UML 정적구조 다이아그램으로부터 LOTOS 명세 생성

김 철 홍[†] · 안 유 환[††] · 이 원 천[†††]

요 약

객체지향 기법과 정형 기법은 미래의 소프트웨어 공학 분야에 지대한 영향을 미칠 잠재력을 가진 두 갈래의 큰 영역으로 인식되고 있다. 따라서 이 두 영역의 접목 "객체지향 기술을 이용한 시스템 명세의 정형적 접근"은 빠른 속도로 성장하고 있으며 많은 연구결과를 산출하고 있다. LOTOS는 객체 기반 접근에 매우 적절하나, 완전한 객체지향 접근 방법을 제공하기 위하여 일반화(상속과 다형성)를 모델링할 수 있어야 한다. 이러한 주제를 연구해온 대부분의 연구자들은 LOTOS을 확장을 제안하였다. 본 논문은 ISO 8807 LOTOS로의 변환에 관심을 두며, 이러한 연구 동향의 일환으로 UML 정적구조 다이아그램으로부터 LOTOS 명세를 생성하는 방법을 제안한다.

Generating LOTOS Specifications from UML Static Structure Diagrams

Cheol-Hong Kim[†] · Yu-Whoan Ahn[††] · Won-Chun Lee[†††]

ABSTRACT

It is recognized that object-oriented methods and formal methods are two different main streams that will influence on the future direction of software engineering. A merging effort on these two technologies, named "a formal approach on system specifications using object-oriented methods" emerges rapidly and produces remarkable research results. LOTOS is well-suited to an object-based approach. However, to provide a full object-oriented approach, we need to model generalization (i.e. inheritance and polymorphism). Most authors who have examined this topic have proposed extensions to LOTOS. As an extension of such an effort, this paper proposes a method that generates LOTOS specification from static structure diagrams in UML.

1. 서 론

객체지향 분석 설계 방법이 산업계에서 시스템 개발에 실질적으로 사용되고 있는 반면 정형기법(Formal Method)은 주로 학계 및 연구소에서 연구를 목적으로 사용되고 있다. 정형기법이 산업계에 널리 보급되지 않는 이유로는 학문의 미성숙, 교육의 부족, 지원도구의 결여와 같은 여러 요인이 지적되고 있으나 근본적인 원인은 대부분의 개발자가 정형기법의 필요성을 인식하지 못하고 있기 때문이다. 따라서 이미 필요성이 인식되고 있는 것과 연계되어 보급되어야 한다. 예를 들면, 테스팅과 연계 등을 들 수 있다.

정형기법은 엄격한 수학적 바탕을 가진 정형명세 언

† 정 회 원 : 한국전자통신연구원 선임연구원
†† 정 회 원 : 한국전자통신연구원 S/W품질보증연구팀 팀장
††† 정 회 원 : 한국전자통신연구원 연구원
논문접수 : 1999년 7월 14일, 심사완료 : 1999년 12월 4일

어를 사용하여 시스템을 명세하고, 이러한 명세는 애매모호함이 없는 간결하고 정확한 표현으로 이루어져 있으므로 수학적 검증 및 확인이 가능하다. 따라서 객체지향 기법인 UML로 작성된 비정형적 명세를 LOTOS와 같은 정형명세 언어로 변환하고, 실행가능한 프로그램 코드를 생성함으로써 개발자는 이 정형언어를 기반으로 자신이 작성한 요구명세를 개발초기에 테스팅하거나 시뮬레이션함으로써 오류를 초기에 발견하여 시스템의 신뢰성을 높일 수 있다.

본 논문에서는 UML 모델의 정적 구조와 클래스간의 관계, 타입과 객체들을 보여주는 UML 정적 구조 다이아그램을 고찰하고, UML 구조 다이아그램과 LOTOS간에 개념적 변환 방법을 제시한다. 또한 변환 규칙에 따라 UML 다이아그램으로부터 LOTOS 언어를 생성하는 실제 적용 사례를 제시한다.

2. 관련 연구

2.1 UML 정적 모델

UML 정적구조 다이아그램을 작성할 때 기술할 수 있는 것이 무엇인지를 명확하게 하는 것이 중요하다. Cook과 Daniels[1], Fowler[6]에 의하면 세 가지 관점이 있다.

Conceptual : 다이아그램이 문제 영역에 있는 엔티티들을 나타낸다.

Specification : 문제 해결 영역에 있는(S/W와 H/W) 엔티티들의 영향을 명세하고, 그들간의 인터페이스에 관심을 둔다.

Implementation : 내부 표현과 알고리즘에 관심을 둔다.

비록 각 관점들이 다른 영역에 관심을 두고 있어도, Conceptual과 Specification 모델은 비슷한 점이 많다. Specification 모델에서 한가지 다른 점은 제시된 시스템과 외부 환경간에 경계점을 찾아내는 것이다. LOTOS가 Specification과 시스템 개발의 상위 설계 단계에 사용됨에 따라 본 논문은 작성자들이 UML 모델을 작성할 때, Conceptual 또는 Specification 관점을 취한다고 가정한다.

클래스 다이아그램은 주요한 정적 구조 다이아그램이고 UML 모델의 정적 구성요소와 그들간에 관계를 보여준다. 클래스 다이아그램은 LOTOS 명세에서 나타낼 엔티티들을 표현하고, 변환을 위한 주요 입력 자료가 된다. 클래스 다이아그램은 병렬로 존재하는 클래스 인스턴스를 표현하지 못한다. LOTOS 명세도 병행하여 발생하는 객체를 표현할 수는 있어도, 실제로 구현이 동시에 일어나는 것을 의미하는 것은 아니다.

정적 객체 다이아그램은 클래스가 아닌 객체를 포함하는 다이아그램이다. 객체 다이아그램의 값은 최상위 수준의 LOTOS 행위 표현식에서 표현되는 클래스의 인스턴스로 변환된다.

2.2 LOTOS

LOTOS(Language Of Temporal Ordering Specification)는 개방형 분산 시스템 특히 OSI의 서비스와 프로토콜을 명세하기 위하여 ISO에 의하여 개발된 정형명세 언어이다[15]. LOTOS의 바탕이 되는 개념은 시스템 명세는 외부 환경과 일어나는 상호작용간의 시간적인 관계를 정의함으로써 이루어질 수 있다는 것이다. 따라서 LOTOS는 프로세스 대수학에 바탕을 두고 있으며, 이를 위하여 Milner의 CCS와 Hoare의 CSP를 도입하였다. 상호작용시 자료의 교환이 일어날 수 있는데 이러한 자료의 구조와 값을 표현하기 위하여 ACT ONE[16]을 사용한다. ACT ONE은 추상자료형을 정의하기 위한 대수적 방식의 정형명세 언어이다.

LOTOS는 객체기술이 널리 사용되기 이전에 만들어졌기 때문에 설계시 객체지향 개념이 고려되지 않았다. 그럼에도 불구하고 캡슐화, 추상화 및 정보 은닉과 같은 객체지향 방식의 명세에 쓰이는 기본적인 개념을 지원하고 있다.

2.3 비정형 기법과 정형기법 통합 연구

학계와 연구계에서 산업체로의 기술이전을 위하여 정형기법(Formal Method)을 전통적인 비정형적 기법과 결합하려는 여러 연구가 진행되고 있다[8]. 이론적 토대를 마련하기 위하여 비정형적 기법에서 사용되는 표기법에 수학적 의미를 부여하려는 노력도 많이 있다. 예를 들면, Bourdeau는 OMT 객체 모형의 의미론을 대수 방식의 Larch 명세언어를 사용하여 정의하였으며[9], Moreira는 LOTOS로 객체지향 개념을 정형화하려 했다[10]. 이론적 토대를 바탕으로 하여 다양한 형태의 결합 및 통합이 시도되고 있다. 예를 들면, Statechart와 Z[11], SA와 VDM[7,8], OOA와 LOTOS[10]의 통합을 들 수 있다. 이밖에도 OOA&D와 CORBA IDL, Larch

등을 이용한 연구가 있다.

Toetenel은 구조적 분석과 객체지향 설계에 VDM을 도입하는 연구를 하였다[7]. 기본 개념은 DFD, ERD와 같은 SA의 결과물을 VDM 명세로 변환하고 자료명세를 사용하여 상위의 추상명세를 하위의 구체 명세로 반복적으로 정제한다는 것이다. 정제 과정은 궁극적으로 오퍼레이션 분할을 통해 자동변환될 수 있는 명세를 도출한다. 이를 위해 VDM-SL를 객체지향 구조와 장치를 사용하여 확장하였다. Toetenel의 방법은 각 단계가 DFD, ERD, VDM-SL과 같이 상이한 표현을 사용하므로 응집력이 없고 매끄럽지 않다. 또한 여러 표기법을 알아야 하고, 표기법 변화에 따른 어려움이 있다.

기존의 정형명세 언어에 객체지향 개념을 지원하도록 확장하는 언어 연구도 진행되고 있다. 예를 들면 Z을 바탕으로 한 Object Z, Z++, OOZE 등과 VDM을 바탕으로 한 Fresco, VDM++, OOVDM 등이 있다[12, 13], 이들은 모델지향적 방법으로 자료의 모형화에 중점을 두었다.

LOTOS를 객체지향 개념으로 확장한 OOLOTOS에 대한 연구도 진행되었다[14]. 본 OOLOTOS에서는 자료와 제어를 객체지향 개념안에 통합하도록 노력하였다. 또한 OOLOTOS와 OMT 객체 모형을 통합하는 연구도 함께 진행되었다. 통합의 바탕은 시스템의 전체 구조는 객체모형을 사용하여 그래픽적으로 나타내고 객체의 구조와 동적 행위는 OOLOTOS를 사용하여 정형적으로 명세하자는 것이다.

본 논문은 이러한 비정형적 기법과 정형적 기법의 통합을 위한 노력의 일환으로서 OMG에서 표준으로 채택되어 널리 사용되고 있는 UML의 정적구조다이아그램을 LOTOS로 변환하기 위한 시도이다.

3. UML로부터 LOTOS 생성

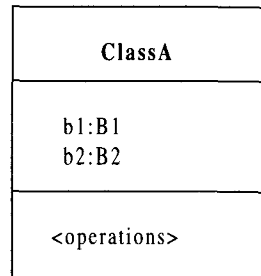
3.1 클래스

클래스는 비슷한 구조와 행위 및 관계를 갖는 객체 집합의 descriptor이다. 클래스 다이아그램에서 클래스는 세 구역으로 나누어진 이름, 속성, 오퍼레이션을 갖는 직사각형으로 그려진다.

UML 다이아그램에서 클래스는 객체의 특성들을 정의한다. 객체와 속성의 차이점은 객체는 identity와 상태를 갖는 반면에 속성은 단지 값만을 갖는다. 객체를 정의하는 클래스는 프로세스로 표현되고 속성을 정의하는 클래스는 ADT로 표현된다.

속성은 프로세스의 매개변수로 표현한다. 객체 식별자는 독립된 매개변수로 표현하고, 나머지 속성들은 객체 상태를 나타내는 ADT로 정의한다. 클래스를 정의하는 프로세스는 두 개의 매개변수를 갖는다. 이 방법은 객체 상태를 정의하는 ADT가 합성 컴포넌트를 정의하는 ADT와 같은 구조이므로 큰 장점을 가지고 있다.

UML 클래스 다이아그램에서 클래스는 행위를 정의하지 않는 반면, 생성되는 LOTOS 프로세스는 행위를 정의한다. 이러한 이유로 추가적인 정보가 다른 다이아그램으로부터 추출되거나 클래스 노드가 구조화된 annotation 형태의 추가 정보를 포함해야 한다. 이러한 정보가 없으면, 단지 개략적인 정의만 생성된다. 클래스 명이 ClassA 일 때, 클래스의 상태는 소트(sort) ClassA_State를 정의하는 ADT ClassA_State_Type으로 정의된다.



(그림 1) 클래스 구조

이러한 방법을 사용하여, (그림 1)의 클래스는 다음과 같이 LOTOS ADT와 프로세스로 변환된다.

```
type ClassA_State_Type is B1_Type, B2_Type
   sorts ClassA_State
   opns
      the_ClassA_State: -> ClassA_State
      make_ClassA_State: B1, B2 -> ClassA_State
      get_b1: ClassA_State -> B1
      get_b2: ClassA_State -> B2
      <other operations>
   eqns
      forall classA_State: ClassA_State
      ofsort B1
         get_b1(classA_State) = the_B1;
      ofsort B2
         get_b2(classA_State) = the_B2;
      <other equations>
endtype (* ClassA_State_Type *)

process ClassA[g](id: ClassA_Id, s: ClassA_State): noexit :=
   <process body>
endproc (* ClassA *)
```

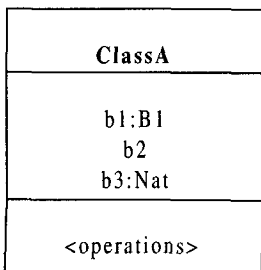
위의 명세는 상태를 위한 *symbolic* ADT를 정의한다. 소트 ClassA_State는 두 개의 생성자를 가지고 있는데 make_ClassA_State는 클래스 속성으로부터 ClassA_State의 인스턴스를 생성하고, the_ClassA_State는 클래스 ClassA의 객체의 상태를 위한 symbolic 값을 생성한다. 소트 B1과 B2 그리고 symbolic 값 the_B1과 the_B2는 타입 B1_Type과 B2_Type으로 정의된다. selector 오퍼레이션 get_B1과 get_B2를 통해 각 상태 컴포넌트, 즉 클래스 속성에 접근할 수 있다. 소트 ClassA_State는 단지 프로세스 정의 ClassA와 관련하여 사용된다.

```
process ClassA[g](id: ClassA_Id, s: ClassA_State): noexit :=
{   < event for operation 1 >
    exit( < new ClassA_State > )
  []
    < event for operation 2 >
    exit( < new ClassA_State > )
  []
    ...
} >> accept new_s; ClassA_State
       in ClassA[g](id, new_s)
endproc (* ClassA *)
```

클래스가 함성(composition)의 컴포넌트를 표현할 때, 클래스 A는 소트 A를 정의하는 ADT_Type으로 정의된다. ADT A_Type은 위에서 보여준 ClassA_State_Type과 동일한 구조를 갖는다.

3.2 속 성

클래스 인스턴스의 내부 상태는 속성 값으로 표현된다. 각 클래스는 객체 식별자를 갖고 있으므로 클래스의 인스턴스는 유일하게 식별될 수 있다. 클래스를 LOTOS로 변환할 때, UML 클래스의 속성 정의는 클래스의 내부 상태를 정의하는 소트로 구성된 ADT로 표현한다. 클래스를 정의하는 LOTOS 프로세스에서 객체 식별자와 상태는 프로세스의 매개변수로 표현된다.



(그림 2) 클래스 구조

UML에서 속성은 public, protected 또는 private인지를 표시하기 위해 시각적 표시자를 사용한다. LOTOS에서 속성은 프로세스 매개변수로 표현되며, 직접 시각적으로 나타나지 않는다. UML은 상세한 속성 타입은 정의하지 않고 목표 언어에 의존한다. 속성 타입은 LOTOS의 소트로 변환된다. 만약 속성에 타입이 없으면, UNKNOWN 소트를 갖는 LOTOS 명세로 표현된다. 다른 클래스에 대한 참조는 관계로 모델링된다.

(그림 3)의 UML 클래스에서 생성된 ADT는 다음과 같다.

```
type ClassA_State_Type is B1_Type, UNKNOWN_Type, NaturalNumber
   sorts ClassA_State
   opns
      the_ClassA_State: -> ClassA_State
      make_ClassA_State: B1, UNKNOWN, Nat-> ClassA_State
      get_b1: ClassA_State -> B1
      get_b2: ClassA_State -> UNKNOWN
      get_b3: ClassA_State -> Nat
      <other operations>
   eqns
      forall classA_State: ClassA_State
      ofsort B1
         get_b1(classA_State) = the_B1;
      ofsort UNKNOWN
         get_b2(classA_State) = the_UNKNOWN;
      ofsort Nat
         get_b3(classA_State) = the_Nat;
      <other equations>
endtype (* ClassA_State_Type *)
```

- 객체 식별자

각 객체는 다른 객체와 명확히 구별된다. 두 개의 객체는 같은 상태 값을 갖고 같은 서비스를 제공할 수 있으나 다른 객체이다. 객체들은 LOTOS 명세에서 객체 식별자로 구별되는데, 객체 식별자의 소트(sort)는 객체의 클래스를 결정하고, 값은 같은 클래스의 인스턴스를 구별하는데 사용된다. 본 논문에서는 각 클래스가 다중 인스턴스를 갖는다고 가정하고 변환한다. 클래스 ClassA가 하나의 인스턴스를 가질 때, 식별자를 정의하는 소트를 다음과 같이 정의할 수 있다.

```
type Class_Id_Type
   sorts ClassA_Id
   opns the_ClassA_Id -> ClassA_Id
endtype (* Class_Id_Type *)
```

클래스가 하나의 인스턴스를 가지고 있을 때, 객체 식별자가 필요 없으나, 다수의 인스턴스를 가지고 있

을 때, 객체 식별자를 정의하고 구분하는 메카니즘이 필요하다. 소트 Id가 식별자 값의 set을 정의한 타입 Id_Type을 다음과 같이 정의할 수 있다.

```
id, succ(id), succ(succ(id))
```

클래스가 다수의 인스턴스를 가지고 있을 경우, 현재 객체의 set을 알고 있어야 한다. 그러기 위하여 매개변수화된 라이브러리 ADT set을 개체화하는 다음과 같은 타입 Set_Id_Type이 필요하다.

```
type Set_Id_Type is Set actualizedby Id_Type using
   sortname Id for Element
            Bool for FBool
endtype (* Set_Id_Type *)
```

- 객체 생성자

객체지향 언어는 클래스의 새로운 인스턴스를 생성하기 위한 생성자를 가지고 있다. 클래스가 다수의 인스턴스를 가질 수 있는데, 이 때는 동적으로 새로운 객체를 생성할 수 있는 LOTOS 매카니즘이 있어야 한다. 이를 위해 객체 생성자를 사용한다. 클래스 ClassA을 위한 객체 생성자를 정의하고자 한다면 다음과 같은 LOTOS 명세를 정의한다.

```
process ClassA[g](id: ClassA_Id, s: ClassA_State): noexit :=
   <process body>
endproc (* ClassA *)
```

위의 정의를 변화시키지 않는 객체 생성자 매카니즘이 필요한데, 즉 객체를 정의하는 프로세스는 인스턴스가 하나이든 다수이든 관계없이 독립적이어야 한다. 이름이 복수인 객체 생성자를 다음과 같이 정의하여 이러한 문제를 해결한다.

```
process ClassAs[g](ids: ClassA_Set): noexit :=
   g !create ?id: ClassA_Id ?s: ClassA_State [id notin ids];
   ( ClassA[g](id, s)
     |||
     ClassAs[g](Insert(id, ids))
   )
 endproc (* ClassAs *)
```

매개변수 ids는 이미 할당된 객체 식별자 set을 제공하고, 객체 생성자를 개체화할 때, set을 빈 공간으로 초기화한다. 클래스 ClassA의 객체는 객체 생성자가 제공하는 create 서비스와 동시 발생하는 다른 객체에 의해 생성된다. 매개변수들이 새로운 객체의 상태를 초기화하기 위해 전달된다. 객체 생성자 ClassAs는 매개변수 ids에 이미 할당된 객체 식별자 set을 가지고 있다. *selection predicate* 조건식인 [id notin ids]는 기존의 것과 다른 새로운 객체 생성자 값을 생성한다.

3.3 오퍼레이션

UML 클래스 다이아그램의 클래스는 오퍼레이션의 시그니처를 표현하며, 오퍼레이션 호출은 다이아그램에 나타나지 않는다. 두 가지 형태의 오퍼레이션이 UML 클래스에 표현되는데, *selector*는 값을 리턴하고 *modifier*는 객체 상태를 수정한다. *selector*는 UML 클래스 다이아그램에 function으로 나타나고 (즉, 리턴되는 값의 타입을 보여준다) 다음과 같은 형식을 갖는다.

<visibility> <service name>(<in-parameters>) ; <return-type> {<optional property string>}

매개변수는 *in-parameter*이다. 즉 객체에 대한 정보만을 전달한다. 모든 function들은 객체 상태를 수정하지 않는다고 가정한다. modifier는 값을 리턴하지 않고 다음과 같은 형식을 갖는다.

<visibility> <service name>(<in-parameters>) {<optional property string>}

visibility는 public, protected 또는 private이며, property string은 optional이다. protected 또는 private 오퍼레이션은 객체의 내부 상태를 구성하는 ADT에 정의된 오퍼레이션으로 변환된다. 본 논문에서 service는 public 오퍼레이션을 의미한다.

메시지 전달 즉, 서비스 호출은 LOTOS에서 두 개의 프로세스가 동시에 이벤트를 발생시키는 것으로 모델링된다. 서비스를 호출하는 것은 asymmetric하며, 클라이언트는 서버가 제공하는 서비스를 호출한다. LOTOS 프로세스간에 동시에 발생하는 이벤트는 symmetric하며, LOTOS에서는 클라이언트와 서버에 대한 구분이 없다. UML 클래스에서 제공하는 서비스 set은 LOTOS choice 표현식에서 선택적인 구조적 이벤트 set를 제공하는 서버 프로세스로 모델링된다. 본 논문의 규칙에서, 구조화된 이벤트의 첫 번째 값은 서비스명을 나타내고, 두 번째 값은 서버 객체의 객체 식별자를 나타낸다. 그리고 세 번째는 optional 매개변수 목록을 표현한다.

<gate> !<service name> !<server object-id> <optional parameters>

객체를 정의하는 LOTOS 프로세스는 객체 상태를 정의하는 ADT를 갖는다. LOTOS 프로세스에 정의된 각 서비스는 각 상태 ADT에 관련된 오퍼레이션 정의를 갖는데, 어떻게 필요한 정보를 객체 상태로부터 얻는지 또는 어떻게 객체 상태가 수정되는가를 보여준다. LOTOS ADT의 각 오퍼레이션은 한 개 이상의 algebraic equation에 의해 정의된다. LOTOS에서 ADT 정의는 모든 상황을 기술해야 하고 equation이 각 생성자에게 주어져야만 하기 때문에 매우 길어질 수가 있다. 그러나 UML 다이아그램으로부터 필요한 모든 정보를 가져올 수 없으므로, *symbolic* ADT를 정의함으로써 이 문제를 해결한다. *symbolic* ADT는 명세를 프로토타입화 하는데 필요한 상태 정보와 통신하는 동안 전달되는 값에 대한 정보만 포함한다. selector 오퍼레이션은 ClassA를 위한 것으로 UML 다이아그램에서 다음과 같은 형태로 표현된다.

```
op(p: B): C
```

UML 클래스 다이아그램에서 오퍼레이션은 암시적인 매개변수로 객체의 상태를 나타낸다. LOTOS ADT의 정의에서, 이 매개변수는 명확하게 표현되어야 한다. 위의 selector 오퍼레이션은 소트 ClassA_State를 위해 ADT에서 다음과 같이 정의한다.

```
op:B, ClassA_State -> C
```

equation 정의는 다음과 같이 정의한다.

```
forall classA_State: ClassA_State, p: B
ofsort C
   op(p, classA_State) = the_C
```

즉, symbolic 값만 리턴 된다. 클래스 ClassA의 LOTOS 프로세스 정의는 매개변수 id와 객체 식별자를 표현하는 s 그리고 각각의 상태를 갖는다. 만약 op가 atomic이면, 오퍼레이션은 LOTOS에서 다음과 같은 행위 표현식으로 정의된다.

```
g !op !id ?p: B !op(p, s);
exit(s)
```

즉, 리턴 되는 값은 매개변수 p와 s를 가지고 ADT 오퍼레이션 op를 호출하는 것으로 정의된다. 만약 op가 non-atomic이면, 클래스 ClassA를 위한 LOTOS 프로세스는 다음과 같은 오퍼레이션으로 정의된다.

```
g !op !id ?p: B;
(* internal processing *)
g !rtn_op !id !op(the_B, s);
exit(s)
```

내부 처리 결과로서 소트 B의 수정된 값을 얻을 수 있다. modifier 오퍼레이션은 ClassA를 위한 것으로 UML 클래스 다이아그램에서 다음과 같이 나타낸다.

```
op(p: B)
```

UML 클래스 다이아그램에서 오퍼레이션은 암시적 매개변수로서 객체 상태를 갖고, 오퍼레이션의 결과는 객체 상태를 갱신한다. LOTOS ADT 정의에서는 이들 두 가지 경우가 명확하게 나타난다. 이 오퍼레이션은 equation 정의를 갖지 않는 생성자로 정의된다. 오퍼레이션인 생성자로서 정의된다. 만약 op가 atomic이면, ClassA를 위한 LOTOS 프로세스 정의는 다음과 같이 오퍼레이션을 정의한다.

```
g !op !id ?p: B;
exit(op(p, s))
```

즉, 상태는 매개변수 p와 s를 가지고 ADT 오퍼레이션을 호출함으로써 갱신된다. 만약 op가 non-atomic이면, 클래스 ClassA를 위한 LOTOS 프로세스 정의는 다음과 같이 오퍼레이션을 정의한다.

```
g !op !id ?p: B;
(* internal processing *)
g !rtn_op !id;
exit(op(the_B, s))
```

3.4 객체

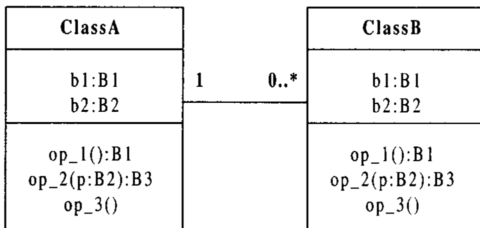
UML 클래스 다이아그램은 클래스뿐만 아니라 객체도 포함할 수 있다. 객체는 클래스의 인스턴스이며 독자성과 속성 값을 가지고 있다. 객체는 UML 정적 구조 다이아그램에서 두 구역으로 이루어진 직사각형으로 표현된다. 객체명은 다음과 같이 표현된다.

<object> : <full class name>

3.5 이진 관계

ClassA의 객체와 ClassB의 하나 이상의 객체간에 관계가 존재할 때, 이 관계는 다중성에 따라, LOTOS 명세에서 ClassB의 객체 식별자 또는 객체 식별자 집합을 표현하는 ClassA의 속성으로 표현된다. 관계를

나타내는 속성은 클래스 상태를 정의하는 ADT로 통합된다. (그림 3)은 ClassA와 ClassB간에 관계를 보여주고 있다. ClassA 객체가 zero 또는 하나 이상의 ClassB 객체와 연결되어 있는 반면 각 ClassB 객체는 하나의 ClassA 객체와 연결되어 있다.



(그림 3) 관계

ClassA_State를 위한 ADT는 타입 ClassB_Set_Type을 import 하도록 확장되어 ClassB 객체 식별자를 포함할 수 있다. ClassB는 다수의 인스턴스가 있으므로, ADT ClassB_Set_Type이 이미 존재해야 한다. ADT 정의에서 오퍼레이션 get_ClassB_Id는 각 객체 식별자로 정의한다. ClassB_State를 위한 ADT는 ClassA가 다수의 인스턴스를 갖는지 또는 아닌지에 따라 ClassA_Id_Type 또는 ClassA_Set_Type을 import한다. ClassB와 각 객체가 연결된 ClassA 객체의 식별자를 갖을 수 있도록 ClassB_State는 소트 Class_Id의 컴포넌트를 포함한다.

(그림 3)의 관계에서, ClassA의 객체들은 ClassB 객체들의 set을 알고 있어야 한다. 만약 ClassA 객체가 새로운 ClassB 객체를 개체화시키면, ClassA 객체는 ClassB 객체 참조를 이 클래스 객체 set에 추가해야 한다. ClassA_State내에 있는 ClassB 객체 식별자의 집합이 수정될 수 있도록, ClassA_State 소트의 LOTOS 정의는 add_ClassB_Id 오퍼레이션을 포함해야 한다. ClassA의 인스턴스가 단지 하나만 있을 때, 두 개의 상태 ADT는 다음과 같은 LOTOS 명세로 기술된다.

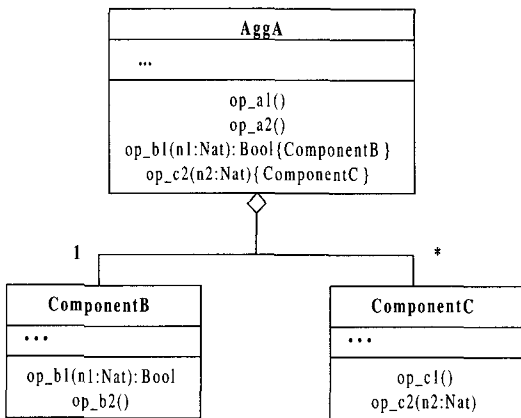
```
type ClassA_State_Type is B1_Type, B2_Type, B3_Type, ClassB_Set_Type
   sorts ClassA_State
   opns
      the_ClassA_State: -> ClassA_State
      make_ClassA_State: B1, B2, ClassB_Set -> ClassA_State
      get_b1: ClassA_State -> B1
      get_b2: ClassA_State -> B2
      get_ClassB_Id: ClassA_State -> ClassB_Id
      add_ClassB_Id: ClassB_Id, ClassA_State -> ClassA_State
      op_1: ClassA_State -> B1
      op_2: B2, ClassA_State -> B3
      op_3: ClassA_State -> ClassA_State
   eqns
      forall classA_State: ClassA_State
      ofsort B1
         get_b1(classA_State) = the_B1;
      ofsort B2
         get_b2(classA_State) = the_B2;
      ofsort ClassB_Id
         get_ClassB_Id(classA_State) = the_ClassB_Id;
      ofsort B1
         op_1(classA_State) = the_B1;
      ofsort B3
      forall p: B2
         op_2(p, classA_State) = the_B3;
endtype (* ClassA_State_Type *)

type ClassB_State_Type is B1_Type, B2_Type, B3_Type, ClassA_Id_Type
   sorts ClassB_State
   opns
      the_ClassB_State: -> ClassB_State
      make_ClassB_State: B1, B2, ClassA_Id -> ClassB_State
      get_b1: ClassB_State -> B1
      get_b2: ClassB_State -> B2
      get_ClassA_Id: ClassB_State -> ClassA_Id
      op_1: ClassB_State -> B1
      op_2: B2, ClassB_State -> B3
      op_3: ClassB_State -> ClassB_State
   eqns
      forall ClassB_State: ClassB_State
      ofsort B1
         get_b1(classB_State) = the_B1;
      ofsort B2
         get_b2(classB_State) = the_B2;
      ofsort ClassA_Id
         get_ClassA_Id(classB_State) = the_ClassA_Id;
      ofsort B1
         op_1(classB_State) = the_B1;
      ofsort B3
      forall p: B2
         op_2(p, classB_State) = the_B3;
endtype (* ClassA_State_Type *)
```

3.6 집단화(Aggregation)

aggregation은 aggregate인 클래스에 연결된 다이아몬드로 나타낸다. aggregation은 객체간의 관계로 복잡한 객체인 aggregate를 좀더 단순한 객체들의 조합인 컴포넌트들로 표현한다. aggregate와 컴포넌트는 객체로 간주되며, 그들은 독자성을 가지며, 서비스를 제공하고 속성을 갖는다. 여러 종류의 aggregate가 있는데 [2, 3], 본 논문에서는 컴포넌트가 다른 객체에 의해 공유되지 않는 경우만 고려한다. 컴포넌트는 aggregate와 part_of 관계에 있다. 컴포넌트가 컴포넌트를 포함

할 수 있으나, 다른 클래스들과 또 다른 관계를 가질 수는 없다. aggregate는 숨겨진 컴포넌트의 범위를 정의하고 그들 각각을 캡슐화한다. 컴포넌트의 서비스는 자신의 aggregate에게만 제공된다. 시스템상의 다른 객체들은 단지 aggregate와 통신하고 다른 컴포넌트의 존재는 알지 못한다. aggregate의 다중성은 언제나 1이며, 컴포넌트의 다중성은 일반적인 규칙을 따른다. aggregate를 위한 오퍼레이션을 수행하기 위하여, aggregation의 컴포넌트가 제공하는 하나 이상의 오퍼레이션을 호출해야 한다. 하나의 컴포넌트에 하나의 오퍼레이션이 필요한 경우, 같은 시그니처를 갖는 오퍼레이션이 컴포넌트와 aggregate에 의해 제공되도록 하며, UML 다이아그램의 오퍼레이션 특성 문자열은 컴포넌트 클래스명을 포함하도록 한다. (그림 4)는 aggregate에서 서비스 op_b1과 op_c2를 만족시키기 위하여 ComponentB가 서비스 op_b1을 제공하고, ComponentC가 서비스 op_c2를 제공하는 상황을 보여준다.



(그림 4) 집 단 화

aggregate는 ComponentB의 인스턴스 하나와 ComponentC의 zero 또는 하나 이상의 인스턴스를 포함한다. 새로운 ComponentC 객체가 인스턴스화 될 때, ComponentC 객체 식별자 set이 갱신될 수 있도록 ADT AggA_State_Type은 오퍼레이션 add_ComponentC_Id를 포함한다. 전체 aggregate는 최상위 프로세스로 변환이 되는데, 이 프로세스는 aggregate와 같은 이름을 갖는 제어 프로세스와 함께 컴포넌트의 프로세스 인스턴스화를 캡슐화 한다. aggregate의 몸체는 캡슐화된 프로세스를 LOTOS 병렬 연산자를 사용하여 나타낸 LOTOS 행위 표현식으로 변환된다.

숨겨진 게이트 g_ComponentB와 g_ComponentC는 제어 프로세스와 ComponentB와 ComponentC간의 상호작용을 위해 정의된다. 제어 프로세스의 상태는 컴포넌트의 객체 식별자를 포함하는데, 즉 aggregation 관계는 컴포넌트에 대한 항해성을 갖는 관계로 번역된다. aggregation의 상태는 각 컴포넌트로 분산된다. 전체 프로세스 캡슐화는 제어 프로세스와 같은 소트의 상태를 갖는데 제어 프로세스의 상태를 단지 초기화하기 위하여 사용된다. (그림 4)의 aggregation 관계로부터 생성된 LOTOS 명세는 다음과 같다.

```
process AggA[g](id: AggA_Id, s: AggA_State) : noexit :=
   hide g_ComponentB, g_ComponentC in
   AggA[g, g_ComponentB, g_ComponentC](id, s)
   | [g_ComponentB, g_ComponentC] |
   (      ComponentB[g_ComponentB](the_ComponentB_Id,
   the_ComponentB_State)
    |||
    ComponentCs[g_ComponentC] ({} of ComponentC_Set)
  )
where

process AggA[g, g_ComponentB, g_ComponentC](id: AggA_Id,
s: AggA_State): noexit :=
{  g !op_a1 !id;
   exit(op_a1(s))
 []
  g !op_a2 !id;
  exit(op_a2(s))
 []
  g !op_b1 !id ?n1: Nat;
  g_ComponentB !op_b1 ?idc: ComponentB_Id !n1 ?rtn_val: Bool;
  g !rtn_op_b1 !id !rtn_val;
  exit(s)
 []
  g !op_c2 !id ?n2: Nat;
  g_ComponentC !op_c2 ?idc: ComponentC_Id !n2;
  g !rtn_op_c2 !id;
  exit(op_c2(n2, s))
) >> accept new_s: AggA_State
      in AggA[g, g_ComponentB, g_ComponentC](id, new_s)
endproc (* of inner AggA *)
endproc (* AggA *)

process ComponentB[g](id: ComponentB_Id, s: ComponentB_State):
 noexit :=
(  g !op_b1 !id ?n1: Nat !op_b1(n1, s);
   exit(s)
 []
  g !op_b2 !id;
  exit(op_b2(s))
) >> accept new_s: ComponentB_State
      in ComponentB[g](id, new_s)
endproc (* ComponentB *)

process ComponentCs[g](ids: ComponentC_Set): noexit :=
  g !create ?id: ComponentC_Id ?s: ComponentC_State [id
  notin ids];
```

```
  ( ComponentC[g](id, s)
    |||
    ComponentCs{g](Insert(id, ids))
  )
endproc (* ComponentCs *)

process ComponentC[g](id: ComponentC_Id, s: ComponentC_
 State): noexit :=
(  g !op_c2 !id ?n2: Nat;
   exit(op_c2(n2, s))
 []
  g !op_c1 !id;
  exit(op_c1(s))
) >> accept new_s: ComponentC_State
      in ComponentC[g](id, new_s)
endproc (* ComponentC *)
```

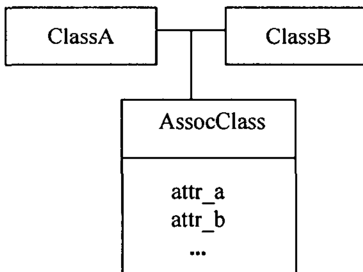
3.7 다중성(Multiplicity)

일반적으로 다중성의 형태는 bound가 자연수인 lower-bound와 upper-bound이다. 만약 다중성이 고정된다면, 하나의 자연수로 표현될 수 있다. 무제한 upper-bound는 *로 표현되는데 이것은 0..*와 동일하다. 다중성 1은 LOTOS 명세에서 다른 객체의 식별자를 표현하는 객체내의 속성으로 변환된다. 다른 모든 다중성은 한 객체에서 다른 객체의 식별자의 집합을 표현하는 속성을 정의하는 것으로 표현한다.

3.8 관계 클래스(Association Class)

관계는 클래스와 같은 속성을 갖을 수 있다. 관계 클레스는 관계와 관련된 속성을 갖는다. 관계 클래스는 LOTOS에서 독립된 객체 식별자를 갖지 않는다는 점만 제외하고 일반적인 클래스와 같은 방법으로 표현된다. 대신 그것은 연결된 두 클래스의 객체 식별자로 구성된 합성 객체 식별자를 갖는다. 다음은 (그림 5)의 관계 클래스로부터 변환된 LOTOS 프로세스 정의 명세를 보여준다.

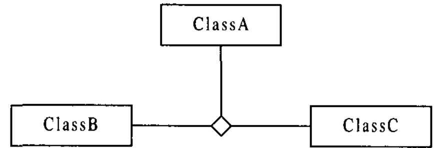
```
process AssocClass[g, h] (ida: ClassA_Id, idb: ClassB_Id,
 s: AssocClass_State): noexit :=
...
endproc (* AssocClass *)
```



(그림 5) 관계 클래스

3.9 N_ary 관계

n_ary 관계는 3개 이상의 클래스간의 관계이다. LOTOS에서 관계를 구성하는 클래스 인스턴스의 객체 식별자인 속성을 갖는 클래스로서 표현된다.



(그림 6) N_ary 관계

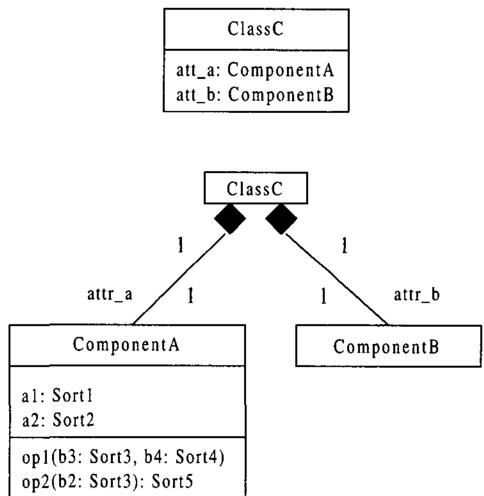
(그림 6)의 n_ary 관계에 대응하는 LOTOS 프로세스의 골격은 다음과 같다.

```
process Nary[g, h](ida: ClassA_Id, idb: ClassB_Id,
                   idc: ClassC_Id): noexit :=
 ...
endproc (* Nary *)
```

n_ary 관계는 어떤 데이터도 포함하지 않으며, 단지 관계를 구성하고 있는 객체 식별자만을 가지고 있다.

3.10 합성(Composition)

UML에서 합성은 aggregation의 강한 형태이다. 합성은 색이 채워진 다이아몬드 심벌로 표현된다. 클래스의 속성은 UML 클래스 다이아그램에서 합성 관계로 표현될 수 도 있다. (그림 7)은 같은 의미를 지니는



(그림 7) 속성과 합성

ClassC에 대한 두 가지 정의를 나타낸다. 첫 번째 그림은 소트 ComponentA와 ComponentB가 속성으로 표현된 반면, 두 번째 그림은 컴포넌트로 표현된다. 합성 관계에 있어서 다중성은 속성의 다중성을 1로 제한하는 것처럼 1 : 1로 제한한다.

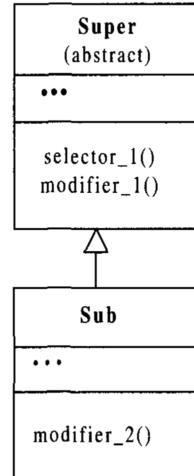
(그림 7)에서 구조화된 타입의 컴포넌트는 ClassC의 속성으로 표현된다. ComponentA가 제공하는 오퍼레이션은 ClassC안에서 속성을 조작하기 위한 것이다. 그러므로 ClassC의 클라이언트들은 사용할 수 없다. ClassC의 LOTOS 표현은 ComponentA와 ComponentB가 클래스 다이아그램에서 속성으로 정의되든 또는 합성된 컴포넌트로 정의되든 영향을 받지 않는다. 두 가지 경우에서, ClassC의 상태를 정의하는 ADT는 타입 ComponentA_Type과 ComponentB_Type을 import 한다. 구조적 타입 ComponentA에 대응되는 LOTOS ADT 정의는 다음과 같다.

```
type ComponentA_Type is Sort1_Type, Sort2_Type, Sort3_Type,
                                    Sort4_Type, Sort5_Type
   sorts ComponentA
   opns
     make_ComponentA: Sort1, Sort2 -> ComponentA
     the_ComponentA: ComponentA-> ComponentA
     get_a1: ComponentA -> Sort1
     get_a2: ComponentA -> Sort2
     op1: Sort3, Sort4, ComponentA -> ComponentA
     op2: Sort3, ComponentA -> Sort5
   eqns
     forall componentA: ComponentA
     ofsort Sort1
       get_a1(componentA) = the_Sort1;
     ofsort Sort2
       get_a2(componentA) = the_Sort2;
     ofsort Sort5
     forall b2: Sort3
       op2(b2, componentA) = the_Sort5;
endtype (* ComponentA_Type *)
```

따라서 구조화된 속성을 정의하는 ADT는 타입이 ComponentA_State_Type이 아닌 ComponentA_Type이라는 점만을 제외하면 객체 상태를 정의하는 ADT와 같은 형태를 갖는다.

3.11 일반화(Generalization)

LOTOS는 객체 기반 접근에 매우 적절하다. 그러나 완전한 객체지향 접근 방법을 제공하기 위하여 일반화(상속과 다형성)를 모델링할 수 있어야 한다.



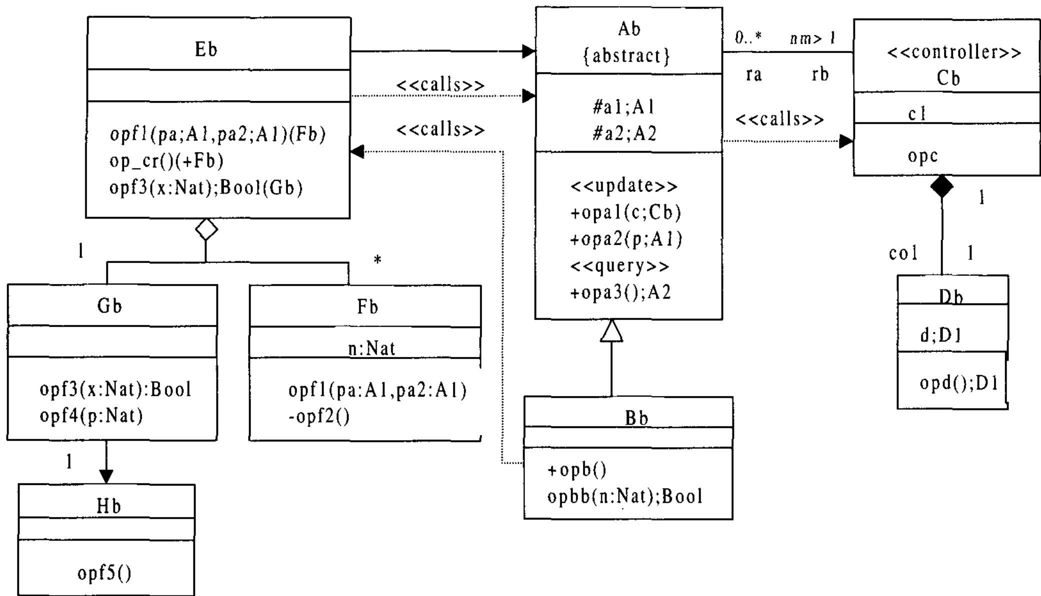
(그림 8) 수퍼클래스와 서브 클래스

대부분의 객체지향 언어에서, subtyping은 구현 상속의 제한된 형태로 구현된다. 이러한 제한사항들은 제공되는 서비스의 시그니쳐로 나타낸다. subtype의 객체가 supertype 객체의 모든 특성을 상속받기를 원하며, 또한 행위 상속이 복잡한 소프트웨어의 결과를 추론할 수 있도록 요구한다[5]. (그림 8)에서 보는바와 같이 수퍼클래스 Super로부터 상속을 받아 확장한 클래스 sub가 있다고 가정한다. 클래스 Super의 특성들은 [4]에 서술된 데로 LOTOS 프로세스 Super로 다음과 같이 정의될 수 있다.

```
process Super[g](id: Super_Id, s: Super_State) : exit(Super_State) :=
    g !selector_1 !id !selector_1(s);
    exit(s)
   []
    g !modifier_1 !id;
    exit(modifier_1(s))
endproc (* Super *)
```

4. 적용 사례

위에서 정의한 변환 규칙에 근거하여 UML로부터 LOTOS를 생성하기 위해서는 External File Format인 중간 코드로 변환하여 최종적으로 LOTOS 명세를 생성한다. (그림 9)의 클래스 다이아그램은 4.1절의 External File Format으로 변환되고, 이 중간 파일로부터 4.2절의 LOOTOS 명세를 생성한다. 본 논문에서는 전체 다이아그램중 Aggregation 관계에 있는 클래스 Eb, 클래스 Gb, 클래스 Fb에 대한 사례를 보여준다.



(그림 9) 클래스 다이아그램 예

4.1 External File Format 예

클래스 다이아그램의 Aggregation 관계에는 클래스 Eb, 클래스 Gb, 클래스 Fb에서 클래스 Eb와 클래스 Fb로부터 생성된 중간 파일의 예만 보여준다. 이 파일의 신텍스는 LL(1) 문법으로 정의되었으며, 다음과 같이 텍스트로 표현이 된다.

```
package System
class Eb
multi -
attribute
association
   Ab role (1) (*) navigable +; aggregate -; composite -,
   Fb role (1) (*) navigable +; aggregate >=; composite -,
   Gb role (1) (1) navigable +; aggregate >=; composite -,
operation
   opf1(pa: A1, pa2: A1) {Fb},
   op_cr() {*Fb},
   opf3(x: Nat): Bool {Gb}
superclass
call
   Ab, Cb
calledby
   Bb

class Ab {abstract}
...
class <<controller>> Cb
...

class Fb
multi +
attribute
   n: Nat
association
   Eb role (*) (1) navigable -; aggregate <=; composite -
operation
   opf1(pa: A1, pa2: A1),
   -opf2()
superclass
call
calledby

class Gb
...
calledby

...
```

4.2 생성된 LOTOS

생성된 LOTOS 명세는 4.1절의 External File로부터 생성되었다. Aggregation 관계인 클래스 Eb와 클래스 Fb의 생성 예를 보여주고 있는데, 3.6절의 Aggregation 변환 규칙에 따라 생성되었다. 코드의 구성은 각 클래스의 자료 타입 선언부와 프로세스로 나누어져 있다. 클래스 Fb는 다수의 인스턴스를 포함할 수 있으므로, 객체 생성자(Object Generator) 프로세스인 Fbs가 생성되었다.

```
specification example: noexit
library
```

```
   Set, NaturalNumber, Boolean
endlib
type Id_Type is Boolean, NaturalNumber
   sorts Id
   opns id1, id2, id3, id4, id5, id6, id7, id8, id9, id10
         id11, id12, id13, id14, id15, id16, id17, id18
         id19, id20: -> Id
         h: Id -> Nat
         _eq_, _ne_, _lt_, _gt_: Id, Id -> Bool
   eqns forall n1, n2: Id
         ofsort Nat
         h(id1) = 0;
         h(id2) = succ(h(id1));
         ...
         h(id20) = succ(h(id19));
         ofsort Bool
         n1 eq n2 = h(n1) eq h(n2);
         n1 ne n2 = h(n1) ne h(n2);
         n1 lt  n2 = h(n1) lt h(n2);
         n1 gt n2 = h(n1) gt h(n2);
endtype

type Eb_Id_Type is
   sorts Eb_Id
   opns the_Eb_Id: -> Eb_Id
endtype (* Eb_Id_Type *)

type  Eb_State_Type  is  Ab_Set_Type,  Fb_Set_Type,
 Gb_Id_Type, A1_Type, NaturalNumber, Bool
   sort Eb_State
   opns
      the_Eb_State: -> Eb_State
      make_Eb_State: Ab_Set, Fb_Set, Gb_Id ->Eb_State
      get_Ab_Id: Eb_State -> Ab_Id
      add_Ab_Id: Ab_Id, Eb_State -> Eb_State
      get_Fb_Id: Eb_State -> Fb_Id
      add_Fb_Id: Fb_Id, Eb_State -> Eb_State
      get_Gb_Id: Eb_State -> Gb_Id
      opf1: A1. A1, Eb_State -> Eb_State
      op_cr: Eb_State -> Eb_State
      opf3: Nat, Eb_State -> Bool
   eqns
      forall eb_State: Eb_State
      ofsort Ab_Id
         get_Ab_Id(eb_State) = the_Ab_Id;
      ofsort Fb_Id
         get_Fb_Id(eb_State) = the_Fb_Id;
      ofsort Gb_Id
         get_Gb_Id(eb_State) = the_Gb_Id;
      ofsort Bool
      forall x: Nat
         opf3(x, eb) = the_Bool;
endtype (* Eb_State_Type *)

type Fb_Set_Type is Set_Id_Type
   renamedby
   sortnames
      Fb_Id for Id
      Fb_Set for Set
   opnnames the_Fb_Id for id20
```

```
endtype (* Fb_Set_Type *)

type Fb_State_Type is NaturalNumber, A1_Type
   sorts Fb_State
   opns
      the_Fb_State: -> Fb_State
      make_Fb_State: Nat -> Fb_State
      get_n: Fb_State -> Nat
      opf1: A1, A1, Fb_State -> Fb_State
      opf2: Fb_State -> Fb_State
   eqns
      forall fb_State: Fb_State: Fb_State
      ofsort Nat
         get_n(Fb_State: Fb_State
         ofsort Nat
            get_n(fb_State) = the_Nat;
endtype (* Fb_State_Type *)

behaviour (* behaviour expression *)
where (* package system *)
(* process Eb is an aggregate *)
process Eb[g, g_Ab, g_Cb](id: Eb_Id, s: Eb_State): noexit :=
   hide g_Fb, g_Gb in
   Eb[g, g_Fb, g_Gb, g_Ab, g_Cb](id, s)
   |[g_Fb, g_Gb]|
   (Fbs[g_Fb]({} of Fb_Set)
    |||
    Gb[g_Gb](the_Gb_Id, the_Gb_State)
   )
where
(* Definition of inner control process *)
process  Eb[g,  g_Fb,  g_Gb,  g_Ab,  g_Cb](id:  Eb_Id,
          s:Eb_State): noexit :=
(    g !opf1 !id ?pa: A1 ?pa2: A1;
     g_Fb !opf1 ? idc: Fb_Id !pa !pa2;
     g !rtn_opf1 !id;
     exit(opf1(pa, pa2, s))
   []
     g !op_cr !id;
     g_Fb !create ?idc: Fb_id !the_Fb_State;
     exit(op_cr(add_Fb_Id(idc, s))
   []
     g !opf3 !id ?x; Nat
     g_Gb !opf3 ?idc: Gb_id !x ?rtn_val: Bool;
     g !rtn_opf3 !id !rtn_val;
     exit(s)
) >> accept new_s: Eb_State
   in Eb[g, g_Fb, g_Gb, g_Ab, g_Cb](id, new_s)
endproc (* of inner Eb *)
endproc (* Eb *)

...

(* process Fbs is an object generator *)
process Fbs[g](ids: Fb_Set): noexit :=
     g !create ?id: Fb_id ?s: Fb_State [id notin ids];
     (    Fb[g](id, s)
          |||
          Fbs[g](Insert(id, ids))
endproc (* Fbs *)


process Fb[g](id: Fb_Id, s: Fb_State): noexit :=
```

```
(   g !opf1 !id ?pa: A1 ?pa2: A1;
    exit(opf1(pa, pa2, s))
) >> accept new_s: Fb_State
        in Fb[g](id, new_s)
endproc (* Fb *)
```

5. 결 론

UML과 같은 표기법은 가시적이고 사용이 용이하나 분석설계가 비정형적이고 임기응변식으로 적용된다. 소프트웨어 개발에 정형기법의 적용은 많은 장점을 가지고 있다. 예를 들면, 정확하고 증명 자동화 가능한 표기법의 사용을 들 수 있다. 그러나 요구명세를 정형기법을 사용하여 작성하는 것은 상당히 힘든 일이다. 따라서 UML로 작성된 비정형 설계문서로부터 LOTOS와 같은 정형명세를 생성하고 이 명세를 기반으로 시스템의 정확성을 테스팅하고 검증할 수 있도록 함으로써 개발자들이 좀 더 쉽게 정형기법에 접근할 수 있도록 하고자 하는 것이다.

본 논문은 UML 정적 구조 다이아그램으로부터 LOTOS 명세로의 변환 방법을 정의하였다. UML 클래스 다이아그램의 각 클래스는 3개의 엔티티로 변환이 되는데, 즉 LOTOS 프로세스, 클래스 인스턴스의 상태를 정의 하는 LOTOS ADT와 객체 식별자를 정의하는 LOTOS ADT이다. 본 논문에서는 클래스 다이아그램과 이로부터 생성된 LOTOS 명세에 대한 예를 제시하였다. 또한 UML 모델로부터 완벽한 LOTOS 객체지향 명세를 생성하기 위해서는 UML의 동적 행위 모델을 고려해야 한다. 즉, Statechart와 Collaboration Diagram이 행위 정보를 제공하는데 사용될 수 있다.

객체지향 분석-설계 방법은 산업계에서 소프트웨어 개발에 실질적으로 사용되는 반면 정형방법은 주로 학계 및 연구계에서 연구를 목적으로 사용되고 있다. 두 방법의 통합은 정형 방법이 소프트웨어 개발자에 의해 실질적으로 사용되는 촉매 역할을 할 수 있을 것으로 기대된다.

참 고 문 헌

[1] S. Cook and J. Daniels, *Designing Object Systems,* Prentice-Hall, 1994.

[2] A.M.D. Moreira and R.G. Clark, *Complex Objects : Aggregates,* Stirling Computing Science and Mathematics Technical Report 123, May 1994.

[3] A.M.D. Moreira and R.G. Clark, *Formal Specification of Aggregates,* Sistemas de Informacao, paper accepted, 1997.

[4] A.M.D. Moreira and R.G. Clark, LOTOS in the Object-Oriented Analysis Process, Formal Methods and Object Technology, Goldsack, S and Kent, S(Editors), Chapter 3, pp.33-46, Springer-Verlag, 1996.

[5] T. Bar-David, Practical Consequence of Formal Definition of Inheritance, Journal of Object-Oriented Programming, 5 43-49, July/August 1992.

[6] M. Fowler, *UML Distilled,* Addison-Wesley, 1997.

[7] H. Teotenel, J.V. Katwijk, and N. Plat, Structured Analysis-formal design, using stream & object oriented formal specification. *ACM SIGSOFT Software Engineering Notes,* 15(4) : 118-127, Sep 1990.

[8] K. Kumar, M.D. Fraser and V.K. Vaishnavi. Strategies for incorporating formal specifications in software development. *Communication of the ACM,* 37(10):74-86, October 1994.

[9] Robert H. Bourdeau and Betty H. C. Cheng. A formal semantics for object model diagrams. *IEEE Transaction on Software Engineering,* 21(10) : 799-821, October 1995.

[10] A.M.D. Moreira. *Rigorous Object-Oriented Analysis,* PhD thesis, University of Striling, Stirling FK9 4LA UK, August 1994.

[11] Matthias Weber. Combining statecharts and Z for the design of safety-critical control systems. In M.C Gaudel and J. Woodrok, editors, *FME'96 ; Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Europe, Oxford, UK, March 1996 Proceedings,* volume 1051 of *Lecture Notes in Computer Science,* pp.307-326. Springer-Verlag, 1996.

[12] Kevin Lano and Howard Haughton. *Object-Oriented Specification Case Studies.* The Object-Oriented Series. Prentice-Hall, Inc. 1994.

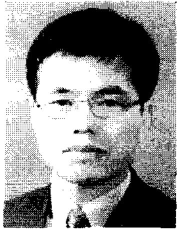
[13] Amarit Laorakpong and Motoshi Saeki. Object-oriented formal specification development using VDM. In Shojiro Nishio and Akinori Yonezawa, editors, *Object Technologies for Advanced Software,* volume 742 of *Lecture Notes in Computer*

Science, pp.529-543. Springer-Verlag, August 1993.

[14] 천윤식외 3인, "정형적 방법과 비정형적 방법의 통합, OOLOTOS와 OMT 객체모형", 한국정 보처리학회 학술발표논문집 제3권 2호, 1996 10.

[15] T.Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Network and ISDN System,* 14(1):25-59, 1987.

[16] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1 : Equation and Initial Semantics.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, New York, N.Y., 1985.



김 철 홍

e-mail : kch@etri.re.kr

1982년 충남대학교 문과대(학사)

1993년 성균관대학교 정보처리학과 (석사)

1983년~현재 한국전자통신연구원 컴퓨터-소프트웨어기술연구소 실시간컴퓨팅연구부 선임연구원

관심분야 : 소프트웨어 재사용, 소프트웨어 설계 방법론, 정형기법

안 유 환

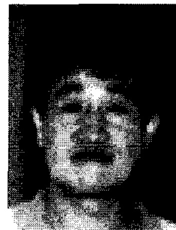
e-mail : ywahn@etri.re.kr

1984년 서울대학교 공과대학 산업공학과 졸업(학사)

1986년 한국과학기술원 경영과학과(석사)

1986년~현재 한국전자통신연구원 컴퓨터-소프트웨어기술연구소 실시간컴퓨팅연구부 S/W품질보증연구팀 팀장

관심분야 : 소프트웨어 품질보증 및 품질평가, 소프트웨어 프로세스 관리, 통합 방법론



이 원 천

e-mail : wclee@etri.re.kr

1981년 아주대학교 전자공학과 (학사)

1984년~현재 한국전자통신연구원 컴퓨터-소프트웨어기술연구소 실시간컴퓨팅연구부 연구원

관심분야 : 소프트웨어 시험 및 품질보증, 소프트웨어 재사용