C++ IDL 컴파일러 구현

박찬모* · 이 준*

Implementation of C++ IDL Compiler

Chan-Mo Park* . Joon Lee*

이 논문은 1999년도 조선대학교 교수해외파견 연구비의 지원을 받아 연구되었음

요 약

본 논문에서는 IDL 정의를 입력 받아 파싱하는 컴파일러의 전반부를 위해 OMG IDL 컴파일러를 사용하였다. 또한 ORB를 위해 omniORB3[1]를 사용했다. OMG IDL CFE는 IDL 정의를 입력 받아 어휘 및 구문 분석을 한 후 AST 트리를 생성하며, 생성된 각 노드는 우리가 새로 추가한 BE_* 클래스의 인스턴스로 구성된다. IDL 컴파일러의 후반부는 AST의 각 노드를 반복자인 UTL_ScopeActiveIterator 클래스를 사용하여 반복적으로 각 순회하면서 해당하는 출력을 덤프한다. 이때 두개의 출력 파일을 생성토록 했다. 모든 코드 생성은 BE_produce.cc에서 시작되며, idl_global->root() 노드를 시작으로 하여 각 클래스에 해당 코드를 생성하는 dump* 함수를 호출하여 생성했다. 본 논문은 IDL 정의를 C++언어로 맵핑만을 실험했으며, 이것은 omniORB3에서 제공하는 IDL 컴파일러와 동일한 결과를 생성했으며, omniORB3[1] 환경에서 동작하는 변환된 C++ 코드임을 실험했다. 향후 IDL 컴파일러를 통한 성능 향상을 위해 마샬링 코드의 최적화를 할 수 있도록 하는 코드를 생성하는 문제에 관심을 가지고 있다.

ABSTRACT

In this paper, OMG IDL CFE, provided by SunSoft, is used to take a IDL definitions as inputs and parse those. OmniORB3 is introduced to support functionality of the ORB. Suns CFE produce AST after parsing inputs. Actually, the node of AST is instances of classes which are derived from CFE classes. As the compiler back end visit the node of the AST using iterator class, UTL_ScopeActiveIterator, it dumps codes of output. During processing, two files are generated. Routines of generating code are invoked by BE_produce.cc and codes are produced while visiting root of AST, idl_global->root(). The dump* functions which dump codes is called according to the type of node.

In this paper, Mapping C++ of IDL definition is experimented and results in the same as that of omniidl which is provided by omniORB. The code of results behavior correctly on omniORB3. In the future, we are interested in optimizing the performance of marshalling code via IDL compiler.

키워드

CORBA, IDL, Compiler, C++

*조선대학교

접수일자 : 2001. 8. 30

I. 서론

어플리케이션 개발에 대한 요구들은 더 이상 중앙 집중식으로서는 빠른 시간내에 개발해서, 유지하는 것이 어렵게 되고 있다. 그러므로 어플리케이션은 유지가 용이하고, 개발하기에 적당한 정도로 분할되어 네트워크상에 분산되어 개발될 필요가 있다. 이것은 현재 계속 증가하는 네트워크의 속도와 함께 중요한 이슈가 되고 있다.

하지만 분산 어플리케이션은 개발에 몇 가지 문제점을 갖고 있다. 개발자의 능력에 의지하여 개발된 어플리케이션의 컴포넌트들이 서로 다른 언어를 사용하므로 이질적으로 되며 하드웨어 종속성을 갖게 된다. 따라서 이러한 문제점들을 해결하기 위한 방안으로 미들웨어가 필요하다. CORBA(Common Object Request Broker Architecture) 또한 이들 미들웨어의 하나이다.

본 논문은 미들웨어인 CORBA상에서 IDL(Interface Definition Language)을 사용한 인터페이스 정의를 실제 개발자들이 이용하는 C++언어로 변환하는 IDL 컴파일러를 구현한다. 본 논문의 구성은 CORBA와 IDL 컴파일러에 대해 간략히 살펴보고, IDL 컴파일러를 구현하는 데 필요한 맵핑 규칙을 설명하고, IDL 컴파일러를 구현하는 방법을 설명한다. 마지막으로 결론 및 향후 연구 과제에 대해 논의한다.

II. CORBA

OMG의 CORBA는 분산 환경상에 존재하는 객체간의 상호 운용성을 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 스터브나 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[4][5].

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[4][5].

• 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.[4][5]

• 클라이언트 스터브와 구현 골격

클라이언트 스터브는 클라이언트가 정적으로 요청을 생성하고 전달하는 메커니즘이며 구현 골격은 정적 요청을 객체 구현으로 전달하는 메커니즘이다. IDL 컴파일러가 OMG IDL 인터페이스 정의를 이용하여 클라이언트 스터브와 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. [4][5]

III. IDL 컴파일러

CORBA 시스템에서 응용프로그램 개발단계는 다음과 같다.

단계 1. IDL로 객체의 인터페이스를 정의한다. IDL은 클라이언트 객체가 호출하고 객체 구현에서 제공하는 인터페이스를 기술하는 데 사용하는 언어이다. IDL은 객체의 인터페이스, 오퍼레이션, 형을 정의하는데 특정 프로그램 언어에 독립적이며, 캡슐화, 인터페이스 다중 상속, 객체지향적 예외 처리 등의 객체지향 개념을 지원하며, 구현을 위한 언어가 아니라 인터페이스를 기술하기 위한 선언적 언어이다. IDL은 분산 환경에서 특정 프로그래밍 언어에 독립적으로 CORBA 응용프로그램을 작성할 수 있게 해준

다. 바꾸어 말하면 일단 IDL로 작성된 인터페이스 정의는 CORBA 시스템에서 제공하는 IDL 컴파일러를 통해 원하는 프로그래밍 언어에 적합하게 변환된다.[4][5]

단계 2. 클라이언트 스터브와 구현 골격을 생성한다. IDL 컴파일러는 IDL로 정의한 객체 인터페이스로부터 프로그래밍 언어로 된 클라이언트 스터브와 구현 골격을 생성한다. 클라이언트 스터브와 구현 골격은 서로 다른 언어로 작성될 수 도 있다. 클라이언트 스터브는 IDL로 정의한 오퍼레이션이 클라이언트가 호출할 수 있는 특정 프로그래밍 언어로 변환된 인터페이스이다. 클라이언트 스터브는 각 인터페이스에 대한 요청을 생성하는데 특히, 인터페이스 호출에 필요한 메서드 선언을 제외한 나머지 부분을 내부적으로 감추며, 특정 ORB에 대하여 최적화된 상태로 만들어진다. 구현 골격은 객체 어댑터(Object Adapter)에 의존적으로 작성된 객체의 메서드에 대한 인터페이스이다. 이 인터페이스를 호출하여 ORB는 요청을 객체 구현에 전달한다. 이 인터페이스를 업-콜(up-call) 인터페이스라 한다.[4][5]

단계 3. 개발자가 적절한 코드를 추가하여 완전한 응용프로그램을 개발한다.

단계 4. IDL 컴파일러를 통하여 생성된 코드에 개발자가 적절한 코드를 추가하여 완전한 CORBA 응용프로그램을 작성하게 된다.

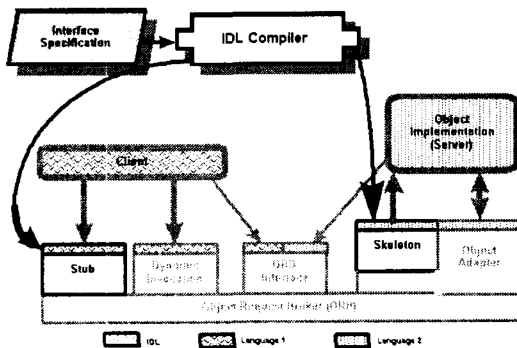

그림 1. IDL 컴파일러의 역할
Fig.1 Role of IDL Compiler

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

IDL 컴파일러는 특정 CORBA 시스템을 위해 정의된 클라이언트 스터브와 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 스터브와 구현 골격 변환 규약을 확정하여야 한다. 이 규약은 CORBA 시스템의 ORB 구현에 따라 달라지게 되는데 IDL 컴파일러는 정의한 변환 규약에 따라 IDL로 정의된 인터페이스를 특정 프로그래밍 언어로 작성된 클라이언트 스터브와 구현 골격으로 생성하게 된다.[4][5]

Ⅳ. 변환 규칙 및 구현

본 논문은 IDL 정의를 입력받아 어플리케이션을 만드는 데 많이 사용되는 C++ 언어로 변환하는 IDL 컴파일러를 제안한다. 본 논문에서 사용된 환경은 SunSoft에서 제공하는 IDL CFE(INTERFACE DEFINITION LANGUAGE COMPILER FRONT END)와 ORB로 AT&T의 omniORB3를 사용한다.[1][7] 우리는 제안하는 컴파일러를 두 단계로 구성한다. 첫 번째 단계는 SunSoft가 구현하여 제공하는 IDL CFE를 사용하여 IDL 정의를 입력받아 파싱을 한 후, AST(Abstract Syntax Tree)를 생성한다. 이 단계에서 IDL 정의의 어휘 및 구문 분석을 한다. 만약 에러가 발생하면 두 번째 단계는 수행되지 않는다. 두 번째 단계는 AST를 입력 받아 omniORB3에서 동작하는 C++ 언어로 된 파일들을 생성한다.[2][6] 먼저 우리는 OMG 표준안의 IDL 정의를 C++로 맵핑하는 규칙에 대해 논하고, 이에 따라 컴파일러를 구현하는 방법을 논한다.[3]

4.1 C++ Mapping 규칙

Module

IDL에서 module은 데이터 형과 인터페이스를 독립된 이름 공간으로 그룹화한다. 이것은 C++의 namespace나 C++의 이름 공간 구분자인 "::"으로 모든 하위 선언들을 변경하는 방법으로 맵핑이 된다.

namespace 사용시 발생하는 문제를 피하기 위해 표준 C++ 이름 공간 구분자인 "::" 을 사용한다.[3]

기본 데이터 형들은 C++의 데이터 형으로 직접 맵핑한다. 표 1에서 C++ out 형들은 interface의 out 인자로 사용되는 경우 필요한 데이터 형들이다. 이들은 모두 CORBA 모듈에 정의되어 있다.

표 1. 기본 데이터형의 맵핑
Table. 1 Mapping of Basic Types

OMG IDL	C++	C++ Out Type
short	CORBA::Short	CORBA::Short_out
long	CORBA::Long	CORBA::Long_out
long long	CORBA::LongLong	CORBA::LongLong_out
unsigned short	CORBA::UShort	CORBA::UShort_out
unsigned long	CORBA::ULong	CORBA::ULong_out
unsigned long long	CORBA::ULongLong	CORBA::ULongLong_out
float	CORBA::Float	CORBA::Float_out
double	CORBA::Double	CORBA::Double_out
long double	CORBA::LongDouble	CORBA::LongDouble_out
char	CORBA::Char	CORBA::Char_out
wchar	CORBA::WChar	CORBA::WChar_out
Boolean	CORBA::Boolean	CORBA::Boolean_out
octet	CORBA::Octet	CORBA::Octet_out

String

OMG IDL 스트링 형은 크기가 정해져 있는 것과는 무관하게 C++ char*로 맵핑한다.[3]

Structure

IDL의 struct 형은 C++ struct로, 각각의 멤버들은 대응하는 데이터 형으로 변환한다. 이것은 필드 액세스를 간단하게 하고 대부분의 고정 길이 struct들의 초기화를 한번에 할 수 있게 하는 장점이 있다. 이러한 초기화가 가능하도록 하기 위해서, C++ struct는 사용자 정의 생성자, 대입 연산자, 혹은 소멸자를 지정할 필요가 없으며 각 struct 멤버들은 C++ 변환 규칙에 따라 관리하도록 변환한다.[3]

Sequence

IDL sequence는 현재 길이와 최대 길이를 가진 배열과 같은 방법으로 처리되는 C++ 클래스로 변환한다. 고정 길이 sequence인 경우, 최대 길이는 sequence에서 얻을 수 있으며, 최대 길이의 초기값은 sequence의 생성자가 초기 버퍼 할당 크기를 조절할 수 있게 할 수 있다. 고정 길이가 아닌 sequence의 경우 현재 길이보다 큰 값으로 길이를 설정하므로 sequence 데이터는 재배치되어야 한다. 재배치는 필요한 크기의 새로운 sequence를 만드는 것과 같다.[3]

Union

IDL union 선언은 멤버를 액세스하는 함수와 구분자를 가지는 C++ 클래스로 변환한다. 일부 멤버 함수는 멤버에 대한 접근만을 제공하거나 쓰기만을 제공할 수도 있다. Union의 기본 생성자는 구분자나 멤버들을 어플리케이션에 필요한 상태로 초기화를 하지 않는다.[3]

Array

IDL array 선언은 array를 사용해서 정적으로 초기화된 데이터의 정의가 가능한 C++ array로 변환한다. 만약 배열 요소가 string이나 객체 참조인 경우, 구조체 멤버와 마찬가지 타입을 사용하여 변환한다.[3]

Exception

IDL exception은 CORBA 모듈에 정의된 UserException을 상속하는 C++ 클래스로 변환한다. exception이 가변 길이 멤버를 가지지 않더라도 가변 길이 구조체와 유사한 클래스를 생성한다. 가변 길이 구조체와 마찬가지로, 각 예외 멤버는 변환 규칙에 따라 변환되어 관리하게 한다.[3]

Object Reference

CORBA 환경에서 클라이언트는 객체 참조(Object Reference)라고 하는 ORB 객체를 사용한다. 객체 참조는 CORBA 객체의 속성과 오퍼레이션으로 사상시켜주는 메서드를 가진 C++ 객체이다. 클라이언트는 다른 C++ 객체를 사용하는 것과 동일하게 객체 참조를 사용할 수 있다. 하지만 객체 참조는 연관된 ORB 객체에 요청을 전달하는 기능을 가진다. 각 IDL에 정의된 각 interface 선언은 동일한 이름으로 된 객체 참조에 대한 C++ 클래스를 생성한다. 이 클래스의 인스턴스는 서버 객체로 보내는 모든 정보를 유지한다. 객체 참조와 구현 클래스는 ineterface에 정의된 각 오퍼레이션마다 하나의 메서드를 갖는다. 메서드의 이름은 IDL에서 주어진 이름과 같다. 메서드의 파라메터들도 C++로 변환된 형을 가지며 이름은 동일하다.[3]

Typedef

IDL에 정의된 typedef는 데이터 형에 대한 병칭을

지정하는 것으로 C++의 typedef로 변환한다. 데이터형에 별칭을 부여하고자 하는 경우, 몇 개의 C++ 데이터형으로 구성된 경우 각 타입에 대한 별칭을 지정하도록 변환한다.[3]

Interface

IDL에 정의된 interface는 데이터 형, 상수, 오퍼레이션들, 인터페이스에 정의된 예외를 가진 C++ 클래스로 변환하다. 변환된 C++ 클래스는 포인터나 참조 연산자를 사용하거나 직접 인터페이스 클래스를 인스턴스화할 수 없다. 이것은 객체 참조를 사용하여 접근할 수 있으므로 객체 참조에 대한 클래스를 생성한다. 또한 객체 참조를 사용하는 경우 모든 메모리 관리를 사용자가 해주어야 하므로 에러가 발생하기 쉽고 메모리 부족을 야기할 수 있으므로 이것을 자동화해 주기 위한 객체 참조 변수 형 클래스를 추가로 생성한다.[3]

4.2 컴파일러 구현

코드 생성은 CFE IDL에서 생성한 AST를 입력으로 받아 트리의 각 노드를 운행하면서 필요한 코드를 생성하고 이 코드로 구성되는 각각의 파일을 생성한다. 우리는 구현 단계를 설명하기 간단한 IDL 정의로 그림 2를 사용한다.

```
interface Intf1 {
string op1(in string arg1);
void op2(in string srg2, out string arg3);
};
```

그림 2. example.idl
Fig. 2 example.idl

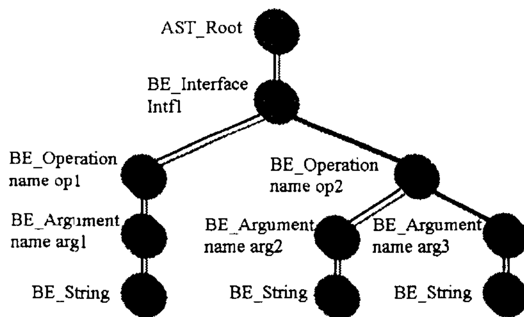

그림 3. AST 트리
Fig. 3 AST Tree

그림 2에 제시된 IDL 정의는 그림 3과 같은 AST를 생성한다.

그림 3의 각 노드들은 IDL CFE의 전반부에서 사용되는 클래스들을 상속한 클래스의 인스턴스들이다. 예를 들어, 클래스 BE_Interface는 AST_Interface를 상속한 것이다. 컴파일러 후반부를 구성하는 주요 클래스들은 다음과 같다.

- 오퍼레이션의 인자를 나타내고 처리하는 BE_argument 클래스
- 상수를 나타내고 처리하는 BE_constant 클래스
- 나열형을 나타내고 관리하는 BE_enum 클래스
- 예외 정의를 나타내고 관리하는 BE_e xception 클래스
- 인터페이스를 나타내고 관리하는 BE_int erface 클래스
- 모듈을 나타내고 관리하는 BE_module 클래스
- 오퍼레이션을 나타내고 관리하는 BE_oper ation 클래스
- 시퀀스를 나타내고 관리하는 BE_sequence 클래스
- 구조체를 나타내고 관리하는 BE_structure 클래스
- 타입 정의를 나타내고 관리하는 BE_type def 클래스
- 몇 개 클래스에 범용적인 유틸리티 함수들

코드 생성은 함수 BE_produce()를 통해 이루어진다. 이 함수는 그림 4와 같은 파일을 생성한다. 이 파일은 omniORB를 이용하기 위해 기존의 omnidl 컴파일러의 결과와 동일하게 생성토록 구성한다.[8]

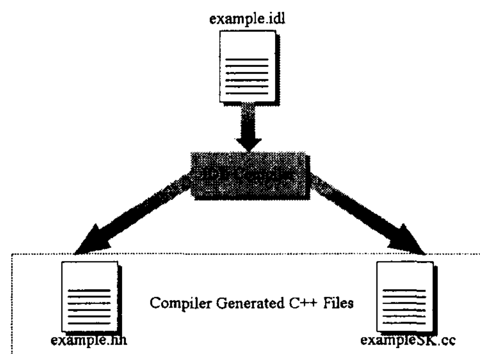

그림 4. 생성 파일
Fig. 4 Generated Files

example.hh 파일은 각 인터페이스에 대한 맵핑을 정의한다. 이 파일은 클라이언트 스터브와 서버 구현 클래스와 관련된 내용을 갖는다. 이 파일은 다음과 같은 과정을 통해 생성한다.

1. IDL 입력의 파일 이름을 사용하여 example.hh 파일을 생성한다.
2. 입력된 파일이름을 사용하여 omniORB를 사용하기 위해 "#include <omniORB3/ CORBA.h>"를 포함하도록 헤더를 생성 한다.[1]
3. idl_global->root()를 통해 최상위 노드인 AST_Root에서 시작하여 각각의 노드를 트래버설하여 모듈, 시퀀스, 열거형, 예외등등의 노드가 발견되면 이에 대한 선언을 추가하고 각 인터페이스 정의에 대해 매크로 정의, 포워딩, 예외, 그리고 typedef등등에 대한 선언을 생성한다.
4. 인터페이스 노드가 발견되면 해당하는 인터페이스의 Helper 클래스를 생성한다.
5. 인터페이스 객체 참조 변수 및 인터페이스의 out 인자로 사용되는 클래스를 정의한다.
6. 인터페이스에 대한 클래스를 정의한다.
7. 인터페이스 참조 클래스를 정의한다.
8. 인터페이스의 프락시 클래스를 정의한다.
9. 인터페이스의 서버측 구현 클래스를 정의한다.
10. POA_Interface_name 클래스를 정의한다.
11. 인터페이스 정의 클래스에서 사용하는 마샬링 코드를 정의한다.

위와 같은 과정을 통하여 생성되는 example.hh 파일은 그림 5와 같다.

```
#include <omniORB3/CORBA.h>
#ifndef __Intf1__
#define __Intf1__
class Intf1;
class _objref_Intf1;
class _impl_Intf1;
typedef _objref_Intf1* Intf1_ptr;
typedef Intf1_ptr Intf1Ref;
class Intf1_Helper {
..
};
typedef
_CORBA_ObjRef_Var<_objref_Intf1,
Intf1_Helper> Intf1_var;
typedef
 _CORBA_ObjRef_OUT_arg<_objref_Intf1,
Intf1_Helper > Intf1_out;
#endif
class Intf1 {
..
};
class _objref_Intf1 :
   public virtual CORBA::Object, public
   virtual omniObjRef
{
public:
  char* op1(const char* arg1);
   void op2(const char* srg2,
            CORBA::String_out arg3);
        ..
};
class _pof_Intf1
   : public proxyObjectFactory {
        ...
};
class _impl_Intf1 :
  public virtual omniServant
{
        ..
};
```

그림 5. example.hh
Fig. 5 example.hh

위와 같은 내용의 파일은 파일을 관리하는 함수인 BE_produce_file()함수를 사용하여 열고, 해당 노드에 있는 메서드 dumpIncludeFile()를 사용하여 각 내용을 출력토록 한다.

exampleSK.cc 파일은 각 인터페이스 맵핑에 대해 정의된 구현 코드를 갖는다. 이 파일은 다음과 같은 과정을 통해 생성한다.

1. IDL 컴파일러의 전반부에서 입력된 파일 이름을

사용하여 헤더 파일인 example.hh를 포함시킨다.

2. Intf1_Helper 클래스의 구현 부분을 정의한다. 이것은 BE_interface 클래스의 dumpHelper()에 의해서 이루어진다.
3. 인터페이스에 대한 객체에 대한 구현을 정의한다. 이것은 dumpInterfaceCode()를 사용하여 코드를 정의한다.
4. 인터페이스 객체 참조 클래스에 대한 구현 코드를 정의한다. DumpInterfaceRefCode()를 사용한다.
5. 클라이언트 프락시 클래스의 구현을 정의한다. 이것은 dumpProxy()를 사용하여 생성한다.
6. 서버 구현에 사용되는 클래스인 구현 클래스에 대한 코드를 정의한다. 이것은 dumpImplCode() 함수를 사용하여 생성한다.

V. 결론

본 논문에서는 IDL 정의를 입력받아 파싱하는 컴파일러의 전반부를 위해 OMG IDL 컴파일러를 사용하였다. 또한 ORB를 위해 omniORB3를 사용했다. OMG IDL CFE는 IDL 정의를 입력 받아 어휘 및 구문 분석을 한 후 AST 트리를 생성하며, 생성된 각 노드는 우리가 새로 추가한 BE_* 클래스의 인스턴스로 구성된다. IDL 컴파일러의 후반부는 AST의 각 노드를 반복자인 UTL_ScopeActiveIterator 클래스를 사용하여 반복적으로 각 순회하면서 해당하는 출력을 덤프한다. 이때 두 개의 출력 파일을 생성토록 했다. 모든 코드 생성은 BE_produce.cc에서 시작되며, idl_globa ->root() 노드를 시작으로 하여 각 클래스에 해당 코드를 생성하는 dump* 함수를 호출하여 생성했다.

본 논문은 IDL 정의를 C++언어로 맵핑을 실험했으며, 이것은 omniORB3에서 제공하는 IDL 컴파일러와 동일한 결과를 생성했으며, omniORB3 환경에서 동작하는 변환된 C++ 코드임을 실험했다. 향후 IDL 컴파일러를 통한 성능 향상을 위해 마샬링 코드의 최적화를 할 수 있도록 하는 코드 생성이 연구되어야 한다.

참고문헌

[1] Sai-Lai Lo, David Riddoch, Duncan Grisby, The omniORB Version 3.0 Users Guide, AT&T Laboratories Cambridge, 2000

[2] DIMMA Team, DIMMA Design and Implementation, APM Ltd, 1997

[3] OMG, C++ Language Mapping Specification, OMG, 1999

[4] OMG, The Common Object Request Broker : Architecture and Specification, OMG, 1999

[5] 박성진, 이동헌, 김영곤, 박양수, 이명준, ReCA CORBA 시스템을 위한 IDL 컴파일러, 한국정보처리학회 논문지, 제5권, 2호, p.437-449, 1998

[6] Nigel Edwards, A Stub Compiler for CGI and HTTP: The Programmer's Guide, APM Ltd, 1995

[7] SunSoft, WRITING_A_BE(OMG_IDL_CFE /doc), SunSoft, 1994

[8] Duncan Grisby, omniidl The omniORB IDL Compiler, AT&T Laboratories Cambridge, 2000

박찬모(Chan-Mo Park)

1995년 조선대학교 컴퓨터공학과(공학사)

1997년 2월 조선대학교 컴퓨터공학과(공학석사)

1997년 3월～현재 조선대학교 컴퓨터공학과(박사과정)

관심분야: 분산처리, 컴파일러 및 운영체제

이 준(Joon Lee)

1979 조선대학교 전자공학과(공학사)

1981 조선대학교 전자공학과(공학석사)

1997년 숭실대학교 전자계산학과(공학박사)

1982년～현재 조선대학교 컴퓨터공학부 교수

관심분야: 분산 운영체제, 병렬 처리, 프로그래밍 환경