SOP 객체지향 데이타베이스 시스템에서의 객체 버전의 지원

(Object Versioning in SOP Object-Oriented DBMS)

이 상 원 † 김 형 주 ††

(Sang Won Lee) (Hyung Joo Kim)

요 약 본 논문에서는 ODMG-93 표준안을 지원하는 객체지향 데이타베이스 시스템 SOP상에서 객체 버전 가능을 구현한 'SOP 객체 버전 시스템의 설계와 구현 내용을 다룬다. 사용자는 ODMG C++ OML 환경에서, 클래스 라이브러리 형식으로 제공되는, 객체 버전 시스템의 API를 통해 버전 기능을 이용할 수 있다. 이 SOP 객체 버전 시스템의 설계상의 중요한 특징은 ODMG-93과의 호환(compliance)이다. 즉, SOP ODMG-93의 C++ OML 인터페이스의 세만틱을 보존하면서 이를 버전 기능의 지원을 위해 최대한 활용하고자 했다. 그리고, SOP 객체 버전 시스템에서 지원하는 객체 버전 모델은, 다양한 응용분야에 SOP 객체지향 데이타베이스 시스템을 이용할 수 있게 하기 위해(applicability), 기본적인 기능들을 제공한다.

Abstract In this paper, we describe the design and implementation of SOP Object Version System which is based on SOP, an ODMG-compliant OODBMS. To support object versioning in ODMG C++ OML environment, SOP Object Version System provides a set of APIs(application programming interfaces) as a class library. One important design goal of SOP Object Version System was to achieve the full compliance with ODMG-93 standard. That is, in designing the class library, we try to utilize the ODMG-93 C++ OML interface as much as possible, while preserving its semantics as defined in ODMG-93. Our object version model follows the basic functionalities of the main-stream object version models. Although this approach burdens programmers with the management of the application-specific version policy, it helps users to apply SOP Object Version System to diverse application areas.

1. 서 론

데이타베이스의 객체는 데이타베이스에서 모델링하고자 하는 특정 실세계의 개체를 나타낸다. 그런데, 응용에 따라서는 이들 개체를 다양한 측면, 예를 들어 시간에 따른 상의 변화, 하나의 특정 디자인 개체에 대한 여러 대안들(alternatives)을 데이타베이스에서 직접 모델링할 필요가 있다. 하나의 버전(version)은 한 데이타베이스 객체의 특정 상태나 대안을 일컫는다[1].

객체 버전은 CAD, CASE 분야나 시간 데이타베이스 분야 등 다양한 분야에서 필요로 하는 개념으로, 1980년 중반 객체지향 데이타베이스의 등장과 더불어 데이타베이스 시스템에서 반드시 지원해야 하는 기능으로 주목을 받았다[10,18] 그리고, 최근에 들어서 새로이 부상하고 있는 데이타베이스 응용분야들, 예를 들어 Repository[4,19]나 WWW[3,5] 등에서도 버전 기능을 필요로 하고 있다. 따라서, Exodus, Itasca, ObjectStore, Objectivity, O2, Ode 등 이들 새로운 응용분야

· 본 연구는 과학기술처 지원 과제, "웹 트랜잭션 서버를 위한 객체지향 콤포넌트 기술개발"과 정보통신부 과제 "미래로/DB 통합 소프트웨어 시스템을 위한 SRP 확장"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 30일
심사완료 : 1998년 3월 27일

1) 엄밀하게는 본 논문에서 다루는 버전 기능은 사용자 수준의 버전 기능이고, 시스템 레벨에서 동시성제어(concurrency control)[17]나 긴 트랜잭션(long transaction)[12]의 지원과 관련한 내용은 본 논문의 범위를 벗어남.

들의 지원을 위해 개발된 대부분의 객체지향 데이타베이스 시스템에서 버전 기능을 지원하고 있다[2,8, 9,14,15,16].

본 논문은 서울대학교에서 개발된 객체지향 데이타베이스 시스템 SOP(*SNU OODBMS Platform*)에서 객체 버전 기능의 설계와 구현 내용을 다룬다. SOP는 ODMG(Object Data Management Group)에서 제안한 객체지향 데이타베이스 표준안 ODMG-93[6]의 데이타 정의어 ODL(Object Definition Language), 데이타 질의어 OQL(Object Query Language)및 데이타 조작어 C++ ODL/OML 바인딩을 지원한다(ODMG-compliant). SOP의 아키텍처를 구성하는 주요 모듈로는 객체 저장 시스템(Soprano[1]), 비용기반 질의 처리및 최적화 모듈(Sopoqles), C++ ODL/OML 전처리기(preprocessor)등으로 이루여져 있다. 본 논문에서 기술하는 SOP의 버전 기능(이하 'SOP 객체 버전 시스템'이라 한다.)은 현재로는 사용자들이 ODMG C++ ODL/OML 바인딩을 이용해서 버전 기능을 이용할 수 있도록 지원한다. SOP 객체 버전 시스템은 클래스 라이브러리 형식으로 지원되는데, 사용자는 이 버전 기능을 사용하는 응용 프로그램을 이 클래스 라이브러리와 링크(link)해서 사용할 수 있다.

본 논문의 구성은 다음과 같다. 우선 2장에서 SOP에서의 객체 버전을 설계할 때 고려한 목표를 기술하고 3장에서는 SOP의 객체 버전 모델을 설명한다. 4장에서는 SOP 객체 버전 시스템에서 제공하는 사용자 인터페이스와 그 사용을 예를 통해 설명하고, SOP 객체 버전 시스템의 구현과 관련한 자세한 내용을 5장에서 설명한다. 6장에서 타 상용/연구용 객체 지향 데이타베이스 시스템의 버전 관련 연구를 살펴보고, 7장에서 결론은 맺는다.

2. SOP 객체 버전 시스템의 설계 목표

다음은 버전 기능의 지원과 관련하여, SOP 객체 버전 시스템에서 고려한 설계 목표들을 나열한 것인데, 이 장에서는 이들 각각에 대해 간단히 설명하겠다

1. ODMG 데이타베이스 표준과의 호환
2. 객체저장 시스템 Soprano에 대한 영향의 최소화
3. 일반적인 버전 메카니즘의 지원

2.1 ODMG 데이타베이스 표준과의 호환(ODMG-compilant)

ODMG C++ OML(Object Manupulation Language)에서는 데이타베이스에 저장되는 지속성 객체(persistent object)들을 C++ 환경에서 접근해서 조작하는 인터페이스를 제공하고 있다. 그런데, 버전 객체들도 일종의 지속성 객체이기 때문에, 버전 객체들에도 ODMG C++ OML에서 지속성 객체 조작을 위한 인터페이스를 그대로 적용할 수 있어야만 한다. 예를 들어, ODMG C++ OML에서는 지속성 객체들에 대한 참조(reference)는 **Ref**라는 스마트 포인터(smart pointer)를 이용하는데, 버전 객체에 대한 참조도 마찬가지로 이 **Ref**를 통해서 이루어질 수 있도록 할 수 있어야 한다. 그리고, 3장에서 설명할 SOP 버전 모델들의 각 기능의 구현과 관련해서는 의미적으로 적합한 ODMG C++ OML 인터페이스를 최대한 활용한다. 예를 들어, 특정 버전의 내용에 대한 변경을 막기위해 ODMG C++ OML의 **markmodified()**라는 지속성 객체의 멤버함수를 이용한다. 이와 관련한 내용은 5장에서 자세히 설명하겠다. 이와 같이 ODMG 표준과의 호환을 고려하는 이유는 기존의 ODMG C++ OML 사용자들이 추가된 버전기능의 사용을 위해 새로 익혀야할 인터페이스를 최소화하기 위해서이다.

2.2 Soprano 객체저장 시스템에 대한 영향의 최소화

SOP 객체 버전 시스템 설계의 두번째 목표는, ODMG와의 호환이라는 목표를 만족시키면서 동시에 객체 버전 기능을 추가로 인한 일반 지속성 객체의 연산시에 발생할 수 있는 성능의 저하를 최소화 하는데 있다. 자세한 내용은 5.4절에서 설명하겠다. 그리고, 버전 기능의 구현과 관련해서, Soprano 객체 저장 시스템의 변경사항및 인터페이스를 최소화해서 추후의 기능 확장 및 변경을 용이하게 하고자 한다.

2.3 일반적인 버전 메카니즘의 지원

SOP 객체 버전 시스템은 몇몇 응용들에서 자주 사용되는 특수한 버전 정책(policy)을 지원하는 대신, 일반적인 버전기능의 메카니즘(mechanism)을 제공을 목표로 한다 이는, 비록 응용 프로그램의 작성시에 버전의 세만틱 관리를 위한 사용자의 부담이 늘어나는 단점이 있지만, 특정한 버전 정책이 몇몇 응용 분야들에서 요구되는 세만틱에 배치되는 문제를 막기 위한 것이다. 그리고, 앞에서 지적한 사용자의 버전 세만틱 관리의 부담과 관련해서는, Objectivity[16], Itasca[9] 등에서 제공하는 사용자에 의한 버전 세만틱의 재단(customization)과 유사한 기능을 향후에 추가함으로써 해결하고자 한다.

3. SOP 버전 모델의 구성요소

이 장에서는 그림 1을 중심으로 SOP 객체 버전 시스템에서 지원하는 버전 모델의 각 구성요소 - 버전 객체의 생성, 버전의 유도, 버전 객체의 삭제, 버전 상태,

버전에 대한 동/정적 바인딩, 버전 객체의 탐색 - 를 설명한다.

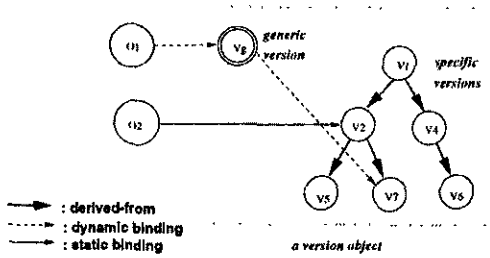

그림 1 버전 예

3.1 버전 객체 및 버전 클래스

SOP 객체 버전 시스템에서의 버전 객체(versionable object)는 지속성 객체(persistent object)[2]로써, 하나의 생성객체(generic version)와 하나 이상의 버전(specific version)들로 이루어져 있다. 이때 이들 버전들사이에는 유도관계(derived-from relationship)가 성립한다. 그림 1에서 점선에 둘러싸인 부분이 하나의 개념적인 버전 객체를 나타낸다. 두개의 원으로 표시된 객체가 생성객체이고 나머지의 v_i들이 버전 객체를 이루는 버전(specific version)들이다. 각 버전들은 지속성 객체들과 마찬가지로 자신의 유일한 객체식별자를 갖는다.

SOP 객체 버전 모델은, Orion[11]에서처럼, 버전화 가능(versionable) 클래스와 일반(non-versionable) 클래스를 구분한다. 특정 클래스를 버전화 가능 클래스로 선언하기 위해서는, ODMG의 지속성 클래스 **PObject**에서 직접 상속받는 대신에, 다음 장에서 나와있듯이 SOP 객체 버전 시스템에서 제공하는 클래스 ***Version***에서 상속받아야 한다. 하지만, ODMG에서 제공하는 객체의 접근및 조작과 관련한 인터페이스들은 버전화 가능 클래스와 일반 클래스의 객체들에 공통적으로 사용할 수 있다.

3.2 버전 객체의 생성

버전 객체의 생성은, ODMG에서 제공하는 인터페이스 ***new()***를 통해 버전화 가능 클래스의 객체를 생성함으로써 이루어지는데, 이때 해당 버전 객체의 생성객체와 하나의 특정 버전, 즉 루트 버전(root version)이 생성된다. 특히, 생성객체는 해당 버전 객체의 디폴트 버전, 루트 버전, 버전들 사이의 유도관계 등의 정보를 유지한다.

2) A versionable object is-a persistent object.

3.3 새로운 버전의 유도

SOP 객체 버전 모델에서는 이미 생성된 하나의 버전 v_i에서 새로운 버전 v_j의 유도를 허용한다. 이때, v_i를 부모(parent) 버전, v_j를 자식(child) 버전이라 한다. 하나의 부모 버전에서 둘 이상의 자식 버전을 유도할 수 있다. 하지만, 하나 이상의 부모 버전에서 새로운 버전의 유도, 즉 버전 합병(version merging)은 지원하지 않는다. 이와 같은 버전들 사이의 유도관계 정보는 생성객체에서 관리되는데, 이 버전들사이의 유도 관계를 VDH(Version Derivation Hierarchy)라 한다. SOP 객체 버전 모델에서는 버전 합병을 지원하지 않기 때문에 SOP 객체 버전 모델에서는 트리구조(tree)의 VDH 만을 지원한다. 그리고, 가장 최근에 유도되어진 버전이 디폴트 버전이 되는데, 사용자의 필요에 의해 명시적으로 다른 버전을 디폴트 버전으로 지정하는 경우 바뀌게 된다.

3.4 버전 상태 지원

SOP 버전 모델에서 각 버전들은 **FROZEN**과 **WORKING**의 두 버전 상태(version status)를 갖는다[3]. **FROZEN** 상태의 버전은 버전 내용의 갱신이 불가능하고 **WORKING** 버전은 변경이 가능하다 SOP 버전 모델에서는, 2장에서 밝힌 바와 같이, Orion 객체 버전 모델 등에서 버전의 상태와 관련하여 제공하는 특정 정책(policy)과 달리, 버전의 상태 변경은 전적으로 사용자의 책임이다. 예를 들어, Orion 객체 버전 모델에서는 특정 버전에서 새로운 버전이 유도되었을 경우, 부모 버전은 자동적으로 변경이 불가능한 상태로 바뀌지만, SOP 객체 버전 모델서는 부모 버전의 변경을 허용하지 않고자 하는 경우, 사용자자 명시적으로 부모 버전을 **FROZEN** 상태로 바꾸어야 한다.

3.5 버전 객체에 대한 참조(reference)

SOP 객체 버전 모델에서는, 앞에서 언급한 바와 같이, 버전 객체에 대해 생성객체를 통한 동적 바인딩(dynamic binding)과 특정 버전에 대한 정적 바인딩(static binding)을 모두 지원한다 버전 객체의 생성시에 생성객체의 객체식별자가 리턴된다. 다른 객체에서 이 객체식별자를 통해 참조하는 경우 동적 바인딩이 이루어진다. 그리고, 이전의 버전에서 새로운 버전의 유도시에는 새로 생성된 버전에 대한 객체식별자가 리턴된다. 객체식별자를 통해 일반 지속성 객체를 참조하듯이, 다른 객체에서 특정 버전을 해당 버전의 객체식별자를

3) 5장에서 데이타베이스의 특정 객체는 구현상의 이유로 DUMMY 상태를 갖는다. 그러나, 이는 구현상의 문제이고 모델상에서는 두가지의 버전 상태를 지원한다

통해 참조하는 경우를 정적 바인딩이라 한다. 그림 1에서 객체 $o1$은 v_g를 통해 버전 객체에 대해 동적 바인딩(이때, v_7이 디폴트 버전이라 가정하자.)을 이루고, 객체 $o2$는 v_2에 대해 정적 바인딩을 이루고 있다. 이후에 v_7이 아닌 다른 버전이 디폴트 버전이 되는 경우 $o1$에서 버전 객체에 대한 참조는 v_7이 아닌 다른 버전으로 이루어진다.

3.6 버전의 삭제

우리가 아는 범위에서, Orion을 제외한 버전 관련 연구들에서 버전 삭제와 관련한 내용을 찾아보기가 힘들다. Orion에서는 생성객체과 일반 버전에 대한 삭제를 지원하고 있는데, 생성객체의 삭제시에는 해당 생성객체뿐만 아니라, 버전 객체내의 모든 버전을 지운다. 특정 버전 vd를 지우는 경우에 해당 버전은 지우지만 생성객체의 버전 계승 구조상에는 vd가 계속 남아 있다. SOP 객체 버전 모델은 Orion과 마찬가지로 버전 객체나 특정 버전의 삭제를 지원한다. 생성객체를 통해서 버전 객체를 삭제하는 경우 해당 생성객체뿐만 아니라, 모든 버전들도 함께 삭제한다. 그리고, 특정 버전 vi만을 삭제할 수도 있다. 버전 vi가 VDH 상에서 단말 노드(leaf node)인 경우, VDH 상에서 부모 버전 v_p과 v_i사이의 유도 관계 정보도 함께 삭제된다. 버전 v_i가 VDH 상에서 내부 노드(internal node)인 경우, v_i의 부모 버전 v_p가 v_i의 모든 자식 버전 v_{ci}의 새로운 부모 버전이 된다. v_i가 루트 버전인 경우 v_i의 각 자식버전 v_{ci}를 루트로 하는 포리스트(forest) 구조를 이루게 된다. 예를 들어, 그림 1에서 특정 정적 바인딩을 통해서 v_1을 삭제하게 되는 경우 VDH의 구조는 v_2와 v_3를 각각 루트로 하는 두개의 버전 트리의 집합이 된다.

3.7 VDH 탐색

SOP 객체 버전 모델에서는 사용자로 하여금 생성객체 및 특정 버전을 통해서 특정 버전 객체의 VDH의 탐색을 허용한다. 다음은 SOP 객체 버전 모델에서 VDH 탐색을 위해 제공하는 기본 기능들이다.

- 부모 버전 탐색: 특정 버전에서 자신의 부모 버전을 탐색할 수 있다. 만일 해당 버전이 루트 버전인 경우에는 **NIL**이 리턴된다.
- 형제 버전 탐색: 현재 버전의 형제 중에서 바로 손위 형제 버전과 바로 손아래 형제 버전에 대한 탐색을 허용한다. 손위 형제에 대한 탐색의 경우, 자신이 가장 손위일 경우, 그리고, 손아래 형제에 대한 탐색의 경우, 자신이 가장 아래일 경우 각각 **NIL**을 리턴한다.
- 자식 버전 탐색: SOP 객체 버전 모델에서는 현재 버전의 모든 자식 버전을 탐색할 수 있다. 버전의 자식 버전은 하나 이상이 가능하므로 모든 자식 버전에 대한 인터페이스는 우선 가장 오래된 자식 버전을 리턴하고 다음 자식에 대해서는 형제 버전에 대한 탐색으로 이루어진다. 단말 버전인 경우에는 **NIL**을 리턴한다
- 시간적인 생성 탐색: 버전이 생성된 시간적인 순서를 따라, 시간적으로 가장 먼저 생성된 버전, 가장 최근 버전, 특정 버전의 이전 혹은 이후 버전에 대한 탐색을 지원한다.

이때, 생성객체를 통한 동적 바인딩의 경우 위의 VDH 탐색은 모두 동적으로 바인딩되어 있는 디폴트 버전에 대한 결과가 리턴된다.

4. SOP 객체 버전 시스템의 API 및 사용 예

이 장에서는 앞에서 설명한 SOP 객체 버전 모델을 지원하기 위해 본 연구에서 구현한 SOP 객체 버전 시스템인 클래스 라이브러리에서 제공하는 버전 관련 API(Application Program Interfaces)와 그 사용 예를 설명한다.

4.1 SOP 객체 버전 시스템의 API

앞에서 기술한 바와 같이, SOP 객체 버전 모델에서는 그림 2의 **Version** 클래스로부터 유도되는 클래스들이 버전화 가능 클래스가 된다.(C++의 템플릿([20]) 타입으로 선언된 것에 관해서는 다음 장을 참조하기 바란다.) SOP 객체 버전 기능은 이 **Version** 클래스와 다음 장에 설명된 **GenericVersion** 클래스의 라이브러리 형태로 제공된다. 그림에서 보여지듯이 **Version** 클래스는 SOP에서 제공하는 지속성 클래스인 **PObject**의 하위클래스로 선언되어 있다. SOP 객체 버전 모델의 버전 생성과 삭제는 이 **Version** 클래스의 생성자와 소멸자를 통해서 제공된다. 이외의 나머지 버전 관련 인터페이스는 **Version** 클래스의 멤버함수로 제공되고 있다. 이들 각각은 앞장의 버전 모델에서 설명한 버전의 유도, VDH 탐색, 생성객체에 대한 참조, 그리고 버전의 상태와 관련한 인터페이스들이다. 이들 중 많은 인터페이스들이 결과 객체(생성객체나 버전)에 대한 객체 식별자를 ODMG에서 제공하는 스마트 포인터인 **Ref<T>** 타입으로 리턴한다[4] ODMG 모델에서 C++ 언어에 대한 바

4) 엄밀하게는 Ref와 객체식별자는 차이가 있지만, ODMG C++ OML 환경에서 Ref는 개념적으로 객체 식별자의 역할을 하기 때문에 본 논문에서는 객체식별자 대신에 Ref를 사용하겠다

인딩은 스마트 포인터인 **Ref<T>**에 기반한다. 이는 지속성 클래스 **T**의 인스턴스 객체에 대한 참조를 가능하게 한다. 즉, 각 클래스 **T**에 대해 정의된 **Ref<T>**를 통해, **T**의 인스턴스 객체를 참조할 수 있다. 그림 2의 두 인터페이스 **defaultVer()**, **genericObj()**의 역할은 다음과 같다. **defaultVer()**는 해당 버전 객체의 디폴트 버전에 대한 **Ref<T>**를 리턴하고, **genericObj()**는 생성객체에 대한 **Ref<T>**를 리턴한다. **genericObj()**를 통해 얻은 생성객체의 **Ref<T>**는 SOP 객체 버전 모델의 동적 바인딩을 가능하게 해준다.

클래스 **Version**의 멤버 **ver_num**는 버전 객체내에서 해당 버전의 버전 번호[5]를, 멤버 **genver**는 자신의 생성객체를 가리킨다. 이들 멤버의 용도에 대해서는 5장에서 설명한다.

```
// 헤더화일 version.h
enum version_flag { WORKING, FROZEN, DUMMY };   // 버전 상태 flag

template<class T>
class Version: public PObject {          // 템플릿 클래스 Version
public:
    // 버전 생성자 및 소멸자
    Version(void);                       // 버전 생성
    Version(T*);                         // 복사 생성자
    ~Version();                          // 버전 삭제

     Ref<T>  derivever();                // 새로운 버전 유도
     Ref<T>  defaultVer();               // 디폴트 버전 탐색
     Ref<T>  genericObj();               // 생성객체의 Ref 리턴

    // VDH 탐색 API
    Ref<T>  parentVer();                 // 부모 버전 탐색
    Ref<T>  childVer();                  // 자식 버전 탐색
    Ref<T>  prevSibVer();                // 손위 형제 버전 탐색
    Ref<T>  nextSibVer();                // 손아래 형제 버전 탐색
    Ref<T>  oldestVer();                 // 가장 오래된 버전 탐색
    Ref<T>  latestVer();                 // 가장 최근 버전 탐색
    Ref<T>  prevVer();                   // 시간적 이전 버전 탐색
    Ref<T>  nextVer();                   // 시간적 이후 버전 탐색

    // 버전 상태 API
    void    freeze();                    // FROZEN 상태로 변경
    void    unfreeze();                  // WORKING 상태로 변경
    version_flag  ver_status();          // 버전 상태 정보
          .

private:
    int             ver_num;             // 버전 번호
    Ref<GenericVersion<T> > genver; // 생성객체에 대한 포인터
};
```

그림 2 **Version** 클래스및 객체 버전 API)

4.2 버전 기능 사용 예

그림 3은 앞에서 설명한 SOP 객체 버전 시스템에서 제공하는 API를 통해서 SOP의 C++ OML 환경에서 버전 기능을 사용하는 예를 보이고 있다. 그림 3에서 버전화 가능 클래스 **Emp**는 **Version** 클래스의 하위 클래스로 선언되었는데, 주의해서 볼 점은 **Version** 클래스가 템플릿 클래스로 정의되어 있으므로 **Emp**를 선언할 때, 상위클래스로 **Version** 클래스의 인자로 **Emp**를 지정해야 한다. 클래스 **Dept**는, SOP에서 제공하는 지속성 루트 클래스 **PObject**의 하위클래스로써, 버전화 가능 클래스 **Emp**의 객체를 가리키는 **mgr**라는 멤버를 갖고 있다. 물론, 그림에서 보여지듯이 **Ref<Emp>** 타입으로 선언된 C++ OML의 변수들로 **Emp** 클래스의 객체를 가리킬 수 있다. 그리고, **defaultVer()** 및 **parentVer()** 인터페이스를 이용해서 버전 객체 **emp1**의 VDH 탐색 예를 보이고 있다. 마지막으로, **genericObj()** 인터페이스를 통해서 생성객체에 대한 **Ref**를 획득해서 버전 객체 **emp1**에 대한 동적 바인딩을 사용하는 예도 보이고 있다.

```
OBase obase;                                  // Soprano의 Object Base

class Emp  public Version<Emp> {              // 버전화 가능 클래스 Emp
public
   char name[20];
   Emp(char* Name)  Version<Emp>() { strcpy(name,Name); }
   void setname(char* Name),
          .
};

class Dept' public PObject {                  // 일반 클래스 Dept
public'
   Ref<Emp> mgr,                              // 버전 객체에 대한 ref
          .
};

void
main()
{
  // Soprano Start-Up!!

  Ref<Emp>  emp1 = new(obase) Emp("root");  // Emp 버전 객체 생성
  Ref<Dept> dept = new(obase) Dept(emp1);   // 일반 Dept 객체 생성

  // 버전 객체의 버전 'version2', , 'version9' 유도
  Ref<Emp> emp2 = emp1->derivever();        // 루트 버전에서 'version2'
                                               유도

  emp2->setname("version2"),
          .
```

5) SOP 객체 버전 모델에서 버전 번호를 지원하지 않는다. 따라서 이 버전 번호는 구현과 관련한 내용이다.

```
Ref<Emp> emp9 = emp5->derivever(),          // 'version5'에서 'version9' 유도
emp9->setname("version9");
cout << "emp1. " << emp1->name << endl,  // 'version1'의 이름 출력
Ref<Emp> default_emp = emp4->defaultVer(), // 디폴트 버전(단, 정적 바인딩)
cout << "default " << default_emp->name << endl,
Ref<Emp> parent_emp4 = emp4->parentVer(),   // 'version4'의 부모 버전 탐색
cout << "parent_emp4. " << parent_emp4->name << endl,
        :
// 디폴트 버전 내용을 동적 바인딩으로 출력
Ref<Emp> generic_emp = emp5->genericObj(), // 생성객체에 대한 ref 획득
cout << "generic object " << generic_emp->name << endl;
// Soprano Shutdown!
}
```

그림 3 SOP에서의 버전 기능 사용: 예제 C++ 바인딩 프로그램

5. 구현 내용

이 장은 실제 SOP 객체 버전 시스템의 구현과 관련하여, 클래스 라이브러리로 제공되는 두개의 클래스 **Version**과 **GenericVersion**를 설명한다. 이장에서는 효과적인 설명을 위해, 3장에서 설명한 SOP 객체 버전 모델의 개념들이 **Version**과 **GenericVersion** 클래스의 멤버함수들의 상호 작용을 통해 어떻게 지원되는가를 중심으로 기술한다. 그리고, 버전 기능의 구현을 위해 ODMG OML C++ 인터페이스를 어떻게 이용하고, 또한 Soprano 객체 저장 시스템에서 영향을 받은 부분도 설명한다 특히 이장에서는 독자들이 ODMG C++ OML 바인딩[6]과 C++ 객체지향 언어[20]에 익숙하다고 가정한다.

5.1 SOP 객체 버전 시스템의 아키텍쳐

그림 4는 SOP 객체 버전 시스템의 동작 원리를 보여주고 있다. 그림에서 사각형의 점선으로 둘러싸인 부분이 하나의 버전 객체를 나타낸다. 그리고 점선모양의 타원은 SOP 객체 버전 모델의 생성객체 역할을 하는데, 두개의 객체로 이루어져 있다. 사각형의 **dummy** 생성객체는 ODMG C++ OML 환경에서 다른 객체가 **Ref<T>**를 통해 해당 버전객체에 대한 동적 바인딩을 가능하게 하기 위한 일종의 구현상의 편법으로 도입되었다. 실질적인 생성객체의 역할, 즉 디폴트 버전이나 VDH 정보의 관리 등은 **GenericVersion**의 인스턴스 객체(이를 해당 버전 객체의 **real** 생성객체라 한다.)에 의해 수행된다. 실제로 이 **dummy** 생성객체에 동적으로 바인딩하고 있는 다른 객체에서 버전 API를 통해 접근할 경우(예를 들어, 그림 4의 **Ref<T> B**), 이 **dummy** 생성객체는 자신에게 오는 요구 사항, 즉 버전 API를 **real** 생성객체인 타입 **GenericVersion<T>**로 돌린다. 동적 바인딩 지원을 위한 **real** 생성객체와 **dummy** 생성객체의 상호 작용에 관해서는 뒤에서 구체적으로 설명하겠다. **dummy** 생성객체및 **real** 생성객체는 해당 버전 객체의 생성시에 루트 버전과 함께 생성된다. 정적 바인딩은 그림의 **Ref<T> A**처럼 특정 버전을 직접 가리킨다. 그런데, 이들 버전들 - **dummy** 생성객체도 포함해서 - 은 **real** 생성객체에 대한 포인터를 유지하고 있다(그림 4 참조). 이는 버전 API의 세만틱의 구현을 위해서는 **real** 생성객체에서 유지하는 해당 버전객체의 관련정보가 필요하기 때문이다. 예를 들어, 특정 버전의 부모버전을 탐색하는 API에 대해 해당 부모버전에 대한 **Ref**를 리턴하기 위해서는 **real** 생성객체에서 유지하는 VDH 정보가 필요하다.

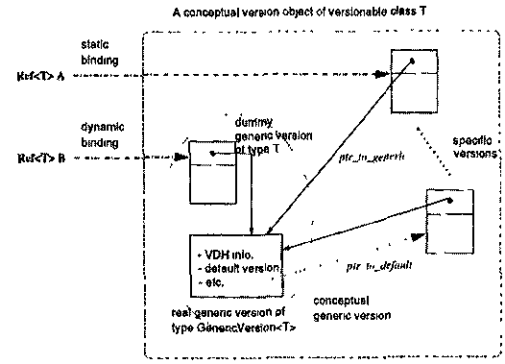

그림 4 SOP 객체 버전 시스템의 동작 원리

5.2 GenericVersion<T>

그림 4는 앞에서 설명한 **real** 생성객체의 클래스의 정의의 일부분을 보이고 있다. SOP 객체 버전 시스템에서는 사용자가 정의한 버전화 가능 클래스 **T**에 대해서 **GenericVersion<T>**가 자동적으로 생성된다. 그림에서 알수 있듯이 **GenericVersion**의 멤버로는 해당 버전 객체의 **dummy** 생성객체나 디폴트 버전에 대한 **Ref**를 갖고 있고, VDH 정보도 포함하고 있다. 그림 5에는 자세히 나와 있지는 않지만, **GenericVersion** 클래스는 실질적으로 버전 기능의 구현을 위해 많은 멤버함수를 갖고 있다. 다음 절들에서 설명이 필요한 멤버함수들은 구체적인 코드를 설명하겠다.

```
// 헤더화일 generic.h
typedef unsigned VNUM;  // VNUM 타입

// VDH(version derivation hierarchy)의 유도관계를 위한 자료구조 struct VDHEdge {
  VNUM parent;          // 부모 버전 번호
  VNUM child;           // 자식 버전 번호
};

template<class T>
class GenericVersion  public PObject {
                        // 템플릿 클래스 GenericVersion
  Ref<T> _dummy;        // dummy 생성객체
  Ref<T> _default;      // 디폴트 버전

  Ref<T> versions[20];  // 해당 버전 객체내의 현재 버전들

  // VDH 및 VTH 정보
  VDHEdge VDH[20],      // 각 버전들사이의 유도관계
  int edgenums,         // 유도관계의 수
  VNUM    VTH[20],      // VTH 관계
  int vernums;          // 버전 수
  VNUM    nextvernum;   // 다음 버전 번호

  // 기타 버전 기능의 구현을 위해 클래스 T와의
  // 인터페이스를 위한 멤버 함수들
      .
}
```

그림 5 GenericVersion<T>의 구조

5.3 버전 상태의 구현

5.3.1 버전 상태 정보의 인코딩및 버전 상태의 변경

앞에서 기술한 바와 같이 SOP 객체 버전 시스템에서는 **WORKING**과 **FROZEN**의 두가지의 버전 상태(구현의 측면에서는 **dummy** 생성객체의 구분을 위한 **dummy** 상태도 포함)를 지원하는데, 이들 상태정보는 각 특정 버전에 인코딩(encoding)된다. Soprano 객체 저장 시스템의 각 지속성 객체는 사용자에게는 보이지 않지만, 시스템에서 필요로 하는 정보를 다음의 **PHeader**라는 자료구조에 포함하고 있다[6]. 이 자료구조 내의 **utags** 항목은 사용자로 하여금 특정 기능의 구현을 위해 남겨두었는데(reserved), SOP 객체 버전 시스템은 이 **utags** 항목을 이용해서 각 버전의 상태정보를 인코딩한다.

```
struct PHeader {
      :
   ushort   stags;    // Soprano에서 내부적으로 사용하는 flag 정보
   ushort   utags;    // Soprano에서 제공하는, 사용자 이용가능 flag
};
```

앞에서 4장에서 설명한 바와 같이, 특정 버전의 상태 변경은 클래스 **Version**의 두 멤버함수 **freeze(), unfreeze()**를 이용해서 사용자가 명시적으로 관리해야 한다.

5.3.2 FROZEN 버전의 변경 금지

ODMG C++ OML에서는 객체 값의 변경을 위해, **PObject::markmodified()**를 제공한다. 즉, ODMG의 C++과 같은 프로그래밍 언어의 환경에서 발생하는 객체의 삭제나 값의 변경을 객체지향 데이타베이스 시스템에서 실행시에 자동적으로 체크할 수 있는 방법이 없기 때문에, 이 인터페이스를 통해서 사용자가 명시적으로 시스템에 알려주어야 한다. 시스템은 해당 트랜잭션의 종료(commit)시에 이 정보를 이용해서 객체 값의 변경을 디스크에 반영한다. 이점에 착안하여 SOP 객체 버전 시스템에서는 **FROZEN** 상태의 버전의 값의 변경을 막기위해 **PObject::markmodified()** 멤버함수의 처음에서 객체의 상태정보를 검사해서 **FROZEN** 상태의 버전인 경우 데이타베이스에 반영을 막는다

```
int PObject::markmodified() {
    if (utags == FROZEN)
        return error;
    // original markmodified() code,
        :
}
```

5.4 버전 객체의 생성및 동적 바인딩의 구현

버전 클래스 T의 버전 객체를 ODMG C++ 바인딩의 **new()** 인터페이스를 통해 생성하는 경우, 클래스 **Version**과 **GenericVersion**의 생성자들이 버전 객체의 생성과 관련한 일련의 작업을 다음과 같이 수행한다.

```
template<class T> inline
Version<T>::Version(void) {
      // 이 생성자 호출에 의해 생기는 버전이 dummy 생성객체로 표시 markflag(DUMMY);

      // 이 dummy 생성객체의 Ref를 인지로 GenericVersion 생성자 호출
      genver = new(obase)
                GenericVersion<T>(Ref<T>((T*)this)),
}

template<class T> inline
GenericVersion<T>::GenericVersion(Ref<T>& dummy) {
```

6) 즉, 실제 데이타베이스에 저장되는 모든 지속성 객체 레이아웃(layout)의 앞부분에 이 정보가 포함되어 있다.

```
        _dummy = dummy;     // dummy 생성객체 값 세팅
        _default.clear();   // 디폴트 객체에 대한 포인터를
          clear
        // 기타 생성객체의 관리 정보 초기화
        edgenums = 0;
        nextvernum = 1;
          :
}
```

그런데, 여기서 주의해서 볼 두가지 사항이 있다. 우선, 버전 객체에 대한 **new()** 연산의 리턴 값은 **dummy** 생성객체에 대한 **Ref**가 리턴된다. 예를 들어, 4장의 그림 3의 변수 **emp1**은 생성된 버전 객체를 동적으로 바인딩하고 있다. 둘째, **new()** 연산의 수행이 완료된 시점에서는 **dummy** 생성객체와 **real** 생성객체뿐이고, 실질적인 루트 버전은 아직 생겨나지 않았다.(이는 구현상의 한계이다.) 이 문제에 대한 우리의 해결책은 다음과 같다. 버전 객체의 생성이후, 처음으로 **new()**에서 리턴된 **dummy** 생성객체를 동적으로 바인딩하고 있는 **Ref**를 통해서 접근할 경우, Soprano 객체 저장 시스템의 템플릿 클래스 **Ref<T>**의 연산자 **operator ->**를 그림 6과 같이 수정함으로써 해결하였다.

```
template<class T> inline
T* Ref<T>::operator ->(void) const
{
    // Soprano의 정상적인 객체 fixing및 포인터 스위즐링 과정
                .
    if ( ((PHeader*)memptr)[-1].utags == DUMMY ) {
         // dummy 생성객체에 대한 traversal인 경우(동적
            바인딩)
        // 디폴트 버전에 대한 포인터 리턴
        return (T*)(((PObject*)memptr)->defaultver());   // (1)
    }
    else // 정적 바인딩인 경우 해당 버전에 대한 메모리 포인
    터 리턴
         return (T*)memptr;
}
```

그림 6 동적 바인딩을 위한 Ref<T>::operator->의 확장

그림 6에서 보여지듯이, 특정 **Ref**가 가리키는 객체를 연산자 **operator ->**를 통해 접근할 경우, 만일 해당 객체가 **dummy** 생성객체인 경우 **real** 생성객체에 디폴트 버전의 객체 식별자를 얻기위해 자신의 **defaultVer()**멤버함수를 호출한다. 그런데, 그림에서 (1)의 문장이 컴파일되기 위해서는 클래스 **PObject**에 다음 멤버함수가 있어야만 한다.

```
virtual PObject* defaultver(void) { return this; }
```

실제로 이 멤버함수는 일반 객체에 대해서는 무의미하고 단지 컴파일만을 위한 것이다. 클래스 **Version**에서 동적인 바인딩을 위해 다음과 같이 재정의된다[7].

```
template<class T>
T* Version<T>::defaultver(void) {
  T* defaultver;

  if (usertags() == DUMMY) {  // dummy 생성객체
     markflag(WORKING);   // 임시로 WORKING으로 세팅
     defaultver = genver->defaultver(); // real 생성객체로
                                          forward!!!
     markflag(DUMMY);              // DUMMY로 복원
  }
  else {                        // 일반 버전인 경우
     defaultver = genver->defaultver(); // real 생성객체로
                                          forward
  }
   return defaultver;
}
```

다음은 **real** 생성객체 클래스 **GenericVersion**의 멤버함수 **defaultver()**를 보여주고 있는데, 만일에 버전 객체가 생성된 뒤 처음으로 불려질 때는 아직 디폴트 버전(동시에 루트버전)이 생성되지 않은 상태이기 때문에 자신의 멤버함수 **genroot()**를 통해서 디폴트 버전을 생성한다. 이때 디폴트 버전은 **dummy** 생성객체의 값을 복사하게 된다. 이때서야 실질적으로 **dummy**및 **real** 생성객체와 하나의 루트 버전(동시에 디폴트 버전임)이 존재하게 된다.

```
template<class T>
T* GenericVersion<T>::defaultver(void) {

    // 디폴트 버전이 아직 세팅되지 않은 경우,(즉,....)
    if (_default.isnull()) {
        // genroot를 통해서 루트 객체 생성
        // 이때, genroot에서는 루트 버전을 디폴트로 세팅
        genroot(_dummy.operator->()),
    }
    return _default.operator->(); // 디폴트 버전에 대한 포인
                                     터 리턴
}
```

마지막으로 이 절에서 주목해야 할 것은 2장에서 언급한 일반 지속성 객체에 관한 성능의 최소화와 관련된 내용이다. ODMG 표준에 있어서 일반 지속성 객체에 대한 접근은 **Ref**에 대한 연산자 **operator->**를 통해서

7) 그림 2의 API defaultVer()과는 다르다. 즉, defaultVer()는 사용자에게 제공되는 public 멤버함수이고, defaultVer()는 동적 바인딩의 구현을 위한 SOP 객체 버전 시스템에서 내부적으로 사용하는 private 멤버함수이다.

이루어지는데, SOP 객체 버전 시스템에서는, 버전 객체와 일반 지속성 객체 모두에 대해, 그림 6에서 보여지듯이 해당 객체가 **dummy** 생성객체인지를 검사하는 부분이 추가된다. 이 문제를 제외하고 SOP 객체 버전 시스템에서 버전 기능의 추가로 인한 일반 지속성 객체에 대한 성능상의 오버헤드는 전혀 없다. 그런데, 일반 지속성 객체에 대한 성능에 전혀 영향을 주지 않기 위해 **Ref** 대신에 버전 객체에 대한 또다른 스마트 포인트(예를 들어, **VRef**)를 제공해서 버전 객체에 대한 참조는 이 타입의 포인트를 통해서 접근하게 할 수 있다. 그러나, 이는 사용자로 하여금 버전 기능의 추가로 인해 일반 객체와 버전 객체를 명시적으로 구분해야 하는 부담을 주는 단점이 있다. 따라서, SOP 객체 버전 시스템에서는 Ref의 연산자 **operator->**를 확장하는 방법을 선택했다.

5.5 새로운 버전의 유도

그림 7은 하나의 버전에서 새로운 버전의 유도할 때, 사용되는 **Version** 클래스의 멤버함수 **derivever()**을 보여주고 있다. 우선 이 함수를 호출받는 객체가 **dummy** 생성객체인 경우 요구사항을 **real** 생성객체에 보내서 디폴트 버전에서 새로운 버전을 유도하게 한다. 만일 정상적인 버전 객체의 경우, 새로운 버전을 생성하고 **real** 생성객체에 VDH 관련 정보를 갱신한 후 새로운 버전의 **Ref** 값을 리턴한다. 이때, 새로운 버전의 내용은 부모 버전과 동일하다.

```
template<class T>
Ref<T> Version<T>::derivever(void) {
    Ref<T> newver,

    // dummy 생성객체일 경우 real 생성객체로 redirection
    if (usertags() == DUMMY) {
        newver = genver->derivever();
        return newver;
    }

    // 버전 클래스의 복사 생성자를 이용해서 새로운 버전을 생성
    newver = new(obase) T(*(T*)this),

    // 새 버전의 ver_num 값 세팅
    VNUM newvernum = genver->nextvernum;
    newver->ver_num = newvernum,

    // 생성객체의 VDH 관련 정보 등을 갱신
    genver->markmodified(),
    genver->nextvernum++;
    genver->add(newver),          // add Ref to new version
    return newver;                // 새로운 버전에 대한 Ref 리턴
}
```

그림 7 버전 유도 멤버 함수 derivever()

5.6 VDH 탐색

SOP 객체 버전 시스템은, 앞에서 언급한 바와 같이, 특정 버전이나 디폴트 버전에 대한 VDH 탐색을 위한 요구사항은, **real** 생성객체에서 관리하고 있는 VDH 정보를 이용해서 수행하게 된다. **dummy** 생성객체에 대한 VDH 탐색의 경우, 앞에서 설명한 다른 기능의 구현과 마찬가지로, **real** 생성객체가 디폴트 버전에 대한 탐색 결과를 리턴한다. 정상적인 버전의 경우에는, 자신의 버전 번호를 인자로 **real** 생성객체의 대응 멤버함수를 호출해서 해당 결과 버전의 **Ref** 값을 리턴한다.

5.7 버전의 삭제

SOP 객체 버전 시스템에서는 버전의 삭제를 위한 인터페이스를 따로 제공하지 않고 ODMG C++ OML에서 제공하는 **Ref<T>::delete_object**[8]를 통해서 버전의 삭제를 지원한다. 실제로 이 **delete_object** 인터페이스는 궁극적으로 해당 객체의 클래스에 정의된 소멸자를 호출하게 되는데, 이를 이용해서 다음과 같이 클래스 **Version**의 소멸자(destructor)에 버전 삭제의 세만틱을 구현했다.

```
template<class T>
Version<T>::~Version(void) {
    if (usertags() == DUMMY) {
        // 디폴트 버전인 경우, 해당 버전 객체를 삭제
        // 즉, dummy/real 생성객체와 모든 버전)
        genver->deleteAllVers();
        genver.destroyobj();
    }
    else {
        // 특정 버전인 경우 자신의 버전 번호를 인자로
        // real 생성객체에서 자신의 정보를 삭제
        genver->deleteVer(ver_num);

        // 해당 버전 객체내에서 마지막 버전인 경우,
        // dummy및 real 생성객체도 함께 삭제
        if (genver->vernums == 0)
            genver.destroyobj();
    }
}
```

8) SOP 객체지향 DBMS에서는 delete_object() 대신에 destroyobj()라는 API를 제공한다.

6. 관련 연구

본 장에서는 SOP 객체 버전 시스템을 각각 버전 모델 측면과 구현의 측면에서 다른 시스템과 비교한다.

6.1 버전 모델 측면에서 타 시스템과의 비교

다음 표 1은 각 객체지향 DBMS들의 지원하는 VDH의 구조와 동적 바인딩의 지원여부를 정리한 것이다. 표에서 세번째 칼럼은 특정클래스의 타입에 무관하게 버전 기능의 타입에 대한 직교성(orthogonality)의 여부를 나타낸다. 표에서 DAG 구조의 VDH를 지원하는 객체지향 데이타베이스 시스템들에 있어서 버전의 합병은 사용자의 책임이다.

서로 다른 버전 객체에 속하는 버전들사이의 조합관계를 컨피규레이션(configuration)이라 하는데, CAD/CAM 등과 같은 응용 분야에서는 일관성있는 컨피규레이션의 관리가 매우 중요하다[7]. 대부분의 객체지향 데이타베이스 시스템에서는 버전기능과 동적 바인딩 등의 메카니즘을 통해 사용자가 직접 컨피규레이션을 관리하도록 하고 있다. 반면, O2와 ObjectStore는 각각 Version Unit, Configuration이라는 개념을 통해 시스템에서 기본적인 컨피규레이션 관리기능을 직접 지원한다.

표 1 객체지향 데이타베이스 시스템들의 버전 기능 비교

	VDH	동적바인딩	타입 직교성
SOP 객체 버전 시스템	Tree	o	x
EPVM(EXODUS)([8])	?	x	?
ITASCA([9])	Tree	o	x
O2([14])	DAG	o	o
Objectivity/DB([16])	Tree	o	x
ObjectStore([15])	DAG	o	o
Ode([2])	Tree	o	x
Versant([21])	DAG	o	o

각 응용분야별로 상이한 세만틱과 요구사항을 가진 버전 개념을 현재의 객체지향 데이타베이스 시스템에서 제공하는 기능만으로는 모두 만족시키기 힘들다. 이를 위해서는 좀 더 일반적인(generic) 접근을 통해 버전 기능의 지원이 필요하다. 이와 관련하여, [18]서는 Exodus/Extra 데이타모델에 기반한 버전 모델을 제시하고 있고, Itasca나 Objectivity/DB 등은 사용자로 하여금 자신의 응용 프로그램에서 필요한 버전 세만틱을 재단(customization)할 수 있게 한다.

6.2 구현 측면에서 타 시스템과의 비교

SOP 객체 버전 시스템은 ODMG C++ OML을 확장해서 버전 기능을 제공하는데, 현재 본 저자들이 아는 범위에서 ODMG C++ OML 환경에서 버전 기능을 지원하는 상용 객체지향 데이타베이스 시스템은 없다. O2나 Objectivity 등은 해당 시스템에서 제공하는 특별한 언어 환경(예를 들어, O2의 경우 O2C)에서 버전 기능을 지원하고 있다. 그리고, 각 시스템들의 버전 기능의 구현 메카니즘을 기술하고 있는 참고문헌도 없다. 따라서, 이 절에서는 버전의 구현 내용을 참고문헌에서 언급하고 있는 몇몇 객체지향 데이타베이스 시스템들과 SOP 객체 버전 시스템을 비교한다.

Orion[11]과 같이 Lisp의 인터프리터 환경에서 버전 기능을 제공하는 경우 버전 기능의 구현이 상대적으로 용이하다. 특히, 버전 기능의 구현에 있어 핵심적인 동적 바인딩의 구현에 있어 하나의 객체가 디폴트 버전을 생성객체를 통해 참조하는 경우 타입 문제가 발생하지 않는다. 반면에, SOP 객체 버전 시스템의 경우 **Ref**를 통한 동적 바인딩의 구현과 관련해서 타입 문제를 해결하기 위해 그림 4에서처럼 **dummy** 생성객체를 도입해야 한다.

Ode[2]의 경우, 데이타베이스 프로그래밍 언어 O++ 환경에서 버전 인터페이스를 사용자에게 제공하고, 이 인터페이스를 이용해서 작성된 프로그램은 O++ 컴파일러를 통해서 전처리(pre-processing) 과정을 거쳐서 C++ 환경에서 동작하게 된다. 반면에, SOP 객체 버전 시스템은 이와 같은 전처리 과정을 거치지 않고 직접 C++ 환경에서 버전 기능을 제공하고 있다 특히 Ode의 경우 버전 기능의 구현과 관련하여 전처리 과정을 필요로 했던 이유중의 하나는 Ode의 구현 당시 C++의 템플릿 기능을 지원하는 컴파일러 기술이 없었기 때문이다. 그래서, Ode에서는 버전 클래스와 그에 해당하는 버전 생성 클래스 사이의 타입문제를 해결하기 위해 매크로 기능을 이용했다. SOP 객체 버전 시스템에서는 C++의 템플릿 기능을 이용해서 클래스 **Version<T>**, **GenericVersion<T>**를 정의함으로써 이 문제를 해결했다. 마지막으로, [2]에는 Ode가 동적 바인딩을 지원한다고 기술되어 있지만, 이의 사용에 관한 방법이나 정확한 동작 원리에 대한 설명이 기술되어 있지 않다. 반면에 SOP 객체 버전 시스템에서는, 비록 버전 객체마다 하나의 **dummy** 생성 객체를 둠으로써 저장공간의 문제는 있지만, 동적 바인딩의 개념을 완전히 지원하고 있다.

7. 결 론

본 논문에서는 ODMG-93을 지원하는 SOP 객체지

향 데이타베이스 시스템의 버전 기능의 설계와 구현 내용을 다루었다. ODMG-93 표준과의 호환, Soprano 객체 저장 시스템과의 인터페이스 최소화 등의 설계 목표를 설명하고, SOP 객체 버전 시스템의 버전 모델을 설명하였다. 그리고, ODMG C++ OML 환경에서 사용자에게 제공되는 버전 관련 API와 사용 방법을 예를 통해 보였다. 그리고, 설계 목표를 만족시키기 위해 어떤 방법으로 이들 API를 구현했는가를 설명했다. 특히 SOP 객체 버전 시스템은 ODMG 표준을 확장해서 버전 기능을 제공하는 최초의 시스템이고, C++과 같은 타입 검사가 엄격한 구현 환경에서 동적 바인딩 등의 핵심 버전 기능을 구현을 위한 해결책을 제시했다는 점에 그 의의가 있다.

향후에는 6장에서 지적한 컨피규레이션및 사용자의 버전 세만틱 재단을 지원할 수 있도록 현재의 SOP 객체 버전 시스템을 확장하는 연구를 수행할 예정이다. 또한, 현재의 SOP 객체 버전 모델을 본 저자들이 [13]에서 제안한 스키마 버전 모델과의 통합을 위한 연구도 진행할 것이다.

감사의 글

본 연구에 도움을 준 객체지향 연구실 SOP 팀원들에게 감사를 드립니다.

참 고 문 헌

[1] 안 정호, 이 강우, 송 하주, 김 형주. "Soprano: 객체 저장 시스템의 설계 및 구현". 정보과학회 논문지(C)", 1996.

[2] R. Agrawal, S. J. Buroff, N. H. Gehani, D. Shasha. "Object Versioning in Ode". Proceedings of International Conference on Data Engineering, 1991.

[3] A. Bapat, J. Waesch, K. Aberer, J. M. Haake. "HyperStorM: An Extensible Object-Oriented Hypremedia Engine". The Seventh ACM Conference on Hypertext, 1996.

[4] Philip A. Bernstein. "Repositories and Object Oriented Databases". Proceedings of BTW '97, 1997.

[5] Carl Cargill. "World Wide Web: Authoring, Versioning, and Other Topics". ACM Perspectives on Standardization, 5(1), 1997.

[6] G.G. Cattell. "The Object Database Standard: ODMG-93(Relase 1.1)". Morgan Kaufmann, 1994.

[7] R. G. G. Cattell. "Object Data Management: Object-Oriented and Extended Relational Database Systems". Addison-Wesley Publishing Company Inc., 1991.

[8] EXODUS Project Group. "EXODUS Storage Manager V3.0 Architectural Overview". University of Wisconsin-Madison, 1993.

[9] IBEX Object Systems, Inc.. "ITASCA Technical Summary Release 2.3", 1995.

[10] R. H. Katz. "Towards a Unified Framework for Version Modelling in Engineering Databases". ACM Computing Survey, 22(4), 1990.

[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] Sang-Won Lee, Hyoung-Joo Kim. "A Model of Schema Versions for Object-Oriented Databases, based on the concept of Rich Base Schema". Information and Software Technology(accepted for journal publication), 1998.

[14] O2 Technology. "A Technical Overview of The O2 System". 1994.

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

[16] Objectivity, Inc.. "Objectivity/DB Technical Overview Release 3.0". 1995.

[17] Rajeev Rastogi, S. Seshadri, Philip Bohannon, Dennis W. Leinbaugh, Abraham Silberschatz, S. Sudarshan. "Logical and Physical Versioning in Main Memory Databases". Proceeding of International Conference on Very Large Data Bases, 1997.

[18] Edward Sciore. "Versioning and Configuration Management in an Object-Oriented Data Model". The VLDB Journal , 3(1), 1994.

[19] Avi Silberschartz, Mike Stonebraker, Jeff Ullman. "Database Research: Achivements and Opportunities Into the 21st Century". Report of an NSF Workshop on the Future of Database Systesms Research, 1995.

[20] Bjarne Stroustrup. "The C++ Programming Language(2nd edition)". Addision Wesley, 1991.

[21] Versant Object Technology Copr.. "Versant OODBMS Release 4". 1996.

이 상 원

1991년 서울대학교 컴퓨터공학과 학사 졸업. 1994년 서울대학교 컴퓨터공학과 석사 졸업. 1994년 ~ 현재 서울대학교 컴퓨터공학과 박사 과정 재학. 관심분야는 데이타베이스, 데이타웨어하우스, 컴퓨터 보안.

김 형 주

제 25 권 제 2 호(B) 참조