CORBA 표준의 정형명세와 증명

김 미 희†

요 약

CORBA 표준명세는 표준을 만족하는 구현에서 제공해야 할 기능뿐만 아니라 서비스 제공 모듈의 사용자 인터페이스도 IDL을 사용하여 엄격하게 정의하고 있다. CORBA 표준에 대한 확신과 신뢰성을 가지기 위해서는 IDL(Interface Definition Language)로 기술된 표준명세를 정형화하고 수학적으로 엄격히 증명할 필요가 있다. 본 논문에서는 CORBA 표준을 정형적으로 명세하고 검증할 방법을 제시한다. 먼저 표준모듈을 Larch/CORBA IDL(LCB)를 사용하여 정형적으로 명세하고, LCB의 의미론에 준하여 LCB 명세를 LSL(Larch Shared Language)로 변환한다. 변환한 LCB 명세와 LSL 증명논리를 사용하여 특성을 수학적으로 증명한다. 변환기반의 LCB 의미론을 정립하여 제안한 방법의 이론적 바탕을 마련하고 CORBA 이름서비스명세에 실제 적용하여 그 효용성을 보인다.

A Formal Specification and Verification of CORBA Standards

Mi-Hee Kim†

ABSTRACT

The CORBA standard specification is peculiar in that, in addition to the usual description of architecture and functionality of distributed systems, it also specifies in CORBA IDL(Interface Definition Language) the precise application interfaces to be provided by implementation modules to conform to the standard. However, it is necessary to formally specify the behavioral semantics of the interfaces and to formally verify and reason about their properties. A formal specification and verification will ensure the correctness of CORBA specification, and thus will increase our confidence on the standard. We propose a formal specification and verification method for CORBA standard specification. We first specify CORBA standard modules in Larch/CORBA IDL (LCB), a Larch behavioral interface specification language tailored to CORBA IDL. The LCB specifications are translated into LSL(Larch Shared Language), an algebraic specification language, based on the formal semantics of LCB. An actual mathematical verification is accomplished in terms of LSL using its proof logic and support tools. We also provide the semantic foundation of our method and apply it to the CORBA naming service specification to show its effectiveness and efficiency.

1. 개 요

CORBA는 OMG(Object Management Group)에서 제정한 분산객체시스템 표준으로[11], 객체 상호간의 서비스 요구와 서비스 제공에 대한 브로커 역할을 하는 분산객체간의 통신을 위한 미들웨어이다. 즉, 객체로부터 서비스 요구를 받아 해당 객체를 찾고 관련 함수를 호출한다. 매개값과 결과값을 프로그래밍언어와 구현 시스템에 비종속적으로 전달하기 위하여 자료값도 변환한다.

CORBA 표준명세의 특징은 분산객체시스템의 구조

† 정 회 원 : 한국전자통신연구원 정보고기술연구본부 정보화기반연구부
논문접수 : 1998년 7월 13일, 심사완료 : 1998년 10월 19일

와 기능뿐만 아니라 구현모듈의 사용자 인터페이스도 IDL(Interface Definition Language)를 사용하여 엄격하게 정의하고 있다는 것이다. CORBA IDL은 객체의 인터페이스를 정의하기 위한 프로그래밍언어에 비종속적인 선언 언어이다[11]. 상이한 프로그래밍언어로 구현된 객체도 IDL를 통하여 상호 운용할 수 있다. CORBA 표준이 잘 정의되었고 이에 대한 확신과 신뢰성을 가지기 위해서는 IDL로 기술된 표준명세를 정형화하고 수학적으로 엄격히 증명할 필요가 있다. CORBA 표준에 정형방법을 적용하면 많은 혜택을 받을 수 있는데, 대부분의 좋은 표준이 제공해야 할 대표적 요건이라고 할 수 있는 증명성과 정확성을 제공할 수 있다[2] [5] [8].

본 논문에서는 CORBA 표준을 정형적으로 명세하고 검증하는 방법을 제시한다. 특히, IDL로 기술된 CORBA 표준모듈의 특성을 정리증명기와 같은 지원도구를 사용하여 기계적으로 분석하고 수학적으로 증명할 수 있는 방법을 소개한다. 정형증명을 위해서는 먼저 표준을 정형적으로 명세하여야 한다. 정형명세는 정형증명의 이론적 배경을 제공할 뿐만 아니라 CORBA의 궁극적 목표인 재사용, 투명성, 이식성, 상호 운용성을 실현하기 위한 필수요소다. 부품의 행위가 모호하게 기술되어서는 사용자의 요구사항과 표준에 대한 만족성을 판단할 수 없기 때문이다. CORBA 표준의 정형화는 크게 CORBA 객체모형을 위한 수학적 토대정립과 CORBA 모듈의 인터페이스와 행위에 관한 정형적 명세로 나눌 수 있다. 수학적 토대정립은 CORBA 객체모형을 정형표기법을 사용하여 기술하고 정형의미를 부여하는 것을 말하며, Z 표기법을 사용한 정형화가 시도되고 있다[2] [8]. CORBA모듈의 행위명세는 현재 CORBA IDL을 사용하여 기술된 사용자 인터페이스와 제공 서비스를 엄격하게 정의하는 것을 말한다. Larch/CORBA IDL (LCB) [9]를 사용하여 CORBA이름서비스 모듈을 명세한 예를 찾아 볼 수 있다[1].

본 논문에서 제안한 증명방법의 특징은 CORBA 표준의 핵심개념이나 바탕이론에 대한 증명이 아니라 서비스모듈의 인터페이스와 행위에 관한 특성 검증이라는 것이다. 모듈명세는 LCB로 주어진다고 가정한다. 기본 아이디어는 LCB 명세를 대수명세언어 LSL(Larch Shared Language)[6, 제4장]로 변환하고 LSL를 사용하여 특성을 증명하는 것이다. LSL은 잘 정의된 증명논리와 지원도구를 갖추고 있다[6, 제7장]. LCB 명세의 변환을 위하여 LCB 의미모형을 LSL로 정의하고 의미모형을 이용하여 변환규칙을 정의한다. 제안한 방법의 평가를 위하여 [1]에서 명세한 CORBA 이름서비스모듈의 일부를 새로운 방식으로 명세하고 중요한 특성을 실제로 증명해 본다.

본 논문은 다음과 같이 구성된다. 먼저 제2절에서는 이름서비스모듈의 일부를 LCB로 명세한다. 모형변수와 주석이라는 새로운 개념을 도입한다. 제3절에서는 변환기반의 LCB 의미론을 바탕으로 하여 LCB 명세를 LSL로 변환하여 증명하는 새로운 검증방법을 제안하고 이론적 토대를 정립한다. 제4절에서는 제안한 방법을 이름서비스모듈의 특성 증명에 실제로 적용한다. 마지막으로 제5절에서는 본 연구의 기여 및 향후연구와 함께 결론을 맺는다.

2. CORBA 모듈의 정형명세

CORBA표준모듈의 정형증명에 앞서, 본 절에서는 먼저 CORBA 모듈의 일부를 정형적으로 명세한다. 선택한 예제는 이름서비스이다[12]. 이름서비스는 객체에 이름을 부여하는 것에 대한 CORBA 표준이다. CORBA 이름서비스는 UNIX 파일 시스템과 유사한 트리 구조를 갖는다. 이름은 항상 '이름문맥(naming context)'이라 불리는 객체에 상대적으로 정의되며, 이름 문맥객체에도 이름이 부여될 수 있다. 따라서 이름서비스는 '이름그래프(naming graph)'라 불리는 트리 구조를 형성한다. 본 논문에서는 이름문맥의 일부분을 정형화하고 그 특성을 증명한다.

프로그램 모듈을 정형적으로 명세하기 위한 연구가 활발히 진행되고 있으며[10], 특히 CORBA IDL를 확장한 몇몇 명세언어가 제안되고 있다[9]. 본 논문에서는 Larch 방식을 채택하여[6], CORBA IDL과 Larch 인터페이스 명세기법을 접목한다. Larch 방식의 특징은 양층형으로, 추상모형을 정의하는 부분과 인터페이스를 정의하는 부분으로 구분된다[4] [6]. 추상모형은 일반적으로 시스템의 수학적 모형을 제공하며 LSL이라 불리는 대수명세언어로 정의한다[6, 제4장]. 프로그램모듈이 제공하는 함수나 객체의 의미를 정의하는 인터페이스 부분은 추상모형을 사용하여 프로그래밍 언어에 짜맞추어진 인터페이스 명세언어로 작성한다. 이를 위하여 Hoare 방식의 선·후조건문을 사용한다[7]. Larch와 CORBA IDL를 접목한 명세언어를 Larch/CORBA IDL (LCB)이라 부른다[9].

```
typedef string Istring;
struct NameComponent {
  Istring id;
  Istring kind;
};
typedef sequence NameComponent Name;

interface NamingContext {
  //@ spec SetName ctx;
  //@ spec FiniteMapName,Object om;
  //@ inv ∀ n: Name •
  //@ inv  ctx ∩ dom(om) =Ø∧
  //@ inv  (len(n)>1∧n∈ctx∪dom(om) ⇒init(n) ∈ctx);
  // rest of defs
};
```

(그림 1) NamingContext 클래스의 추상 모형
(Fig. 1) An abstract model of class NamingContext

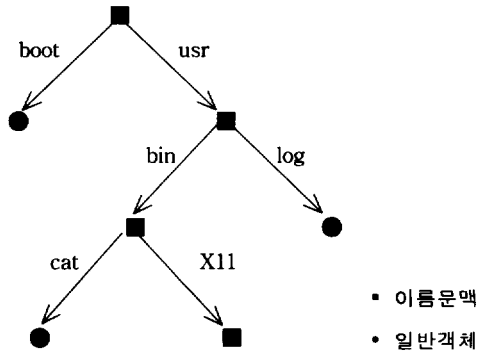
2.1 추상모형

(그림 1)은 이름문맥 클래스 NamingContext를 정의한 일부이다. 참고로, CORBA IDL에서는 클래스를 인터페이스(interface)라 부른다. 명세는 CORBA IDL에 특수한 형태의 프로그램 주석으로 첨부된다. 즉, //@로 시작하는 줄은 LCB 명세를 나타낸다. 먼저 이름(Name)이란 이름구성요소(NameComponent)라는 구조체의 나열을 말한다. 여러 이름구성요소를 가지는 이름을 '복합이름(complex name)'이라고 한다.

선언된 변수 ctx와 om은 실제 구현에 나타나는 프로그램변수가 아니라 NamingContext의 추상값을 나타내기 위한 '모형변수(model variable, ghost variable)'이다. 즉, NamingContext 객체의 추상값은 쌍 [ctx, om]으로 나타낸다. 모형변수 ctx는 현 이름문맥을 루트로하는 서브 트리에 포함된 모든 이름문맥의 이름을 나타낸다. 반면, om은 해당 서브트리에 포함되는 모든 객체와 이름의 쌍을 나타낸다. 즉, 이름에서 객체로의 유한사상이다. 예를 들면, 이름그래프가 (그림 2)와 같이 주어졌을 때, 최상위 이름문맥 객체의 추상값은 다음과 같다. (이름열은 <>로 나타낸다.)

(그림 1)의 명세에서 **inv** 절은 이름문맥 객체가 항상 만족해야 할 불변진리(invariant)를 나타낸다. 동일한 이름이 객체와 이름문맥을 동시에 나타낼 수 없으며, 이름그래프는 트리구조여야 한다. 논리문에서 len은 이름의 길이를, dom은 사상의 정의구역을, init은 마지막 이름구성요소를 제외한 나머지 구성요소로 만들어지는 서브이름을 나타낸다. 이들은 LSL 명세 Sequence와 FiniteMap에 정의되어 있다 [6, 부록 A].

[{<usr>, <usr, bin>, <usr, bin,X11>},
 {(<boot>, •), (<usr, log>, •), (<usr, bin, cat>, •)}]



(그림 2) 이름그래프의 예
(Fig. 2) An example of naming graph

2.2 연산자 명세

CORBA 표준에서는 이름문맥 객체를 조작하기 위한 다양한 멤버함수를 정의한다. 정의된 함수를 이용하여 특정 객체에 이름을 부여하고, 주어진 이름을 가진 객체를 찾고, 이름을 취소하거나 재부여할 수도 있으며, 이름문맥내의 모든 객체를 순차적으로 접근할 수도 있다. 본 논문에서는 이 중 핵심기능인 이름부여와 탐색에 관한 멤버함수를 명세한다.

객체에 새로운 이름을 부여하는 것을 '이름결합'이라 부르는데, 이름결합은 항상 특정한 이름문맥에 상대적으로 이루어 진다. 따라서 **이름결합 함수 bind**는 이름문맥 클래스의 멤버함수로 정의되며, 그 명세는 다음과 같다.

```
void bind(in Name n, in Object o)
  raises(NotFound, CannotProceed, InvalidName,
         AlreadyBound);
//@ behavior {
//@   requires len(n)>0∧n∉ctx∪dom(om^)
//@            ∧ (len(n)>1⇒init(n)∈ctx);
//@   modifies om
//@   ensures om'=update(om^, n, o);
//@ }
```

함수의 정형적 의미는 선후조건문을 사용하여 **behavior**절에 명세한다. 함수명세가 의미하는 바는 **requires**절의 선조건문이 만족한 상태에서만 해당 함수가

호출될 수 있으며, 함수평가는 반드시 종결되어야 하고 종결상태에서는 **ensures**절의 후조건문이 만족되어야 함을 뜻한다. 프레임공리(frame axiom)를 나타내는 **modifies** 절은 어떤 객체의 값이 변화될 수 있는가를 나타낸다. 선조건문은 이름이 유효하고, 이미 다른 객체 혹은 이름문맥에 의해 사용되지 않아야 하며, 이름그래프 상의 경로를 나타내어야 한다는 뜻이다. 논리문의 om^은 om의 초기값을 나타내고 om'은 종결값을 나타낸다. 후조건문은 om의 종결값이 초기값에 새로운 사상원소 (n,o)를 추가한 것과 같아야 한다는 뜻이다. LSL 명세 FiniteMap에 정의된 update는 사상의 변경, 즉 새로운 쌍 (n,o)를 추가함을 의미한다 [6, 부록 A].

함수 bind는 예외상황 NotFound, CannotProceed, InvalidName, AlreadyBound를 일으킬 수 있다. 예외상황은 내장논리문 **raised**로 명세할 수 있으나 편의상 생략하였다 [4] [9].

특정한 이름과 결합된 객체를 찾는 것을 '이름분해(name resolution)'라 부르는데, 이름분해도 항상 문맥에 상대적으로 이루어 진다. 복합이름의 경우, 마지막을 제외한 모든 이름구성요소는 이름문맥을 나타내며, 마지막 구성요소는 해당 이름에 결합된 객체를 나타낸다.

```
Object resolve(in Name n)
  raises(NotFound, CannotProceed, InvalidName);
//@ uses Helper(Set<Name>, FiniteMap<Name, Object>);
//@ behaviour {
//@   requires len(n)>0 ∧ n∈dom(om)) ;
//@   ensures result=apply(om, n);
//@   requires len(n)>0 ∧ n∈ctx;
//@   ensures result=[n⊗ctx, n⊗om)];
//@ }
```

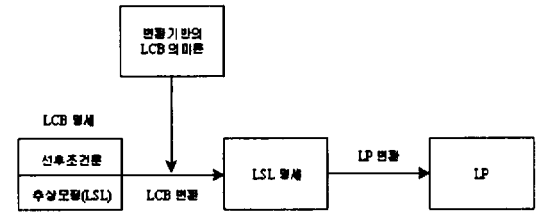
먼저 **uses**절은 해당 함수명세가 LSL 명세 Helper를 사용한다는 뜻이다. LSL 명세 Helper는 두 번째 후조건문에 사용한 연산자 ⊗를 수학적으로 정의한다 ((그림 3) 참조).

```
Helper(NS, OM): trait
includes Set(NS,Name), FiniteMap(OM,Name,Object)
introduces
_⊗_: Name, NS → NS
_⊗_: Name, OM → OM
asserts
  forall n,m: Name, o: Object, s: NS, om: OM | len(m) > 0 •
    n ⊗ Ø == Ø;
    n ⊗ insert(s,n || m) = insert(n⊗s, m);
    n ⊗ empty == empty;
    n ⊗ update(om,n || m,o) == update(n⊗om, m, o);
```

(그림 3) 연산자 ⊗의 정의
(Fig. 3) A formal definition of operator ⊗

함수 resolve는 복수개의 **requires/ensures**절로 명세 되었다. 객체와 결합된 이름에 대해서는 해당 객체를 되돌려 주고 (첫 번째 **requires/ensures**절), 그렇지 않고 이름문맥과 결합되었으면 해당 이름문맥을 되돌려 준다 (두 번째 **requires/ensures**절). 그 밖의 경우에는 NotFound, CannotProceed, InvalidName과 같은 예외상황을 일으킨다 (명세하지 않음). LCB 예약어 result는 결과값을 나타낸다. LSL 함수 apply는 사상의 적용을 나타내는데, LSL 명세 FiniteMap에 정의되었다 [6, 부록 A]. LSL 연산자 는 해당 이름문맥을 루트로 하는 서브이름그래프의 추상값을 계산한다 ((그림 3) 참조). 즉, 다음과 같은 의미이다. (연산자 ||은 이름열 연결(concatenation)을 나타낸다.)

n⊗ctx = {m: Name | len(m)>0∧n || m∈ctx • m}
n⊗om = {m: Name, o: Object | en(m)>0∧(n || m,o)∈om • (m, o)}



(그림 4) 증명전략
(Fig. 4) A formal verification of LCB specification

3. 정형증명

3.1 기본전략

LSL은 잘 정의된 증명논리와 LP(Larch Proof Assistant)라 불리는 정리 증명기를 가지고 있다[6, 제7장]. 만약 의미손실 없이 LCB 명세를 LSL로 변환할 수 있다면, 인터페이스 명세의 검증에 LSL 증명논리와 지원도구를 사용할 수 있다. 특히, 다음과 같은 LCB의 특징은 이러한 접근방법을 가능하게 한다.

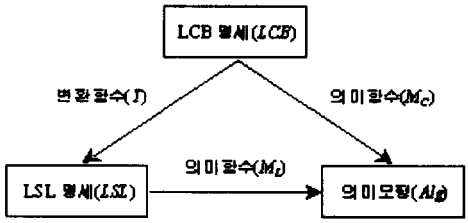
- LCB 명세는 LSL로 작성한 추상모형과 선후조건문 형태의 인터페이스 명세로 구성된다.
- LCB의 정형의미를 LCB에서 LSL로의 변환규칙으로써 정의할 수 있다.

변환규칙이 정의되면 LCB 명세의 구문_ 특히, 선후조건문을 추상모형을 사용하여 LSL 의미로 변환할 수 있다((그림 4) 참조). 일단 LSL 명세가 생성되면 LSL 기반의 증명논리와 필요하다면 지원도구 LP를 사용할 수 있다. 따라서 제안하는 증명방법은 먼저 해당 명세를 변환방식의 의미론을 바탕으로 하여 LSL로 변환한다. 증명하고자 하는 특성을 LSL 형태로 정형화하고 LSL로 변환된 LCB 명세 위에서 LSL 논리를 사용하여 정형적으로 증명한다. 제안한 증명방법의 핵심은 LCB 명세를 LSL로 변환하는 것이며 변환의 이론적 토대는 LCB 정형의미론이 제공한다(제3.3절 참조).

3.2 증명방법

변환기반의 LCB 의미론이 정립된 후라면 LCB 명세의 증명은 다음과 같은 단계를 거쳐서 수행할 수 있다(적용 사례는 제4절 참조).

(1) 대상이 되는 LCB 명세를 변환규칙에 준하여 LSL로 변환한다.
(2) 증명하고자 하는 특성을 기술하고 (1)의 변환결과로 얻어진 LSL 어휘를 사용하여 표현한다. 관심사가 되는 특성은 궁극적으로 LSL 논리로 표현 가능하여야 한다.
(3) LSL 증명논리를 사용하여 특성을 증명한다. 변환된 LSL 명세뿐만 아니라 변환규칙 정의에 필요한 여러 LSL 명세도 사용된다(제 3.3절 참조).
(4) 경우에 따라서는 모든 LSL 명세를 LP 입력형태로 변환하여 LP를 사용하여 기계적으로 증명할 수도 있다.[1)]



(그림 5) 변환기반의 LCB 의미론
(Fig. 5) A translation approach to LCB semantics

1) LP는 LSL를 위한 정리 증명기이지만 대화식 증명을 지원하기 때문에 입력양식과 문법에 다소 차이가 있다.

3.3 LCB의 정형의미론

제 3.2절에서 제안한 증명방법의 이론적 바탕은 LCB 의미론이 제공한다. LCB 정형의미를 LCB에서 LSL로의 변환규칙을 정의함으로써 간접적으로 정의할 수 있다. 이러한 접근방식을 '변환기반 정형의미론'이라 부른다. (그림 5)는 변환기반 LCB 의미론을 도식화하였다. *LCB*와 *LSL*를 각각 가능한 모든 LCB와 LSL 명세 집합이라고 하고, LSL의 의미모형, 즉 대수(algebra) 집합을 *Alg*로 나타내면, LSL의 의미함수(meaning function)를 $M_L : LSL \rightarrow 2^{Alg}$이라 할 때, LCB의 의미함수 $M_C : LCB \rightarrow 2^{Alg}$는 $M_C = M_L \circ T$로 정의할 수 있다. 만약, 의미함수 M_C가 이미 정의되었다면, 변환기반의 의미론이 건전하기 위해서는 (그림 5)의 다이어그램은 교환성(commutation)을 가져야 한다. 즉, 서로 다른 두 경로의 함수가 동일한 집합에 속하는 개체를 나타낼 때, 두 개체는 동일한 개체여야 한다. 예를 들면, 임의의 $c \in LCB$에 대하여 $M_L(T(c))$와 $M_C(c)$가 동일한 개체를 나타내야 한다.

변환기반의 LCB 의미론은 다음과 같은 두 단계로 정의할 수 있다.

- 의미요소 정의: LCB 언어의 핵심개념을 LSL를 사용하여 정의한다. 예를 들면, 객체, 상태, 추상값 등과 같은 의미요소가 이에 해당한다. LCB의 하부논리를 LSL를 사용하여 정의함으로써 LCB 명세를 LSL로 변환할 수 있는 이론을 제공한다.
- 변환규칙 정의: LCB의 각 구성요소에 대하여 어떻게 LSL로 변환하는가를 정의한다. 변환은 앞에서 정의한 의미요소 상에서 정의한다. 변환규칙의 핵심은 **requires**절과 **ensures**절의 논리문을 LSL로 정의된 의미요소로 정의하는 것이다.

물론 LCB 의미론이 잘 정립되기 위해서는 변환규칙이 건전하고 완전하여야 한다. 건전성은 변환규칙에 의한 의미론과 모형론적 의미론이 일치함을 말하고 완전성은 변환규칙이 모든 LCB 구성요소에 대하여 정의되었음을 말한다.

3.3.1 의미요소

LCB의 가장 중요한 개념은 '상태'이다. 상태란 프로그램 실행의 순간포착(snapshot)을 말하며, 프로그램은 특정 상태를 다른 상태로 변환한다. LCB에서 상태

(State)는 유한한 객체의 집합(ObjSet), 환경(Env), 저장(Store)의 세 요소로 구성된다.

State = ObjSet × Env × Store
Env = Var → Obj
Store = Obj → Val

환경 Env는 프로그램변수를 객체로 대응시키고 저장 Store는 객체를 추상값으로 대응시킨다. 일반적으로 환경은 정적으로 결정되는 반면, 저장은 프로그램 실행에 의존하기 때문에 동적으로 결정된다. 프로그램이 실행됨에 따라 상태는 다음과 같이 변할 수 있다. 첫째 새로운 객체가 생성되거나 기존 객체가 소멸됨에 따라 현존하는 객체의 집합을 나타내는 ObjSet이 변할 수 있다. 둘째 변수 범위가 변함에 따라 환경, 즉, 변수에서 객체로의 대응관계가 변할 수 있다. 마지막으로 객체의 값이 변함에 따라 저장이 변할 수 있다.

(그림 6)은 LSL를 사용하여 State를 명세한 것이다. 연산자 @은 변수나 객체에 대하여 특정 상태에서의 값을 되돌려 준다. State의 구성 요소인 환경과 저장의 LSL 명세는 부록 A.1를 참조하기 바란다.

```
StateTrait(State, Obj, Env, Store): trait
  includes EnvTrait(Env, Var, Obj),
    StoreTrait(Store, Obj, Val), Set(Obj, ObjSet)
  introduces
    State tuple of objs: ObjSet, env: Env, str: Store
    _ @ _: Obj, State → Val
    _ @ _: Var State → Val
  asserts
    forall x: Var, o: Obj, st: State
      o @ st  ==  eval(st.str,o);
      x @ st  ==  eval(st.str,eval(st.env,x))
```

(그림 6) 상태의 LSL 명세
(Fig. 6) An LSL formalization of states

3.3.2 변환규칙

LSL로 상태를 명확하게 정의하였으면, 함수명세와 같은 인터페이스 명세를 LSL 의미로 해석할 수 있다[3]. 일반적으로 함수명세는 다음과 같은 형태를 가진다.

behaviour { **requires** P; **modifies** M; **ensures** Q; }

일반적인 의미는 선조건문 P를 만족하는 초기상태에서 해당 함수가 호출되면 프레임공리 M과 후조건문 Q를 만족하는 종결상태에 도달하여야 한다는 뜻이다. 따라서 함수명세는 그러한 모든 종결상태를 나타낸다고 볼 수 있다. 즉, 상태에서 상태집합으로의 함수이다. 이를 의미함수 M: Spec, State → 2^{State}으로 나타내면 다음과 같다.

M[P, M, Q] s = {t ∈ State • T[P ⇒ M ∧ Q] s t},
s ∈ State

T는 진리함수(truth function)로 임의의 두 상태, 즉, 초기상태와 종결상태에 대하여 논리문의 진리값을 나타낸다[3]. 선후조건문에 나타나는 LCB 논리문은 객체의 초기값과 종결값을 언급할 수 있다. 따라서 논리문의 평가는 두 상태에 대하여 상대적으로 이루어 진다. 선후조건문을 변환하기 위한 대표적인 규칙은 다음과 같다(적용 사례는 제4절 참조).

- 변수 평가: 선후조건문의 변수는 문맥에 따라 변수가 나타내는 객체의 초기값, 종결값, 혹은 객체 자체를 나타낸다. 예를 들면, om′ = update(om^, n, o)을 상태쌍 (s,t)에 대해 평가하면, om @ t = update(om @ s, eval(s.env, n), eval(s.env, o))와 같다. 즉, om^과 om′은 초기값과 종결값을 나타내고 n과 o는 객체 자체를 나타낸다. 연산자 @과 LSL 함수 eval은 LCB 의미모형을 정의하는 LSL 명세 StateTrait에서 정의되었다((그림 6) 참조).
- 내장함수 평가: 프레임공리를 나타내는 modifies 절과 같은 LCB 내장함수도 일반 논리문과 마찬가지로 두 상태에 대하여 참 혹은 거짓을 나타낸다. 예를 들면, **modifies** om;를 상태쌍 (s,t)에 대하여 평가하면 다음과 같다. (LSL 함수 modified는 부록 A.1의 LSL 명세 StoreTrait에 정의되었다.)

modified(s.str, t.str) ⊆ {eval(s.env, om)}

4. 제안한 증명방법의 적용

본 절에서는 앞 절에서 제안한 정형증명방법을 이름서비스 명세에 적용한다. 일단 정형명세가 주어지면 사용된 명세언어와 명세언어의 바탕이 되는 증명논리

에 따라 다양한 종류의 특성을 증명할 수 있다. 본 논문에서는 한 예로 안전성(safety property)에 관한 특성을 증명한다.

CORBA 이름서비스 표준은 NamingContext 클래스가 여러 가지 특성을 가질 것을 명세하고 있다[12]. 예를 들면, c를 임의의 NamingContext 객체, e_i (i = 1, ..., n)을 이름구성요소라고 하자. 멤버함수 resolve는 다음과 같은 특성을 가져야 한다.

$c \rightarrow resolve(<e_1,e_2,...,e_n>) \equiv$ [특성 1]
$c \rightarrow resolve(<e_1,e_2,...,e_{n-1}>) resolve(<e_n>)$

즉, 복합이름일 경우 처음 n-1개의 이름구성요소가 나타내는 이름문맥 내에서 마지막 이름구성요소를 사용하여 이름이 분해된다. 참조의 편의를 위해서 위 특성을 [특성 1]이라 부른다.

4.1 특성의 정형해석

[특성 1]의 증명에 앞서 먼저 그 의미를 명확히 해야 한다. 일반적인 의미는 임의의 상태 s에서 왼쪽 문장을 실행했을 때 도달하는 상태와 동일한 상태에서 오른쪽 문장을 실행했을 때 도달하는 상태는 같다는 뜻이다. 제 3절에서 정의한 상태라는 개념을 사용하여 엄격하게 그 의미를 정의할 수 있다. Hoare 논리에 따라 프로그램 문장을 특정 상태에서 다른 상태로 변환시키는 상태변환기(state transformer)로 볼 수 있다[7]. 즉, 프로그램은 임의의 상태에 대하여 도달 가능한 모든 상태의 집합을 나타낸다. 의미함수 M을 사용하여 [특성 1]을 정의하면 다음과 같다.

$\forall$ s: State, c: NamingContext, n, n_i, n_j: Name
$|len(n@s) \geq 2 \wedge n_i@s = init(n@s) \wedge n_j@s = <last(n@s)> \bullet$
$M[c \rightarrow resolve(n)]s = M[c \rightarrow resolve(n_i) \rightarrow resolve(n_j)]s$

위 정의에서 두 가지 사항에 대하여 명확히 해야 한다. 첫째는 함수호출의 의미에 관한 것이고 둘째는 순차적 실행에 대한 의미이다. 함수호출 c→resolve(n)는 해당 함수를 실행한 이후에 도달할 상태, 즉 종결상태를 정의한다. 종결상태의 특징은 함수의 명세에 정의되어 있다. 예를 들어, 함수 resolve의 명세를 SPEC으로 나타내면 종결상태는 다음 논리문을 만족하여야 한다.

SPEC[ctx, om\c.ctx, c.om]

[ctx, om/c.ctx, c.om]는 동시대치를 나타낸다. 즉, 명세 SPEC에서 ctx, om를 각각 c.ctx, c.om로 동시에 대치한다는 뜻이다.

순차적인 함수호출 nc→resolve(n)→resolve(m)의 의미를 의미함수를 사용하여 정형화하면 다음과 같다. (result는 nc→resolve(n)의 결과값을 나타낸다고 가정한다.)

$\forall$ s, t: State, c: NamingContext, n, m: Name $\bullet$
$t \in M[c \rightarrow resolve(n) \rightarrow resolve(m)]s$
$\Leftrightarrow \exists r$: State $\bullet$ $r \in M[c \rightarrow resolve(n)]$ s $\wedge$
$t \in M[result \rightarrow resolve(m)]$ r

4.2 특성의 정형증명

본 논문에서는 정상적인 경우만을 가정하여 [특성 1]을 수학적으로 증명한다. 즉, 예외상황을 일으키지 않는다고 가정한다. 예외상황 시에는 각각의 경우에 대하여 본 절에서와 유사한 방식으로 증명하여야 한다. 증명의 편리를 위해 [특성 1]를 진리함수 T를 사용하여 다음과 같이 재정의 한다.

$\forall$ s, t: State, c: NamingContext, n, n_i, n_j: Name
$|len(n@s) \geq 2 \wedge n_i@s = init(n@s) \wedge n_j@s = <last(n@s)> \bullet$
$T[SPEC[n, ctx, om\n, c.ctx, c.om]]s\ t$
$\Leftrightarrow \exists r$:State $\bullet$ $T[SPEC[n,result,ctx,om\n_i,o,c.ctx,c.om]]sr$
$\wedge T[SPEC[n, ctx, om\n_i, o.ctx, o.om]]s\ t$

좌·우변이 동일(⇔)함을 증명하기 위해서는 ⇒와 ⇐를 독립적으로 증명하면 된다. 본 논문에서는 단지 ⇒만을 보인다. ⇐도 유사한 방법으로 증명할 수 있다. 증명에 앞서 먼저 다음 사실을 알 수 있다. 편의상 임의의 상태에서 변수의 값은 대문자를 사용하여 나타낸다. 예를 들면, N, CTX, OM은 각각 n @ s, (cn @ s).ctx@s,(cn@s).om@s를 나타낸다.

Fact 1. size(N) ≥ 2
Fact 2. init(N) ∈ CTX
Fact 3. N ∈ CTX ∨ N ∈ dom(OM)

Fact 1은 n이 복합이름이란 가정으로부터, Fact 2와 Fact 3은 함수호출이 정상적으로 종료한다는 가정, 즉 resolve의 선조건문으로부터 쉽게 추론할 수 있다.

증명은 좌우변을 독립적으로 간략화 한 후 =>가 성립함을 보인다. 먼저 좌변을 간략화 하면 다음과 같다.

T[SPEC[ctx, om\c.ctx, c.om]]s t
⇔SPEC[n,ctx,om,result\N,CTX,OM, result@t]
(명세 확장)
⇔(size(N)>0∧N∈dom(OM) (논리문 확장)
⇒result@t=apply(OM,N))∧
(size(N)>0∧N∈CTX
⇒result@t=[N⊗CTX, N⊗OM])

NamingContext의 불변진리에 의해 N ∈ dom(OM)와 N ∈ CTX는 상호 배타적이므로 (i.e., CTX ∩ dom(OM)=Ø) 두 가지 경우가 가능하다((그림 1) 참조). 두 경우 모두 증명은 유사함으로 본 논문에서는 첫 번째 경우, 즉 N ∈ dom(OM)일때 만 증명한다.

⇔size(N)>0∧N∈dom(OM) (가정: N∈dom(OM))
⇒result@t=apply(OM,N)
⇔result@t=apply(OM,N) (Fact 1과 가정)

다음은 우변을 간략화한다. 우변은 [L1]과 [L2]의 두 논리문으로 나누어서 분석한다.

∃r: State • T[SPEC[n,result,ctx,om\n_i,o,c.ctx,c.om]]s r [L1]
∧T[SPEC[n, ctx, om\n_i, o.ctx, o.om]]r t [L2]

먼저 논리문 [L1]은 다음과 같이 간략화 할 수 있다. 표기의 편의를 위해 n_i@s를 N_i로 나타낸다.

T[SPEC[n,result,ctx,om\ n_i,o,c.ctx,c.om]]s r
⇔(size(N_i)>0∧N_i∈dom(OM) (명세 확장)
⇒o@r=apply(OM, N_i))∧
(size(N_i)>0∧N_i∈CTX
⇒o@r=[N_i⊗CTX, N_i⊗OM])
⇔o@r=[N_i⊗CTX, N_i⊗OM] (Fact 1과 Lemma 1)

Lemma 1은 N ∈ dom(OM)이면 N_i ∈ CTX이고 N_i ∉ dom(OM)임을 나타낸다(부록 A.2 참조).

다음은 논리문 [L2]를 간략화 하기 위하여 먼저 N_l, CTX_r, OM_r를 각각 n_l@r, (o @ r).ctx @ r, (o @ r).om @ r이라고 하자.

T[SPEC[n, ctx, om\n_l, o.ctx, o.om]]r t
⇔(size(N_l)>0∧N_l∈dom(OM_r) (명세 확장)
⇒result@t=apply(OM_r, N_l))∧
(size(N_l)>0∧N_l∈CTX_r
⇒result@t=[N_l⊗CTX_r, N_l⊗OM_r])
⇔result@t=apply(OM_r, N_l)) (Lemma 2와 아래 설명)
⇔result@t=apply(N_l⊗OM, N_l) (OM의 정의)

Lemma 2는 N ∈ dom(OM)이면 last(N) ∈ dom(init(N) ⊗ OM)이고 last(N) ∉ dom(init(N)⊗OM)임을 말한다(부록 A.2 참조). 두 번째 동치관계(⇔)는 다음과 같은 이유 때문이다.

(1) size(N_l) = 1 (> 0). 왜냐 하면, N_l = n_l @ r = n_l @ s = last(n @ s) = <last(N)>이여야 하기 때문이다.
(2) CTX_r , OM_r 는 각각 N_l⊗CTX, N_l⊗OM이다. 왜냐 하면, o @ r = [N_l⊗CTX, N_l⊗OM]이여야 하기 때문이다.
(3) N_i = init(N)과 Lemma 2에 의해, N_l = last(N) ∈ dom(init(N)⊗OM) = dom(OM_r) 이다. N_i = n_i @ r = n_i @ s = init(n@s) = init(N)이여야 함에 주의하라.

마지막으로 간략화한 좌·우변을 종합하여 다음을 증명하면 된다.

result@t=apply(OM, N) ⇒result@t=apply(N_l⊗OM, N_l)

즉, apply(OM, N) = apply(N_l⊗OM, N_l)를 보이면 된다. 임의의 객체 o에 대하여 apply(OM, N) = o라고 가정하자.

apply(OM, N) = o
⇔ (N, o) OM (apply의 정의)
⇔ (last(N), o) ∈ init(N)⊗ OM (연산자 ⊗의 공리)
⇔ (N_l, o) ∈ N_l⊗OM (N_l과 N_i의 정의)
⇔ apply(N_l⊗OM, N_l) = o (apply의 정의)

그러므로 apply(OM, N) = apply(N_l⊗OM, N_l)이며, 따라서 [특성 1]은 증명되었다.

4.3 교훈 및 제한사항

정형증명에서 가장 중요한 사항은 증명하고자 하는 특성의 정형적 표현과 증명의 모듈화이다. 먼저 특성을 증명에 편리한 형태로 수학적으로 표현해야 한다. 본 예제에서처럼 1차 서술논리를 사용하거나 집합론을 바탕으로 할 수 있다. 다음은 증명의 모듈화이다. 예를 들면, 프레임공리나 객체의 생성과 소멸에 관한 특성을 객체의 값에 대한 논리로부터 분리하여 증명할 수 있다. 프레임공리는 **modifies**절로 객체 생성과 소멸은 **fresh**와 **trashed**절로 명세한다[4]. 객체값에 대한 증명은 정상적인 경우와 예외상황인 경우로 나누어 케이스

분석(case analysis)법을 사용하는 것이 바람직하다.

본 논문의 명세기법과 증명방법에는 한계가 있다. 가장 대표적인 것은 값 지향(value-oriented) 증명방법이라는 것이다. 객체의 고유성(identity)이나 객체공유(aliasing)에 관한 증명이 불가능하다. 예를 들어, 멤버함수 bind와 resolve는 ctx→bind(<e_1,e_2,...,e_n>,o)≡ ctx →resolve(<e_1,e_2,...,e_{n-1}>)→bind(<e_n>,o)와 같은 특성을 지녀야 한다. 즉, 복합이름 일 경우 처음 n-1개의 이름 구성요소가 나타내는 이름문맥에서 객체에 이름이 부여된다는 뜻이다. 이 특성을 증명하기 위해서는 함수 resolve가 객체식별자를 되돌려주어야 하고 결과객체에 함수 bind를 호출한 영향은 전체 시스템에 반영되도록 명세 되어야 한다. 또한 이러한 명세를 바탕으로 하여 객체가 가지는 값이 아니라 객체 자체에 관한 증명논리가 필요하다.

5. 맺음말

CORBA 표준모듈의 행위검증을 위하여 LCB로 작성된 정형명세를 LSL로 변환하고 LSL 증명논리를 사용하여 특성을 정형적으로 증명하는 방법을 제안하였다. 또한 CORBA 이름서비스모듈에 실제 적용하여 제안한 방법의 효용성을 입증하였다. 본 논문의 주요 기여도는 다음과 같다.

- LCB라 불리는 Larch와 CORBA IDL를 접목한 모듈 인터페이스 명세언어를 제안하였다. LCB에 대한 기본개념은 [9]에서 처음으로 제안하였다. 본 연구에서는 모형변수와 주석 개념의 도입을 통하여 실사용 가능한 수준으로 발전시켰다. [1]의 LCB 명세에 비해 본 논문의 명세가 간단명료하며 이해가 쉽다는 점에서 모형변수의 효과를 짐작할 수 있다.
- LCB 정형의미론을 부분적으로 정의하였다. 정형의미론의 바탕이 되는 의미모형을 LSL로 정의하였고 이를 바탕으로 하여 LCB 명세의 LSL로의 변환규칙을 정의함으로써 변환기반 정형의미론을 정립하였다. 의미함수도 부분적으로 정의하였다.
- CORBA 표준모듈의 검증을 위한 방법을 소개하였다. 구현모듈의 사용자 인터페이스와 행위적 특성을 정형적으로 증명할 수 있다는 점에서 큰 의미를 둘 수 있다. 특히, 변환방식을 채택하여 새로운 증명논리를 정의하지 않고 LCB의 바탕인 LSL 논리와 기존 지원도구를 사용할 수 있다는 점이 큰 장점이다. CORBA 이름서비스모듈에 실제 적용하여 제안한 방법이 효과적임을 부분적으로 입증하였다.

본문에서 지적한 바와 같이 제안한 증명방법의 가장 큰 한계는 값 지향적이라는 사실이다. 따라서 향후 가장 중요한 연구과제는 객체와 객체공유에 대한 증명방법으로의 확장이다. 현재 제시한 의미모형에서는 객체공유에 대한 개념을 제공하고 있다. 하지만 인터페이스에서 이를 어떻게 명세하고 증명할 것인가에 대한 연구가 필요하다. 제안한 증명방법에 LSL 정리 증명기인 LP를 사용할 수 있다고 시사하였다. 예제의 일부를 LP를 사용하여 기계적으로 증명하는 것은 흥미로운 향후 연구 주제가 될 것이다.

참 고 문 헌

[1] 김미희. OMG 이름서비스 명세의 정형화. 한국정보처리학회논문지, 5(3):458-474, 1998년 2월.

[2] Tony Bryant and Andy Evans. Formalizing the Object Management Group's Core Object Model. *Computer Standards & Interfaces*, 17(5-6):481-489, September 1995.

[3] Yoonsik Cheon. Inheritance in Larch Interface Specification Languages, Its Semantic Foundation and Formal Semantics. In *Proceedings of International Refinement Workshop and Formal Methods Pacific 1998, September 29-October 2, Canberra, Australia,1998*. Springer-Verlag, 1998. To appear.

[4] Yoonsik Cheon and Gary T. Leavens. A Quick Overview of Larch/C++. *Journal of Object-Oriented Programming*, 7(6):39-49, October 1994.

[5] Grgory Duval. Specification and Verification of an Object Request Broker. In *Proceedings of 20th International Conference on Software Engineering, Kyoto, Japan, April 19-25, 1998*, pages 43-52, IEEE Computer Society, 1998.

[6] John V. Guttag and James J. Horning. Larch: Languages and Tools for Formal Specification.

Springer-Verlag, New York, N.Y., 1993.

[7] C.A.R. Hoare. An Axiomatic Basis for Computer Programming, *Communications of ACM*, 12(10): 576-583, October, 1969.

[8] Iain S.C. Houston and Mark B. Josephs. A Formal Description of the OMG's Core Object Model and the Meaning of Compatible Extension. *Computer Standards & Interfaces*, 17(5-6): 553-558, September 1995.

[9] Gary T. Leavens and Yoonsik Cheon. Extending CORBA IDL to Specify Behavior with Larch. *OOPSLA '93 Workshop: Specification of Behavioral Semantics in OO Information Modeling*, October 1993.

[10] Sriram Sanjar and Roger Hayes. ADL: An Interface Definition Language for Specifying and Testing Software. *ACM SIGPLAN Notices*, 29(8):13-21, August 1994. Proceedings of the Workshop on Interface Definition Language, Jeannette M. Wing (editor), Portland, Oregon.

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

[12] The Object Management Group, Inc. *CORBA services: Common Object Service Specification*. The Object Management Group, Framingham, MA, March 1995. OMG Document Number 95-3-31.

A. 부 록

A.1 환경과 저장의 추상모형

```
EnvTrait(Env, Var, Obj): trait
  includes StoreTrait(Env,Var,Obj)

StoreTrait(St,Obj,Val): trait
  includes Set(Obj,ObjSet)
  introduces
    empty: → St
    bind: St,Obj,Val → St
    bottom: → St
    __∈__: Obj,St → Bool
    eval: St,Obj → Val
    isBottom: St → Bool
    added: St,St → ObjSet
    modified: St,St → ObjSet
    objs: St -> ObjSet
  asserts
    St generated by empty, bottom, bind
    St partitioned by ∈, eval, isBottom
    forall obj,obj1:Obj, s,s1:St, v,v1: Val
      ~(obj empty);
      obj  bind(s,obj1,v) == (obj = obj1) / (obj ∈ s);
      eval(bind(s,obj,v),obj1) ==
        if obj1 = obj then v else eval(s,obj1);
      ~isBottom(empty);
      ~isBottom(bind(s,obj,v));
      isBottom(bottom);
      ~(empty = bottom);
      ~(bind(s,obj,v) = bottom);
      obj∈added(s,s1) == ~(obj∈s) / (obj∈s1);
      obj∈modified(s,s1) == (obj∈s) &
        ~((obj∈s1) / (eval(s,obj) = eval(s1,obj)));
      obj∈objs(s) == obj∈s
    implies
      converts∈:Obj,St→Bool, eval, isBottom, added,
        modified, objs
      exempting forall obj:Obj, s:St
        obj ∈ bottom,
        eval(bottom,obj),
        eval(empty,obj),
        added(bottom,s), added(s,bottom),
        modified(bottom,s), modified(s,bottom),
        objs(bottom)
```

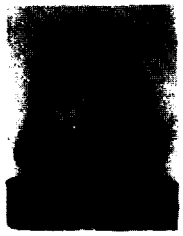
A.2 정형증명

Lemma 1. 만약 $N \in dom(OM)$이면 $N_i \in CTX$이고 $N_i \notin dom(OM)$ 이다

증명: $N \in dom(OM)$이면 Fact 1 ($len(N) \geq 2$)과 트리 구조를 가져야 한다는 NamingContext의 불변진리((그림 1) 참조)에 의해, $N_i \in CTX$이다. $N_i \in CTX$이면 불변진리 $ctx \cap dom(om) = \emptyset$에 의해 $N_i \notin CTX$이다. ■

Lemma 2. 만약 N ∈ dom(OM)이면 last(N) ∈ dom(init(N)⊗OM)이고 last(N) ∉ dom(init(N)⊗OM)이다.

증명: Fact 1 (len(N)≥2)에 의해 last(N)과 init(N)는 정의된다. 연산자 의 정의에 의해 (N,o) ∈ OM이면, (last(N), o) ∈ init(N) ⊗ OM이다 ((그림 3) 참조). 또한, 불변진리 ctx ∩ dom(om) = Ø에 의해 last(N) ∉ dom(init(N)⊗OM)이다.



김 미 희

e-amil : mihee@etri.re.kr

1989년 숙명여자대학교 전산학과 졸업(학사)

1989년~현재 한국전자통신연구원 연구원

1997년~현재 충남대학교 컴퓨터과학과 석사과정

관심분야 : 고속 통신망, 이동 통신망, 분산 처리, 소프트웨어 공학