# Typed Object Bus를 이용한 새로운 개념의 분산플랫폼

## (The Typed Object Bus Platform : A New Paradigm for a Distributed Platform)

김 상 경 † 선 경 섭 †† 안 순 신 †††
(Sangkyung Kim)(Kyungsup Sun)(Sunshin An)

**요 약** 본 논문에서는 Typed Object Bus (TOB) 플랫폼이라는 새로운 개념의 분산플랫폼을 제안한다. 기존의 분산플랫폼들은 분산 객체간 인터페이스 관계에 초점을 맞추어 단일의 통신 메커니즘을 이용하여 비제한적인 통신을 제공하였으나 TOB 플랫폼은 상호작용 관계에 중점을 두고 응용의 특성에 따라 다양한 통신 메커니즘을 사용하여 플랫폼 제어 통신을 제공한다. TOB는 연산동작특성과 TOB를 통해 전달 가능한 데이타 타입, 그리고 객체통신에 제약을 가하는 다수의 속성들에 의해 표현된다. 응용객체는 TOB 플랫폼이 제어하는 Typed Object Bus들을 통해서만 다른 객체와 통신이 가능하다. TOB 플랫폼은 TOB Type Definition Language (TDL) 을 이용하여 여러 가지 형태의 상호작용을 규정함으로써 유연하고 다양하게 연산 및 스트림 특성을 갖는 통신을 제공할 수 있다.

***Abstract*** In this paper, we propose a new paradigm for distributed platform design, the *Typed Object Bus* (TOB) platform. While traditional distributed platforms have focused on interface relationships between distributed objects and provided unrestricted communication using a uniform communication mechanism, the TOB platform is centered on the interaction relationship and provides platform-controlled communications using diverse communication mechanisms that conform to application characteristics. Typed object buses represent the communication behaviors they support, the data types to be delivered through them and various properties that constrain the communications. Application objects can communicate with other objects only via typed object buses the TOB platform controls. The TOB platform provides various and flexible communications by specifying a variety of styles of interactions with the TOB Type Definition Language (TDL) that we have developed. TOB TDL has been designed to accommodate both operation and stream characteristics of an application.

## 1. 서 론

분산플랫폼은 분산 응용을 지원하기 위한 기간구조로서의 중요한 역할을 담당하고 있다. 분산플랫폼은 분산 시스템의 이질성을 극복하고 분산 객체간 원격 통신이 가능하게 한다. OMG의 CORBA [1] 와 마이크로소프트의 DCOM [2] 등 여러 플랫폼들이 개발되어 운용되고 있지만 그것들은 서로 통신하는 응용 객체들간 인터페이스 관계의 유지에만 초점이 맞추어져 있기 때문에 상호작용 패턴, QoS 속성, 객체간 통신구성과 같은 다양한 통신특성을 플랫폼 수준에서 식별하고 적절한 통신 메커니즘을 객체간 통신에 적용하는 능력이 부족하다. 결과적으로 기존의 분산플랫폼들은 단일의 통신 메커니즘을 사용한 비제한적인 통신을 제공하며 응용의 특성에 따라 특화 된 통신 서비스를 유연하고 효율적으로 제공하는데 어려움이 있다. 또한 분산플랫폼은 응용 객체간 통신경로만을 제공하며, 통신특성 처리의 대부분은 응용에 의해 처리되어야 한다. 그리고 서버 객체가

· 본 논문은 1998년도 한국학술진흥재단의 자유공모과제에 의해 지원을 받은 것임.
† 학생회원 : 고려대학교 전자공학과
skkim@dsys.korea.ac.kr
†† 비 회 원 : 한국통신 연구센터 연구원
kssun@kt.co.kr
††† 종신회원 : 고려대학교 전자공학과 교수
sunshin@dsys.korea.ac.kr
논문접수 : 2000년 2월 15일
심사완료 : 2000년 8월 1일

제공하는 인터페이스의 시그니처(signature)와 일치만 한다면 클라이언트 객체로부터의 모든 서비스 요청은 플랫폼에 의한 객체 상호작용의 정당함의 확인과정 없이 서버로 전달된다.

본 논문에서는 위에서 언급한 기존 분산플랫폼들의 취약점들을 해결하는 Typed Object Bus(TOB) 플랫폼을 제안한다. TOB 플랫폼은 상호작용 관계에 초점을 맞추고 있으며 응용 특성에 따라 다양한 통신 메커니즘을 사용하는 플랫폼 제어 통신을 제공한다. TOB는 고유한 타입에 의해 식별되며, 이 타입은 연산동작특성과 TOB를 통해 전달될 수 있는 데이타 타입, 그리고 객체간 통신에 제약을 가하는 여러 가지 속성들에 표현된다. TOB의 타입은 TOB 타입정의언어(Type Definition Language: TDL)를 사용하여 정의된다. 응용 객체들은 TOB 플랫폼에 미리 등록된 TOB 들을 통해서만 다른 객체들과 통신하게 된다. TOB 플랫폼은 객체간 상호작용에 포함된 연산 데이타 타입이나 연산 순서가 사용되는 TOB 타입과 일치하는지 확인하여, 일치하지 않을 경우는 통신을 허용하지 않는다. 즉, TOB 플랫폼은 TOB를 이용하여 플랫폼을 통한 통신 사용패턴을 제어하며 제약을 가한다. 이것은 분산시스템의 관리능력과 분산통신의 신뢰성을 향상시키고 적절하지 못한 트래픽이 플랫폼 상으로 전달되는 것을 방지한다. 또한, 응용에 따라 특화 된 통신 메커니즘이 TOB에 내재되므로 응용의 부담을 경감시키고 통신을 보다 신뢰성 있게 제공할 수 있다.

본 논문의 구성은 다음과 같다. 다음 절은 관련 연구를 다룬다. 3절에서는 TOB 아키텍처의 개요를 설명하고 4절과 5절에서 TOB 시맨틱 속성 트리와 TOB TDL에 대해서 각각 기술한다. 6절에서는 TOB 플랫폼의 기능적 모델과 각 기능 컴포넌트간 상호 운용 시나리오를 설명하고 7절에서 결론을 맺는다.

## 2. 관련 연구

### 2.1 기존 플랫폼과 TOB 플랫폼과의 비교

CORBA [1]와 DCOM [2] 등의 기존 분산플랫폼에 비하여 TOB 플랫폼은 크게 2가지 특징에 의해 구별될 수 있다.

- 통신의 특성에 따라 차별화 된 통신경로의 제공: 기존 분산플랫폼은 IDL에 의해 규정되는 응용 객체간 인터페이스 관계, 즉 인터페이스를 구성하는 연산과 그 연산을 구성하는 파라미터의 타입을 정의하고 그것들을 제대로 타입에 맞게 이용하는 것에만 초점이 맞추어져 있다. 따라서 응용 객체간 상호작용이 어떻게 이루어지는지는 정의하지 않으며, 연산의 상호작용 패턴과 순서, 상호작용에 요구되는 프로토콜 등은 응용이나 플랫폼의 하부 레벨에서 정의되고 구현되어야 한다. 기존 분산플랫폼은 그 위에서 실행되는 응용의 성격에 상관없이 획일화 된 통신 서비스를 제공한다. 반면에 TOB 플랫폼은 응용의 통신특성에 따라 차별화 된 통신경로를 제공하고 필요한 통신 메커니즘을 내재하여 제공하는 것이 가능하다. 예를 들어, 점대다중점 통신이 요구되는 경우 점대다중점 구성을 지원하는 TOB를 통하여 통신 서비스를 제공한다.

- 플랫폼 수준에서의 타입 검사: 기존 분산플랫폼의 경우 클라이언트가 서버로부터의 서비스를 이용하기 위해서는 먼저 서비스를 요청하는 형식이 서버에서 정의한 인터페이스의 타입과 일치될 수 있도록 서버측 스터브와 같이 링크를 하여 구현 코드를 생성하여야 한다 (정적인 타입 검사). 그 서비스 요청은 플랫폼 (ORB 또는 DCOM의 SCM)에 의한 추가적인 검사 없이 그대로 서버로 전달된다. TOB 플랫폼의 경우는 TOB를 통하여 전달되는 연산이나 데이타의 타입이 개별 TOB가 고유하게 가지고 있는 TOB 타입과 일치하는 지를 분산통신 서비스를 제공할 때 런타임(runtime)으로 확인한다. 만일 일치하지 않을 경우 요청된 서비스는 TOB를 통해 제공될 수 없게 된다. 또한 TOB에 접근할 수 있는 대상을 제한함으로써 TOB 플랫폼의 관리자는 TOB를 이용하는 응용 객체간 분산 통신을 통제할 수 있다.

### 2.2 Module Interconnection Language (MIL) 접근 방법

MIL [3] 은 분산 응용의 구성 및 관리를 보다 용이하게 하고 그들 간 상호 운용성을 향상시키기 위하여 개발된 선언 언어이다. MIL 방식은 TOB와 같은 타입별 통신을 제공하기 위한 한 시도로 고려될 수 있으며, Polylith 소프트웨어 버스 [4] 나 MIL을 사용하여 응용모듈간 통신을 기술하는 방법을 제시한 Olan [5] 등이 대표적인 연구이다.

Polylith 소프트웨어 버스는 분산 응용모듈을 어떻게 쉽게 통합할 것인가의 관점으로부터 연구되었다. Polylith 소프트웨어 버스는 소프트웨어 응용의 인터페이스의 상세한 부분을 캡슐화 하는 일종의 에이전트이다. 프로그래머는 먼저 응용의 기능적 요구사항을 어떻게 구현할 것인지를 결정하고 적당한 인터페이스 메커니즘을 제공하는 버스를 선택한다. 프로그램 모듈은 버스에 직접 접속되며 다른 프로그램 모듈에는 버스를 통하여 연결된다. 이 방식은 분산 응용 모듈들의 통합을 쉽게 하지만 정적인 구성만을 지원하고 바인딩 된 응용

모듈의 이름 및 주소를 버스에 명시적으로 나타내야 하므로 다양하고 동적인 분산 통신을 요구하는 분산플랫폼에는 적당하지 않다.

Olan은 상호 운용성을 향상시키기 위하여 응용 컴포넌트 간 바인딩 객체로서 커넥터 개념을 도입하였다. Olan 커넥터는 Darwin [6] 과 Formal Connector [7] 개념을 확장한 것으로서 응용 구성의 동적인 측면을 강조하며, 커넥터의 입출력 요구사항과 프로시져 호 모델(procedure call model), 메시지 모델(message model) 등의 하부 통신 모델을 규정한다. 그러나 응용 객체 간의 연산 상호작용과 전달되는 데이타의 타입들은 커넥터에 정의되지 않고, 응용 구조에서 정의된다. 따라서 Olan 커넥터는 다양한 상호작용 관계를 플랫폼 레벨에서 표현하거나 제어할 수 없다.

### 2.3 Open Distributed Processing (ODP) 바인딩 객체

ODP [8][9]는 2개 이상의 객체들을 서로 바인딩 시키는 중간 매개체로서 바인딩 객체를 정의한다. 바인딩 객체는 클라이언트와 서버 객체 간 상호 통신을 지원하고, 다양한 통신 프로토콜들을 캡슐화 하며, 바인딩 행동에 제약을 가하는 서비스 품질 요구사항을 내재할 수도 있다. 그러나 바인딩 객체는 두 가지 관점에서 TOB와 다르다. 첫째, 바인딩 객체는 객체 통신에 대한 액세스를 제약하거나 통신의 사용패턴을 제한할 수 없다. 즉, 객체들은 그들의 인터페이스가 서로 일치한다면 다른 것과 통신하는 것이 가능하다. TOB 플랫폼 방식에서는 객체의 인터페이스가 TOB의 인터페이스와 일치해야 객체간 통신이 이루어진다. 둘째, ODP에서는 응용객체의 인터페이스의 기술에 의해 바인딩 객체의 인터페이스를 간접적으로 규정하지만, TOB 플랫폼에서는 TOB 자체의 인터페이스를 직접 정의함으로써 플랫폼에 의한 제어가 가능한 통신 서비스를 제공할 수 있다.

## 3. TOB 아키텍처의 개요

### 3.1 Typed Object Bus

TOB는 객체간 통신관계의 특성과 상호작용 패턴에 의해 표현되는 고유한 TOB 타입을 가지며, 분산플랫폼 관리자의 관리정책에 따라 동적으로 구성되고 관리된다. 프로그래밍 언어에 있어 변수의 데이타 타입이 변수가 가질 수 있는 값의 범위를 한정하는 것과 같이 TOB 타입은 분산플랫폼의 통신 서비스에 대한 사용 패턴을 제한한다. 따라서 TOB 타입의 집합은 플랫폼 통신 서비스를 사용하기 위한 구조적 제약으로 작용된다.

분산된 객체들이 TOB를 이용하여 서로 통신할 때 TOB를 통하여 전달되는 연산 및 데이타의 타입과 상호작용 패턴은 TOB에 의해서 그 적정성이 보장된다. 객체 스터브를 이용하여 응용 프로그램의 컴파일 과정에서 타입 검사를 수행하는 기존의 방법과는 달리 TOB는 TOB 타입에 정의된 대로 런타임 시에 타입 검사를 수행한다. TOB 타입에서 정의된 상호작용 패턴과 TOB를 이용하는 객체의 TOB 액세스 순서가 반드시 일치할 필요는 없는데 이것은 TOB에서 내부적으로 순서를 기억하고 있으므로 버스 액세스 요구가 필수적인 시간적 동기 조건을 위배하지 않는 한, 비순서화 된 요구나 일괄(batch) 요구에 대해서도 이를 수용할 수 있기 때문이다.

일반적으로 서버가 제공하는 서비스는 인터페이스나 연산의 이름에 의해 식별된다. TOB가 수행하는 타입 검사는 연산의 구조나 순서 등과 같은 상호작용 패턴의 호환성 검사만을 포함하며 특정한 서비스의 연산이나 행위를 대상으로 하지 않는다. TOB를 통해 전달되는 연산의 이름, 즉 서비스 로직의 식별자는 TOB에서 검사되지 않으며 상대편 객체로 그대로 전달된다. 따라서 동일한 타입의 TOB를 서로 다른 서비스에 이용하는 것이 가능하다.

### 3.2 TOB 바인딩

분산 객체들이 서로 통신하기 위해서는 동일한 타입의 객체 인터페이스 간에 TOB 바인딩이 먼저 성립되어 있어야 하는데 TOB 바인딩은 TOB를 통하여 2개 이상의 객체 인터페이스를 서로 결합시키는 것을 말한다. TOB 바인딩 동작은 연산 및 스트림 인터페이스에 대해 모두 적용 가능하다. 그림 1에 나타낸 바와 같이 TOB 바인딩 동작은 TOB의 양단에서 일어난다. TOB 타입이 응용의 통신특성과 일치한다면 응용 객체의 인터페이스는 TOB에 바인딩 될 수 있다. 일반적으로 서버는 클라이언트의 요구에 의해 TOB에 바인딩 된다. 그러나 클라이언트가 요구하기 전에 플랫폼 또는 서버 관리자에 의해 서버가 미리 TOB에 바인딩 될 수 있다. 이 경우 TOB 바인딩 지연을 줄이고 서비스를 보다 빠르게 제공할 수가 있다.

TOB 바인딩에서 고려되어야 할 사항 중의 하나는 적합한 TOB의 선택에 관련된다. TOB는 객체 사이에 필요한 통신 메커니즘에 메시지 구조, 메시지 순서, 전송 링크의 특성 등 다양한 시맨틱 속성을 개별화해서 생성되는 소프트웨어 버스이므로, 응용 객체 입장에서는 통신 시맨틱 속성이 달라지는 객체 요구가 발생할 때마다 그에 적합한 TOB를 사용해야 한다.

클라이언트나 서버는 응용이 요구하는 통신특성에 따

라서 하나 이상의 TOB에 바인딩 될 수 있다. 그림 1은 TOB를 통한 통신의 한 예를 보여 준다. 그림에서 TOB는 연산 메시지를 전달하는 연산 버스이다. 클라이언트 객체는 다른 통신 패턴을 갖는 3개의 TOB를 통해 2개의 서버 객체와 통신한다. 2개의 서로 다른 서버 객체는 점대다중점 구성의 형태로 동일한 TOB에 바인딩 될 수 있다.

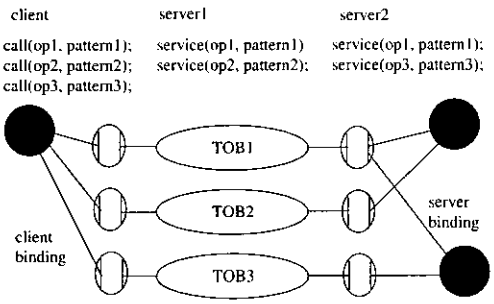


그림 1 TOB 바인딩의 예

### 3.3 상호작용 메커니즘

TOB 바인딩이 성립된 후 다양한 형태의 상호작용이 응용 객체와 TOB사이의 경계에서 발생할 수 있다. 이러한 상호작용은 연산 이름, 정의된 TOB 타입과 일치하는 연산 파라미터의 전달 등으로 구성되는데 발생 가능한 여러 유형의 상호작용 패턴을 자유롭게 표현하기 위하여 그림 2와 같은 일련의 이벤트로써 기술된다. TOB인터페이스에서의 상호작용은 두 가지 형태로 나타날 수 있다. 즉, 그림 2의 (a)에 나타낸 것과 같이 기존에 하나의 고정적 패턴으로 취급되었던 인터페이스 시그니처의 처리는 그것을 구성하는 상호작용 패턴 요소들의 전달 이벤트로 기술된다. 이것은 객체의 상호작용 연산을 보다 세밀한 단위로 표현할 수 있도록 한다. 상호작용 패턴 요소의 분해는 객체간 상호작용 모델에서 고려되지 않았던 각 요소간의 동기 관계를 표현할 수 있도록 해준다. 또한 그림 2의 (b)에 보인 바와 같이 객체간에 발생되는 일련의 연속된 연산 작용은 TOB인터페이스에서는 마치 하나의 트랜잭션 처럼 이벤트의 연결 (concatenation) 패턴으로 처리될 수 있다. 이 때 TOB는 인터페이스 동작에 대한 연산순서를 보장해야 한다.

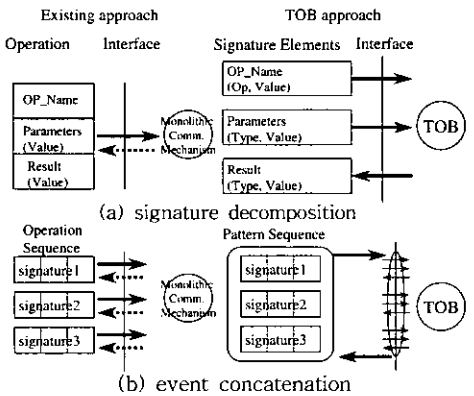


그림 2 TOB 인터페이스에서 발생하는 상호작용 이벤트의 표현

## 4. TOB 시맨틱 트리의 구성

TOB 타입은 여러 시맨틱 속성들의 집합으로 나타난다. 본 논문에서는 TOB 시맨틱 속성 트리라는 트리 구조를 사용하여 시맨틱 속성들을 분류하였다. TOB 시맨틱 속성 트리는 TOB를 기술할 때 공통적으로 요구되는 시맨틱 속성들의 관계를 구조적이고 명확하게 표현한다.

TOB 시맨틱 속성 트리의 각 노드는 같은 레벨에 있는 다른 노드와 직교적인 관계를 유지한다. 속성 노드는 비단말 노드와 단말 노드로 구성된다. 비단말 노드는 속성들의 집합을 표시하며 부속성을 나타내는 하나 이상의 자식 노드를 갖는다. 단말 노드는 개별 속성 값을 가지며 어떠한 자식 노드도 갖지 않는다. 그림 3은 TOB 시맨틱 속성 트리의 구성을 보여 준다.

TOB 시맨틱 속성 트리는 3개의 최상위 속성들로 분류된다. 첫번째 속성은 'topology'이다. 'topology'는 TOB의 연결구성을 나타내며 점대점 및 다중점 구성을 속성 값으로 갖는다.

두 번째 속성은 'interface type'이며 TOB에서 사용될 상호작용의 유형을 식별한다. 이것은 다시 'operation' 및 'stream' 버스 인터페이스 속성으로 구성된다. 'operation' 버스 인터페이스는 서비스 요청을 위한 파라미터와 서비스 실행 결과 등과 같은 불연속 형태의 정보를 전달하기 위한 통신 메커니즘을 요구한다. 반면에, 'stream' 버스 인터페이스는 연속적인 데이타의 흐름을 지원할 수 있는 통신 메커니즘을 요구한다. 'operation' 속성은 'op_bus' 와 'op_sem' 속성으로 나누어 진다. 'op_bus' 속성은 연산 버스 인터페이스의 시그니처를 기술하기 위하여 사용되는데 연산 이름, 파라미터, 리턴 값 타입 등을 포함한다. 'op_sem' 속성은 시스템이나 통신 링크에 고장이 발생할 때 요청된 연산의 처리 방법을 가리킨다. 이 속성은 RPC 요청 시맨틱 개념으로부터 도입되었으며, 'at_least_once'와 'best_

effort'의 2개의 값을 갖는다. 'stream' 속성은 'st_bus'와 'stqos_sem'으로 나누어 진다. 'st_bus' 속성은 지원하는 스트림의 타입을 규정하기 위하여 사용된다. 스트림 버스 규격은 스트림을 구성하는 플로우(flow)들에 대하여 정의하는데 각 플로우는 플로우 역할, 플로우 이름, 플로우 타입, QoS 속성 등으로 기술된다. 'stqos_sem' 속성은 스트림 QoS를 보장하는 정책과 관련되며 요구된 QoS가 하부 트랜스포트 망에서 만족된다면, 'guaranteed,' 아니면, 'best_effort'의 값을 갖는다.

세 번째 속성은 'bus property'이다. 이것은 TOB의 비기능적 속성을 정의하며 TOB에 접근이 허용되는 클라이언트나 서버의 식별 정보인 'access control list'와 가용성 등에 관한 속성을 가지고 있다. 'access control list' 속성은 TOB 접근에 대한 인증, 권한 부여에 이용되며, 문자열 (string_literal)로 표현된다. 가용성 속성은 고장이 발생할 때 TOB를 통한 통신 서비스가 복구되는 방법에 따라 'cold_stby'와 'warm_stby,' 'hot_stby'의 3개의 값을 가질 수 있다.

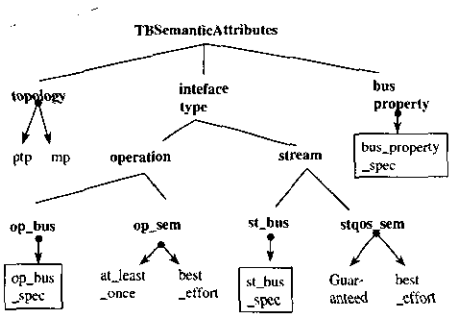


그림 3 TOB 시맨틱 속성 트리

## 5. TOB 타입 정의 언어(Type Definition Language: TDL)

기존의 인터페이스 정의 언어 [1][10]가 응용 객체의 인터페이스를 기술하기 위하여 개발된 반면, TOB TDL은 TOB의 인터페이스를 나타내는 TOB 타입을 기술하기 위하여 개발되었다. 기존의 인터페이스 정의 언어는 TOB의 버스 토폴로지, 버스의 비기능적 속성, 세밀한 상호작용 메커니즘 등 다양한 시맨틱 속성들과 동작 특성들을 표현하는데 부족함이 있기 때문에 우리는 TOB TDL을 개발하였다.

TOB TDL은 확장형 Backus-Naur Form으로 정의되며 C++ 프로그래밍 언어의 'struct' 구문 구조를 기반으로 한다. TOB TDL에서 사용되는 구문 규약은 C++ 프로그래밍 언어의 그것과 일치한다. TOB TDL 문법을 부록에 첨부하였다.

### 5.1 TOB TDL 규격

TOB TDL 규격은 <bus_context_spec_part>, <bus_interface_spec_part>와 <bus_property_spec_part>의 세 부분으로 나누어 진다.

<bus_context_spec_part>는 TOB가 지원하는 토폴로지 속성을 규정한다. 다음은 토폴로지 속성이 점대다중점 구성일 경우의 기술 예이다.

```
bus_context context_id {
        (topology : ptmp);
}
```

<bus_property_spec_part>는 TOB의 'access control list'와 가용성 등과 같은 비기능적 속성을 정의한다. 다음은 <bus_property_spec_part>의 규격 예이다.

```
bus_property property_id {
(access_ctrl : any_full);
   (availability : cold_stby);
}
```

<bus_interface_spec_part>는 연산 버스를 위한 <op_bus_interface_part>와 스트림 버스를 위한 <st_bus_interface_part>로 구성된다. <op_bus_interface_spec>는 <op_sem> 속성 선언부와 연산 버스 타입 선언부로 나누어진다. 연산 버스 타입 선언부는 그림 4에 보인 것과 같이 헤더(header), 몸체(body), 그리고 트레일러(trailer)부로 구성된다. 헤더부는 연산 버스의 이름을 선언하며, 몸체부는 연산과 연산 파라미터, 그리고 트레일러부의 <supporting_type_dcl>에서 규정되는 또 다른 연산 버스를 멤버로 하여 TOB가 지원하는 상호작용 패턴을 정의한다. 몸체부의 각 멤버는 응용 객체와 TOB 간 경계에서 발생하는 하나의 이벤트로서 모델링된다.

TOB 연산 버스의 상호작용 패턴은 멤버들의 리스트로 구성되는 데, 멤버들 중에는 그 자체가 하나의 통신 서브 패턴을 구성하는 멤버(예, sbustype 또는 ubus-type 멤버) 들이 있다. 이 경우 몸체 부분에는 타입 이름만이 기술되므로 그 타입에 대한 패턴 정의를 담고 있는 선언부가 필요하다. 몸체의 타입 정의를 지원하기 위하여 이 서브 패턴의 타입(멤버 리스트)을 규정하는 부분이 지원 타입 선언부 <supporting_type_dcl>로 몸체의 멤버 변수로 되어 있는 버스 타입 멤버들의 상호

작용 패턴에 대한 선언으로 구성된다. 이렇게 서브 패턴에 대한 선언을 멤버 리스트로 변환하는 과정은 최종적으로 버스 타입이 멤버 리스트로만 표현될 때까지 반복된다. 이런 과정을 통해 하나의 버스 타입 선언부는 최종적으로 TOB 인터페이스에서 발생하는 연산 타입과 연산 데이타 타입 멤버의 순서화 된 집합으로 표현된다.

<st_bus_interface_spec>은 스트림 QoS 시맨틱 속성과 스트림 버스 타입을 선언한다. 스트림 버스 타입 선언은 헤더부와 몸체부로 나누어진다. 헤더부는 스트림 버스를 식별하며 몸체부는 스트림 패턴과 그 특성을 정의한다. 스트림은 하나 이상의 스트림 플로우로 구성되는데 각 플로우는 식별자와 방향을 가리키는 지시자, 그리고 QoS 속성들로써 표현된다. 플로우의 방향은 TOB 바인딩을 시작하는 측의 관점에서 정해지며, 'flow_in'과 'flow_out'의 두 값을 가질 수 있다. 'flow_in'은 스트림 플로우가 객체로부터 TOB 스트림 버스로 향하는 것을 의미하며, 'flow_out'은 그 반대를 표시한다. 스트림 플로우는 오디오, 비디오, 그리고 애니메이션과 같은 플로우 타입에 따라 적절한 QoS 값을 가질 수 있다.

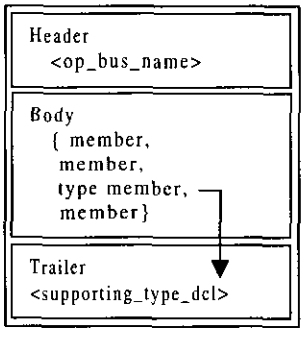


그림 4 연산버스 타입 선언부의 구조

### 5.2 연산 멤버 방향 지시자와 시간 관계

TOB 연산 버스 타입 선언 구문에서는 버스 타입 선언을 구성하는 각 멤버에 대해 선택적으로 정의되는 방향 지시자 <direction_specifier> ::= 'alt' 를 사용하여 멤버의 방향성과 시간 관계를 표현한다.

버스로의 전달 방향을 나타내는 지시자 'alt' 는 멤버의 전달 방향을 바로 이전에 발생한 멤버의 방향과 반전이 되게 하는 효과를 갖는다. 즉, 어떤 이벤트가 객체에서 TOB 방향으로 발생하고 있을 때 이후 발생하는 멤버 이벤트에 방향 지시자가 존재한다면, 그 이벤트는 TOB에서 객체 방향으로 송출되는 것으로 인식된다. 이 효과는 새로운 연산이 시작되거나 또 다른 방향 지시자가 사용되기 전까지 한 연산의 모든 멤버에게 그대로 유지된다.

또한 그림 5에 보인 바와 같이 방향 지시자의 도입은 버스로의 입출력 방향성 표시와 더불어 TOB 인터페이스에서의 발생 이벤트에 대한 시간적 관계인 동기 개념을 표현할 수 있도록 한다. 구문 멤버 앞의 'alt' 표시는 그 멤버 이벤트가 선행하는 멤버 이벤트보다 시간적으로 나중 시점을 표현하게 된다. 반대로 'alt' 표시가 없는 경우에는 그 시점까지의 모든 이벤트가 동일 방향 이벤트이고 그 이벤트들은 병렬적으로 처리됨을 표현한다. 따라서 'alt' 연산자의 존재는 전체 패턴의 시간 레벨의 수를 결정하게 한다.

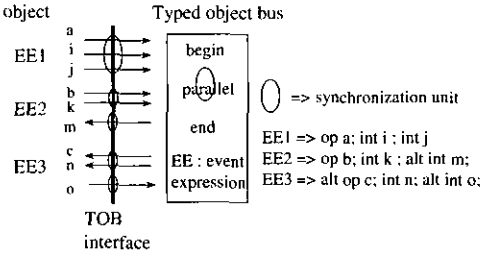


그림 5 'alt' 연산자를 이용한 멤버간 시간 관계의 표현

### 5.3 연산버스 타입 선언 예

본 절에서는 기본형과 중첩구조형, 연합구조형 연산버스 타입에 대한 선언 예를 기술한다.

**기본형 연산버스 타입**

opbustype opbus1 {op a; int b; int c; alt int d;}

위 구문은 멤버로 a라는 이름의 연산과 b, c 라는 정수 타입의 입력 연산 데이타를 갖고, 세 입력 이벤트의 처리 이후에 버스의 출력데이타로 정수 타입의 d라는 멤버 변수를 갖는 상호작용 패턴을 표현한다. 이 버스 타입의 시간 레벨은 2레벨 이다.

opbustype opbus2 {alt op a; int b; int c;}

위 구문은 'alt' 연산자가 op 타입 앞에 위치하므로, opbus1 버스 타입과 반대의 방향성을 갖는다. 연산의 이름을 연산 타입의 a라는 출력 멤버로, 버스의 출력 연산 데이타로 정수 타입의 b, c 라는 멤버를 갖는 상호작용 패턴을 표현한다.

**중첩구조형 연산버스 타입**

그림 6은 하나의 패턴 내에 서브 패턴으로서 앞서 정의된 기본형 버스 타입을 포함하는 선언 예를 보여 준다. 그림 6의 opbus3는 버스의 상호작용 패턴이 sbus-

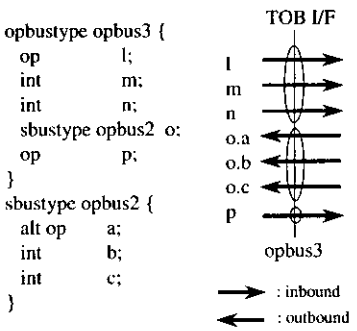


그림 6 중첩구조형 연산버스 타입의 선언과 발생 이벤트의 표현

type의 중첩된 서브 패턴을 갖고 있는 경우로, 서브 패턴 sbustype opbus2에서 'alt' 연산자가 사용되는 이벤트가 있기 때문에, 그 서브 패턴에서의 방향성 변화에 따라 시간적 레벨이 하나 증가하고 sbustype 서브 패턴의 종료 이후에 다시 방향성이 변화하므로 이 버스의 통신 패턴은 시간적으로 3 레벨로 구성된다. 처음의 시간적 레벨은 입력 이벤트로만 구성되는 l, m, n, o.a, o.b, o.c 이벤트가 하나의 그룹으로 묶여지는 레벨로, 출력 이벤트 o.a의 시간적 레벨보다 앞서 실행된다. 그리고 그 출력 이벤트 후에 다시 입력 이벤트 p가 실행되는 세 번째 시간적 레벨이 이어진다.

**연합구조형 연산버스 타입**

그림 7의 opbus4 타입은 버스의 통신 패턴이 내부적으로 ubustype의 상호작용 패턴을 하나의 멤버로 갖는

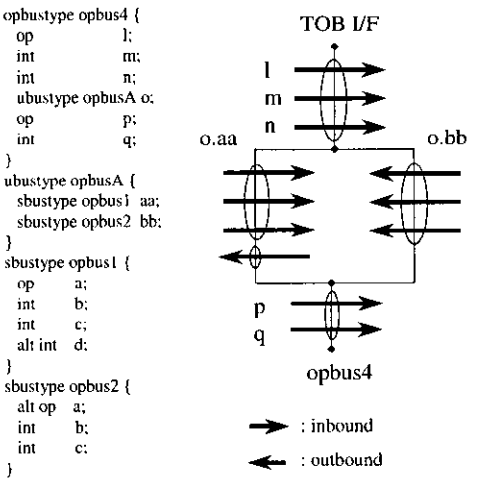


그림 7 연합구조형 연산버스 타입의 선언과 발생 이벤트의 표현

경우이다. 버스의 전체 상호작용 패턴을 결정하는 데 있어, ubustype 멤버는 여러 가지 서브 패턴 중의 하나를 선택하도록 한다. 이 구문 예에서는 ubustype 의 타입을 정의하는 <supporting_type_dcl>에서 opbusA 버스의 멤버 변수는 모두 sbustype 멤버 변수만으로 구성된다. opbustype 선언의 멤버 변수로 ubustype 타입 변수가 있으면 전체 패턴의 시간적 레벨은 ubustype의 선택 결과에 따라 가변적이다.

## 6. TOB 플랫폼의 기능 구조

### 6.1 TOB 플랫폼 컴포넌트

앞서 설명된 버스 개념과 상호작용 과정을 지원하기 위해 TOB 플랫폼에서는 여러 종류의 기능적 컴포넌트들이 필요하다. 그림 8은 TOB 플랫폼을 통한 상호작용에 필요한 기능적 컴포넌트들을 포함하는 플랫폼 구조를 보여주고 있다. 각 컴포넌트의 주요한 기능은 다음과 같다.

**TOB 플랫폼 관리자 (Platform Administrator)**

TOB 플랫폼 관리자는 전반적인 플랫폼의 구성과 운용을 책임지며 버스 관리자와의 상호작용을 통해 TOB 타입을 관리한다.

**버스 관리자 (Bus Manager: BM)와 버스 팩토리 (Bus Factory: BF)**

BM은 TOB 플랫폼에서 TOB 타입 규격과 TOB의 관리 기능을 수행한다. BM은 TOB 플랫폼 관리자에 대한 사용자 인터페이스를 제공하며, 레포지토리 서비스를 이용하여 TOB 타입 규격을 저장하고 관리한다. 또한 BM은 BF를 이용하여 TOB의 생성과 제거 기능 등을 수행한다. BF는 TOB의 생성을 담당하며, BM 내부의 모듈 또는 별도의 컴포넌트로 구현될 수 있다.

BF가 생성한 TOB를 클라이언트와 서버에서 사용할 수 있도록 BM은 생성된 TOB에 대한 정보를 트레이더에 등록한다. TOB의 생성과 제거 과정에서 트레이더와의 상호작용은 동시에 이루어지는데 TOB는 생성되면서 자동적으로 트레이더에 등록되고, 제거와 더불어 트레이더에서 정보가 삭제된다. TOB에 관한 정보는 레포지토리를 이용하여 저장되고 관리된다.

**버스 어댑터 (Bus Adapter: BA)**

BA는 클라이언트와 서버 객체의 TOB 바인딩 과정을 대행하는 프록시(proxy) 객체로 클라이언트나 서버 객체에게는 플랫폼 사용을 위한 바인딩 관련 API (Application Programming Interface)를 제공하고 내부적으로는 트레이더 객체 및 버스 객체와의 상호작용을 통해 필요한 트레이딩과 TOB 바인딩 동작을 처리한다.

바인딩 지연을 줄이기 위하여 BA는 임포트(import) 정보를 위한 캐싱(caching) 기능을 갖는다.

**트레이더(Trader)**

트레이더는 일반적인 객체 트레이딩(trading) [11]과 TOB 트레이딩 기능을 수행한다. TOB 트레이딩을 위한 인터페이스는 객체 트레이딩을 위한 인터페이스와 분리되어 있으며, 이것은 TOB 트레이딩에만 부과되는 제약이나 정책을 별개로 수립할 수 있도록 하기 위함이다. TOB 트레이딩은 TOB의 타입 특성 값과 검색 조건을 이용하여 이루어지며, 조건과 일치하는 TOB가 존재할 때 해당하는 레퍼런스가 반환된다. 트레이더는 활성화 되어 있는 TOB에 대한 정보, 즉 TOB 레퍼런스 및 해당 TOB에 바인딩 되어 있는 클라이언트 및 서버 객체 레퍼런스 정보를 유지한다.

**레포지토리 (Repository)**

레포지토리는 각종 타입과 인스턴스 정보의 영속적인 저장 기능을 제공한다. 일반적으로 레포지토리 서비스 [12] 는 타입 규격에 관한 저장소로 작용하는 규격 (또는 인터페이스) 레포지토리와 구현 객체에 대한 정보를 저장하는 구현 레포지토리가 있다. 이 저장 서비는 사용자 객체에게 정보의 저장(store)과 검색(retrieval)에 대한 서비스를 제공하며, 실제 구현 형태에 있어서는 디렉토리 시스템이나 데이타베이스를 이용하여 이 서버 기능을 대치할 수도 있다.

**TOB**

TOB는 2개 이상의 버스 프로토콜 머신(Bus Protocol Machine: BPM)의 결합관계에 의해 만들어진다. BPM은 데이타 송수신, 런타임 타입 검사, 연산 순서의 유지 등의 역할을 담당한다. BPM은 구현하고 있는 TOB의 타입에 일치되도록 동작하며, TOB 바인딩에 참가하는 역할에 따라 클라이언트 또는 서버로서 행동한다.

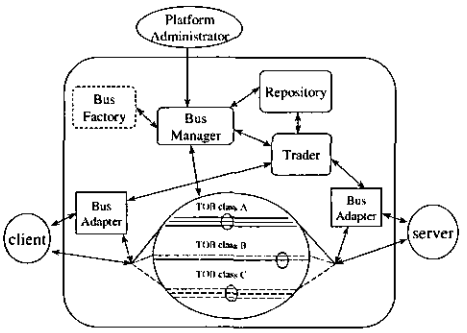


그림 8 TOB 플랫폼의 기능 구조

### 6.2 동작 시나리오

그림 9는 앞서 정의된 기능 컴포넌트를 이용하여 클라이언트 객체와 서버 객체가 통신하는 간단한 동작 시나리오를 표현하고 있다. 이 시나리오에서 클라이언트 객체가 TOB 바인딩을 개시하며, 서버 객체는 서버측 BA에 의해 TOB에 미리 바인딩 되어 있다고 가정한다. 이에 앞서 TOB TDL을 이용하여 TOB 타입이 정의되어야 한다. 정의된 TOB 타입은 TOB TDL 컴파일러에 의해 컴파일되며, 결과로서 TOB 헤더 파일과 스터브가 생성된다. 응용의 클라이언트와 서버측 실행 코드는 객체의 스터브가 아니라 해당 TOB의 스터브와의 결합을 통해 생성된다.

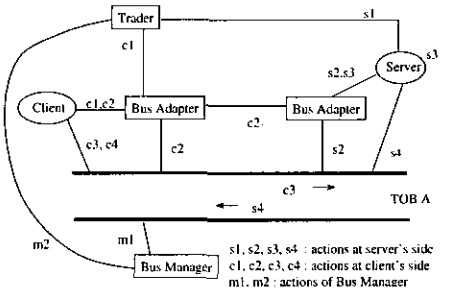


그림 9 TOB 플랫폼의 동작 시나리오

**버스 관리자의 동작**

m1 : 버스 관리자는 TOB 타입 정보를 저장하고 관리하며 TOB를 생성한다.

m2 : 생성된 TOB를 클라이언트와 서버 객체가 바인딩 할 수 있도록 트레이더를 통해 등록한다.

**클라이언트의 동작**

c1: 클라이언트 객체는 버스 어댑터를 통하여 바인딩하기를 원하는 TOB 및 서버 객체의 레퍼런스를 트레이더로부터 얻는다.

c2: 버스 어댑터는 클라이언트 객체를 TOB에 바인딩한다.

c3: 클라이언트 객체는 서버 객체로 서비스를 요구한다.

c4: 서비스 실행 결과가 버스를 통해 클라이언트 객체로 반환된다.

**서버의 동작**

s1: 서버가 클라이언트들에게 서비스를 제공하고자 하면, 서버 관리자나 서버는 트레이더를 통해 객체를 등록한다.

s2: 버스 어댑터는 서버 객체를 TOB에 바인딩 한다.

s3: 서비스 요구가 TOB를 통해 전달되면 요구 매개변수를 입력 받아 자신의 구현 메쏘드를 실행한다.

s4: 메쏘드의 반환 결과가 있으면 그 결과를 버스에 싣는다.

## 7. 요약 및 향후 계획

본 논문에서는 객체간 상호작용의 통신특성의 타입에 따라 차별화 된 통신경로를 이용하여 분산통신 서비스를 제공하는 Typed Object Bus (TOB)에 기반한 새로운 개념의 분산플랫폼을 제안하였다. TOB는 연산동작 특성과 자신을 통해 전달될 수 있는 데이타 타입, 그리고 객체간 통신에 제약을 가하는 여러 가지 속성들에 의해 표현된다. TOB 플랫폼은 이러한 TOB들을 이용하여 플랫폼 수준에서 구조적으로 객체통신을 제어하고 지원한다. 표 1에 TOB 플랫폼과 기존 플랫폼 모델들을 비교하여 정리하였다.

본 논문은 CORBA나 DCOM과는 다른 새로운 방식의 분산통신 서비스의 제공을 위하여 TOB의 개념을 정의하고 TOB 플랫폼의 구조와 TOB TDL을 설계하였다. TOB TDL 문법에는 통신경로의 QoS 지원을 위한 기본적인 틀이 마련되어 있으나 자세한 내용은 정의되어 있지 않다. 향후 우선순위, 지연, latency등의 연산 QoS와 다양한 스트림 QoS 를 구체화하고 세밀하게 규격화 하는 연구를 추진할 예정이며 TOB TDL 컴파일러를 개발할 것이다. 또한 정의된 QoS를 TOB 플랫폼 기반에서 지원하기 위한 방안을 연구하고 이를 구현할 것이다.

표 1 기존 분산플랫폼 모델들과 TOB 플랫폼과의 비교

| 항목 \ 플랫폼 모델 | CORBA/DCOM | OLAN 커넥터 모델 | ODP 바인딩 객체 | TOB 플랫폼 |
|---|---|---|---|---|
| 통신서비스의 특성 | 모든 응용에 대해 획일화 된 통신 제공 | 커넥터에서 프로시저호 모델과 메시지 모델의 2종류 통신모델 제공 | 각 바인딩 객체는 통신구성, 지연, 지터 등의 바인딩 특성을 제공 | 응용에 따라 차별화된 통신패턴과 특성을 갖는 TOB 제공 |
| 통신데이터의 타입검사 | 객체의 스터브를 이용한 정적인 타입 검사 | 커넥터에 상호작용에 대한 연산구조 및 데이터 타입이 규정되어 있지 않아 응용 모듈에서 타입검사 | 객체의 스터브를 이용한 정적인 타입 검사 | TOB에서 런타임으로 타입 검사 |
| 분산객체간 상호작용의 액세스 관리 | 서버객체에서 액세스 제어 | 서버객체에서 액세스 제어 | 서버객체에서 액세스 제어 | 'access control list'에 따라 TOB에서 액세스제어 |
| 인터페이스 기술방법 | IDL을 이용하여 객체의 인터페이스 정의 | MIL을 이용하여 커넥터의 규격을 정의 | IDL을 이용하여 객체의 인터페이스 정의 | TDL을 이용하여 TOB 규격을 정의 |

## 참 고 문 헌


[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, July 1995.

[2] Brown, N. and Kindel, C., *Distributed Component Object Model Protocol DCOM/1.0*, Microsoft, 1996.

[3] Prieto-Diaz, R. and Neighbors, J.M., Module interconnection languages, *The Journal of Systems and Software*, 6(4):307-334, Nov. 1986.

[4] Purtilo, J. M., The POLYLITH software bus, *ACM TOPLAS*, Vol.16(No.1), pp.151-174, Jan. 1994.

[5] Bellissard, L., Atallah, S.B. Boyer, F., and Riveill, M., Distributed application configuration, *in* Proc. of the 16th IEEE International Conference on Distributed Computing System (ICDCS'96), Hong Kong, April 1996.

[6] Magee, J., Dulay, N., and Kramer, J., A constructive development environment for parallel and distributed programs, *in* Proc. of the International Workshop on Configurable Distributed systems, Pittsburgh, March 1994.

[7] Allen, R. and Garlan, D., *Formal Connectors*, Technical Report CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 1994.

[8] ITU-T Rec. X.901, *Reference Model of Open Distributed Processing Part 1: Overview*, International Telecommunication Union, 1995.

[9] ITU-T Rec. X.903, *Reference Model of Open Distributed Processing Part 3: Architecture*, International Telecommunication Union, 1995.

[10] ITU-T Recommendation X.920, *Open Distributed Processing Interface Definition Language*, International Telecommunication Union, 1997.

[11] ITU-T Rec. X.952, *Open Distributed Processing Trading Function: Provision of Trading Function Using OSI Directory Service*, International Telecommunication Union, 1997.

[12] Object Management Group, *CORBAservices - Persistent Object Service*, December 1997.


## 부록. TOB TDL 문법

<tobtdl_spec>::=<bus_template_header> <bus_template_description>

<bus_template_header>::= <bus_template_name_part>

<bus_template_description>::=<bus_context_spec_part>";"<bus_interface_spec_part> <bus_property_spec_part>";"

<bus_template_name_part>::="bus_template" <identifier>";"

```
<bus_context_spec_part>::=
    <bus_context_name_part>"{"<bus_context_spec>"}"
<bus_context_name_part>::="bus_context"  <identifier>
<bus_context_spec>::= <topology_spec> ";"
<topology_spec>::= "(" <topology_type_specifier> ":"
    <topology_type> ")"
<topology_type_specifier> ::=  "topology"
<topology_type> ::=  "ptp"  | "mp"

<bus_property_spec_part>::=<bus_property_name_part>
    "{" <bus_property_spec> "}"
<bus_property_name_part>::= "bus_property"  <identifier>
<bus_property_spec>::=
     <access_control_dcl> ";" <availability_dcl> ";"
     {<other_property_dcl> ";"}
<access_control_dcl>::= "(" <access_control_specifier> ":"
    <access_control_list> ")"
<access_control_specifier>::=  "access_ctrl"
<access_control_list>::=<string_literal> {"," <string_literal>}
<availability_dcl>::= "(" <availability_type_specifier> ":"
    <availability_type> ")"
<availability_type_specifier>::=  "availability"
<availability_type>::= "cold_stby" | "warm_stby" | "hot_stby"
<other_property_dcl>::= "(" <property_type_specifier> ":"
    <property_type> ")"
<property_type>::= <string_literal>

<bus_interface_spec_part>::=<op_bus_interface_spec>
    | <st_bus_interface_spec>
<op_bus_interface_spec>::=
     <op_bus_dcl><op_semantic_dcl>" ; "
<op_bus_dcl>::= <op_bus_specifier>
     "{" <op_bus_spec> "}" ";"  [<supporting_type_dcl>]

<op_semantic_dcl>::= "("<op_semantic_type_specifier>
    ":" <op_semantic_type>")"
<op_semantic_type_specifier>::= "op_sem"
<op_semantic_type>::= "at_least_once" | "best_effort"

<op_bus_specifier>::= "opbustype" <op_bus_name>";"
<op_bus_name> ::= <identifier>
<op_bus_spec>::=<operation> {<op_member>}
    {<operation>{<op_member>}}
<operation> ::= [<direction_specifier>] <op_specifier>
<direction_specifier>::= "alt"
<op_specifier>::= <op_type_specifier> <op_name> ";"
<op_type_specifier>::= "op"
<op_name>::= <identifier>
<op_member>::=[<direction_specifier>] <op_member_type_
    specifier><member_name>";"
<op_member_type_specifier>::= <simple_data_type_specifier>
    | <constr_data_type_specifier>
    | <sub_bus_type_specifier>
    | <union_bus_type_specifier>
    | <op_type_specifier>
<simple_data_type_specifier>::= <integer_type>
    | <char_type>
    | <floating_pt_type>
    | <octet_type>
    | <boolean_type>
<constr_data_type_specifier> ::= <struct_type>
    | <union_type>
    | <enum_type>
<sub_bus_type_specifier> ::= "sbustype" < op_bus_name>
<union_bus_type_specifier>::= "ubustype" <op_bus_name>
<member_name>::= <identifier>
<supporting_type_dcl>::={<sub_bus_type_dcl>|<union_bus_
    type_dcl>}
<sub_bus_type_dcl>::=<sub_bus_type_specifier>
     "{"<op_bus_spec>"}" ";"
<union_bus_type_dcl>::=<union_bus_type_specifier>
    "{"<union_member_list>"}" ";"
<union_member_list>::=<union_member><union_member>
    {<union_member>}
<union_member>::=<sub_bus_type_specifier><member_
    name>
   |<union_bus_type_specifier><member_name>";"

<st_bus_interface_spec>::=<stqos_semantic_dcl>";"
    <st_bus_dcl>";"
<stqos_semantic_dcl>::="("<stqos_semantic_type_specifier>":"
   <stqos_semantic_type>")"
<stqos_semantic_type_specifier> ::= "stqos_sem"
<stqos_semantic_type>::= "guaranteed" | "best_effort"

<st_bus_dcl>::="<st_bus_specifier>" "{"<st_bus_spec>"}"
<st_bus_specifier>::= "stbustype" <st_bus_name>
<st_bus_name>::= <identifier>
<st_bus_spec>::= <st_member> "{"<st_member>"{"
<st_member>::=<flow_dir><flow_name> "("<flow_type>")" ";"
    [<qos_dcl>]
<flow_dir>::= "flow_in" | "flow_out"
<flow_name>::= <identifier>
<flow_type>::= "audio" | "video" | "animation"
<qos_dcl>   ::=  "with" "("<qos_spec>")" ";"
<qos_spec>::=<qos_param_type>":" <qos_param_value>
    {"," <qos_param_type> ":" <qos_param_value>}
<qos_param_type>::= <string_literal>
<qos_param_value>::= <string_literal>
```

김 상 경

1985년 고려대학교 전자공학과 학사. 1987년 고려대학교 대학원 전자공학과 석사. 1989년 ~ 현재 한국통신 선임연구원. 1998년 ~ 현재 고려대학교 대학원 전자공학과 박사과정. 관심분야는 네트워킹 구조, 분산플랫폼 구조, Wireless Ad Hoc 네트워크임.

선 경 섭

1985년 고려대학교 전자공학과 학사. 1987년 고려대학교 전자공학과 석사. 1987년 ~ 1994년 한국통신 연구센터. 1998년 8월 고려대학교 전자공학 공학박사. 관심분야는 정보망 구조, 분산시스템, 네트워킹 아키텍쳐등임.

안 순 신

1973년 서울대학교 공과대학 졸업(B.S). 1975년 한국과학기술원 전기 및 전자과 졸업(M.S). 1979년 블런서 ENSEEIHT 에서 공학박사 취득(Ph.D). 1979년 3월 ~ 1982년 8월 아주대학교 전자과 교수. 1991년 1월 ~ 1992년 2월 NIST(National Institute of Standard and Technology) 방문연구원. 1982년 ~ 현재 고려대학교 전자공학과 교수. 관심분야는 컴퓨터 네트워크 및 분산 시스템임.