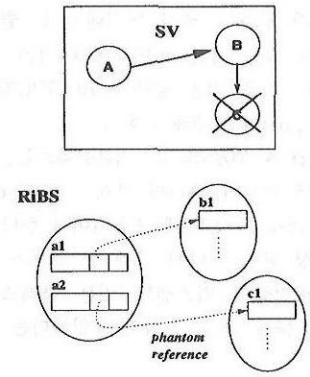

그림 2 Phantom 객체 참조 문제

그림 2에서 보여지듯이, 클래스 버전 C가 삭제된 뒤에도 객체 c1은 객체 a2에서 객체식별자에 대한 참조(reference)를 통해서 접근이 가능하다. 그러나, 클래스 삭제의 세번째 세만틱에 의하면 객체 c1는 현재 스키마 버전 *CSV*하에서는 어떤 클래스 버전의 익스텐트를 통해서도 접근이 불가능하다. 즉, c1이 스키마 버전 SV에서는 phantom 객체인 것이다[3]. 본 논문에서는 phantom 객체 참조 문제를 해결하기 위해 클래스 버전의 삭제의 세만틱을 다음과 같이 정의했다. CSV에서 클래스 버전 C의 익스텐트 *Extent(C)*에 속하는 모든 객체를 논리적으로 C의 특정 상위클래스 버전 S의 익스텐트로 옮기는 것이다[4]. 예를 들어, 그림 2에서 *Extent(C)*에 속하는 모든 객체를 *Extent(B)*로 옮기는 것이다. 다중 상속의 경우에 있어, 어떤 상위클래스 버전으로 C의 익스텐트를 옮겨야 할지를 자동으로 결정할 수가 없기 때문에 사용자가 명시적으로 해당 상위클래스 버전을 지정해야 한다.

3) 이 phantom 객체 참조 문제는 RiBS 모델에 한정된 것이 아니라, 객체지향 데이타모델의 뷰에 있어서는 공통으로 발생할 수 있다는 점은 주목할 필요가 있다.

4) 즉, $B^+(S) = B^+(S) \cup B^+(C)$의 의미이다.

6.2.2 상위클래스 버전의 추가

이 연산은 S를 클래스 버전 C의 직접 상위클래스들 $P(C)$에 추가한다. 이 연산의 결과로 클래스 버전 C는, 3장의 axiom들에 의해서, S의 인터페이스를 상속받는다[5]. 그런데, 만일 이 연산이 현재 스키마 버전 CSV내에서 중복 상속(redundant ISA) 관계를 형성하게 되는 경우, 이 연산은 거부된다.

이 연산은 다음 두 경우를 제외하고는 $RiBS$에 스키마 변경이 파급된다. 첫째는, 이 상위클래스 버전의 추가 변경연산 이전에, 스키마 관리자가, 현재 스키마 버전 CSV내에서 C의 상위클래스 S를 상위클래스에서 삭제했다가, 새로이 S를 C의 상위클래스로 추가하는 경우이다. 둘째는, CSV가 아닌 다른 스키마 버전내의 스키마 변경연산으로 인해 이미 $RiBS$에 원하는 결과가 나타나 있는 경우이다. 이 두 경우는 $RiBS$내의 모든 직/간접 상속관계 $ISA^*(RiBS)$내에 $(B(S),B(C))$의 포함여부를 검사해서 알 수있다. $(B(S),B(C))$가 $ISA^*(RiBS)$에 포함되어 있지 않은 경우, $B(S)$가 $B(C)$의 직접 상위클래스로 추가된다.

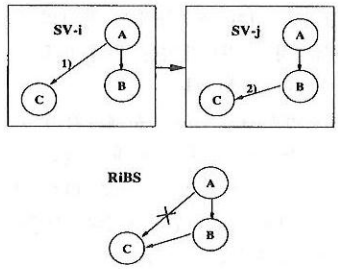

그림 3 상위클래스 버전의 추가

그런데, 이 시점에서 $RiBS$에는 중복 상속 관계가 존재할 수 있다. 즉, 어떤 경우에는 $RiBS$에서 $B(S)$의 직/간접 상위클래스중의 하나가 $B(C)$의 직접 상위클래스이어서 중복 상속 관계가 발생하게 된다. 예를 들어, 그림 3을 보자. 스키마 버전 SV_j이 SV_i에서 유도된 후, SV_i에서 클래스 버전 A가 클래스 버전 C의 직접 상위클래스로 추가되었다고 가정한다. 그런데, 이 후에 SV_j에서 클래스 버전 B의 상위클래스로 C를 추가할 경우, $RiBS$에는 중복 상속 관계가 발생한다. 이 경우에 $RiBS$ 모델에서는 $ISA(B(C),B(A))$를 제거하고 $ISA(B(C),B(B))$를 남겨둔다.

5) 단, 이미 클래스 계층구조상의 다른 경로를 통해서 상속받는 속성들은 제외.

6.2.3 새 클래스 버전 생성

이 연산은 CSV내에 새로운 클래스 버전 C를 생성한다. 이미 CSV에 같은 이름의 클래스 버전이 존재할 경우, 이 연산은 거부된다. 무결성 조건 3을 만족시키기 위해서, 이 연산은 C의 기본 클래스를 $RiBS$에 생성해야 한다. 생성되는 기본클래스는 클래스 버전 C의 각 상위클래스 버전들의 기본 클래스를 자신의 상위클래스로 갖는다. 생성된 기본 클래스는 $B(C)$로 세팅한다.

이 연산은 $RiBS$외에 다른 스키마 버전에도 영향을 미치게 된다. 클래스 버전 C와 이의 기본 클래스 $B(C)$는 이 연산에 의해 새로 생겨나기때문에, 다른 스키마 버전들에는 $B(C)$가 어떤 클래스 버전의 익스텐트 기본 클래스에도 속하지 않는다. 그런데, 무결성 조건 4에 의하면, 새로운 기본 클래스 $B(C)$는 모든 스키마 버전의 특정 클래스 버전의 익스텐트 기본 클래스에 포함되어야 한다. 따라서, CSV를 제외한 다른 스키마 버전들의 특정 클래스 버전의 익스텐트 기본 클래스에 포함시켜야 한다. 우선, 특정 스키마 버전에 클래스 버전 S에 해당하는 클래스 버전 S'이 존재할 경우에는, S'의 익스텐트 기본 클래스 $B^+(S')$에 $B(C)$를 추가한다. 그렇지 않을 경우에는, S의 기본클래스를 익스텐트 기본 클래스에 포함하고 있는 클래스 버전에 추가한다. 이 내용은 정형적 세만틱을 통해 더욱 분명하게 표현되기 때문에, 6.3절을 참조하기 바란다.

6.2.4 스키마 변경 연산의 정형적 세만틱

표 4는 앞에서 설명한 $RiBS$ 세가지 스키마 변경 연산들의 정형적 세만틱을 정리한 내용이다(나머지 연산들의 정형적 세만틱은 [17]에 정리되어 있다.) 표에서 $a(C,p)$를 제외한 연산들의 의미는 쉽게 이해할 수 있으므로, 본 절에서는 $a(C,p)$의 의미만 자세히 설명하겠다.

$a(C,p)$는 클래스 버전 C의 모든 하위클래스 버전들을 탐색하면서 p를 원조 속성 버전으로 갖는 각각의 속성 버전들에 대해, 자신들의 $RiBS$의 기본 속성을 세팅한다. 이 연산의 세만틱의 이해를 돕기위해, 2.2절에서 사용한 예를 보자. 그림 1에서 보여지듯이, 스키마 버전 SV2의 클래스 버전 C1′에 새로 추가된 속성 d는 클래스 버전 C3에 의해 상속된다. 이때 C3에서 상속된 속성 d의 경우도, C1′에 추가된 d와 마찬가지로, 기본 스키마 $RiBS$에 자신의 기본 속성을 필요로 한다. 그래서 그림 1의 예에서 $RiBS$의 클래스 BC3의 속성 d가 클래스 버전 C3의 속성 버전 d의 기본 속성으로 지정되어야 한다. 연산 $a(C1',d)$을 이용해서 이와 같은 기능을 수행할 수 있다. 본 논문에서는, 상속과 관련하여, 하나의 클래스 버전 C에서 정의된 지역 속성은 C의 모든 직/간접

하위클래스 버전으로 상속되는 전체 상속 세만틱을 가정한다. 그리고, 각 하위클래스들은 상속 속성에 대한 정보를 유지한다. 그런데, 이들 상속 속성 버전들도 자신들에 기본 속성을 필요로 한다. 좀더 정확히 표현하면, *a(C,p)*는 *C*의 각 하위클래스 버전 C_{sub}을 탐색하면서, *p*로부터의 상속 속성 *ip*의 기본 속성을 C_{sub}의 기본 클래스에서 찾아, 이를 *B(ip)*로 세팅한다. 이것이 가능한 이유는, *RiBS* 모델의 특성상 *C*의 모든 하위클래스 버전들의 각 기본 클래스는 *RiBS*에서 *B(C)*의 하위클래스이기 때문이다.

표 4 스키마 변경 연산의 정형적 세만틱

Operations	Semantics	
Drop an existing class version C	CSV	if C = Object then Reject else $B^*(C_{sup}) = B^*(C_{sup}) \cup B^*(C)$; for each subclass C_{sub} of C if $P(C_{sub}) = C$ then $P(C_{sub}) = P(C)$ else remove C from $P(C_{sub})$;
Add an edge to make S as a superclass of class version C	*RiBS*	if $B(S) \in P^*(B(C))$ then no action else add B(S) to P(B(C))
	CSV	add S to P(C) *a(C,p)*
Create a new class version C as a subclass of S	*RiBS*	create a base class bc
	CSV	set bc as B(C) add S to P(C) initialize $B^*(C)$ to {bc} for all $p \in LP(C)$, *a(C,p)*
	other SVs	$\exists C_j$ in SV, where $B(S) \in B^*(C_j)$, add bc to $B^*(C_i)$

7. 기타 사항

7.1 객체 변환

스키마 진화(schema evolution)에 있어 가장 중요한 이슈중의 하나는 객체 변환(object adaptation) - 즉, 스키마 변경의 영향을 받는 객체들의 포맷을 새로운 스키마에 맞게 바꾸는 작업 - 이다. 이와 관련하여 객체지향 데이타베이스 스키마 진화 관련 연구들[3,24,32]에 있어서 크게 두가지 접근 방식이 있었다. 첫번째 방식은 지연 갱신(deferred update)이라 불리는 방법인데, 각 객체의 포맷의 변경은 스키마의 변경 후 객체가 프로그램이나 질의를 통해서 실제로 접근되는 시점에 이루어진다[14,32]. 두번째 방식은 즉시 갱신(immediate one)이라 불리는 방법으로, 스카마 변경 연산이 이루어지는 시점에 영향을 받는 모든 객체들의 포맷 변환이 이루어진다.

본 논문에서 제안하는 *RiBS* 모델은 실제 스키마 변경연산과 관련한 세만틱의 정의에 중점을 두고 있기때문에, 객체의 변환에 관한 내용은 자세히 언급하지 않는다. 다만, *RiBS* 모델에서는, 객체베이스에 존재하는 물리적인 객체가 특정 스키마 버전내에서 발생하는 스키마 변경에 의해 영향을 받게 되는 경우, 위의 두가지 방식 모두를 객체 변환을 위해 사용할 수 있다.

7.2 스키마 버전 통합

6장에서 언급한 바와 같이, 스키마 버전 통합은 두개 혹은 그 이상의 이미 존재하는 스키마 버전들은 결합해서 하나의 새로운 스키마 버전을 만드는 과정이다. 그런데, 이 연산은 실제로 객체지향 데이타베이스의 관리 과정에 있어 유용하게 사용될 수 있다. 첫째로, 객체지향 데이타베이스의 초기 설계 단계이다. 대개 이 과정에서는 여러 명의 스키마 설계자들이 관여하게 되고 각각의 사람들은 데이타베이스의 다른 부분의 설계를 책임지게 된다. 그리고, 이들 부분 스키마를 일정한 시점이 지나서 합치게 된다. 둘째로, 어떤 사용자가 이미 존재하는 스키마중에서 자신이 관심이 있는 부분들만을 골라서 자신의 스키마를 재단할 필요가 있다.

우리는 *RiBS* 모델의 스키마 버전 통합에서 발생할 수 있는 여러 이슈들과 이들 이슈들을 해결하는 알고리즘을 고안했다[17]. 본 논문에서는 통합과정에 발생하는 충돌의 유형에 대해서만 간단히 언급하겠다. 스키마 버전 통합의 자세한 내용은 [17]을 참조하기 바란다.

7.2.1 스키마 버전 통합에서 발생하는 충돌

두개 이상의 스키마 버전들에서 하나의 일관된 스키마 버전을 생성하는 과정에서는 우선 다음의 두가지 유형의 이름 충돌이 발생한다.

1. 동음이의어(homonyms) 문제: 각기 다른 스키마(클래스) 버전에 속하는 두개 이상의 클래스(속성) 버전들이 이름은 같으나 서로 다른 기본 클래스(속성)를 가질 수가 있다. 이들을 동음이의어 클래스(속성) 버전이라 부른다.
2. 이음동의어(synonyms) 문제: 각기 다른 스키마(클래스) 버전에 속하는 두 개 이상의 클래스(속성) 버전들이 이름은 틀리지만 서로 같은 기본 클래스(속성)를 가질 수가 있다. 이들은 이음동의어 클래스(속성) 버전이라 부른다.

이와 같은 이름 충돌의 원인은, 스키마 버전 통합 이전에 각 스키마 버전내에서의 독립적인 스키마 변경(예를 들어, 클래스 버전 이름 변경, 새로운 클래스 버전의 생성) 연산들 때문이다.

이들 이름 충돌외에, *RiBS* 모델의 스키마 버전 통합 과정에서는 "익스텐트 이동 충돌"(extent migration conflict)이라 불리는 구조적인 충돌이 발생한다. 통합되는 두 스키마 버전에 대해, *RiBS*에 존재하는 하나의 기본 클래스 *bc*가 1) 어느 스키마 버전에도 *bc*를 기본클래스로 하는 클래스 버전이 없고, 2) 각 스키마 버전에서 서로 다른 기본 클래스를 갖는 클래스 버전의 익스

텐트 기본 클래스에 속할때, *bc*는 '익스텐트 이동 충돌을 갖는다'고 말한다. 예를 들어, 그림 4에서 기본 클래스 *C*는 스키마 버전 SV_i, SV_j를 통합할 때 익스텐트 이동 충돌을 갖는다. 첫째, SV_i나 SV_j에는 *C*를 기본클래스로 갖는 클래스 버전이 존재하지 않는다. 둘째, 기본 클래스 C는 SV_i에서는 $B^+(CV_m)$에 속하는 반면, SV_j에서는 $B^+(CV_n)$에 속한다. 그런데, CV_m와 CV_n는 각기 다른 기본 클래스를 갖는다. 이때 기본 클래스 *C*가 문제가 되는 것은 다음과 같은 이유이다. 새로운 스키마 버전에서는 CV_m와 CV_n에 대응하는 새로운 클래스 버전들이 생성되는데, 기본 클래스 *C*를 어느 클래스 버전의 익스텐트 기본 클래스에 포함시키느냐 하는 문제가 발생한다. 이 익스텐트 이동 충돌은 객체지향 데이타 모델의 다중상속때문에 발생한다.

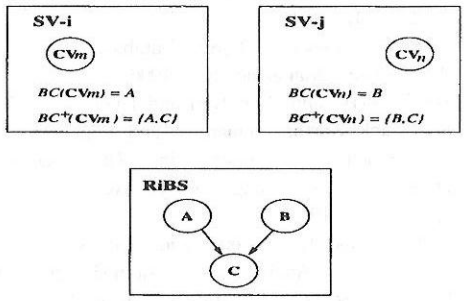

그림 4 익스텐트 이동 충돌

8. 관련 연구

본 *RiBS* 모델과 관련된 연구분야로는, 크게 객체지향 데이타베이스 뷰에 관한 연구[1,6,12,27]와 스키마 진화/버전에 관한 연구[3,10,13,20,21,22,24,26,32]를 들 수 있다. 그리고, 스키마 버전 통합과 관련된 분야로 데이타베이스 스키마 통합에 관한 연구[4,14,25]가 있는데, 본 논문에서는 스키마 버전 통합은 자세히 다루지 않은 관계로 생략하겠다.

8.1 객체지향 데이타베이스 뷰

관계형 데이타베이스의 뷰의 장점-예를 들어, 논리적 데이타 독립성 제공, 내용기반 보안기능-을 객체지향 데이타베이스에서도 이용하기 위해 객체지향 데이타 모델에서 뷰의 지원을 위한 연구가 진행되어 왔다. [1]에서는 O_{2} 객체지향 DBMS에서의 뷰 메카니즘을 다루고 있다. 이 뷰의 특징으로는 클래스 계층구조의 재구성과 가상 클래스(virtual classes)의 지원을 들수 있다. [27]에서는 전역(global) 스키마에서 사용자의 필요에 맞게 뷰 스키마를 정의할 수 있게 해주는 MultiView 방법론을 다루고 있다. [12]에서 객체/관계형 데이타베이스 시스템인 UniSQL에서 관계형 뷰의 의미에다 객체지향 개념들-상속, 메소드, 객체식별자-을 고려해서 객체지향 뷰를 정의하는 방법을 제안했다. [6]에서는 객체지향 뷰를 이용해서 스키마 진화연산을 시뮬레이션하는 방안을 제안했다.

RiBS 모델도 스키마 버전이 *RiBS*에 대한 논리적 뷰라는 점에서 이들 연구와 맥락을 같이한다. 그러나, 이들 방법과 *RiBS* 모델은 다음의 큰 차이가 있다. 즉, 기존 방법에서는, 사용자가 뷰의 스키마를 변경하고자 하는 경우에, 이전 뷰 스키마를 삭제한 후 뷰 스키마를 새로이 정의해야 하는 반면, *RiBS* 모델에서는 스키마 버전을 대상으로 원하는 스키마 변경을 직접 수행할 수 있다.

8.2 스키마 진화/버전 연구

김원 박사 등은 [13]에서 객체지향 데이타베이스를 위한 스키마 버전을 처음으로 제안했다. 이 연구는 기존 ORION 데이타모델의 객체 버전 모델을 확장한 것인데, 모델의 구성요소는 스키마 버전 관리 및 접근 범위에 관한 규칙들로 이루어져 있다. 특히 접근범위와 관련한 규칙은 각 스키마 버전에 보여지는 객체들의 범위를 제한하는 기능을 수행한다. 하지만, *RiBS* 모델에서는 모든 객체가 모든 스키마 버전에서 접근이 가능하다는 점이 차이가 있다. 그리고, 이 연구에서는 오직 하나의 스키마 버전에서만 새로운 스키마 버전을 유도할 수 있는 반면에, *RiBS*에서는 스키마 버전 통합을 지원함으로써 DAG(Direct Acylic Graph) 모양의 스키마 버전 유도를 지원한다.

스키마 버전에 관한 또 다른 연구로는 [26]을 들수 있다. 이 연구는 뷰를 통해서 스키마 진화를 지원하고, 스키마 버전들 사이의 객체 공유, 스키마 버전 통합의 지원 측면에서 *RiBS* 모델과 가장 유사하다. 하지만, 이 연구는 phantom 객체 참조와 같은 무결성에 대한 고려가 부족했고, 스키마 통합에 있어 발생하는 여러 이슈들에 대한 해결책을 제시하지 못하고 있다.

이들 스키마 버전식의 접근 방법이외에도 클래스 버전닝(Class Versioning)이라 불리는 몇몇 연구가 있었다[10,20]. 이들 연구에서는 버전닝의 대상이 전체 데이타베이스 스키마가 아니라 단위 클래스이다. 특히, [20]에서는 동적 객체 변환(dynamic instance conversion)이라는 메카니즘에 기반한 클래스 버전닝 시스템 CLOSQL을 제안했다. 이 동적객체변환 메카니즘은 하나의 객체를 여러개의 클래스 버전 인터페이스를 통해

서 접근할 수 있게 해주고, 접근하는 환경(context)에 따라 객체의 타입을 동적으로 결정한다. *RiBS*에 있는 객체의 타입이, 접근하는 현재 스키마 버전 *CSV*에서 요구하는 클래스 버전의 타입으로 결정된다는 점에서 이 연구와 맥락을 같이 한다. 그러나, 클래스 버전닝 방법은 스키마 관리자나 일반 사용자가 각 클래스의 여러 버전들을 조합해서 일관성있는 스키마를 생성해야 하는 부담이 있다[13].

객체지향 데이타베이스 시스템들의 등장 초기에서부터, 스키마 진화 기능을 제공해왔다[3,24,32]. 이들 연구들은 스키마 진화와 관련한 두가지의 중요한 이슈-즉, 1) 스키마 진화연산의 의미, 2) 스키마 변경후 객체의 포맷 변경-을 고려했다. 첫째 이슈에 대해서는 대개 다음과 같은 프레임워크로 해결책을 제시했다. 스키마에 관한 무결성 조건들을 정의하고, 이들 무결성 조건을 만족하는 범위에서 각 스키마 진화연산들의 의미를 정의한다. 이런 측면에서 *RiBS* 모델도 이 프레임워크의 확장이라고 볼수 있다. 하지만, 단순한 확장이 아니라, 스키마 버전및 *RiBS*와 관련한 몇가지의 무결성 규칙을 새로 제시했고, 기본적인 스키마 진화연산의 의미를 이들 무결성 규칙에 맞게 재정의하는 작업을 수행했다.

9. 결 론

객체지향 혹은 객체/관계형 데이타베이스 시스템들이 새로운 응용분야들(예를 들어, 웹(WWW), PCTE, Repository 등)의 복잡한 스키마 관리 요구를 만족시키기 위해서는 스키마 버전 기능의 제공이 필수적이다. 이를 위해, 본 논문에서는 객체지향 데이타베이스를 위한 스키마 버전 모델 *RiBS*를 제안했다. 각 스키마 버전은 기본 스키마 *RiBS*위에 클래스 계층구조 뷰로 정의되고, 사용자는 단지 스키마 버전을 대상으로 데이타베이스를 접근하게 된다.

각 스키마 버전은 직접 스키마 변경이 가능하고, 이 변경의 영향은, 경우에 따라, 자동적으로 *RiBS*로 전달된다. phantom 객체나 객체의 다중클래스화와 같은 이상현상(anomaly)을 막기위해 무결성 조건을 도입하고 각 스키마 버전 변경 연산들의 세만틱을 이들 무결성 조건을 보존하도록 정의했다. 마지막으로, 스키마 버전 통합 과정에서 발생하는 몇가지 유형의 충돌들을 설명하고 *RiBS* 모델의 해결책을 언급하였다.

본 연구와 관련하여 향후 두가지 방향의 연구를 수행하고자 한다 첫째로는, 다양한 스키마 재단을 위한 연산들을 도입하고 이들의 세만틱을 정의하는 작업이다. 예를 들어, 분할(partitioning), 클래스 통합(class merging), 동적 클래스(dynamic class)[1,11,23,30]와 유사한 연산들을 추가하는 작업이다. 이 작업은 보다 강력한 모델링 기능을 *RiBS*에 제공하리라 본다. 다음으로, 복합 객체에 대한 스키마 버전 기능의 제공을 위해 *RiBS* 모델의 각 구성요소를 확장하는 연구이다. 특히, 이 연구는 웹 뷰(WWW view)[15,31] 기능을 제공하는데 효과적일 것이다. 이와 관련하여, 객체간의 관계를 기반으로 역할-클래스(role-class)를 정의하는 연구[23]가 있는데, 여기서 제시된 각종 연산들과 프레임워크와 유사한 연구가 필요하다라고 본다.

참 고 문 헌

[1] Serge Abiteboul and Anthony Bonner, "Objects and Views," Proceedings of the ACM SIGMOD, pp. 238-247, 1991

[2] Thomas Atwood, "Object Databases Come of Age," Object Magazine, July, 1996

[3] Jay Banerjee and Won Kim and Hyoung-Joo Kim and Hank Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," Proceedings of the ACM SIGMOD, pp. 311-322, 1987

[4] C. Batini and M. Lenzerini and S. B. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," ACM Computing Survey, Vol. 18, No.4, pp. 323-364, 1986

[5] Philip A. Bernstein, "Repositories and Object Oriented Databases," Proceedings of BTW, pp. 34-46, 1997

[6] Elisa Bertino, "A View Mechanism for Object-Oriented Databases," Proceedings of Extending Database Technology, pp. 136-151, 1992

[7] R. G .G. Cattell(editor), "The Object Database Standard: ODMG-93," Morgan Kaufmann, 1996

[8] Francois Charoy, "An Object-Oriented Layer on PCTE," Technical paper available from http://gille.loria.fr:7000/oopcte/oopcte.html, 1994

[9] Richand Hull, "Relative Information Capacity of Simple Relational Database Schemata," Proceedings of ACM Principles of Database Systems, pp. 97-109, 1984

[10] H. J. Kim, "Issues in Object Oriented Database Schema," PhD dissertation (University of Texas at Austin), 1988

[11] Won Kim, "Introduction to Object Oriented Databases," MIT press, 1991

[12] Won Kim, "Modern Database Systems: The Object Model, Interoperability, and Beyond," ACM Press, 1995

[13] Won Kim and H.T. Chou, "Versions of Schema for Object-Oriented Databases," Proceedings of Very Large Databases, pp. 148-159,1988
[14] Won Kim and Jungyun Seo, "Classifying Schematic and Data Heterogeneity in Multidatabase Systems," IEEE Computer, Vol. 24, No.12, pp. 12-18, 1991
[15] David Konopnicki and Oded Shmueli, "W3QS: A Query System for the World-Wide Web," Proceedings of Very Large Databases, pp. 54-65, 1995
[16] Sven-Eric Lautemann, "A Propagation Mechanism for Populated Schema Versions," Proceedings of International Conferences on Data Engineering, pp. 67-78, 1997
[17] Sang-Won Lee and Hyoung-Joo Kim, "A Model of Schema Versions for Object-Oriented Databases, based on the concept of Rich Base Schema," Information and Software Technology, Vol. 40, No. 3, pp. 157-173, 1998
[18] Mary E.S. Loomis, "Object Database - Integrator for PCTE," Journal of Object Oriented Programming, May, 1992
[19] R. J. Miller and Y. E. Ioannidis and R. Ramakrishnam, "The Use of Information Capacity in Schema Integration and Translation," Proceedingd of Very Large Databases, pp. 120-133, 1993
[20] Simon Monk and Ian Sommerville, "Schema Evolution in OODB Using Class Versioning," SIGMOD Records, Vol. 22, No. 3, 1993
[21] Object Design, Inc., "ObjectStore Technical Overview, Release 3.0," 1994
[22] Objectivity, Inc., "Schema Evolution in Objectivity/DB," White paper available from http://www.objy.com/ObjectDatabase/WP/Schema/schema.html.
[23] M. P. Papazoglou and B. J. Kramer, "A Database Model for Object Dynamics," VLDB Journal, Vol. 6, No. 2, pp. 73-96, 1997
[24] D. Jason Penney and Jacon Stein, "Class Modifications in the GemStone Object-Oriented DBMS," Proceedings of OOPSLA, pp. 111-117, 1987
[25] Evaggelia Pitoura and Omran Bukhres and Ahmed Elmagramid, "Object Orientation in Multidatabase Systems," ACM Computing Survey, Vol. 27, No. 2, pp. 141-195, 1995
[26] Y.G. Ra and Elke A. Rundensteiner, "A Transparent Object-Oriented Schema Change Approach Using View Evolution," Proceedings of International Conference on Data Engineering, pp. 165-172, 1995
[27] Elke A. Rundensteiner, "MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases," Proceedings of Very Large Databases, pp. 187-198, 1992
[28] Avi Silberschartz and Mike Stonebraker and Jeff Ullman, "Database Research: Achivements and Opportunities Into the 21st Century," Report of an NSF Workshop on the Future of Database Systesms Research, 1995
[29] Antero Taivalsaari, "On the Notion of Inheritance," ACM Computing Survery, Vol. 28, No. 3, pp. 438-479, 1996
[30] Roel Wieringa and Wiebren de Jonge and Paul Spruit, "Using Dynamic Classes and Role Classes to Model Object Migration," Theory and Practice of Object Systems, Vol. 1, No. 1, pp 61-83, 1995
[31] Jack Jingshuang Yang and Gail E. Kaiser, "An Architecture for Integrating OODBs with WWW," Columbia University Tech-Report CUCS-004-96, 1996
[32] Roberto Zicari and Fabrizio Ferrandina, "Schema and Database Evolution in Object Database Systems," In Part6, Advanced Database Systems, Morgan Kaufmann, 1997

이 상 원
제 25 권 제 7 호(B) 참조

김 형 주
제 25 권 제 2 호(B) 참조