정형기법에 의한 재사용 컴포넌트 및 인터페이스 명세 기술 연구

서 동 수†

요 약

소프트웨어 컴포넌트의 기능에 대한 정확한 서술은 컴포넌트의 활용을 위한 필수 조건이며 특히, 실시간 시스템과 같은 엄밀성을 요하는 분야에는 더욱 중요한 요소로 작용한다. 본 논문에서는 컴포넌트의 이해를 높이는 수단으로 패턴에 기반한 정형적 표현 및 검증에 관한 내용을 소개한다. 특히 본 논문은 컴포넌트 기능 서술 시 VDM++를 이용하는 명세 방법, 주어진 명세에 대한 정제와 적합성 검증에 관한 정형기법의 활용방법을 제시한다.

키워드 : 정형기법, 소프트웨어 컴포넌트, 설계 패턴

Description Techniques for Reusable Components and Interfaces using Formal Methods

Dongsu Seo†

ABSTRACT

Correct descriptions for software component functions become a strong requirement in developing critical software especially on the area of real-time applications. In this paper, we introducc both formalization of software design using patterns and verification methods in order for the components to increase their understandability. In particular, the paper investigates into a means of formal description techniques based on VDM++ for the software components, and provides adequacy proof steps for a given functional descriptions.

Key word : formal methods, software components, design patterns

1. 서 론

객체 기술의 확산과 분산 처리 기술의 발전과 더불어 주목 받는 컴포넌트 기반 소프트웨어 구축 기술은 기존의 프로그래밍 기술을 대체 혹은 보완하는 효율적인 기법으로 평가 받고 있다[1]. 이러한 일련의 변화에 대한 배경에는 비즈니스 환경 및 기술의 변화를 신속히 수용하고자 하는 기업의 요구와 새로운 기능을 손쉽게 확장하거나 재사용을 통한 소프트웨어의 생산성 향상을 기대하는 개발자들의 요구가 주 요소로 작용되고 있으며 실제 이러한 요구를 만족시키려 하는 시도가 COM, EJB, CORBA 등[2]의 여러 형태로 나타나고 있다. 소프트웨어 컴포넌트를 이용하여 응용 프로그램을 제작하거나 재사용 가능한 컴포넌트 자체를 구축하는 일련의 활동은 컴포넌트 기반 소프트웨어공학[2]이라는 분야로 통칭되어 다른 개발 기법과 구분되고 있다.

소프트웨어 컴포넌트 (이하 컴포넌트라 함)란 잘 알려진 기능을 수행하도록 구현한 단위 소프트웨어로서 구체적인 구현 내용은 사용자에게 숨기고 잘 정의된 인터페이스를 통해 해당 기능을 제공한다. 컴포넌트는 객체지향 특성을 많이 가지고 있으나 객체지향 언어로만 구현될 수 있는 것은 아니며 많은 경우 사용자로부터 구현 언어 혹은 작동 메커니즘을 감추는 정보은닉 메커니즘을 가지고 있다.

소프트웨어 재사용의 측면에서 컴포넌트는 프로그램 개발의 생산성 증대에 기여하는 것은 사실이지만 이러한 효과가 현실화 되기 위해서는 기술적인 제반 요소들, 예를 들면 충분히 많은 수의 사용 가능한 컴포넌트의 존재, 검색 및 등록 시스템의 지원, 효과적인 컴포넌트 서술 등, 의 문제가 해결 되어야 한다. 특히, 컴포넌트의 기능명세는 올바른 컴포넌트 선택을 위해 필수적으로 선행되어야 하는 사항이며 부적절한 기능의 컴포넌트를 선택하여 사용할 경우 발생하는 오류는 전체 시스템의 생산성과 신뢰성에 큰 악영향을 줄 수 있다. 이러한 문제가 발생한 이유로는 다음의 사항들을 들 수 있다.

첫째, 컴포넌트 기능의 명세 수단으로 자연어, 혹은 이에

† 정 회 원 : 성신여자대학교 컴퓨터정보학부 교수
논문접수 : 1999년 8월 9일, 심사완료 : 1999년 12월 1일

기반한 구조적 언어를 사용할 경우 사용하기 편리한 반면 명확한 의미 정의가 어려우며 따라서 명세에 대한 모호성, 불완전성을 원천적으로 제거하기 힘들다. 둘째, 컴포넌트의 서비스를 이용하려는 개발자는 인터페이스 정보만을 바탕으로 이용에 관한 결정을 해야 하며 이러한 결정 시기는 코딩 단계가 아닌 설계, 혹은 분석 단계에서 행해지는 경우가 종종 발생한다. 이 경우 사용에 관한 의사 결정 과정에서 컴포넌트에 대한 충분한 정보가 제공되지 못 할 때 이에 대한 실패의 부담은 활용자에게 전가된다. 셋째, 컴포넌트를 이용하여 개발된 시스템에 대한 정확성 검증이 필요한 경우 제 3자가 개발한 컴포넌트 부분에 대해 어떤 방식으로 정확성 증명 혹은 검증을 해야 하는지에 관한 아무런 정보도 제공해 주지 못한다는 점 역시 검증이 필요한 시스템 개발에 있어 부담 요소로 작용한다.

본 논문은 기존 방식의 컴포넌트 명세 방식이 실시간 소프트웨어, 고신뢰 시스템과 같이 엄밀성과 정확성이 필수로 요구되는 분야에 적용될 경우 혹은 개발 과정에 관한 검증을 필요로 하는 분야에 사용될 경우 발생되는 컴포넌트의 비정형성 문제를 다루고 있으며 이러한 문제를 완화 시키는 해결책을 논한다. 특히, 본 논문은 프로그램의 설계 시 VDM++를 컴포넌트의 기능에 관한 명세 수단으로 이용할 경우 컴포넌트에 관한 구조 및 기능 요소 증명에 관한 정보를 함께 제공하며 이를 이용하는 방식을 설명한다. 본 논문은 2절에서 컴포넌트의 명세 기법에 관한 관련 연구를 설명하고, 3절에서는 컴포넌트에 대한 기능 명세의 수단으로 패턴을 이용하는 컴포넌트 명세 기법을 제안하며 정형성을 표현하는 방법으로 확장된 VDM++를 사용하는 방식을 제안한다. 4 절에서는 기존의 정형기법에서 연구되었던 제 증명 방법을 컴포넌트를 위한 패턴 명세에 대해 어떻게 적용하는지에 관한 방법을 제시한다.

2. 관련 연구

컴포넌트에 관한 관심이 증폭되고 있는 속도와 비교할 때 컴포넌트에 대한 정형 처리 관련 연구는 극히 제한적으로 이루어져 왔다. 소프트웨어라는 광의의 시각에서 소프트웨어의 정형 명세에 관련 된 연구는 크게 Z, VDM 등의 상태기반 기법 [8,9], CSP, CCS와 같은 행위기반 기법 [10, 11], OBJ 3, ACT I 등의 대수적 기법 [12]을 생각해 볼 수 있다. [11]의 경우 주어진 두 개의 모형에 존재하는 행위 특성을 상호유사성(bisimulation) 기법을 사용하여 검증하는 방법을 제공하며 이들은 프로토콜 검증과 같은 동적행위특성을 설명하는 수단으로 유용하게 이용되고 있다. 그러나 컴포넌트가 사용자에게 보여지는 기능 정보는 주로 정적인 인터페이스라는 점을 감안할 때 시스템의 동적특성 파악하는 행위기반 모형은 서술 및 활용면에서 복잡해질 수 밖에 없다. 상태기반 기법은 이러한 측면에서 상대적으로 쉬운 명세 방법을 제공하며 컴포넌트 자체가 갖는 객체지향적 성질로 인해 이러한 개념을 구문 혹은 의미론 면에서 지원해 주는 Z++[13], OOZE[14], VDM++[15] 등이 좋은 후보로 고려될 수 있다. 이들은 객체를 정의하는 부분에 대해 많은 공통점이 있는 반면 객체의 메소드를 호출하는 방법으로 Lano, Haughton은 Z++를 위한 기능적 표기[13]를, Alencar, 와 Goguen은 OOZE를 위한 Z 스키마 방식의 호출을[14], 그리고 Durr는 VDM++는 메시지 패싱 방식[15]을 사용하는 차이점이 있기도 하다.

다른 관련 연구로는 컴포넌트를 이용하는 정형적 개발 절차에 관한 분야가 있다. 정형기법에 있어 구현의 의미는 추상적 기능 서술에 대해 구체화가 가능한 여러 대상 중 하나를 선정하는 과정이라 보며 이를 명세의 정제(refinement)라 부른다. 명세의 정제는 더 이상 구체화 될 수 없는 프로그램 코드 수준까지 진행될 경우 그 반복적인 활동을 계속할 수 있다. 정형 기법에서의 정제는 구조적 기법, 객체지향 기법 등의 분할(decomposition) 개념과 유사할 수 있으나 이들 설계 기법과 차이를 보이는 부분은 각 정제 단계는 엄밀한 증명을 요한다는 점이다. 정제 단계의 증명 기법으로는 Woodcock의 스키마에 기반한 정제 기법, Jones [9]는 선 조건과 후 조건의 연관 관계에 관한 관찰을 바탕으로 자료 정제와 오퍼레이션 정제를 분리하여 정제기법 등을 들 수 있다.

Lano와 Bicarregui, Goldsack[16]는 명세의 정제에 있어 해석표(interpretation table)를 이용하여 정제의 무결성을 증명한 바 있다. 이들의 연구에서 주목할 점은 기존의 설계 패턴에 대해 정형적 의미론을 부여했다는 점과 더불어 명세의 정제에 있어 해석표를 사용했다는 점이다. 이 해석표는 동일한 추상 명세로부터 구체화된 두개의 다른 명세에 대해 이들간에 존재하는 대응 관계를 매핑 테이블로 표현함으로서 정제되는 변수들의 관계를 쉽게 추적할 수 있도록 한다. 이러한 방식의 검증은 직관적인 대응 관계를 보여줄 수 있는 장점이 있지만 문제가 되는 것은 VDM, Z과 같은 모형 기반의 명세 방식에서는 검증을 위해 [16]에서와 같은 해석표 대신 검색함수(retrieval function)라 부르는 함수 형식의 정의를 필요로 하며 이때 해석표에 나타난 대응 관계는 일반적인 검색 함수로 항상 표현 가능한 것은 아니란 점이다. 이에 관한 자세한 설명은 4장에서 다루어지며 이며 대체 방법으로 해석표를 사용할 경우는 이에 대한 엄밀한 고려가 필요하다.

3. 컴포넌트 기능 표현을 위한 VDM++의 확장

컴포넌트 명세는 많은 경우 자연어를 이용한 서술 방식이 채택되었으나 컴포넌트의 특성에 충실히 표현하기 위해

구조화된 자연어를 패턴의 표현 단위로 사용하기도 하였다 [5, 6]. 이것은 자연어가 제공하는 사용의 편리성과 이해 용이성이 작용한 이유라 생각되나 동일 표현 수단으로 성격이 다른 두 요소들, 즉 기능 요소와 컴포넌트의 목적, 배경, 효과 등에 관한 비기능 요소를 모두 서술하고자 하는 시도에는 문제의 소지가 있다. 즉, 엄밀한 기능 이해를 필요로 하는 응용의 경우 개발자는 컴포넌트의 기능을 명확히 이해할 수 있어야만 새로운 응용에 적용이 가능한지를 판단할 수 있다. 이러한 관점에서 컴포넌트 서술에 VDM++를 도입한 [14]의 접근은 좋은 시도로 판단되나 VDM++ 구조를 그대로 컴포넌트에 채용했다는 점에서 기능 이해에 관한 명료성을 제공해 줄 수는 있지만 본 논문에서 의도한 '컴포넌트 기능의 검증'이라는 목표를 충족시키기에는 미흡한 것으로 판단된다. 본 논문에서는 이러한 개발 정보가 포함될 수 있는 방법을 고려하며 VDM++에 기반한 명세 언어의 확장을 살펴본다.

VDM을 객체지향적으로 확장시킨 VDM++는 클래스의 메소드와 속성 관계에 관해 두 가지 종류의 표기법을 제공한다. 먼저, 선언적 서술이라 불리는 방법으로 이것은 메소드가 호출되기 전 시점에서 입력 매개변수에 대해 참으로 확인 되어야 할 조건들을 표현하는 선 조건(pre-condition)과 입력 매개변수와 결과 매개변수 사이에 만족해야 할 조건들을 표현한 후 조건(post-condition) 만으로 메소드의 특성을 명세하는 방법이다. 다음으로, 절차적 서술기법으로 메소드를 구성하는 알고리즘 혹은 수행 절차를 절차적으로 표현하는 방식이다. 명시적 표기는 암시적 표기를 구체화시킨 다른 종류의 표현이라 볼 수 있다. 이러한 이유로 VDM++와 같은 기법을 컴포넌트에 이용하기 위해서는 명시적 명세가 암시적 명세의 조건을 만족 시키는지, 혹은 정제의 전후 단계가 올바른 연관성을 가지고 있는지에 관해 확인할 수 있는 수단이 함께 고려되어야 한다.

본 절에서는 하향식 설계 시 VDM++를 이용하여 컴포넌트의 기능을 서술하는 방식으로 다음의 구문 구조를 정의한다.

ComponentSpec :: AbsSpec : AbsVdm++Body
ConcSpec : ConcDef
ConcDef :: Refiner : RefVdm++Body
Theory : TheoryBody

먼저, *AbsSpec*의 VDM++ 클래스는 해당 컴포넌트의 기능 사항을 서술을 추상적으로 표현하며 이 부분에 관한 구체적인 서술은 *ConcDef* 부분에서 서술된다. *ConcDef*에서 표현되는 *Refiner*는 특정 *AbsSpec* 부분에 대해 구체화시킨 명세의 집합이다. *Theory* 부분에서는 *Refiner*의 내용이 Abs Spec에서 표현된 제약 사항을 그대로 보전하면서 이를 만족시키는 구체화된 서술이라는 것을 증명할 경우 필요한 각종 증명 규칙들을 제공한다.

*AbsSpec*의 세부 구성하는 *AbsVdm++Body*의 내용을 살펴보면 다음과 같다.

AbsVdm++Body ::
Type : TypeName → TypeDef
State : [ObjectState]
Method : MethodName → MethodDef
TypeDef :: Name : Type
Inv : {ExprDef}
Type = BasicType | CompositeType |
UnionType | SetType | SeqType |
MapType | FnType | TypeName

VDM++의 자료형 *Type* 정의 부분에서는 클래스 내의 자료 속성에 관한 자료 불변자(data invariant)를 정의한다. 자료 불변자는 자료를 생성하는 생성 함수인 'mk' 함수를 구현함에 있어 반드시 만족시켜야 하는 조건을 포함하는 진리 함수로서 *TypeDef*에서 정의된다. 자료 불변자의 이해를 돕기 위해 예로서 년, 월, 일과 같은 요소로 구성되는 *Date*를 생성하는 *mk-Day* 함수를 정의하는 과정에서 반드시 준수되어야 하는 제약 조건으로 일년의 총 일수를 표현하는 *day* 요소는 윤년 혹은 정상 일수를 확인하는 *inv-Date*로 정의한다면 다음과 같이 표현한다.

$$inv\text{-}Date(mk\text{-}Day(d, y)) := is\text{-}leapyear(y) \vee day \leq 365$$

두 번째로, 객체의 상태 명세 *State*는 초기 상태와 불변 상태에 관한 명세로 구성된다. 이를 표현하는 *initState*는 초기 상태를 생성할 시점에서 만족시켜야 할 조건을, 그리고 *invState*는 각 상태에 관한 접근 시 반드시 만족되어야 할 자료의 제약사항은 각각 담고있다. 이에 대한 구조 정의는 다음과 같다.

State :: objName : Name
tp : CompositeType
invState : [ExprDef]
initState : [ExprDef]

세 번째로, *methods*는 메소드 집합에 대한 VDM++ 서술을 담고 있으며 이를 위해서는 선언적 방식의 명세를 제공한다. 선언적 방식이 표현되는 *DeclMethodDef*에서는 입력 파라메터와 출력 파라메터에 관한 선조건과 후조건을 표현한다. 이에 대한 자세한 구조 정의는 다음과 같다.

Method = DeclMethodDef
*DeclMethodDef :: dom : TypePair**
rng : TypePair
pre : Expr
post : Expr

*ConcSpec*은 정의한 바와 같이 *Refiner*와 *Theory* 부분으로 구성된다. *Refiner*는 주로 *AbsSpec*의 *Method*부분을 구체화시킨 명세로서 절차적 방식의 알고리즘 형태로 함수를 정의한다. 이를 위해 *RefVdm++Body*는 대체적으로 *Abs Vdm++Body* 정의 내용과 유사하나 다음 부분에서 정의를

달리한다.

```
Method = ProcMethodDef
ProcMethodDef :: type : [MethodType]
                 params : ParamType*
                 boby : Expr
```

*Theory*는 컴포넌트 표현에서 중요한 부분으로서 증명규칙에 관한 정보를 제공한다. 컴포넌트를 이용하는 소프트웨어 개발 과정은 여러 단계로 구분되어 구성되며 각 단계는 이전 단계에서 생성된 명세에 대해 구체적인 표현을 고안하거나 선택하는 정제 작업을 행한다. *Theory*는 이전 단계의 명세 조건들이 올바르게 보전되고 있는지를 증명할 필요가 있으며 이를 위해 반드시 증명되어야 하는 증명 대상을 포함하며 이 내용은 *ConcDef*에서 정의된 *TheoryBody*에 표현한다. *TheoryBody*는 자료와 메소드 정제에 관한 증명 규칙을 다루고 있으며 이들은 각각 *DataReif*와 *MethodRefine*부분에 나타난다. 이들의 세부 설명은 4절에서 언급되며 구문에서 나타난 *Proof Rule*은 증명 규칙을 나타내는 진리 함수식이다.

```
TheoryBody :: DataReif : DataReifBody
              MethodReif : MethodReifBody
DataReifBody :: concrType : Type
                abstrType : Type
                retrFn : Expr
                adequacyRule : ProofRule
MethodRefineBody :: abstrMethods : methodName
                    concMethod : methodName
```

이상에서 설명한 명세 구조를 바탕으로 보일러의 안전 모니터링 시스템에 관한 명세에 관해 살펴본다. 이 시스템은 보일러 내의 각 센서에서 발생하는 신호를 감지하여 위험 신호에 대한 위치와 경고를 발생시키는 감시 시스템으로서 보일러 각 부분의 온도를 측정하는 최대 n개의 온도 센서와 압력을 측정하는 1개의 압력센서로 구성된다. 측정된 온도 혹은 압력이 위험 값을 넘어서면 히터의 전원을 차단한다. 만일 히터의 전원이 차단되지 않을 경우 해당 센서에 대한 경고 램프를 작동시키고 보일러 시스템에 대한 경보를 발생시킨다. 이러한 시스템에 대한 명세를 구성하는 요소로, *n*은 최대 센서의 수를, *Mode*는 *on, off* 상태를, *State*는 경보 시스템의 상태를 표현하기 위한 레코드 타입으로서 필드 요소로 *sensor, pressure, tempr, heater, alarm, alertPos*를 갖는다고 정의한다. *sensor*는 *n*개의 센서 중 경계치 이상의 값들이 들어오는 센서를 의미하며, *pressure*는 압력 센서를, *alarm*은 시스템이 경보 상태인지의 여부를, *alertPos*는 각 센서 번호와 위험 수준인지의 여부를 표시하는 플래그 쌍으로 구성되어 그 신호 값이 위험 수준에 있는지를 표시한다. 이 때 *alertPos*에 표시되는 신호는 반드시 센싱된 신호 중의 일부가 되어야 하며 보일러 내의 압력과 온도는 *MaxP, MaxH* 이하로 유지해야 한다. 이들 사항을 레코드 타입을 이용하여 VDM++로 명세하면 다음과 같다.

```
class BoilerMonitor
instance variables
  IdNum = {1,···,n}
  Mode = {ON, OFF}
  MaxP : nat
  MaxH : nat
  State :: sensor : IdNum-set
           pressure : nat
           heater : Mode
           alarm : Mode
           alertPos : IdNum-set → Mode
```

이때, nat는 자연수 타입을, *IdNum-set*는 *IdNum*으로 구성된 새로운 셋 타입임을 표현한다. *Boiler Monitor*의 자료 불변자로 처리해야 할 부분은 "*alertPos에 표시되는 신호는 반드시 센싱된 신호 중의 일부가 되어야 한다*" 부분이며 이는 다음과 같이 표현된다.

```
BoilerState = State
inv-BoilerState(s) : = dom(s.alertPos) ⊆ s.sensor ∧
                       rng(s.alertPos) ⊆ s.alarm
```

*inv-BoilerState(s)*의 정의에 나타난 추출 오퍼레이터 *dom*와 *rng*는 *(IdNum-set, Mode)* 형식의 맵으로 부터 정의역과 치역을 각각 분리하는 함수다.

*BoilerMonitor*가 *alarm* 상태에 진입하기 위해 필요한 조건으로는 먼저, 센싱된 신호 중 위험 수치 이상의 신호에 한해서만 해당 위치에 대한 *alert* 플래그를 할당한다. 그리고 이때 *alert* 플래그가 *on*상태이면 반드시 경보를 발생시킨다. 또한 *sensor, pressure, alarm, alertPos* 등의 값은 외부에서 강제적으로 변경하지 않는 한 스스로 값을 변경시키지 못한다. 이에 대한 내용을 *BoilerMonitor*의 메서드 *alert*을 선언적으로 서술하면 다음과 같다. 단, 이때 VDM++에서는 메서드 수행결과 변화된 변수의 값과 그 이전의 값을 구분하기 위해 이전 값을 갖는 변수는 기호 '←'(후크)를 사용하여 구분한다.

```
alert (s) : =
  ext wr s : BoilerState
  pre true
  post (s.alarm = ON ) ⇔
  ( ¬ (s.heater = OFF) ∧ (s.pressure ≥ MaxP ∨
    tempr ≥ MaxT ) ∨ (s.alarm = ON)) ∧
  s.sensor = ←s.sensor ∧ s.pressure = ←s.pressure ∧
  s.alertPos = ←s.alertPos ∧rng(s.alertPos) = ON
```

이상에서의 정형 명세가 자연어로 서술된 *Boiler Monitor* 명세의 의미를 충분히 반영하고 있는지의 확인이 필요하다. 이를 위해 *Theory* 부분의 구성을 고려하며 *Theory* 부분에서 나타나는 기본적인 확인 대상 중 하나는 선언적 명세에서 어떠한 후 조건의 내용이라도 모든 선 조건에 의해 암시될 수 있음을 보여주는 만족성(satisfiability) 확인에 관한 것이다. 다음과 같은 타입의 메소드 *m*에 대해

$$m\ (s_1 : State)\ s_2 : State$$

m의 선 조건, 후 조건에 대한 진리함수를 각각 *pre-m, post-m*이라 할 때 m에 관한 만족성 확인이란 다음과 같이 정의할 수 있다.

$$pre\text{-}m : State \rightarrow B$$
$$post\text{-}m : State \times State \rightarrow B$$
$$\forall s_1 \in State \bullet pre\text{-}m(s_1) \Rightarrow \exists\ s_2 \in State \bullet post\text{-}m(s_1, s_2)$$

BoilerMonitor 명세의 만족성 증명을 위해 먼저*pre-alert*와 *post-alert*를 정의할 필요가 있다. *pre-alert*은 가정에 의해 true로 주어지며 *post-alert*는 후 조건의 구조를 이용하는 다음과 같은 진리함수로 정의한다.

$$post\text{-}alert\ (s, \overleftarrow{s}) : = (s.alarm = ON) \Leftrightarrow$$
$$(\neg\ (s.heater = OFF) \wedge (s.pressure \geq MaxP \vee$$
$$tempr \geq MaxT\ ) \vee (\overleftarrow{s.alarm} = ON)) \wedge$$
$$s.sensor = \overleftarrow{s.sensor} \wedge s.pressure = \overleftarrow{s.pressure} \wedge$$
$$s.alertPos = \overleftarrow{s.alertPos} \wedge rng\ (s.alertPos) = ON$$

명세에 나타난 여러 주건들 중 "*sensor*의 값은 외부에서 강제적으로 변경하지 않는 한 스스로 값을 변경시키지 못한다"는 조건에 대한 만족성 확인 정리는 [정리 3.1]과 같이 표현된다.

[정리 3.1] $\forall s, \overleftarrow{s} \in BoilerState \bullet post\text{-}alert(s, \overleftarrow{s}) \Rightarrow$
$s.sensor = \overleftarrow{s.sensor}$

이에 대한 증명은 [정리 3.1]의 조건부 post-alert이 결론부를 논리곱으로 포함하고 있으므로 직접 증명이 된다. 다른 예로서 "보일러는 alarm 상태가 아닌 한 내부 압력과 온도를 *MaxP, MaxH* 이하로 유지해야 한다"는 사항에 대한 만족성 확인 정리는 [정리 3.2]와 같이 표현된다.

[정리 3.2] $\forall s, \overleftarrow{s} \in BoilerState \bullet \neg post\text{-}alert(s, \overleftarrow{s}) \wedge$
$(pressure < MaxP \wedge tempr < MaxT\ ) \Rightarrow$
$\neg\ (rng(s.alertPos) = ON)$

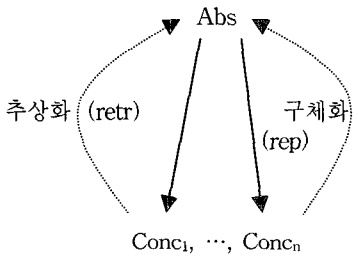
이에 대한 증명 역시 [정리 3.2] 조건 부의

$\neg$ post-alert(s,$\overleftarrow{s}$) $\wedge$ (pressure < MaxP $\wedge$ tempr <MaxT)를 정리한 논리식은 결론부 $\neg\ (rng(s.alertPos)$를 내포하고 있으므로 직접적으로 증명된다.

4. 컴포넌트 명세의 정제

컴포넌트 패턴은 특정 기능에 관한 명세로서 이를 근거로 새로운 명세를 구성할 수 있도록 하는 이론적 근거를 제공해 주어야 한다. 이것은 컴포넌트가 가지고 있는 정보를 정제하여 보다 구체적인 정보로 표현 가능하도록 할 수 있어야 하고 또한 정제 과정을 여러 번 반복하여 얻은 결과가 프로그램 코드에 직접 대응될 수 있을 정도로 구체적으로 표현될 때 믿을 수 있는 코드를 생성할 수 있다는 판단을 가능 하도록 하는 직접적인 이유가 된다.

패턴 명세의 정제 시 고려 사항은 크게 두 가지로 구분된다. 먼저 추상 명세에 포함되어 있는 내부 제약조건이 그대로 구체 명세에서도 역시 보전되고 있는가 하는 점이다. VDM++로 표현된 상태모형은 컴포넌트를 함수에 관한 입출력 매개 변수들의 변화를 위주로 서술한다는 특징이 있다. 이것은 추상 명세에 대해 구체적인 클래스 서술이 올바른 특성 반영이라는 점을 보일 때 효과적으로 쓰일 수 있는 표현 방법이다. 둘째, 컴포넌트와 구현 사이에 존재할 수 있는 다중 대응에 관한 관계 설정 문제이다. 개발자 입장에서 정형적으로 서술된 명세는 이를 바탕으로 생성된 구현에 대해 상당부분의 검증을 할 수 있는 기반을 마련해 준다 이것은 이미 Jones[6]에 의해 체계화된 부분으로 이 부분의 내용을 확장 설명하면 다음과 같다.



(그림 1) 구체화와 추상화의 관계

(그림 1)과 같이 컴포넌트의 추상 기능 Abs와 코드 수준의 구체성을 가진 명세 Conc의 관계를 살펴보기로 하자. 추상 기능 a Abs를 구체적으로 표현해 줄 수 있는 다수의 구체 표현 $i_1, i_2, i_3, \cdots \in$Conc이 존재한다고 할 때 이들 사이에는 함수로 표현될 수 있는 관계 설정이 가능하다. 이때 Abs로부터 Conc로의 함수가 존재할 경우 이를 표현 함수(representation function) *rep*라 부르고 이와 반대되는 역할을 하는 함수를 검색함수(retrieval function) *retr*라 부른다.

함수 *retr*를 정의 가능하다는 것은 i_1과 i_2가 동일한 컴포넌트 명세 *a Abs*로 사상 될 수 있음을 보여주는 것이고 이러한 함수의 존재를 증명할 때 그 정제는 '적합하다' 표현하며 이를 적합성(adequacy) 증명이라 한다.

$$rep : Abs \rightarrow Conc$$
$$retr : Conc \rightarrow Abs$$
$$\forall a \in Abs \bullet \exists i \in Conc \bullet retr(i) = a$$

이상에서 살펴본 내용들을 구체적으로 설명하기 위해 3절의 *BoilerMonitor* 명세에 대한 구체 명세로서 다음과 같은 *BMRefined*을 살펴본다. *BMRefined*는 상태 필드를 구현함에 있어 *Value* 리스트를 사용한다. 리스트는 셋과는 달

리 멤버들의 중복을 허용하나 순서를 중요시 여긴다.

```
class BMRefined
instance variables
  Value = {1, ···, max}
  Mode = {ON, OFF}
StateR :: sensor1 : IdNum*
          pressure1 : N
          heater1 : Mode
          alarm1 : Mode
          alertPos1 : IdNum* → Mode

BMRefinedState = StateR
inv-BMRefinedState((mk-StateR(s1,p1,h1,al1,aP1)) : =
∀i : N1 • (i ∈ inds dom(s.aP1[i]) ∧ i ∈ inds s.s1[i]) ⇒
(s.s1[i] ⇒ dom(s.aP1[i]) ∧ ((i ∈ inds rng(s.aP1[i]) ∧
i ∈ inds s.al1) ⇒ (s.al1 ⇒ rng(s.aP1[i])))
```

*inv-BMRefinedState*는 *BoilerState* 상태를 생성함에 있어 불변자 *inv*-BoilerState 나타났던 다음의 조건들을 표현하며 이때 명세에 나타난 함수 *inds*는 리스트 내의 원소 위치 혹은 인덱스를 반환하는 함수로 *inds s = {1, ···, len s}*으로 정의된다.

*BMRefined*의 상태를 갖는 메소드 *newAlart*의 정의를 살펴보면 다음과 같다.

```
newAlart() : =
  ext wr s : StateR
  pre true
  post ((s.alarm1 = ON) ⇔
  ( ¬ (s.heater1 = OFF) ∧ (s.pressure1 ≥ MaxP ∨
      s.tempr1 ≥ MaxT ) ∨ (s.alarm = ON)) ∧
      ∀ i : nat • i ∈ (inds s.sensor[i] = s.sensor[i]) ∧
      s.pressure = s.pressure ∧
∀i ∈ (inds s.alertPos1[i] ∩ inds s'.alertPos1[i]) ⇒
inds s.alertPos1[i] = inds s.sensor1[i]  ∨ s.sensor[i]
```

*BoilerMonitor*에 대해 *BMRefined*가 정제된 명세라는 점을 증명하기 위해서는 [정의 4.1]과 같은 검색함수 *retr-BoilerState : BMRefinedState → BoilerState*에 관한 정의가 필요하다.

[정의 4.1] *retr - BoilerState* 함수선언

```
retr-BoilerState (mk-BMRefiened(s1, p1, h1, al1, aP1)) : =
  let sensor   = s1[i],
      pressure = p1,
      heater   = h1,
      alarm    = al1,
      alertPos = aP1[i]
  in mk-BoilerState(sensor, pressure, heater, alarm, alertPos)
```

retr-BoilerState 함수에 관한 정의는 비교적 단순한 형태로 구성된다. 즉, 리스트 형태의 *BMRefined*가 집합 원소로 이루어진 *BoilerState*의 원소에 대응되도록 하는 전사함수의 형식을 갖는다. 이러한 *retr-BoilerState* 함수가 검색함수로서 정확한 것인가를 보여주기 위해서는 일련의 증명과정을 필요로 한다. 먼저 구체명세는 추상명세의 반영임을 증명하기 위해서는 이들 간의 관계를 표현할 수 있는 검색함수 *retr-BoilerState*이 구체명세에 나타나는 모든 상태들에 대해 정의 가능여야 한다. 두 번째로, 모든 $s_1 \in BoilerState$은 *retr-BoilerState* 함수에 그 값을 제공해 주는 어떤 원소 $s_2 \in BMRefined$가 존재함을 설명해주는 적합성이 증명되어야 한다.

[정리 4.1] (retr-BoilerState의 정의가능성)

$$sBMRefined \bullet retr\text{-}BoilerState(s) : BoilerState$$

[정리 4.2] (retr-BoilerState의 적합성)

$$\forall s_1 \in BoilerState \bullet \exists s_2 \in BMRefined \bullet s_1 = retr\text{-}BoilerState(s_2)$$

이상에서 언급된 사항 중 [정리 4.2]의 적합성 증명 과정은 다음과 같다.

[정리 4.2]의 증명 :

먼저 [정리 4.2]를 증명하기 위해 필요한 첫 단계는 증명에 쓰일 *BoilerState*와 *BMRefined*의 정의이다. 이 정의의 내용은 3절에 나타난 명세에 기반하며 [공리 4.1, 공리 4.2]로 정의된다.

[공리 4.1] *BoilerState*-formation :

{(a1 : IdNum-set), (a2 : nat), (a3 : Mode), (a4 : Mode), (a5 : IdNum-set→Mode)}

---

mk-BoilerState(a1,a2,a3,a4,a5) : State

[공리 4.2] *BMRefined* formation

{(a1 : IdNum*), (a2 : nat), (a3 : Mode), (a4 : Mode), (a5 : IdNum* → Mode)}

---

mk-BMRefined(a1, a2, a3, a4, a5) : BMRefinedState

단, 본 논문의 정리 및 공리의 표기는 전제부와 결론부를 수평 막대로 구분하여 표기한다.

*BoilerState*에 대한 검색함수 *retr-BoilerState* 함수는 이미 [정의 4.1]에 나타났으며 표현함수 *rep- RMRefined*의 정의는 다음 [정의 4.2]에 나타난다.

[정의 4.2] rep- BMRefined :

```
rep-BoilerState(mk-BoilerState(sensor,pressure,  heater,  alert,
alertPos) : =
  let   s1  = mk-List(sensor, n),
        p1  = pressure,
        h1  = heater,
        a1  = mk-List(alert, n),
        ap1 = mk-List(alertPos, n)
  in mk- BMRefined (s1, p1, h1, a1l, aP1)
```

단, mk-List(x, n) 함수는 최대 n의 인덱스를 갖는 배열 x를 생성 시키는 함수이다.

[공리 4.3] -Introduction

$$\frac{s \in X : E(s/x)}{\exists x \in X \bullet E(x)}$$

[보조정리 4.1] rep-BoilerState

$$\frac{\{(s : BoilerState)\}}{(rep\text{-}BoilerState[s]) : BMRefinedState}$$

(증명) [정의4.2]과 [공리 4.1]에 의한 직접 유도함

[보조정리 4.2] retr-BoilerState

$$\frac{\{(s : BMRefined)\}}{(retr\text{-}BoilerState[s]) : BoilerState}$$

(증명) [정의4.1]과 [공리 4.2]에 의한 직접 유도

[보조정리 4.3] rep/retr

$$\frac{\{\ \},\ \{(s : BoilerState)\}}{(retr\text{-}BoilerState(rep\text{-}BMRefined(s)) = s)}$$

(증명) [보조정리 4.1, 보조정]리 4.2]의 적용으로 BM Refined에 나타난 필드 중 첫번째 필드 sensor1에 대해 (rep-BMRefined(s.sensor)) = mk-List(s.sensor, n)임을 증명할 수 있다. 같은 방법으로 나머지 필드인 pressure, heater, alarm, alertPos 등에 대해서도 각각 이에 대응하는 리스트 표현이 존재함을 동일한 방식으로 증명할 수 있다.

이상의 공리와 보조정리로부터 다음 과정을 통해 [정리 4.2]에서 표현된 다음의 *BMRefined* 적합성 증명을 소개한다.

h s_1 : *BoilerState* 가정 h

1 (*rep-BMRefined*(s_1)) : *BMRefined*
[정의 4.2], 가정 h

2 *retr-BoilerState* (*rep-BMRefined*(s_1)) = s_1
[보조정리 4.3], 스텝1, 가정 h

3 s_1 = *retr-BoilerState*((*rep-BMRefined*(s_1))
교환법칙, 스텝 2

4 s_2 : *BMRefined* • (s_1 = (*retr-BMRefined*(s_2)))
[공리 4.3], 스텝 1,3

따라서, $\forall s_1 BoilerState \bullet \exists\, s_2 BMRefined \bullet s_1 = retr\text{-}BoilerState\ (s_2)$ 관계가 성립함

5. 정형 명세의 패턴화

이상에서의 내용은 클래스 기능을 정형 명세하고 이에 관한 적합성을 증명하기 위한 구조와 검증 방법을 살펴보았다. 이러한 내용은 컴포넌트의 기능 이해를 위해 좋은 정보가 될 수 있으며 이들은 컴포넌트 명세를 위한 패턴을 구성하는 일부로 활용될 수 있다. 패턴에 관한 구조 및 정의는 [5, 6]에 나타났으며 본 논문에서는 Gamma의 패턴 언어를 확장하여 BoilerMonitor 명세에 적용한다.

⊙ Pattern name

보일러 모니터 패턴(Boiler Monitor Pattern)

○ Abstract

본 패턴은 보일러에서 나오는 온도와 압력에 대한 이상 신호를 감지하여 경보를 발생시키는 경보 제어 시스템의 기능을 서술하며 추상 명세로부터 설계에 이르는 단계를 검증가능한 형식으로 서술한다.

◎ Context

이 시스템은 보일러 내의 각 센서에서 발생하는 신호를 감지하여 위험 신호에 대한 위치와 경고를 발생시키는 감시 시스템으로서 보일러 각 부분의 온도를 측정하는 최대 n개의 온도 센서와 압력을 측정하는 1개의 압력센서로 구성된다. 측정된 온도 혹은 압력이 위험 값을 넘어서면 히터의 전원을 차단한다. 만일 히터의 전원이 차단되지 않을 경우 해당 센서에 대한 경고 램프를 작동시키고 보일러 시스템에 대한 경보를 발생시킨다. 이러한 시스템에 대한 명세를 구성하는 요소로, *n*은 최대 센서의 수를, *Mode*는 *on, off* 상태를, *State*는 경보 시스템의 상태를 표현하기 위한 레코드 타입으로서 필드 요소로 *sensor, pressure, tempr, heater, alarm, alertPos*를 갖는다. *sensor*는 *n*개의 센서 중 경계치 이상의 값들이 들어오는 센서를 의미하며, *pressure*는 압력 센서를, *alarm*은 시스템이 경보 상태인지의 여부를, *alertPos*는 각 센서 번호와 위험 수준인지의 여부를 표시하는 플래그 쌍으로 구성되어 그 신호 값이 위험 수준에 있는지를 표시한다. 이 때 *alertPos*에 표시되는 신호는 반드시 센싱된 신호 중의 일부가 되어야 하며 보일러 내의 압력과 온도는 *MaxP, MaxH* 이하로 유지해야 한다.

◍ Problem

초기 명세단계부터 정형기법을 적용하여 보일러 경보 시스템을 설계할 경우 다음과 같은 문제가 발생한다.

- 보일러 경보 시스템 서술에서 나타나는 각종 제약 사항들을 빠짐없이 정형적으로 표현하는 문제
- 정형적으로 서술된 경보시스템 명세에 대한 설계가 초기 제약사항을 모두 포함하고 있는지 확인하는 문제
- 프로그램 작성시 프로그램 코드가 설계의 제약사항을 정확히 준수하는지 확인하는 문제

◉ Forces

본 설계 컴포넌트에서는 위에 서술된 문제들 중 다음 사항에 대해 중점적으로 설명한다.

- 초기 자연어 명세로부터 정형 명세를 유도하는 방법.
- 자연어 명세와 정형명세 간의 정확성을 확인하는 방법
- 초기 정형명세로부터 구체적인 정형명세를 개발하는 방법
- 초기 정형명세와 구체화된 정형명세 간에 정확성을 확인하는 방법

• Solution

제어 시스템의 핵심 요소는 시스템이 갖는 제약사항을 빠트림 없이 기록하는 것이다. 시스템의 제약사항은 선 조건과 후 조건, 그리고 자료 불변자로서 표현된다.

1) 초기 자연어 서술에서 나타난 제약 사항들 각각에 대해 다음 사항을 설정한다
   - 선조건, 후조건
   - 자료 불변자
   - 메서드
2) 클래스를 표현하는 이론으로서 각종 증명 규칙을 정의하거나 도구를 사용하여 유도한다
3) 적합성 검증을 하기 위한 검색함수를 정의한다
4) 정리 및 보조정리를 사용하여 위에서 정의된 증명 조건들이 만족되는지를 확인한다

• Consequences

- 유지보수 : 모든 조건들은 관리할 수 있는 수준으로 명시화되어 변화 요소를 추적할 수 있음
- 자료무결성 : 자료 표현에 관한 불변자를 사용하므로 원천적으로 각 자료를 언급하거나 변경하는데 있어 자료의 무결성을 보장함
- 정보은닉 : 클래스 구조를 채용한 표현으로 자료은닉을 자연스럽게 지원함

• Functional Description in VDM++

```
class BoilerMonitor
instance variables
 IdNum = {1,···,n}
 Mode  = {ON, OFF}
 MaxP : nat
 MaxH : nat
 State :: sensor   : IdNum-set
          pressure : nat
          heater   : Mode
          alarm    : Mode
          alertPos : IdNum-set → Mode

BoilerState = State
inv-BoilerState(s) : = dom(s.alertPos) ⊆
s.sensor ∧ rng(s.alertPos) ⊆ s.alarm

methods
alert (s) : =
 ext wr s : BoilerState
 pre true
 post (s.alarm = ON ) ⇔
   (¬(s.heater = OFF) ∧ (s.pressure ≥ MaxP ∨
   s.tempr ≥ MaxT) ∨ (s.alarm' = ON)) ∧
   s.sensor = s'.sensor
   s.pressure = s'.pressure ∧
   s.alertPos = s'.alertPos ∧
   rng(s.alertPos) = ON

end BoilerMonitor

class BMRefined
instance variables
 Value = {1,···,max}
 Mode = {ON, OFF}
StateR ::   sensor1 : IdNum*
            pressure1 : N
            heater1 : Mode
            alarm1 : Mode
            alertPos1 : IdNum* → Mode
BMRefinedState = StateR
inv-BMRefinedState((mk-StateR(s1, p1, h1,
 a11, aP1)) : =
∀i : N1 • (i ∈inds dom(s.aP1[i]) ∧ i ∈ inds
s.s1[i]) ⇒ (s.s1[i] ⇒ dom(s.aP1[i]) ∧
(∀i ∈inds rng(s.aP1[i]) ∧ i ∈ inds s.a11) ⇒
(s.a11 ⇒ rng(s.aP1[i])

methods
newAlart() : =
  ext wr s : StateR
  pre true
  post ((s.alarm1 = ON) ⇔
    (¬(s.heater1 = OFF) ∧ (s.pressure1 ≥ MaxP∨
     s.tempr1 ≥ MaxT) ∨ (s.alarm1' = ON)) ∧
   ∀i : nat • i∈ ( inds s.sensor[i] =
   s'.sensor[i]) ∧
   s.pressure = s'.pressure ∧
   ∀i∈(inds s.alertPos1[i] ∩ inds
     s'.alertPos1[i]) ∧
     inds s.alertPos1[i] = inds
    s'.alertPos1[i] ∨ s.sensor1[i]

retr-BoilerState (mk-BMRefiened(s1, p1, h1, a11,
aP1)) : =
  let sensor = s1[i],
      pressure = p1,
      heater = h1,
      alarm = a11,
      alertPos = aP1[i]
  in  mk-BoilerState(sensor, pressure, heater,
      alarm, alartPos)

rep-BoilerState(mk-BoilerState(sensor, pressure,
heater, alert, alertPos) : =
let  s1  = mk-List(sensor, n),
     p1  = pressure,
     h1  = heater,
     a1  = mk-List(alert, n),
     ap1 = mk-List(alertPos, n)
in mk- BMRefined (s1, p1, h1, a11, aP1)

∀ s1∈BoilerState • ∃ s2∈BMRefined • s1 =
 retr - BoilerState(s2)

end BMRefined
```

단, 위1의VDM++ 구문 상의 ' (프라임) 기호는 (후크)를 대신하여 사용되었음

⊛ Related Patterns

- Nuclear Reactor Controller patterns 원자력 발전소 shutdown 시스템의 기능을 서술한 제어 컴포넌트
- Control Software Process Boolling pattern 제어 소프트웨어 개발에 대해 적용했던 소프트웨어 개발 공정을

관리하는 각종 도구 및 절차에 관해 설명하는 개발 컴포넌트

6. 결 론

컴포넌트의 올바른 사용을 위해서는 명확한 기능의 명세와 이해가 선행되어야 한다. 컴포넌트의 기능 요소에 대한 명확한 기능 표현과 이해의 수단으로 본 논문에서는 정형 기법의 활용을 살펴보았다.

본 논문의 핵심 내용은 다음 두 가지 사항으로 요약될 수 있다. 첫째, 컴포넌트의 기능 부분에 관한 명세 단위로 패턴에 기반한 정형 명세를 시도하였고 이를 위해 VDM++의 표기와 의미의 확장을 시도하였다. 컴포넌트에 대한 문서화 방법으로의 정형 명세 기법은 소프트웨어 개발자들에게 적극적으로 보급되기에는 국내 산업계 현실상 시기 상조라 볼 수도 있으나 언급한 바와 같이 검증 혹은 증명이 요구되는 실시간 응용, 고신뢰 시스템 개발에서는 아직 이를 대체시킬 수 있는 대체 기술이 아직 제시되지 못한 실정이다. 관련연구 부분에서 언급한 바와 같이 이 부분은 Lano, Larman 등에 의한 VDM++를 이용하여 객체를 표현한 시도가 있었으나 검증을 필요로 하는 소프트웨어의 경우 인터페이스 서술만을 정형 언어로 했다고 해서 검증에 관한 근본적인 문제 해결에 직접 도움을 준다고는 생각하지 않는다. 명세 기법으로서의 패턴이 기여하는 부분이 단순한 기능 서술이 아니라 개발 단계에 도움을 줄 수 있는 정보를 제공하는데 있다면 검증에 관한 정보 전달도 중요한 역할로 생각될 수 있으며 이를 위한 적절한 표현 수단으로서의 패턴 구조를 확장하였다.

둘째, 컴포넌트 명세의 검증에 관한 사항으로 설계 혹은 분석 단계에서 컴포넌트 사용이 고려되고 이로인해 컴포넌트의 명세가 동일한 단계에서 검토된다면 이 부분 역시 다른 설계 부분과 마찬가지로 증명이 가능해야 한다. 이 부분에 관해서는 [16]에서 보여진 해석표의 방법이 있으나 2절 후반부에 이 방법의 한계를 지적한 바 있다. 본 논문은 검색함수가 갖는 중요성과 적합성 증명 과정에서의 활용을 소개하였고 이러한 요소가 패턴의 명세에 역시 포함되어야 할 요소임을 제안하였다.

컴포넌트의 특성 중 하나는 동적인 행위 특성에 관한 명세에 관한 사항이다. 시스템의 동적 특성은 모형 기반 명세 언어인 VDM++로 표현하기에는 한계가 있으며 필요하다면 CCS, LOTOS를 이용하는 명세 기법의 도입을 고려할 수도 있다. 이질적인 정형 명세 기법을 혼용하여 시스템의 동적 특성을 서술할 경우 발생하는 문제는 좀더 심도 깊게 다루어져야 할 사항이며 본 연구의 연장선에서 다루어야 할 향후 연구 주제이다.

참 고 문 헌

[1] G. Grahn, Transition from Conventional to Component-based Development, Int'l Workshop on Component-Based Software Engineering, pp.78-82, 1999.

[2] C. Szyperski, Component Software : Beyond Object Oriented Programming, Addison-Wesley, 1997.

[3] 홍기형, 서동수, "차세대 웹에서의 컴포넌트 소프트웨어, 정보처리학회지", Vol.6, No.3, pp.45-51, 1999.

[4] Butler Group, Component-Based Development : Application Delivery and Integration Using Componentised Software, Butler Group White Paper, 1998.

[5] T. Mowbray, R. Malveau, CORBA Deign Pattern, John Wiley & Sons, 1997.

[6] Gamma, E. et al, Design Patterns : Elements of Reusable Object-Oriented Software, Addison- Wesley, 1995.

[7] J. Coplien, D. Schmidt, Pattern Languages of Programming Design, Addison-Wesley 1995.

[8] J. Woodcock, J. Davis, Using Z Specification, Refinement, and Proof, Prentice Hall, 1996.

[9] C. B. Jones, Systematic Software Development using VDM, 2nd ed. Prentice Hall 1990.

[10] R. Milner, Communication and Concurrency, Prentice Hall, 1991.

[11] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[12] H. Ehrig, B. Mahr, Fundamentals of Algebraic Specification 1 : Equations and Initial Semantics, Springer-Verlag, 1985.

[13] K. Lano, H. Haughton, Reasoning and Refinement in Object-oriented Specification Languages, ECOOP '92 Conference Proceedings, Springer-Verlag, 1992.

[14] A. Alencar, J. Goguen, OOZE : An Object-Oriented Z Environment, ECOOP '91 Proceedings, LNCS 512, pp.180-199, Springer-Verlag, 1991.

[15] E. Durr, VDM++ Language Reference Manual, Afrodite Report AFRO/CG/ED/LRM/V11, 1995.

[16] K. Lano, et al, Formalising Design Patterns, BCS-FACS Northern Formal Methods Workshop, Iikley, UK. Springer-Verlag 1996.

[17] J. Fiadeiro, T. Maibaum, Describing, Structuring and Implementing Objects, in Foundations of Object Oriented languages, LNCS 489, Springer-Verlag, 1991.

[18] C. Larman, Applying UML and Patterns, Prentice Hall PTR, 1998.

서 동 수

Email: dseo@cs.sungshin.ac.kr

1986년 중앙대학교 컴퓨터공학과(이학사)

1990년 영국 맨체스터 이공대학(이학석사)

1994년 영국 맨체스터 이공대학(공학박사)

1994년~1998년 한국전자통신연구원 선임연구원

1998~현재 성신여자대학교 컴퓨터정보학부 조교수

관심분야 : 정형기법, 소프트웨어공학, 분산객체기술