OMG 이름 서비스 명세의 정형화

김 미 희†

요 약

OMG의 목표인 광범위한 소프트웨어의 재사용과 상호 운용성을 실현하기 위해서는 인터페이스와 행위에 관한 정확한 명세를 필요로 한다. 소프트웨어 부품의 행위가 모호하게 기술되어서는 사용자의 요구 사항과 표준에의 만족 여부를 판단하기 어렵기 때문이다. 본 논문에서는 정형 명세 언어 Z과 Larch를 사용하여 OMG 이름 서비스 명세를 정형화한다. 먼저 Z 표기법으로 이름 서비스의 기본 개념과 용어를 수학적으로 정의하고, 각 단위 모듈의 인터페이스와 행위를 Larch/CORBA-IDL를 사용하여 정형적으로 명세 한다. 이는 OMG 표준에 정형성을 도입하는 이론적 토대와 혜택의 구체적 근거 자료가 된다. 또한 표준의 정형화에 복수 개의 명세 언어가 조화롭게 사용될 수 있으며, Larch 방식이 분산 객체 시스템의 인터페이스와 행위 명세에 이상적임을 보여준다.

A Formalization of OMG Naming Service Specification

Mihee Kim†

ABSTRACT

To realize the OMG's ultimate goal – a wide reuse and interoperability of software systems, it is necessary to formally specify the software components. Without a formal behavioral interface specification of the components, it is difficult to determine conformance to the standard and satisfaction of user's requirements. In this paper, we formalize the OMG Naming Service Specification using the formal specification languages Z and Larch. Firstly is defined in Z a mathematical model for OMG concepts of naming services and its terminology. Based on this, the interface and behavior of OMG naming service modules are formalized using Larch-style Behavioral Interface Specification Language (BISL) for CORBA IDL, called Larch/CORBA-IDL. Our work establishes a foundation for applying formal methods to standard specification, and shows its benefits, in particular, of behavioral interface specification. It also demonstrates that multiple formalism can be adapted to specify the standards, and Larch is ideally suited to specify the behavioral semantics for objects in distributed systems.

1. 개 요

객체 지향 기술은 C++, Smalltalk, Java, Eiffel과 같은 프로그래밍 언어를 통하여 상업용 및 고객 응용 소프트웨어 개발에서 꾸준한 호응을 받고 있다. 객체 지향의 핵심 기술 중의 하나는 소프트웨어의 재사용이며, 이른바 부품 소프트웨어(component software)라는 소프트웨어 개발의 새로운 문화를 형성시키고 있다. 부품 소프트웨어는 여러 개발자로부터 독자적으로 개발된 응용 요소들을 결합하여 새로운 시스템을 구성하는 것으로, 최근 들어 기업체들이 분산 시스템을 논리적 중앙 집중형으로 간주함으로써 기존 소프트웨어의 가치를 향상시키려는 경향과 일치한다. 이상적으로 네트워크 사용자는 기업의 모든 응용 소프트웨어와 자료를 조화롭게 결합된 단일 시스템으로 볼 수

† 정 회 원:한국전자통신연구원
논문접수:1997년 6월 2일, 심사완료:1997년 12월 9일

있어야 한다. 부품 소프트웨어와 같은 객체 지향 기술을 이용한 새로운 문화가 정착되기 위해서는 객체 지향 기술을 이용한 소프트웨어 개발의 공통된 틀이 마련되고 이를 위한 산업계의 지침서와 객체 시스템의 표준이 필요하다. 이러한 맥락에서 Object Management Group(OMG)이라는 산업계를 주축으로 한 단체가 설립되었으며, 소프트웨어 개발에 객체 지향 기술의 채택을 독려하고 관리함을 목적으로 한다. OMG는 핵심 객체 모형, 객체 관리 구조, CORBA, 객체 서비스 등의 표준을 제정하고 있다[15][17][18].

OMG의 궁극적 목표인 소프트웨어의 재사용, 투명성, 이식성, 상호 운용성은 인터페이스와 구조에 관한 정확한 명세 없이는 실현될 수 없다. 부품의 행위가 모호하게 기술되어서는 사용자의 요구 사항과 표준에의 만족 여부를 검사할 수 없기 때문이다. 명세는 기계적으로 처리될 수 있어야 한다. 효율적인 재사용을 위하여 지원 도구를 사용하여 적합한 부품을 찾을 수 있어야 하고 요구 사항에의 만족 여부를 검사할 수 있어야 하기 때문이다. 따라서 일부 OMG 문서에서 지적되고 있는 바와 같이 OMG 표준의 정형화가 시급하다.

OMG 표준의 정형화는 크게 (1) OMG 객체 모형을 위한 수학적 토대 정립과 (2) CORBA, 공통 객체 서비스, 공용 부대시설 등의 인터페이스와 행위에 관한 정형적 명세로 분류할 수 있다. OMG 표준의 수학적 토대 정립은 OMG의 핵심 객체 모형과 확장을 정형적 표기법을 사용하여 기술하고 정형적 의미를 부여하는 것이다. 핵심 객체 모형에 건전한 바탕을 제공하는 것은 여러 중요한 혜택을 준다. 이들 대부분은 좋은 표준이 제공해야 할 사항들이다. 예를 들면, 증명성, 부품·제품·방법 등의 통합, 벤더의 독자성, 상호 운용성, 이식성과 디자인의 재사용 등을 들 수 있다[3]. OMG 객체 모형의 핵심을 이해하기 위하여 Z 표기법을 사용한 정형화가 시도되고 있으며, 이를 통해 호환성의 의미가 명확해졌다고 보고되고 있다[8].

인터페이스는 사용자와 구현 사이에 뚜렷한 경계를 설정하므로 사용자에게 보여지는 구현 종속적인 사항을 제한한다. 사용자와 구현자를 정형화된 인터페이스 명세를 통해 서로 독립시키므로 소프트웨어 시스템의 모듈화와 유연성을 극대화할 수 있다. 인터페이스를 정형화하는 한 방법은 OMG에서 정의한 인터페이스 정의 언어(Interface Definition Language, IDL)를 사용하는 것이다[17]. 현 IDL은 의미 – 행위에 관한 명세 – 를 포함하고 있지 않다. 비록 프로그램 소스 코드를 행위에 관한 정확한 명세로 취급할 수는 있지만, 일반적으로 명세가 가지는 기능을 충분히 발휘하지 못한다. 또한 소스 코드에는 명세로서는 불필요한 구현 종속적인 사항과 세부적인 내용을 포함하므로 프로그래머가 읽고 이해하기에는 부적당할 뿐만 아니라, 소프트웨어가 외부 소프트웨어 업체로부터 구매되었을 경우 이들 업체는 소스 코드를 공개하길 꺼린다. 따라서 소스 코드 보다는 더 추상적인 행위에 관한 정형 명세가 필요하다. 인터페이스와 행위에 관한 정형 명세는 표준에의 만족성 검사를 가능하게 할 뿐만 아니라 정형 방법의 혜택을 OMG의 기본 틀 안에 제공한다. 특히, 명세는 개발자가 코드의 참조 없이 부품을 재사용할 수 있도록 하며, 정형성은 명세의 이용도를 증대시킨다. 예를 들어, 재사용 가능한 부품을 의미에 바탕을 두고 검색하고, 검색된 부품이 사용자의 요구 사항을 만족하는지 검증하고, 결합할 수 있도록 하는데, 이는 재사용을 강조하는 OMG의 목표와 일치한다.

현재 인터페이스를 정형적으로 명세하기 위한 연구가 활발히 진행되고 있으며[9][13], CORBA IDL를 확장한 몇몇 명세 언어가 제안되고 있다[1][10]. 그러나 OMG 표준에 정의된 모듈의 인터페이스와 행위를 실제적으로 명세한 예는 찾아 보기 힘들다. 따라서 본 논문에서는 OMG에서 제정한 객체 서비스 중에서 이름 서비스 명세를Larch 방식을 사용하여 정형화한다. 본 연구의 궁극적 목적은 (1) OMG의 기본 모형과 개념의 정형화 뿐만 아니라 인터페이스와 행위에 관한 정형적 명세가 필요하다는 것과, (2) 이를 위하여 Larch 방식의 명세가 이상적으로 적용될 수 있다는 것을 보이는 것이다. 부수적으로 정형화를 통하여 OMG 이름 서비스를 심도 있게 이해하고 이를 검증하는 것과 분산 객체의 행위 명세에는 어떠한 요구 사항이 있는지 그리고 이를 충족하기 위해서는 어떠한 명세 기법이 필요한 지와 같은 분산 객체 시스템의 인터페이스와 행위 명세에 관한 전반적인 사항들을 연구하는 것이다.

표준의 정형화는 OMG 뿐만 아니라 여러 국제 표준화 기구에서 시도하고 있다[6][14]. ISO는 이 분야에

서 선각자적인 역할을 하고 있으며, 최근에는 Open Distributed Processing(ODP)을 주축으로 하여 객체 지향 개념을 정형화하고 통신 프로토콜과 표준의 정형화에 객체 지향 정형 방법을 적용하고 있다[2][11]. 또한 정형 명세 언어 LOTOS와 SDL의 객체 지향 확장도 연구하고 있다. 하지만 연구 노력 대부분을 이론적 토대 정립에 치중하고 있으며, 실제 표준의 정형화는 아직까지 미흡하다.

본 논문은 다음과 같이 구성된다. 먼저 이 절의 나머지 부분에서는 Larch/C++를 중심으로 하여 Larch 방식의 명세 기법을 간단히 소개한다. Larch/C++는 본 논문에서 사용하는 정형 명세 언어 Larch/CORBA-IDL(LCB)의 근간이다. 제 2절에서는 OMG 이름 서비스의 기본 개념과 용어를 정형 명세 언어 Z를 사용하여 정형화한다. 본 논문의 핵심 부분인 제 3절에서는 이름 서비스의 각 모듈을 LCB를 사용하여 정형적으로 명세 한다. 마지막으로 제 4절에서는 본 연구에서 얻은 교훈, OMG와 LCB에 대한 조언, 향후 연구 방향과 함께 본 논문의 결론을 맺는다.

1.1 Larch 방식의 명세

Larch는 프로그램 모듈을 정형적으로 명세하기 위한 양층형 접근 방식이다. 명세자는 대수 명세 언어인 Larch Shared Language (LSL)를 사용하여 시스템 내 객체의 추상 값을 정형적으로 묘사한다[7, 제 4장]. 객체의 동적 행위는 해당 객체 또는 관련된 객체가 가지는 추상 값의 변화로 묘사한다. 객체의 추상 값이 수정될 수 있으므로 상태라는 개념을 사용한다. 프로그램 언어마다 상이한 구문을 가지므로 인터페이스는 행위적 인터페이스 명세 언어 (Behavioral Interface Specification Language, BISL)라 불리는 특정한 언어에 맞추어진 표기법으로 묘사하는데, Hoare 방식의 선·후조건문을 사용한다. 현재 C, C++, Smalltalk, CORBA-IDL 등을 위한 BISL이 제공된다[4][7][10]. 선·후조건문에서는 LSL로 정의한 수학 함수를 사용하여 추상 값을 조작한다. 따라서 추상 값을 묘사하는데 주어진 어휘로 제한되는 것이 아니라 추상화 정도를 조율할 수 있다. Larch의 특징은 어떤 객체가 수정 혹은 비 수정될 수 있는가를 묘사할 수 있다는 것이며, 예외 상황과 병렬 시스템을 명세하기 위한 기법이 제안되고 있다.

```
FiniteMap(M, D, R): trait
  % An M is a map from D's to R's
  introduces
    {}: → M
    update: M, D, R → M
    apply: M, D → R
    defined: M, D → Bool
  asserts
    M generated by {}, update
    M partitioned by apply, defined
    ∀m: M, d, d1, d2: D, r: R
      apply(update(m, d2, r), d1) ==
        if d1 = d2 then r else apply(m, d1);
      ¬defined({}, d);
      defined(update(m, d2, r), d1) ==
        d1 = d2  ∨defined(m, d1)
  implies
    converts apply, defined
      exempting  ∀d: D apply({}, d)
```

(그림 1) 유한 사상의 LSL 명세
(Fig. 1) An LSL specification of finite map

(그림 1)은 유한 사상의 수학 모형을 LSL를 사용하여 명세 한다. LSL 명세는 '트레이트 함수(trait function)'라 불리는 수학 함수를 정의하는데, 이는 추상 값을 표현하기 위한 어휘이다. 예를 들어, {}와 update ({}, d, r)는 유한 사상의 가능한 두 추상 값을 나타낸다. LSL 명세에서 **asserts** 절의 등식 공리는 선언된 함수 사이의 동치 관계를 정의한다. 특수한 형태인 **generated by** 절은 {}와 update가 유한 사상의 모든 값을 표현하는데 충분하다는 것을 나타낸다. 즉, 자료형 M의 모든 추상 값에 대한 특성을 증명할 수 있는 귀납적 추론 규칙을 제공한다. 반면, **partitioned by** 절은 낱말간의 동치 관계를 정의한다. 두 낱말이 나열된 함수에 의해 구별될 수 없을 경우 같은 낱말로 취급한다. 즉, 두 낱말은 동일한 추상 값을 나타낸다. 이는 연역적 추론 규칙을 제공한다. 명세의 이해를 돕기

```
uses FiniteMap(Map, Dom, Ran);
template<Map*, Dom*, Ran*>
void map_update(Map *m, Dom *d, Ran *r) {
  requires ¬defined(*m^,*d^);
  modifies *m;
  ensures *m' = update(*m^,*d^,*r^);
}
```

(그림 2) 함수 map_update의 명세
(Fig. 2) Specification of function map_update

위하여 명세로부터 추론할 수 있는 사항은 **implies**절에 나열할 수 있다. LSL에 관한 자세한 내용은 [7, 제4절]를 참조하기 바란다.

(그림 2)는 C++ 일반(generic) 함수 map_update를 C++를 위한 BISL인 Larch/C++로 명세한 것이다[4]. 먼저 **uses** 절은 명세의 수학적 모형은 LSL 명세 FiniteMap에 정의됨을 나타낸다((그림 1) 참조). 일반적으로 함수 명세는 **requires** 절, **modifies** 절, **ensures** 절로 구성된다. 함수를 호출할 때 만족되어야 할 선조건문은 **requires** 절에 기술한다. 프레임 공리를 나타내는 **modifies** 절은 명세 된 함수가 나열된 객체만 수정할 수 있다는 것을 나타낸다. 이는 어떤 객체의 추상 값이 수정될 수 있는가를 나타내는 강한 간접적 단언이다. 즉, 나열된 객체 이외의 모든 객체의 추상 값은 변경될 수 없다. 함수 실행 후 만족되어야 하는 후조건문은 **ensures** 절에 명세 한다. 선조건문이 사용자를 제약하는 반면 후조건문은 개발자를 제약한다. 즉, 사용자는 선조건이 만족된 상태에서 함수를 호출해야 하며, 개발자는 후조건에 명세 된 대로 동작하는 함수를 구현할 의무가 있다. 함수 명세의 의미는 선조건이 만족된 상태에서 함수가 호출되면, 함수의 실행은 반드시 종결되어야 하고 종결 상태에서 후조건은 만족되어야 한다. 만약 선조건이 만족되지 않는 상태에서 호출되면 아무것도 보장되지 않는다.

예에서 선·후조건문의 *m^과 *m'은 각각 함수 호출 직전과 직후의 포인터 m이 가리키는 객체(*m)의 추상 값을 나타내고, defined와 update는 LSL 명세 FiniteMap에서 그 의미가 잘 정의된 트레이트 함수이다. Larch/C++에 관한 자세한 내용은 참고 문헌 [4]를 참조하기 바란다.

2. 기본 개념과 용어 정의

OMG는 분산 객체 관리 시스템을 지원하기 위하여 객체 서비스 표준을 정의하고 있다[18]. 공동 객체 서비스 명세(Common Object Service Specification, COSS)라 불리는 이 표준은 객체를 실현하고 관리하기 위한 기본적인 함수－즉, 객체의 생성에서부터 소멸까지의 생명 주기－를 표준화하고 있다. 객체를 생성하고, 객체의 사용을 통제하고, 객체의 이동을 관장하고, 객체 간의 연관 관계를 관리하기 위한 인터페이스를 제공한다. 본 논문에서 다룰 이름 서비스는 명세는 COSS에 규정된 객체에 이름을 부여하는 것에 대한 OMG 표준이다. 본 절에서는 먼저 OMG 이름 서비스의 기본 개념과 사용되는 용어를 정형 명세 언어 Z를 사용하여 엄격하게 정의한다[16].

OMG는 객체 이름에서 해당 이름을 갖는 객체로의 관계를 '이름 결합(name binding)'이라 부른다. 이름 결합은 항상 특정한 '이름 문맥(naming context)'에 상대적으로 정의된다. 즉, 이름 문맥은 이름 결합 집합을 나타내는 객체이다. 이름 문맥에서 이름은 유일해야 하며, 한 객체는 여러 이름을 가질 수 있다. 그러나 모든 객체가 이름을 가질 필요는 없다. 가능한 모든 이름과 객체를 각각 집합 N과 Obj로 나타낼 때, 이름 결합 NB와 이름 문맥 NC를 다음과 같이 정의할 수 있다. (Z에서 $\times$와 $\nrightarrow$은 각각 곱 집합과 부분 함수를 나타낸다.)

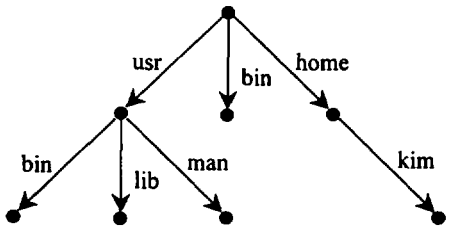
$$
\begin{aligned}
&[N, Obj] \\
&NB = N \times Obj \\
&NC = \{b : NB \mid \\
&\qquad \forall n : N; o, o_1 : Obj \bullet \\
&\qquad\quad (n, o) \in b \wedge (n, o_1) \in b \Rightarrow o = o_1\} \\
&\quad = N \nrightarrow Obj
\end{aligned}
$$

임의의 이름 문맥에서 특정한 이름과 결합된 객체를 찾는 것을 '이름 분해(name resolution)' 또는 '이름을 분해한다'라고 한다. 반면 객체에 새로운 이름을 부여하는 것을 '이름 결합' 또는 '이름을 결합한다'라고 한다. 이름은 항상 문맥에 상대적으로 분해되고 결합되며, 절대 이름이란 없다. 이름 분해와 이름 결합을 Z 함수 resolve와 bind로 정의할 수 있다(아래 참조).

이름 문맥도 객체이므로 특정한 이름 문맥에서 이름이 주어지는 객체가 또 다른 이름 문맥일 수 있다. 이는 이름 문맥을 일반 객체와 동등하게 취급한다는 뜻이다. 따라서 이름 문맥이 아닌 순수한 객체 집합을 $PObj$로 나타낼 때, 이름 부여가 가능한 모든 객체 집합 Obj는 순수 객체 집합 $PObj$와 이름 문맥 집합 NC의 합 집합으로 주어진다.

$$
\begin{aligned}
&[PObj] \\
&Obj = PObj \cup NC
\end{aligned}
$$

$[PObj]$
$Obj = PObj \cup NC$



(그림 3) 이름 그래프의 예
(Fig. 3) An example of naming graph

임의의 이름 문맥에서 다른 이름 문맥에 이름을 결합하는 것을 '이름 그래프(naming graph)'로 가시화할 수 있다((그림 3) 참조). 이름 그래프는 노드가 객체이고 변이 이름으로 구성되는 방향성을 지닌 그래프이다. 이름 그래프는 Z 스키마를 사용하여 다음과 같이 정의할 수 있다. (논리 함수 *wfNG*는 노드와 변 속성이 그래프 조건을 만족함을 나타낸다. 지면 관계상 해당 정의는 생략한다.)

NG
$nodes: \mathbb{P} Obj$
$edges: \mathbb{P}(Obj \times N \times Obj)$
---
wfNG(*nodes*,*edges*)

임의의 이름 문맥에 해당하는 이름 그래프를 생성하는 함수 mkNG를 다음과 같이 정형적으로 정의할 수 있다.

$mkNG: NC \rightarrow NG$

$\forall c: NC; g: NG; n: N; x, y: Obj \bullet$
$mkNG(c) = g \Leftrightarrow$
$x \in g.nodes \Leftrightarrow x = c \vee$
$\exists (n_1,o_1),\ldots,(n_k,o_k): NB; o_0: NC \mid o_0 = c \bullet$
$(n_i,o_i) \in o_{i-1} \wedge x = o_k$
$(x,n,y) \in g.edges \Leftrightarrow$
$\exists (n_1,o_1),\ldots,(n_k,o_k): NB; o_0: NC \mid o_0 = c \bullet$
$(n_i,o_i) \in o_{i-1} \wedge x = o_{k-1} \wedge n = n_k \wedge y = o_k$

예를 들면, 이름 문맥 nc_1, nc_2, nc_3가 각각 아래와 같이 정의되었을 때, 이름 문맥 nc_1에 해당하는 이름 그래프 $mkNG(nc_1)$는 (그림 3)과 같다. 그림에서 편의상 객체 식별자는 생략하였다.

$nc_1 = \{(usr, nc_2), (bin, o_1), (home, nc_3)\}$
$nc_2 = \{(bin, o_2), (lib, o_3), (man, o_4)\}$
$nc_2 = \{(kim, o_5)\}$

이름 그래프는 객체에 복잡한 이름을 부여할 수 있도록 한다. 즉, 임의의 이름 문맥에서 이름 열(name sequence)로 객체를 나타낼 수 있다. 이름 열은 이름 분해 시 거치는 이름 그래프 상의 경로(path)를 나타낸다. 지금부터는 이름 열을 줄여서 이름이라 부르고 이름 열을 구성하는 요소를 이름 구성 요소(name component)라 부른다. 따라서 이름은 여러 이름 구성 요소로 구성되는 구조물이다. 이름 구성 요소를 집합 N로 나타낼 때 이름 NS는 다음과 같이 정의된다.

$NS = \text{seq}\, N$

하나의 이름 구성 요소를 가지는 이름을 '단순 이름(simple name)'이라 하고, 여러 개의 이름 구성 요소를 가지는 이름을 '복합 이름(complex name)'이라고 한다. 복합 이름의 경우, 마지막을 제외한 모든 이름 구성 요소는 이름 문맥을 나타내며, 마지막 구성 요소는 해당 이름에 결합된 객체를 나타낸다. 이름을 정형화하였으므로 이름 분해와 이름 결합 함수를 수학적으로 정의할 수 있다. (Z 표기법 $\langle\rangle$, $\frown$, $\oplus$, $\mapsto$은 각각 열, 열 연결, 함수 겹침, 함수 원소를 나타낸다.)

$resolve: NC, NS \rightarrow\!\!\!\!\!\!\bullet\;\; Obj$

$\forall c: NC; e: N \mid e \in \text{dom}(c) \bullet$

$resolve(c, \langle e \rangle) = c(e)$
$\forall c: NC; e: N; n: NS \bullet$
$resolve(c, n \frown \langle e \rangle) = resolve(resolve(c,n), \langle e \rangle)$

$bind: NC, NS, Obj \rightarrow\!\!\!\!\!\!\bullet\;\; NC$

$\forall c: NC; e: N; o: Obj \bullet$
$bind(c, \langle e \rangle, o) = c \oplus \{e \mapsto o\}$
$\forall c: NC; e: N; n: NS; o: Obj \mid e \in \text{dom}(c) \bullet$
$bind(c, \langle e \rangle \frown n, o) =$
$\text{let } c_1 == c(e) \text{ in } c \oplus \{e \mapsto bind(c_1,n,o)\}$

이름 구성 요소는 이름과 종류를 나타내는 두 속성으로 구성된다. 두 속성 모두 CORBA IDL 문자 열로 정의된다.

[*IDLString*]

N

id : *IDLString*
kind : *IDLString*

종류 속성을 사용하면 문법에 비종속적으로 표현력을 향상시킬 수 있다. 예를 들면, 종류 속성은 객체의 종류를 나타내는 cpp_source, object_code, executable 등의 값을 가질 수 있다. 이름 서비스 시스템은 이들 값에 의미를 부여하거나 해석하거나 관리하지 않는다. 상위 시스템이 이를 관리하고 사용에 관한 정책을 수립한다고 가정한다. OMG 이름 서비스 명세는 이름을 항상 위에서 설명한 구조적 형태로만 취급한다. 즉, 이름을 위한 문법이나 문자를 규정하지 않는다. 응용 프로그램 또는 시스템 서비스가 구조적 형태의 이름을 구현 종속적인 표현으로 변환시켜야 한다.

3. 이름 서비스 명세

OMG에서는 이름 서비스 명세를 CORBA IDL를 사용하여 비정형적으로 정의하고 있다[18]. 본 절에서는 먼저 이름 서비스 명세의 기본 골격을 Z를 사용하여 정의하고 각 모듈의 행위를 Larch를 사용하여 정형화 한다.

OMG 이름 서비스는 크게 *CosNaming*과 *NamesLibrary*의 두 모듈로 구성된다. 모듈 *CosNaming*은 이름 서비스를 정의하는 인터페이스[1]의 집합으로 구성되며, *NamesLibrary*는 이름 객체를 가객체(pseudo object)로 구현하는 라이브러리이다. 이름 라이브러리는 이름 객체를 주기억장치 내에서 효율적으로 조작할 수 있도록 한다.

$$NamingService \cong CosNaming \wedge NamesLibrary$$

이름 서비스의 핵심이 되는 *CosNaming* 모듈은 각각 이름과 이름 구성 요소를 나타내는 *Name*, *NameComponent*라는 타입과 이름 결합과 이름 결합 열을 나타내는 *Binding*, *BindingList*라는 타입을 정의한다. 이름 문맥은 *NamingContext*라는 인터페이스로 정의한다. 이름 문맥에 정의된 이름 결합들을 순차적으로 접근하기 위한 순환기(iterator)인 *BindingIterator*라는 인터페이스도 정의하고 있다.

[*Type*, *Interface*]

N

NameComponent, *Name* : *Type*
Binding, *BindingList* : *Type*
NamingContext, *BindingIterator* : *Interface*

이름 라이브러리는 사실상 라이브러리 함수로 구현되는데, OMG 표준에는 편의상 PIDL(pseudo IDL)를 사용하여 명세하고 있다. 각각 이름 구성 요소와 이름의 라이브러리로의 구현을 나타내는 인터페이스 *LNameComponet*, *LName*과 이들 가객체를 생성하기 위한 함수를 정의하고 있다.

[*Function*]

NamesLibrary

LNameComponent, *LName* : *Interface*
creat_lname, *creat_lname_component* : *Function*

아래에서는 CORBA IDL를 위한 Larch 방식의 BISL인 Larch/CORBA-IDL(LCB)를 사용하여 모듈 CosNaming과 NamesLibrary의 행위를 정형적으로 명세 한다[10].

3.1 이름 모듈

OMG에서는 이름 서비스를 지원하는 프로그램 모듈을 CosNaming이라 부르는데, 이는 NamingContext와 BindingIterator의 두 인터페이스로 구성된다. 본 절에서는 이 두 인터페이스를 정형적으로 명세 한다. 긴 명세의 효과적인 작성을 위하여 Noweb이라는 문학 프로그래밍 (literate programming) 도구를 사용하여 구조적으로 표시한다[12]. 예를 들면, 아래 CosNaming 모듈의 정의에서 이탈릭 폰트를 사용하여 ⟨ ⟩로 표시되는 부분은 실제 명세로 대치될 장소를 나타낸다.

[1] OMG의 CORBA IDL에서는 클래스를 인터페이스라고 부른다.

즉, ⟨*type def*⟩는 다음 단락의 ⟨*type def*⟩≡로 시작하는 텍스트에 의해 대치되어야 한다.

```
⟨CosNaming spec⟩≡
  module CosNaming
  {
    uses NamingContextTrait(NamingContext, Name,
           NameComponent),
         BindingIteratorTrait(BindingIterator,
           NamingContext for NC);
    ⟨type def⟩
    ⟨Interface NamingContext spec⟩
    ⟨Interface BindingIterator spec⟩
};
```

CosNaming 모듈의 수학적 추상 모형은 LSL 명세 NamingContextTrait와 BindingIteratorTrait에 의해 주어진다(제 3.2절과 3.3절 참조).

이름 서비스는 다음과 같은 타입을 정의하고 있다.

```
⟨type def⟩≡
  typedef string Istring;
  struct NameComponent {
    Istring id;
    Istring kind;
  };
  typedef sequence <NameComponent> Name;
```

구조체인 NameComponent는 이름 구성 요소를 나타내며, 이름을 나타내는 Name은 NameComponent의 열로 정의된다. Istring은 향후 IDL의 국제화를 위하여 미리 확보해 둔 문자 열 타입이다.

3.2 인터페이스 NamingContext

이름 문맥을 나타내는 인터페이스 NamingContext는 이름의 결합, 분해, 분리, 이름 문맥의 생성과 소멸, 이름 결합의 나열을 위한 오퍼레이션을 정의한다.

```
⟨Interface NamingContext spec⟩≡
interface BindingIterator;
interface NamingContext {
  ⟨exception def⟩
  ⟨binding objects⟩
  ⟨resolving names⟩
  ⟨unbinding names⟩
  ⟨creating naming contexts⟩
  ⟨deleting naming contexts⟩
  ⟨listing a naming context⟩
};
```

NamingContext의 수학적 모형은 (그림 4)에 정의되는데, 이름 문맥의 실제 모형은 NamingContextFlat이라는 LSL 명세로 주어 진다(부록 A 참조). NamingContextFlat은 평평한 이름 문맥 – 즉, 깊이가 1인 특수한 형태의 이름 그래프를 정의하고, NamingContextTrait는 일반적인 이름 그래프를 정의한다. 전자에 정의된 LSL 함수는 이름 구성 요소를 입력으로 갖는 반면 후자에 정의된 함수는 이름을 입력으로 갖는다. 일반적으로 후자 함수는 전자 함수를 사용하여 정의된다.

```
NamingContextTrait(NC,N,NE): trait
includes NamingContextFlat(NC,NE,Obj)
introduces
  bind: NC, N, Obj → NC
  bindCxt: NC, N, NC → NC
  unbind: NC, N → NC
  resolve: NC, N → Obj
  resolveCxt: NC, N → NC
  bound, boundCxt, boundAny: NC, N → Bool
asserts
  ∀c,c1: NC, n: N, e: NE, o: Obj
    ⟨axioms⟩
```

(그림 4) 인터페이스 NamingContext의 추상 모형
(Fig. 4) An LSL model for interface NamingContext

3.2.1 결합 연산자

결합 연산자는 이름 문맥 내에서 객체에 이름을 부여한다. 객체에 이름이 주어지면, 해당 객체를 분해 연산자를 사용하여 찾을 수 있다. 이름 서비스는 이름 부여를 위한 4 개의 연산자 – bind, rebind, bind_context, rebind_context를 제공한다. 연산자 bind는 주어진 이름 문맥 내에서 주어진 객체에 주어진 이름을 부여한다. 주의할 것은 매개 변수의 타입이 Object이므로 NamingContext 객체도 입력 값으로 사용될 수 있다는 것이다. 하지만 이렇게 결합된 이름 문맥 객체는 복합 이름의 이름 분해에는 사용되지 않는다(제 3.2.2절의 연산자 resolve의 명세 참조).

```
⟨binding objects⟩≡
void bind(in Name n, in Object obj)
  raises(NotFound, CannotProceed, InvalidName,
         AlreadyBound) {
  requires ¬isEmpty(n) ∧ boundCxt(self^,init(n))
           ∧ ¬boundAny(self^,n);
```

```
  modifies self;
  ensures self' = bind(self^,n,obj);
  ⟨error cases⟩
}
```

선 조건문은 이름 결합이 성공적으로 종결되기 위한 조건을 나타낸다. 즉, 이름은 하나 이상의 이름 구성 요소로 이루어지고, 마지막 이름 구성 요소를 제외한 접두어는 이름 문맥을 나타내고, 해당 이름 문맥 내에서 마지막 이름 구성 요소는 정의되지 않아야 한다. LSL 연산자 isEmpty와 init은 내장 LSL 명세 Sequence에 정의되며 [7, 부록 A], 각각 이름 열이 공집합인 지와 마지막 요소를 제외한 이름 열의 접두어를 나타낸다. NamingContextTrait에 선언된 LSL 함수 bind, boundAny, boundCxt, bound을 위한 공리는 다음과 같이 정의된다.

```
⟨axioms⟩≡
bind(c, ⟨e⟩,o) == bind(c,e,o);
bind(c,e ┤ n,o) ==
  bindCxt(c,e,bind(resolveCxt(c,e),n,o));
boundAny(c,n) == boundCxt(c,n) ∨ bound(c,n);
boundCxt(c,empty);
boundCxt(c,n ├ e) ==boundCxt(c, n) ∧ boundCxt(c,e);
¬bound(c,empty);
bound(c,n ├ e) ==
  boundCxt(c,n) ∧ bound(resolveCxt(c,n),e);
```

이름 문맥 관련 LSL 함수는 일정한 패턴으로 정의된다. 즉, 이름을 입력 값으로 갖는 NamingContext에 정의된 함수는 이름 구성 요소를 입력 값으로 갖는 NamingContextFlat에 정의된 함수를 사용하여 정의된다. 예를 들어, 첫째 줄과 둘째 줄에서 보는 바와 같이 bind:NC, N, Obj→NC의 공리는 bind:NC, NE, Obj→NC를 사용하여 정의된다. (LSL은 함수 이름의 중첩 사용(overloading)을 허용 한다.) 공리 정의에 사용된 LSL 연산자 ⟨⟩, ┤, ├은 내장 명세 Sequence에 정의되며, 각각 열, 열의 처음과 끝에 원소 첨가를 나타낸다[7, 부록 A]. 복합 이름이 주어질 때 연산자 bind는 아래와 같은 특성을 지님을 명세로부터 추론할 수 있다. 즉, 처음의 $n-1$개의 이름 구성 요소가 나타내는 이름 문맥 내에서 객체에 이름이 부여된다.

$$ctx \rightarrow bind(\langle e_1, e_2, ..., e_n\rangle, o) \equiv$$
$$ctx \rightarrow resolve(\langle e_1, e_2, ..., e_{n-1}\rangle) \rightarrow bind(\langle e_n\rangle, o)$$

결합 연산자 bind는 InvalidName, AlreadyBound, NotFound, CannotProceed와 같은 예외 상황을 일으킬 수 있다. 예외 상황 InvalidName은 입력 값으로 주어진 이름이 불합리할 경우에 발생된다. 예를 들면, 길이가 0인 이름을 들 수 있다.

```
⟨exception def⟩≡
exception InvalidName {};

⟨error cases⟩≡
requires isEmpty(n);
ensures raised(InvalidName);
```

LCB 내장 논리문인 **raised**는 해당 예외 상황이 발생하여 함수가 비정상적으로 종결됨을 나타낸다.

예외 상황 AlreadyBound는 주어진 이름이 이미 다른 객체에 부여되었음을 나타낸다.

```
⟨exception def⟩+≡
exception AlreadyBound {};

⟨error cases⟩+≡
requires ¬isEmpty(n) ∧ boundAny(self^,n);
ensures raised(AlreadyBound);
```

예외 상황 NotFound는 주어진 이름이 발견되지 않음을, 즉 그러한 이름 결합이 없음을 나타낸다. NotFound는 예외 상황의 이유를 나타내는 why와 예외 상황에 도달할 때까지 처리되고 남은 이름을 나타내는 rest_of_name의 두 요소로 구성된 구조체이다.

```
⟨exception def⟩+≡
enum NotFoundReason {missing_node,not_context,
                     not_object};
exception NotFound {
  NotFoundReason why;
  Name rest_of_name;
};

⟨error cases⟩+≡
requires ¬isEmpty(n) ∧ ¬boundCxt(self^,init(n));
ensures ∃ i:Int (0 < i ∧ i < len(n) ∧
  boundCxt(self^,prefix(n,i-1)) ∧
  ¬boundCxt(self^,prefix(n,i)) ∧
  (if bound(self^,prefix(n,i))
  then raised(NotFound).why = not_Context
  else raised(NotFound).why = missing_node) ∧
  raised(NotFound).rest_of_name = removePrefix(n,i-1));
```

LSL 함수 prefix는 이름 열에서 i 번째 원소까지로 구성되는 접두어를 되돌려 주는 함수로, LSL 명세

```
⟨exception def⟩+≡
exception CannotProceed {
   NamingContext cxt;
   Name rest_of_name;
};

⟨error cases⟩+≡
let rn be raised(CannotProceed).rest_of_name,
    pn be prefix(n,len(n) - len(rn));
requires ¬isEmpty(n);
ensures n = pn || rn ∧ boundCxt(self^,pn) ∧
   raised(CannotProceed).cxt = resolveCxt(self^,pn);
```

객체에 이미 이름이 결합되었을 경우, rebind 사용하여 새로운 이름을 결합할 수 있다. 연산자 bind와 마찬가지로 rebind로 이름이 결합된 NamingContext 객체는 복합 이름의 이름 분해 시 사용되지 않는다. NotFound, CannotProceed, InvalidName과 같은 예외 상황을 일으킬 수 있는데, 그 명세는 연산자 bind에서와 같다. 편의상 앞으로는 예외 상황을 명세하지 않는데, 생략된 예외 상황 명세는 연산자 bind에서와 같다.

```
⟨binding objects⟩+≡
void rebind(in Name n, in Object obj)
   raises(NotFound, CannotProceed, InvalidName) {
   requires ¬isEmpty(n) ∧ boundCxt(self^,init(n));
   modifies self;
   ensures self' = bind(self^,n,obj);
   ⟨error cases⟩
}
```

연산자 bind_context와 rebind_context는 각각 이름 문맥 객체에 이름을 결합, 재결합하며, 이렇게 이름이 결합된 이름 문맥은 복합 이름의 이름 분해 시 사용된다.

```
⟨binding objects⟩+≡
void bind_context(in Name n, in NamingContext nc)
   raises(NotFound, CannotProceed, InvalidName,
          AlreadyBound) {
   requires ¬isEmpty(n) ∧ boundCxt(self^,init(n))
            ∧ ¬boundAny(self^,n);
   modifies self;
   ensures self' = bindCxt(self^,n,nc);
}
void rebind_context(in Name n, in NamingContext nc)
   raises(NotFound, CannotProceed, InvalidName) {
   requires ¬isEmpty(n) ∧ boundCxt(self^,n);
   modifies self;
   ensures self' = bindCxt(self^,n,nc);
}
```

후조건문에 나타나는 LSL 함수 bindCxt는 LSL 명세 NamingContextTrait에 정의되고 해당 공리는 다음과 같다.

```
⟨axioms⟩+≡
bindCxt(c, ⟨e⟩,o) == bindCxt(c,e,o);
bindCxt(c,e ┤ n,o) ==
   bindCxt(c,e,bindCxt(resolveCxt(c,e),n,o));
```

복합 이름일 경우 bind_context는 다음과 같은 특성을 지님을 명세로부터 추론할 수 있다.

$$ctx \rightarrow bind\_context(\langle e_1, e_2, ..., e_n\rangle, o) \equiv ctx \rightarrow resolve(\langle e_1, e_2, ..., e_{n-1}\rangle)).bind\_context(\langle e_n\rangle, o)$$

3.2.2 분해 연산자

이름 분해를 위하여 연산자 resolve가 제공되는데, 주어진 이름 문맥 내에서 특정한 이름과 결합된 객체를 검색할 수 있도록 한다. 입력으로 주어진 이름은 결합된 이름과 동일해야 하며, 이름 서비스는 검색된 객체의 타입에 대한 정보를 제공하지 않는다. 사용자가 타입에 대한 책임을 진다. 즉, 필요하다면 사용자가 되돌려 받은 Object 타입의 객체를 더 정제된 타입으로 투영해야 한다. 예외 상황 NotFound, CannotProceed, InvalidName을 발생시킬 수 있는 데, 그 의미는 결합 연산자에서와 같다.

```
⟨resolving names⟩≡
Object resolve(in Name n)
   raises(NotFound, CannotProceed, InvalidName) {
   requires ¬isEmpty(n) ∧ boundAny(self^,n);
   ensures (bound(self^,n) ⇒
                result = resolve(self^,n)) ∧
           (¬bound(self^,n) ⇒
                result = resolveCxt(self^,n));
}
```

후 조건문에서 사용된 LSL 함수 resolve와 resolveCxt를 위한 공리는 다음과 같이 정의된다.

```
⟨axioms⟩+≡
resolve(c, ⟨e⟩) == resolve(c,e,o);
resolve(c,n ├ e) == resolve(resolveCxt(c,n),e);
resolveCxt(c, ⟨e⟩) == resolveCxt(c,e,o);
resolveCxt(c,n ├ e) == resolveCxt(resolveCxt(c,n),e);
```

복합 이름에서와 같이 이름이 복수 개의 이름 구성 요소로 이루어질 수 있기 때문에, 이름 분해는 여러 이름 문맥을 거치면서 이루어 질 수 있다. 따라서 복합 이름의 분해는 다음과 같이 정의됨을 알 수 있다.

$$ctx \rightarrow resolve(\langle e_1, ..., e_n \rangle) \equiv ctx \rightarrow resolve(\langle e_1, ..., e_{n-1} \rangle) \rightarrow resolve(\langle e_n \rangle)$$

3.2.3 분리 연산자

연산자 unbind는 특정한 이름 결합을 이름 문맥에서 제거한다. 즉, 주어진 이름과 결합된 객체의 이름 결합을 해체한다. 발생될 수 있는 예외 상황은 NotFound, CannotProceed, InvalidName인데, 그 의미는 결합 연산자에서와 같다.

```
⟨unbinding names⟩≡
void unbind(in Name n)
   raises(NotFound, CannotProceed, InvalidName) {
   requires ¬isEmpty(n) ∧ boundAny(self^,n);
   modifies self;
   ensures self' = unbind(self^,n);
}
```

LSL 함수 unbind을 위한 공리는 다음과 같다.

```
⟨axioms⟩+≡
unbind(c, ⟨e⟩) == unbind(c,e,o);
unbind(c,e ┤ n) ==
   bindCxt(c,e,unbind(resolveCxt(c,e),n));
```

복합 이름일 경우 unbind는 다음과 같은 특성을 지님을 알 수 있다.

$$ctx \rightarrow unbind(\langle e_1, ..., e_n \rangle) \equiv ctx \rightarrow resolve(\langle e_1, ..., e_{n-1} \rangle) \rightarrow unbind(\langle e_n \rangle)$$

3.2.4 생성 연산자

이름 서비스는 이름 문맥의 생성을 위하여 new_context와 bind_new_context라는 두 연산자를 정의한다. 연산자 new_context는 새로운 이름 문맥을 되돌려 주는데, 결과 이름 문맥은 현재 (즉, 연산자를 처리하는) 이름 문맥을 구현하는 이름 서버에 의해 구현되어야 한다. 연산자 new_context에 의해 생성된 이름 문맥은 이름을 갖지 않는다.

```
⟨creating naming contexts⟩≡
NamingContext new_context() {
   ensures result' =.empty ∧ fresh(result);
}
```

결과 객체의 초기 값은 이름 결합이 하나도 정의되지 않은 공 이름 문맥이다. LCB 내장 논리문 fresh는 해당 객체가 새로 생성되었음을 나타낸다. 즉, 함수 호출 직전인 전 상태(pre-state)에서는 존재하지 않으나 함수 호출 직후인 후 상태(post-state)에서는 존재함을 나타낸다.

연산자 bind_new_context는 새로운 이름 문맥을 생성할 뿐만 아니라 이름까지 부여한다. NotFound, AlreadyBound, CannotProceed, InvalidName과 같은 예외 상황을 발생할 수 있다.

```
⟨creating naming contexts⟩+≡
NamingContext bind_new_context(in Name n)
   raises(NotFound, AlreadyBound, CannotProceed,
          InvalidName) {
   requires ¬isEmpty(n) ∧ boundCxt(self^,init(n)) ∧
            ¬boundAny(self^,n);
   modifies self;
   ensures result' = empty ∧ fresh(result) ∧
           self' = bindCxt(self^,n,result);
}
```

생성된 이름 문맥에 새로운 이름을 결합하기 때문에, bind_new_context의 명세는 연산자 bind_contex와 new_context를 합친 형태이다. 복합 이름일 경우 다음과 같은 특성을 지님을 추론할 수 있다.

$$ctx \rightarrow bind\_new\_context(\langle e_1, ..., e_n \rangle) \equiv ctx \rightarrow resolve(\langle e_1, ..., e_{n-1} \rangle) \rightarrow bind\_new\_context(\langle e_n \rangle)$$

3.2.5 소멸 연산자

연산자 destroy는 해당 NamingContext 객체를 소

멸시킨다. 소멸될 이름 문맥이 이름 결합을 갖고 있으면, NotEmpty라는 예외 상황을 발생시킨다. 후조건문의 **trashed**는 해당 객체가 더 이상 사용될 수 없음을 나타내는 LCB 내장 논리문이다.

```
⟨exception def⟩+≡
exception NotEmpty {};

⟨deleting naming contexts⟩≡
void destroy() raises(NotEmpty) {
   modifies self;
   ensures if isEmpty(self^) then raised(NotEmpty)
           else trashed(self);
}
```

3.2.6 인터페이스 Iterator

경우에 따라서 이름 문맥에 정의된 모든 이름 결합을 순차적으로 접근할 필요가 있다. 이를 위하여 OMG 이름 서비스는 list라는 연산자와 BindingIterator라는 인터페이스를 정의하고 있다. 후자는 이름 문맥 객체를 위한 순환기(iterator)이다. 이름 결합을 표현하기 위하여 Binding과 BindingList라는 두 타입을 정의한다. Binding은 이름과 결합 종류를 나타내는 두 속성을 갖는 구조물이고, BindingList는 Binding의 열이다.

```
⟨type def⟩+≡
enum BindingType {nobject, ncontext};
struct Binding {
   Name binding_name;
   BindingType binding_type;
};
typedef sequence<Binding> BindingList;
```

NamingContext의 멤버 함수로 정의된 연산자 list는 이름 문맥 내의 모든 이름 결합을 순차적으로 접근할 수 있도록 한다.

```
⟨listing a naming context⟩≡
void list(in unsigned long how_many,
          out BindingList bl, out BindingIterator bi) {
   modifies bl, bi;
   ensures if size(toSet(self^)) > how_many
      then toSet(bl') ⊆ toSet(self^) ∧
           len(bl') = how_many ∧
           bi' = [self^,toSet(bl')]
      else toSet(bl') = toSet(self^) ∧ bi' = nil
}
```

즉, BindingList 타입의 출력 매개 변수 bl을 사용하여 요구한 개수 how_many 만큼의 이름 결합을 되돌려 준다. 만약, 요구한 수 보다 더 많은 이름 결합이 존재하면, 이는 출력 매개 변수 bi에 의해 BindingIterator 타입의 순환기로 되돌려 진다(BindingIterator 명세는 아래 참조). 그렇지 않을 경우, 매개 변수 bi는 공 객체를 나타내는 nil로 설정된다.

```
BindingIteratorTrait(BI): trait
assumes NamingContextTrait
includes Set(Binding, BS)
BI tuple of nc: NC, done: BS
introduces
   toSet: NC → BS
   toSet: BindingList → BS
   toSetPri: NC, Name → BS
   rest: BI → BS
asserts
   ∀i: BI, c, c1: NC, e: NameComponent, o: Obj,
      n: Name, l: BindingList, b: Binding
   rest(i) == toSet(i.nc) - i.done;
   toSet(c) == toSetPri(c,empty);
   toSetPri(bind(c,e,o),n) ==
      {[n ⊢ e,nobject]} ∪ toSetPri(c,n);
   toSetPri(bindCxt(c,e,c1),n) ==
      {[n,ncontext]} ∪ toSetPri(c,n ⊢ e);
   toSet(empty:BindingList) == {};
   toSet(b ⌐ l) == insert(b,toSet(l))
```

(그림 5) 인터페이스 BindingIterator의 LSL 명세
(Fig. 5) An LSL model for interface BindingIterator

순환기 BindingIterator의 추상 모형은 (그림 5)의 LSL 명세 BindingIteratorTrait로 주어진다. 순차적으로 접근할 이름 문맥(nc)과 해당 이름 문맥에서 이미 접근한 이름 결합들(done)을 나타내는 두 요소를 가지는 튜플로 표현한다. LSL에서 튜플 요소의 참조는 점 표기법을 사용하고 (예를 들면, bi.done), 수정은 set_ 함수를 사용한다 (예를 들면, set_done(bi, s)). 집합을 정의하는 LSL 명세 Set은 LSL Handbook [7, 부록A]에 정의된다. BindingIteratorTrait는 각각 이름 문맥, 이름 결합 열로부터 결합 집합을 도출하는 중첩(overloaded) 함수 toSet를 정의한다. 함수 toSetPri는 함수 toSet : NC → BS를 정의하기 위한 내부 함수이다.

인터페이스 BindingIterator는 이름 문맥을 위한 순환기로써, 사용자로 하여금 임의의 이름 문맥 내의 모든 이름 결합을 순차적으로 접근할 수 있도록 한다. 순환을 위한 연산자 next_one과 next_n, 소멸자 de-

stroy를 제공한다.

```
⟨Interface BindingIterator spec⟩≡
interface BindingIterator {
  boolean next_one(out Binding b) {
    modifies self, b;
    ensures if isEmpty(rest(self^))
      then ¬result ∧ unchanged(self)
      else result ∧ b' ∈rest(self^) ∧
           self= set_done(self^,insert(self^.done,b'));
  }
  boolean next_n(in unsigned long how_many,
                 out BindingList bl) {
    requires how_many > 0;
    modifies self, bl;
    ensures if isEmpty(rest(self^))
      then ¬result ∧ unchanged(self)
      else result ∧
        let s be toSet(bl')
          size(s) = min(how_many,size(rest(self^)))
           ∧ s ⊆ rest(self^) ∧
          self' = set_done(self^,self^.done ∪ s);
  }
  void destroy() {
    modifies self;
    ensures trashed(self);
  }
};
```

연산자 next_one은 이름 결합이 더 있을 경우 결과 매개 변수를 사용하여 다음 이름 결합을 되돌려 주고, 결과 값으로는 참을 되돌려 준다. 더 이상 이름 결합이 없을 경우에는 결과 값으로 거짓을 되돌려 준다. LCB 내장 논리 함수 **unchanged**는 해당 객체의 추상 값에 변화가 없음을 나타낸다. 이는 self가 **modifies** 절에 나열되었기 때문에 필요하다. 연산자 next_n은 next_one과 유사하나, 요구한 개수 만큼의 이름 결합을 되돌려 준다.

3.3 이름 라이브러리

기존 사용자에게 영향을 미치지 않고 이름 표현을 수정할 수 있도록 하기 위해서는 이를 사용자로부터 은닉하여야 한다. 이상적으로는 이름 자체가 CORBA IDL 객체가 되어야 한다. 반면, 이름 객체가 주기억 장치 내에서 효율적으로 생성되고 조작되기 위해서는 가벼운 존재여야 한다. 따라서 OMG 이름 서비스 명세에서는 이름 조작을 쉽게 하고 표현의 독자성을 제공하기 위하여, '이름 라이브러리(names library)' 형태로 제공할 수 있도록 한다. 이름 라이브러리는 이름을 '가객체(pseudo object)'로 구현한다. 가객체는 IDL 인터페이스를 통하여 전달할 수는 없으나, 일반 객체에서와 같은 방법으로 연산자를 호출한다. 이름 라이브러리는 이름 구성 요소와 이름을 위한 두 인터페이스를 제공한다. 제공되는 인터페이스를 사용하여 이름 구성 요소의 속성을 참조하고 수정할 수 있으며, 가객체를 실 객체로, 실 객체를 가객체로 변환할 수 있다.

이름 라이브러리는 이름 구성 요소와 이름을 위한 두 인터페이스 LNameComponent와 LName, 해당 객체를 생성하기 위한 함수를 정의한다. 사실상 이들은 CORBA IDL 인터페이스가 아니라 C 혹은 C++와 같은 구현 언어로 정의되어야 한다. 편의상 본 논문에서는 COSS에서와 같이 PIDL(pseudo-IDL)로 명세한다.

```
⟨NamesLibrary spec⟩≡
uses LNameComponentTrait(LNameComponent),
     LNameTrait(LName,LNameComponent);
⟨Interface LNameComponent⟩
⟨Interface LName⟩
⟨creation functions⟩
```

라이브러리 이름 구성 요소 LNameComponent와 이름 LName의 수학적 모형은 각각 LSL 명세 LNameComponentTrait와 LNameTrait에서 정의된다(해당 명세는 제 3.3.1절과 3.3.2절 참조).

3.3.1 인터페이스 LNameComponent

NameComponent 객체와 마찬가지로 LNameComponent는 이름과 종류의 두 속성을 가진다. 따라서 추상 값은 이름과 종류 속성을 나타내는 집합 {id, knd}에서 실제 값으로의 유한 사상으로 모형화 한다.

```
LNameComponentTrait(LNC): trait
includes FiniteMap(LNC,Tag,Str)
Tag enumeration of id, knd
```

인터페이스 LNameComponent는 이름과 종류 속성을 참조하고 수정하기 위한 연산자 get_id, set_id, get_kind, set_kind와 소멸 연산자 destroy를 제공한다.

참조 연산자는 해당 속성이 아직 설정되지 않았으면, NotSet이라는 예외 상황을 일으킨다.

```
⟨Interface LNameComponent⟩≡
interface LNameComponent { // PIDL
  exception NotSet{};
  string get_id() raises(NotSet)
    ensures if defined(self^,id)
              then result' = apply(self^,id)
              else raised(NotSet);
  }
  void set_id(in string s) {
    modifies self;
    ensures self' = update(self^,id,s);
  }
  string get_kind() raises(NotSet) {
    ensures if defined(self^,kind)
              then result' = apply(self^,kind)
              else raised(NotSet);
  }
  void set_kind(in string s) {
    modifies self;
    ensures self' = update(self^,kind,s);
  }
  void destroy() {
    modifies self;
    ensures trashed(self);
  }
};
```

3.3.2 인터페이스 LName

라이브러리 이름은 하나 이상의 라이브러리 이름 구성 요소로 구성된다. 따라서 라이브러리 이름 구성 요소의 열로 모형화 한다. 마지막을 제외한 이름 구성 요소는 서브 문맥을 결정하는데 사용되고, 마지막 구성 요소는 이름과 결합된 객체를 나타낸다.

```
LName(LN,LNC): trait
assumes LNameComponentTrait(LNC)
includes Sequence(LNC,LN)
introduces
  insert: LN, Int, LNC → LN
  delete: LN, Int → LN
  toIdlName: LN → Name
  toLibName: Name → LN
  __ =, < __: LN, LN → Bool
asserts
  ∀n: Name, l, l1: LN, i: Int, e: LNC
    insert(l,i,c) ==
      prefix(l,i) ‖ {e} ‖ removePrefix(l,i);
    delete(l,i) ==
      prefix(l,i) ‖ removePrefix(l,i+1);
    toIdlName(toLibName(n)) == n;
    toLibName(toIdlName(l)) == l;
```

LSL 함수 toLibName과 toIdlName은 가객체인 라이브러리 이름과 실객체인 CORBA IDL 이름 간의 변환을 나타낸다. 명세의 편의를 위해 insert, delete와 같은 몇몇 부수적인 함수도 정의된다.

인터페이스 LName은 라이브러리 이름을 조작하기 위한 다양한 종류의 연산자를 제공한다.

```
⟨Interface LName⟩≡
interface LName { // PIDL
  ⟨exception defs⟩
  ⟨component management⟩
  ⟨testing⟩
  ⟨converting to IDL form⟩
  ⟨destruction⟩
};
```

연산자 insert_component는 i번째 다음에 새 이름 구성 요소를 삽입한다. 만약 $i-1$번째 구성 요소가 정의되지 않았고 i가 1보다 크면, NoComponent라는 예외 상황을 일으킨다. 삽입될 구성 요소를 위한 자원을 할당할 수 없으면, OverFlow라는 예외 상황을 발생시킨다.

```
⟨exception defs⟩≡
  exception NoComponent{};
  exception OverFlow{};

⟨component management⟩≡
  LName insert_component(in unsigned long i,
                         in LNameComponent n)
  raises(NoComponent, OverFlow) {
  requires i ≤ len(self^);
  modifies self;
  ensures self' = insert(self^,i-1,n);

  requires i ≤ len(self^);
  ensures raised(OverFlow);

  requires i > len(self^);
  ensures raised(NoComponent);
}
```

연산자 get_component는 i번째 구성 요소를 되돌려 준다. 첫 번째 구성 요소는 1번, 두 번째는 2번 순으로 번호가 매겨진다. 해당 구성 요소가 없으면 NoComponent라는 예외 상황이 발생된다.

```
⟨component management⟩+≡
  LNameComponent get_component(in unsigned long i)
    raises(NoComponent) {
    ensures if len(self^) < i
              then raised(NoComponent)
              else result' = self^[i-1];
  }
```

연산자 delete_component는 i 번째 구성 요소를 제거한다. 해당 구성 요소가 존재하지 않으면, NoComponent라는 예외 상황을 발생한다. 연산자가 성공적으로 실행된 후에는 하나 적은 개수의 구성 요소를 가지며, $i+1$, $i+2$, ..., n 번째의 구성 요소는 각각 i, $i+1$, ..., $n-1$ 번째의 구성 요소가 된다.

```
⟨component management⟩+≡
  LNameComponent delete_component(
      in unsigned long i) raises(NoComponent) {
    requires len(self^) ≥ i;
    modifies self;
    ensures self' = delete(self^,i-1);

    requires len(self^) < i;
    ensures raised(NoComponent);
  }
```

연산자 num_components는 구성 요소의 개수를 되돌려 준다.

```
⟨component management⟩+≡
  unsigned long num_components() {
    ensures result = len(self');
  }
```

연산자 equal은 두 라이브러리 이름이 동일한 지를 검사한다. 라이브러리 이름의 대소는 less_than을 사용하여 알 수 있다. 이름간의 순서 결정은 구현 종속적이며,COSS 표준은 이를 명세하지 않고 있다. (따라서 LSL 연산자 =과 <을 위한 공리는 정의되지 않는다.)

```
⟨testing⟩≡
  boolean equal(in LName ln) {
    ensures result = (self^ = ln);
  }
  boolean less_than(in LName ln) {
    ensures result = self^ < ln;
  }
```

가객체는 CORBA IDL 인터페이스를 통하여 전달될 수 없다. 따라서 가객체인 LName과 IDL 객체인 Name 간의 변환을 위하여, to_idl_form과 from_idl_form이라는 두 연산자를 제공한다.

```
⟨exception defs⟩+≡
  exception InvalidName{};
```

```
⟨converting to IDL form⟩≡
  Name to_idl_form() raises(InvalidName) {
    ensures if isEmpty(self^)
            then raised(InvalidName)
            else result' = toIdlName(self^);
  }
  void from_idl_form(in Name n) {
    modifies self;
    ensures self' = toLibName(n);
  }
```

연산자 to_idl_form은 해당 이름이 불합리할 경우 – 예를 들면, 길이가 0일 경우, InvalidName이라는 예외 상황을 일으킨다.

라이브러리 이름 가객체는 destroy 연산자에 의해 소멸된다.

```
⟨destruction⟩≡
  void destroy() {
    modifies self;
    ensures trashed(self);
  }
```

3.3.3 생성 함수

라이브러리 이름과 이름 구성 요소를 생성하기 위하여 create_lname과 create_lname_component라는 함수를 제공한다. 생성되는 이름의 추상 값은 empty – 즉, 이름 구성 요소를 하나도 갖지 않는 이름 열이고, 이름 구성 요소는 이름과 종류 속성 모두 정의되지 않은 객체 – 즉, 공 함수이다.

```
⟨creation functions⟩≡
LName create_lname() {
  ensures fresh(result) ∧ result' = empty;
}
LNameComponent create_lname_component() {
  ensures fresh(result) ∧ result' = {};
}
```

4. 결 론

정형 방법은 OMG 표준화와 OMG 기반의 분산 객체 시스템 개발에 매우 중요한 역할을 할 것으로 기대된다. 표준 문서에 내재한 모호성과 부정확성의 제거를 제쳐두고서라도 OMG 객체 관리 시스템을 사용하는 명세자와 개발자에게 정형화된 지침서를 제공한다. OMG 표준의 정형화는 좋은 표준이 제공해야

할 여러 혜택 외에 정형 방법의 장점을 OMG 기본 틀 안에서 제공한다. 예를 들면, 표준의 수학적 검증과 표준 만족에 대한 정형적 증명을 가능하게 할 뿐만 아니라 의미에 바탕을 둔 재사용 부품의 검색과 합성을 가능하게 한다. 또한, 표준을 만족하는 지에 대한 테스트 자동화 도구 개발의 기반이 된다[5]. 정형화된 검사치를 표준에서 자동으로 추출하고 이를 바탕으로 하여 만족성 검사를 수행할 수 있다.

최근 들어 표준의 정형화에 대한 연구가 OMG와 ISO ODP를 주축으로 하여 활발히 진행되고 있기는 하나 [2][3][6][8][11][14], 대부분이 개념과 이론적 토대 정립에 치중되어 있어 정형적으로 명세 된 표준의 실례를 찾아 보기는 힘들다. 특히, OMG 표준 객체의 행위와 인터페이스에 관한 명세는 전무한 상황이다. 따라서 본 논문에서는 OMG에서 제정한 객체 서비스 중 이름 서비스 명세를 Larch를 사용하여 행위적 인터페이스 측면에서 정형화 하였다. 아래에서는 정형화 과정에서 얻은 몇몇 교훈을 소개하고 향후 연구 과제와 함께 OMG와 Larch에 대해 조언한다.

이미 여러 문헌에서 지적된 바와 같이 한 정형 방법으로는 표준 정형화에서 발생하는 모든 요구 사항을 만족하기가 불가능하다[14]. 정형 방법의 가능성을 충분히 활용하기 위한 바람직한 접근 방법은 여러 방법을 조화롭게 결합하여 사용하는 것이다. 예를 들어, 본 논문에서는 Z과 Larch를 혼합하여 사용하였다. OMG 이름 서비스의 핵심 개념 및 용어와 서비스 모듈의 전체 구조는 수학적 의미가 명확하고 표현이 간결하며 구조를 가시화할 수 있는 Z 명세 언어를 사용하였다. 반면 Z 언어는 CORBA IDL로 표현되는 객체의 행위적 인터페이스를 정확히 명세할 수 없기 때문에, 단위 모듈의 인터페이스와 행위 명세를 위해서는 Larch/CORBA-IDL(LCB)를 사용하였다. Larch의 장점은 인터페이스를 정확하게 기술할 수 있을 뿐만 아니라 LSL를 사용하여 추상화 정도를 조율할 수 있다는 것이다.

서론에서 잠시 언급한 바와 같이 표준 정형화의 출발점은 수학적 토대 정립이다. 표준에서 사용하고 있는 개념과 기본 골격에 대한 확고한 이해 및 수학적 바탕 없이 정형화를 시도하는 것은 사상누각에 불과하다. 먼저 핵심 개념을 간결, 정확, 명료하게 정의하고 이를 바탕으로 기본 구조와 세부 사항들을 정형화하는 것이 바람직하다. 본 논문에서 먼저 Z를 사용하여 OMG 이름 서비스 개념을 정의한 것도 이러한 이유에서이다. 표준의 모든 내용을 정형화 하는 것은 불가능할 뿐만 아니라 경우에 따라서는 비정형적으로 기술하는 것이 더 효과적일 수도 있다. 따라서 세부 내용에 앞서 어떤 부분을 정형화하는 것이 효과적이고 바람직한가에 대한 연구가 선행되어야 한다. 정형과 비정형을 조화롭게 혼합하여 사용하는 것도 바람직한 방법이다. 나아가서는 정형 명세 언어에 비정형 기법의 통합도 시도할 수 있다. 예를 들면, Larch에서는 이를 위하여 informally 절이 제안되고 있다. 통합에 대한 체계적인 연구가 필요하다. 명세 작성시 문법, 구문 오류에서부터 정적, 동적 의미 오류까지 다양한 종류가 오류가 나타날 수 있다. 명세 오류를 찾고 제거하는 것은 Z이나 Larch와 같이 실행 불가능한 언어에서는 더욱더 힘들고 번거로운 작업이다. 따라서 적절한 지원 도구를 사용하는 것이 바람직하다. 현재 LCB를 위한 지원 도구가 개발되어 있지 않으므로, 본 논문에서는 명세의 구문 검사를 위하여 LCB 명세를 전처리하여 Larch/C++로 번역하고 Larch/C++ 타입 검사기를 사용하였다. LCB가 표준화 기구에 의해 채택되고 사용되려면 지원 도구에 대한 연구와 개발이 있어야 한다.

정형 명세의 가장 큰 장점은 수학적 증명이 가능하고 명세의 기계적 처리가 가능하다는 것이다. 본 논문에서와 같이 표준을 정형화 한다는 노력 자체만으로도 많은 혜택－예를 들면, 비정형 문서에서는 숨겨졌던 많은 사항이 들어 남－을 얻을 수 있으나, 이를 극대화하기 위해서는 표준의 특성을 검증하고 증명할 수 있는 방법과 도구에 대한 연구가 필요하다. 명세의 모의 실험, 애니메이션, 프로토타입 생성, 정형 증명 등을 들 수 있다. 특히, Larch 방식의 행위 명세는 정형 증명에 있어서 흥미로운 기회를 준다. Larch는 LP라 불리는 LSL를 위한 논리 증명기를 제공하는데 [7, 제 5장], LCB로 작성된 명세의 특성 검증에도 사용할 수 있을 것으로 본다. 즉, LCB 명세를 LSL로 변환하고 증명하고자 하는 특성을 LSL로 기술한 다음, LP를 사용하여 기계적으로 증명한다. 건전한 증명이 되기 위해서는 LCB의 정형 의미론을 LSL를 사용하여 모형 지향적으로－즉, LSL 변환 규칙으로 정의하고, 정의된 규칙이 건전하고 완전하다는 것을 보

여야 한다. 인터페이스 명세 증명에 LP를 사용하는 것은 흥미로운 향후 연구 과제가 될 것으로 기대한다. 명세로부터 프로토타입을 자동적으로 생성하고 정형 방법을 적용하여 표준을 만족하는 구현을 자동 개발하는 것도 중요한 향후 연구 주제이다.

참 고 문 헌

[1] 천윤식, 김창갑. CORBA 분산 객체의 행위 명세. *한국정보처리학회 '95 추계학술발표논문집, 1995년 10월 14일, 건국대학교*, 제 2권 2호, 272-276쪽, 1995.

[2] Howard Bowman, John Derrick, Peter Linington, and Maarten Steen. FDT for ODP. *Computer Standards & Interfaces*, 17(5-6):457-479, September 1995.

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

[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] J. L. Crowley, J. F. Leathrum, and K. A. Liburdy. Issues in the Full Scale Use of Formal Methods for Automated Testing. *ACM SIGSOFT Software Engineering Notes*, 21(3):71-78, May 1996.

[6] Elsepth Cusack and Gregor von Bochmann. Formal Object-Oriented Methods in Communication Standards. *OOPS Messenger*, 3(2):7-8, April 1992.

[7] John V. Guttag and James J. Horning. Larch: Languages and Tools for Formal Specification. Springer-Verlag, New York, N.Y., 1993.

[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] Doug Lea and Jos Marlowe. Interface-Based Protocol Specification of Open Systems Using PSL. In W. Olthoff, editor, *ECOOP '95, European Conference on Object-Oriented Programming, Aarhus, Denmark, August 7-11, 1995, Proceedings*, volume 952 of Lecture Notes in Computer Science, pages 374-398. Springer-Verlag, 1995.

[10] 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.

[11] Elie Najm and Jean-Bernard Stefani. A Formal Semantics for the ODP Computational Model. *Computer Networks and ISDN Systems*, 27(8): 1305-1329, July 1995.

[12] Norman Ramsey. Literate Programming Simplified. *IEEE Software*, 11(5):97-105, September 1994.

[13] 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.

[14] Richard O. Sinnott and Kenneth J. Turner. Applying Formal Methods to Standard Development: The Open Distributed Processing Experience. *Computer Standards & Interfaces*, 17(7): 615-630, October 1995.

[15] Richard Mark Soley, editor. *Object Management Architecture Guide*. John Wiley & Sons, Inc., third edition, September 1995. Revision 3.0.

[16] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, New York, N.Y., 1989.

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

[18] The Object Management Group, Inc. *CORBAservices: Common Object Service Specification*. The

Object Management Group, Framingham, MA, March 1995. OMG Document Number 95-3-31.

부록A. LSL 명세 NamingContextFlat

NamingContextFlat은 평평한 이름 문맥 – 즉, 이름 구성 요소에서 해당 객체로의 함수를 정의한다. 이름에 결합될 수 있는 객체가 일반 객체 뿐만 아니라 이름 문맥이 될 수도 있기 때문에, 일반 객체 결합을 나타내는 obj와 이름 문맥 결합을 나타내는 cxt의 두 함수로 모형화한다.

```
⟨NamingContextFlat⟩ ≡
NamingContextFlat(NC, E, Obj): trait
includes FiniteMap(OM, E, Obj), FiniteMap(CM, E, NC)
NC tuple of obj: OM, cxt: CM
introduces
    bind: NC, E, Obj → NC
    bindCxt: NC, E, NC → NC
    unbind: NC, E → NC
    resolve: NC, E → Obj
    resolveCxt: NC, E → NC
    isEmpty: NC → Bool
    bound, boundCxt, boundAny: NC, E → Bool
asserts
  ∀c: NC, e: E, o: Obj, n: NC
    bind(c, e, o) == set_obj(c, update(c.obj, e, o));
    bindCxt(c, e, n) == set_cxt(c, update(c.cxt, e,n));
    unbind(c, e) == set_obj(c, unbind(c.obj, e));
    unbindCxt(c, e) == set_cxt(c, unbind(c.cxt, e));
    resolve(c, e) == apply(c.obj, e);
    resolveCxt(c, e) == apply(c.cxt, e);
    isEmpty(c) == [{}, {}];
    bound(c, e) == defined(c.obj, e);
    boundCxt(c, e) == defined(c.cxt, e);
    boundAny(c, e) == bound(c, e) ∧ boundCxt(c, e)
```

김 미 희

1989년 숙명여자대학교 전산학과 졸업(학사)
1989년~현재 한국전자통신연구원 연구원
1997년~현재 충남대학교 컴퓨터과학과 석사과정
관심분야: 고속 통신망, 이동 통신망, 분산 처리, 소프트웨어 공학