OODBMS를 위한 효율적인 메소드 지원방안
(Efficient Method Support in OODBMS)

이 병 준 † 박 주 홍 †† 김 형 주 †††
(Byung-Joon Lee) (Juhong Park) (Hyoung-Joo Kim)

요 약 객체지향 데이타베이스 상에서의 질의어는 기존 관계형 DBMS와는 달리 객체에 대해서 정의된 메소드에 대한 호출을 포함할 수 있다. 이러한 메소드가 C++ 같은 일반적인 컴파일러 기반 언어에 의해서 만들어졌을 경우, 대화형 환경에서 메소드를 포함한 질의어를 지원하기 위해서는 여러 가지 문제를 해결해야만 한다. 어떻게 하면 사용자로 하여금 인자의 개수나 자료형에 신경 쓰지 않고 메소드를 정의할 수 있도록 하는가 하는 것도 그러한 문제중 하나이다. 본 논문에서는 메소드 지원 시스템 구현과 관련하여 제기되는 이러한 문제점들을 짚어보고, 사용자의 제약을 최소화하는 동시에 시스템의 이식성을 높이는 해결방안으로서 전처리 방식을 제안한다

Abstract In contrast to traditional RDBMS, queries for OODBMS can contain method calls which are defined in objects. If these methods are written by compiler-based general programming languages such as C++, we must cope with various problems to support queries including methods in interactive OQL environment. One of main issues is how to give users freedom in defining methods without limitations on the number and types of arguments. In this paper, we suggest preprocessing method as a proper solution for minimizing limitations of users and enhancing the portability and extensibility of OODBMS system.

1. 서 론

CAD/CAM등의 디자인 응용(design application)의 요구가 대두됨에 따라, 긴 트랜잭션 모델(long transaction model)이나 버전(version) 관리, 복잡 개체(complex object)의 모델링 능력 등을 갖춘 객체지향 데이타베이스 관리 시스템(Object-Oriented Database Management System: OODBMS)이 각광받기 시작하였다. 객체지향 데이타베이스 관리 시스템은 캡슐화(encapsulation), 계승(inheritance), 다형성(polymorphism) 등 객체지향 모델을 채택하고 있는 시스템이었다. 이러한 객체지향 데이타베이스 관리 시스템은 그 모델의 특성상 디자인 응용에서 필요로 하는 여러 가지 요구를 만족시키기 용이하였다. 그것은 객체지향 개념 자체가 가지는 일관된 모델링 능력과, 관계형 데이타베이스 관리 시스템(Relational Database Management System: RDBMS)에 이미 구현된 다양한 기술들이 맞물려 발전한 결과였다[2] 그러나 객체지향 데이타베이스 관리 시스템은 이전에는 찾아볼 수 없었던 다양한 이론적, 기술적 문제를 데이타베이스 관리 시스템의 개발자와, 시스템 사용자에게 안겨주었다. 질의문 처리(query processing)의 영역에서도 다양한 문제들이 제기되었으며[7, 6, 4] 대략 다음과 같이 요약해 볼 수 있다.

첫째로, 관계형 데이타베이스 관리 시스템을 위한 자료의 모델(model)과 객체지향 데이타베이스 시스템을 위한 자료의 모델이 크게 다르므로, 관계 해석(relational calculus)이나 관계 대수(relational algebra) 같은 이론적 성과를 객체지향 데이타베이스 관리 시스템에 그대로 적용할 수 없다는 문제가 있었다.

· 본 연구는 과학기술처의 국가 지리 정보 시스템 기술개발사업과 정보통신 연구관리단의 미래로/DB 통합 소프트웨어 시스템을 위한 SRP 확장 프로젝트로부터 지원받았습니다
† 학생회원 : 서울대학교 컴퓨터공학과
bjlee@oopsla.snu.ac.kr
†† 비 회 원 : 서울대학교 컴퓨터공학과
jhpark@oopsla.snu.ac.kr
††† 종신회원 서울대학교 컴퓨터공학과 교수
hjk@oopsla.snu.ac.kr
논문접수 : 1997년 12월 11일
심사완료 : 1998년 4월 10일

두번째로, 질의문 처리라는 큰 틀 내에 메소드(method) 호출, 패스 표현식(path expression)같은 객체지향적 특징을 포함시키기 위해서는 이전의 관계형 데이타베이스 관리 시스템에서 찾아볼 수 없었던 다양한 문제들을 해결해야만 한다는 문제가 있었다. 특히 메소드는 질의문 처리에 있어서, 최적화 측면에서나 구현의 측면에서나 다양한 문제를 야기하게 된다. 메소드는 객체(object)에 던져지는 메시지(message)를 처리하기 위해, 객체 내에 실제 자료와 함께 캡슐화된 함수(function)를 일컫는 말이다. 객체지향 데이타베이스는 메소드에 대한 호출을 질의문 내에 포함시키는 것을 허용하고 있다. 예를 들어, 다음과 같은 질의문이 있다고 가정하자.

```
select a.name
from a in Employee
where a.employer.retire(1995) == 20
```

위의 질의문은 1995년 기준으로 퇴직 연한이 이십년 후로 예정된 상사를 둔 고용인의 이름을 얻는 질의문이다. 이 질의문은 retire(1995)라는 메소드 호출을 포함하고 있으며, 아울러 이 메소드 호출 결과는 질의문에서 원하는 객체를 검출하기 위한 술어문(predicate)으로 사용되고 있다.

질의어 최적화 관점에서 보면 위의 질의문은 메소드의 수행 비용을 알 수 없기 때문에 최적화 하기 힘들다. 또한, 위와 같은 질의문을 대화형 객체 질의 언어(interactive Object Query Language: Interactive OQL)와 같은 대화형 환경(interactive environment)에서 실행하려고 하거나 함수 호출 형태로 질의문을 처리하는 내포형 OQL(Embedded OQL)을 사용하는 경우, 질의문 내에 포함된 메소드가 인자(argument)를 몇 개를 가질지 또 그 인자의 형(type)은 무엇인지 시스템이 예측할 수 없으므로, 메소드 호출을 처리하는 부분을 질의 처리기(query processor) 내에 반드시 포함시켜야 하며, 이 과정에서 구현에 관련된 여러 가지 어려움에 직면하게 된다.

본 논문에서는 위에서 두번째로 제시한, 대화형 객체 질의 언어(Interactive OQL) 환경이나 특정 형태의 내포형 OQL 환경에서 메소드를 포함한 질의어를 수행하고자 할 때 발생하는 구현상의 난점을 해소하는 것을 목적으로 하고 있다. 본 논문에서 제시하는 방안은 두가지 환경 전부에 동일하게 적용될 수 있으므로, 본문 중에서는 대화형 OQL환경만을 다루도록 하겠다. 2장에서는 이러한 문제에 대한 기존의 접근법을 살펴본다. 아울러 그러한 접근방식의 비효율성과 제약점을 짚어보고, 개선된 접근 방식을 제시할 것이다. 3장에서는 본 논문에서 제시하는 개선된 접근 방식을 어떻게 구현하였는지 모듈별로 살펴본다. 4장에서는 현재 발표된 OODBMS 들을 살펴보고, 각각의 시스템들이 어떻게 메소드를 지원하고 있는지 요약해 볼 것이며, 5장에서 결론과 향후 연구 방향을 제시할 것이다.

2. 메소드 지원 시스템 구현상의 문제점들

앞서 밝혔듯이, 질의문을 대화형 환경 내에서 실행하고자 할 때에는 사용자가 제시한 메소드가 인자를 몇 개나 가지는지, 아니면 그 형(type)이 무엇인지 미리 알 수는 없다. 사용자가 retire(1995) 메소드를 수행할 것인지, 아니면 1995년과 1996년 사이에 퇴직하는 사람을 알아내기 위해서 retire(1995, 1996)을 수행할 것인지에 관한 정보를 메소드 컴파일 시에 데이타베이스 관리 시스템에 알릴 수 없다는 것이다.

따라서 질의문에 포함된 메소드 호출을 처리하기 위해 통상적으로는 다음과 같은 코드가 질의 처리기 내에 포함된다.

```
// f is function pointer
f = getfunction( mangle( cppFunctionName ) );
switch ( num_of_argument ) {
    case 1 :
            if ( arg[1].type==INT )
              return (*f)( int(arg[1].val) );
      else if ( arg[1] type==FLOAT )
          ... .
    case 2 :
            if (arg[1].type==INT && arg[2] type == INT)
              return (*f)( int(arg[1].val), int(arg[2].val) );
            else if (arg[1].type == INT && arg[2].type
                                    == FLOAT)
              return      (*f)( int(arg[1].val),
                                  float(arg[2].val) );
                        .. .
    case 3 :
        ......
}
```

위의 코드에서 num_of_argument는 사용자가 입력한 질의문 내에 포함된 메소드가, 인자를 몇 개나 가지고 있는가를 나타내는 변수다. 질의문 파싱 시기에 이 변수에 적절한 값이 대입되며, 그 값에 따라 case문으로 메소드에 적정 개수의 인자를 전달하게 된다 또한 case문 내에서는 if-else 문을 통해서, 인자를 형변환(type

casting)시켜 전달하고 있다. 이런 방식을 택할 경우, num_of_argument의 값이 k이고 인자로 올 수 있는 자료형이 총 t개일 때, case k: 내에 포함되는 if와 else if문의 개수는 t^k개에 달한다.

또한 위의 방식은 값 기반 호출(call-by-value) 방식만을 고려한 코드이므로, 참조 기반 호출(call-by-reference) 방식까지 고려하면 if 문의 개수는 훨씬 더 늘어나게 된다. C++ 에 기반한 시스템에서는 참조기반 호출이 포인터에 기반한 방식과 참조형 변수(reference type variable)에 기반한 방식의 두 가지로 지원되므로, 각각의 자료형에 대해서 세가지 방식의 호출을 모두 지원할 경우 if문의 개수는 $(t\times3)^k$에 달하게 된다.

또한 메소드의 복귀 형(return type)까지 고려할 경우에는, 위의 switch 문을 시스템이 지원하고자 하는 복귀 형의 개수에 맞추어 중복해서 써 주어야 한다. 그러므로 시스템이 m가지의 복귀형을 지원한다면, 코드의 복잡도는 $m\times(t\times3)^k$에 달하게 될 것이다.

그러므로, 위와 같은 단순한 접근 방식으로 많은 자료형을 지원하는 시스템을 설계하는 경우, 메소드 처리 루틴이 엄청나게 복잡해져 구현상의 부담이 기하급수적으로 증가하게 된다.

컴파일러 기반 언어를 사용하는 대부분의 OODBMS 시스템들이 이러한 문제에 직면하게 된다. 그러나 인터프리터 기반 언어를 사용하는 OODBMS 시스템에서는 일반적으로 이러한 문제가 발생하지 않는다.

인터프리터 기반 언어를 이용하여 구현된 OODBMS 중 대표적인 것으로는 Smalltalk로 구현된 GemStone [13]과, Lisp를 이용하여 구현된 Orion[3]을 들 수 있다. 이 시스템들은 객체에 지속성을 부여하기 위해 인터프리터 자체를 확장하거나, 아니면 그 언어로 구현된 라이브러리를 제공하는 접근법을 취한다. 메소드를 정의할 때도 시스템 구현에 사용된 언어만을 사용하도록 제한한 다 따라서 메소드 호출은 별다른 추가적 구현이 없이도 처리되는데, 왜냐하면 인터프리터는 모든 함수 호출을 항상 동적인 환경에서 처리하기 때문이다. 그러므로 따로 메소드를 지원하기 위한 모듈을 정의할 필요가 없으며, 아울러 인터프리터가 OQL 처리기의 기능을 제공하므로, OQL 처리기도 따로 구현할 필요가 없다.

그러나 대부분의 OODBMS는 컴파일러 기반 언어에 의해서 구현되는데, 이는 성능 면에서 컴파일러 언어가 보다 유리하기 때문이다. 컴파일러 기반 언어에 의해 구현된 OODBMS 시스템은 OQL을 처리하기 위해 OQL 처리기를 내장하고 있다 OQL 처리기의 설계는, OQL을 위한 인터프리터를 설계하는 것과 동등하게 생각할 수 있다. 그러나, 인터프리터 기반 언어 인터프리터의 기술을 동일하게 적용 할 수는 없다. 왜냐하면, 데이타베이스 관리 시스템과 메소드가 컴파일러 기반 언어에 의해서 만들어지고 컴파일되었기 때문이다. 따라서 앞서 기술하였던 여러 가지 문제를 피할 수 없게 된다. 그러므로, 대부분의 OODBMS가 인자의 개수나 형에 대해서 제약을 두고 있다.

실제로 ObjectStore같은 시스템은 사용자 정의 메소드의 인자개수를 0, 1, 2개로 제한하고 있으며[17, 18], O2 시스템은 인자의 형에 제약이 있다[24]. 아울러, OODBMS는 아니지만 메소드를 지원하는 ORDBMS의 경우에도 비슷한 제약은 존재하고 있다. Illustra같은 시스템은 사용자가 시스템이 제공하는 데이타 형 이외의 형은 인자로 사용할 수 없도록 하며, 인자를 값 기반(call-by-value)으로 넘겨줄 것인가, 아니면 참조 기반(call-by-reference)으로 넘겨줄 것인가에 관련된 사항을 시스템이 제공하는 인자의 형의 크기(size of type)에 따라서 결정하도록 제약을 두고 있다[21].

인자 개수에 제약이 존재하는 것은 앞서 보인 단순한 해결방식을 사용하는 데서 오는 구현상의 복잡도 문제 때문이라고 볼 수 있다. 반면 인자 형에 제약이 존재하는 경우에는 원인을 두가지 정도로 구분해 볼 수 있다. 첫번째는 앞서 밝혔던 구현의 복잡성이 원인이 되는 경우이며, 두번째는 메소드 지원 시스템의 높은 OODBMS 의존도가 원인이 되는 경우이다. O2 시스템의 경우, 사용자가 정의한 C++ 헤더를 임포트(import)하여 O2 시스템 내부적으로 관리하는 자료형으로 사상(mapping)시키는 과정을 거치는데, 이 과정이 OODBMS 내부의 자료형 정보에 종속적인 것이다. 따라서 OODBMS 내부의 자료형이 확장될 때마다 사상과정 또한 변경될 필요가 있으므로, 확장성이 떨어지게 된다.

OODBMS에의 종속성이 좀 더 극단적으로 나아가는 경우, Illustra와 같이 사용자 정의 메소드에 사용가능한 인자의 자료형을 시스템에서 제공하는 몇가지의 자료형으로 제한하기도 하는데[21], 역시 확장성에서 문제를 겪게 된다 또한, 이러한 제한은 사용자가 몇 개 이상의 인자나 자료형을 필요로 하지 않을 것이라는 가정에 기반한다. 그러나 이러한 접근법은 사용자에게 많은 제약을 준다. 메소드를 설계할 때 기존의 프로그래밍 언어가 가지는 여러 가지 제약 이외에도 메소드의 인자에 대해서 새로운 제약이 더 존재한다는 사실을 배워야만 하는 것이다.

3. 개선된 접근 방식

개선된 메소드 지원 시스템을 설계하는 기본 개념은 사용자가 C++로 정의한 클래스(class)와 메소드를 전처리(preprocessing)하는 데 있다. 사용자가 다음과 같은 클래스를 정의하고 그것을 데이타베이스에 저장할 자료의 스키마로 사용하기를 원한다고 가정하자.

```
class MyClass : public d_Object {
private :
// some code
public  :
int f( int, float*, double& );
};
```

이 클래스는 ODMG 2.0 표준에 맞추어 정의되었다[1][1]. 이 클래스는 메소드로 int f(int, float*, double&)을 가지고 있다. 이 클래스를 전처리해서 다음과 같은 코드를 생성한다. 제시된 코드는 원래 헤더 파일(header file: *.h)과 정의 파일(definition file: *.C)의 두 부분으로 나누어 생성되지만, 설명의 편의를 위하여 하나로 합쳐서 제시하였다.

```
class MyClass : public d_Object {
private :
// some code
public  :
int f( int, float*, double& );
};

// this is generated function declaration
extern "C" call_MyClass_f_vIpFrD(d_Ref_Any,callarg&);

// function definition
int call_MyClass_f_vIpFrD(d_Ref_Any object, callarg& arg)
{
        d_Ref<MyClass> obj = object;
        return obj->f(*(int*)(arg[0]), *(float**)arg[1],
*(double*)arg[2]);
}
```

전처리된 코드의 핵심적 내용은, C++로 정의된 메소드를 불러주는 C 함수의 선언과 정의를 자동으로 생성한다는 것이다. 이러한 C 함수를 본 논문에서는 중계함수(relay function)라고 부르도록 하겠다. 위의 코드에서 call_MyClass_f_vIpFrD라는 이름으로 정의된 함수가 바로 중계 함수에 해당한다. 중계 함수의 이름 안에는 자신이 호출하는 C++ 메소드의 이름과, 메소드가 속한 클래스, 그리고 인자의 형과 개수에 관한 정보들이 담겨져 있다. 또한 그 정의(definition) 부분에는 인자로 넘어온 MyClass 형의 객체에 대해서 실제로 f()을 호출해주는 코드가 담겨져 있다.

질의어 처리 시스템은 이제 사용자가 정의한 C++ 메소드를 직접적으로 부르는 것이 아니라, 대신 그 메소드를 부르는 중계 함수를 호출하게 되는 것이다. 이러한 방법을 취할 경우, 다음과 같은 이득이 주어진다.

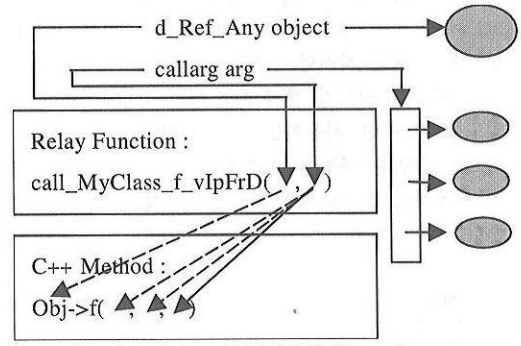


그림 1 중계함수를 통한 C++ 메소드의 호출과 인자전달

1. **사용자는 메소드를 정의할 때 그 인자의 개수에 대해서 어떠한 제약도 받지 않는다.** 실제 메소드 호출은 전처리기에 의해서 생성된 중계 함수가 담당하고 있으며, 전처리기는 사용자가 제출한 헤더 파일을 분석해서 인자 개수에 따라 적절한 함수 호출을 하도록 중계 함수 코드를 생성하기 때문이다. 중계 함수의 인자는 항상 d_Ref_Any 형과 callarg 형의 두 가지로 고정되어 있다. d_Ref_Any는 메소드 호출의 대상이 되는 객체를 가리키는 포인터의 역할을 한다. callarg는 메소드에 인자를 전달하기 위한, 인자들을 가리키는 void형 포인터들의 리스트이다. 이 중계함수를 적절한 인자와 함께 호출하는 책임은 어디까지나 시스템이 맡고 있다. 사용자가 질의문을 대화형 환경에서 입력시키면, 시스템은 사용자의 질의문을 분석하여 인자와 메소드 호출을 받을 객체를 추출해내며, 인자를 callarg 형의 객체로 만들어 중계 함수를 호출할 때 넘겨주게 된다(그림 1). callarg 형의 객체 내에 들어있는 실인자들을 원래의 C++ 메소드

1) ODMG 표준은 객체지향 데이타베이스에서 지속성(persistence)을 가지는 객체의 클래스가 d_Object 클래스의 하위 클래스로 선언되어야 함을 명시하고 있다. 따라서 MyClass는 d_Object의 하위 클래스로 정의되었다.

에 배분해주는 코드는 전처리기가 자동생성하였으므로, 사용자는 인자개수가 몇 개인지, 그리고 그 인자들이 원래의 C++ 메소드에 제대로 전달되었는지에 대해서는 신경쓸 필요가 없다. 그러므로, 시스템 구현자의 입장에서 보면 인자의 개수에 따라 비슷한 코드를 중복해 여러 번 나열해야 하는 부담이 덜어지는 것이고, 사용자의 입장에서 보면 인자의 개수에 신경쓰지 않아도 좋다는 장점이 있게 된다.

2. **사용자는 메소드의 인자 형에 있어서 제약을 받지 않는다.** 위의 예에서 확인할 수 있듯이, 전처리기는 C++ 메소드를 호출할때 callarg 형의 객체 내에 리스트 형태로 존재하는 void형의 포인터 들을 실제 C++ 메소드의 인자 형에 맞도록 형변환(type casting) 해주는 부분을 *(int*) 과 같이 중계 함수 코드에 포함시킨다. 이처럼 전처리기에 의해 생성된 형변환 코드는 시스템 구현자로 하여금 인자의 형에 따라 비슷한 코드를 중복해야 하는 부담을 피할 수 있도록 해 주며, 사용자는 인자를 전달하는 과정에서 필요한 형변환 과정에 신경쓰지 않고 보통 프로그램을 사용하듯 편안하게 인자를 전달 할 수 있도록 해 준다.

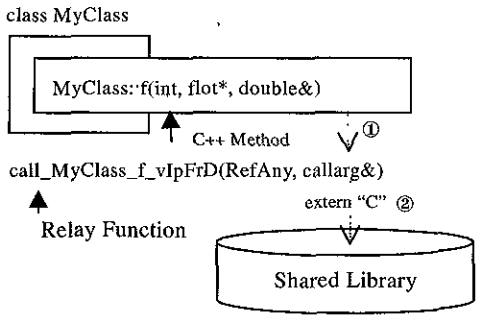


그림 2 컴파일러 중립적 이름의 생성

3. **최대한 컴파일러 중립적(compiler- independent)인 구조를 유지할 수 있으며, 컴파일러, OS, 하드웨어에 의한 제약을 받지 않으므로 시스템 간에 이식성(portability)이 증진된다.** 보통 컴파일된 C++ 메소드는 대화형 환경 내에서의 동적인 호출을 위해 공유 라이브러리(shared library) 형태로 저장된다. 그런데 C++ 컴파일러는 이 과정에서 애초의 C++ 메소드 이름을 C 함수명으로 변환해서 저장한다. 새로이 생성된 C 함수명은 그 메소드가 속한 클래스 정보와, 메소드의 인자 형에 관련된 정보를 포함하고 있다. 이러한 과정을 일반적으로 이름변환(mangling)이라고 부른다. 대부분의 C++ 컴파일러에서 함수 오버로딩을 지원하기 위해 이러한 방식을 채택하고 있다. 이 이름변환 과정의 구현은 컴파일러나 OS 마다 다르며, 따라서 라이브러리에 저장되는 C 함수명의 형태도 컴파일러마다 다르다. 따라서 중계 함수를 사용하지 않고 직접 메소드 호출을 하려면 프로그래머는 컴파일러가 수행하는 이름변환 과정과 동일한 변환 과정을 직접 구현한 후, 찾고자 하는 메소드의 이름을 생성하고, 그 이름으로 공유 라이브러리 내의 메소드를 찾아내어 호출해야만 한다. 그런데 이름변환 과정이 컴파일러마다 다를 수 있다고 했으므로, 중계 함수를 사용하지 않는 시스템은 공유 라이브러리에서 메소드를 찾아내는 루틴의 컴파일러 종속성 때문에 생기는 이식성의 저하를 피할 수 없게 된다. 그러나, 전처리 방식을 따르게 되면 시스템이 호출하는 대상이 C++ 함수에서 중계함수로 이동하게 되므로, 그림 2처럼 이름변환의 대상 또한 중계 함수로 바뀐다. 그런데 이 중계 함수는 C 함수이므로, extern "C" 키워드를 이용하여 원래 이름에 변화를 가하지 않고 그대로 공유 라이브러리에 저장할 수 있다[2]. 즉, call_MyClass_f_vIpFrD와 같이 메소드 지원 시스템 내부적으로 생성된 중계 함수명이 공유 라이브러리 내에 아무 수정 없이 저장되며, 시스템은 그 이름을 메소드를 호출할 때 이용하게 된다는 것이다. 결국, 호출되는 중계 함수의 이름을 컴파일러와 무관하게 만들 수 있다는 특성을 이용하면 메소드 호출과정을 시스템 중립적으로 만드는 것이 가능해 지는 것이다.

4. **중계 함수 이름에 클래스 정보와 인자 형에 관한 정보를 포함시킴으로써 함수 오버로딩(function overloading) 지원문제가 해결된다.** 보통 객체지향 언어에서는 여러 개의 함수 정의(function definition)가 동일한 함수명을 가질 수 있도록 허용하고 있는데, 이것을 함수 오버로딩이라고 부른다. 오버로딩된 여러 개의 함수 가운데서 호출 대상을 정확하게 구별해 내기 위해서는 함수명만으로는 충분하지 못하며, 인자 형과 개수에 관한 정보가 추가적으로 필요하다. 따라서 데이타베이스 관리 시스템 상에서 함수 오버로딩을 지원하기 위해서는 질의문에 포함된 메소드의

2) 이러한 메커니즘은 오직 C 함수에 대해서만 적용 가능하다. C++ 메소드에 대해서는 이러한 메커니즘을 적용할 수 없다. 이 기능은 기존에 존재하는 C 라이브러리를 재컴파일 없이도 사용할 수 있도록 하기 위해서 제공되었다.

인자 개수와 형을 모두 고려하여 호출될 함수를 찾아내야만 한다. 이러한 문제는 중계 함수명을 생성할 때 이에 관련된 정보를 그 이름 내에 포함시키도록 독자적인 이름 변환 루틴을 구성하면 해결될 수 있다. 아울러 여러 클래스에 동일한 이름의 함수가 존재할 수 있는데, 이러한 함수들의 검색 문제는 중계 함수명에 클래스 관련 정보를 삽입함으로써 해결할 수 있다.

5. **데이타베이스 관리 시스템의 확장성이 향상된다.** 중계함수를 이용한 전처리 방식은 데이타베이스 관리 시스템과 아무런 연관 없이 독립적으로 동작한다. 데이타베이스 관리 시스템 내부적으로 사용하는 자료형에 메소드 지원 시스템이 밀접하게 관련되어 있는 다른 시스템들과는 달리, 중계 함수를 도입한 전처리 방식은, 그러한 자료형들과는 무관하다. 사용자가 입력한 코드에 포함된 여러가지 자료형들에 대한 형검사는 중계 함수 코드와 사용자 코드를 컴파일하여 공유 라이브러리에 저장되는 순간에 정적으로 검사되므로, 형 검사 과정 또한 데이타베이스 관리 시스템과는 상대적으로 무관하게 동작한다. 따라서, 추후 데이타베이스 관리 시스템이 관리하는 자료형이 변경되더라도, 메소드 지원 시스템 자체의 코드를 변경해야 하는 경우는 없다. 그러므로 데이타베이스 관리 시스템 자체의 확장성이 증진된다.
6. **구현이 용이하다.** 데이타베이스 관리 시스템에 중립적이라는 특성은 시스템의 확장성을 증진시킬 뿐 아니라, 메소드를 지원하지 않는 기존의 객체지향 데이타베이스 관리 시스템의 확장 또한 용이하게 해준다. 컴파일러 기반 언어를 사용하는 객체지향 데이타베이스 관리 시스템의 경우, 중계 함수를 이용한 전처리 방식을 도입할 경우, 손쉽게 메소드 지원 시스템을 설계하고 구현할 수 있는 것이다.

이처럼, 중계 함수 개념을 도입한 전처리 방안은 간단한 개념에도 불구하고 인자의 개수나 형과 관련된 제약을 없앨 수 있다는 장점을 가질 뿐만 아니라, 데이타베이스 관리 시스템의 이기종간 이식성을 증진시키고, 함수 오버로딩 등 객체지향 데이타베이스 관리 시스템에 특징적인 사항들을 쉽게 처리할 수 있다는 장점을 가지게 된다.

4. 메소드호출 처리과정과 시스템 구조

제시한 전처리 방안은 서울대학교 객체지향 시스템 연구실(SNU OOPSLA Laboratory)에서 지난 5년간 개발한 객체지향 데이타베이스 관리 시스템 SOP(SNU OODBMS Platform)[22, 23]에 구현되었다. 메소드 지원과 관련해서 SOP 시스템 중 SOPOQLES(SOP Object Query Language Extensible System)과 임포트 툴(Import tool)이 변경되었고, 메소드 전처리 모듈(method preprocessing module)이 새로이 구현되어 SOP 에 첨가되었다. 아울러 중계 함수명을 생성하기 위해 이름변환 루틴(mangling routine)이 구현되었다.

SOPOQLES는 ODMG 2.0 표준에 맞게 설계된, OQL(Object Query Language)을 위한 질의어 처리기이며, 임포트 툴은 C++ 헤더 파일(header file)에서 스키마 정보를 추출하여 데이타베이스에 등록하는 모듈이다. 메소드 전처리 모듈은 중계 함수 개념을 도입한 전처리 방안을 SOP 상에서 구현한 것으로서, 상술한 여러 가지 문제점을 해결하고 있다. 또한 메소드에 관련된 스키마 정보를 관리하기 위해 스키마 관리자(schema manager)가 변경되었다. 이 외에도 이름변환(mangling) 루틴, 메소드 호출 루틴 등이 라이브러리로 만들어졌다[3]. 그림 3은 이러한 루틴들이 어떻게 맞물려 움직이는지를 보여준다.

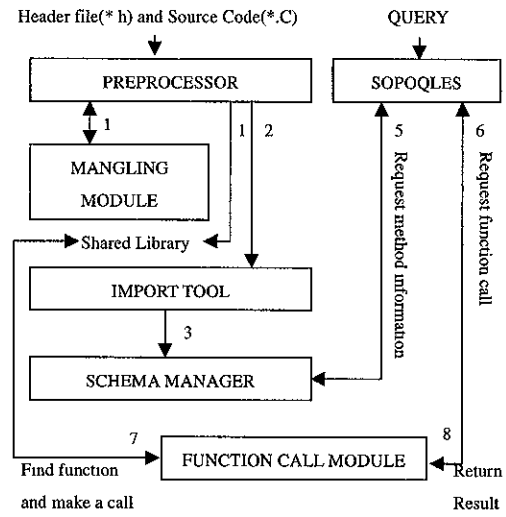


그림 3 메소드 호출이 이루어지는 과정

4.1 메소드 전처리기

메소드 전처리기는 메소드 지원을 위한 가장 핵심적인 부분이다. 이 모듈은 사용자가 제출한 C++ 헤더 파

3) 전처리기, 이름변환 루틴, 메소드 호출 루틴을 구성하는데 7개의 클래스에 약 1300라인의 코드가 소요되었으며, 기존의 모듈 중 400 라인 정도가 변경되거나 추가되었다.

일을 분석하여 메소드를 호출하는 중계 함수 코드를 만들어낸다. 생성된 중계함수 코드와 C++ 메소드 코드는 함께 컴파일되어 공유 라이브러리(shared library)형태로 저장된다. 이 과정에서 중계 함수명을 생성하기 위해 이름변환 루틴을 이용하게 된다.

이름변환 루틴은 전처리 단계에만 사용된다. 이 루틴은 메소드가 속한 클래스, 메소드의 인자 개수, 메소드의 인자 형 등의 정보를 이용해서 다음과 같은 중계 함수명을 생성한다.

call_<ClassNamc>_<FunctionName>_<ArgumentSpecifiers>

ClassName은 그 메소드가 속한 클래스를 지칭하는 것이고, FunctionName은 C++ 메소드 이름을 나타내는 것이다. ArgumentSpecifier는 종류에 따라 인자의 전달 방식을 나타내는 부분과 인자의 형을 나타내는 부분의 두 가지로 구분 지울 수 있다. 표 1에 그 표기법이 제시되어 있다.

표 1 Argument Specifier의 표기법

전달방식		인자 형	
값 전달	v	정수형(integer)	I
참조 전달(&)	r	부동소수점형(float)	F
포인터 전달(*)	p	부동소수점형(double)	D
		문자형(char)	C
		부울형(bool)	B
		일반객체참조형(d Ref Any)	R
		길이가 n인 사용자정의형	n<사용자정의형 이름>

표 1의 표기법에 따르면, 만일 인자의 형과 전달 방식이 int, float*, d_Ref_Any &와 같이 주어져 있다면, ArgumentSpecifier는 vIpFrR과 같이 나타나게 된다. 따라서, Class MyClass의 메소드 int myfunction(int, float*, d_Ref_Any&)로부터는 call_MyClass_myfunction_vIpFrR이 중계 함수 이름으로 생성되는 것이다.

4.2 임포트 툴

전처리가 끝난 C++ 헤더 파일은 임포트 툴(import tool)로 넘겨진다. 임포트 툴은 사용자가 제출한 헤더 파일을 분석하여 스키마 정보를 생성하는 일을 한다.

메소드를 지원하기 위하여, 파서는 C++ 헤더 파일을 분석하여 메소드 선언을 만나게 되면 그 메소드의 인자 개수, 인자 형, 메소드 이름과 클래스 정보, 복귀 형, 기본값 인자(default argument) 등의 정보를 검출해 낸다. 전처리기가 생성한 중계 함수의 선언이나 정의는 이 과정에서 전혀 이용되지 않는다. 오직 사용자가 제출한 C++ 클래스의 정의만이 이용된다. 중계 함수명을 유지하는 대신 C++ 클래스를 분석하여 이러한 정보들을 모두 유지하려고 하는 이유는 이후에 SOPOQLES에서 실제 함수 호출을 필요로 할 때 후보 중계 함수명들을 생성하는 과정에서 반드시 필요하기 때문이다. 후보 중계 함수명 선정 과정에 대해서는 뒤에서 설명한다.

이들 정보는 스키마 관리자에 기록된다. 또한 이들 정보는 소스코드 생성기로 전달되어, 데이타베이스의 작동에 필요한 여러 가지 함수를 자동 생성하는 과정에 이용된다. 소스코드 생성기에서 생성하는 정보는 클래스 내에 정의된 변수에 관련된 것들이므로 메소드 처리와는 무관하다. 이들 코드는 컴파일되어 임포트 툴이 관리하는 공유 라이브러리에 저장된다.

4.3 SOPOQLES

위와 같은 과정을 거쳐 함수에 관한 정보가 등록되고 나면, 실제 함수 호출이 가능해진다. 대화형 OQL의 사용자가 메소드 호출이 필요한 질의어를 제출하면, SOPOQLES는 질의문 내의 메소드 호출문을 파싱하여 실인자와 인자 형에 대한 정보를 추출한 후, 원래의 C++ 메소드 호출을 대응되는 중계 함수에 대한 호출로 바꾸어 수행한다. 따라서 SOPOQLES 내에서의 함수 호출은 두 단계로 이루어진다. 첫번째는 사용자가 제출한 질의문 내의 정보를 바탕으로 실제 호출 대상이 되는 중계 함수를 선정하는 것이며, 두번째는 함수 호출 루틴에 선정된 중계 함수명과 실제 인자를 넘겨주고 호출을 하는 단계이다.

첫번째 단계는 함수 호출을 구성하는데 가장 중요한 부분이다. 가령, 사용자가 retire(1995)라는 함수 호출을 질의문 내에 집어넣었다고 가정해보자. 만일 retire() 함수가 C++로 정의되어 있다고 하면, 이 함수는 기본값 인자(default argument)를 가지도록 선언되어있을 가능성이 있다. 다시 말하면 사용자는 retire(1995)로 함수 호출을 하였지만, 이 함수의 실제 선언은 int retire(int, int = 0)와 같이 되어 있을 가능성이 있다는 것이다. 따라서, 사용자가 입력한 함수 호출문을 분석하여 호출될 중계 함수 이름을 만들어 낼 때, 추가적인 해석 작업을 거치지 않는다면 라이브러리에 저장된 중계 함수를 찾지 못할 가능성이 생기는 것이다. 변경된 SOPOQLES 시스템은 이 문제를 해결하기 위해서, 다음과 같은 과정

을 거쳐 메소드 호출을 처리한다.

a. **질의문 파싱** 사용자가 제시한 질의문을 분석하여, 메소드 호출이 있다면 그 메소드의 클래스, 실인자, 인자 형을 분석해 낸다.

b. **스키마 관리자 참조** 스키마 관리자를 참조하여 사용자의 메소드가 속한 클래스를 검색하여 그 클래스에 속한 메소드의 리스트를 얻어온다. 리스트의 원소들은 각각 메소드 이름, 인자 개수, 기본값 인자 개수, 인자 형에 관한 정보를 관리한다.

c. **리스트 검색** 리스트 아이템들을 검색하여 사용자가 제시한 메소드 이름과 일치하는 것이 있는지 알아본다. 동일한 이름의 리스트 아이템에 대하여 다음 과정을 수행한다.

c-1. **형 검사** 첫번째 인자값부터 형이 일치하는지 검사한다. 만일 일치하지 않는 경우가 발생하면, 사용자가 제시한 메소드에 제출된 실인자의 형을 현재 리스트 아이템의 인자형으로 형변환(type casting)할 수 있는지 알아본다. 가능하다면, 다음 인자에 대한 형검사를 수행한다.

c-2. **기본값 인자 검사** 사용자가 제시한 메소드의 인자에 대해서 형검사가 끝났으나 현재 리스트 아이템의 인자 개수(c)와 사용자 제시 메소드의 인자 개수(u)가 다른 경우에 개시한다. 기본값 인자 개수가 $c-u$ 값과 동일하면 검사가 성공적으로 끝난 것이다.

c-3. **후보 중계 함수 선정 종료** 형 검사와 기본값 인자 검사가 모두 성공적으로 끝났다면, 중계 함수 선정 과정을 종료하고 함수 호출을 시도한다. 만일 그렇지 않다면, 리스트 검색 과정으로 돌아간다.

4.4 메소드 호출 루틴

메소드 호출 루틴은 SOPOQLES 에서 공유 라이브러리에 저장된 메소드를 실행시간(run-time)에 부르기 위해 사용하는 라이브러리 함수들의 집합이다. 현재 다음과 같은 함수들이 정의되어있다

```
int dlcall_int(char*,d_Ref_Any,callarg&);
double dlcall_double(char*,d_Ref_Any,callarg&);
char dlcall_char(char*,d_Ref_Any,callarg&);
bool dlcall_bool(char*, d_Ref_Any, callarg&),
void dlcall_void(char*, d_Ref_Any, callarg&);
char* dlcall_charp(char*,d_Ref_Any,callarg&);
d_Ref_Any dlcall_RefAny(char*,d_Ref_Any,callarg&);
```

이 루틴들은 호출될 중계함수 이름, 그 메소드가 정의된 클래스의 객체, 그리고 메소드에 넘겨질 인자들의 리스트를 인자로 받는다. 이 루틴들은 공유 라이브러리(shared library)를 동적으로 검사해 원하는 함수에 대한 포인터를 찾고, 함수를 호출한다. 이 함수들은 복귀형에 따라서 각각 하나씩 정의가 되어 있다 왜냐하면, 비록 전처리 방법에 의해서 인자들의 개수와 그 형에 대한 제약은 제거하였으나 복귀형에 관련된 제약은 전처리 방안으로도 제거하기 힘들기 때문이다. 메소드 호출 결과를 보존하기 위해서 SOPOQLES 내부에 정의되는 변수의 형은 반드시 컴파일 시에 하나로 고정되어야 하는 것이다. 따라서, 결국 복귀 형에 맞추어 지정된 dl_call* 함수를 호출하는 코드를 중복해서 사용하지 않을 수 없게 된다. 따라서 추가적인 복귀형이 필요한 경우에는 그 복귀형에 대해 위와 같은 함수를 작성하여 추가한 후, SOPOQLES 코드 중 메소드 호출 루틴을 부르는 부분을 적절히 변경시켜야 한다.

5. 관련 연구

객체지향 데이타베이스 관리 시스템에서 메소드를 지원하는 것에 대한 연구는 주로 컴파일러 기반 언어를 이용하도록 설계된 데이타베이스 관리 시스템을 중심으로 활발하게 이루어져 왔다. 이는 인터프리터(interpreter) 기반 언어를 표준으로 사용하고 있는 시스템들의 경우 메소드를 지원하기가 상대적으로 쉬운 편이기 때문이다. 이에 대해서는 앞서 살펴보았다.

그러나, 컴파일러 기반 언어를 사용하는 시스템은 메소드 호출을 반드시 데이타베이스 시스템 내부에서 처리해주어야 한다. 이러한 시스템들은 대개 C++를 데이타베이스 관리 시스템을 구성하기 위한 표준 언어로 사용하고 있으며, 사용자가 이 언어를 통해 데이타베이스 관리 시스템을 사용할 수 있도록 C++ 바인딩(binding)을 지원하고 있다[4]. 그러나 상용화된 대부분의 시스템들이 C++ 바인딩을 지원하면서도 OQL을 ODMG 표준에 맞게 완벽하게 지원하지는 않고 있는데, 이것은 지금의 ODMG 표준이 C++ 바인딩과 OQL을 동시에 제공할 것을 명시적으로 요구하지 않고 있기 때문이다[12].

따라서 ODMG 표준에 맞는 OQL을 제공하는 시스템은 무척이나 드문 상황이며, 설사 OQL을 제공하더라도 ODMG 표준을 따르지 않거나, ODMG 표준을 따른다고 하더라도 부분적으로만 따르고 있는 실정이다. 메

4) C++ 바인딩은 C++ 언어로 DB 응용 프로그램을 개발할 수 있도록 하는 것을 말하며, 보통 데이타베이스 관리 시스템의 API를 부를 수 있는 라이브러리를 구현하는 것으로 이루어진다.

소드 지원에 있어서 이러한 문제는 더욱 심각하여, ODMG 표준에 맞는 메소드 호출을 완벽하게 지원하고 있는 시스템은 거의 없다. Poet은 ODMG 표준을 따르는 OQL을 제공하나 메소드 호출은 아직 지원하지 않고 있으며[19], Versant도 마찬가지[20]이다. 기타 시스템들에 대해서는 앞서 밝혔다. 표 2에 이러한 지원 여부를 요약해놓았다.

표 2 데이타베이스 관리 시스템별 OQL지원여부

제품명	ODMG표준	OQL지원
GemStone	X	X
Orion	X	△
O2	O	O
ObjectStore	X	△
POET	O	△
Versant	O	△

그러나 O2는 예외적으로 ODMG 표준에 맞는 OQL을 완벽하게 지원한다[24,17]. 사용자의 측면에서 보면 메소드를 정의할 때 O2에서 허용하는 인자형만을 메소드의 인자로 전달할 수 있도록 해야하는 제약이 존재하긴 하나, 현재 ODMG 표준은 O2에서 허용하는 인자형들이 지원한다면 충분히 만족될 수 있는 수준이다.

그러나 O2는 앞서 밝혔듯이 사용자가 정의한 C++ 클래스와 메소드 정보를 시스템 내부적으로 관리하는 자료형으로 변환하는 과정이 시스템 내부의 자료형에 종속적이므로, 시스템의 확장성이 떨어진다는 단점을 가지게 된다. 대표적인 예로 ODMG 자료형이 확장되거나 시스템 내부적으로 자료형의 선언이 변경되는 경우를 들 수 있다.

전처리 방식은 위에서 제시한 여러 가지 문제들에 대해 간결한 해결방안을 제시한다. 현재 C++ 바인딩만을 지원하는 OODBMS 시스템이 OQL을 지원하기 위해 시스템을 재설계 하는 경우, 전처리 방식은 최소의 비용으로 메소드 지원에 대한 요구를 만족시키는 동시에, 확장성의 증진, 사용자 제약의 제거 등의 여러 가지 이점 또한 지니게 된다. 또한 향후 표준이 변경되거나 하였을 때, OODBMS 시스템이 좀 더 유연하게 확장되도록 한다는 점에서 의의를 가진다.

6. 결론 및 향후 연구방향

본 논문에서는 객체지향 데이타베이스 관리 시스템에서 메소드를 지원하려고 할 때 발생하는 문제들을 짚어보고, 그 해결방안으로써 중계 함수 개념을 도입한 전처리 방식을 제안하였다.

전처리방식은 기존의 방안들과는 달리 코드의 복잡도가 낮으면서도, 컴파일러 중립적이고 사용자의 제약을 최소화하며, 시스템 확장성을 증진시킨다. 아울러, 전처리 방식은 데이타베이스 관리 시스템에 독립적으로 동작하므로, 기존의 OODBMS 시스템에 손쉽게 적용이 가능하다는 잇점을 지닌다. 또한, 전처리 방안은 C++이 아니더라도 컴파일러 기반 언어로 구현된 시스템이라면 용이하게 적용될 수 있다. 그러나 본 방식은 C++ 헤더를 데이타베이스 스키마로 변환하기 위해서 필수적으로 거쳐야 하는 한번의 전처리과정 외에도, 메소드를 지원하기 위해 추가적인 전처리과정을 한번 더 거쳐야 하므로 그에 따른 성능저하도 어느 정도 있게 된다.

향후 더 연구되어야 할 구현상의 문제들은 여러 가지가 있겠으나, 일단 정의된 메소드를 라이브러리로 관리할 것이냐 아니면 DBMS 내에 한 자료형태로 관리할 것인가 하는 문제가 있을 수 있다. 두가지 중 어느 방법을 선택할 것인가 하는 문제의 선택기준으로서 객체지향 개념에의 부합정도, 그리고 구현의 용이성과 성능 등 여러 가지를 들 수 있겠으나 아직 구체적인 비교는 나오지 못하고 있는 실정이다.

아울러 본 논문에서는 구현상의 문제만을 중점적으로 다루었으나, 메소드를 포함한 질의문을 최적화하는 방안에 대한 이론적인 연구도 중요한 분야 중 하나이다. 본 논문의 서두에서 간단하게 밝혔듯이, 메소드는 그 비용산정(cost evaluation)의 어려움으로, 기존의 질의 최적화 패러다임을 그대로 적용할 수 없도록 만든다[6, 4]. 그래서 지금까지 메소드를 포함한 질의어를 최적화 하기 위해서 다양한 연구[10, 9, 11]가 이루어져 왔으나, 아직 실제로 OODBMS에 적용된 해결책은 드문 실정이다. 따라서, 향후 구현과 이론 양 측면에서 메소드에 대한 심도 있는 접근이 이루어져야 할 것이다.

참 고 문 헌

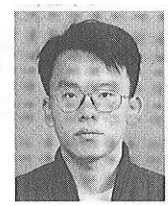
[1] R.G.G. Cattell and Douglas K. Barry, *The Object Database Standard: ODMG 2.0*, Morgan-Kaufmann Publishers, Inc., 1997.

[2] Won Kim, Modern Database Systems, ACM Press, 1995.

[3] Won Kim, Introduction to Object-Oriented Databases, MIT Press, 1990.

[4] Elisa Vertino, Mauro Negri, Giuseppe Perlagatti and Licia Sbatella, "Object-Oriented Query Languages: The Notion and the Issues", IEEE TKDE, June 1992.

[5] Dave D. Straube and M. Tamer Ozsu, "Queries and Query Processing in Object-Oriented Database Systems", ACM TOIS, October, 1990.
[6] Gail Mitchell, Stanley B. Zdonik and Umeshwar Dayal, "Object-Oriented Query Optimization: What's the Problem?", Technical Report No. CS-91-41, Brown University, June, 1991.
[7] Jay Banerjee, Won Kim and Kyung-Chang Kim, "Queries in Object-Oriented Databases", ICDE, 1988.
[8] Matthias Jarke and Jurgen Koch, "Query Optimization in Database Systems", ACM TODS, 1984.
[9] Karl Aberer and Gisela Fischer, "Semantic Query Optimization for Methods in Object-Oriented Database Systems", ICDE, 1995.
[10] G. Graefe and D. Maier, "Query Optimization in Object-Oriented Database Systems: The REVELATION Project", volume 334 of Advances in Object-Oriented Database Systems, Lecture Notes in Computer Science, Sringer-Verlang, 1988.
[11] Praveen Seshadri, Miron Livny and Raghu Ramakrishnan, "The Case for Enhanced Abstract Data Types", VLDB, 1997.
[12] Michael J. Carey and David J. DeWitt, "Of Objects and Databases: A Decade of Turmoil", VLDB, 1996.
[13] Paul Butterworth, Allen Otis and Jacob Stein, "The GemStone Object Database Management System", CACM, October 1991.
[14] Sophie Gamerman, Personal Communication, O2 Technology, 1997.
[15] Didier Plateu, Personal Communication, O2 Technology, 1997.
[16] Patrick Borras, Personal Communication, O2 Technology, 1997.
[17] Kenneth H. Sinclair, Personal Communication, Object Design. Inc., 1997.
[18] Gordon Landis, Personal Communication, Object Design. Inc., 1997.
[19] Dirk Bartels, Personal Communication, POET Software, 1997.
[20] Henry Strickland, Personal Communication, Versant Object Technology, 1997.
[21] Illustra User's Guide - Illustra Server Release 3.2, Illustra Information Technology. Inc., 1995.
[22] SOP Implementation Note - Version 1.0, SNU OOPSLA Lab, 1997.
[23] SOP User's Guide - Version 1.0, SNU OOPSLA Lab, 1997.
[24] The O2 System - OQL User Manual, O2 Technology, 1996.



이 병 준
1996년 서울대 컴퓨터공학과(학사). 1998년 서울대 컴퓨터공학과(석사). 198년 ~ 현재 서울대 컴퓨터공학과 박사과정. 관심분야는 객체지향 시스템, 데이타베이스, 질의어 처리, 트랜잭션.

박 주 홍
1993년 서울대 계산통계학과(학사). 1995년 서울대 컴퓨터공학과(석사). 1995년 ~ 현재 서울대 컴퓨터공학과 박사과정. 관심분야는 객체지향 시스템, 데이타베이스, 질의어 처리.

김 형 주
제 4 권 제 3 호(C) 참조