객체지향 데이타베이스의 복합 객체를 위한 스키마 버전 모델

(A Schema Version Model for Composite Objects in Object-Oriented Databases)

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

요 약 본 논문에서는 복합객체 계층구조의 재구성을 지원하는 객체지향 데이터베이스 스키마 버전 모델을 제안한다. 이 모델은 풍부한 기본 스키마(Rich Base Schema) 개념에 기반한 스키마 버전 모델 RiBS를 확장한다. RiBS 모델에서 각 스키마 버전은 하나의 기본스키마에 대한 갱신가능한 클래스 계층구조 뷰이고, 이 기본스키마는 모든 스키마 버전들에서 필요로 하는 스키마 정보를 갖고 있다. 본 논문에서는 스키마 버전의 복합객체 계층구조의 재구성을 위한 스키마 진화 연산들을 도입하고, 이 연산들의 의미를 설명한다. 그리고, 이 연산들을 통해 재구성된 복합객체 계층구조에 대한 질의의 처리방안을 다룬다. 또한, 둘 이상의 스키마 버전 통합시 발생하는, 복합객체 재구성 연산들에 의한 충돌현상을 설명하고 해결책을 제시한다. 본 논문의 독창성은 1) 복합객체 계층구조의 재구성을 위한 연산들을 최초로 도입한 점과 2) 확장된 RiBS 모델이 객체지향 데이타베이스의 데이타 독립성(data independence)을 제공한다는 점이다.

Abstract In this paper, we propose a schema version model which supports restructuring composite objects in object-oriented databases. This model extends the schema version model RiBS based on the concept of Rich Base Schema. With RiBS model, each schema version is in the form of updatable class hierarchy view over one base schema, called RiBS, which has richer schema information than any existing schema version in the database. In this paper, we introduce new operations for restructuring composite object hierarchy in schema versions, and explain their semantics. And we touch upon the ways to transform queries posed against restructured composite object hierarchy to queries against base schema. In addition, we identify several types of conflicts in merging schema versions, which result from the restructuring operations, and provide our solution. The originality of this paper lies in that 1) we introduce several new operations to restructure composite object hierarchy, and 2) this extended RiBS model operations raise the level of data independence of object oriented databases.

1. 서 론

객체지향 데이타베이스 시스템(Object Database Management Systems)과 관계형 데이타베이스 시스템의 큰 차이 중의 하나는 스키마 변경에 대한 지원이다. 초창기 객체지향 데이타베이스의 주된 응용분야인 CAD/CAM, CASE 등에서 동적인 스키마 변경에 대한 요구사항이 많았다. 따라서, 객체지향 데이타베이스 스키마 변경에 관한 많은 연구가 진행되었고 [1, 2], 현재 대부분의 상용 객체지향 데이타베이스 시스템들(예를 들어, O2 [2], ObjectStore [3], Objectivity [4])에서 기본적인 스키마 변경 연산을 온라인으로 지원하고 있다. 이와 관련하여, 최근의 주목할 만한 사실은 객체지향 데이타베이스 표준인 ODMG 2.0 [5]에서 스키마 정보 접근을 위한 인터페이스를 포함시켰고, 곧 스키마 변경 연

· 본 연구는 과학기술처 프로젝트 "웹 브라우저 서버를 위한 객체지향 콤포넌트 기술개발"과 서울공대 공학연구소, 학술진흥재단의 "신진연구인력 연구장려금 지원" 사업의 일부 지원에 의한 것임.
† 정 회 원 한국오라클 연구원
swlee@candy.snu.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학과 교수
hjk@im4u.snu.ac.kr
논문접수 : 1998년 7월 9일
심사완료 : 1999년 2월 22일

산을 추가한다는 점이다[1].

그런데, 이들 방법에서는 한 스키마 변경 연산이 끝나면 이전 스키마는 더이상 존재하지 않는다. 즉, 한 시점에 하나의 스키마만 데이타베이스 내에 존재한다. 이와 같은 메카니즘은 몇 가지 단점이 있다 [6]. 첫째, 스키마 변경 이전의 스키마를 대상으로 작성된 응용 프로그램이 더 이상 동작하지 않게 된다. 둘째, 모든 사용자가 하나의 스키마를 공유하기 때문에 한 사용자에 의한 스키마 변경은 다른 사용자들의 뷰를 변경시킨다. 스키마 버전 기능은 이와 같은 문제점들을 해결하기 위해 도입되었다 [7, 8].

더군다나, WWW [9], Repository [10], PCTE(Portable Common Tool Environment) [11] 등의 새로운 객체지향 데이타베이스 응용분야에서 스키마 버전 기능을 필요로 하고 있다. Atwood [9]는 스키마에 대한 동적인 요구사항을 갖는 WWW 응용들은 스키마 버전 기능을 반드시 필요로 함을 지적하고 있다. Bernstein [10]은 Repository 환경에서도 역시 스키마 버전의 관리가 필수적이라고 기술하고 있다. Loomis [12]는 PCTE 환경에서 각 도구(tool)들은 각기 서로 독립적으로 진화하는 스키마 버전들을 대상으로 동작해야 한다고 지적했다. 이들 사실로 알 수 있듯이, 객체지향 데이타베이스가 이들 새 응용분야들을 효과적으로 지원하기 위해서는 스키마 버전 기능의 지원이 필수적이다. 상용 객체지향 데이타베이스 시스템인 POET은 초보적인 형태의 스키마 버전 기능을 지원하고 있다 [13].

객체지향 데이타베이스에서의 스키마 버전 지원에 관한 몇몇 연구들이 있었지만 [7, 14, 15], 아직까지 만족할 만한 해결책이 없다. 특히 이들 연구는 주로 상속과 관련된 클래스 계층구조를 대상으로 한 스키마 버전을 다루고 있다. 그런데, 객체지향 데이터모델에서, 상속개념과 더불어 중요한 개념인 복합객체(composite objects)를 위한 스키마 진화 및 스키마 버전에 관한 체계적인 연구는 지금까지 없었다. 실제로 앞서 언급한 새로운 객체지향 데이타베이스 응용분야들은 복합객체 재구성과 관련한 스키마 버전 기능을 필요로 하고 있다.

우리는 [16]에서 풍부한 기본 스키마(rich base schema)와 갱신 가능한 클래스 계층구조 뷰(updatable class hierarchy view) 개념에 기반한 단순하면서 강력한 스키마 버전 모델 *RiBS*를 제시했다. 본 논문에서는 이 *RiBS* 모델을 확장하여 복합객체 계층구조의 스키마 재구성을 지원하는 방안을 제시한다. 본 논문의 기여점은 복합객체 재구성을 위한 스키마 진화 연산들을 최초로 제안하고, *RiBS* 모델의 확장을 통해서 복합객체 계층구조의 재구성을 효과적으로 지원한다는 점이다. 이 확장된 *RiBS* 모델은 사용자들로 하여금 데이타베이스의 스키마를 자신의 필요에 맞도록 손쉽게 재단(customization)할 수 있게 해준다. 즉, 데이타베이스의 논리적 스키마 위에 사용자가 필요로 하는 개념적 스키마를 지원함으로써 객체지향 데이타베이스에 데이타독립성(data independence)을 제공한다. 객체지향 데이타베이스가, 많은 장점들 - 예를 들어, 성능, 모델링 능력 등 - 에도 불구하고, 실제 응용영역에서 활발히 도입되지 않는 이유 중의 하나는 융통성있는 스키마 관리기능의 부재에 있다. 따라서, 본 논문의 *RiBS* 모델과 같은 스키마 버전 기능은 새로운 데이타베이스 응용분야에서 객체지향 데이타베이스의 경쟁력을 높여줄 것이다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문에서 가정하고 있는 객체 모델을 기술하고, *RiBS* 모델을 본 논문의 이해에 필요한 범위에서 요약한다. 3장에서는 복합객체 계층구조의 재구성을 위해 새로 제안하는 스키마 진화 연산들과 그 의미를 설명한다. 그리고, 이들 연산을 이용해 재구성된 복합객체 계층구조를 포함하는 스키마 버전에 대한 질의 처리와 스키마 버전 통합 과정을 4장, 5장에서 각각 다룬다. 6장에서 관련 연구에 대해 살펴본 후, 7장에서 논문내용을 정리하고 향후 연구방향에 대해 언급한다.

2. 객체 모델과 *RiBS* 모델

본 장에서는 본 논문에서 가정하는 객체 모델과 *RiBS* 모델에 대해 논문의 전개에 필요한 범위에서 설명한다. 이들 모델의 자세하고 정형적인 내용은 관련 논문 [16]을 참조하기 바란다.

2.1 객체 모델

객체지향 데이타베이스의 스키마는 클래스들간의 상속관계에 의한 클래스 계층구조(class hierarchy)이다. 클래스는 자신에 속하는 객체들을 위한 공통의 속성(attributes)과 메소드(methods)를 정의한다. 하위클래스(subclass)는 자신의 상위클래스(superclass)에서 모든 속성을 상속받는다. 그리고, 한 클래스에 직접 속하는 소속 객체들의 집합을 해당 클래스의 익스텐트(extent)라 한다. 각 객체는 자신의 유일한 객체식별자(object identifier)를 갖는데, 객체에 대한 참조는 이 객체식별자를 통해서 이루어진다.

클래스의 특정 속성의 도메인은 다른 사용자 정의 클

1) 추가될 스키마 변경 연산들은 참고문헌 [14]에 자세히 나와 있다.

래스를 가질 수 있고, 이 때 객체의 해당 속성 값은 참조되는 객체의 객체식별자를 갖는다. 이는, 관계형 데이타베이스가 값에 의해 참조관계를 암시적(implicit)으로 표현하는 것과 달리, 객체식별자를 통해 참조관계를 명시적(explicit)으로 표현하는 객체지향 데이타모델의 특징이다.

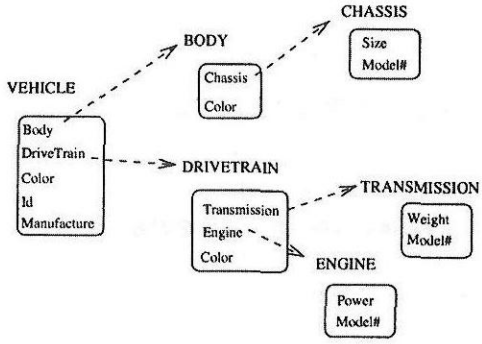

그림 1 VEHICLE 복합객체 계층구조

클래스들간의 상속관계에서 생기는 클래스 계층구조처럼, 참조관계에 있는 클래스들은 복합객체 계층구조(composite object hierarchy)를 이룬다[2]. 그림 1은 본 논문에서 사용할 복합객체 계층구조의 예를 보이고 있다. 이 복합객체 계층구조에서 클래스 VEHICLE은 루트 클래스(root class)이고 이 외의 클래스들은 VEHICLE 클래스의 부분 클래스(part class)가 된다. 이 예에서는 이해를 돕기 위해 클래스의 이름과 해당 클래스를 참조하는 속성의 이름을 동일하게 사용하고 있다. 점선화살표가 참조관계를 표시하고 있다. 클래스내의 속성은 단말(leaf)및 비단말(non-leaf) 속성으로 나눌 수 있는데, 이는 속성의 도메인에 따라 결정된다. 즉, 정수, 문자열같은 기본클래스를 도메인으로 갖는 속성을 단말 속성이라 하고 사용자 정의 클래스를 도메인으로 갖는 속성을 비단말 속성이라 한다.

객체지향 데이타모델에서 복합객체 계층구조와 관련하여 중요한 개념 중의 하나가 경로식(path expression)이다 [5, 17]. 이는 복합객체 계층구조상의 한 경로를 표현하는 식으로, 특히 질의에 있어서 간단하게 질의 조건을 명시할 수 있게 해준다. 다음 질의는 경로식을 사용해서 그림 1의 VEHICLE 객체 중에서 엔진 파워가 100마력 이상인 차를 선택한다.

```
select Car
from VEHICLE Car
where Car.DriveTrain.Engine.Power >= '100hp';
```

2.2 *RiBS* 모델

RiBS 모델의 아키텍처는 그림 2에서 보여지듯이, 크게 스키마 버전 계층, 기본스키마 *RiBS* 계층, 그리고 물리적인 객체들이 존재하는 객체베이스로 구성된다. 스키마 버전 계층과 기본스키마 *RiBS*는 모두 클래스 계층구조로 표현된다. 하나의 스키마 버전은 기본스키마 *RiBS* 위에 정의되는 클래스 계층구조 뷰이고, *RiBS* 계층은 현재 존재하는 모든 스키마 버전에서 필요로 하는 모든 스키마 정보 - 클래스, 상속관계, 속성 등 - 를 유지한다. 사용자는 특정 스키마버전(사용자가 현재 접근하는 스키마 버전을 현재 스키마 버전(Current Schema Version)이라 한다.)을 통해서 데이타베이스를 접근하게 된다. 이 때 스키마 버전을 통해 보여지는 객체는 실제로 기본스키마 *RiBS*에 속하는 객체베이스의 물리적 객체에서 유도되는 논리적인 객체이다. 특정 스키마 버전을 대상으로 작성된 프로그램이나 질의는 전처리 과정을 거쳐 *RiBS* 계층에 대한 질의나 프로그램으로 변환되어 수행된다.

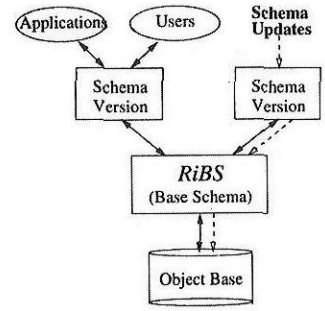

그림 2 *RiBS* 모델의 아키텍처

스키마 진화와 관련하여, *RiBS* 모델에서 사용자는 현재 스키마 버전에서 자기가 원하는 임의의 스키마 갱신 연산을 수행할 수 있다. *RiBS* 모델은 Orion [6]에서 제시한 클래스 계층구조의 진화에 필요한 모든 연산을 지원하고 있다. 그림 2에서 보여지듯이, 특정 스키마 버전에 대한 스키마 진화 연산은 해당 스키마 버전의 스키마 정보를 갱신하고, 필요한 경우 그 영향이 사용자의 개입없이 자동적으로 기본스키마 *RiBS*와 객체베이스에 존재하는 객체들로 파급된다.

2) [15] 등에서는 복합객체 용어는 part-of라는 특수한 참조관계에 한해서 사용한다. 본 논문에서는 일반적인 참조관계에 있는 클래스들도 복합객체 계층구조에 포함시킨다.

한편, 그림 2에서 주의해서 볼 것은 특정 스키마 버전에 대한 스키마 변경이 다른 스키마 버전에는 영향을 미치지 않는다는 점이다. 즉, 기본 *RiBS* 모델에서 지원하는 모든 스키마 변경 연산들은 해당 스키마 버전과 *RiBS* 계층 스키마에만 영향을 미친다. 만일에 *RiBS* 계층의 스키마가 변경되는 경우에, *RiBS* 모델에서 각 스키마 버전들은 공통의 물리적 스키마 *RiBS*에 대한 논리적인 뷰 형식으로 구성되어 있기 때문에, 변경된 *RiBS* 계층과 다른 스키마 버전들 사이의 사상 정보만을 바꾸어 주면 된다. 따라서, 다른 스키마 버전들의 스키마 정보는 아무런 영향을 받지 않는다. 또한, 다른 스키마 버전들을 접근하는 프로그램이나 사용자들의 경우 특정 스키마 버전에서 발생한 스키마 진화 연산에 전혀 영향을 받지 않게 된다.

RiBS 모델의 가장 큰 특징은 기본스키마에 클래스 계층구조 뷰 형식으로 정의된 스키마 버전에 대해 직접적인 스키마 변경연산을 허용한다는 점이다. 즉, *RiBS* 모델에서 스키마 버전은 갱신가능한 클래스 계층구조 뷰(updatable class hierarchy view)이다. 이러한 *RiBS* 모델의 특징은 기존 모델들의 단점 1) 기존 스키마 버전 모델들 [7, 15]에서 발생하는 인스턴스 객체의 중복에 따른 기억공간의 오버헤드, 2) 기존 뷰 방식 [18]에서 발생하는 스키마 변경 연산 및 스키마 변경 과정의 복잡성, 그리고 (3) 클래스 버전화 모델 [14]의 일관성 있는 스키마 관리의 부담 등을 모두 극복한다 [16]. 첫째, 모든 인스턴스 객체들은 *RiBS*라는 공통 스키마의 물리적 객체로 존재하기 때문에 각 스키마 버전들 사이의 인스턴스 객체를 중복적으로 관리할 필요가 없다. 또한, 뷰 형식의 스키마 버전에 대한 모든 스키마 변경 연산을 직접적으로 허용하기 때문에, 원하는 스키마 변경의 효과를 얻기 위해 기존 스키마 버전을 삭제하고 새로운 스키마 버전을 정의해야 하는 번거로움을 없애준다. 마지막으로, 클래스 버전화 방법에서는 원하는 스키마 버전을 만들기 위해 각 클래스마다 존재하는 클래스 버전들을 조합해야만 하는 관리의 부담이 있는 반면, *RiBS* 모델은 클래스 계층구조를 갖는 스키마 버전에 대해 직접 스키마 변경 연산을 수행하기 때문에 이와 같은 관리 부담이 전혀 없다.

RiBS 모델에서 사용자는 이미 존재하는 스키마 버전(들)에서 명시적으로 새로운 버전을 유도해야 한다. 둘 이상의 스키마 버전들에서 새로운 스키마 버전을 유도하는 과정을 스키마 버전 통합(schema version merging)이라 한다. 이 스키마 버전 통합과정에서 서로 다른 진화과정을 거친 스키마 버전들 사이의 여러 가지 충돌현상(conflicts)이 발생하는데, 이는 5장에서 자세히 다루겠다. 스키마 버전 계층은 스키마 버전들 사이의 유도관계에 의해 스키마 버전 유도 그래프를 이룬다. 이상에서 설명한 클래스 계층구조에 대한 스키마 버전의 진화를 지원하는 *RiBS* 모델을 기본 *RiBS* 모델이라 부르겠다. 기본 *RiBS* 모델의 자세한 내용에 관심이 있는 독자는 [16]을 참조하기 바란다. 반면에, 본 논문에서 복합객체 계층구조의 재구성을 위해 추가한 연산들을 포함하는 모델을 확장 *RiBS* 모델이라 부른다.

3. 복합객체 계층구조 재구성

본 장에서는 복합객체 계층구조의 재구성을 위한 스키마 변경 연산들을 설명한다.

3.1 복합객체 계층구조 재구성 연산들

다음은 스키마 버전의 복합객체 계층구조 재구성을 지원하는 스키마 진화 연산들이다. 처음 두 연산은 기본 *RiBS* 모델의 이미 제공하는 스키마 변경연산들인데, 이들도 일종의 복합객체 재구성 연산으로 재해석할 수 있기 때문에 포함시켰다. 나머지 네 가지 연산은 본 논문에서 새로 제안하는 스키마 변경 연산들이다.

1. 부분 클래스의 추가
2. 부분 클래스의 삭제
3. pull : 부분 클래스의 속성을 루트 클래스로 이동
4. unnest : 부분 클래스의 정보를 루트 클래스의 속성들로 모델링
5. nest : 부분 클래스들의 임의의 속성들을 임의의 가상 클래스로 모델링
6. move : 부분클래스의 속성을 다른 부분 클래스로 이동

부분 클래스의 추가 연산을 제외한 나머지 연산들은 스키마 버전 계층에만 영향을 미치고 *RiBS* 계층에는 영향을 주지 않는다. 이들 연산은 주로 현재 스키마 버전내에서 복합객체 계층구조를 사용자의 관점에 맞게 재구성하는 데 목적이 있기 때문에, *RiBS* 계층에서 새로운 스키마 정보의 추가는 필요가 없고, *RiBS* 계층과 변화된 스키마 버전 사이의 스키마 정보 사상(mapping)만을 변화시켜주면 된다.

3.2 부분 클래스의 추가

복합객체 계층구조에 새로운 부분 클래스의 추가는 기본 *RiBS* 모델에서 제공하는 클래스 계층구조상의 스키마 진화 연산인 속성 추가(add attribute)를 통해서 가능하다 [16]. 즉, 복합객체 계층구조에 추가할 클래스를 새로운 속성의 도메인으로 설정하고 이 속성을 복합객체 계층구조내의 특정 클래스에 추가한다. 이때 추가

되는 새로운 속성에 대응하는 기본 속성이 *RiBS* 계층에 추가되어야 한다.

3.3 부분 클래스의 삭제

복합객체 계층구조로부터 특정 부분 클래스를 삭제하고자 하는 경우에는 기본 *RiBS* 모델의 스키마 진화 연산인 속성 삭제(drop attribute)를 통해서 가능하다. 즉, 삭제될 부분클래스를 도메인으로 갖는 복합객체 계층구조상의 속성을 소속 클래스에서부터 삭제한다. 이 경우, 부분 클래스의 추가와는 달리, 해당 속성 정보가 단지 현재 스키마 버전 계층에서만 사라지고, 이에 대응하는 *RiBS* 계층의 기본 속성은 그대로 남아있게 된다.

3.4 복합객체 재구성 연산: pull

객체지향 데이타모델에서는 클래스 계층구조에서 상속을 통해 상위클래스에 정의된 속성이 하위클래스로 상속된다(우리는 이를 클래스 계층구조상에서 아래방향 상속(*downward inheritance*)라 부르겠다.). 이와 유사하게, 복합객체 계층구조에서 부분클래스의 특정 속성은 개념적으로 루트클래스의 속성으로도 볼 수 있다(이를 아래방향 상속과 구분해서 윗방향 상속(upward inheritance)이라 부르겠다.)

어떤 사용자는 복합객체 계층구조상에서 부분클래스에 정의된 속성을, 속성에 대한 접근의 편의 등의 이유로, 마치 루트클래스에 정의된 속성으로 보고자 하는 필요가 생긴다. 다음 연산은 이 기능을 제공한다.

pull path_expression [**as** new_attribute_name] **in class** class_name;

위 연산의 의미는 루트클래스 class_name을 가진 복합객체 계층구조에서 경로식 path_expression상의 마지막 속성을 루트클래스의 속성으로 이동하는 연산이다.

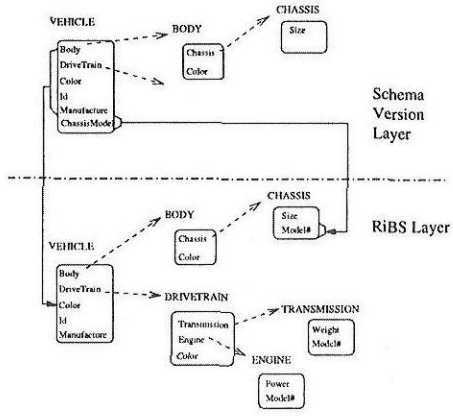

그림 3 pull 연산이후의 VEHICLE 복합객체 계층구조

만일 이 속성의 이름이 루트 클래스에 지역적으로(locally) 정의된 속성과 이름이 충돌하는 경우에 이 연산은 취소된다. 윗방향 상속되는 속성에 대해, 이름 충돌을 피하거나 좀 더 의미있는 이름을 부여할 수 있도록 as절을 제공한다. 다음은 그림 1의 복합객체에서 차체 샤시의 모델 번호를 루트 클래스로 이동하는 연산의 예이다.

pull Body.Chassis.Model# **as** ChassisModel **in class** VEHICLE;

그림 3은 위 연산의 결과를 보여주고 있는데, pull 연산 이전에 스키마 버전 계층의 VEHICLE 복합객체 계층구조는 *RiBS* 계층의 복합객체 계층구조와 동일하다고 가정한다. 그림에서 pull 연산의 결과로 VEHICLE에는 ChassisModel이라는 속성이 추가적으로 생겨나게 된다. 이 때 ChassisModel의 도메인은 Body.Chassis.Model#과 동일하다. 반면에, CHASSIS 클래스에서 속성 Model#는 삭제되는데, 이유는 뒤에서 설명하겠다.

VEHICLE 클래스에 새로운 속성 ChassisModel이 추가된다는 측면에서 기본 *RiBS* 모델의 속성 추가 연산과 유사하다. 그러나, 중요한 차이점은, 속성 추가의 경우 추가되는 속성에 대응하는 기본 속성이 *RiBS* 계층의 대응 클래스에 물리적으로 추가되는데 반해, pull 연산의 경우 해당 속성의 정의만 현재 클래스 버전에 추가되고, *RiBS* 계층에는 아무런 변화도 생기지 않는다. 해당 속성의 값은 원래 경로식에 해당하는 값으로부터 유도된다.

RiBS 모델에 대한 독자들의 이해를 돕기 위해, 그림 3을 이용해서 스키마 버전 계층과 *RiBS* 계층 사이의 스키마 정보사이의 관계를 살펴본 후, 논문의 내용을 전개하겠다. 그림에서 스키마 버전 계층의 클래스(속성) 버전은 *RiBS* 계층에 자신의 대응 클래스(속성)를 갖는데, 이를 클래스(속성) 버전의 **기본 클래스(속성)**라 한다. 예를 들어, 스키마 버전 계층의 클래스 버전 VEHICLE의 기본 클래스는 *RiBS* 계층의 VEHICLE이고, VEHICLE 클래스 버전의 Color 속성 버전의 기본 속성은 기본 클래스 VEHICLE의 Color이다. 클래스 계층구조에 대한 스키마 진화연산을 지원하는 기본 *RiBS* 모델에서는 모든 속성 버전의 기본 속성은 해당 속성버전의 클래스 버전의 기본클래스에 속해있다. 그런데, pull 연산을 포함한 본 논문에서 제안하는 복합객체 재구성 연산들은 특정 속성 버전이 자신의 기본 속성으로, 해당 속성 버전이 속한 클래스 버전의 기본 클래스가 아닌, *RiBS* 계층의 다른 기본 클래스에 정의된 속성을 가질 수 있다. 예를 들어, 그림 3의 VEHICLE 클래스

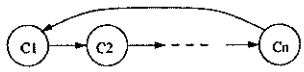

그림 4 사이클을 포함하는 복합객체 계층구조

버전의 ChassisModel의 기본 속성은 기본 클래스 CHASSIS의 Model#인데, ChassisModel의 기본 속성이 ChassisModel이 속한 클래스 버전 VEHICLE의 기본 클래스가 아니다. 용어 클래스(속성) 버전과 기본 클래스(속성)을 구분해서 사용해야 하지만, 본 논문에서는 설명의 편의상 구분할 필요가 없는 경우 단순히 클래스(속성)로 사용하겠다.

다시 pull 연산의 내용에 대해 계속 알아보자. 위의 예에서 VEHICLE 클래스로 위방향 상속된 Model#는 단말 속성이지만, 다음과 같이 비단말 속성에 대해서도 pull 연산은 유효하다.

pull DriveTrain.Engine **in class** VEHICLE;

또한 그림 4와 같이 사이클을 포함하는 복합객체 계층구조의 경우에 있어서 사이클을 포함하는 경로식에 대해서도 pull 연산이 가능하다.

그림 5는 사이클을 포함하는 복합객체 계층구조의 특별한 형태인 재귀적(recursive) 복합객체 클래스 PERSON에 대해 Father 속성에 대해 다음의 pull 연산을 적용해서 GrandFather 속성을 생성하는 예를 보이고 있다.

pull Father.Father **as** GrandFather **in class** PERSON;

pull 연산으로 변경된 루트클래스는 관계형 데이터베이스에서의 뷰와 비슷한 개념이다. 예를 들어, 관계형 데이타베이스 관점에서 pull 연산으로 변경된 위의 VEHICLE 클래스는 VEHICLE, BODY, 그리고 CHASSIS 테이블을 조인해서 VEHICLE 테이블의 모든 애트리뷰트와 CHASSIS 테이블의 Model#를 선택한 뷰로 볼 수 있다. 그런데, pull 연산과 관계형 모델의 뷰와의 차이점은 값의 갱신가능성(updatability)이다. 관계형 데이터베이스에서는 두 개 이상의 테이블에 대한 조인 뷰에 대해서는, 뷰 갱신의 모호성 문제 때문에, 갱신을 허용하지 않는다 [19]. 반면에 pull 연산에 의해 변경되는 루트 클래스의 경우, 객체지향 데이타모델의 특성상 복합객체를 구성하는 각 객체들이 객체식별자를 통해 연결되어 있기 때문에 관계형 모델의 뷰 갱신모호성과 같은 문제는 없다.

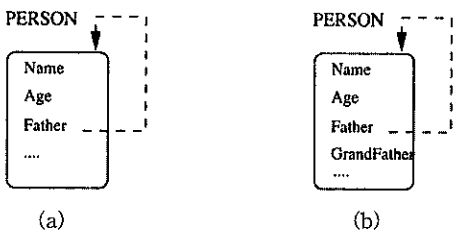

그림 5 재귀적 복합클래스에 대한 pull 연산 예

마지막으로, 그림 3의 예에서 pull 연산의 결과 클래스 CHASSIS에서 Model#라는 속성을 삭제하는 이유를 알아보자. 이는, RiBS 모델의 원칙상 [16], *RiBS* 계층의 특정 정보가 스키마 버전내에서 두 개 이상의 속성 버전으로 표현되는 것을 방지하기 위한 것이다. 이는 본 논문에서 제안하는 모든 연산에 적용된다.

3.5 복합객체 재구성 연산: unnest

이 절에서는 앞의 pull 연산을 좀 더 일반화한 연산인 unnest 연산에 대해 설명하겠다. 이 연산은 부분클래스 형태로 모델링된 정보를 루트클래스의 속성들로 표현하게 해준다.

unnest path_expression **in class** class_name;

위의 구문에서 경로식 path_expression의 마지막 속성의 도메인에 해당하는 특정 클래스의 모든 속성들을 루트클래스의 속성으로 윗방향 상속시킨다. 이를 위해서는 경로식의 마지막 속성은 반드시 비단말 속성이어야 한다. 다음은 그림 1의 복합객체에서 ENGINE 클래스의 정보를 VEHICLE 클래스의 속성들로 모델링하도록 unnest 연산을 적용하는 예를 보이고 있다.

unnest DriveTrain.Transmission **in class** VEHICLE;

이 연산의 결과 VEHICLE 복합객체 계층구조는 그림 6과 같은 상태로 변한다. 즉, TRANSMISSION의 모든 속성들의 이름과 도메인 정보는 루트클래스 VEHICLE로 그대로 상속되고, DRIVETRAIN의 속성 Transmission과 TRANSMISSION 클래스는 현재 스키마 버전에서 사라지게 된다.

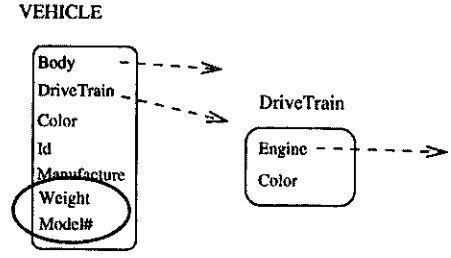

그림 6 unnest 연산 이후의 VEHICLE 클래스

그런데, 그림 6의 예에서 윗방향 상속된 속성 Weight의 경우 실제로 차체의 무게가 아니라 Transmission의 무게이기 때문에 이 속성의 이름을 바꿀 필요가 있다. 이는 기본 *RiBS* 모델의 속성 이름 변경연산을 이용해서 변경할 수 있다.

unnest 연산은 부분 클래스의 속성들을 루트클래스로 이동하게 한다는 점에서 pull 연산과 비슷하다. 그러나, 이 두 연산은 의미적으로 다음과 같은 차이가 있다. unnest 연산은 사용자가 복합객체 계층구조상에서 클래스로 모델링된 정보를 없애고 해당 정보를 다른 클래스의 속성들로 모델링하는 반면, pull 연산은 단지 특정 속성만을 다른 클래스의 속성으로 이동해서 모델링한다.

3.6 복합객체 재구성 연산: nest

이 절에서는 unnest 연산의 반대 개념인 nest 연산에 대해 설명한다. 이 연산은 루트 클래스 및 부분 클래스에 속하는 속성들을 조합해서 새로운 클래스로 따로 모델링함으로써 복합객체 계층구조의 재구성을 가능하게 한다.

nest *new_class_name*{attribute_list} **as** new_attribute_name **in class** class_name;

new_class_name은 nest 연산의 결과로 생성되는 가상 클래스(virtual class)의 이름을, new_attribute_name은 루트 클래스에서 이 클래스를 도메인으로 갖는 속성의 이름을 각각 나타낸다. 그리고, attribute_list는 가상 클래스에 포함될 부분 클래스의 속성들을 나타내는데, 다음과 같다.

{path_expression [as new_attribute_name] }*

nest 연산의 결과 새로 생성되는 클래스를 가상 클래스라고 부르는 이유는 다음과 같다. 기본 *RiBS* 모델에서 스키마 버전의 각 클래스 버전은 *RiBS* 계층에 자신의 기본 클래스를 갖는 반면, nest 연산을 통해 생성되는 클래스 버전의 경우 자신의 모든 속성들이 다른 클래스에 속하는 속성들의 조합으로 이루어질 수 있기 때문에 *RiBS* 계층에 자신의 기본 클래스를 갖지 않는다.

다음은 이 nest 연산을 이용해서 복합 객체 클래스 VEHICLE의 각 부품들의 모델번호를 새로운 가상 클래스 PARTMODEL로 정의하는 예를 보이며 그 결과는 그림 7에 나와 있다.

nest PARTMODEL {Body.Chassis.Model# **as** ChassisModel#,
DriveTrain.Transmission Model# **as** TransModel#,
DriveTrain.Engine.Model# **as** EngineModel# }
as PartModel **in class** *VEHICLE*;

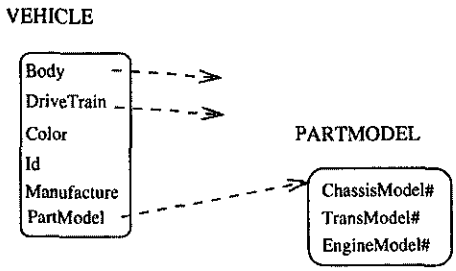

그림 7 nest 연산: VEHICLE 및 가상 클래스 PART-MODEL

위에서 언급한 바와 같이, nest 연산의 결과 스키마 버전내에 하나의 가상 클래스가 생성된다. 그러나, 이 가상 클래스도, 일반 클래스와 마찬가지로, 속성 추가 등의 스키마 진화 연산이 가능해야 한다. 기본 *RiBS* 모델의 경우, 각 클래스 버전마다 *RiBS* 계층에 자신에 대응하는 기본 클래스를 갖기 때문에, 추가되는 속성버전의 기본 속성을 해당 기본 클래스에 추가하면 된다. 그러나, nest 연산의 결과 생긴 가상 클래스의 경우 자신의 기본 클래스를 *RiBS*에 갖고 있지 않다. 따라서, 새로 추가되는 속성을 *RiBS* 계층의 어떤 클래스에 추가할 것인가의 문제가 생긴다. 본 논문에서는 이 문제를 다음과 같이 해결한다. 즉, 가상 클래스에 대응하는 기본 클래스를 *RiBS* 계층에 생성하고 추가되는 속성버전의 기본 속성을 생성된 기본 클래스에 추가한다. 따라서, 해당 가상 클래스는 더 이상 가상 클래스가 아니다.

3.7 복합객체 재구성 연산: move

사용자는 복합객체의 한 부분 클래스의 정보를 다른 부분 클래스의 정보로 모델링할 필요가 있다. 이를 위해 본 논문에서는 다음과 같은 속성의 이동연산을 지원한다.

move src_path_exp **to** dest_path_exp [**as** new_attribute_name] **in class** class_name;

경로식 src_path_exp에 해당하는 속성(단말 혹은 비단말)을 경로식 dest_path_exp가 가리키는 클래스로 이동시킨다. 이 때 dest_path_exp의 마지막 속성은 비단말 속성이어야 한다. 만일 속성의 이름을 바꾸고자 하는 경우에는 as 절을 이용해서 새 이름을 부여한다. 이동된 속성은 원래 소속 클래스에서 삭제된다. 다음은 이 연산 move를 이용해서 그림 1의 Chassis 정보를 클래스 DRIVETRAIN으로 이동시키는 연산인데, 이 연산의 결과는 그림 8에 나와 있다.

move Body.Chassis **to** DriveTrain **as** Body Chassis **in class** VEHICLE;

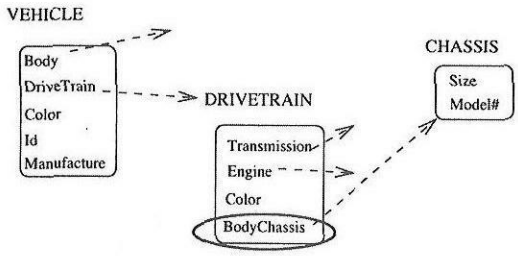

그림 8 move 연산의 예

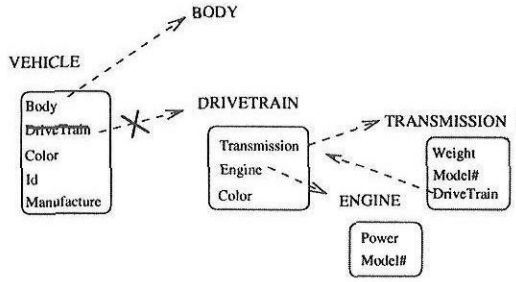

그림 9 잘못된 move 연산의 예

그런데, move 연산의 두 경로식 src_path_exp, dest_path_exp은 다음과 같은 제약점을 갖는다. 경로식 src_path_exp가 dest_path_exp의 복합객체 계층구조상의 부분이어서는 안된다는 점이다. 다음은 이 조건을 위배하는 move 연산의 예이다.

move DriveTrain **to** DriveTrain.Transmission **in class** VEHICLE;

이 연산은, 그림 9에서 보여지듯이, 두 가지 문제점을 야기한다; 즉, 1) 더 이상 DRIVETRAIN 클래스가 VEHICLE 클래스의 부분 클래스가 아니고, 2) DRIVETRAIN이 TRANSMISSION 클래스의 부분 클래스가 된다. 첫번째 문제는 DRIVETRAIN 클래스 이하의 정보를 루트 클래스에서 더이상 접근할 수 없게 만든다. 두번째 문제는, 관계 *IsPartOf(A,B)*에 의미적으로 배치되는 *IsPartof(B,A)*의 관계를 초래한다.

마지막으로, 이 장을 마치기 전에 여기서 제안한 재구성 연산들의 선정기준에 관한 설명을 하고자 한다. 우선 재구성 연산들의 선정 기준은 특정한 클래스 버전 C를 루트로 하는 복합객체 계층의 DAG(Direct Acyclic Graph) 구조를, 역시 C를 루트로 임의의 또다른 DAG 구조로 재구성하는 것이다. 단, 재구성 이전의 DAG 구조의 두 클래스 버전 A, B사이에 성립하는 *IsPartOf (A,B)* 관계에 의미적으로 배치되는 *IsPartOf (B,A)*가 재구성된 DAG 구조에는 포함되어서는 안된다. move 연산의 경우가 이런 현상을 초래할 수 있는데, 앞에서 기술한 바와 같이 이런 경우를 배제하기 위해 move 연산의 구문을 제한했다. 연산 pull과 unnest는 임의의 속성 버전(들)을 복합객체 계층구조 DAG의 윗방향으로 이동시키고, move 연산은 윗방향이나 다른 경로로 이동시키므로 이전 DAG의 *IsPartOf* 관계를 거스러는 경우는 발생하지 않는다. nest 연산의 경우도 새로운 가상 클래스 버전을 도입해서 다른 경로상의 속성 버전을 해당 가상 클래스 버전으로 이동하는 것으로 해석할 수 있으므로, 이전 DAG의 PartOf 관계를 역시 거스러지 않는다. 마지막으로, 부분 클래스의 추가와 삭제를 위한 두 연산의 경우 *IsPartOf(A,B)*에 배치되는 *IsPartOf* 관계를 야기하지는 않는다. 요약하자면, 확장 *RiBS* 모델에서 제안하는 연산들은 복합객체 계층구조상의 임의의 속성 버전을, 이전 DAG 구조의 *IsPartOf* 관계에 의미적으로 상충되는 DAG 구조를 배제하면서, 해당 복합객체 계층구조의 임의의 위치로 이동을 허용한다. 이런 의미에서, 본 장에서 제안한 6가지의 연산들은 복합객체 재구성을 위해 완전(complete)하다.

4. 재구성된 복합객체 계층구조의 질의 처리

2장에서 언급한 바와 같이 기본 *RiBS* 모델에서 특정 스키마 버전을 대상으로 작성된 질의나 응용 프로그램은 전처리 과정을 거쳐서 *RiBS* 계층의 스키마를 대상으로 한 질의나 응용 프로그램으로 변환되어 수행된다. 본 장에서는 질의 변환을 중심으로 재구성된 복합객체 계층구조의 전처리가 어떻게 이루어지는가를 살펴보자.

사용자가 그림 3의 스키마 버전의 VEHICLE 클래스 버전을 대상으로 다음의 좌측과 같은 질의를 작성한 경우, 실제로 전처리를 거쳐 *RiBS* 계층에 대한 우측의 질의로 변환되어 수행된다. 즉, pull 연산을 통해 루트클래스로 이동된 ChassisModel이라는 속성을 *RiBS* 계층에서 대응하는 경로식 Body.Chassis.Model#로 대체한다. 이와 같은 상대적으로 단순한 변환은 unnest 연산을 통해 재구성된 스키마에 대한 질의에도 마찬가지로 적용된다.

```
select Car.Color                    select Car.Color
from VEHICLE Car        ==>         from VEHICLE Car
where Car.ChassisModel = 'MD001';       where Car.Body.Chasssi.Model# = 'MD001';
```

move 연산은 좀 더 복잡한 형태의 질의 변환을 필요로 한다. 예를 들어, 그림 8의 스키마 버전에 대한 다음

질의를 보자. 하지만, 질의 그래프가 VEHICLE 클래스 버전을 포함하지 않는 다음의 예를 보자.

```
select Engine.Model#
from DRIVETRAIN Train
where Train.BodyChassis.Model#='MD001';
```

위의 예에서 조건절의 경로식은 *RiBS* 계층에서 DRIVETRAIN 클래스 버전의 기본 클래스에서 시작하는 경로식으로 표현할 수 없고, DRIVETRAIN 클래스 버전을 부분 클래스로 갖는 VEHICLE 클래스 버전의 기본 클래스를 통해서 표현해야 한다. 즉, 다음과 같은 *RiBS* 계층에 대한 질의로 변환되어야 한다.

```
select Car.DriveTrain.Engine.Model#
from VEHICLE Car
where Car.Body.Chassis.Model# = 'MD001';
```

nest의 연산의 결과로 생기는 가상클래스가 질의 그래프의 루트클래스로 사용되는 경우에도 이와 유사한 변화 과정을 필요로 한다.

5. 스키마 버전 통합

2장에서 기술한 바와 같이, *RiBS* 모델에서는 두 개 이상의 스키마 버전들에서 새로운 스키마 버전을 유도하는 연산을 지원하고 있다. 이 때 스키마 버전들의 스키마 정보를 일관성있게 결합하는 과정을 스키마 버전 통합(schema version merging)이라 한다. 본 장에서는 복합객체 계층구조에 대한 스키마 진화 연산들을 통해 재구성된 스키마 버전들을 통합하는 과정에서 발생하는 충돌(conflicts)의 유형을 살펴보고 이를 해결하는 스키마 버전 통합 알고리즘을 제시한다. 이 알고리즘은 충돌의 검사(detection)는 알고리즘에서 자동적으로 수행하지만, 충돌의 해결(resolution)은 사용자의 책임이라는 점에서 반자동적(semi-automatic)이다.

5.1 충돌 유형

기본 *RiBS* 모델에서 스키마 버전 통합 과정에서 발생하는 충돌유형으로 1) 동음이의어(homonyms) 문제, 2) 이음동의어(synonyms) 문제 그리고 3) 익스텐트 이동 충돌(extent migration conflicts)의 세 가지가 있다. 이 때 앞의 두 가지 문제는 이름 충돌(name conflicts)의 문제인데, 이들의 원인은 기본 *RiBS* 모델의 스키마 진화 연산 - 예를 들어, 클래스(속성) 버전 이름 변경과 새로운 클래스(속성) 버전의 생성 - 들을 각기 다른 스키마 버전에서 수행했기 때문이다. 한편, 익스텐트 이동 충돌의 문제는 구조적인 충돌(structural conflicts)로 객체지향 데이타모델의 다중상속 개념에 의해 발생한다. 이들에 대한 자세한 내용과 해결 알고리즘은 [16]를 참조하기 바란다.

3장에서 제안한 복합객체에 대한 스키마 재구성 연산들은, 위의 기본 *RiBS* 모델에서 발생하는 충돌 이외에, 다음의 세 가지의 새로운 충돌 문제를 야기한다. 여기서 동음이의어와 이음동의어 문제는 기본 *RiBS* 모델에서도 존재하지만, 새로 도입된 연산에 의해 생기는 다른 형태의 문제이다.

1. 동음이의어(homonyms) 문제: 각기 다른 스키마(클래스) 버전에 속하는 두 개 이상의 클래스(속성) 버전들이 이름은 같으나 서로 다른 기본 클래스(속성)를 가질 때, 이들을 동음이의어 클래스(속성)라 한다.
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 때, 이들은 이음동의어 클래스(속성)라 한다.
3. 클래스-속성 충돌: 같은 개념이 서로 다른 객체지향 데이타모델의 구성요소, 즉, 클래스와 속성으로 표현되는 구조적인 충돌이다. 즉, 한 스키마 버전에서는 특정 정보를 독립적인 클래스 버전으로 모델링하고, 다른 스키마 버전에서는 속성들로 표현하는 경우에, 두 스키마 버전이 해당 정보에 대해 클래스-속성 충돌을 갖는다.

그림 10의 두 스키마 버전 SV_i와 SV_j의 통합은 위의 세가지 충돌 유형을 모두 포함하고 있는데, 이들 스키마 버전에 대해 다음과 같은 가정을 한다; 각 스키마 버전

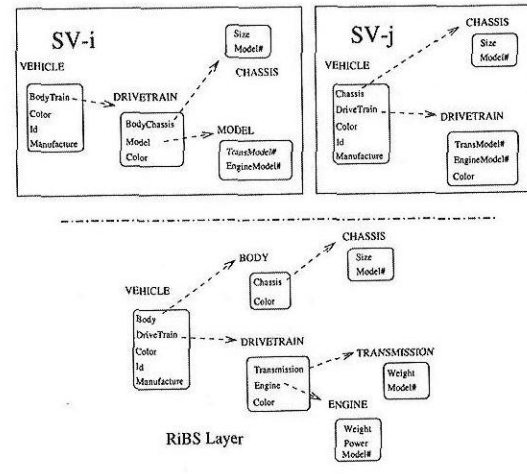

그림 10 스키마 버전 통합 예

은, *RiBS* 계층의 기본 스키마와 같은 스키마 정보를 갖는 스키마 버전에서 유도되어, 각각 다음과 같은 스키마 진화 과정을 거쳤다. 다음에서 연산 drop과 rename은 기본 *RiBS* 모델의 스키마 진화 연산으로 각각 속성 버전의 삭제와 속성 버전의 이름 변경을 나타낸다.

* SV_i의 진화과정
  1. drop Color from VEHICLE;
  2. pull Body.Chassis.Color in class VEHICLE;
  3. move Body.Chassis to DriveTrain as Body-Chassis in class VEHICLE;
  4. rename DriveTrain as BodyTrain in class VEHICLE;
  5 nest MODEL{ Transmission.Model# as Trans-Model#, Engine.Model# as EngineModel# } as Model in class DRIVETRAIN;

* SV_j의 진화과정
  1. pull Body.Chassis in class VEHICLE;
  2. pull Transmission.Model# as TransModel# in class DRIVETRAIN;
  3. pull Engine.Model# as EngineModel# in class DRIVETRAIN;
  4. drop Transmission from DRIVETRAIN;
  5. drop Engine from DRIVETRAIN;

그림 10에서 동음이의어의 예로는 SV_i의 클래스 버전 VEHICLE의 속성 버전 Color와 SV_j의 클래스 버전 VEHICLE의 속성 버전 Color를 들수 있다. 전자는 *RiBS* 계층의 VEHICLE 클래스의 Color를, 후자는 BODY 클래스의 Color를 각각 기본 속성으로 갖는다. 다음으로 그림 10에는 두가지 이음동의어의 예를 포함하고 있다. 첫째, SV_i의 클래스 버전 VEHICLE의 속성 버전 BodyTrain과 SV_j의 클래스 버전 VEHICLE의 DriveTrain 속성버전이다. 이들은 이름은 다르지만, 둘다 기본 스키마 *RiBS*의 클래스 VEHICLE에서 DriveTrain 속성을 자신들의 기본 속성으로 갖는다. 두번째 이음동의어의 예는 SV_i의 클래스 버전 DRIVETRAIN의 속성 버전 BodyChassis와 SV_j의 클래스 버전 VEHICLE의 속성 버전 Chassis이다. 이들은 둘다 기본 클래스 BODY 클래스의 Chassis를 자신들의 기본 속성으로 갖는다. 여기서 주목할 점은 이들 두 이음동의어 속성버전이, 다른 기본 클래스를 갖는 클래스 버전에 속한다는 점이다. 기본 *RiBS* 모델의 스키마 버전 통합과정에서는 속성 버전의 이음동의어 문제는 같은 기본 클래스를 갖는 두 클래스 버전에 속하는 속성 버전들 사이에서만 발생한다. 마지막으로, 클래스-속성 충돌의 예는 SV_i에서 클래스 버전 MODEL로 표현된 정보와 SV_j에서 클래스 버전 DRIVETRAIN의 두 속성 Trans-Model#와 EngineModel#로 표현된 정보를 들 수 있다.

5.2 스키마 버전 통합 과정

기본 *RiBS* 모델에서의 스키마 버전 통합 과정은 대략 다음의 4 단계로 이루어지는데, 자세한 알고리즘은 [19]을 참조하기 바란다.

1. 새 스키마 버전에 포함될 필요한 기본 클래스들 계산
2. 각각의 기본 클래스에 해당하는 새로운 클래스 버전들 생성
3. 새 클래스 버전들에 필요한 지역 속성들 계산
4. 새 스키마 버전의 클래스 계층구조 생성

단계 2의 과정은 클래스 버전의 이음동의어, 동음이의어 및 익스텐트 이동 충돌에 대한 해결을, 단계 3에서는 속성 버전에 대한 이음동의어및 동음이의어 해결을 하게 된다. 복합객체 스키마 진화 연산으로 인하여, 스키마 버전 통합의 과정은 크게 다음 세 가지 확장을 필요로 한다.

1. 가상 클래스 버전: nest 연산에 의해 생성된 가상 클래스 버전을 고려해야 한다.
2. 클래스-속성 충돌: 사용자가 각기 다른 스키마 버전에서 클래스 버전과 속성 버전들로 표현된 같은 정보에 대하여 충돌을 해결해야 한다.
3. 이음동의어 문제: 앞에서 지적한 바와 같이, 서로 다른 기본 클래스를 갖는 클래스 버전들사이에 발생하는 속성 버전의 이음동의어 문제에 대한 고려가 필요하다.

다음은 이들을 고려해서 확장된 스키마 버전 통합 과정인데, * 표시가 된 부분은 위의 확장과 관련한 단계이다.

1. 새 스키마 버전에 포함될 필요한 기본 클래스들 계산
2. 새 클래스 버전들의 생성
   (1) 각 기본 클래스에 해당하는 새로운 클래스 버전 생성
   (2) 클래스-속성 충돌 검사및 해결(*)
   (3) 가상클래스 버전 생성(*)
3. 새 클래스 버전들의 지역 속성버전들 계산
   (1) 각기 다른 클래스 버전들에 속하는 이음동의어 속성들의 충돌해결(*)
   (2) 각 클래스 버전의 지역 속성버전들 계산
4. 새 스키마 버전의 클래스 계층구조 생성

본 논문에서는 이 과정을 예를 통해 설명하겠다. 그리고, 단계 4는 복합객체 계층구조와는 직접적인 관련성이 없는 관계로 언급하지 않겠다. 그림 11은 그림 10의 두 스키마 버전을 통합하는 과정을 단계별로 보여주고 있

다. 통합의 결과 생기는 새 스키마 버전을 SV_{new}로 표기하겠다. 그림 11.(a)는 위 단계 2.(1)을 거친 상태를 보여주고 있다. 그림 10의 두 스키마 버전의 클래스 버전들이 갖는 기본 클래스로는 *RiBS*의 VEHICLE, DRIVETRAIN, 그리고 CHASSIS이다. 단계 1.에서 이 기본 클래스들을 구하고서 이들에 대응하는 클래스 버전을 단계 2.(1)에서 SV_{new}에 생성하는 것이다. 그림 11.(b)는 단계 2.(2)와 2.(3)에서 클래스-속성 충돌의 해결 후 생성되는 클래스 버전 MODEL이 새 스키마 버전에 추가적으로 생성된 것을 나타낸다. 앞에서 기술한 바와 같이, 그림 10에서 스키마 버전 SV_i의 클래스 버전 MODEL과 SV_j의 클래스 버전 DRIVETRAIN의 두 속성 TransModel#와 EngineModel#사이에는 클래스-속성 충돌이 있다. 이 충돌에 대해 사용자가 클래스로 모델링된 정보를 선택했다고 가정한다. 그런데, MODEL이 스키마 버전 SV_i에서 nest 연산에 의해 생성된 가상 클래스 버전이기 때문에 이에 해당하는 가상 클래스 버전을 SV_{new}에도 생성해야 한다. 그림 11.(c)는, 그림 10의 예에서 이음동의어 충돌을 갖는 스키마 버전 SV_i의 속성 버전 BodyChassis와 SV_j의 속성 버전 Chassis에 대해 사용자가 후자를 선택한 결과를 보여준다. 마지막으로, 그림 11.(d)는 SVnew의 각 클래스 버전의 속성 버전을 자신에 대응하는 SV_i와 SV_j의 클래스 버전에서 계산한 결과를 보이고 있다. 여기서 클래스 버전 VEHICLE의 속성 Color와 BodyColor는 동음이의어 충돌을 갖는 SV_i와 SV_j의 클래스 버전 VEHICLE의 속성 버전 Color에 대해 사용자가 SV_i의 Color를 BodyColor로 이름을 바꾸어서 충돌을 해결한 결과이다.

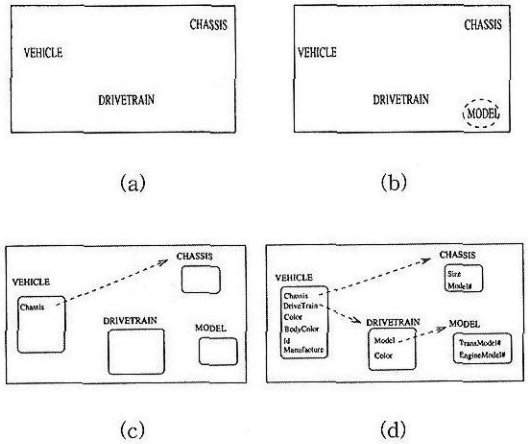

그림 11 스키마 버전 통합 과정

여기서 주목할 점은, 스키마 버전 통합의 결과로 새로운 스키마 버전 SV_{new}에는 같은 기본 속성을 갖는 두 개 이상의 속성 버전이 존재하지 않는다는 것이다. 이는, 3장에서 언급한 *RiBS* 모델의 원칙상, *RiBS* 계층의 특정 정보가 스키마 버전내에서 서로 다른 속성으로 중복적으로 모델링되는 것을 방지하기 위한 것이다. 스키마 버전 통합과정에서 이음동의어와 클래스-속성 충돌의 해결이 이를 보장해준다.

6. 관련 연구

본 저자들이 아는 범위에서, 복합객체 계층구조에 대한 스키마 진화 연산이나 이를 고려한 스키마 버전 모델에 관한 연구는 없다. 따라서, 이 장에서는 논문 내용과 부분적으로 관련있는 연구들, 1) 객체지향 데이타베이스 뷰 [20, 18, 21], 2) 스키마 진화/버전 [1, 7, 22, 14, 23, 2], 3) 데이타베이스 스키마 통합 [24, 21], 그리고 4) 웹 사이트 관리 시스템 [25, 26, 27, 28]을 들 수 있다.

6.1 객체지향 데이타베이스 뷰

관계형 데이타베이스에서 뷰가 제공하는 장점들 - 예를 들어, 논리적 데이타 독립성 제공, 내용기반 보안기능 - 을 객체지향 데이타베이스에서도 제공하려는 노력들이 있었다. [20]은 O2 시스템의 뷰 메카니즘을 다루고 있는데, 클래스 계층구조의 재구성과 가상 클래스(virtual classes)의 정의를 지원한다. 객체/관계형 데이타베이스 시스템인 UniSQL은 관계형 뷰 기능에 객체지향 개념들 - 상속, 메소드, 객체식별자 - 을 추가해서 객체지향 뷰를 지원한다 [21]. [29]에서는 전역(global) 스키마에서 사용자의 필요에 맞게 뷰 스키마를 정의할 수 있게 해주는 MultiView 방법론을 다루고 있다. [18]에서는 객체지향 뷰를 이용해서 스키마 진화연산을 시뮬레이션하는 방안을 제안했다.

RiBS 모델의 스키마 버전이 기본스키마 *RiBS*에 대한 논리적 뷰라는 점에서 이들 연구와 맥락을 같이한다. 그러나, 이들 뷰 메카니즘들은 복합객체의 재구성은 전혀 고려하지 않은 반면, 확장 *RiBS* 모델은 사용자로 하여금 복합객체 계층구조의 재구성을 가능하게 해준다.

6.2 스키마 진화/버전 연구

[7]은 객체지향 데이타베이스를 위한 스키마 버전 모델을 처음으로 제안했다. 이 연구는 기존 ORION 데이타모델의 객체 버전 모델을 확장한 것인데, 모델의 구성요소는 스키마 버전 관리 및 접근 범위에 관한 규칙들로 이루어져 있다. [14]는 클래스 계층구조 전체를 버전의 단위로 하는 스키마 버전 방법과 달리, 클래스를 버

전닝의 단위로 하는 클래스 버전닝(class versioning) 시스템 CLOSQL을 제안했다.

서론에서 기술한 바와 같이, 객체지향 데이타베이스 시스템들의 등장 초기에서부터, 스키마 진화 기능을 제공해왔다 [1, 30, 2] 이들은 클래스 계층구조에 대한 스키마 진화 연산들을 온라인으로 지원하며 스키마 갱신후 영향을 받는 객체의 포맷은 새로운 스키마에 맞게 자동적으로 변경된다.

그런데, 이들 작업들은 모두 상속 위주의 클래스 계층구조 변경만을 고려하며, 복합 객체의 구조 변경에 관한 연구는 없다. 다만, 데이타베이스 재구성과 관련하여 [22]의 **moving information** 연산과 [23]의 **capacity reducing transformation** 연산은 각각 본 논문에서 제안한 pull, unnest 연산과 비슷한 의미를 갖는다.

6.3 데이타베이스 스키마 통합 연구

데이타베이스 분야에서 여러 데이타베이스들의 스키마 통합에 관한 많은 방법론들이 뷰 통합(view integration), 스키마 통합(schema integration), 혹은 다중 데이타베이스(multi-database)의 이름으로 제안되었다. 이들 방법론들의 핵심은 스키마 통합 과정에서 충돌(conflicts)을 검사(detection)하고 충돌을 해결(resolution)하는 것이다

[24]에서는 이전의 뷰 및 데이타베이스 통합 방법론들을 비교/분석하고 있는데, 특히 충돌과 관련하여 이름충돌과 구조적인 충돌로 나누었다. [21]에서는 객체지향 데이타모델을 다중 데이터베이스 통합 모델로 제시하면서 이때 생겨나는 충돌현상의 종류들을 나열하고 이들의 해결책을 다루고있다. 이들 두 연구에서 모두 본 논문에서 언급한 동음이의어, 이음동의어, 그리고 클래스-속성 충돌과 유사한 문제를 언급하고 있다.

그런데, 본 논문의 스키마 버전 통합과 데이타베이스 스키마 통합은 충돌의 해결방안에 있어서 큰 차이가 있다. 즉, *RiBS* 모델에서는 통합될 스키마 버전들의 각 스키마 구성요소들은 기본 스키마 *RiBS*의 스키마 구성요소를 매개로 의미적 정보를 공유하고 있다. 이는 *RiBS* 모델에서의 스키마 버전 통합의 과정에서 자동적인 충돌 검사를 가능하게 해준다. 반면, 기존의 데이타베이스 스키마 통합 연구에서는 충돌의 검사 과정에 사용자의 부담이 훨씬 크다.

6.4 웹사이트 관리 시스템

인터넷의 급속한 확산과 더불어, 많은 정보들이 웹(Web) 페이지 형태로 구축되고 있다. 대규모 웹 사이트 구축과정은, 데이타베이스의 설계와 유사하게, 대략 (1) 웹사이트에서 제공할 정보의 내용 선택, (2) 각각의 페이지에 포함될 정보 내용 구성, (3) HTML 등을 이용해서 내용의 시각적인 표현의 세 단계로 거친다. 그런데, 현재의 웹사이트 관리의 문제점은 웹사이트에 구축된 정보의 물리적 구조(physical organization)와 사용자에게 보여지는 논리적인 구조(logical view)가 분리가 안되어 있어 웹사이트의 재구성 작업이 굉장히 어렵다 [28]. 이는 데이타베이스 용어를 빌리자면, 데이타 독립성(data independence)이 부족한 것이다.

이와 같은 문제를 해결하기 위해 웹사이트에 관리되는 정보의 물리적인 구조에 독립적으로 사용자에게 보여지는 논리적인 구조의 손쉬운 재구성을 지원하는 웹사이트 관리시스템(web-site management system)에 관한 연구가 활발히 진행중이다 [25, 26, 28]. WebOQL [25]과 STRUDEL [28]의 경우, 특정한 웹사이트에 물리적으로 구축되어 있는 내용을 사용자가 자신이 보고 싶은 논리적인 형태로 재구성할 수 있게 해준다. 이를 위해 이 두 시스템은 각각 WebOQL과 StruQL이라 불리는 특수한 형태의 질의어를 제공하고 있다. ARANEUS 시스템의 경우 [26], 웹 데이타에 대한 관계형 뷰 정의를 위한 Ulixes와 관계형 뷰에서 재구성된 웹 내용의 유도를 위한 Penelope라는 언어를 제공해서 웹 데이타의 재구성을 지원한다. 마지막으로, [27]에서는 객체지향 데이타베이스위에 웹 뷰 정의를 지원하는 언어 - 즉, 데이타베이스에 저장된 정보를 여러 다양한 형태로 재구성해서 웹 페이지로 사용자에게 제공하는 기능 - 를 제안했다. 만일 객체지향 데이타베이스 시스템에서 본 논문에서 제안한 복합객체 계층구조 재구성 연산들을 지원한다면, 현재 이들 웹 사이트 관리시스템에서 제공하는 많은 기능을 대체할 수 있고, 또한 객체지향 데이타베이스가 웹 데이타의 저장소로서 지금보다 나은 대안이 될 것이다.

이들 시스템과 본 논문의 스키마 버전 모델의 공통점은 모두 복잡한 그래프 구조의 데이타의 재구성을 통해서 사용자자 원하는 형태로 데이타를 볼 수 있게 해준다는 것이다. 그러나, 다음의 몇 가지 점에서 차이가 있다. 첫째, 복합객체 뷰는 구조적인(structured) 객체지향 데이타베이스의 스키마의 재구성을 지원하는 반면, 위의 웹사이트 관리 시스템들은 반-구조적인(semi-structured) 웹 데이터를 다룬다는 점이다. 둘째, WebOQL이나 StruQL 언어는 재구성과정에서 웹 데이타의 값에 의한 선택적인 재구성을 지원하는 반면, 확장 *RiBS* 모델은 단지 구조적인 측면에서 복합객체 계층구조의 재구성을 지원한다는 점이다.

7. 결 론

객체지향 데이타베이스 시스템이 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, 클래스 계층 구조에 대한 스키마 버전 모델인 기본 *RiBS* 모델을 확장해서, 복합객체의 재구성을 지원하는 방안을 제시했다. 우선, 복합객체 재구성 연산들을 최초로 제안하고, 이 연산들을 통해 재구성된 복합객체 계층 구조에 대한 질의 처리 방법도 언급했다. 그리고, 이 복합객체 재구성 연산을 고려한 스키마 버전 통합의 알고리즘을 고안했다.

본 논문의 기여점은 크게 1) 복합객체 계층구조의 재구성을 위한 연산들을 최초로 도입한 점과 2) 이 연산들이, 갱신가능한 클래스 계층구조 뷰 형태의 스키마 버전 모델 *RiBS*와 결합해서, 객체지향 데이타베이스의 데이타 독립성(data independece)을 제공한다는 점이다.

우리는 향후 다음 두가지 연구를 할 것이다. 첫째, *RiBS* 모델을 SOP 객체지향 데이타베이스 시스템 [31] 상에 구현한다. 현재 SOP 시스템은 클래스 계층구조에 관한 스키마 진화 기능을 제공하는데, 이를 기반으로 스키마 버전 계층을 구현한다. 둘째, 이렇게 구현된 스키마 버전 기능을 웹사이트 관리 시스템의 구현에 적용함으로써 강력한 스키마 버전 기능을 제공하는 객체지향 데이타베이스 시스템이 웹사이트 관리와 같은 새로운 응용분야에 얼마나 효율적으로 사용될 수 있는지를 검증하고자 한다.

참 고 문 헌

[1] J. Banerjee and Won Kim and H.-J. Kim and H. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," Proceedings of SIGMOD, pp. 311-322, 1987.

[2] Roberto Zicari and Fabrizio Ferrandina, "Schema and Database Evolution in Object Database Systems," In Part6, Advanced Database Systems, Morgan Kaufmann, 1997.

[3] Object Design, Inc, "ObjectStore Technical Overview, Release 3.0," 1994.

[4] Objectivity, Inc., "Schema Evolution in Objectivity/DB," White paper available from "http://www.objy.com/ObjectDatabase/WP/Schema/schema.htm"]

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

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

[7] Won Kim and H.T. Chou, "Versions of Schema for Object-Oriented Databases," Proceedings of VLDB, pp. 148-159, 1988.

[8] S.-E.c Lautemann, "A Propagation Mechanism for Populated Schema Versions," Proceedings of ICDE, pp. 67-78, 1997

[9] T. Atwood, "Object Databases Come of Age," Object Magazine, July, 1996.

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

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

[12] M. E. Loomis, "Object Database - Integrator for PCTE," Journal of Object Oriented Programming, May, 1992.

[13] POET Software, "POET Technical Overview, Release 3.0", White paper available from "http://www.poet.com/techover/"

[14] S. Monk and I. Sommerville, "Schema Evolution in OODB Using Class Versioning," SIGMOD Records, 22(3), 1993.

[15] Y.G. Ra and E. A. Rundensteiner, "A Transparent Object-Oriented Schema Change Approach Using View Evolution," Proceedings of ICDE, pp. 165-172, 1995.

[16] 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.

[17] C. Zaniolo, "The Database Language GEM", Proceesings of SIGMOD, pp. 207-218, 1983

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

[19] A. Silberschartz and H. Korth and S. Sudarshan, "Database System Concepts(3rd Edition)", McGraw Hill, 1997.

[20] Serge Abiteboul and Anthony Bonner, "Objects and Views," Proceedings of SIGMOD, pp 238-247, 1991

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

[22] B Lerner and A. Nico Habermann, "Beyond Schema Evolution to Database Reorganization", Proceedings of OOPSLA, 1990.

[23] M. Tresch and M. Scholl, "Schema Transformation without Database Reorganization", SIGMOD Record, 22(1), 1993.

[24] C. Batini and M. Lenzerini and S. B. Navathe, "A

Comparative Analysis of Methodologies for Database Schema Integration," ACM Computing Survey, 18(4), pp. 323-364, 1986.
[25] G. Arocena and A. Mendelzon, "WebOQL: Restructuring Documents, Databases and Webs". Proceedings of ICDE, 1998.
[26] P. Atzeni and G. Mecca and P. Merialdo, "To Weave the Web", Proceesing of VLDB, 1997.
[27] G. Falquet and J. Guyot and L. Nerima, "Languages and Tools to Specify Hypertext Views on Databases". International Workshop on the Web and Databases, 1998.
[28] M. Fernandez and D. Florescu and Jaewoo Kang and A. Levy, "Catching the Boat with Strude: Experiences with a Web-Site Management System", Proceedings of SIGMOD, 1998.
[29] E. A. Rundensteiner, "MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases," Proceedings of VLDB, pp. 187-198, 1992
[30] D. Penney and J. Stein, "Class Modifications in the GemStone Object-Oriented DBMS," Proceedings of OOPSLA, pp. 111-117, 1987.
[31] 안정호, 이강우, 송하주, 김형주, "Soprano: 객체저장 시스템의 설계 및 구현". 정보과학회 논문지(C), 1996.

이 상 원
1991년 서울대 컴퓨터공학과 학사. 1994년 서울대 컴퓨터공학과 석사. 1999년 서울대 컴퓨터공학과 박사. 1999년 1월 ~ 현재 한국 오라클 근무. 관심분야는 데이타베이스, 데이타웨어하우스, ERP, EC.

김 형 주
제 26 권 제 1 호(B) 참조