객체지향 데이타베이스 시스템에서 확장된 관계성의 설계와 구현

(Design and Implementation of Extended Relationship Semantics in an ODMG-Compliant OODBMS)

이 현 주 † 송 하 주 †† 이 상 원 ††† 김 형 주 ††††

(Hyun-Ju Lee) (Ha-Joo Song) (Sang-Won Lee) (Hypung-Joo Kim)

요 약 관계성(relationship)은 개체(entity)와 더불어 데이타베이스에서 모델링하는 중요한 실세계 개념이다. 특히, CAD/CAM, CASE, 멀티 미디어 등과 같은 객체지향 데이타베이스 응용분야들은 복잡한 관계성을 필요로 한다. 데이타베이스 시스템에서 다양한 관계성을 지원하지 않을 경우, 관계성의 표현을 각 응용프로그램에서 직접 프로그래밍을 해야하기 때문에 프로그램의 개발 및 유지, 보수 단계에서 관계성의 관리와 관련한 많은 오버헤드가 발생한다.

본 논문에서는 객체지향 데이타베이스의 표준인 ODMG-2.0을 확장한 관계성 모델을 제안한다. 이 관계성 모델은 관계형 데이타베이스, 객체지향 데이타모델 등의 영역에서 필요로 하는 관계성을 모두 지원하며, ODMG-2.0 표준의 초보적인 관계성 기능을 객체지향 상속 개념을 이용해서 자연스럽게 확장한 것이다. 그리고 응용프로그램의 수행시 관계성과 관련해서 발생할 수 있는 이상현상에 대한 해결책을 제시한다.

Abstract Relationships, in addition to entities, are important in real-world database modeling. In particular, many object oriented database applications including CAD/CAM, CASE and multi-media necessitate the modelling of various and complex

relationships. Without support from database systems for the relationship management, there is a huge overhead related to the management of relationships during both application development and maintenance, since the relationships need to be hard-coded directly into the application program itself.

In this paper, we propose a powerful relationship model which extends the ODMG-2.0 object database standard. The proposed relationship model can support all of the relationship functionalities found in the relational database model, and the object oriented data model. In order to design and implement this relationship model, we seamlessly extend the ODMG-2.0 relationship facility using the inheritance concept. We also identify several run-time anomalies in objects with the relationship and provide solutions for their problems.

1. 서 론

기존의 관계형 모델[5]은 단순한 구조를 가지는 많은 양의 데이타를 모델링하기에 적합하다. 기본적인 데이타 아이템들은 약 몇 백 바이트 정도의 짧은 고정 길이의 레코드로 구성되어 있으며, 이러한 데이타들은 테이블로 나타내어진다. 그리고, 데이타들 사이의 관계는 외래키(foreign key)가 다른 테이블의 기본 키(primary key)의 값을 가짐으로써 나타낸다. 관계형 모델의 이러한 특징은 데이타들 간의 관계가 다양하고 복잡하게 나타나는 응용 분야를 모델링하기에는 적합하지 않다.

반면 객체지향 모델은 순환적으로 중첩하여 복잡한 객체(complex object)를 나타낼 수 있을 정도로 강력하다.[10] 객체지향 모델은 모델링하고자 하는 데이타들의

· 본 연구는 1999년도 BK21 정보기술분야 사업에서 지원을 받았음

† 비 회 원 : (주)자이닉스 부설연구소 연구원
hjlee@papaya.snu.ac.kr
†† 비 회 원 : 서울대학교 컴퓨터공학부
hjsong@papaya.snu.ac.kr
††† 비 회 원 : 한국오라클 연구원
swlee@oopsla.snu.ac.kr
†††† 종신회원 : 서울대학교 컴퓨터공학부 교수
hjk@oopsla.snu.ac.kr

논문접수 : 1998년 11월 3일
심사완료 : 2000년 5월 4일

행동적인 측면과 데이타들의 구조적인 관계를 추상화(abstraction)하여 하나의 단위로 나타낸다. 객체(object)가 실제 세계의 물체(entity)를 모델링하고, 객체의 상태(state)는 객체 자신의 메소드(method)를 통해서만 변경될 수 있다. 클래스(class)는 동일한 물체를 모델링한 객체들의 집합이며, 클래스간에는 계승 관계가 존재한다. 그리고, 클래스간의 구조적 관계는 클래스의 attribute로 표현하여 복잡한 객체(complex object)를 순환적으로 나타낼 수 있다.

이처럼 객체지향 모델의 모델링 파워가 강력하긴 하지만, 객체들의 집합을 하나의 논리적 단위로 나타내지는 못한다. CAD(Computer-Aided Design), CASE(Compute-rAided System Engineering), 정보저장소 등의 객체지향 데이타베이스의 많은 응용분야에서는 관련이 있는 객체들의 집합을 하나의 단위로 정의하고 연산하는 것을 요구한다[6, 10]. 특히 Bernstein은 [3]에서 정보저장소에서 필요로 하는 관계성 관리 기능을 기술하고, 현재의 객체지향 데이타베이스에서는 이와 같은 관계성을 제대로 지원하지 못하고 있음을 지적하고 있다.

관계성이란 두 개 이상의 객체가 논리적으로 하나의 단위로 취급될 수 있어서, 한 객체에 대한 연산이 다른 객체들에게도 영향을 미칠 수 있는 상호의존성(inter-dependency)을 뜻한다. 이미 여러 논문[2,9,10]에서 데이타베이스에서의 관계성 지원의 필요성을 강조하였으며, 이에 관한 연구도 이루어졌다. 이 중 대표적인 것이 ORION[10]의 복합 객체(composite object)이다. 이러한 논문들에서 언급한 관계성의 필요성을 정리하면 다음과 같다.

첫째, 명시적으로 스키마에 관계성이 표현된다. 따라서, 스키마에 나타난 관계성과 객체의 논리적 관계에 따른 연산의 전파를 통해 객체간의 의미적 구조가 명백해지며, 코드의 크기가 줄어든다.

둘째, 데이타베이스 어플리케이션 작성자가 잘못된 코드를 작성할 가능성을 배제한다. 복잡한 객체간의 연산을 스키마를 통해 간단히 표현할 수 있기 때문이다.

셋째, 어플리케이션 작성 후, 유지보수가 간편해진다. 관계성이 지원되지 않는다면 객체간의 시맨틱을 바꿀 일이 있을 때 이와 관련된 모든 메소드를 수정하여야 하지만, 관계성이 지원될 경우에는 스키마의 관계성만을 수정하면 된다.

객체지향 데이타베이스의 표준인 ODMG-2.0[4]의 객체 모델을 살펴보면 상태(state)를 모델링하기 위한 속성을 두 가지로 분류하여 제공하는데 이 중 한 타입의 상태에 대해 정의한 것을 **attribute**라고 하고, 두 타입 사이의 상호 참조를 정의한 것을 **relationship**이라고 한다.[1] 두 객체 사이의 논리적인 관계성을 나타내는 **relationship**은 관련된 두 타입에 양방향 패스(traversal path)를 선언하여 나타낼 수 있도록 하였다. ODMG-2.0의 C++ 바인딩에서는 이 패스를 템플릿 클래스 **d_Rel_Ref<class T, const char* M>**로 나타낸다. 이 템플릿 클래스를 변수로 가지고 있는 클래스는 클래스 **T**를 참조하고 있으며, 클래스 **T**에서도 자신을 참조하고 있는데, 클래스 **T**에서 자신을 참조하는 변수가 **M**이라는 것을 의미한다. 이 클래스는 관련이 있는 객체를 접근할 수 있도록 하고, 이미 삭제된 객체는 참조하지 않도록 해준다. 즉, 참조 무결성(referential integrity)을 보장하는 포인터를 제공하는 것이다. 이 ODMG-2.0에서의 관계성의 문제점은 관계성을 각 객체의 정적인 상태의 일부로만 인식한다는 것이다. 객체 사이의 참조 무결성을 보장하는 역할만을 할 뿐 관계성을 가짐으로써 발생하는 동적인 시맨틱을 표현하지는 못한다.

본 논문에서는 ODMG-2.0에서의 상태를 모델링하는 관계성의 정적인 역할을 확장시켜서 관계성에 체계적인 시맨틱을 부여하고, 이 시맨틱을 나타낼 수 있는 메소드를 추가하였다. 우선 관계성이 가질 수 있는 시맨틱의 모델을 **전체-부분 관계**(Part-Whole)에 바탕을 두어 제시하였고, 이것을 ODMG-2.0의 C++ 바인딩을 확장한 타입으로 나타내었다. 그리고, ODMG-2.0을 따르는 ODBMS인 **SOP(*S*NU *O*ODBMS *P*latform)**의 저장 시스템 **Soprano**[1]에 구현하였다.

본 논문의 구성은 다음과 같다. 우선 2절에서 객체지향 모델링 분야나 ODBMS 분야에서 지금까지 연구된 관계성의 여러 가지 시맨틱에 관해서 설명한다. 3절에서는 본 논문에서 제시하는 확장된 관계성의 모델을 소개하고, 이를 ODMG-2.0의 C++ 바인딩에 따르는 클래스로 구현한 것을 제시하고 그 사용을 예제를 통해 설명한다. 4절에서는 구현에 관련된 몇 가지 이슈에 대해서 설명하고, 5절에서 결론을 맺는다.

2. 관계성에 관련된 연구들

관계성에 관한 지금까지의 연구는 크게 두 가지 측면에서 살펴 볼 수 있다. 첫째는 객체들 사이의 관계에 자식-부모 관계나 콜렉션 등의 특별한 의미를 부여하여

1) ODMG-2.0 객체 모델에서 상태를 나타낼 때 키워드로써 attribute와 relationship을 사용한다.

객체들 사이의 관계를 체계적으로 정의하는 것이고, 둘째는 두 객체가 서로 관련이 있을 때 객체 사이의 무결성 제약을 보장하는 것이다.

첫 번째 측면으로 볼 수 있는 것에는 UML(United Modelling Laguage)에서의 관계성과 SORAC이라는 데이타 모델링 시스템에서 관계성을 폭넓게 정의한[12] 등이 있다.

이 중 UML은 객체들 사이의 관계성을 일반화(generaliziation), 집합(aggregation), 관련(association)의 세 가지로 분류하였다. 일반화는 **IS-A** 관계를 나타낸다. 만약 객체 **O1**이 객체 **O2**와 **IS-A** 관계이면 **O2**의 모든 멤버는 **O1**의 멤버가 된다. 집합은 한 객체가 여러 부분 객체들로 구성되어 있다는 것을 나타낸다. 즉, 부분-전체 관계에 해당한다. 대표적인 예로 차는 바퀴, 엔진, 문 등으로 이루어져 있다는 것이다. 이 집합은 많은 객체를 하나로 다룸으로써 복잡도를 줄일 수 있는데, 집합은 명확히 정의되지 않은 일종의 상위 개념이며, 이를 구체적으로 표현하기 위해서는 좀 더 세분화된 하위 개념이 필요하다. 관련은 한 객체가 다른 객체가 가지고 있는 서비스를 요청할 수 있는 것을 나타낸다. 예를 들어서 배우자 사이의 관계를 나타낼 수 있다. 이 UML은 객체들 사이의 관계에 의미를 부여하는 것에 중점을 두었고, 관련이 있는 객체들 사이에서 서로의 행동이 어떠한 영향을 미칠 수 있는가에 대해서는 설명하지 않았다.

무결성 유지의 측면에서 관계성을 다룬 것 중에는[9]가 있는데, 이 논문에서는 ODBMS에서 객체들 사이의 무결성을 유지하기 위한 방안을 제시하였다. 객체들 사이의 무결성 중에서 특별한 경우로써 관계 무결성(relational integrity)과 참조 무결성(referential integrity), 유일성 (uniqueness)을 제안했는데, 이들은 관련이 있는 객체들 사이에서 한 객체의 행동(behavior)에 따른 다른 객체의 행동을 클래스 선언에서 나타낼 수 있도록 하였다. 이러한 객체의 행동은 RDBMS의 무결성 제약을 ODBMS에 맞게 적용함으로써 나타난 것이다. RDBMS에서의 외래키에 대한 참조 무결성이 이 논문에서의 관계 무결성과 참조 무결성에 해당한다고 볼 수 있다. 이 논문에서는 이러한 객체들의 행동에 특정한 의미를 부여하지 않고 단지 무결성 유지의 측면에서만 설명하였다.

서로 관련이 있는 객체들 사이의 관계성에 체계적인 의미를 부여하고, 이 의미에 따른 객체들의 행동을 규정한 연구들 중에서 가장 널리 알려진 것이 ORION[10]의 복합 객체(complex object)이다. 이 복합 객체는 부분-전체 관계를 나타내는데, 전체 객체와 부분 객체들 사이의 관계를 **exclusiveness**와 **dependency**의 두 가지로 표현하였다. 하지만, 이들은 부분-전체 관계 중에서 상당히 제한적인 의미만을 나타내므로, 응용 프로그램에서 원하는 시맨틱을 모두 나타내기에는 한계가 있다. [4]는 ODMG에서 복합객체(composite object)를 지원한다는 점에서 본 논문과 유사하다. 하지만, 본 논문에서는 ODMG-2.0을 지원하는 OODBMS인 **SOP**에 관계성을 구현하였고, 구현에 관련된 이슈들과 그에 따른 해결책을 제시하였다.

3. 확장된 관계성 모델

3.1 확장된 관계성 모델의 정의

이 절에서는 부분-전체 관계성을 나타낼 수 있는 관계성 모델을 제안한다. UML[15]에서 분류한 일반화, 집합, 관련의 세 관계성을 ODMG-2.0의 객체 모델(Object Model)과 연관시켜 살펴보면 다음과 같은 사실을 발견할 수 있다. IS-A관계의 일반화는 객체 모델의 서브 타입(subtyping)과 상속(inheritance)을 통해서 표현될 수 있으며, 관련은 현재 ODMG-2.0의 관계성 모델이 표현하는 것과 같다. 따라서, 본 논문에서는 UML에서의 집합에 해당하는 부분-전체 관계성을 확장된 관계성 모델로 나타내고자 한다.

이미 다른 논문들에서 부분-전체 관계에 대한 관계성 모델을 제안한 것이 있으나, 이들은 부분-전체 관계 중에서 상당히 제한적인 의미만을 나타내거나 혹은 특정한 시스템에만 적합한 모델이다. 따라서, 본 논문에서 제안하는 관계성은 여러 종류의 시맨틱을 제공하여 응용 프로그램에서 서로 관련이 있는 객체들이 그 시맨틱에 알맞은 관계성을 선택할 수 있도록 하였다. 그리고, 이 관계성은 ODMG-2.0 표준을 따르는 모든 객체지향 데이타베이스에서 사용할 수 있는 모델이다.

부분-전체 관계성을 다음의 세 요소로 분류하여 정의하였고, 부분-전체 관계성에서 부분에 해당하는 객체를 부분 객체라고 하고, 그 관계성에서 전체에 해당하는 객체를 전체 객체라고 하였다.

- exclusiveness
- multiplicity
- dependency

모델의 복잡성과 그것을 사용할 때의 효율성 간단함에는 상호 장단점이 있다. 응용 프로그램에서 필요로 하는 관계성의 의미나 행동이 사용되는 분야에 따라서 다르기 때문에 이들을 모두 포함하는 객체들 사이의 관계를 정의하는 것은 모델의 복잡도를 높이며, 효율성을 떨

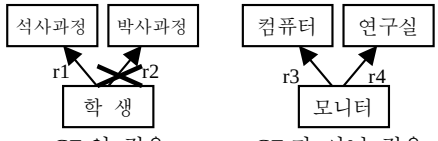
어뜨린다. 관계성에 관련된 이전의 논문들[9,10,12]을 살펴보았을 때 위의 세 요소가 부분-전체 관계를 나타내는 가장 핵심적인 요소로 판단되었다.

이 세 가지 요소 중 **exclusiveness**는 부분 객체가 동시에 여러 전체 객체의 부분이 될 수 있는가를 나타낸다. **multiplicity**는 전체 객체가 가질 수 있는 부분 객체의 수, 부분 객체가 속할 수 있는 전체 객체의 수의 두 가지로 나누어진다. **dependency**는 전체 객체의 존재 여부가 부분 객체에 영향을 미치는 것 부분 객체의 존재 여부가 전체 객체에 영향을 미치는 것의 두 가지로 나누어진다.

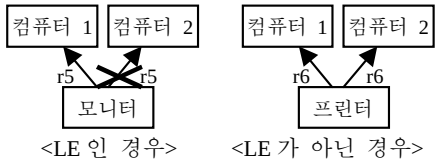
다음에서는 이 세 가지 요소가 각각 어떠한 시맨틱을 가질 수 있는지를 살펴본다.

• exclusiveness는 다음과 같이 세 가지 측면이 있을 수 있다.

{Global-Exclusive, local-Exclusive, Fully-Shared}



(a) Global-Exclusive



(b) Local-Exclusive

그림 1 Exclusiveness의 예제

Global-Exclusive(GE)는 전체 스키마의 모든 관계성의 측면에서 본 것으로, 부분 객체가 단지 하나의 관계성에만 관련되어 전체 객체에 속할 수 있다는 것을 나타낸다.

즉, 동시에 서로 다른 관계성에 대해서 부분 객체가 될 수 없다는 것이다. 그림 1의 (a)에서와 같이 학생은 석사 과정과 박사 과정에 동시에 속할 수 없다. **Local-Exclusive(LE)**는 하나의 관계성의 측면에서 본 것으로, 부분 객체가 하나의 관계성을 통해서 여러 개의 전체 객체에 동시에 속할 수 있는가의 여부를 나타낸다. 그림 1의 (b)에서와 같이 모니터는 두 개의 컴퓨터에 관계성 **r5**로 동시에 속할 수 없으므로 **LE**이지만, 프린터는 두 개의 컴퓨터에 동시에 공유되어 사용될 수 있으므로 **LE**가 아니다. **Fully-Shared (FS)**는 **GE**도 아니고, **LE**도 아닌 경우를 나타낸다. 즉, 동시에 여러 관계성에 대해서 여러 전체 객체의 부분 객체가 될 수 있다는 것이다. **GE**와 **LE**는 서로 양립(orthogonal)할 수 있는 관계이다. 그림 1에서 모니터는 연구실에는 **GE**가 아닌 관계성을 가지고, 컴퓨터에 대해서는 **LE**인 관계성을 가지지만, 프린터는 연구실에는 **GE**인 관계성을 가지지만, 컴퓨터에는 **LE**가 아닌 관계성을 가진다.

• **multiplicity**는 한 관계성에 대해 몇 개의 전체 객체와 부분 객체가 있을 수 있는지를 나타내며 다음과 같은 두 가지 측면이 있다.

{NumPart, NumWhole}

NumPart는 전체 객체가 최대한 몇 개의 부분 객체를 가질 수 있는지를 나타내고, **NumWhole**은 부분 객체가 최대한 몇 개의 전체 객체에 속할 수 있는지를 나타낸다. 여기서 부분 객체가 최대한 몇 개의 전체 객체에 속할 수 있는가는 **exclusiveness**의 LE와 관련이 있다. 속할 수 있는 전체 객체의 최대수가 1인 경우가 LE이기 때문이다.

• **dependency** 는 전체 객체와 부분 객체의 존재 여부가 각각 대응하는 부분 객체나 전체 객체에 달려 있는 것을 나타낸다. **dependency**에는 다음과 같은 세 가지 측면이 있고, 이들은 전체와 부분 모두에 적용이 된다.

{Deletion, Nullify, Blocking}

전체 객체에서 부분 객체에 대해 위의 세 가지 요소를 적용시킬 때의 의미는 다음과 같다.

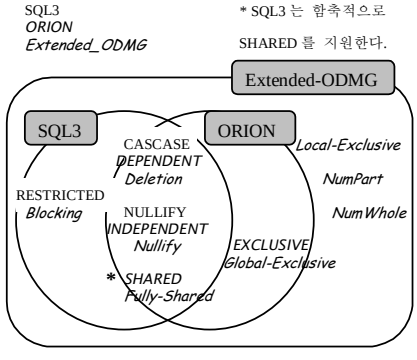


그림 2 SQL3, ORION, 그리고 확장된 ODMG 관계성의 표현력 비교

Deletion은 전체 객체가 삭제되거나, 부분-전체 관계를 나타내는 관계성을 전체 객체가 제거하면 부분 객체도 삭제되는 것이다. Nullify는 전체 객체가 삭제되거나, 부분-전체 관계를 나타내는 관계성을 전체 객체가 제거하여도 부분 객체는 그대로 존재하는 것이다. Blocking는 전체 객체를 삭제하거나, 부분-전체 관계를 나타내는 관계성을 전체 객체가 삭제하려고 할 때 부분 객체가 있으면 전체 객체를 삭제하지 못하는 것이다.

부분 객체에서 전체 객체에 대해 위의 세 가지 요소를 적용시킬 때도 위의 의미가 동일하게 적용된다. 이들이 나타내는 시맨틱의 포괄성을 SQL3와 ORION과 비교한 것이 그림 2이다.

3.2 ODMG C++ 바인딩을 이용한 확장 관계성 모델의 지원

3.2.1 확장된 관계성 타입

앞 절에서 제시한 확장된 관계성 모델을ODMG-2.0의 C++ 바인딩에서 효과적으로 표현하기 위해서 본 논문에서는 템플릿 클래스 d_Rel_Ref를 계승한 새로운 관계성 클래스를 제공하여 이 관계성 클래스를 사용자가 정의하는 클래스에서 내부 타입으로 쓸 수 있도록 하였다. 또한, 콜렉션에 대한 관계성을 나타낼 수 있도록 템플릿 클래스 d_Rel_Set, d_Rel_List, d_Rel_Bag 등을 계승한 클래스도 제공한다. 이처럼 기존의 관계성 클래스를 계승함으로써 두 객체 사이의 참조 무결성을 유지할 수 있고, 객체 핸들러인 d_Ref를 계승받은 d_Rel_Ref 관계성 타입이 갖는 구현상의 장점도 살릴 수 있다.[2] 그리고, 각 관계성이 하나의 클래스와 대응되므로 관계성이 단순히 객체 사이의 구조를 효과적으로 표현하는 것을 넘어서 메소드를 사용하여 한 객체의 행동이 다른 객체에 미치는 영향을 표현할 수 있게 된다.

본 논문에서 제안하는 확장된 관계성 타입에는 크게 두 가지 종류가 있는데, 전체에 해당하는 클래스의 선언 부분에서 자신의 부분 객체를 가리키는 관계성 타입과 이에 대응되는 부분 클래스에서 자신의 전체 객체를 가리키는 관계성 타입이다.

```
template<class T, const char* Member,
        const char* Option>
class d_Part_Ref: public d_Rel_Ref<T,
Member> {
public :
  d_Part_Ref();          // constructor
 ~d_Part_Ref();          // destructor
  d_Part_Ref<T, Member, Option>& operator
=(Ref<T>& from);
  d_Part_Ref<T, Member, Option>& operator
=(void* from);
  void clear();
  void destroyobj();
};
```

그림 3 d_Part_Ref 클래스

전체 객체가 한 관계성에 대해서 하나의 부분 객체를 가질 때 이 부분 객체를 가리키기 위해서 본 논문에서 제공하는 관계성 타입은 d_Rel_Ref 클래스를 계승한 그림 3의 d_Part_Ref 클래스이다. 전체 객체에서 한 관계성에 대해서 자신의 부분 객체가 다수일 경우에 이를 가리키기 위해서 콜렉션 타입을 계승한 클래스들을 제공하는데, 다음은 각각 d_Part_Set 클래스와 d_Part_List 클래스를 계승한 부분 객체를 가리키는 관계성 클래스이다.

```
template<class T, const char* Member, const
char* Option, const char* Max>
class d_Part_Set
: public d_Rel_Set<T, Member> { }
template<class T, const char* Member, const
char* Option, const char* Max>
class d_Part_List
: public d_Rel_List<T, Member> { }
```

부분 객체가 자신이 속해 있는 전체 객체를 가리키기 위해서 본 논문에서 제공하는 관계성 타입에는 템플릿 클래스 d_Rel_Ref를 계승한 d_Whole_Ref 클래스와 콜렉션을 나타내는 클래스들을 계승한 d_Whole_Set과 d_Whole_List와 같은 클래스들이 있다. d_Whole_Ref는 한 관계성에서 하나의 전체 객체에만 속한다는 것을 나타내고, d_Whole_Set과 d_Whole_List들의 한 관계성에서 다수의 전체 객체에 속할 수 있다는 것을 나타낸다. 이들은 앞에서 설명한 d_Part_Ref나 d_Part_Set, d_Part_List 등과 대응되어 사용한다.

```
template<class T, const char* Member,
const char* Option>
class d_Whole_Ref:public d_Rel_Ref<T,
```

2) 객체 핸들러는 지속성 객체를 접근하기 위해서 제공되는 일종의 스마트 포인터이다.

```
Member>{}
  template<class T, const char* Member,
  const char* Option, const char* Max>
  class d_Whole_Set:public d_Rel_Set<T,
Member>{}
  template<class T, const char* Member,
  const char* Option, const char* Max>
  class d_Whole_List:public
d_Rel_List<T,Member> {}
```

const char* Member는 ODMG-2.0에서와 같이 상대 객체에 선언되어 있는 대응되는 관계성 변수의 이름을 알기 위해서 사용되고, const char* Option은 이 관계성들이 원하는 시맨틱을 선택하는데 사용된다그리고, 콜렉션 클래스에만 있는 const char* Max는 multiplicity를 나타내기 위해 사용된다

본 논문의 확장된 관계성 클래스와 기존의 관계성 클래스와의 계승 관계는 그림4와 같다.

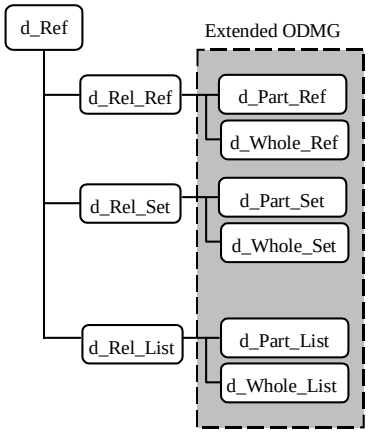


그림 4 관계성 클래스 계승 구조

3.2.2 부분을 가리키는 관계성 타입의 시맨틱

3.2.1 절에서 제시한 관계성 모델의3가지 요소, exclusiveness, multiplicity, dependency를 부분 객체를 가리키는 관계성 타입에서 어떻게 표현할 수 있는지 알아보자.

세 가지 요소 중에서 exclusiveness는 전체 객체와 부분 객체에 동시에 적용되므로 본 논문에서 GE는 부분 객체를 가리키는 타입에서 표현하고 LE는 전체 객체를 가리키는 타입에서 표현한다 이 두 측면을 분리해서 나타내는 이유는 GE와 LE는 양립하는 개념이기 때문이다. 그리고, LE를 전체 객체를 가리키는 타입에서 표현하는 이유는 전체 객체를 가리키는 타입의 multiplicity로 표현될 수 있기 때문이다 LE는 부분에 해당하는 클래스의 선언에서 자신의 전체를 가리키는 관계성 타입이 하나의 전체에만 속한다는 것을 나타내는 d_Whole_Ref이거나, 다수의 전체에 속할 수 있는 콜렉션 타입일 경우에는 전체 객체의 최대의 값이 1일 경우이다. multiplicity와 dependency는 전체 객체와 부분 객체에서 모두 나타나야 하므로 부분 객체를 가리키는 관계성 타입에서는 multiplicity는 전체 객체가 가질 수 있는 부분 객체의 수를 나타내고 dependency는 전체 객체의 존재 여부가 부분 객체에 어떠한 영향을 미치는가를 나타낸다 multiplicity는 콜렉션을 나타내는 타입인 d_Part_Set과 d_Part_List 등에서 템플릿 인자 Max를 사용하여 나타낼 수 있고, exclusiveness와 dependency는 d_Part_Ref와 d_Part_Set, d_Part_List 등에서 템플릿 인자인 Option를 사용하여 나타낸다

이 Option에는 표 1의 여섯 종류가 있다.

표 1 부분 객체에 대한 시맨틱

	Global-Exclusive	Non-Global-Exclusive
Deletion	ED(Exclusive Deletion)	SD(Shared Deletion)
Nullify	EN(Exclusive Nullify)	SN(Shared Nullify)
Blocking	EB(Exclusive Blocking)	SB(Shared Blocking)

이 Option을 살펴보면 앞 절에서 제시한 관계성 모델의 dependency와 exclusiveness 중 GE인 경우와 아닌 경우를 조합한 것이라는 것을 알 수 있다 이것은 부분 객체를 가리키는 관계성에서는 exclusiveness의 GE만을 표현하기 때문이다 여기서 GE가 아닌 경우를 Shared라고 했는데, 3.2.1 절에서 제시한 관계성 모델에서의 FS가 표현되기 위해서는 부분 객체를 가리키는 관계성에서는 시맨틱을 Shared로 하고, 전체 객체를 가리키는 관계성에서는 관계성의 타입이 콜렉션을 나타내는 d_Whole_Set이거나 d_Whole_List이고, 최대 전체 객체의 수가 2 이상이어야 한다

다음에서는 관계성 타입의 시맨틱을 나타내는 각 Option들의 의미에 대해서 알아보도록 하자 이 Option은 하나의 부분 객체를 가리키는 d_Part_Ref와 다수의 부분 객체를 가리키는 d_Part_Set이나 d_Part_List에 모두 해당이 되지만, 여기서는 d_Part_Ref를 예를 들어 설명하도록 하겠다 다른 타입에도 이 의미가 동일하게 적용된다

• **ED(Exclusive Deletion)**는 두 가지 의미를 가진다. 첫째, 부분 객체가 런타임에 단 하나의 부분-전체 관계성을 가지는 것이다. 이것은 전체 객체의 인터페이스 중 참조에 관련된 것이다. 전체 객체가 관계성 **R1**으로 참조하려는 부분 객체가 이미 다른 관계성 **R2**로 다른 전체 객체(자신일 경우도 포함된다.)에 속해있다면 이 부분 객체를 참조할 수 없다. 둘째, 현재 가리키고 있는 부분 객체의 존재 여부가 자신 즉, 전체 객체의 삭제와 관련이 있다는 것이다. 전체 객체가 삭제되거나 전체 객체의 **d_Part_Ref** 관계성이 제거되면 이에 해당하는 부분 객체도 함께 삭제된다.

• **SD(Shared Deletion)**은 **ED**와는 달리 이미 다른 관계성 타입에 의해 부분 객체가 된 객체를 동시에 자신의 부분 객체로 가리킬 수 있다. 따라서, 그 부분 객체가 이미 **Option**이 **SD**이거나 **SN**, **SB**인 **d_Part_Ref** 관계성에 의해서 다른 전체 객체에 속해 있을 경우에는 그 부분 객체를 참조할 수 있다. 하지만, **Option**이 **ED**이거나 **EN**, **EB**인 **d_Part_Ref** 관계성에 의해서 다른 전체 객체에 속해 있을 경우에는 그 부분 객체를 참조할 수 없다. 그리고, 전체 객체가 삭제되거나 전체 객체의 **d_Part_Ref** 관계성이 제거될 때는 그에 속한 부분 객체가 현재 다른 전체 객체에도 속해 있는지를 확인한다. 만약 다른 어떠한 전체 객체에도 속해 있지 않을 경우에는 부분 객체도 함께 삭제된다.

• **EN(Exclusive Nullify)**는 부분 객체가 런타임에 단 하나의 부분-전체 관계성을 가지며, 전체 객체의 존재 여부가 부분 객체의 존재 여부에 영향을 미치지 않는다. 따라서, 전체 객체가 관계성 **R1**으로 참조하려는 부분 객체가 이미 다른 관계성 **R2**로 다른 전체 객체에 속해있다면 이 부분 객체를 참조할 수 없다. 전체 객체가 삭제되거나, **d_Part_Ref** 관계성이 제거되어도 부분 객체는 삭제되지 않는다.

• **SN(Shared Nullify)**는 이미 다른 전체 객체에 속해 있는 부분 객체를 자신의 부분 객체로 할 수 있으며, 전체 객체의 존재 여부가 부분 객체의 존재 여부에 영향을 미치지 않는다. 따라서, 전체 객체를 삭제할 때는 부분 객체와 무관하지만 부분 객체를 참조할 때는 그 부분 객체가 이미 **Option**이 **ED**이거나 **EN**, **EB**인 **d_Part_Ref** 관계성에 의해서 다른 전체 객체에 속해있는가를 검사한 후, 이 관계성에 의해서 다른 전체 객체의 일부가 아닐 경우에는 그 부분 객체를 참조할 수 있다.

• **EB(Exclusive Blocking)**은 현재 가리키고 있는 부분 객체가 런타임에 단 하나의 부분-전체 관계를 가지며, 전체 객체를 삭제하려고 할 때 부분 객체가 있으면, 전체 객체를 삭제하지 못한다는 것을 나타낸다. 따라서, 전체 객체가 관계성 **R1**으로 참조하려는 부분 객체가 이미 다른 관계성 **R2**로 다른 전체 객체에 속해 있다면 이 부분 객체를 참조할 수 없고, 전체 객체를 삭제하려고 할 때는 이 관계성에 속해 있는 부분 객체를 삭제한 다음에야 전체 객체를 삭제할 수 있다.

• **SB(Shared Blocking)**은 이미 다른 전체 객체에 속해 있는 부분 객체를 자신의 부분 객체로 할 수 있으며, 전체 객체를 삭제하려고 할 때 부분 객체가 있으면 전체 객체를 삭제하지 못한다. 따라서, 참조하고자 하는 부분 객체가 이미 **Option**이 **ED**이거나 **EN**, **EB**인 **d_Part_Ref** 관계성에 의해서 다른 전체 객체에 속해 있지 않을 경우에는 그 부분 객체를 참조할 수 있다. 그리고, 전체 객체를 삭제할 때는 이 관계성에 속해 있는 부분 객체를 삭제한 다음에야 전체 객체를 삭제할 수 있다.

3.2.3 전체 객체를 가리키는 관계성 타입의 시맨틱

전체 객체를 가리키는 관계성 타입에서는 부분 객체에서 전체 객체에 대한 **multiplicity**와 **dependency**를 표현하고, **exclusiveness**는 **LE**만을 표현한다. 전체 객체를 가리키는 관계성 타입에서의 **multiplicity**는 부분 객체가 속할 수 있는 전체 객체의 수를 나타내고, **dependency**는 부분 객체의 존재 여부가 전체 객체에 어떠한 영향을 미치는가를 나타낸다. **LE**를 전체 객체를 가리키는 관계성 타입에서 어떻게 표현하는지는3.2.2절에서 설명하였다. **multiplicity**는 콜렉션을 나타내는 타입인 **d_Whole_Set**과 **d_Whole_List** 등에서 템플릿 인자 **Max**를 사용하여 나타낼 수 있고, **dependency**는 **d_Whole_Ref**와 **d_Whole_Set** 등에서 템플릿 인자인 **Option**을 사용하여 나타내며, 이 **Option**에는 표 2의 세 종류가 있다.

표 2 전체 객체에 대한 시맨틱

Deletion	Nullify	Blocking
DT	NF	BK

다음에서는 관계성 타입의 시맨틱을 나타내는 각 **Option**들의 의미에 대해서 알아보도록 하자. 이 **Option**은 하나의 전체 객체에만 속하는 **d_Whole_Ref**와 다수의 전체 객체에 속하는 **d_Whole_Set**이나 **d_Whole_List**에 모두 해당이 되지만, 여기서는

d_Whole_Ref를 예를 들어 설명하도록 하겠다 다른 타입에도 이 의미가 동일하게 적용된다

• DT(DeleTion)은 부분 객체를 삭제하면 전체 객체도 삭제된다. 즉, 전체 객체는 이 부분 객체 없이는 존재할 수 없다는 것을 나타낸다

• BK(BlocKing)은 자신 즉, 부분 객체를 삭제하고자 할 때, 이 객체가 이 BK 시맨틱을 가지고 다른 객체의 부분 객체로 속해 있을 경우는 이 객체를 삭제하지 못한다는 것을 나타낸다 즉, 반드시 전체 객체가 먼저 삭제되거나 전체 객체가 부분-전체 관계성을 없앤 후에야 부분 객체를 삭제할 수 있다

• NF(NulliFy)는 부분 객체의 삭제 여부가 전체 객체와의 관계성을 가지고 있는 것에 독립적이다 즉, 부분 객체가 삭제되어도 전체 객체는 삭제되지 않으며 현재 전체 객체에 속해있어도 부분 객체를 삭제할 수 있다.

3.3 관계성을 사용한 예제

```
extern const char _ED[], _NF[];
extern const char _monitor[], _computer[];
class Computer : public PObject  {
public:
  d_Part_Ref<Monitor, _computer, ED> monitor;
};
class Monitor : public PObject {
public :
  d_Whole_Ref<Computer, _monitor, NF>
computer;
};
const char _ED[] = "ED";
const char _NF[] = "NF";
const char _monitor[] = "Monitor";
const char _computer[] = "computer";
```

그림 5 관계성을 사용한 스키마

그림 5는 확장된 관계성 타입을 사용하여 나타낸 스키마이다. 이 스키마는 컴퓨터와 컴퓨터의 부분인 Monitor와의 관계를 나타낸 것이다 전체 객체에 해당하는 컴퓨터가 부분 객체인 Monitor를 가리키는 관계성 타입은 d_Part_Ref이고, 시맨틱은 ED이다. 시맨틱이 ED인 것은 Monitor는 컴퓨터외의 다른 객체의 부분 객체는 될 수 없는 GE인 것을 나타내고, 컴퓨터가 삭제되면 Monitor도 함께 삭제된다는 것을 나타낸다 부분 클래스인 Monitor가 전체 클래스 Computer를 가리키는 관계성 타입은 d_Whole_Ref이고, 시맨틱은 NF이다. 전체 클래스 Computer를 가리키는 관계성 타입이 d_Whole_Ref인 것은 Monitor는 단지 한 개의 컴퓨터에만 속할 수 있는 LE라는 것을 나타내고, 시맨틱이 NF인 것은 Monitor가 삭제되는 것은 전체 객체인 Computer에 영향을 미치지 않는다는 것을 나타낸다.

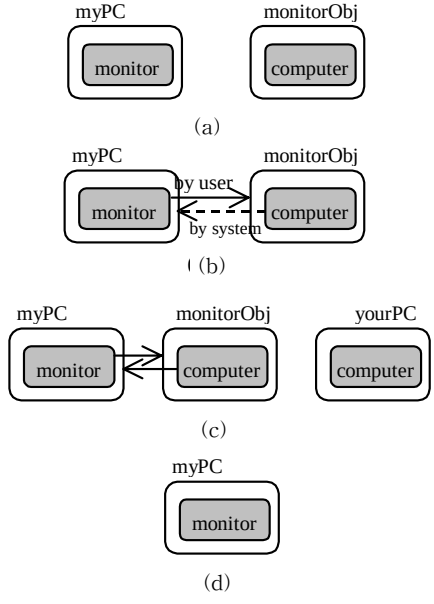


그림 6 관계성을 사용한 예제

이 스키마를 사용하여 전체-부분 관계를 갖는 객체들이 실제로 어떻게 행동하는지 알아보도록 하자 클래스 Computer의 객체인 myPC와 yourPC가 있고, 클래스 Monitor의 객체인 monitorObj가 있다고 하자. 그림 6의 (a)는 myPC와 monitorObj사이에 관계성이 없는 것을 나타낸다. 여기에 대입 연산자인 operator =를 사용하여 monitorObj를 myPC의 부분 객체로 한다

```
myPC.monitor = monitorObj;
```

결과는 그림 6의 (b)처럼 된다. 응용 프로그램에서 myPC의 monitor에 monitorObj를 대입하면, 시스템에서 자동적으로 monitorObj의 computer에 myPC를 대입한다. 만약 객체 monitorObj가 이미 다른 객체의 부분 객체로 속해있는 그림 6의 (b)와 같은 상태에서 객체 yourPC가 다음과 같이 monitorObj를 부분 객체로 하려고 한다고 하자

yourPC.monitor = monitorObj;

하지만 결과는 그림 6의 (c)처럼 yourPC는 monitorObj를 부분 객체로 할 수 없다. monitorObj는 이미 다른 객체의 부분 객체가 되어있기 때문이다. myPC가 다음과 같이 함수 clear()을 사용하여 객체 monitorObj와의 전체-부분 관계를 제거하면 객체 monitorObj도 함께 삭제된다. 삭제되고, 그림 6의 (d)와 같은 상태가 된다.

myPC.monitor.clear();

그림 6의 (b)의 상태에서 객체 myPC를 삭제하는 연산 destroy()를 아래와 같이 수행하면 객체 monitorObj도 함께 삭제된다.

monitorObj.destroyObj();

만약 확장된 관계성 타입을 사용하지 않고ED와 같은 시맨틱을 표현하고자 하면 응용 프로그램에서 응용 프로그램 수준에서 그러한 시맨틱을 표현해 주어야 한다. 이것은 객체들 사이의 관계를 명확히 나타내지 못할 뿐만 아니라, 시맨틱이 바뀌었을 경우 응용 프로그램에서 이에 해당하는 부분을 찾아서 수정해 주어야 한다. 반면, 확장된 관계성 타입을 지원할 경우에는 클래스의 스키마 부분에서 그 타입의Option만을 바꾸면 된다.

4. 관계성 타입의 구현

4.1 소멸자와 대입 연산자

확장된 관계성 타입의 중요한 인터페이스는 삭제와 대입에 관련된 것이다. 이들은 각각 관계성을 나타내는 클래스의 소멸자와 대입 연산자에 의해서 구현이 된다.

소멸자와 대입 연산자의 구현에 대해서 설명하기에 앞서 관계성 타입이 가지는 정보를 스키마 매니저에 등록하는 것에 대해서 설명하도록 하겠다. 응용 프로그램이 수행되는 도중에 현재 객체가 부분 객체를 가지고 있는가, 만약 가지고 있다면 부분 객체에 대한 관계성 시맨틱이 무엇인가, 이 부분 객체가 또 다른 전체 객체에 속해 있는가 등의 부분 객체와 전체 객체에 대한 정보를 알 수 있어야 한다. 이러한 정보를 스키마를 임포트(import)할 때, 스키마 매니저에 등록해 둔다. 스키마 매니저는 등록된 클래스에 대한 정보를 가진 자료 구조를 리스트로 가지고 있고, 이 클래스들은 자신의 멤버에 대한 정보를 가진 자료 구조를 리스트로 가지고 있다. 만약 그림 5의 스키마가 등록되면, 클래스 Computer는 자신의 멤버 중 관계성 d_Part_Ref<Monitor, _computer, ED> monitor;에 대한 자료 구조를 가지게 된다. 이 자료 구조가 가지는 정보는 다음과 같다.

- 멤버 이름 : monitor
- 멤버의 도메인 클래스의 자료 구조에 대한 포인터 : Monitor
- 관계성의 종류 : d_Part_Ref
- 멤버와 대응하는 관계성 변수의 이름 computer
- 관계성의 시맨틱 : ED

다음은 부분 객체를 나타내는 클래스인 d_Part_Ref의 소멸자와 대입 연산자가 어떻게 구현되었는가를 설명한 것이다. 이들은 앞의 스키마 매니저에 등록된 정보를 사용하여 구현하였다.

```
template<class T, const char* Member,
const char* Option>
d_Part_Ref<T, Member, Option>::
~d_Part_Ref()
{          :
  IF ( Option이 'ED' ) {
    가리키고 있는 부분 객체를 삭제한다.
  } ELSE IF (Option이 'SD') {
  IF ( 가리키고 있는 객체의 관계성 중에서
d_Whole_Ref가 있고, 실제로 다른 객체를 가리키고 있지 않다 ) {
        부분객체를 삭제한다.
    }
}
```

알고리즘 1 : 클래스 d_Part_Ref의 소멸자

• **소멸자** 그림 5의 클래스 Computer의 객체인 myPC가 삭제되면, myPC의 부분 객체인 클래스 Monitor의 객체도 함께 삭제되어야 한다. 객체 myPC가 삭제될 때 클래스 Computer의 소멸자가 불리게 되면, d_Part_Ref<Monitor, _computer, ED>의 소멸자도 함께 불리게 되는 점을 이용하여 그림 2의 클래스 d_Part_Ref의 소멸자를 알고리즘 1와 같이 구현하였다.

이 소멸자에서 Option이 EB이거나 SB인 경우는 구현되어 있지 않다. 이에 관해서는 4.2 절에서 설명하였다.

• **대입 연산자** 한 객체를 부분 객체로 대입하려는 경우, 이 부분 객체가 하나 이상의 전체 객체의 부분이 될 수 있는지를 확인해야 한다. 따라서, 그림 2의 클래스 d_Part_Ref의 operator =는 알고리즘 2와 같이 구현하였다.

```
template<class T, const char* Member,
const char* Option>
d_Part_Ref<T, Member ,Option>&
d_Part_Ref<T, Member, Option>
```

```
::operator=(Ref<T>& from)
{              :
IF ( Option이 'ED', 'EN', 'EB' 아니면 ) {
   IF (부분 객체에 해당하는 from이 가리키는
   객체가 현재 다른 객체의 부분 객체가 아니면) {
        from을 부분 객체로 한다.
   }
} ELSE {
   IF (from이 관계성으로 'ED'이거나 'EN', 'EB'인
   전체 객체에 속해 있지 않으면) {
        from을 부분 객체로 한다.
   }
}
```

알고리즘 2 : d_Part_Ref의 operator =

4.2 연산의 전파에 따른 문제점

부분-전체 관계를 나타내는 타입은 한 객체에 대한 연산이 다른 객체로 전파가 될 수 있으므로, 스키마를 잘못 디자인했을 경우에 런타임에 예기치 못한 결과를 가져올 수 있다.

RDBMS에서도 외래 키의 **action**을 지원함으로써 연산 전파에 따른 여러 가지 이상 현상(anomaly)이 발생하므로, 이를 해결하기 위한 방안들이 제시되었다. SQL3([11,7])에서는 이상 현상을 해결하기 위해서 수행시에 연산이 수행되는 순서를 제약하는 방법을 선택하였다. [13]에서는 ODBMS에서 관계성을 지원할 때 일어날 수 있는 이상 현상들을 설명한 후 이를 해결하기 위한 방법을 제시하였다. 이 논문에서 사용한 방법은 디자인된 스키마를 검사하여 문제를 일으킬 수 있는 스키마를 사용자에게 알려주는 것이다.

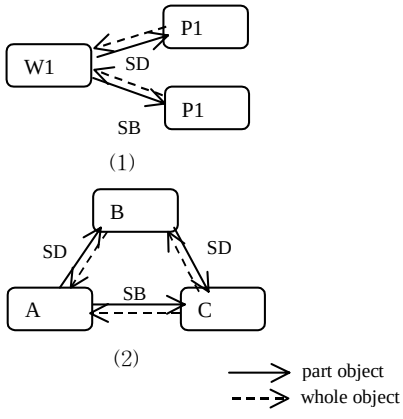


그림 7 연산 전파에 따른 문제점

본 논문에서 정의한 관계성의 시맨틱 중 연산이 다른 객체로 전파되면서 이상 현상을 일으킬 수 있는 시맨틱은 **Dependency**이다. **Dependency**의 **Deletion**과 **Blocking**이 **Exclusiveness**의 **Shared**와 결합하여 함께 사용될 때, 다음과 같은 문제를 일으킬 수 있다.

• 부분 객체만 삭제되는 경우

그림 7 (1)의 스키마를 생각해 보자. 객체 **W1**를 삭제하려는 연산이 수행되면 객체 **W1**은 시맨틱이 **SB**인 관계성을 가지고 있으므로, 전체 객체인 **W1**과 이 객체의 모든 부분 객체들이 삭제되지 않아야 한다. 즉, 객체 **W1**에 삭제 연산을 하지 않은 것과 같은 결과를 가져야 한다.

하지만, 실제로 객체 **W1**을 삭제하려고 할 때, 시맨틱이 **SD**인 관계성의 소멸자가 먼저 불려지고, 다음에 시맨틱이 **SB**인 관계성의 소멸자가 불려지는 경우를 살펴보자. 이 때에는 전체 객체와 부분 객체 **P2**는 삭제되지 않았는데, 부분 객체 **P1**만 삭제되는 잘못된 결과가 일어나게 된다.

• 수행 순서에 따른 결과의 다름

그림 7 (2)와 같은 스키마를 생각해 보자. 객체 **A**가 삭제될 때 부분 객체들의 소멸자가 호출되는 순서에 따라서 객체 **A**에 대한 삭제 연산의 수행 결과가 달라지게 된다. 만약 **SD** 시맨틱을 가진 관계성의 소멸자가 먼저 호출되면 **객체 A - 객체 B - 객체 C**의 순서로 소멸자가 불리게 된다. 이 경우는 **객체 C**가 삭제된 후 **객체 A**의 **SB** 시맨틱을 가진 관계성의 소멸자가 불리므로, 세 개의 객체가 모두 삭제된다. 하지만, 객체 **A**가 삭제될 때 **SB** 시맨틱을 가진 관계성의 소멸자가 먼저 호출되면 객체 **C**가 존재하므로 객체 **A**를 삭제할 수 없다. 따라서, 객체 **A**의 **SD** 시맨틱을 가진 관계성의 소멸자는 호출되지 않고, 세 개의 객체는 모두 삭제되지 않는다.

부분 객체의 소멸자가 불리는 순서는 사용자가 스키마를 정의할 때, 부분 객체를 가리키는 관계성의 순서에 따라 달라지므로, 이는 사용자의 의도와는 상관없이 사용자가 이러한 원리로 작동한다는 것을 모를 때 그 결과가 달라지게 된다.

이러한 문제를 해결하기 위해서 본 논문에서는 시맨틱이 **SB**인 관계성의 소멸자가 다른 시맨틱을 가진 관계성의 소멸자보다 먼저 호출되도록 하였다. 그림 7의 (1)과 같은 경우에 객체 **W1**이 삭제될 때 SB 시맨틱을 가진 관계성의 소멸자가 먼저 호출된다면, 전체 객체 **W1**은 삭제되지 않았는데 부분 객체 **P1**만 삭제되는 경우는 발생하지 않는다. 그림 7의 (2)와 같은 경우에도

객체 A가 삭제될 때 SB 시맨틱을 가진 관계성의 소멸자가 먼저 호출된다면, 항상 세 개의 객체 모두 삭제되지 않는다.

이것을 구현하고자 하면 객체가 삭제될 때 그 객체의 멤버들의 소멸자가 호출되는 순서를 바꾸어야 한다. C++에서는 멤버들이 선언된 순서의 반대로 소멸자가 호출된다. 그리고, 계승 관계에 있는 클래스에서는 자식 클래스의 소멸자가 먼저 호출되고, 멤버들의 소멸자가 호출되고, 마지막으로 부모 클래스의 소멸자가 호출된다. 소멸자가 호출되는 순서를 바꿀 수는 없으므로, 본 논문에서는 시맨틱이 SB인 관계성이 소멸자에서 해야 하는 일을 객체를 삭제하기 전에 먼저 하도록 하였다. 4.1 절에서 설명한 소멸자에서는 관계성의 시맨틱이 SB인 경우에는 아무 것도 수행하지 않는다. 그 대신 객체를 삭제하는 destroyobj() 함수에서 실제로 객체를 삭제하는 delete가 호출되기 전에 스키마 매니저에서 시맨틱이 SB인 관계성을 찾는다. 만약, 시맨틱이 SB인 관계성이 현재 객체를 가리키고 있으면, 함수를 더 이상 수행하지 않고 리턴함으로써 객체가 삭제되지 않도록 하였다. destroyobj() 함수는 알고리즘 3과 같이 구현하였다.

```
int RefAny::destroyobj(void)
{                    :
FOR( 삭제할 객체내의 관계성 변수 )
 IF(관계성 시맨틱이 'SB' 또는'EB') {
  IF(관계성이 현재 객체를 가리킨다) {
    return ErrorMsg;
  }
 }
 객체를 삭제한다.
}
```

알고리즘 3 destroyobj() 함수

시맨틱이 EB인 경우는 소멸자에서 구현해도 되지만, 구현상의 편의를 위해서 SB와 함께 처리하였다. 전체 객체를 가리키는 관계성 타입인 d_Whole_Ref의 시맨틱 중 BK도 destroyobj() 함수에서 d_Part_Ref의 SB의 경우처럼 구현하였다.

스키마 매니저는 자신의 멤버 뿐 아니라 상속받은 멤버에 대한 정보도 함께 가지고 있다. 부모 클래스에 SB 시맨틱을 가진 관계성이 있고 자식 클래스에 SD 시맨틱을 가진 관계성이 있어도, destroyobj() 함수에서 객체를 삭제하는 delete가 수행되기 전에 부모 클래스의 SB 시맨틱을 가진 관계성을 찾아서 리턴 한다. 따라서, 클래스가 계승될 경우에도 연산 전파에 따른 문제는 발생하지 않는다.

5. 결 론

본 논문에서는 ODMG-2.0의 C++ 바인딩을 확장하여 객체지향 데이타베이스에서의 **부분-전체 관계**를 지원하는 관계성의 디자인과 구현에 대하여 설명하였다. 관계성을 디자인할 때의 원칙은 ODMG-2.0 표준에 호환하는 타입을 제공하는 것과 사용자가 제공된 타입 중에서 자신의 응용프로그램에 적합한 타입을 선택하여 쉽게 사용할 수 있도록 하는 것이다. 부분-전체 관계성의 모델은 **exclusiveness**, **deletion**, **multiplicity**의 세 가지 요소를 사용하여 나타내었고, 전체 객체가 부분 객체에 대해서 요구하는 것과 부분 객체가 전체 객체에 대해서 요구하는 것 양쪽 모두를 나타내도록 하였다. 이 모델을 ODMG-2.0의 C++ 바인딩에서 나타낼 수 있는 사용자 인터페이스를 설명하고, 이 인터페이스의 구현에 대해서 설명하였다. 그리고, 전체 객체와 부분 객체가 상대 객체에 영향을 미침으로써 발생할 수 있는 문제점을 지적하고, 이를 해결할 수 있는 방안을 제시하였다.

참 고 문 헌

[1] 안정호, 이강우, 송하주, 김형주. "Soprano: 객체 저장 시스템의 설계 및 구현". *정보과학회 논문지(C)*, 2(3):243-255, 1996

[2] A. Albano, G. Ghelli, and R. Orsini. "A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language," *Proceedings of International Conference on Very Large Data Bases*, pages 565-575, Sept. 1991.

[3] P. A .Bernstein, "Repositories and Object Oriented Databases," *SIGMOD Record*, pages 88-96, 1998

[4] E. Bertino "Extending ODMG Object Model with Composite Objects," *Proceedings of International Conference in Object-Oriented Programming: Systems, Languages, and Applications, pages 259-270, 1998*

[5] R. G. G. Cattell and D. K. Barry, "The Object Database Standard : ODMG 2.0," *Morgan Kaufmann Publishers, Inc*, 1997

[6] E. F. Codd, "A relational model for large shared databases". *Communication of ACM*. 13(6):377-387, 1970

[7] Database Engineering, IEEE Computer Society, vol. 8, no. 4. December 1985 special issue on Object-Oriented Systems (edited by F. Lochovsky)

[8] M. Halper and J. Celler and Y. Perl, "Integrating a Part Relationship into an Open OODB System

using Metaclasses," *Proceedings of the Third International Conference on Information and Knowledge Management*, pages 10-17, Dec.1994

[9] B. M. Horowitz, "A Run-time Execution Model for Referential Integrity Maintenance". *Proceedings of International Conference on Very Large Data Bases*, pages 548-556, 1992.

[10] H. V. Jagadish, "Integrity Maintenance in an Object-Oriented Database". *Proceedings of International Conference on Very Large Data Bases*, pages 469-480, 1992.

[11] W. Kim, "Composite Object Support in an Object-Oriented Database System". *Proceedings of International Conference in Object-Oriented Programming: Systems, Languages, and Applications*, pages 118-125, Oct. 1987

[12] V. M. Markowitz, "Safe Referential Integrity Structures In Relational Databases". *Proceedings of International Conference on Very Large Data Bases*, pages 123-132, Sept. 1991.

[13] J. Peckham and B. MacKellar and M. Doherty, "Data Model for Extensible Support of Explicit Relationships in Design Databases," *The VLDB journal*, 4(2):157-191, 1995.

[14] J. Peckham and F. Maryanski and S. A. Demurjian. "Toward the Correctness and Consistency of Update Semantics in Semantic Database Schema". *IEEE Transaction on Knowledge and Data Engineering* 8(3):503-507, 1996.

[15] J. E. Rumbaugh. "Relations as Semantic Constructs in an Object-Oriented Language". *Proceedings of International Conference in Object-Oriented Programming: Systems, Languages, and Applications*, pages 466-481, 1987.

[16] J. E. Rumbaugh, I Jacobson, G. Booch. "The Unified Modeling Language Reference Manual". *Addison Wesley Publishers, inc.*

이 현 주

1997년 2월 한국과학기술원(KAIST) 전산학과 학사 졸업. 1999년 2월 서울대학교 컴퓨터공학과 석사 졸업. 현재, (주) 자이닉스 부설 연구소 근무

송 하 주

1993년 2얼 서울대학교 컴퓨터공학과 졸업. 1995년 2월 서울대학교 컴퓨터공학과 졸업 (공학석사). 1995년 3월 ~ 현재 서울대학교 컴퓨터공학가 박사과정. 관심분야는 트랜잭션 ttmxpa, 객체지향 시스템

이 상 원

1991년 서울대학교 컴퓨터공학과 학사 졸업. 1994년 서울대학교 컴퓨터공학과 석사 졸업 1999년 서울대학교 컴퓨터공학과 박사. 1999년 ~현재 한국 오라클 근무. 관심분야는 데이터베이스, 데이터웨어하우스, 컴퓨터보완

김 형 주

1982년 서울대학교 컴퓨터공학과 졸업. 1985년 8월Univ. of Texas at Austin, 전자계산학 석사. 1988년 5월 Univ. of Texas at Austin, 전자계산학 박사. 1988년 5월 ~ 9월 Univ. of Texas at Austin. Post-Doc. 1988년 9월 ~ 1990년 12월 Georgia Institute of Technology, 부교수. 1991년 1월 ~ 현재 서울대학교 컴퓨터공학가 부교수. 관심분야는 객체지향 시스템 사용자 인터페이스, 데이타베이스