□특집□

컴포넌트 정형명세

박 찬 진[†] 우 치 수[††]

◈ 목 차 ◈

컴포넌트 기술의 등장으로 인해 부품 조립 형태의 개발 프로세스가 일반화되고 있다. 하지만, 소프트웨어 부품에 대한 명세는 구문적인 측면은 잘 정의되어 있으나, 의미적인 측면은 자연언어로 기술된 비정형적인 문서에 의존하고 있다. 컴포넌트 기술이 성숙하기 위해서는 컴포넌트 및 컴포넌트 프레임워크를 엄밀하고 애매함이 없게 표현할 수 있는 정형명세 기법이 도입되어야 한다. 또한, 정형명세가 실제적으로 사용되기 위해서는 표현 언어가 이해하기 쉬워야 하며, 자동 분석을 위한 도구 지원이 필요하다.

본 논문에서는 정형명세 언어로 컴포넌트 인터페이스의 추상 모델을 기술하고, 이 추상모델을 사용하여 컴포넌트 사용 오류를 검출하는 방법을 보인다.

1. 서 론

최근 컴포넌트 기술이 복잡해지고 규모가 커지는 소프트웨어 개발에 대한 해결책으로 등장함에 따라, 국내 및 국외에서 소프트웨어 개발 생산성 향상 수단으로 널리 이용되고 있다. 또한, EJB, CORBA, COM과 같은 컴포넌트 프레임워크가 발표되고, 이를 기반으로 하여 전자상거래와 같은 특정 도메인에서의 컴포넌트 패키지들이 만들어지고 있다. 이러한 컴포넌트 기술의 성숙은 소프트웨어 재사용 범위를 기업 차원에서 전세계로 확장 시키고 있다. 부품 형태로 판매되고, 이들을 조립하여 시스템을 구성하는 프로세스는 소프트웨어 생산성을 증대 시키며, 유지보수 비용을 감소시키는 데 효과적으로 적용되고 있다.

컴포넌트를 통해 시스템을 구성하고자 하는 개발자에게는 컴포넌트가 의존하고 있는 프레임워크와 컴포넌트 자체에 대한 이해가 필수적이다. 그러나, 이러한 컴포넌트 및 컴포넌트 프레임워크에 대한 명세가 비정형적인 문서의 형태로 전달되기 때문에 명세 자체의 오류나 명세 내의 문장들 간의 불일치를 포함할 수 있다. 이는 컴포넌트 자체의 에러나 컴포넌트의 오용을 통해 시스템의 오작동을 유발할 수 있다.

현재 사용되고 있는 컴포넌트에 대한 명세는 컴포넌트가 구현하는 인터페이스의 구문적인 측면을 기술하는 IDL(Interface Definition Language)로 배포된다. 이러한 구문적인 측면을 기술하는 IDL은 컴포넌트의 기능을 사용하기 위한 클라이언트의 구문적인 정합성만을 보장할 수 있다. 컴포넌트를 올바른 사용을 보장하기 위해서는 컴포넌트 명세에 의미적인 측면을 기술할 수 있어야

† 정 회 원 · 서울대학교 컴퓨터공학부 박사과정
†† 정 회 원 : 서울대학교 전산학과 교수

한다. 컴포넌트에 대한 의미 측면을 기술하는 일반적인 방법은 비정형적인 자연언어 문장들을 통해 이루어진다.

컴포넌트에 대한 명세를 정형화하는 것은 정형적이지 못한 명세를 수학적인 모델링을 통해 표현으로써, 명세의 모호성 및 명세 자체 내에 포함된 불일치들을 제거할 수 있게 한다. 또한, 명세를 통해 컴포넌트 구현 및 컴포넌트 사용 상의 오류를 찾아낼 수 있도록 한다. 현재까지 Z나 VDM과 같은 여러 정형 명세 언어들이 소개되었지만, 수학적인 배경지식이 부족한 사람들에게는 표현 방식이 너무 어려운 단점을 가지고 있다. OCL(Object Constraint Language)[1]은 이해하기 쉽고, 일반적인 개발자들을 대상으로 작성된 언어이기 때문에 이러한 정형명세의 유용성과 정형명세의 어려움 간의 차이를 좁혀줄 수 있는 명세 언어라고 할 수 있다. OCL을 통해, 정형 명세가 학술적인 분야의 주제에서 소프트웨어 산업계에서의 적용의 대상으로 이어질 수 있을 것이다.

본 논문에서는 현재 컴포넌트의 구문적 측면을 기술하는 IDL과 다이어그램 위주의 객체지향 설계모델에 정형화된 제약조건을 표현할 수 있는 OCL을 결합하여, 컴포넌트의 올바른 사용을 검증할 수 있는 방법을 제시한다. OCL을 통해 컴포넌트의 인터페이스에 대해 추상적인 모델을 기술할 수 있고, 이 추상 모델은 컴포넌트의 사용 및 구현에 대한 오류 분석에 사용될 수 있다.

본 논문의 구성은 다음과 같다. 기존의 컴포넌트를 기술하는 방식을 설명하기 위해 2장에서는 IDL과 OCL을 설명하고, 3장에서는 정형명세를 통해 컴포넌트 구현 및 사용 상의 오류를 분석할 수 있는 방법을 제안한다. 4장에서는 관련연구로 컴포넌트 프레임워크 명세를 정형화하여 비정형적인 명세 상의 모순 및 불일치를 찾아내려는 시도를 소개한다. 5장에서는 결론 및 향후 연구에 대해 기술한다.

2. 기존의 컴포넌트 기술 방식

본 논문에서 컴포넌트는 COM 또는 CORBA, EJB등의 컴포넌트 프레임워크를 바탕으로 정의되는 물리적으로 캡슐화된 기능단위를 말한다. 컴포

```
import "unknwn.idl";
 [
  object,
  uuid(E16F55D1-C216-40EE-A960-42747ECBC104),
  helpstring("IStack Interface"),
  pointer_default(unique)
 ]
 interface IStack : IUnknown
 {
  [helpstring("method Pop")]
    HRESULT Pop([out, retval] short* elt);
  [helpstring("method Push")]
    HRESULT Push([in]short elt);
  [helpstring("method IsEmpty")]
    HRESULT IsEmpty([out, retval]BOOL* ret);
  [helpstring("method IsFull")]
    HRESULT IsFull([out, retval]BOOL* ret);
 };
```

```
[
 uuid(8EA4EC9D-6272-4E66-A1B4-5ADEF6482228),
 version(1.0),
 helpstring("StackComp 1.0 Type Library")
]
library STACKCOMPLib
{
 importlib("stdole32.tlb");
 importlib("stdole2.tlb");
 [
  uuid(64AA7C68-982B-4E56-9B9F-FCE7EDC6B066),
  helpstring("Stack Class")
 ]
 coclass Stack
 {
  [default] interface IStack;
 };
};
```

(그림 1) Stack에 대한 IDL 표현

넌트는 잘 정의된 아키텍처를 바탕으로, 인터페이스에 의해 정의되는 명확한 기능을 수행하는 독립적이고 대치 가능한 단위이다[2].

컴포넌트에 대한 일반적인 명세 방식은 컴포넌트 사용을 위한 구문 구조(IDL)와 사용자 매뉴얼 및 간단한 데모 프로그램 형태이다. 사용자 매뉴얼이나 데모 프로그램으로 표현된 컴포넌트의 의미적 측면에 대한 기술은 정상적인 시나리오에서의 컴포넌트 작동 만을 표현하고 있어서 컴포넌트에 대한 명확한 이해에 도움을 줄 수 없다. 먼저, 컴포넌트 구문구조 명세로 사용되는 IDL을 설명하고, 컴포넌트의 의미적 측면을 정형적으로 표현하기 위해 객체지향 설계 모델에서의 정형 제약조건 표현 언어인 OCL을 소개한다.

2.1 IDL

IDL은 컴포넌트의 서비스를 프로그램 언어 독립적인 방식으로 구문적인 측면을 기술한다. IDL은 컴포넌트 개발 관점과 통합 관점을 중계하는 역할을 한다. (그림 1)은 COM의 IDL을 사용하여 스택을 구현하는 컴포넌트에 대한 기술을 보여주고 있다.

위의 예에서 Stack은 컴포넌트의 이름이고, IStack은 Stack이 구현하는 인터페이스이다. Stack을 사용하는 입장에서는 IStack에서 지원되는 4개 오퍼레이션을 통해 컴포넌트에 접근할 수 있다. 전술한 바와 같이 IDL은 컴포넌트 사용의 구문적인 부분만을 기술하며, 오퍼레이션에 대한 의미 또는 제약 사항들을 기술하지 않는다[3]. 위의 IDL로 알 수 있는 것은 Stack 컴포넌트가 IStack이라는 인터페이스를 가지고 있고, IStack의 4개 오퍼레이션을 통해 컴포넌트에 접근할 수 있고, 각 오퍼레이션을 사용하기 위해서는 인자를 어떻게 주어야 하고 리턴 값은 어떻게 받을 수 있는 가에 대한 정보 뿐이다.

하지만, Stack 컴포넌트를 사용하기 위해서는 일반적으로 알려진 Stack에 대한 의미적인 측면을 알고 있어야 한다. 다음은 Stack 컴포넌트의 IStack을 사용하기 위한 규칙이다.

(1) 처음에 Pop을 할 수 없다.

(2) Pop의 횟수가 Push의 횟수보다 작거나 같아야 한다.

하지만, 이러한 가정은 위와 같이 정형화 되지 않은 자연언어로 표현되므로, 규약이 복잡하거나 많아지게 되면, 자체적으로 모순을 가질 수 있고, 모호성을 가질 수 있다. 더욱이, 컴포넌트 사용에 대한 검증이 컴포넌트와의 통합작업 이후라야 가능하다면, 컴포넌트 개발 작업과 컴포넌트를 기반으로 하는 시스템 구성 작업이 병렬적으로 진행되기 힘들다. 또한, 컴포넌트에 대한 잘못된 이해에서 비롯된 오류가 통합 이후에 발견된다면, 전체 시스템 구조에 영향을 줄 수 있다. 컴포넌트의 의미적 측면을 수학적 모델링을 통해 정형 명세화 하여 컴포넌트 사용 코드를 도구를 통해 분석할 수 있다면, 컴포넌트 구현과 사용에 대한 작업을 독립적으로 진행하여 개발 작업의 생산성을 향상시킬 수 있다.

2.2 OCL

OCL은 객체 모델 상에 불변조건, 선조건, 후조건 및 기타 제약사항을 기술하기 위한 표준언어로, 설계 모델링 언어 표준인 UML에 포함되어 있다. OCL은 원래 IBM의 보험관련 업무를 위한 비즈니스 모델링 언어로 개발되었고, 객체지향 개발 메소드 중의 하나인 Syntropy 메소드에 기초를 두고 있다[4]. OCL은 VDM이나 Z와 같이 엄격한 수학적 모델링을 적용하기 위한 것이 아닌 객체지향 설계자를 위한 평이하고, 이해하기 쉬운 형태의 언어이다. 다음의 예(그림 2)는 앞에서 표현된 Stack에 대한 의미적 측면을 OCL로 표현한 것이다.

IStack은 Sequence 타입의 Elements를 추상 상태변수로 가지며, Elements는 항상 0보다 크거나 같

```
IStack
self elements          -- Seuence(short)
self elements->size >= 0
self.elements->size < MAX
initial: self.elements->size = 0

IStack..Pop(elt. short*) . HRESULT
pre ' self.elements->size > 0
post' self.elements@pre = self.element->append(*elt)

IStack Push(elt. short) : HRESULT
pre ' self.elements->size < MAX
post. self element = self.element@pre->append(elt)

IStack..IsEmpty(). BOOL
post. result = ( self element->size = 0 )

IStack .IsFull(). BOOL
post  result = ( self element->size = MAX )
```

(그림 2) IStack 인터페이스에 대한 OCL 표현

고 MAX보다 작아야 한다는 불변조건을 가진다. Pop 오퍼레이션은 수행 전에 Elements의 크기가 0보다 커야 한다는 선조건을 가지고, 수행 후에는 수행 전 Elements가 수행 후 Elements에 elt를 Append한 결과와 같아야 한다는 후조건을 가진다[1]. Push에 대해서도 같은 선조건과 후조건을 기술할 수 있다. IsEmpty 오퍼레이션은 오퍼레이션의 결과가 현재 Elements의 개수가 0이면 참이고, 그렇지 않으면 거짓을 가짐을 의미한다. Elements는 IStack의 의미를 명세로 표현하기 위해 시퀀스 타입의 속성을 포함시켰다.

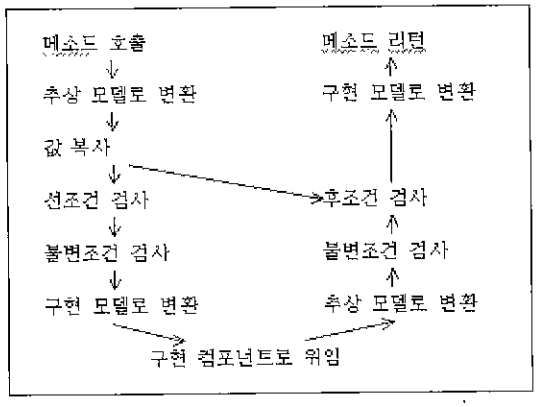
OCL은 UML에 포함된 언어로 설계 모델을 대상으로 한다. 위와 같은 컴포넌트의 인터페이스에 대한 명세는 컴포넌트 개발자와 시스템 구성자 간의 약정으로 사용될 수 있다. 컴포넌트 개발자는 위와 같은 인터페이스 명세를 정제(Refinement)하여 컴포넌트를 구현하고, 시스템 구성자는 명세를 사용하여 컴포넌트의 행위를 명확히 파악할 수 있다.

1) @pre는 OCL에서 시간을 표현하기 위한 표기이며, 여기에서는 오퍼레이션 이전의 elements를 표현하기 위해 사용되었다.

3. 컴포넌트 정형명세

컴포넌트에 대한 정형 명세를 기술하는 것은 수학적 모델을 통해 모호성 및 명세 자체의 불일치를 없애 컴포넌트의 기능을 엄격하게 선언할 수 있다. 또한, 컴포넌트 정형명세는 잘못된 컴포넌트 사용을 검증할 수 있는 제약조건으로 사용될 수 있다. 즉, 인터페이스(클라이언트와 컴포넌트 간의 약정) 위반에 대해 예외(Exception)를 발생시켜, 위반 사항에 따른 행동을 수행할 수 있다[2].

컴포넌트 명세는 컴포넌트 개발작업 및 시스템 구성작업에서의 인터페이스 위반을 검출하는 데 사용될 수 있다[5]. [5]에서는 클래스의 추상적 모델을 RESOLVE와 같은 정형 명세 언어를 사용하여, 집합, 스트링, 함수, 트리와 같은 수학적인 모델로 표현한다. 이를 바탕으로 인터페이스 위반 검사를 위한 클래스 템플릿을 생성하여, 클래스 사용코드의 인터페이스 위반과 클래스 구현코드의 인터페이스 위반을 체크하는 방법을 제안하고 있다.



(그림 3) 인터페이스 위반 검출 과정

2) OCL로 표현된 불변조건, 선조건, 후조건 등의 제약사항을 자바 언어의 예외를 통해 표현하는 노력이 있다. http://www-st.inf.tu-dresden.de/ocl/

〈표 1〉 OCL 명세언어 요소에 해당하는 테스트 스텁 요소

OCL 명세 언어	테스트 스텁
elements	list<short> elements
pre: self.elements->size > 0	if (elements.size() <= 0) { Error 발생 }
post: self.elements@pre = self.elements->append(*elt)	if (elements.back() != *elt) { Error 발생 } elements.pop_back();

이와 같은 방식의 인터페이스 위반 검사는 컴포넌트 소프트웨어에도 적용될 수 있다. 컴포넌트의 인터페이스에 대한 OCL 정형 명세를 바탕으로 컴포넌트의 잘못된 사용을 체크하는 코드를 생성할 수 있다. 컴포넌트의 인터페이스는 객체지향 설계 모델에서 추상 클래스에 해당하고, OCL은 이 추상 클래스를 문맥으로 하여 클래스의 불변조건 및 오퍼레이션들의 선조건, 후조건을 기술할 수 있다. OCL로 표현된 인터페이스의 추상 상태를 구현 언어 요소[3]에 맵핑하여, 테스트 스텁을 위한 코드를 생성할 수 있다. 이러한 테스트 스텁 코드는 클라이언트의 인터페이스 위반 검사를 위한 가상 컴포넌트로 사용될 수 있다.

예를 들어, Stack 컴포넌트를 사용하는 클라이언트 코드가 첫번째 규칙인 처음에 Pop을 할 수 없다를 어기고, Pop을 먼저 호출하고 있다면, Pop의 선조건인 "self.elements->size > 0"을 어기게 된다. 이러한 규약 위반을 검출하기 위해 명세 상의 요소를 테스트 스텁 상의 요소로 변환한다.

명세 상의 속성인 Elements를 C++ 표준 템플릿 라이브러리 상의 List 타입의 변수에 맵핑할 수 있고, 선조건인 "self.elements->size > 0"은 Pop 오퍼레이션의 시작점에서의 체크 코드로 맵핑할 수 있다. 후조건인 "post: self.elements@pre = self.elements->append(*elt)"는 Pop 호출 후의 Stack 컴포넌트의 상태이며, 테스트 스텁에 의해 후조건을 만족하도록 하여야 한다. 위의 경우에는 현재의 Elements에 Append를 해야 이전 Elements와 같아 지므로, 후조건을 만족시키기 위해서 테스트 스텁의 Pop에서는 List 변수 Elements에 Pop을 수행해야 한다.

명세 상의 선조건과 후조건에 대한 테스트 스텁 코드는 개발자에 의해 직접 작성될 수 있고 가능한 부분에 대해서는 자동 생성될 수 있다. 도구에 의한 자동 생성을 위해서는 OCL 타입 및 오퍼레이션에 대한 구현 언어 맵핑이 선행되어야 하고, 후조건을 만족시키기 위한 테스트 스텁 코드를 자동으로 생성하기 위한 전략이 필요하다.

정형명세는 구현 언어와 매우 다르며, 표현법이 어렵기 때문에 적용하기가 힘들다. 따라서, 소프트웨어의 모든 구성요소에 대해 정형명세를 적용하는 것은 신뢰성이 특히 요구되는 도메인이 아닌 경우 무리한 작업이다. [6]에서는 과도 명세를 피하기 위한 명세의 비결정적 접근에 대해 설명한다. 또한, 명세와 구현 사이의 관계를 정제관계(Refinement)로 정의하고, 비결정적인 명세는 다양한 구현을 위한 선택의 여지를 남길 수 있으며, 불필요한 구현 세부사항을 감출 수 있다고 한다.

컴포넌트는 하나의 부 시스템을 구성하고, 구현 내부에 여러 구성요소들과 이 들간의 관계 및 행위로 이루어져 있으며, 인터페이스를 통해 구현 기능을 감춘다. 컴포넌트에 정형명세를 적용하는 것은 개별 구성요소 각각에 대한 명세를 기술하지 않고 인터페이스에 대해서만 작성되기 때문에, 정형명세 표현을 위한 노력과 정형명세를 통한 잇점을 고려하면 적절한 수준에서의 정형성을 부여하는 것이라고 볼 수 있다.

3) C++의 경우, Standard Template Library의 vector<>와 같은 클래스를 사용할 수 있다.

4. 관련 연구 - 컴포넌트 프레임워크 관점

컴포넌트 프레임워크 명세가 정형적이지 않다면, 프레임워크가 컴포넌트 사용 및 구현을 위한 규칙을 정확히 이해하기 힘들다. 그러나, 대개의 컴포넌트 프레임워크 명세는 정형화되지 않은 자연언어 형태의 문서를 통해 발표되어 왔다. 정형화되지 않은 명세는 자체 내에 오류를 가질 수 있고, 서로 모순되는 문장을 가질 수 있다. 이 외에도, 관련 정보가 서로 떨어져 있어 전체를 파악하기 힘들고, 불명확한 정의 때문에 프레임워크의 의미를 이해하기 힘들 수 있다[8]. CORBA, COM, EJB와 같은 컴포넌트 프레임워크의 비정형적 명세를 통해 야기될 수 있는 문제를 해결하기 위해, 이 들 프레임워크 표준에 대한 정형 명세를 작성하는 노력들이 있다[7, 8].

[7]에서는 문장으로 표현된 COM 명세 상의 컴포넌트의 집합합성 규칙[4]이 COM 인터페이스 획득 규칙과 모순됨을 COM 명세를 Z언어로 정형화하는 작업을 통해 보여준다.

5. 결론 및 향후 연구

지금까지 컴포넌트를 정형적으로 기술하기 위하여, 기존의 컴포넌트 기술 방식인 IDL을 소개하고 OCL을 사용하여 컴포넌트의 의미적 측면을 정형적으로 표현하는 방식에 대해 설명하였다. 또한, 이러한 정형적 표현은 컴포넌트 사용 시의 오류를 검출하는 데 사용될 수 있음을 보였다. 컴포넌트 프레임워크는 컴포넌트 기술이 사용되는 모든 부문에 영향을 줄 수 있다. 컴포넌트 프레임워크에 대한 정형화를 통해 명세의 명확성 및 일관성을 유지할 수 있다.

본 논문에서는 컴포넌트의 인터페이스에 대해 정형 명세를 기술함으로써 컴포넌트의 의미적 측면을 나타낼 수 있다고 가정하였다. 그러나, 컴포넌트는 기능적으로 분리된 하나 이상의 인터페이스들을 구현할 수 있으며, 인터페이스들이 상호의존성을 가지는 경우에 인터페이스 각각에 대한 정형 명세가 컴포넌트 전체의 정형명세를 표현하지 못할 수 있다. 현재 OCL은 단일 클래스(하나의 인터페이스)에 대한 제약 조건을 표현하는 수준이며, 상호 작용하는 클래스 그룹의 명세를 표현하는 데 한계를 가진다. OCL의 표현능력 및 도구지원에 대해서는 활발히 연구가 수행되고 있다[10, 11].

컴포넌트 프레임워크는 일반적으로 컴포넌트의 생성과 소멸, 트랜잭션, 데이터베이스 연동, 보안 등과 같은 프레임워크 지원 기능을 인터페이스를 통해 제공하고 있다. 컴포넌트 프레임워크에 대한 정형화 시도는 단순히 프레임워크 상의 모호성 및 불일치를 찾아내는 것 이외에도, 컴포넌트 구현 및 사용 시 프레임워크 사용에 대한 오류를 검출하기 위해 사용될 수 있다. 이러한 지원 기능에 대한 추상 모델을 정형화된 형태로 명세함으로써, 프레임워크 사용에 대한 잘못된 사용을 검사할 수 있을 것이다.

4) COM에서 지원하는, 기존의 컴포넌트를 바탕으로 새로운 컴포넌트를 작성하는 컴포넌트 재사용 방법에는 포함합성과 집합합성이 있다. COM 표준은 집합 합성된 컴포넌트가 하나의 컴포넌트로 표현되어야 한다고 규정하는 동시에, 한 컴포넌트에서 하나의 인터페이스를 얻으면 컴포넌트 내의 모든 인터페이스를 얻을 수 있어야 한다고 규정한다. 두 개의 컴포넌트를 하나의 컴포넌트로 표현하기 위해 임기응변적인 컴포넌트 구현 방식을 채용하였고, 이러한 구현 방식은 많은 참고 문헌에 자연언어 및 도해를 통해 기술되어 있다[9]

참고문헌

[1] Object Management Group, "Object Constraint Language Specification", Version 1.1, 1997. http://www-4.ibm.com/software/ad/standards/ocl.html

[2] Philippe Kruchten, "Modeling Component Systems with the Unified Modeling Language", First Int'l Workshop on CBSE, in conjunction with ICSE'98, 1998

[3] Cynthia Della Torre Cicalese and Shmuel Retenstreich, "Behavioral Specification of Distributed Software Component Interfaces", IEEE Computer, Vol.32, No. 7, 1997

[4] Steve Cook, John D. Daniels, "Designing Object Systems: Object-Oriented Modeling with Syntropy", Prentice Hall, 1994

[5] Stephen H. Edward, Gulam Shakir, Murali Sitaraman, "A Framework for Detecting Interface Violations in Component-Based Software", Proceedings. Fifth International Conference on Software Reuse, 1998.

[6] Martin Buchi and Emil Sekerinski, "Formal Methods for Component Software: The Refinement Calculus Perspective", Proceedings of the Second Workshop on Component-Oriented Programming (WCOP) in conjunction with ECOOP, June 1997.

[7] Kevin J. Sullivan, Mark Marchukov, and John Socha, "Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model", IEEE Transactions on Software Engineering, Vol. 25, No. 4, July/August, 1999

[8] Joao Pedro Sousa and David Garlan, "Formal Modeling of the Enterprise JavaBeansTM Component Integration Framework", FM' 99: World Congress on Formal Methods, Sep, 1999

[9] Dale Rogerson, "Inside COM", Microsoft Press, 1997

[10] Philippe Collet and Roger Rousseau, "Towards Efficient Support for Executing the Object Constraint Language", 30th Int. Conf. on Technology of Object Oriented Languages and Systems(TOOLS USA'99), August, 1999

[11] Luis Mandel and Maria Victoria Cengarle, "On the Expressive Power of OCL", FM' 99: World Congress on Formal Methods, Sep, 1999

박 찬 진

1994년 서울대학교 계산통계학과 졸업
2000년 서울대학교 전산학 석사.
2000년 서울대학교 컴퓨터공학부 박사과정.
관심분야 : 소프트웨어 공학, 객체지향 설계.

우 치 수

1972년 서울대학교 응용수학과 학사
1977년 서울대학교 전산학 석사
1982년 서울대학교 전산학 박사
1972년-1974년 한국과학기술연구원 연구원
1978년 영국 라피러 대학 연구원
1975년-1982년 울산대학교 전자계산학과 부교수
1982년-현재 서울대학교 전산학과 교수
1985년-1986년 미시간 대학 Post-doc
관심분야 : 소프트웨어 공학, 프로그래밍 언어