CFE를 사용한 IDL 중간 표현 생성

박찬모*, 송기범*, 홍성표*, 이 혁*, 이정기*, 이 준**
*조선대학교 대학원 컴퓨터공학과
**조선대학교 컴퓨터공학과

Generating Intermediate Representation of IDL Using the CFE

Chanmo Park*, Gibeom Song*, Seongpyo Hong*, Hyok Lee*, Jeongki Lee*, Joon Lee**
* Dept. of Computer Eng., Chosun Univ.
** Dept. of Computer Eng., Graduate School, Chosun Univ.

요 약

분산 프로그램을 작성하는 프로그래머들은 시스템 통신 코드를 작성할 때 딜레마에 빠지게 된다. 코드를 직접 혹은 부분적으로 직접 작성하면 응용 프로그램의 속도는 최대화될 수 있지만, 응용 프로그램을 구현하고 유지하는데 많은 노력이 필요하게 된다. 반면에 코드를 CORBA IDL 컴파일러를 사용하여 생성하는 경우 프로그래머의 노력은 감소되지만 응용 프로그램의 수행성은 나빠진다. 그래서 우리는 CORBA IDL 컴파일러에 의해 생성된 코드를 최적화할 필요가 있다.

우리는 전형적인 프로그래밍 언어에서 사용되고 있는 기법들을 IDL 컴파일을 하는데 도입한다. 우리는 IDL 컴파일의 단계를 세단계로 분리한다. 첫 번째 단계는 전단계로 IDL의 파싱 및 스코프 관리와 AST 생성을 한다. 두 번째 단계는 최적화 단계를 구현한다. 세 번째는 이것을 타켓 언어의 코드로 생성하는 단계이다. 본 논문은 첫 번째 단계에 초점을 맞춘다. 우리는 이 단계에서 AST에서 인터페이스와 메시지 표현을 분리하여 표현한다. 이것은 최적화 단계에서 분리된 최적화를 지원한다.

ABSTRACT

Programmers who write distributed programs are faced with a dilemma when writing the systems communication code. If the code is written by hand or partly by hand, then the speed of the application may be maximized, but the human effort required to implement and maintain the system is greatly increased. On the other hand, if the code is generated using a CORBA IDL compiler then the programmer effort will be reduced, but the performance of the application may be poor. So we need the optimizing the code generated by CORBA IDL compiler.

We introduce the techniques which have been used by typical programming languages into compilation of IDL. We separate the phase of compilation into three phase. The first phase parses interface definition in IDL, manages nested scope and generates AST(Abstract Syntax Tree). The second phase implements the optimization. The third phase generates the code in target language. In this paper, we focus on the first phase. We separate interface definition into interface and message representation from AST. This supports the separate optimization of code in second phase.

I. 서론

CORBA의 IDL은 인터페이스를 정의하기 위한 언어로 여러 컴파일러들이 사용되고 있다. 하지만 이러한 컴파일러들은 IDL 정의를 파싱후 스켈레톤과 스터브 루틴을 생성하게 되므로 최적화 및 다양한 트랜스포트 메카니즘 지원을 어렵게 한다.

본 논문에서는 IDL의 컴파일을 전반부와 후반부로 나누고 이들 사이에 중간 표현 생성 부분을 도입함으로써 기존 프로그래밍 언어의 컴파일러에서 사용되는 다양한 최적화 기법들을 지원할 수 있게 한다.

본 논문의 구성은 2장과 3장에서 CORBA와 IDL 컴파일러에 대해 고찰하고 4장과 5장에서 구현에 필요한 사항들을 기술한다. 그리고 마지막 부분인 결론으로 구성한다.

II. CORBA

2.1 CORBA 개요

OMG(Object Management Group)에서 제안한 CORBA(Common Object Request Broker Architecture)는 이종의 분산된 환경하에서 응용 프로그램들을 서로 통합할 수 있는 개방 분산 객체 시스템을 위한 표준이며[2] 또한 OMA(Object Management Architecture)는 네트워크 상에 분산된 객체들의 전송을 위해 CORBA라는 객체 버스(Object Bus)를 제공한다. 즉, OMA는 응용 객체(Application Objects), 일반 기능(Common Facilities), 객체 서비스(Object Services)와 가장 중심적인 부분으로서 이런 객체들의 버스 역할을 하는 ORB(Object Request Broker)를 포함하고 있는 CORBA로 이루어져 있다.

CORBA 2.0 규정 이후로 인터넷 환경에서 ORB 간의 통신에 대한 표준인 GIOP(General Inter-ORB Protocol)가 정하여졌고 이 가운데서도 TCP/IP를 기반으로 하는 규약에 대하여는 IIOP(Internet inter-ORB Protocol)라고 명명하고 필수적으로 구현하도록 요구하고 있다. 이를 통하여 기존 CORBA 제품들 간의 상호 작동이 가능하고 인터넷 환경에서는 ORB를 이용하여 보다 폭넓은 분산 객체에 대한 지원이 가능하다. [1]

CORBA는 분산 객체들 사이의 통신을 위해 기존의 프로그래밍 언어와는 독립적인 IDL (Interface Definition Language)을 정의하여 이용함으로써 클라이언트에서의 객체 구현(Object Implementation)이 다양한 프로그래밍 언어가 가능하도록 지원하고 있다. 특히 새로운 CORBA2.0 규정에서는 자바 언어에 대한 IDL 매핑(Mapping) 표준안이 제정됨으로써 자바언어를 이용한 CORBA분산 객체 환경의 구현이 가능하여졌다. 이는 자바언어의 플랫폼 독립적인 특성과 CORBA의 분산객체에 대한 자원을 통합할 수 있게 하여 이질적 시스템이 함께 존재하는 분산 네트워크 환경에서 보다 생산성 높은 응용 시스템 구축을 위한 해결방안이 가능해진 것이다.

2.2 CORBA 장점

CORBA의 장점은 주요 미들웨어 업체들의 지원, 개방형 표준이라는 점이다.

CORBA를 사용할 때 시스템 구현 장점

◉네트웍 프로그램을 할 필요가 없으므로 개발 시간이 단축된다.

◉객체라는 사용자 중심 모듈을 사용하므로 프로그램을 작성하기 쉽고, 사용자가 시스템을 이해하기 쉽다.

◉모듈이 언제든지 확장, 수정 가능하므로 유지 보수가 쉽다.

◉모듈을 다시 결합하여 새로운 시스템 환경을 구성할 수 있다.

◉기존 시스템을 수정하지 않고, 기존 모든 자원을 재사용 할 수 있다.

2.3 CORBA 시스템

OMG의 CORBA는 분산 환경상에 존재하는 객체간의 상호 운영성을 지원하며 요청과 응답의 투명성을 제공한다. 그림.1에 나타낸 것과 같이 클라이언트는 서버의 구현 객체에 있는 메소드를 수행시키고, 그 결과값을 돌려 받기를 원하는 프로그램이다. 그리고, 구현 객체는 클라이언트에서 필요로 하는 객체를 구성하는 애트리뷰트와 오퍼레이션을 구현한 프로그램이다.

클라이언트와 구현 객체간의 동작 원리를 간단히 설명하면 클라이언트는 필요로 하는 구현 객체의 오퍼레이션으로부터 서비스를 받기 위해 서비스 요청(request)을 ORB에게 보내고, ORB는 서비스요청을 수행할 구현 객체를 찾는다. 그리고, 해당 구현 객체가 서비스요청을 받을 수 있도록 하며 구현 객체는 ORB가 보내준 서비스 요청을 해당 오퍼레이션을 수행한 후 그 결과를 ORB를 통하여 클라이언트에게 돌려준다. 그러므로 ORB에 의해서 클라이언트는 구현 객체가 어디에 있으며 어떤 프로그램언어로 작성되었는가 사전 정보 없이 독립적으로 구현 가능하다.

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 수련으로 전달하는 메커니즘이다. IDL컴파일러가 OMG IDL인터페이스 정의를 이용하여 클라이언트 스텁과 구현 골격을 생성한다.

◉ 동적 호출

CORBA 시스템은 실행시에 인터페이스 정보가 저장된 인터페이스 저장소를 이용 하여 동적 호출 인터페이스(dynamic invocation interface)를 통하여 요청의 생성을 지원한다.[1]

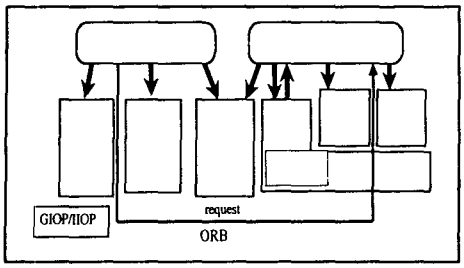

그림 1. CORBA 구조

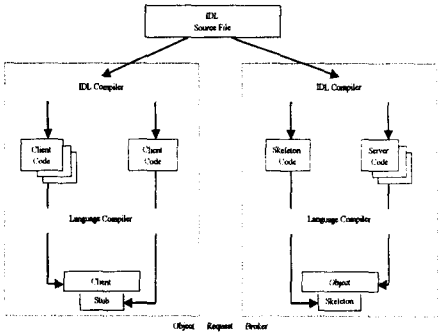

그림 2. IDL 컴파일러의 역할

III. IDL 컴파일러

3.1 IDL 컴파일러의 역할

IDL 컴파일러는 일반적으로 기계어 코드를 생성하는 범용 컴파일러들과는 달리 서버의 객체가 클라이언트들에게 제공하고자 하는 서비스들에 대한 표현을 포함하고 있는 IDL 파일을 입력받아서 특정 프로그래밍 언어로의 사상 규약을 통한 스터브나 스켈레톤 코드를 생성해주는 소프트웨어이다.

분산 환경에서 프로그램을 개발하는 경우 CORBA에서 정의된 IDL을 특정 프로그램 언어로 사상하는 IDL 컴파일러의 사용으로 인해서 시스템 개발자나 사용자들은 공통된 인터페이스를 통해 복잡한 프로그램을 분담하여 개발할 수 있는 편의를 제공받을 수 있을 뿐만 아니라 IDL의 독립적인 특성으로 인해 작성 후에도 소프트웨어의 유지 보수에 상당한 도움을 받게 된다.

그림.2에서 보는 바와 같이 IDL 컴파일러는 IDL파일을 입력받아서 컴파일 한 출력물로 클라이언트와 구현 객체(서버) 프로그램 이외에도 특정 프로그래밍 언어로 작성된 클라이언트 스터브 코드와 구현 객체 스켈레톤 코드가 생성된다. 최종적으로 IDL컴파일러에서 출력된 스터브 코드와 클라이언트 프로그램, 스켈레톤 코드와 객체 구현 프로그램을 함께 특정 언어 컴파일러로 컴파일하여 실질적으로 ORB에서 수행되는 프로그램들을 생성하게 된다. 특히 사상되는 프로그래밍 언어의 종류에 따라 스터브과 스켈레톤이 다른 프로그래밍 언어로 생성될 수도 있는데 이러한 점은 클라이언트와 객체 구현이 서로 다른 프로그래밍 언어로 작성되는 것을 가능하게 하여 기존에 구현되었던 프로그램들이 다른 언어로 구현된 프로그램과 호환될 수 있는 장점이 있다.[1][2][3]

3.2 IDL 파일 구조

IDL 문법을 분석한 결과 IDL 컴파일러의 입력 파일이 되는 IDL 소스 파일의 구조는 그림.3과 같다.

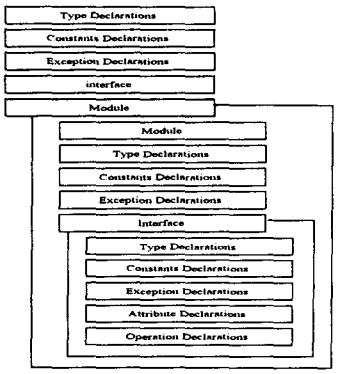

그림 3. IDL 파일의 구조

사용되는 범위를 한정하는 기능을 가진 모듈은 중첩이 가능하며 모듈 내에는 서버 객체에서 사용되는 데이터 형 및 상수 선언문, 서버 객체에서 수행되는 오퍼레이션들을 포함하고 있는 인터페이스 정의들이 있다. 그 중에서 모듈 외부에 전역 선언이 가능한 데이터형 및 상수 선언분, 예외 선언문, 인터페이스 정의들은 CORBA 환경 내에서 어떠한 곳에서도 참조가 가능하게 된다.

실질적으로 서버 객체들을 명시하게 되는 인터페이스는 서버 객체에서 사용하는 데이터 형 및 상수 선언문, 서버 객체 수행 시 발생되는 예외에 관한 예외선언문, 서버 객체의 속성들인 속성 선언문, 서버 객체의 서비스를 수행하는 오퍼레이션들로 구성된다.

3.3 어휘 분석기

실제 IDL 파일이 IDL 컴파일러의 어휘 분석기에 전달되기에 앞서 일반적인 C나C++언어에서 사용되는 #include, #define과 같은 전처리 문을 처리하는 전처리기(preprocessor)를 거치게 된다. 해당 전처리기를 거친 IDL 소스 파일(source file)은 어휘 분석기에서 정규 표현식으로 표현한 패턴에 맞는 토큰으로 분리되어 문법을 검사하는 파서로 전달된다. 여기서 키워드, 연산자(operator), 식별자(identifier) 등 이 토큰에 해당된다. 어휘 분석기에서는 이와 같이 IDL 문법에서 정의된 토큰들을 구문 분석기에 순서대로 전달하는 동시에 전달할 토큰이 해당 정규 표현식에 맞지 않을 경우에는 에러를 출력한다.

3.4 파서(parser)

IDL 컴파일러에서의 파서는 구문 분석 단계에서 어휘 분석기로부터 토큰 스트림을 전달받아 입력 파일 구문이 IDL 문법에서 허용하는 올바른 문장인가를 검사하며 의미 분석 단계에서는 각각의 식별자에 대해서 생성된 심벌 테이블을 참조하여 상수와 형 정의문에서 사용되는 각각의 데이터형과 범위 규칙(scope rule)을 올바르게 지키는지를 검사한다.

최종적으로 구문 분석과 의미 분석을 거친 뒤에 목적 코드 생성기의 입력이 되는 중간 코드를 생성하게 된다. 중간 코드는 일반적으로 구문 트리를 많이 사용하는데 본 논문에서는 AST를 사용하였다.[3][4][5][6]

IV. AST

인터페이스 정의 언어 컴파일러인 CFE(Compiler Front End)는 Sun Microsystems에서 제공하는 IDL 컴파일러로 IDL로 정의된 인터페이스를 어휘 분석 및 구문 분석을 하여 AST(Abstract Syntax Tree)를 생성한다. 이 AST는 CFE가 정의하는 IDL 구문의 구조와 유사한 계층 구조를 가진 클래스 인스턴스로 구성 되는 트리이다. 그림.4는 CFE가 제공하는 클래스들의 계층도이다.[7]

UTL_Scope 클래스는 정의 스코프 관리와 네임 룩업을 제공하며, AST_Decl은 AST_Expression을 제외한 모든 AST 클래스의 베이스 클래스이다. AST_Decl들은 노드 타입, 이름, 정의 스코프와 정의된 파일 이름 그 파일에서 정의된 라인등을 멤버 데이터로 갖는다. AST_Type 클래스는 IDL에서 타입 정의나 배열등을 나타내는 구문에 대한 베이스 클래스이다. AST_ConcreteType 클래스는 interface를 제외한 비인터페이스 타입을 나타내는 모든 클래스의 베이스 클래스이다.

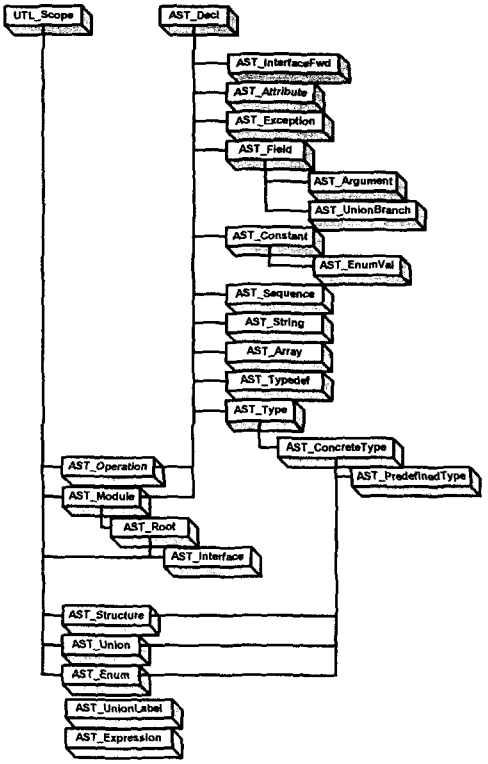

그림 4. CFE 클래스 계층도

AST_PredefinedType 클래스는 long, char등과 같은 정의된 타입을 나타내는 클래스로 AST_ConcreteType을 상속하며 지정된 특정 타입을 갖는다. AST_Module 클래스는 모듈 구문을 정의하는 클래스로 AST_Decl과 UTL_Scope 클래스등을 상속한다. AST_Root 클래스 노드는 AST의 루트이다. AST_Root는 AST_Module의 서브클래스로 AST와 연결을 위해 사용할 수 있다. AST_Interface 클래스는 IDL의 인터페이스를 나타내는 클래스로 AST_Type과 UTL_Scope 클래스를 상속한다. AST_InterfaceFwd 클래스는 IDL 인터페이스의 포워딩 선언을 나타내는 클래스로 AST_decl 클래스를 상속한다. AST_Attribute 클래스는 IDL의 속성 구문을 나타내며 AST_Decl 클래스를 상속한다. AST_Exception 클래스는 IDL의 예외 구문을 나타내며 AST_Decl 클래스를 상속한다. AST_Operation 클래스는 IDL 오퍼레이션 구문을 나타내며 AST_Decl과 UTL_Scope를 상속한다. AST_Argument 클래스는 IDL 오퍼레이션 구문의 인자를 나타내며 AST_Field 클래스를 상속한 클래스이다.[7]

V. Intermediate Representation

기존 IDL 컴파일러는 전반부에서 출력으로 나오는 AST를 사용하여 스켈레톤 코드와 스터브 루틴을 생성하였다. 이것은 다중 목적 언어를 지원하기 위한 방법을 어렵게 하며 다양한 트랜스포트 메카니즘 적용을 어렵게 한다. 그와 함께 후반부에서 제공할 수 있는 최적화 기법 적용을 어렵게 한다. 그러므로 이러한 AST를 사용한 코드 생성을 대신할 수 있는 방법으로 기존 프로그래밍 언어에서 사용하는 중간 코드 생성 부분을 IDL 컴파일에 도입한다.[8]

중간 표현을 생성 부분은 다음과 같은 세 가지 기본 타입을 사용하여 표현한다. 첫 번째는 중간 표현에서 사용하는 IDL 구문을 나타내기 위한 타입과 두 번째는 기본 타입을 사용한 복합 타입을 나타내기 위한 타입이다. 세 번째는 사용자 타입 정의와 같은 경우 사용하는 타입이다.

표.1은 중간 표현에 사용되는 타입들을 보인다.

우리는 이러한 타입들을 사용하여 CFE에서 제공하는 각 클래스에 emit_representation() 메서드를 추가하고 AST 생성후 idl_global->ast_root()를 호출하여 AST의 루트 노드에 액세스하여 각 클래스의 emit_representation() 메서드를 차례로 호출하여 중간 표현 결과를 생성한다.

표 1. 중간 표현에 사용되는 타입들

indirect	다른 정의에 대한 참조
integer	32bit 정수(unsigned 포함)
float	실수
char	문자
array	배열
struct	고정 길이를 가진 구조체
union	분리자가 사용되는 유니언
const	상수값
void	void 형
enum	열거형
exception	예외
interface	IDL 인터페이스
Item	복합 구조의 항목
definition	사용자가 정의하는 데이타형

아래 그림.5는 IDL로 정의된 인터페이스와 이것에 대한 중간 표현 결과를 나타낸 것이다.

```
module compute {
interface   math {
            long        add(in unsigned long a,
in long b);
            void        sub(in long a, in long b);
        };
};
```

```
Slot #1
Scope #0
Name "compute"
  kind = NAMESPACE
  namespace
Slot #2
Scope #1
Name "math"
  kind = INTERFACE
  code:
    CONST_ARRAY:
    #0 -        'm'
    #1 -        'a'
    #2 -        't'
    #3 -        'h'
    #4 -        '\000'
  parents:
  operations:
    operation name: "add"
    flags:
      none
    params:
    #0 -        name = "a"
      direction = IN
      type =
        kind = INTEGER
        0-4294967295
    #1 -        name = "b"
      direction = IN
      type =
        kind = INTEGER
        -2147483648-2147483647
    return type:
      kind = INTEGER
      -2147483648-2147483647
    exceptions:
    request code:
      CONST_ARRAY:
      #0 -        'a'
      #1 -        'd'
      #2 -        'd'
      #3 -        '\000'
    reply code:
      CONST_ARRAY:
      #0 -        '$'
      #1 -        'a'
      #2 -        'd'
      #3 -        'd'
      #4 -        '\000'
    operation name: "sub"
    flags:
      none
  params:
    #0 -        name = "a"
      direction = IN
      type =
        kind = INTEGER
        -2147483648-2147483647
    #1 -        name = "b"
      direction = IN
      type =
        kind =INTEGER
        -2147483648-2147483647
    return type:
      kind = VOID
      void
    exceptions:
    request code:
      CONST_ARRAY:
      #0 -        's'
      #1 -        'u'
      #2 -        'b'
      #3 -        '\000'
    reply code:
      CONST_ARRAY:
      #0 -        '$'
      #1 -        's'
      #2 -        'u'
      #3 -        'b'
      #4 -        '\000'
    attributes:
    exceptions:
```

그림 5. IDL 정의와 중간 표현 결과

VI. 결론

본 논문에서는 기존 프로그래밍 언어에서 후반부 최적화 지원을 위해 생성되는 중간 코드 생성을 방법을 사용하는 방식을 IDL 컴파일에 적용을 하였다. CFE 컴파일러의 AST를 보다 추상화된 인터페이스 표현인 중간 표현으로 변환하여 다음 단계의 최적화 및 목적 코드 단계에서 다양한 방식을 적용할 수 있게 했다.

이러한 컴파일러 후반부 처리에 필요한 중간 표현이 좀 더 세분되어져야 할 필요성이 있음을 알게 되었다. 모든 인터페이스는 스켈레톤과 스터브 루틴이 생성 되어 전송에 필요한 메시지 구성 형식을 지원하도록 하기 위해 메시지 구성을 위한 중간 표현이 앞으로 더 연구 되어야 할 것이다.

[참고문헌]

[1]OBJECT MANAGEMENT GROUP. The Common Object Request Broker: Architectureand Specification, 2.0 ed., July 1995.

[2]SRINIVASAN, R. RPC: Remote procedure call protocol specification version 2. Tech. Rep. RFC 1831, Sun Microsystems, Inc., Aug. 1995.

[3]이진호,이근영, 정태명, "Java ORB 시스템을 위한 IDL-to-Java컴파일러 개발" 한국 정보처리 학회 논문지 제5권 제8호 1998.8

[4]SUN MICROSYSTEMS, INC. ONC+ Developer's Guide, Nov. 1995. SUNSOFT, INC. SunSoft Inter-ORB Engine, Release 1.1, June 1995. ftp://ftp.omg.org/pub/interop/iiop.tar.Z.

[5]SRINIVASAN, R. XDR: External data representation standard. Tech. Rep. RFC 1832, Sun Microsystems, Inc., Aug. 1995.

[6]NETBULA, LLC. PowerRPC, Version1.0,1996. http://www.netbula.com/products/powerrpc/

[7]SUN MICROSYSTEMS, INC. 1992. ftp://ftp.omg.org/pub/OMG_IDL_CFE_1.3.

[7]O'MALLEY, S., PROEBSTING, T. A., AND MONTZ, A. B. USC: A universal stub compiler. In Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM) (London, UK, Aug. 1994), pp. 295--306.

[8]JANSSEN, B., AND SPREITZER, M. ILU 2.0 alpha Reference Manual. Xerox Corporation, May,1996. ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.