ReCA CORBA 시스템을 위한 IDL 컴파일러

박 성 진[†] · 이 동 헌[†] · 김 영 곤[††] · 박 양 수[†††] · 이 명 준[†††]

요 약

OMG에서 제안된 CORBA는 분산 객체 컴퓨팅의 표준 기반구조를 제공한다. OMG IDL로 기술된 분산 객체의 인터페이스는 분산 객체와 관련된 오퍼레이션과 형을 정의함으로써 분산 객체가 지원하는 서비스를 명시한다. OMG IDL 컴파일러는 관련된 프로그래밍 언어의 형변환 뿐만 아니라 *클라이언트 스텁과 구현 골격을* 생성한다. 클라이언트 스텁은 요청을 생성하기 위한 부분이며, 구현 골격은 생성한 요청을 실제 객체 구현에게 전달하는 부분이다.

본 논문에서는 ReCA(a Reliable CORBA system using Ada95) 분산 객체 시스템을 위한 클라이언트 스텁과 구현 골격을 생성하는 방법에 관하여 기술한다. 또한 Ada95 언어 변환과 ReCA 시스템에 맞게 클라이언트 스텁과 구현 골격을 생성하는 IDL 컴파일러의 구현에 관하여 설명한다.

An IDL Compiler for ReCA CORBA System

Seong-Jin Park[†] · Dong-Hun Lee[†] · Young-Gon Kim[††] · Yang-Su Park[†††] · Myung-Joon Lee[†††]

ABSTRACT

CORBA announced by the OMG provides a standard infrastructure for distributed object computing. Interfaces for the distributed objects, described by OMG IDL, specify the services available to the distributed objects by defining the operations and types associated with those objects. An OMG IDL compiler generates *client stubs* and *implementation skeletons* as well as the associated programming language type mapping. A client stub is a mechanism for generating requests, while a implementation skeleton is a mechanism for delivering requests to the actual object implementation.

In this paper, we describe the method of generating client stubs and implementation skeletons for ReCA(a Reliable CORBA System Using Ada95) distributed object system. Also, we present an implementation of IDL compiler which generates client stubs and implementation skeletons appropriate to ReCA system, supporting Ada95 language mapping.

1. 서 론

오늘날의 컴퓨팅 환경은 컴퓨터 성능의 향상과 더불어 정보통신 기술의 급속한 발달로 인하여 네트워크를 통한 원거리 데이터의 공유와 정보 전달 속도가 향상되었으며, 네트워크상에 있는 서비스는 언제든지 이용 가능하게 되었다. 네트워크를 기반으로 한 클라이언트/서버 시스템은 분산 컴퓨팅 환경의 대표적인 기술이다. 분산 환경은 사용자에게 상호운용성(interoperability), 이식성(portability), 통합성(integrity), 확장성을 지원하여야 한다. 그러나, 개발자의 입장에서 본

※본 연구는 '95~'97년 정보통신부 대학 기초연구 지원 사업과제 지원으로 수행되었음.

† 준 회 원 : 울산대학교 전자계산학과
†† 정 회 원 : (주)퓨쳐시스템 정보통신연구소 근무
††† 정 회 원 : 울산대학교 전자계산학과

논문접수 : 1997년 6월 10일, 심사완료 : 1997년 12월 9일

다면 다양한 하드웨어와 소프트웨어의 결합으로 인한 네트워크의 이질성은 쉽게 분산 시스템 환경을 제공할 수 없게 한다. 분산 응용 프로그래밍은 기존의 단일 컴퓨터에서의 프로그래밍 보다 정보의 분산, 다양한 기종간 이질성(heterogeneity), 구현 언어에 대한 종속성, 네트워크의 이용에 따른 보안 등을 고려하여야 한다. 이러한 문제점들을 해결하기 위하여 다양한 기종간에 여러 응용프로그램의 통합을 표준화하는 작업이 활발히 진행되고 있다.

OMG(Object Management Group)에서 제안한 CORBA(Common Object Request Broker Architecture)는 이기종간 개방 분산 객체 시스템을 위한 표준안이다[2]. 개방 분산 객체 시스템(Open Distributed Object System)은 명확하게 정의된 표준 규격을 따라서 개발된 여러 개의 제품들이 상호운용성을 지원하면서 네트워크 투명성(transparency)을 보장하는 분산 시스템으로 분산 컴퓨팅의 기본 단위는 객체가 된다. 현재 CORBA 명세(specification)를 따라서 개발된 제품들이 출시되고 있으며 구현 중인 시스템들도 있다[4]. 이 중에서 ReCA(a Reliable CORBA system using Ada95) 시스템은 객체지향 프로그래밍 언어 중에서 최초로 국제 표준이 정해진 언어인 Ada95로 개발 중인 CORBA 시스템이다[8-11].

CORBA 시스템은 이질적인 분산 환경에서 응용프로그램들을 통합하기 위한 기반환경(infrastructure)을 제공하여 상위 수준의 객체 협력(object collaboration)을 제공하고 있으며 캡슐화(encapsulation), 인터페이스 상속(interface inheritance), 객체 기반 예외 처리(object-based exception handling)등의 객체지향적인 프로그래밍 기법들을 제공한다. 또한 네트워크를 통한 서비스의 요청(request)과 응답(reply)의 전달을 CORBA 시스템에서 처리해 주기 때문에 CORBA 응용프로그램의 개발자는 네트워크와 관련된 코드는 직접 작성하지 않아도 된다. 결과적으로 CORBA 시스템 상에서 분산 응용프로그램을 개발하는 것은 기존의 단일 컴퓨터 상에서 프로그램을 개발하는 것보다 어렵지 않다.

본 논문에서는 OMG IDL(Interface Definiton Language)에서 Ada95 클라이언트 스텁(client stub)과 구현 골격(implementation skeleton) 변환 규약을 ReCA CORBA 시스템에 적합하게 정의하고, 이 규약을 따르는 ReCA IDL 컴파일러의 구현에 관하여 기술하고자 한다.

본 논문의 구성은 다음과 같다. 1장 서론에 이어 2장에서는 CORBA 시스템, IDL 컴파일러와 ReCA 시스템에 관하여 살펴본다. 3장에서는 IDL 인터페이스 정의를 Ada95로 된 클라이언트 스텁과 구현 골격으로 변환하는 규약에 관하여 설명하고, 4장에서는 구현한 IDL 컴파일러의 구성과 컴파일러가 생성하는 파일들에 대하여 설명한다. 마지막으로 5장에서는 결론과 향후 연구 방향에 대하여 언급한다.

2. ReCA CORBA 시스템

2.1 CORBA 시스템

OMG의 CORBA는 분산 환경상에 존재하는 객체간의 상호운용성을 지원하며 요청과 응답의 투명성을 제공한다. CORBA 시스템에서 클라이언트는 서비스에 대한 요청의 생성자이며 객체 구현(Object Implementation)은 서비스 요청에 대한 처리자이다. 서비스의 처리 과정은 먼저 클라이언트가 클라이언트 스텁이나 동적 호출 인터페이스(Dynamic Invocation Interface)를 통하여 요청을 생성하며, 생성된 요청은 ORB(Object Request Broker)를 통하여 객체 구현쪽으로 전달된다. 전달된 요청은 구현 골격이나 동적 골격 인터페이스(Dynamic Skeleton Interface)를 통하여 객체 구현에게 전달되어 서비스가 처리된다. 처리의 결과인 응답은 ORB를 거쳐서 클라이언트에게 전달된다[2].

CORBA는 다음과 같은 요소들로 구성된다.

- ORB

ORB는 객체에게 요청을 전달하고 요청을 생성한 클라이언트에게 응답을 전달하는 부분이다. 일반적으로 ORB는 객체 위치, 객체 구현, 객체 실행 상태, 통신 메커니즘에 대한 투명성을 제공한다[14].

- 인터페이스 정의 언어(IDL)

객체의 인터페이스는 객체가 지원하는 오퍼레이션과 형을 명시한다. CORBA에서는 객체의 인터페이스를 언어 독립적인 IDL을 이용하여 기술한다. OMG IDL은 프로그래밍 언어가 아니라 선언적 언어(declarative language)이다. 이것은 객체가 서로 다른 프로그래밍 언어로 구현될 수 있도록 한다.

• 클라이언트 스텁과 구현 골격

클라이언트 스텁은 클라이언트가 정적으로 요청을 생성하고 전달하는 메커니즘이며 구현 골격은 정적 요청을 객체 구현으로 전달하는 메커니즘이다. IDL 컴파일러가 OMG IDL 인터페이스 정의를 이용하여 클라이언트 스텁과 구현 골격을 생성한다.

• 동적 호출

CORBA 시스템은 실행시에 인터페이스 정보가 저장된 인터페이스 저장소(Interface Repository)를 이용하여 동적 호출 인터페이스(dynamic invocation interface)를 통하여 요청의 생성을 지원한다.

2.2 IDL 컴파일러

CORBA 시스템에서 응용프로그램 개발 단계는 다음과 같다.

단계 1) IDL로 객체의 인터페이스를 정의한다.

IDL은 클라이언트 객체가 호출하고 객체 구현에서 제공하는 인터페이스를 기술하는데 사용하는 언어이다. IDL은 객체의 인터페이스, 오퍼레이션, 형을 정의하는데 특정 프로그램 언어에 독립적이며, 캡슐화, 인터페이스 다중 상속, 객체지향적 예외 처리 등의 객체지향 개념을 지원하는 선언적 언어이다. IDL은 분산 환경에서 특정 프로그래밍 언어에 독립적으로 CORBA 응용프로그램을 작성할 수 있게 해준다. 바꾸어 말하면 일단 IDL로 작성된 인터페이스 정의는 CORBA 시스템에서 제공하는 IDL 컴파일러를 통해 원하는 프로그래밍 언어에 적합하게 변환된다.

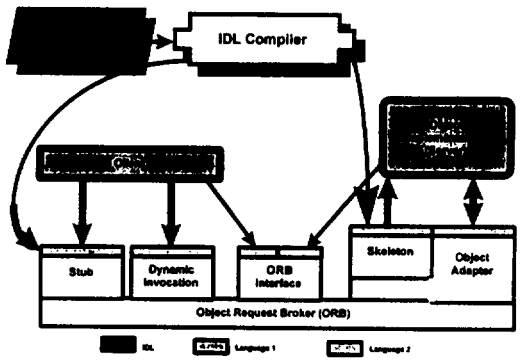
단계 2) 클라이언트 스텁과 구현 골격을 생성한다.

IDL 컴파일러는 IDL로 정의한 객체 인터페이스로부터 프로그래밍 언어로 된 클라이언트 스텁과 구현 골격을 생성한다. 클라이언트 스텁과 구현 골격은 서로 다른 언어로 작성될 수 있다. 클라이언트 스텁은 IDL로 정의한 오퍼레이션이 클라이언트가 호출 할 수 있는 특정 프로그래밍 언어로 변환된 인터페이스이다. 클라이언트 스텁은 각 인터페이스에 대한 요청을 생성하는데 특히, 인터페이스 호출에 필요한 메쏘드 선언을 제외한 나머지 부분을 내부적으로 감추며, 특정 ORB에 대하여 최적화된 상태로 만들어진다. 구현 골격은 객체 어댑터(Object Adapter)에 의존적으로 작성된 객체의 메쏘드에 대한 인터페이스이다. 이 인터페이스를 호출하여 ORB는 요청을 객체 구현에 전달한다. 이 인터페이스를 업-콜(up-call) 인터페이스라 한다[2].

단계 3) 개발자가 적절한 코드를 추가하여 완전한 응용프로그램을 개발한다.

마지막으로 IDL 컴파일러를 통하여 생성된 코드에 개발자가 적절한 코드를 추가하여 완전한 CORBA 응용프로그램을 작성하게 된다.

(그림 1)은 CORBA 응용프로그램의 개발과정에서 IDL 컴파일러의 역할을 나타낸 것이다.



(그림 1) IDL 컴파일러의 역할
(Fig. 1) Role of IDL compiler

IDL 컴파일러는 특정 CORBA 시스템을 위해 정의된 클라이언트 스텁과 구현 골격 변환 규약을 지원하는데, 분산 응용프로그램을 보다 신속히 개발하기 위하여 개발자의 코드 추가를 최소화하는 방향으로 구현되어야 한다. IDL 컴파일러를 개발하기 위해서는 먼저 IDL을 특정 CORBA 시스템에서 지원하는 프로그래밍 언어로 작성하기 위한 클라이언트 스텁과 구현 골격 변환 규약을 확정하여야 한다. 이 규약은 CORBA 시스템의 ORB 구현에 따라 달라지게 되며, IDL 컴파일러는 정의한 변환 규약에 따라 IDL로 정의된 인터페이스를 특정 프로그래밍 언어로 작성된 클라이언트 스텁과 구현 골격으로 생성하게 된다.

2.3 ReCA 시스템

ReCA(a Reliable CORBA system using Ada95)는 신뢰성(reliability), 추상화(abstraction), 효율성을 지

원하는 언어인 Ada95로 개발 중인 CORBA 시스템이다[8-11].

Ada95는 완전한 객체지향 프로그래밍 언어로는 최초로 국제 표준이 정해진 언어이다. Ada95는 강력한 형(type) 검사로 신뢰성을 증가시키며 패키지(package)와 보호 형(private type)을 통한 추상화와 태그드 형(tagged type)을 이용한 클래스-와이드(class-wide) 프로그래밍으로 유연성을 제공한다. 또한 보호 객체(protected object)를 이용하여 추가적인 제어가 필요 없는 효율적인 공유 데이터에 대한 접근을 제공하고, 태스크(task) 형을 제공하여 병행성 모델을 지원하는 객체지향 프로그래밍 언어이다[6, 20]. 그 밖에도 Ada95는 계층적인 라이브러리와 다른 언어와의 인터페이스도 제공하고 있다. ReCA 시스템은 이러한 Ada95의 특징을 이용하여 개발되고 있는 CORBA 시스템이다.

ReCA 시스템은 하위 통신 시스템과의 연계를 위하여 VOOL(Virtual Object-Oriented Low-Level Layer)을 정의하였고, 태스크 형과 보호 형을 이용하여 ORB와 BOA(Basic Object Adaptor)가 개발되어 있다[8-11].

3. 클라이언트 스텁과 구현 골격 변환 규약

3.1 IDL에서 Ada 언어로의 변환 규약

IDL 컴파일러는 IDL로 된 인터페이스 정의를 목적언어로 된 클라이언트 스텁과 구현 골격으로 변환하는 역할을 하기 때문에 IDL과 변환하고자 하는 목적 언어의 구조를 정확히 분석해야 한다. 이 분석을 바탕으로 IDL과 목적 언어와의 기본적인 변환 규약을 정의하여야 한다. OMG IDL에서 Ada 언어로의 기본적인 변환 규약은 〈표 1〉과 같다[1].

정의된 기본 변환 규약을 바탕으로 CORBA 제품에 의존적인 클라이언트 스텁과 구현 골격의 변환 규약을 정의하게 된다. IDL 컴파일러는 정의된 변환 규약에 적합한 코드를 IDL로 된 인터페이스 정의로부터 생성하게 된다.

기본 변환 규약에서 IDL 인터페이스 정의를 포함한 소스 파일은 Ada의 라이브러리 패키지로 변환이 되며, 소스 파일 안에 있는 모듈은 패키지로 변환되어야 한다. 소스 파일이나 모듈 안에 있는 인터페이스는 태그드(tagged) 형의 패키지로 변환된다. 태그드 형의 패키지로 변환하는 이유는 상속 기능을 지원하기 위함이다. 오퍼레이션은 Ada의 서브프로그램으로 변환이 된다. 오퍼레이션이 반환값이 존재하고 in 모드 인자들로 구성된다면, 오퍼레이션의 반환 형으로부터 변환된 Ada 형을 반환하는 Ada의 함수로 변환되어야 한다. 그 이외의 경우에 오퍼레이션은 Ada의 프로시져로 변환된다.

IDL의 속성은 Ada의 서브프로그램으로 변환되어야 한다. readonly 속성일 경우는 *attribute name*_Of라는 이름의 함수로 변환되어야 하며, 그렇지 않을 경

〈표 1〉 IDL에서 Ada로의 기본 변환 규약
〈Table 1〉 Basic mapping rule from IDL to Ada language

IDL construct	Ada construct
소스 파일(Source File)	라이브러리 패키지(Library Package)
모듈(Module)	패키지
인터페이스(Interface)	태그드형의 패키지
오퍼레이션(Operation)	서브프로그램(subprogram)
속성(Attribute)	"Set_*attribute name*"와 "*attribute name*_Of" 서브프로그램들
단일 상속(Single Inheritance) 다중 상속(Multiple Inheritance)	태그드형 상속 Mix-in 상속
데이터 타입들(Data Types)	Ada 타입들
예외(Exception)	예외와 레코드(record) 타입

〈표 2〉 IDL 타입에서 Ada 타입으로의 변환
〈Table 2〉 Mapping from IDL types to Ada types

IDL type	Ada type
short	CORBA.Short
long	CORBA.Long
unsigned short	CORBA.Unsigned_Short
unsigned long	CORBA.Unsigned_Long
float	CORBA.Float
double	CORBA.Double
char	CORBA.Char
octet	CORBA.Octet
boolean	CORBA.Boolean
any	CORBA.Any
string	Unbounded_String

우에는 함수 뿐만 아니라 Set_*attribute name*이라는 이름의 프로시져로도 변환되어야 한다.

IDL 데이터형에서 Ada 데이터형으로의 변환은 다음 〈표 2〉와 같다[1].

IDL이 상속을 지원하기 때문에 목적 언어로 변환할 때도 상속이 지원되도록 변환해야 한다. IDL에서 Ada 언어로의 변환에서 단일 상속의 경우는 생성되는 패키지 내에 CORBA 패키지의 테그드 형의 Object.Ref를 상속을 받아서 새로운 Ref 형을 선언한다. 그리고, 오퍼레이션은 첫 번째 인자로 Ref 형을 받음으로써 단일 상속을 지원할 수 있다. Ada에서는 직접적으로 다중 상속을 지원하지 않는데, 이 때 Mix-in 상속을 이용하여 구현할 수 있다. Mix-in 상속은 한쪽은 테그드 형을 상속받고 나머지 부분은 오퍼레이션을 다시 정의하여 다중 상속을 구현한다.

IDL의 예외는 Ada95의 예외 정보를 포함하는 레코드 형과 예외가 발생했을 때 예외의 정보를 얻을 수 있는 Get_Members( ) 함수로 변환되어야 한다.

3.2 기본 설계

ORB는 서비스를 의뢰하는 객체인 클라이언트가 생성한 요청을 서비스를 제공하는 객체 구현에게 전달한다. 이 때 클라이언트의 요청은 편리한 관리를 위하여 하나의 단위로 묶어서 전달하는 것이 바람직하다. ReCA 시스템에서는 요청의 기본 단위로 (그림 2)의 메시지 블록(Message Block)을 정의하여 사용한다.

(그림 2)에 Ada의 레코드 형인 Object가 메시지 블록이다. 각 구성 요소는 다음과 같다. HostName은 클라이언트가 존재하고 있는 호스트의 주소가 되며 Pid는 클라이언트의 프로세스 ID이다. MsgID는 한 클라이언트에서 생성되는 요청들에게 순차적으로 부여되는 값이다. ORB에서 요청을 관리하기 위하여 단일 식별자(UniqueID)로서 HostName, Pid, MsgID를 이용한다.

ObjectName은 실제 서비스를 제공하는 객체 구현의 이름이 되며, Operation은 수행하고자 하는 오퍼레이션의 이름이고, Arg_List는 in, inout 모드로 전달되는 실인자(arguments)나 out, inout으로 객체 구현쪽에서 클라이언트쪽으로 전달되는 결과 값들을 위한 것이다. Result는 오퍼레이션이 함수로 변환될 경우 결과 값을 전달받기 위한 것이고, ReqException

```
package CORBA.Request is
private
   type Object is record
      HostName     : CORBA.String;          -- Host Address
      Pid          : CORBA.Short;           -- Process ID
      MsgID        : CORBA.Long;            -- Request ID
      ObjectName   : CORBA.String;          -- Object Name
      Operation    : CORBA.String;          -- Operation Name
      Arg_List     : CORBA.NVList.Object;   -- Arguments
      Result       : CORBA.NamedValue;      -- Return Value
      ReqException : CORBA.String;          -- Exception Information
      Returns      : CORBA.Status;          -- Return Status
      Destination  : CORBA.String;          -- Destination Address
   end record;
end CORBA.Request;
```

(그림 2) 메시지 블록
(Fig. 2) Message block

은 오퍼레이션이 객체 구현에서 수행되는 중에 예외가 발생했을 경우, 예외의 종류를 저장한다. Returns는 오퍼레이션의 수행된 상태를 저장하는데 ST_OK, ST_EXCEPTION이나 ST_ERROR 값이 설정 될 수 있다. Destination은 객체 구현이 존재하는 컴퓨터의 주소 정보가 존재한다.

클라이언트쪽에서 설정하여 객체 구현에게 전달할 정보는 HostName, Pid, MsgID, ObjectName, Operation, Arg_List이며, 객체 구현에서 클라이언트에게 오퍼레이션 수행 후 보내줄 정보는 Arg_List, Result, ReqException, Returns이다.

3.3절과 3.4절에서는 〈표 1〉의 기본 변환 규약을 기반으로 하여 정의한 클라이언트 스텁과 구현 골격의

```
//
// dir.idl
//
exception NO_SUCH_ENTRY{};
enum Kind {File, Dirct};

module filesys{
  typedef boolean IsOpen;

  interface directory {
    void lookup(in char name, out any entry)
       raises(NO_SUCH_ENTRY);
  };
};
```

(그림 3) IDL 정의 예제
(Fig. 3) An example of IDL specification

변환 규약을 설명한다. (그림 3)의 IDL 정의를 이용하여 설명할 것이며 (그림 3)은 모듈 외부에 예외와 열거형이 선언되어 있으며, filesys 모듈 안에 typedef 문으로 Boolean 형의 IsOpen 데이터 형을 선언하고, directory 인터페이스를 갖고 있다. directory 인터페이스의 lookup 오퍼레이션은 인자로 in 모드 char 형과 out 모드 any 형을 가지며, raise문으로 NO_SUCH_ENTRY 예외를 기술하고 있다.

3.3 클라이언트 스텁 변환 규약

클라이언트 스텁은 클라이언트의 요청을 객체 구현에게 전달하고, 객체 구현에서 수행된 요청의 결과인 응답을 ORB로 부터 받아서 클라이언트에게 전달하는 부분이다. IDL 컴파일러는 IDL 인터페이스 정의를 목적 언어로 된 클라이언트 스텁으로 자동 생성하게 되는데, 이 때 변환 규약이 필요하다. 이 규약을 클라이언트 스텁 변환 규약이라 한다. 클라이언트 스텁 변환 규약은 IDL에서 Ada 언어로의 기본 변환 규약을 토대로 하여 IDL로 작성된 인터페이스 정의를 Ada 언어로 된 클라이언트 스텁 부분, 즉 요청을 생성하고 응답을 받아서 처리하는 과정을 정의하고 있다.

본 논문에서 정의한 ReCA 시스템을 위한 클라이언트 스텁 변환 규약은 다음과 같다. IDL의 인터페이스는 CORBA.Object.Ref를 루트(root)로 하여 상속된 Ref 레코드 형을 가지는 Ada의 패키지로 변환된다. Ref 형은 객체에 대한 객체 참조(object reference)를 얻고, 다중 상속을 지원하기 위해 사용된다. Ref 형은 오퍼레이션을 서브프로그램으로 변환시킬 때 서브프로그램의 첫 번째 인자가 된다. 두 번째 인자부터는 오퍼레이션에 정의된 인자들의 순서에 따르고, 인자 전달 모드는 in, out과 inout을 그대로 적용시킨다. (그림 4)는 (그림 3)의 IDL 정의를 ReCA IDL 컴파일러를 통하여 Ada95 클라이언트 스텁으로 변환된 명세(specification) 파일이며 (그림 5)는 클라이언트 스텁의 본체(body) 파일이다.

IDL로 작성된 오퍼레이션에서 변환된 Ada95 클라이언트 스텁 코드는 오퍼레이션의 이름, 실인자들과 클라이언트 정보를 포함하여 메시지 블록을 생성하여 객체 구현으로 보내고, 객체 구현으로부터 전달되는 응답을 처리하기 위하여 다음과 같은 과정을 수행한다.

단계 1) 오퍼레이션의 인자들을 인자 리스트로 만든다.

클라이언트의 오퍼레이션 호출을 객체 구현으로 전달하기 위해서는 하나의 메시지 블록을 생성하고 생성된 메시지 블록을 객체 구현으로 전달하여야 한다. 메시지 블록을 생성하는 첫 번째 작업은 오퍼레이션의 호출로 전달되는 실인자들을 메시지 블록에 담기 위하여 하나로 묶어야 한다. 이 과정은 CORBA.NVList.Add_Item( )을 호출하여 인자들을 하나의 인자 리스트로 만든다.

단계 2) 요청을 생성한다.

CORBA.Object.Create_Request( )는 오퍼레이션 이름, 단계 1)에서 생성한 인자 리스트와 클라이언트의

```
-- filesys_directory.ads
with CORBA;
use CORBA;
with ReCA;
use ReCA;

with dir_IDL_FILE;
package filesys.directory is
  type Ref is new CORBA.Object.Ref with null record;
  procedure lookup(Self: in Ref; name : in CORBA.Char; entry : out CORBA.Any;
                   C_type : in ReCA.CallType);
end filesys.directory;
```

(그림 4) 클라이언트 스텁의 명세 파일
(Fig. 4) Specification file for client stub

```
-- filesys_directory.adb
package body filesys.directory is
  procedure lookup(Self: in Ref; name : in CORBA.Char; entry : out CORBA.Any;
                   C_type : in ReCA.CallType) is
  begin
  CORBA.NVList.Add_Item(Arglist, CORBA.To_Unbounded_String("name"),
                                  CORBA.To_Any(name),  CORBA.ARG_IN, Stat);
  CORBA.Object.Create_Request(Cobj,
                          CORBA.To_Unbounded_String("lookup"),
                          Arglist,
                          Result,
                          Dest,
                          Req,
                          CORBA.OUT_LIST_MEMORY,
                          Stat);
  CORBA.Request.Invoke(Req, flag, Stat);
  if CORBA.Request.Get_Returns(Req) = CORBA.ST_EXCEPTION then
    if CORBA.Request.Get_ReqException(Req) = "NO_SUCH_ENTRY" then
      raise dir_idl_file.NO_SUCH_ENTRY;
      else
      raise CORBA.UNKNOWN;
    end if;
  elsif CORBA.Request.Get_Returns(Req) = ST_OK then
    CORBA.NVList.Get_Item(Req.Arg_List, entry, 2);
 end if;
end if;
end filesys.directory;
```

(그림 5) 클라이언트 스텁의 본체 파일
(Fig. 5) Body file for client stub

정보를 이용하여 객체 구현으로 전달될 하나의 요청을 생성하게 된다. 이 때, 클라이언트의 정보는 응답이 호출자(caller)에게 전달되기 위하여 필요한 것으로 호스트 주소, 프로세스 ID, 메시지 ID이다.

단계 3) 요청을 객체 구현에게 전달하고 응답을 받는다.

CORBA.Request.Invoke( )를 호출하여 단계 2)에서 생성된 요청을 객체 구현에게 보내고, 객체 구현이 요청을 수행한 결과인 응답을 분석하게 된다.

단계 4) 예외처리를 한다.

응답을 분석하는 과정의 첫 번째는 예외가 발생했는가를 확인하는 것이다. 객체 구현은 요청의 처리 결과가 예외이면, 예외의 종류를 응답에 담아서 클라이언트에게 전달하는데 클라이언트 스텁은 예외 정보를 분석하여 적합한 예외를 클라이언트에게 발생시키게 된다. 예외 정보는 메시지 블록의 Returns 요소에 예외의 발생 여부와 ReqException 요소에 예외의 종류가 포함되어 있다.

단계 5) 인자값을 재할당한다.

오퍼레이션에 out이나 inout 모드의 인자가 있을 경우, 객체 구현이 요청의 오퍼레이션을 정상적으로 수행하고 결과값인 인자들을 되돌려 주게 된다. 클라이언트 스텁은 응답에서 이러한 인자들을 분석하여 클라이언트에게 전달한다. 요청을 생성할 때 인자들은 NamedValue의 리스트 형태로 생성되었기 때문에 응답 역시 인자들을 리스트의 형태로 전달한다. 응답의 Arg_List에서 인자값을 구해서 변수에 할당한다.

단계 6) 리턴값을 되돌려 준다.

오퍼레이션이 함수일 경우, 함수 수행 후의 결과는 NamedValue형으로 돌려 받게 되며 이를 원하는 형으로 변환하고 변수에 할당하여 클라이언트에게 되돌려 주게 된다.

3.4 구현 골격 변환 규약

구현 골격은 클라이언트로부터 전달되는 요청을 전달받아서 분석한 뒤 실제 오퍼레이션을 호출하고,

```
-- filesys-directory-impl-skeleton.ads --
with Ada;
use Ada;
with CORBA;
use CORBA;
with VOOL;

with filesys.directory.Impl;
with dir_IDL_FILE;
package filesys.directory.Impl.Skeleton is
  procedure sk_lookup(Self: access Object; Req: in out CORBA.Request.Object);
  procedure CORBA_loop(Self: access Object);
end filesys.directory.Impl.Skeleton;
```

(그림 6) 구현 골격의 명세 코드 파일
(Fig. 6) Specification file for implementation skeleton

```
-- filesys-directory-impl-skeleton.adb --
package body filesys.directory.Impl.Skeleton is
  procedure sk_lookup(Self: access Object; Req: in out CORBA.Request.Object) is
    Param: CORBA.NVList.Object;
    Result: CORBA.NamedValue;
    name:     CORBA.Char;
    entry:    CORBA.Any;
  begin
    Param := Req.Arg_List;
    -- Get Argument
    CORBA.NVList.Get_Item(Param, Item, 1);
    name := CORBA.To_char(Item.Argument);
    -- Call Real Method
    lookup(Self, name, entry, Stat);
    if Stat = CORBA.ST_EXCEPTION then
      Req.returns := CORBA.ST_EXCEPTION;
      Req.ReqException := CORBA.To_Unbounded_String("NO_SUCH_ENTRY");
    else
      Req.returns := CORBA.ST_OK;
    end if;

    -- Set Argument
    CORBA.NVList.Get_Item(Param, Item, 2);
    Item.Argument := CORBA.To_any(entry);
    CORBA.NVList.Add_Item(Temp, Item, Stat);

    CORBA.Request.RmcLayer.RmcSendReply(Req);
    if Stat = ST_ERROR then
      Text_IO.Put("Reply Error!");
    end if;
  end sk_lookup;

end filesys.directory.Impl.Skeleton;
```

(그림 7) 구현 골격의 본체 파일
(Fig. 7) Body file for implementation skeleton

오퍼레이션 수행의 결과를 받아 응답을 생성하여 클라이언트에게 전달한다. IDL 컴파일러는 IDL 인터페이스 정의를 목적 언어로 된 구현 골격으로 자동 생성하게 되는데 이 때 변환 규약이 필요하다. 이 규약을 구현 골격 변환 규약이라 한다. 구현 골격 변환 규약은 IDL에서 Ada 언어로의 기본 변환 규약을 토대로 하여 IDL로 작성된 인터페이스 정의를 Ada 언어로 된 구현 골격으로의 변환 규약을 정의하고 있다. (그림 6)은 (그림 3)의 IDL 정의를 ReCA IDL 컴파일러를 통하여 Ada95 구현 골격으로 변환한 명세 파일이며 (그림 7)은 구현 골격의 본체 파일이다.

구현 골격에 존재하는 CORBA_loop 프로시져는 요청을 받아서 적절한 구현 골격의 서브프로그램을 호출하는 역할을 담당한다.

lookup 오퍼레이션의 구현 골격에 해당하는 'sk_lookup' 프로시져는 전달 받은 요청을 분석하여 실제 오퍼레이션을 호출하고, 오퍼레이션의 수행 결과인 응답을 클라이언트에 전달하기 위하여 다음과 같은 과정을 수행한다.

단계 1) 인자 리스트를 분석한다.

구현 골격은 클라이언트로부터 전달된 메시지 블록에서 인자들을 분리한다. 오퍼레이션의 인자들은 NamedValue 형으로 인자 리스트에 담겨서 전달되는데 CORBA.NVList.Get_Item()을 이용하여 인자를 하나씩 얻어 오퍼레이션의 해당 인자 변수에 할당한다.

단계 2) 실제 오퍼레이션을 호출한다.

값을 할당받은 인자 변수들을 이용하여 객체 구현으로 실제 오퍼레이션을 호출하게 된다. 그리고, 객체 구현으로부터 결과를 기다린다.

단계 3) 예외 상황을 응답에 저장한다.

오퍼레이션의 실행 과정에서 예외가 발생했다면, 예외의 발생여부와 예외의 종류를 응답 정보에 추가한다.

단계 4) 인자값을 설정한다.

오퍼레이션 수행이 예외 상황의 발생 없이 정상이었다면, 인자가 out이나 inout 모드인 인자값들을 CORBA.NVList.Add_Item()을 호출하여 인자 리스트로 만들어 응답 정보에 추가한다.

단계 5) 응답 정보에 리턴값을 지정한다.

구현 골격은 수행한 오퍼레이션이 함수라면 리턴값을 응답정보에 담아서 클라이언트에게 전달한다. 객체 구현에서 전달받은 리턴값을 NamedValue형으로 변환하여 응답 정보에 추가한다.

단계 6) 응답 정보를 BOA에게 보낸다.

응답 정보가 담겨 있는 메시지 블록을 CORBA.Request.RmcLayer.RmcSendReply()을 호출하여 BOA 즉, 클라이언트에게 보낸다. RmcSendReply()는 ReCA CORBA 시스템에서 하위 통신 모듈과의 통신을 위하여 정의한 API이다.

4. ReCA IDL 컴파일러의 구현

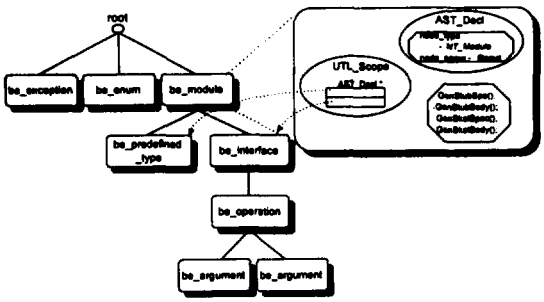
본 장에서는 3장에서 정의한 클라이언트 스텁과 구현 골격 변환 규약을 지원하는 IDL 컴파일러의 구현에 관하여 기술하고자 한다. 구현된 ReCA IDL 컴파일러는 선 마이크로시스템즈(Sun Microsystems)에서 공개한 CORBA IDL 컴파일러 전처리기(Front End)에 클라이언트 스텁과 구현 골격을 생성하는 후처리기(Back End)를 추가하여 제작되었다.

전처리기는 IDL로 정의된 인터페이스를 어휘 분석과 구문 분석 과정을 거쳐서 AST(Abstract Syntax Tree)를 생성한다. 후처리기는 전처리기에서 구성한 AST의 각 노드를 탐색하면서 클라이언트 스텁과 구현 골격 변환 규약을 적용하여 적합한 스텁과 골격 코드를 생성하게 된다.

4.1 컴파일러의 전처리기(Front End)

C++로 작성된 전처리기에서는 IDL 인터페이스 정의를 어휘 분석 과정을 통하여 토큰으로 구분하고, 이를 구문 분석 과정으로 전달한다. 구문 분석 과정에서는 구문 에러가 있는지 검사하고, 없으면 토큰의 형에 해당하는 C++ 클래스의 인스턴스(instance)를 생성한다. 인스턴스화 되는 클래스는 토큰의 형마다 하나씩 존재하는데 이것은 전처리기에서 제공하는 클래스를 상속받아 스텁과 골격 코드들을 생성하기 위한 멤버 함수들을 추가하여 작성된 클래스이다. 최초로 생성된 노드는 루트 노드 밑의 하위노드로 존재하게 되며 중첩된 영역(nested scope)내의 노드들은 상위 노드의 자식 노드로 생성된다. 각 노드는 IDL 인터페이스 정의에 대응되는 형 정보와 형 이름, 자식 노드들에 관한 정보와 스텁과 골격 코드를 생성하기 위한 루틴을 포함하고 있다. (그림 8)은 (그림 3)의

IDL 정의 예제를 IDL 컴파일러의 전처리기로부터 생성된 AST의 구조를 나타낸다.

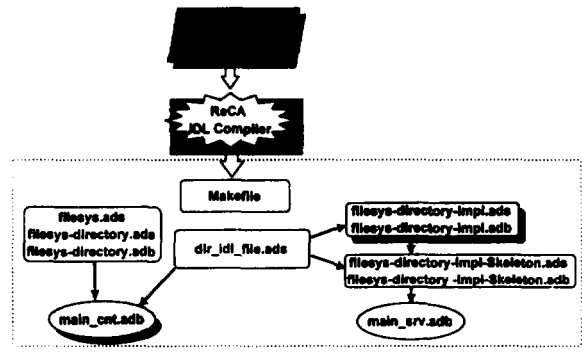


(그림 8) ReCA IDL 컴파일러의 AST 구조
(Fig. 8) AST structure of ReCA IDL compiler

ReCA IDL 컴파일러는 (그림 3)에서 exception 선언과 enum 선언을 만나게 되면 각 형에 해당하는 be_exception과 be_enum 클래스의 인스턴스를 생성하고 루트 노드의 자식 노드로 등록한다. module 선언을 만나서 다시 be_module 클래스의 인스턴스를 루트의 자식 노드로 등록한다. be_module은 노드의 이름인 filesys, 노드의 형인 NT_Module, 코드를 생성할 멤버 함수 그리고 자식 노드로 'IsOpen'형에 해당하는 be_predefined_type과 'directory' 인터페이스에 해당하는 be_interface의 포인터를 갖고 있다. directory 인터페이스 안에 있는 lookup 오퍼레이션은 be_operation 클래스의 인스턴스로 생성되어 be_interface의 자식 노드가 된다. 'lookup' 오퍼레이션에 있는 두 인자는 be_argument 클래스의 인스턴스로 생성되어 be_operation의 자식 노드에 위치한다.

4.2 후처리기(Back End)의 구현

후처리기는 전처리기에서 IDL 정의로부터 생성된 AST의 각 노드를 방문하면서 변환 규약에 적합한 코드를 생성한다. 후처리기는 클라이언트 스텁과 구현 골격 등 10개 이상의 목적 파일들을 생성한다. AST의 각 노드에는 원하는 목적파일을 생성하는 멤버 함수가 존재하게 되는데, 서로 다른 멤버 함수를 호출하여 목적 파일을 생성하게 된다. (그림 9)는 (그림 3)의 IDL 인터페이스 정의 파일인 dir.idl을 ReCA IDL 컴파일러를 통하여 생성한 파일들이며 이 파일들은 클라이언트측 파일과 객체 구현측 파일, 공통 파일로 구분할 수 있다.



(그림 9) ReCA IDL 컴파일러가 생성하는 파일들
(Fig. 9) Created files from ReCA IDL compiler

먼저 클라이언트측 파일로는 클라이언트의 요청을 ReCA ORB를 통하여 객체 구현에게 전달하고 객체 구현으로부터 전달되어온 응답을 클라이언트에게 전달하는 코드가 포함된 클라이언트 스텁 파일(filesys.ads, filesys-directory.ads, filesys-directory.adb)과 프로그래머가 직접 코드를 추가하여 응용프로그램을 작성하기 위한 기본 코드를 담고 있는 클라이언트 파일(main_cnt.adb)로 구성된다. IDL의 모듈과 인터페이스 정의는 각각 하나의 클라이언트 스텁 파일로 생성되는데 filesys 모듈은 오퍼레이션이 존재하지 않으므로 Ada95의 본체 파일은 생성되지 않고 명세 파일(filesys.ads)만 생성된다.

객체 구현측 파일들로는 클라이언트의 요청을 객체 구현에 전달하고 응답을 클라이언트에게 전달하는 코드가 포함된 구현 골격 파일들(filesys-directory-Impl-Skeleton.ads, filesys-directory-Impl-Skeleton.adb)과 ORB와 BOA를 초기화 시키고 클라이언트의 요청을 받아서 구현 골격을 거쳐서 객체 구현에게 전달하는 서버 파일(main_srv.adb)과 실제 요청한 오퍼레이션을 수행하고 결과를 생성하는 코드가 포함된 객체 구현 파일(filesys-directory-Impl.ads, filesys-directory-Impl.adb)로 구성된다. 객체 구현 파일에는 프로그래머가 오퍼레이션을 정의할 수 있도록 기본적인 코드들이 생성

된다. 스텁 파일의 생성처럼 구현 골격 파일도 IDL의 모듈과 인터페이스 정의로부터 생성되는데 모듈에는 오퍼레이션이 존재하지 않으므로 Ada95 명세 파일과 본체 파일은 생성되지 않는다.

공통 파일로는 UNIX 프로그래밍에서 유용한 Makefile 파일과 객체 구현과 클라이언트쪽에 공통으로 포함될 전역 형 선언을 담고 있는 전역 파일(dir_idl_file.ads)이 있다. 전역 파일에는 IDL 모듈과 인터페이스의 외부에 선언된 형 선언, 상수 선언과 예외 선언이 Ada95로 변환되어 저장된다. (그림 10)은 (그림 3)의 IDL 정의가 ReCA IDL 컴파일러를 통하여 생성된 전역 파일이다.

```
-- dir_idl_file.ads --
with CORBA;

package dir_idl_file is
  NO_SUCH_ENTRY : exception;
  type NO_SUCH_ENTRY_Members is new
      CORBA.IDL_Exception_Members with null record;
  function Get_Members(X: Ada.Exceptions.Exception_Occurance)
          return NO_SUCH_ENTRY_Members;
  type Kind is (File, Direct);
end dir_idl_file;
```

(그림 10) ReCA IDL 컴파일러가 생성한 전역 파일
(Fig. 10) Created global file from ReCA IDL compiler

(그림 10)의 전역 파일은 NO_SUCH_ENTRY 예외를 선언하고, 예외 정보를 저장할 수 있는 NO_SUCH_ENTRY_Members 형을 선언하고, 예외가 발생했을 때 정보를 얻어오기 위한 함수인 Get_Members( )를 선언했다. 마지막으로 열거형인 Kind형을 선언하고 있다. 개발자는 클라이언트 파일(main_cnt.adb)과 객체 구현 파일(filesys-directory-Impl-Skeleton.adb)에만 필요한 코드를 추가하여 완전한 ReCA 시스템의 응용프로그램을 개발할 수 있다.

개발된 IDL 컴파일러의 정확성을 검증하기 위하여 IDL 컴파일러로 부터 생성된 클라이언트 스텁을 포함한 클라이언트와 구현 골격을 포함한 객체 구현을 ReCA 시스템에서 실험하였다. 클라이언트와 분산 객체는 gnat 3.10 컴파일러를 이용하여 생성하였으며, Sun OS 4.1.3 상에서 ISIS 2.1 툴킷을 이용하여 동작하는 ReCA ORB 환경에서 (그림 3)의 IDL 예제 파일 뿐만 아니라 다양한 IDL 예제로 요청과 응답이 정확히 전달되는지 검사하였다.

5. 결 론

OMG IDL은 객체 구현에서 제공하는 인터페이스 즉, 오퍼레이션과 형을 기술하는 언어로 특정 프로그래밍 언어에 독립적이다. CORBA IDL 컴파일러는 IDL로 작성된 인터페이스 정의를 클라이언트가 원하는 서비스의 요청을 생성하고 ORB에게 전달하는 클라이언트 스텁과 클라이언트로부터 받은 요청을 객체 구현에게 전달하기 위한 인터페이스인 구현 골격을 생성한다. 생성되는 클라이언트 스텁과 구현 골격은 선언적 언어가 아닌 특정 프로그래밍 언어로 만들어 진다.

본 논문에서는 ReCA 시스템의 IDL 컴파일러를 구현하기 위해 필요한 클라이언트 스텁과 구현 골격 변환 규약을 정의하였으며, 정의한 변환 규약에 따라 ReCA IDL 컴파일러를 구현하였다.

클라이언트 스텁과 구현 골격 변환 규약은 IDL에서 Ada 언어로의 기본 변환 규약을 바탕으로 ReCA 시스템에 맞추어 정의하였다. 요청의 기본 단위로 메시지 블록을 정의하고 이것을 이용하여 요청과 응답의 전달을 가능하게 구현하였다. 클라이언트 스텁은 클라이언트가 호출한 오퍼레이션의 인자와 클라이언트 정보를 메시지 블록에 담아서 요청을 생성하여 객체 구현 쪽에 전달하고 그 응답을 기다리며, 구현 골격에서는 전달되는 요청에서 인자를 분석한 뒤 실제 오퍼레이션을 호출하고 그 결과를 메시지 블록에 묶어서 응답을 클라이언트에게 전달한다. 응답을 받은 클라이언트 스텁은 응답을 분석하여 예외가 발생했다면 예외 처리를 하고 그렇지 않다면 결과값을 클라이언트에게 돌려준다. 본 논문에서 제시된 클라이언트 스텁과 구현 골격 변환 규약의 알고리즘은 다른 CORBA 시스템 개발에 적용 가능할 것이다.

ReCA IDL 컴파일러는 AST를 생성하는 전처리기 부분과 AST를 탐색하면서 실제 코드를 생성하는 후처리기 부분으로 구성된다. 개발된 IDL 컴파일러는 다수의 클라이언트 스텁, 구현 골격 관련 파일 뿐만 아니라 객체 구현, 전역 파일, Makefile, 클라이언트와 서버 관련 파일을 생성한다. 이렇게 스텁과 골격 코드 외에 필요한 파일들을 생성함으로써 개발자의

코드 추가를 최소화하여 신속하게 CORBA 응용프로그램을 개발 할 수 있도록 지원한다.

참 고 문 헌

[1] The MITRE Corporation, "IDL ⇒ Ada Language Mapping Specification", May 1995.

[2] Object Management Group, "The Common Object Request Broker Architecture and Specification: Revision 2.0", OMG Document, July 1995.

[3] Landis, S., "Building Reliable Distributed Systems with CORBA", 1995.

[4] Maffeis, S. "Run-Time Support for Object-Oriented Distributed Programming", PhD thesis, University of Zurich, Department of Computer Science, 1995.

[5] Barnes, J. G. P, "Programming in Ada", ADDISON_WESLEY, 1993.

[6] "Ada95 Rationale", Intermetrics Inc, Jan. 1995.

[7] Van Renese, R., Birman, K. P., "Fault-Tolerant programming using Process Groups, In Distributed Open Systems", F. Brazier and D. Johansen, Eds., IEEE Computer Society Press, 1994.

[8] 유양우, 홍창열, 김영곤, 박양수, 이명준, "신뢰성 있는 CORBA 시스템 구현을 위한 객체지향적인 기반 계층에 관한 연구", 한국정보과학회 96 춘계 학술발표 논문집 제23권 1호, 1996. 4.

[9] 홍창열, 김영곤, 유양우, 박성진, 이명준, "Ada95의 protected object를 이용한 다중 쓰레드 CORBA 서버의 구현에 관한 연구", 한국정보과학회 96 춘계 학술발표 논문집 제23권 1호, 1996. 4.

[10] 유양우, 김영곤, 박성진, 박양수, 이명준,, "Ada95를 이용한 CORBA 분산객체시스템의 구현을 위한 통신계층에 관한 연구", 한국정보과학회 96 추계 학술발표논문집 제23권 2호, 1996. 10.

[11] 홍창열, 김영곤, 이동헌, 박성진, 이명준, "Ada95의 병행성 모델을 이용한 CORBA 기본 객체 어댑터", 한국정보과학회 96 추계 학술발표논문집 제23권 2호 1996. 10.

[12] Zhonghua Yang, Keith Duddy, Distributed Object Computing with CORBA, DSTC Technical Report 23, June 1995.

[13] Steve Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogenerous Environments", IONA Technologies Inc., 1996.

[14] Thomas J. Brando, "Comparing DCE and CORBA", The MITRE Corporation, March 1995.

[15] George Couloursi, Jean Dollimore, Tim Kindberg, "Distributed Systems: Concepts and Design, 2nd Edition", ADDISON_WESLEY, 1994.

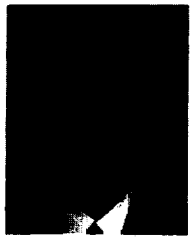
[16] Robert Orfali, Dan Harkey, Jeri Zahavi, "The Essential Distributed Objects Survival Guide", John Wiley & Sons Inc., 1995.

[17] Michael A. Smith, "Object-Oriented Sofeware in Ada95", Thomson, 1996.

[18] Douglas C. Schmidt, Steve Vinoski, "Object Interconnections, Column 1-8", C++ Report magazine, 1995-1996.

[19] Fledman, Koffman, "Ada95-Programming Solving and Program Design, 2nd Edition", Addsion Wesley, 1996.

[20] Jon Siegel, "CORBA Fundamentals and Programming", John Wiley & Sons Inc., 1996.



박 성 진

1996년 2월 울산대학교 전자계산학과 졸업(학사)

1996년 3월~현재 울산대학교 전자계산학과 석사과정

관심분야: 분산객체 시스템, 프로그래밍 언어, 네트워크 프로그래밍 등.

김 영 곤

1994년 2월 울산대학교 전자계산학과 졸업(학사)

1996년 2월 울산대학교 전자계산학과 졸업(석사)

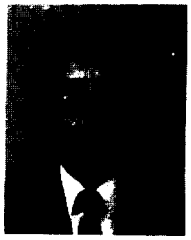
1998년 2월 울산대학교 전자계산학과 박사과정 수료

1997년 12월~현재 (주)퓨처시스템 정보통신연구소 근무

관심분야: 프로그래밍 언어 시스템, 병행 프로그래밍, 분산객체 프로그래밍 등.

이 동 헌

1996년 2월 울산대학교 전자계산학과 졸업(학사)
1996년~현재 울산대학교 전자계산학과 석사과정
관심분야:프로그래밍 언어, 분산 객체 프로그래밍, 정보통신 등.



박 양 수

1978년 2월 울산대학교 전자계산학과 졸업(학사)
1981년 2월 서울대학교 계산통계학과 졸업(석사)
1986년 3월~현재 서울대학교 계산통계학과 박사과정
1980년 3월~현재 울산대학교 전자계산학과 근무(현재 부교수)
관심분야:분산처리, 컴퓨터알고리즘 등.

이 명 준

1980년 2월 서울대학교 수학과 졸업(학사)
1982년 2월 한국과학기술원 전산학과 졸업(석사)
1991년 8월 한국과학기술원 전산학과 졸업(박사)
1982년 3월~현재 울산대학교 전자계산학과 근무(현재 교수)
1993년 8월~1994년 7월 미국 버지니아대학 교환교수
관심분야:프로그래밍언어, 분산 객체 프로그래밍 시스템, 병행 실시간 컴퓨팅, 인터넷 프로그래밍시스템 등.