제약 언어를 이용한 객체 모델 검증시스템

김 진 수[†] · 강 권 학[††] · 이 경 환[††]

요 약

소프트웨어를 개발하는 과정은 일련의 다양한 모델을 구축하는 과정이라고 할 수 있다. 그러나 개발자들이 작성된 모델을 검증할 수 있는 적당한 방법이 없었다. 기존의 검증 도구들은 주로 구현 단계에서 사용되고 있으나 본 논문에서는 이러한 검증 도구의 원리를 개발의 초기 단계인 분석 단계에 적용해 보았다. 본 논문에서는 기존의 객체 모델링 방법론에서 제시하였던 지침들 뿐만아니라 시스템의 개발자가 객체 모델에 대해서 규정하고자 하는 지침들까지도 제약으로서 표현할 수 있는 제약 언어를 정의하였다. 정의된 제약 언어를 이용하여 객체 모델 작성기에 의해 생성된 객체 모델의 품질 및 일관성을 향상할 수 있는 검증시스템을 구축하였다.

An Object Model Verification System based on the Constraint Language

Jin Soo Kim[†] · Kwon Kang[††] · Kyung Hwan Lee[††]

ABSTRACT

A software development process is regarded as the process of building a series of various models. But developers had no method to verify these models which created subjectively. This paper has adopted initiatives of verification tools to an early phase of analysis, whereas the existing verification tools have mostly been used for implementations. This paper has defined a constraint language that expresses disciplines suggested in the existing methodology as well as guidelines applied by object model editor in the object model. The verification system has been built upon the constraint language, which is proven to enhance the quality and consistency of models constructed by object model editor.

1. 서 론

소프트웨어를 개발하는 과정은 일련의 다양한 모델을 구축하는 과정이라고 할 수 있다. 기존의 기능 중심 모델링은 기능 추상화를 기반으로 하기 때문에 하나의 기능을 여러개의 세부 기능으로 분할하여 시스템의 모습을 모델링한다. 이 모델링의 단점은 기능 중심의 모델을 작성하기 때문에 구현에 의해 큰 영향을 받는다. 자료 중심의 모델링에서는 사용자들이 현재 사용중인 자료들의 논리적 구성체인 엔티티를 중심으로 모델링하므로써 정보 시스템을 구성하는데 적합한 방법론일 수 있다. 그러나 이 모델링의 단점은 다루는 자료와 자료의 처리 부분이 분리되어 있기 때문에, 자료의 변경과 처리 부분의 변경이 서로 독립성을 보장해 주지 못한다는 것이다. 이를 해결하기 위하여 최근에는 실세계의 정보와 정보 처리에 관련된 연산을 함께 객체라는 단위로 묶어 모델링하므로써 실세계를 자연스럽게 모델링할 수 있고, 변화의 가능성이 비교적 적은 객체를 중심으로 모델링하기

† 정 회 원:중앙대학교 기술과학연구소 연구원
†† 정 회 원:국방정보체계연구소 전임연구원
논문접수:1995년 9월 23일, 심사완료:1996년 1월 12일

때문에 개발 과정에서 발생할 수 있는 변화로 인한 파급효과를 최소화할 수 있다[1][2].

이러한 객체 모델링의 산물인 객체 모델은 문제의 분석 및 시스템의 설계에 기반이 된다[2][3][8][16]. 그러나 객체는 단지 응용 영역에 따라서 의미가 달라지는 것이므로, 객체 모델은 응용 영역에 따라서 상당히 다른 양상으로 나타나게 된다. 또한 다양하고 변화가 많은 사용자들의 요구를 정확히 반영하여 올바른 모델을 작성하기 힘들뿐아니라 반영하는 과정에서 개발자의 오류가 많이 발생하게 된다. 그러하여 기존의 각 객체지향 방법론에서는 객체 모델의 품질과 일관성을 높이기 위해 각 객체 모델에 대해서 지켜질 수 있도록 지침을 제시하므로써 일관된 객체 모델을 유지하려고 한다. 또한 모델링 방법론에서 정의되지는 않지만, 일정한 객체 모델링의 경험을 바탕으로 하나의 관례가 된 것들도 있다. 그러나 객체 모델에는 이러한 지침과 관례를 표현할 명시된 방법이 없고, 또한 서로 다른 방법론에 의해 작성된 각 객체 모델에 대해서 일정한 지침과 관례를 올바로게 따르는가의 여부를 자동으로 검증할 도구가 없었다.

소프트웨어를 개발하는 과정에서 초기 단계에 검출된 오류는 유지보수 단계에서 발견된 오류에 비해 수정하는 비용 및 노력이 거의 들지 않는다[1]. 이러한 이유로 객체지향 개발 방법론은 기존의 방법론에 비해 분석 단계에 많은 투자를 요구하고 있다. 따라서 분석 단계에서 일정한 지침을 가지고 정확하게 모델을 검증할 수 있는 도구는 더욱 중요한 의미를 지니게 된다.

기존의 소프트웨어 개발 환경에서는 lint와 같이 C 원시 코드에 대해서 일정한 지침을 검증하는 도구들이 사용되어 왔고 이러한 도구를 객체지향 언어인 C++언어로 확장하고자 하는 노력이 진행되어 왔다[10][11][12]. 본 논문에서는 이러한 구현 단계에 적용되었던 검증 도구를 분석 단계에 적용해 보았다. 먼저 객체 모델에 대한 지침을 명세하기 위하여 모델 제약 언어(MCL:Model Constraint Language)를 제안하였고, 객체 모델을 작성할 수 있는 객체 모델 작성기와 작성된 객체 모델을 명세된 제약 언어로 검증할 수 있는 객체 모델 검증기를 포함한 객체 모델 검증 시스템을 개발하였다.

2. 관련 연구

본 장에서는 기존의 소프트웨어 개발 단계에서 사용되었던 검증 도구인 lint, lint++ 및 Clean++을 살펴보고자 한다.

2.1 lint

lint는 기존의 UNIX 시스템에서 널리 사용되던 C 원시 코드 분석 도구로서 문법적으로는 맞지만 낭비적이거나 오류의 원인이 되기 쉬운 C의 구성자들을 검사하는 도구이다. 주로 다음의 내용들을 검사하며, 필요에 따라서 이러한 사항들은 취사 선택될 수 있다.

- 사용되지 않은 변수들과 함수들
- 값이 할당되기 전에 사용되는 변수들
- 프로그램 내에서 도달할 수 없는 부분들
- 함수에서 반환되지 않는 값을 사용하고자 할 때
- C 컴파일러보다 강력한 타입 검사
- 오류일 가능성이 큰 타입 변형
- 비 이식적인 문자의 사용
- long 타입의 값을 int 변수에 할당하고자 할 때
- 결과가 사용되지 않는 연산자의 사용
- 이전 구문에서 사용되던 오류 발생이 쉬운 형태
- 포인터의 배열(alignment)에 관련된 사항
- 연산의 순서가 정해지지 않은 경우

2.2 lint++

lint++는 C 언어가 C++ 언어로 확장되면서 기존의 lint를 C++ 언어에 맞게 확장하기 위하여 만들어졌다. 다음과 같이 합법적인 C++ 구문이지만 오류일 가능성이 큰 조건들을 검사한다.

- new와 delete 연산의 짝이 맞지 않는 경우
- 파괴자(destructor)를 비가상(non-virtual) 멤버 함수로 선언한 경우
- 포인터 멤버를 가지고 있는 클래스가 복제 생성자(copy constructor)나 할당 연산자에 대해 선언하지 않는 경우
- 할당 연산자가 자기 클래스에 대한 상수 레퍼런스(constant reference)를 반환(return)하지 않는 경우
- 할당 연산자가 자신을 자신에 할당하는 경우를 검

사하지 않는 경우
- 상수 멤버 함수에서 비상수 레퍼런스를 반환하는 경우
- 멤버 함수가 private이나 protected 멤버 데이타에 대한 비상수 레퍼런스를 반환하는 경우
- 상속받은 비가상 멤버 함수를 재정의(override)하는 경우
- 기본 인자(default argument)를 사용하므로써 함수 호출이 모호해 질 수 있는 경우
- public 접근 권한을 가지는 멤버 데이타를 선언하는 경우
- 객체에 대한 포인터를 상속 계층의 하위에 있는 클래스의 포인터로 타입 변환하는 경우
- 객체를 값으로서 전달하는 경우
- new 연산으로 초기화된 객체에 대한 객체화된 객체 포인터(dereferenced object pointer)를 반환하는 경우
- 생성자에서 멤버 데이타의 초기화 순서와 다르게 멤버 초기화 리스트를 나열한 경우

2.3 Clean^{++}

Clean^{++}은 미리 정해진 제약 조건을 검사하는 기존의 lint 나 lint^{++}와는 달리 프로그래머가 정의한 제약을 검증기가 인식할 수 있는 제약 언어인 CCEL [11][12]로써 명세하고 검증한다. CCEL(C^{++} Constraint Expression Language)은 C^{++} 원시 코드에 대해서 프로그래머가 부과하고자하는 제약을 표현하는 언어이다. CCEL 명세는 C^{++} 원시 코드에 포함되거나, 별도의 화일에 명세된다. 먼저 C^{++} 원시코드가 분석되어져서 C^{++} DB에 저장되고, CCEL 명세는 CCEL 컴파일러에 의해서 C^{++} DB에 대한 질의로 바뀌게 된다. CCEL 처리기는 질의어를 사용하여 C^{++} DB에 대한 질의를 수행하고 정의된 형식 또는 기본적인 형태로 메세지를 출력하게 된다.

C^{++} 원시 코드에 부과하고자 하는 제약사항은 C^{++} 원시 코드에는 나타날 수 없고, 제약 사항을 명세하는 언어인 CCEL에서는 각 클래스를 단위로 하여 다음과 같은 세가지 형태의 제약을 명세할 수 있도록 한다. 첫째는 설계 결정에 의한 제약으로 설계 단계에서 어떠한 의사 결정에 따라서 결정되는 제약 사항들을 말한다. 둘째로 구현을 위한 제약으로 실제 구현을 위해서 미리 정의하는 제약 사항들을 말한다. 마지막으로 형태적(stylistic) 제약으로 명명 관례(naming convention)등과 같이 형태적인 측면의 제약 사항들을 말한다.

Clean^{++}은 설계 및 구현 단계에서 미리 정의된 제약을 C^{++} 원시 코드에 대하여 검증하는 것이 아니고, 프로그래머가 원시 코드에 대해서 부과하고자 하는 제약을 표현하고 검증할 수 있도록 하는 도구이다.

본 논문에서의 제약은 실체(entity)의 내부 또는 객체들 간의 관계를 나타내는 식(expression)이라고 할 수 있다[20]. 따라서 제약 언어는 여러가지 제약들을 효율적이고 직관적으로 표현할 수 있어야 한다[12]. 그러나 객체 지향 언어에 기반하는 제약 언어는 각 클래스나 객체들의 상태에 기반하여 각 객체들 간의 관계를 표현하므로, 상속 관계를 제외한 객체들 간의 관계는 각각의 객체 안에서 암시적으로 나타낼 수밖에 없다. 그러나, 객체 모델링에서는 각 객체들 간의 관계 자체도 객체들과 같은 중요성을 가진다[3]. 따라서 객체지향 언어에 기반하여 제약 사항을 명세하는 언어와는 다르게, 객체 모델에 대해서 제약 사항을 명세하는 언어는 제약 사항에 있어서 객체들 간의 관계 자체도 객체들처럼 명백히 명세할 수 있어야 한다. 본 논문에서 제안하는 제약 언어인 MCL은 이렇게 각 객체들 간의 관계도 객체들처럼 명백히 명세할 수 있도록 설계되었다.

3. MCL: 모델 제약 언어

3.1 MCL 개요

객체 모델에 대한 제약 사항을 명세하는데 MCL이 사용된다. 이 MCL은 일련의 제약 구성자(constraint construct)들로 구성이 되는데, 각 제약 구성자의 개괄적인 구조는 다음과 같다.

상세한 구문은 〈부록 1:MCL의 정규 표현〉에서 기술하였다.

각 제약 사항들은 이름을 가지게 되고 이름은 〈NAME〉에서 기술한다. 그리고 모든 제약 조건들이 설계자에게 똑같이 중요한 것은 아니므로, 각 제약 조건들은 중요도에 따라서 CRITICAL-ERROR, ERROR, WARNING의 세가지 수준으로 나누어서 〈LEVEL〉에서 명세하게 된다.

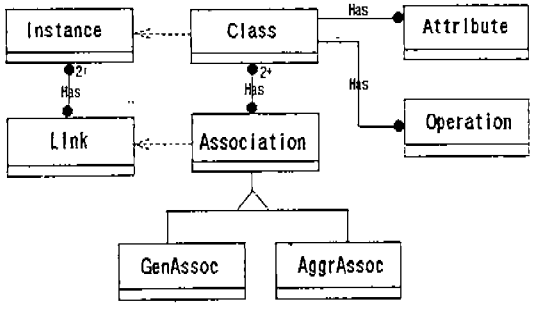
```
Constraint <NAME> : <LEVEL> {
 <VAR_DECLARATION>
assert:
 <CONSTRAINT_EXPRESSION>
message:
 <MESSAGE_FORMAT>[, <VAR_LIST>]
}
```

(그림 1) 제약 구성자의 구조
(Fig. 1) Structure of the constrint construct

〈VAR_DECLARATION〉은 제약의 명세에 참여하게 되는 변수들을 선언하는 부분이다. 각 변수들은 (그림 3)에서 보여지는 것과 같이 MCL에서 미리 정의된 타입에 의해서 선언되는데, 적용되는 성질에 따라서 모든 객체가 assert:를 만족해야 하면 FORALL 그리고 하나이상의 객체가 assert:를 만족해야 하면 FORSOME의 수식구(qualifier)가 붙여진다.

〈CONSTRAINT_EXPRESSION〉은 하나의 Bool 값을 가지는 조건식 또는 관계식인데, 이 수식의 값이 거짓이 될 때, message:이후에 정의된 메세지가 출력되게 된다.

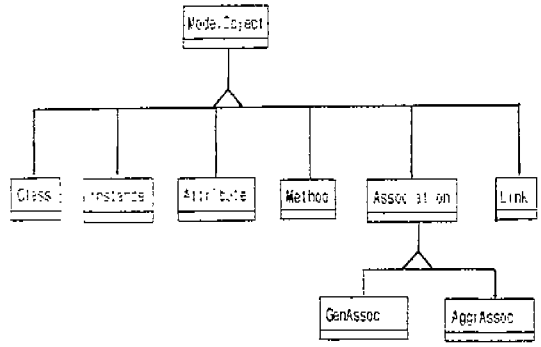
객체 모델은 다양한 구성자들로 이루어지게 되는데 객체 모델에 대한 메타 모델을 표현하면 다음의 (그림 2)와 같다.



(그림 2) 객체 모델에 대한 메타 모델
(Fig. 2) Metamodel of the object model

MCL에서 미리 정의된 타입들은 객체 모델을 표현하는 위와 같은 객체 모델에 대한 메타 모델에 따라서 다음의 (그림 3)과 같은 구조로 되어 있다.

MCL을 구성하고 있는 각 타입들은 객체 모델에



(그림 3) MCL에서 미리 정의된 타입들
(Fig. 3) Predefined types in the MCL

있어서 각기 주요한 부분들을 나타내고 있는데, 〈표 1〉은 각 타입들 및 이 타입들에 대해 적용할 수 있는 주요한 멤버 함수들을 보여 주고 있다.

ModelObject는 객체 모델에 있어서 각 구성 요소들이 공통적으로 가지는 성질을 나타내는 추상 클래스이고, Attribute는 각 클래스 안에서 속성들이 가지는 속성을 나타내는 클래스이고, Method는 각 클래스 안에서 연산들이 가지는 속성을 나타내는 클래스이고, Class는 객체 모델에서의 클래스들의 속성을 나타내는 클래스이고, Instance는 객체 모델에서 실제로 인스탄스로 나타나는 객체들의 속성을 나타내는 클래스이고, Association은 클래스들 간의 관계를 나타내는 속성을 가지는 클래스이고, Link는 결합(association)이 객체화되어서 나타내는 속성을 가지는 클래스이고, GenAssoc은 일반화 관계를 나타내는 결합의 특정한 형태를 나타내는 클래스이고, AggrAssoc은 집단화 관계를 나타내는 결합의 특정한 형태를 나타내는 클래스이다.

위의 각 클래스들은 모델 분석기가 분석을 수행하

된 객체 모델에 대한 정보를 가지고 있게 되며, 모델 검증기가 이 클래스들이 가지고 있는 내용들에 대해서 질의를 수행하고 결과를 출력하게 된다.

〈표 1〉 MCL에서 미리 정의된 타입과 연산들
〈Table 1〉 Predefined types and operations in the MCL

타입 이름	적용 가능 연산	
ModelObject	CString	name( )
	Bool	isExists( )
	Bool	inheritFrom (ModelObject&)
Attribute	ModelObject&	definedBy( )
Method	ModelObject&	definedBy( )
	Bool	isRedefined (ModelObject&)
Class	int	maxDerivedDepth( )
Instance	Bool	isInstanceOf (ModelObject&)
Association	int	numObjects( )
	int	cardinality (Class&)
	CString	firstRoleName( )
	CString	secondRoleName( )
	CString	nextRoleName( )
	ModelObject&	firstObject( )
	ModelObject&	secondObject( )
	ModelObject&	nextObject( )
	Class&	parent( )
	Class&	firstChild( )
	Class&	nextChild( )
Link	Bool	isInstanceOf (Association&)

3.2 객체 모델링 지침

객체 모델에 대해서 일반적으로 널리 받아들여지는 지침들은 많지 않지만, 다음과 같은 모델링 지침들은 객체 모델링 방법론 및 도구에 익숙한 모든 사람들이 대부분 동의하는 지침들이다.

- 두 클래스간에 두개 이상의 결합이 있을 때, 각 결합에는 반드시 이름이 있어야 한다. 각 결합의 의미를 명백히 구분하기 위해서이다.
- 3차 이상의 결합은 될 수 있으면 피하도록 한다. 이해하기 어렵기 때문이다.
- 클래스와 인스탄스는 하나의 다이아그램 안에서 같이 나타내지 않는 것이 좋다. 하나의 다이아그램 안에 클래스와 인스탄스가 같이 존재한다면 혼동하기 쉽고 다이아그램이 복잡해 지기 때문이다.
- 일반화할 때 너무 깊이 내포(nesting)하지 않도록 한다. 일반화의 깊이가 너무 깊으면 상속을 통하여 하위의 클래스의 속성과 연산을 모두 파악하기 어려워지기 때문이다.

위와 같이 널리 받아들여지는 지침들 외에도 각 개별 응용영역 안에서 별도의 지침으로 정의하고 지켜지기 원하는 제약들이 있다. 이러한 제약들은 객체 모델 검증시스템에 의해 미리 정의될 수 있는 것이 아니고, 모델 작성자들이 객체 모델에 부과하고자 하는 지침들을 별도의 형태로 표현해야 한다. 본 논문에서는 이러한 형태를 표현할 수 있는 제약 언어인 MCL을 제안하였고, 이러한 제약 언어를 사용하여 각 모델 작성자들은 객체 모델들이 따라야 하는 지침을 명세하고, 본 논문에서 제안한 객체 모델 검증기에 의해서 자동으로 검증된다. 다음 절에서는 객체 모델에 대한 지침을 MCL로 표현하는 예를 보인다.

3.3 MCL 명세 예

이 절에서는 객체 모델에 대한 지침을 MCL로 표현하는 예를 보인다. 첫번째 예로서, "두 클래스간에 두개 이상의 결합이 있을 때, 각 결합에는 반드시 이름이 있어야 한다"라는 지침을 MCL로 표현하면 다음과 같다.

```
Constraint TwoAssocWithNoName : ERROR {
    FORALL Association a1;
    FORSOME Association a2;
assert ! (( a1 != a2 ) &&
    (( a1.firstObject() == a2.firstObject() &&
    a1.secondObject() == a2.secondObject()) ||
    ( a1.firstObject() == a2.secondObject() &&
    a1.secondObject() == a2.firstObject() )) ||
    ( a1.name() != " " && a2.name() != " ");
message: "There are two associations with no name between %s
    and %s\n", a1.firstObject().name(), a1.secondObject().name();
}
```

(그림 4) 첫번째 예에 대한 MCL 표현
(Fig. 4) MCL expression of the first example

위의 제약의 이름은 TwoAssocWithNoName이고, 오류 수준은 ERROR이다. a1 변수는 현재 검증 대상이 되는 객체 모델에서의 모든 결합을 나타내는 변수

이고, a2는 임의의 어떤 결합을 나타낸다. 같지 않은 두개의 결합이 같은 두개의 객체를 연결한다면, 각각의 결합의 이름이 있어야 함을 검증한다. 만약 이 조건을 만족하지 않는 결합이 객체 모델에 존재한다면, message:이후의 형식과 변수에 의해서 메세지를 출력하게 된다.

두번째 예로서, "3차 이상의 결합은 될 수 있으면 피하도록 한다"라는 지침을 MCL로 표현하면 다음과 같다.

위의 제약의 이름은 TernaryAssociation이고, 오류 수준은 WARNING이다. 오류 수준을 WARNING으로 한 이유는 삼각 결합은 이해하기 어렵고 오류의 위험이 있기는 하지만, 불가피하게 나타나야만 할 경우가 있기 때문이다. a1은 검증의 대상이 되는 객체 모델 안에서의 모든 결합을 나타내는 변수이다.

만약 어떠한 결합이 삼각 결합이라면 위의 assert:구절은 거짓이 되며, message:구절에서 명세된 형태로 오류 메세지를 출력하게 된다.

세번째 예로서, "클래스와 인스탄스는 하나의 다이아그램 안에서 같이 나타내지 않는 것이 좋다"라는 지침을 MCL로 표현하면 다음과 같다.

위의 제약의 이름은 NoExistClassAndInstance이고, 오류 수준은 WARNING이다. 오류 수준을 WARNING으로 한 이유는 클래스와 인스탄스가 하나의 다이아그램내에 같이 존재하지 않는 것이 바람직하지만 불가피하게 나타나야만 할 경우도 있기 때문이다. c1은 검증의 대상이 되는 객체 모델 안에서의 클래스를 나타내고 i1은 인스탄스를 나타내는 변수이다.

네번째 예로서, "일반화할 때 4차 이상 내포(nesting)하지 않도록 한다"라는 지침을 MCL로 표현하면 다음과 같다.

위의 제약의 이름은 TooMuchDrivedClass이고, 오류 수준은 ERROR이다. 오류 수준을 ERROR로 한 이유는 일반화의 깊이가 너무 깊으면 상속을 통하여 하위의 클래스의 속성과 연산을 모두 파악하기 어려워지기 때문에 적용 범위에 맞게 깊이의 제약을 두어야 한다. (그림 7)의 예에서는 깊이를 4로 제한하였으나 적용 영역에 따라서 조정이 가능하다. c1은 검증의 대상이 되는 객체 모델 안에서의 클래스를 나타내는 변수이다.

```
Constraint TernaryAssociation : WARNING {
    FORALL Association a1;
assert: a1.numObjects() != 3;
message: "There are ternary association among %s and %s and %s\n",
          a1.firstObject().name(), a1.secondObject().name(),
          a1.nextObject().name();
}
```

(그림 5) 두번째 예에 대한 MCL 표현
(Fig. 5) MCL expression of the second example

```
Constraint NoExistClassAndInstance : WARNING {
    FORSOME Class c1;
    FORSOME Instance i1;
assert: ! ( c1.isExists() && i1.isExists() );
message: "There are both class %s and instance %s \n",
          c1.name(), i1.name();
}
```

(그림 6) 세번째 예에 대한 MCL 표현
(Fig. 6) MCL expression of the third example

```
Constraint TooMuchDrivedClass : ERROR {
    FORALL Class c1;
assert:  c1.maxDerivedDepth() <= 4;
message: "There are too much derived class %s \n", c1.name();
}
```

(그림 7) 네번째 예에 대한 MCL 표현
(Fig. 7) MCL expression of the fourth example

4. 객체 모델 검증시스템

기존의 객체 모델링 도구들은 모델 작성자가 임의의 형태로 모델을 작성하는 도구들이었다. 이러한 도구들은 객체 모델이 개발 단계에서의 결정 사항들을 따르는지 검증하지 못하고, 단지 객체 모델 내에서의 일관성을 검증할 수 있을 뿐이었다.

본 논문에서 제안한 객체 모델 검증시스템은 모델을 작성하는 도구뿐만 아니라, 작성되어진 객체 모델이 분석 단계에서의 결정 사항이나 지침들을 따르는지 검증하는 도구로 구성이 된다.

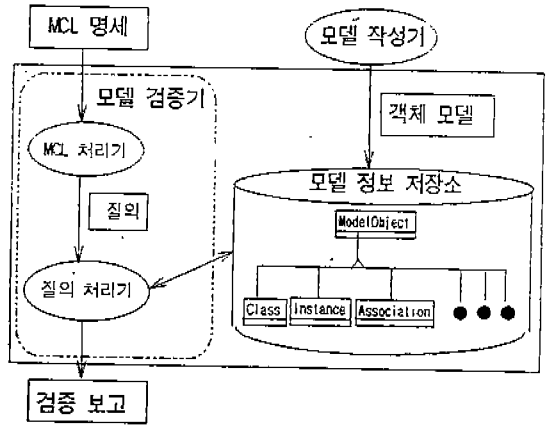
본 장에서는 모델링 지침을 MCL로 명세하고, 이러한 지침을 바탕으로 작성되어진 객체 모델에 대하여 검증하는 과정을 기술한다.

4.1 시스템의 구성

본 논문에서 제안한 객체 모델 검증시스템의 구조는 다음 (그림 8)과 같다.

본 시스템은 크게 모델 작성기와 모델 검증기, 정보 저장소로 구성된다. 모델 작성기는 모델 작성자와 대화적으로 객체 모델을 작성하는 도구이다. 모델 검증기는 MCL 언어로 명세된 제약을 객체 모델에 대해서 검증하는 도구로서, 객체 모델에 대하여 분석 작업을 수행하는 부분과 이 분석을 통하여 얻어진 정보에 대해서 MCL 명세에 해당하는 질의를 수행하여 결과를 분석하는 부분으로 구성이 된다. 모델 정보 저장소는 (그림 3)의 구조와 같은 형태로 객체 모델에 대한 정보를 저장하고 〈표 1〉에 정의된 연산에 따라서 정보를 제공하는 부분이다.

여기서 객체 모델 작성기는 특정의 객체 모델링 방법을 지원하는 객체 모델 작성 도구이다. 본 논문에서는 객체지향 방법론중 가장 널리 사용되는 OMT



(그림 8) 객체 모델 검증시스템의 구조
(Fig. 8) Architecture of the Object Model Verification System

방법론을 지원하는 도구를 사용하였다. 도구는 사용자와 대화적으로 객체 다이아그램을 작성한 후, 객체 모델 분석기에 의해서 처리가 가능한 아스키 코드 형태로 모델의 내용을 변환하여 저장한다. 객체 모델의 명세를 위한 구문은 〈부록 2:OMT 객체 모델의 정규 표현〉에 기술하였다.

4.2 객체 모델의 논리적 표현

OMT 방법에 의해 작성된 객체 모델을 모델 검증기가 처리하기 위하여 논리적으로 표현한 형태는 (그림 7)과 같다. 밑줄친 부분은 예약어이며, 상세한 구문은 부록에서 기술하였다.

OMT 방법에 의해 작성된 객체 모델의 논리적 표현은 (그림 2)의 객체 모델에 대한 메타 모델에 따라서 정의되었다. 이 구문에 의해서 표현된 검증의 대상이 되는 객체 모델은 객체 모델 검증기의 입력으로 사용되게 된다.

```
Class name
{
    attribute: name, ...;
    operation: name, ...
    association: name( name ), ...;
    instance: name, ...;
}
Instance name
{
    class: name;
    attribute: name( name ), ...
}
Association name
{
    attribute: name, ...;
    class: name( role: name, constraint: name, cardinal: name,
                 qualification: name ),
           name( role: name, constraint: name, cardinal: name,
                 qualification: name ) ;
}
Link name
{
    association: name;
    attribute: name( name ), ...;
    link: name, name;
}
GenAssoc name
{
    parent: name;
    child: name( role: name, constraint: name, cardinal: name,
                 qualification: name ) ...;
}
AggrAssoc name
{
    parent: name;
    child: name( role: name, constraint: name, cardinal: name,
                 qualification: name ) ...;
}
```

(그림 9) OMT 객체 모델의 논리적 표현
(Fig. 9) Logical expression of the OMT Object Model

4.3 시스템의 실행 예

본 절에서는 3.3절에서 작성한 MCL 명세를 이용하여, 객체 모델 작성기에 의해서 객체 모델을 작성하고, 작성되어진 객체 모델을 검증하는 예를 보인다.

4.3.1 시스템의 실행 예(1) : TwoAssocWithNoName

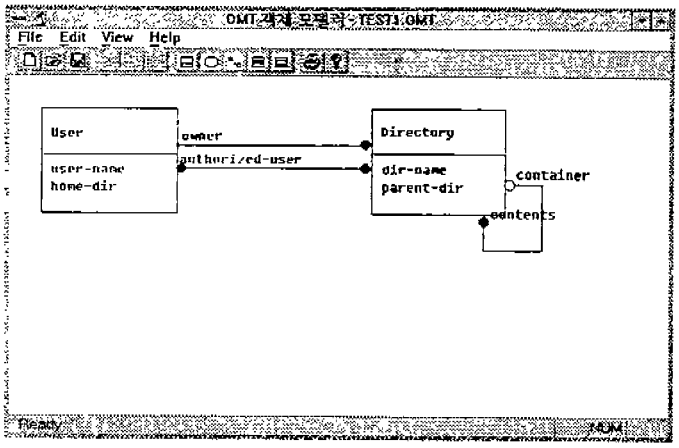
첫번째 실행 예로서, "두 클래스간에 두개 이상의 결합(association)이 있을 때, 각 결합에는 반드시 이름이 있어야 한다"는 지침으로 이것을 MCL로 표현하면 3.3절의 (그림 4)와 같다. 이것을 먼저 MCL 처리기를 통하여 C++ 프로그램으로 변환하고, 이것을 컴파일하여 실행가능한 프로그램을 만들면 객체 모델 검증기가 만들어진다. 다음 (그림 10)에서는 객체 모델 작성기가 객체 모델을 만든 예를 보이고 있다.

(그림 10)에서 작성한 객체 모델의 논리적 표현은 다음 (그림 11)과 같다.

여기서 작성한 객체 다이아그램은 두 클래스간에 하나 이상의 결합이 있고, 각 결합에 이름이 없으며 단지 역할 이름(role name)만 있는 다이아그램으로서 위에서 명세한 제약에 위배된다. 이 다이아그램의 검증을 수행한 결과는 (그림 12)와 같다.

4.3.2 시스템의 실행 예(2) : TernaryAssociation

두번째 예로서, "가급적이면 삼각 결합은 삼가하도록 한다"는 지침을 명세하고 이 지침을 검증하는 예를 보인다. 먼저 이 제약을 MCL로 명세한 예는 3. 3

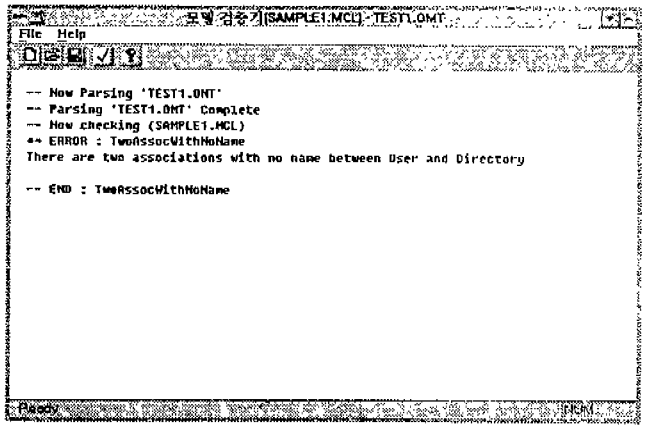


(그림 10) 첫번째 예에 대한 객체 모델
(Fig. 10) Object Model of the first example

```
Class User
{
        attribute: user-name, home-dir;
        operation: ;
        association: (Directory) ;
        instance: ;
}
Class Directory
{
        attribute: dir-name, parent-dir;
        operation: ;
        association: (User) ;
        instance: ;
}
Association
{
        attribute: ;
        class:  User( role: owner,),
                Directory( cardinal: CARDINAL_MANY, );
}
Association
{
      attribute: ;
      class: User( role: authorized_user, cardinal: CARDINAL_MANY, ),
             Directory( cardinal: CARDINAL_MANY, );
}
Association
{
      attribute: ;
      class: Directory( role: contents, cardinal: CARDINAL_MANY, ),
             Directory( role: container, cardinal: CARDINAL_OPT, );
}
```

(그림 11) 첫번째 예에 대한 객체 모델의 논리적 표현

(Fig. 11) Logical expression of the Object Model in the first example



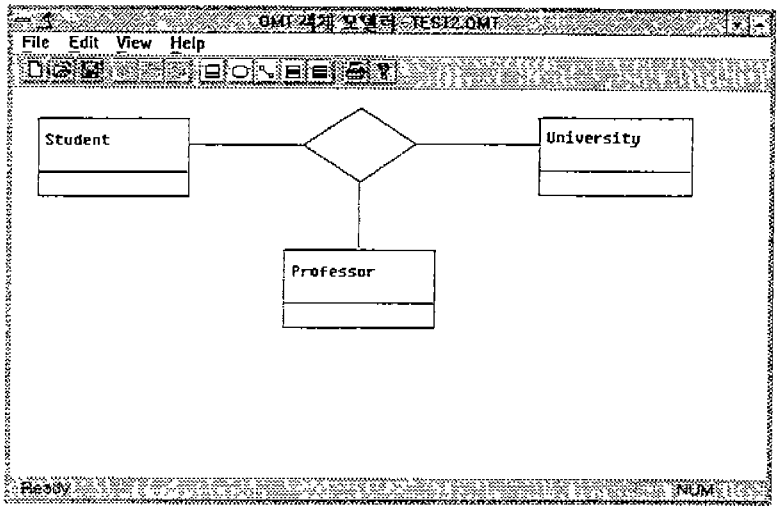
(그림 12) 첫번째 예에 대한 객체 모델의 검증 결과

(Fig. 12) Verified Result of the Object Model in the first example

절의 (그림 5)와 같고 작성된 객체 모델은 (그림 13)과 같다.

(그림 13)의 객체 모델에 대한 논리적인 표현은 다음 (그림 14)와 같다.

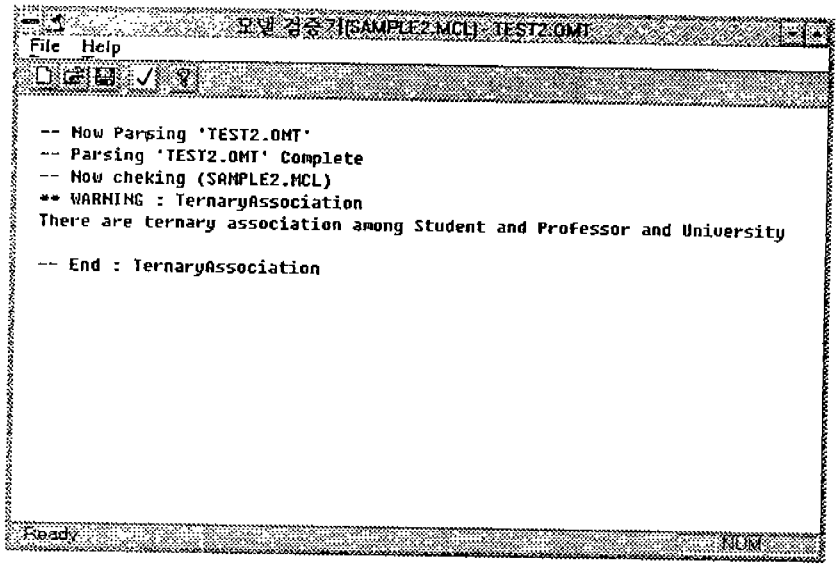
여기서 작성된 객체 모델을 검증한 결과는 다음 (그림 15)와 같다.



(그림 13) 두번째 예에 대한 객체 모델
(Fig. 13) Object Model of the second example

```
Class Student
{
        attribute: ;
        operation: ;
        association: (Professor), (University) ;
        instance: ;
}
Class Professor
{
        attribute: ;
        operation: ;
        association: (Student), (University) ;
        instance: ;
}
Class University
{
        attribute: ;
        operation: ;
        association: (Student), (Professor) ;
        instance: ;
}
Association
{
        attribute: ;
        class:  Student( ),
                Professor( ),
                University( ) ;
}
```

(그림 14) 두번째 예에 대한 객체 모델의 논리적 표현
(Fig. 14) Logical expression of the Object Model in the second example



(그림 15) 두번째 예에 대한 객체 모델의 검증 결과
(Fig. 15) Verified Result of the Object Model in the second example

4.4 시스템의 비교평가

본 논문에서 제안한 객체 모델 검증시스템을 다음과 같은 항목에 의해서 평가한다.

첫째, 검증시스템의 효율성 및 비용절감에 대한 평가이다. 기존의 검증 도구들은 주로 구현단계에서 코드에 대한 검증을 수행한다. 그러나 검증은 가능한한 개발의 초기단계에서 수행해야 상당한 효과를 가져올 수 있다. 객체 모델 검증시스템은 소프트웨어 개발 초기에서부터 오류를 검증할 수 있으므로 소프트웨어 개발자에게 상당한 효율성과 유비보수 비용을 감소시켜 준다.

둘째, 검증대상의 일관성 및 품질 보증에 대한 평가이다. 기존의 객체 모델링 도구들은 여러 개발자의 주관적인 지침에 의해 객체 모델이 작성되기 때문에 일관된 지침이나 개발과정의 결정사항이 정확하게 반영되는지를 검증할 방법이 없다. 그러나 객체 모델 검증시스템은 일관된 지침 및 결정사항을 통하여 작성된 객체 모델들을 검증하므로 객체 모델에 대한 일관성 및 품질을 보장할 수 있다.

셋째, 검증시스템의 융통성 및 확장성에 대한 평가이다. 기존의 검증 도구들은 주로 정해진 지침만을 검증하지만 객체 모델 검증시스템은 개발자가 검증하려고 하는 지침 및 결정사항들을 언제나 추가하여 검증할 수 있고, 검증 대상도 다른 모델로 확장하기가 용이하다.

5. 결 론

소프트웨어가 크고 복잡해짐에 따라서 소프트웨어 개발 단계에 있어서 분석 단계의 역할이 더욱 중요하게 되었다. 특히 기존의 기능중심 모델링이나 자료중심 모델링의 단점을 보완하기 위해 등장한 객체 모델링에서는 분석 단계를 더욱 강조하고 있다. 그러나 기존의 모델링 도구들은 작성되어진 객체 모델이 일정한 모델링 지침을 만족하는지 검증할 수 있는 방법이 없었다. 결국 객체 모델은 모델 작성자의 주관적인 방법으로 작성되기 때문에 생성되는 객체 모델마다 일관성 및 품질을 보장할 수 없게 되었다. 따라서 작성된 객체 모델을 검증할 수 있는 검증 도구가 필요하게 되었다. 기존의 검증 도구들은 주로 구현 단계에서 프로그램 원시 코드에 대해서 일정한 지침을 검증하였으나 본 논문에서는 이러한 검증 도구의 원리를 개발의 초기 단계인 분석 단계에 적용해 보았다.

본 논문에서는 기존의 객체 모델링 방법론에서 제

시하였던 지침들뿐만 아니라 시스템의 개발자가 객체 모델에 대해서 규정하고자하는 지침들까지도 객체 모델에 대한 제약으로서 표현할 수 있는 언어를 정의하고, 이 언어를 적용하여 객체 모델의 일관성 향상을 이룰 수 있는 검증시스템을 구축하였다.

객체 모델 검증시스템의 장점은 다음과 같다. 첫째, 검증 단계를 기존의 구현 단계에서 분석 단계로 가져옴으로써 설계상의 오류 수정의 비용을 대폭 줄일 수 있다. 둘째, 개발자가 객체 모델에 대해서 부여하려는 모든 지침들을 적용할 수 있고 언제든지 변경할 수 있다. 따라서 다양한 방법론들을 적용할 수 있다. 셋째, 생성되는 모든 객체 모델에 대하여 품질 및 일관성을 보장할 수 있다. 따라서 다수의 개발자가 서로 간의 의사소통 없이도 일관된 객체 모델을 작성할 수 있다. 이 시스템의 단점으로는 개발자 스스로가 지침에 대한 제약 언어를 작성해야 하므로 상당한 노력이 필요하게 된다. 또한 현재 검증시스템의 객체 모델 작성기는 OMT 방법론만을 지원하여 객체 모델을 작성하고 검증기가 처리할 수 있는 형태로 명세를 생성하고 있으며, 동적 모델 및 기능 모델의 작성은 지원하고 있지 않다. 객체 모델 검증기는 주어진 제약 명세에 대해서 객체 모델을 검증하는 실행 프로그램을 생성하고 있는데, 실행 속도면에는 인터프리팅이나 재 컴파일을 요구하지 않으므로 빠르다. 그러나 객체 모델을 작성하는 과정에서 대화적으로 검증하는 기능은 없다.

향후 연구과제로는 개발자가 명세한 언어를 OMT 방법론 외에 여러 방법론에 대해서 대화적으로 검증하고, 메세지를 출력하는 환경으로 발전한다면 더욱 효율적인 검증시스템이 될 것이다. 또한 이렇게 기술된 제약 명세들은 앞으로 통합 소프트웨어 개발 환경에서 모델링의 자동화를 위한 자료가 될 것이다.

참 고 문 헌

[1] 이경환 외, *소프트웨어 공학*, 청문각, 1993.

[2] 이경환, *소프트웨어 재사용을 위한 객체 모델링 기법*, 교학사, 1993.

[3] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, Object-Oriented *Modeling and Design*, *Prentice-Hall*, 1991.

[4] Bjarne Stroustrup, *The C++ Programming Language*, 2nd Ed., Addison-Wesley, 1991.

[5] Carma McClure, *CASE is Software Automation*, Prentice-Hall, 1989.

[6] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener, *Designing Object-Oriented Software*, Prentice-Hall, 1990.

[7] Grady Booch, *Object Oriented Design with Applications*, Benjamin/Cummings, 1991.

[8] Ivar Jacobson, *Object-Oriented Software Engineering : A Use Case Driven Approach*, Addison-Wesley, 1992.

[9] David W. Embley, Barry D. Kurtz, Scott N. Woodfield, Object-Oriented Systems Analysis : A Model-Driven Approach, Yourdon Press, 1992.

[10] Debra L. Hudson, *Practical Model Management Using CASE Tools*, QED Publishing, 1993.

[11] Scott Meyers, Moises Lejter, "Automatic Detection of C++ Programming Errors : Initial Thoughts on a lint++," *Proceedings of USENIX C++ Conference*, pp. 29-40, Apr. 1991.

[12] Carolyn K. Duby, Scott Meyers, Steven P. Reiss, "CCEL : A Metalanguage for C++," Proceedings of USENIX C++ Conference, Aug. 1992.

[13] Yeuh-hong Lin, Scott Meyers, "CCEL : The C++ Constraint Expression Language, An Annotated Reference Manual, Ver. 0.5," Brown Univ. Technical Report CS-93-23, May 1993.

[14] Fiona Hayes, Derek Coleman, "Coherent Models for Object-Oriented Anaylsis," *Proceedings of OOPSLA '91*, pp. 171-183, 1991.

[15] Peter Wegner, "Dimensions of Object-Oriented Modeling," *IEEE Computer*, Vol. 25, No. 10, pp. 12-20, Oct. 1992.

[16] Sally Shlaer, Stephen J. Mellor, Wayne Hywari, "OODLE : a Language-Independent Notation for Object-Oriented Design," *JOOP Focus on Analysis and Design*, pp. 98-106, 1992.

[17] Jean Pierre Lejacq, "Semantic-Based Design Guidelines for Object-Oriented Programs," *JOOP*

Focus on Analysis and Design, pp. 86-97, 1992.

[18] James J. Odell, "Object-Oriented Analysis and Design," *JOOP Focus on Analysis and Design*, pp. 74-84, 1992.

[19] Premkumar T. Devanbu, "GENOA-A Customizable, Language-and Front-End Independent Code Analyzer," Proceedings of 14th ICSE, pp. 307-317, 1992.

[20] Nabil M. Zamel, Timothy A. Budd, "Integrating Constraints into a Multiparadigm Language," Proceedings of InfoScience '93, pp. 402-409, 1993.

부록 1 : MCL의 정규 표현

MCL의 문법을 BNF로 표현하면 다음과 같다.

```
<mcl_spec> ::= <constraint_spec>|<mcl_spec><constraint_spec>
<constraint_spec> ::= <constraint_part><brace><var_decl_list>
                      <assert_part><message_part><brace>
<constraint_part> ::= Constraint <user_def_name><colon><err_level>
<user_def_name> ::= <identifier>
<identifier> ::= <letter>{<letter>|<digit>}
<colon> ::= :
<err_level> ::= CRITICAL-ERROR | ERROR | WARNING
<brace> ::= { | }
<var_decl_list> ::= <var_decl>|<var_decl_list><var_decl>
<var_decl> ::= <qualifier><type_name><var_name_list>
<qualifier> ::= FORALL | FORSOME
<type_name> ::= ModelObject | Attribute | Method | Class | Instance | Association | Link
<var_name_list> ::= <var_name>{<comma><var_name>}
<var_name> ::= <identifier>
<comma> ::= ,
<assert_part> ::= assert<colon><constraint_expression><semi>
<constraint_expression> ::= <composite_expression>
<composite_expression> ::= <simple_expression>|<not_operator><simple_expression>|
                           <composite_expression><logical_operator><composite_expression>
<simple_expression> ::= <term><relational_operator><term>|
                        <term><logical_operator><term>
<term> ::= <method>|<string>|<digit>
<method> ::= <var_name><point><method_name>{<point><method_name>}
<method_name> ::= <identifier><paren><paren>
<string> ::= <character>{<character>}
<point> ::= .
<paren> ::= ( | )
<not_operator> ::= !
<logical_operator> ::= && | ||
<relational_operator> ::= == | != | > | >= | < | <=
<semi> ::= ;
<message_part> ::= message<colon><message_content><comma><method_list><semi>
<message_content> ::= "<string>"
<method_list> ::= <method>{<comma><method>}
```

부록 2 : OMT 객체 모델의 정규 표현

OMT 방법의 객체 모델의 논리적 측면을 정형화하여 BNF로 표현하면 다음과 같다.

<omt_spec> ::= <object_spec>|<omt_spec><object_spec>
<object_spec> ::= <class_spec>|<instance_spec>|<assoc_spec>|<link_spec>|
<genassoc_spec>|<aggrassoc_spec>
<class_spec> ::= **Class** <user_def_name><brace><class_body><brace>
<class_body> ::= <attribute_name_list><operation_name_list>
<association_name_list><instance_name_list>
<instance_spec> ::= **Instance** <user_def_name><brace><instance_body><brace>
<instance_body> ::= <class_name_list><attribute_name_list>
<assoc_spec> ::= **Association** <user_def_name><brace><assoc_body><brace>
<assoc_body> ::= <attribute_name_list><class_name_list>
<link_spec> ::= **Link** <user_def_name><brace><link_body><brace>
<link_body> ::= <association_name_list><attribute_name_list><link_name_list>
<genassoc_spec> ::= **GenAssoc** <user_def_name><brace><genassoc_body><brace>
<genassoc_body> ::= <parent_name_list><child_name_list>
<aggrassoc_spec> ::= **AggrAssoc** <user_def_name><brace><aggrassoc_body><brace>
<aggrassoc_body> ::= <parent_name_list><child_name_list>

<attribute_name_list> ::= **attribute**<colon><name_list><semi>
<operation_name_list> ::= **operation**<colon><name_list><semi>
<association_name_list> ::= **association**<colon><name_list><semi>
<instance_name_list> ::= **instance**<colon><name_list><semi>
<class_name_list> ::= **class**<colon><name_list><semi>
<link_name_list> ::= **link**<colon><name_list><semi>
<parent_name_list> ::= **parent**<colon><name_list><semi>
<child_name_list> ::= **child**<colon><name_list><semi>

<name_list> ::= <empty>|<name>{<comma><name>}
<name> ::= <identifier>|<identifier><paren><class_decl><paren>
<class_decl> ::= <empty>|<content_list>{<comma><content_list>}
<content_list> ::= <keyword><colon><name>
<keyword> ::= **role** | **constraint** | **cardinal** | **qualification**

김 진 수

1986년 중앙대학교 공과대학 전자계산학과 졸업(이학사)
1988년 중앙대학교 대학원 전자계산학과 졸업(이학석사)
1996년 중앙대학교 대학원 컴퓨터공학과(박사과정수료)
1995년~현재 중앙대학교 기술과학연구소 연구원
관심분야:소프트웨어 공학, 객체지향 방법론, 정형화 기법, 하이퍼미디어 시스템

강 권 학

1993년 중앙대학교 공과대학 컴퓨터공학과 졸업(공학사)
1995년 중앙대학교 대학원 컴퓨터공학과 졸업(공학석사)
1995년 1월~현재 국방정보체계연구소 전임연구원
관심분야:시스템 보안, 소프트웨어 공학, 멀티미디어 시스템

이 경 환

1980년 중앙대학교 대학원 응용수학 전공(이학박사)
1982년~1983년 미국 Aurban대학 객원교수
1986년 서독 Bonn대학 객원 교수
1971년~현재 중앙대학교 컴퓨터공학과 교수
관심분야:소프트웨어 공학, 객체지향 모델링, 소프트웨어 재사용.