UML 다이어그램의 정확성 검증을 위한 메타모델과 OCL로 명세한 검증규칙

하 일 규† · 강 병 욱††

요 약

다이어그램의 일관성이란 하나의 요구사항으로부터 설계된 여러가지 UML 다이어그램이 통일된 의미로 작성되었는가를 나타내는 성질이고, 정확성은 사용자가 작성한 다이어그램이 UML 표준에 적합하게 작성이 되었는가를 나타내는 성질이다. 본 연구에서는 UML(Unified Modeling Language) 버전 1.4 표준에 의해 작성된 객체지향 다이어그램의 일관성과 정확성을 검증하는 방법으로서 UML 표준의 모델제약언어로 사용되는 OCL(Object Constraint Language)을 이용하여 검증하는 방법을 제시한다. 검증의 초기작업으로서 구성요소와 관계로 표현된 각 다이어그램의 메타모델을 유도하고, 메타모델을 통해 정확성 및 일관성 검증규칙을 유도한다. 유도된 검증규칙은 명확화와 자동화를 위하여 특징적으로 OCL을 사용하여 정형적으로 명세한다. 마지막으로 명세된 규칙은 USE 도구를 이용하여 그 유용성을 검증한다.

Metamodels and Verification Rules for Verifying the Correctness of UML Diagrams

Il kyu Ha† · Byung wook Kang††

ABSTRACT

The consistency of UML diagrams is a nature for checking whether diagrams are coherently designed with only one requirements, and the correctness is a nature for checking whether user diagrams are designed according to UML standard. In this paper we propose a verification model that verifies the correctness of UML Diagrams, especially it uses OCL (Object Constraint Language) which is standard constraint language in UML. Firstly we devise metamodels that are described with components and relationships, then we derive verification rules from each metamodels for verifying correctness and consistency, and then we formally specify the rules with OCL for automatic verification. Finally we verify the rules with USE TOOL.

키워드 : UML, OCL, Consistency, Correctness

1. 서 론

기존의 객체지향 소프트웨어 개발 기법들을 통합하여 객체지향 모델링 언어의 통합표준으로 제안되고 있는 UML(Unified Modeling Language)[3, 4]을 이용한 객체지향 개발방법이 급속하게 확산되고 있다. UML은 시스템 개발 대상을 관점(view)에 따라 상세하게 표현할 수 있다는 장점을 가지기는 하지만, 반면에 UML로 작성된 다이어그램에 대한 정확함은 보장되지 않는다는 문제점을 가지고 있다. 따라서 초기 모델링 단계에서부터 모델에 대한 정확함의 검증을 통해 오류를 최소화하는 것이 중요하다. 모델의 정확함은 세 가지로 구분할 수 있다. 즉, 완전성(completeness), 일관성(consistency), 정확성(correctness)이다. 완전성은 사용자 요구사항의 분석에 따라 당초에 목표한 기능을 제대로 수행할 수 있도록 UML 다이어그램이 설계되었는지 즉 사용자 요구사항을 제대로 반영하였는지를 파악하는 성질이다. 일관성(consistency)은 하나의 사용자 요구 사항으로부터 유도된 9가지 UML 다이어그램이 형태는 비록 다르더라도 그 의미하는 바는 같은지를 파악하는 성질이다. 또 정확성(correctness)은 사용자 요구 사항을 기반으로 작성된 UML 다이어그램이 UML의 표준에 적합하게 작성이 되었는지를 파악하는 성질이다.

일관성과 정확성을 검증하기 위해서는 각 다이어그램의 표준모델과 다이어그램간의 관계를 파악할 필요가 있다. 즉 다이어그램간의 관계와 다이어그램 자체의 표준모델을 적절하게 표현하여 주는 메타모델(Metamodel)을 구성하는 작업이 필요하다. 메타모델은 UML을 구성하는 요소들과 그들간의 관계를 표현하여 모델링을 하는데 도움을 주는 표준모델과 같은 것으로 일반적으로 클래스 다이어그램 표기

† 정 회 원 : 영남대학교 대학원 컴퓨터공학과
†† 종신회원 : 영남대학교 전자정보공학부 교수
논문접수 : 2003년 1월 27일, 심사완료 : 2003년 8월 18일

법을 이용하여 작성된다. 메타모델은 또한 각 다이어그램별로 작성될 수 있으며 다이어그램간의 관계를 표현할 수도 있다. 따라서 사용자가 작성한 다이어그램은 메타모델의 하나의 인스턴스가 된다. 각 다이어그램 별로 구성된 메타모델을 기초로 일관성과 정확성을 검증하기 위한 검증규칙을 유도한다. 유도된 규칙은 텍스트 형태로 우선 정의되며 이를 자동화 검증도구에 이용하기 위하여 정형적인 명세로 표현된다. 정형적 명세의 한 방법으로서 본 연구에서는 UML specification[3]의 일부분인 OCL(Object Constraint Language)[1]을 사용한다. OCL로 표현된 규칙을 기존의 OCL 처리 도구를 이용하여 검증한다.

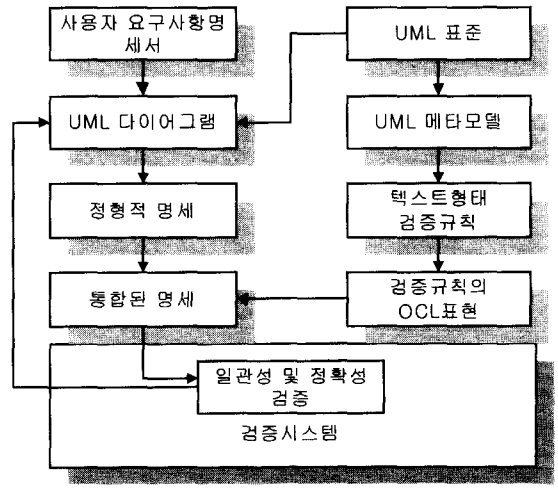
본 논문의 구성은 2장에서는 일관성 및 정확성 검증과 관련한 기존 연구에 관하여 조사 분석하고 본 연구에서 제시하고 있는 OCL을 이용한 검증방법의 전체구조에 대하여 설명한다. 3장에서는 검증의 기초가 되는 각 다이어그램별 메타모델을 유도하고 4장에서는 유도한 메타모델을 기초로 검증규칙을 유도한다. 5장에서는 검증규칙을 OCL을 이용하여 정형적으로 명세하고 6장에서는 검증 도구를 이용하여 검증하며 마지막으로 7장에서는 결론과 향후 연구 방향을 제시한다.

2. 일관성 및 정확성 검증 관련 연구

UML 다이어그램의 일관성과 정확성 검증방법에 관련된 연구로는 시나리오기반의 검증방법[17], 규칙기반의 검증방법[18, 19], 그래프를 이용한 검증방법[10-12], 대수적 명세기법을 이용한 검증방법[15], 체약언어기반의 검증방법[7-9, 14, 16]등이 있다.

시나리오 기반의 검증방법은 UML 분석모델로부터 쓰임새(use case) 정형명세구조를 만들어내면서 거기에 포함된 클래스 정보와 시나리오 정보를 이용하여 테이블을 만들고 각 클래스가 시나리오에 포함되었는지의 여부를 조사하여 일관성을 검증하는 방법이다. 이 방법은 객체의 정적구조를 표현하는 클래스의 시나리오 포함 여부라는 단지 하나의 규칙에 의해서만 일관성을 검증하므로 완벽하지 못하다. 또한 구체적인 UML 다이어그램의 비교방법은 제시되어 있지 않다. 규칙기반의 검증방법은 UML 표준으로부터 각 다이어그램의 메타모델을 유도하고 메타모델을 통해 일관성 및 정확성 검증규칙을 유도하고 있으며 규칙의 표현 방법에 있어서 서로 상이하다. 이러한 연구는 다이어그램의 명세언어로서 prolog와 같은 언어를 이용하고 있다. 따라서 이러한 명세언어는 각 다이어그램의 구성 요소와 그 관계를 표현하기에 부족한 면이 있다. 그래프를 이용한 검증방법은 클래스 다이어그램과 순차 다이어그램을 Attributed Typed Graphs로 변형하고 그들간의 일관성을 graph grammar를 이용하여 검증하고 있다. 대수적 명세기법을 이용한 검증방법은 클래스 다이어그램을 Larch Prover를 이용하여 검증하고 있다.

제약언어를 이용한 검증방법은 객체지향모델이 정확하게 작성되어야함을 나타내는 제한조건을 사용하여 모델의 정확함을 검증하는 방법이다. 이 방법은 객체모델을 고유한 명세로 표현하고, 일관성 또는 완전성과 같은 제약조건을 특정한 제약언어로 표현한다. 제약조건은 검증 시스템의 검증 알고리즘으로 사용하고, 명세된 객체모델을 입력으로 하여 검증하는 방법을 취한다. [16]에서는 제약언어로서 MCL(Model Constraint Language)을 이용하고, [7-9, 14]에서는 OCL을 이용하고 있으나 다이어그램별 일관성과 정확성의 검증방법은 제시하지 않고 있다. 제약언어를 이용한 검증방법은 객체모델을 고유한 명세로 표현하기 위한 표현규칙이 요구되며, 검증규칙을 표현하기 위한 제약언어가 요구된다. 본 연구는 규칙기반의 검증방법과 제약언어를 이용한 검증방법을 혼용한 방법으로서 특징적으로 UML의 표준제약언어로 사용되고 있는 OCL을 이용하여 다이어그램의 일관성 및 정확성을 검증하고자 하는 것으로 그 기초작업으로서 UML 명세로부터 메타모델을 유도하고 이를 바탕으로 검증규칙을 유도하고 유도된 검증규칙을 OCL을 이용하여 표현한다. OCL을 이용한 검증방법의 전체구조는 (그림 1)과 같이 표현할 수 있다.



(그림 1) OCL을 이용한 검증방법의 전체구조

OCL을 모델의 검증에 사용함으로써 다음과 같은 효과를 얻을 수 있다. 첫째, UML 표준에 맞게 고안된 언어를 사용함으로써 UML 구성요소와 제한조건을 포함하는 검증규칙을 명세하는 데 보다 적합하다는 이점을 가진다. 둘째, 기존의 복잡한 정형명세 방법에 비하여 프로그래밍언어 형태로 구성되어 있으므로 검증규칙을 명세하기가 용이하고 이해하기 쉽다. 셋째, OCL은 정형적인 언어로서 문법이 정의되어 있으므로 검증을 위한 자동화가 용이하다는 이점을 가진다.

3. 메타모델의 유도

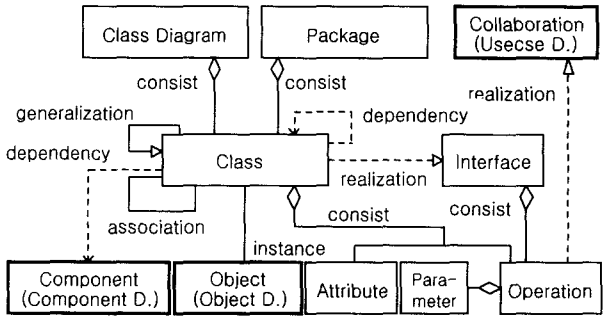
UML Semantics[2]에는 Top-Level에서 Behavioral Ele-

ments, Model Management, Foundation 패키지(package)로 UML 구성요소를 나누고, 각 패키지를 세부 패키지로 다시 나누어서 메타모델을 제시하고 있다. 본 연구에서 다루게 되는 Foundation 패키지에는 Core, Extension Mechanisms, Data Types와 같은 세부 패키지가 있으며 각 세부 패키지별로 메타모델이 제시되어 있다. 그러나 여기서 제시하는 메타모델은 UML 구성요소의 쓰임새와 구성요소간 관계의 정확한 의미를 전달하는데는 적합하나 사용자가 실제 작성하게 되는 다이어그램별 메타모델과는 거리가 있다. 즉 사용자가 실제 다이어그램을 그릴 때 사용하는 클래스 다이어그램 등 9가지 다이어그램에 대해서는 UML 표준이 메타모델을 제시하지 않고 있다는 것이다. 따라서 사용자가 UML 표준을 이용하여 다이어그램을 설계할 때 UML 구성요소간의 관계를 파악하기 위해 UML Semantics의 패키지별 메타모델을 확인해보는 작업은 상당히 불편한 작업이다. 그러므로 모델링 작업시에 구성요소간 관계와 의미를 쉽게 파악할 수 있는 다이어그램별 메타모델이 필요하다.

메타모델을 설계할 때는 각 다이어그램을 구성하고 있는 구성요소와 구성요소간 관계를 고려하여 설계하여야 하고, 일관성 검증을 위한 대응요소를 도출하기 위하여 각 다이어그램과 연관관계를 가지는 다이어그램을 파악하여 그 관계요소를 표현할 필요가 있다. 클래스 다이어그램과 관계를 가지는 주요 다이어그램의 메타모델은 다음과 같이 설계할 수 있다.

3.1 클래스 다이어그램(Class Diagram)

클래스 다이어그램은 시스템 내 객체 타입과 그들 사이에 존재하는 여러 가지 정적인 관계를 설명한다[4]. 클래스 다이어그램은 클래스를 중심으로 일반화(generalization), 의존(dependency), 연관(association) 관계를 가진다. 클래스의 인스턴스(instance)는 객체(object)가 되며 클래스를 실체화한 것이 인터페이스(interface)가 된다. 클래스 다이어그램의 메타모델은 (그림 2)와 같다.



(그림 2) 클래스 다이어그램의 메타모델

클래스 다이어그램의 클래스는 객체, 컴포넌트(component) 등과 대응관계를 가지고 오퍼레이션(operation)은 쓰임새(use case)와 대응관계를 가진다. 따라서 위의 메타모델을 기반으로 대응 다이어그램과의 일관성 검증요소를 도출해 보면 <표 1>과 같다.

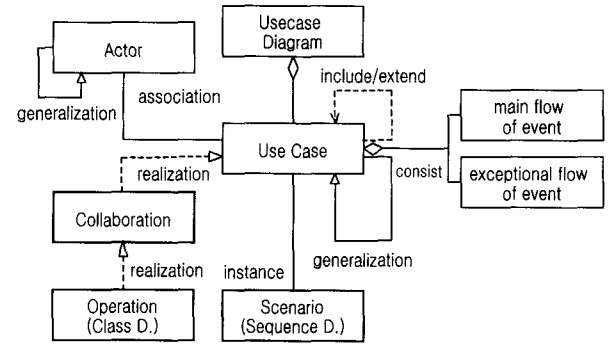
〈표 1〉 클래스 다이어그램의 일관성 검증요소

검증요소	관계(요소)	대응요소	대응다이어그램
Class	Instance	Object	Object D.
Class	Realization(Interface)	Component	Component D.
Class	Dependency	Component	Component D.
Operation	Collaboration	Use Case	Use case D.

이와 같은 메타모델과 대응요소는 정확성과 일관성 검증을 위한 초기 작업으로서의 중요한 의미를 가진다. 즉 정확성 검증은 메타모델을 구성하고 있는 구성요소의 포함 여부를 조사하고 구성요소간의 관계가 적합한지를 조사함으로써 이루어질 수 있고 일관성 검증은 관계 다이어그램간 대응요소의 존재여부를 조사함으로써 이루어진다.

3.2 쓰임새 다이어그램(Use Case Diagram)

쓰임새 다이어그램은 시스템의 동적인 면을 모델링하기 위한 것으로 메타모델은 (그림 3)과 같다. 쓰임새 다이어그램은 쓰임새를 중심으로 구성되어 있다. 쓰임새 사이에는 일반화, 포함/확장(include/extend) 관계를 가진다. 행위자(actor)는 쓰임새를 수행하는 역할을 한다. 쓰임새의 인스턴스는 시나리오이다. 쓰임새는 이름, 사건주흐름, 사건예외흐름으로 구성되어 있다.



(그림 3) 쓰임새 다이어그램의 메타모델

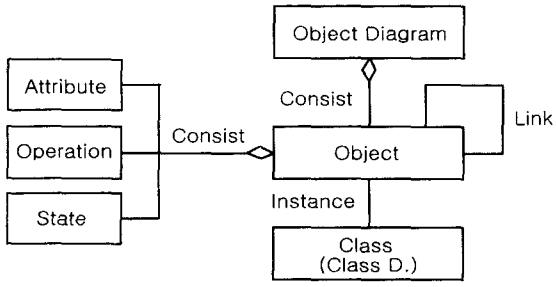
쓰임새 다이어그램의 쓰임새는 순차 다이어그램의 시나리오(scenario)와 관계를 가지며 클래스 다이어그램의 오퍼레이션(operation)과 대응관계를 가진다. 쓰임새 다이어그램의 일관성 검증요소를 도출하면 <표 2>와 같다.

〈표 2〉 쓰임새 다이어그램의 일관성 검증요소

검증요소	관계(요소)	대응요소	대응다이어그램
Use Case	Instance(scenario)	·	Sequence D.
Use Case	Realization(Collaboration)	Operation	Class D.

3.3 객체 다이어그램(Object Diagram)

객체 다이어그램은 어느 특정한 시간에 객체들의 집합과 그 관계를 나타낸 것이다. 객체 다이어그램은 객체를 중심으로 구성되며 객체는 클래스의 인스턴스가 된다. 객체 다이어그램은 교류도가 표현하는 동적인 흐름 중에서 하나의 정적인 프레임을 말한다. 객체 다이어그램의 메타모델은 (그림 4)와 같다.



(그림 4) 객체 다이어그램의 메타모델

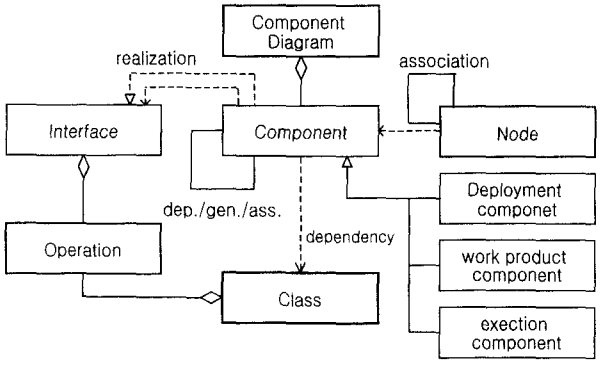
객체 다이어그램의 객체는 클래스, 순차 다이어그램의 객체 등과 대응관계를 가진다. 메타모델을 통해 도출한 객체 다이어그램의 일관성 검증요소는 <표 3>과 같다.

〈표 3〉 객체 다이어그램의 일관성 검증요소

검증요소	관계(요소)	대응요소	대응다이어그램
Object	Instance	Class	Class D.
Object	Correspond	Object	Sequence D.

3.4 컴포넌트 다이어그램(Component Diagram)

컴포넌트 다이어그램은 시스템의 물리적인 관점을 모델링하며 정적인 관점을 가진다. 컴포넌트 다이어그램은 시스템의 다양한 컴포넌트들과 그들 사이의 의존관계를 나타낸다. 컴포넌트 다이어그램의 메타모델은 (그림 5)와 같다.



(그림 5) 컴포넌트 다이어그램의 메타모델

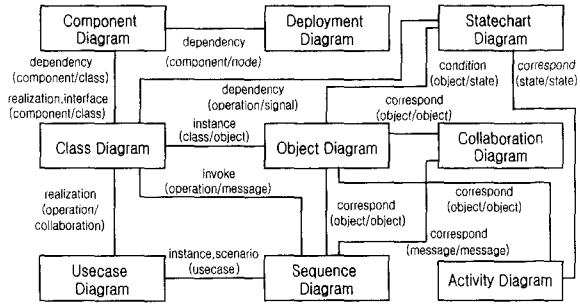
컴포넌트 다이어그램의 컴포넌트는 클래스, 노드 등과 대응관계를 가진다. 컴포넌트 다이어그램의 일관성 검증요소를 도출하면 <표 4>와 같다.

〈표 4〉 컴포넌트 다이어그램의 일관성 검증요소

검증요소	관계(요소)	대응요소	대응다이어그램
Component	Dependency	Class	Class D.
Component	Dependency	Node	Deployment D.
Component	Realization(Interface)	Class	Class D.

3.5 다이어그램간의 관계

각 다이어그램의 일관성 검증을 위한 관계요소를 조사하여 UML의 9개 다이어그램 전체에 대하여 다이어그램간의 관계요소를 표현하면 (그림 6)과 같다.

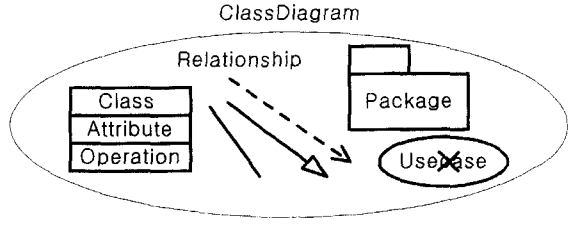


(그림 6) 9개 다이어그램의 일관성검증 관련 요소

4. 검증규칙의 유도

검증규칙은 메타모델을 통해 유도한다. UML Specification에서는 각 구성요소별 Well-Formedness Rule을 제시하고 있다. 그러나 각 다이어그램별 검증규칙은 정해져 있지 않다. 따라서 본 연구에서는 UML Specification을 기초로 하여 각 다이어그램에 맞는 정확성 검증규칙을 유도하고 다이어그램간의 관계를 고려하여 일관성 검증규칙을 유도한다. 검증규칙은 모든 다이어그램에 공통적으로 적용되는 기본규칙을 기초로 각 다이어그램의 세부규칙을 도출한다. 클래스 다이어그램을 위한 정확성 검증은 다음과 같이 표현할 수 있다.

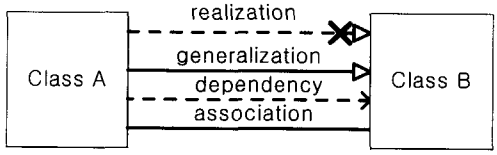
● **<정확성 RULE 기본규칙1 [구성요소의 포함여부]**: 하나의 다이어그램은 고유의 구성요소만을 가진다> 이 규칙은 정확성 검증의 가장 기본이 되는 규칙으로 (그림 7)과 같이 각 다이어그램이 그들이 포함할 수 있는 구성요소로 이루어져 있는가를 판단하는 것이다. 클래스 다이어그램의 경우는 다음과 같은 세부규칙을 유도할 수 있다.



(그림 7) 구성요소의 포함여부

- 세부규칙 11 : 클래스 다이어그램은 클래스, 인터페이스, 패키지와 같은 기본구성요소를 포함한다.
- 세부규칙 12 : 클래스 다이어그램은 의존, 연관, 일반화, 실체화와 같은 관계(Relationship)를 포함한다.
- 세부규칙 13 : 클래스는 속성, 오퍼레이션을 포함한다.
- 세부규칙 14 : 패키지는 클래스, 인터페이스를 포함한다.
- 세부규칙 15 : 인터페이스는 오퍼레이션을 포함한다.

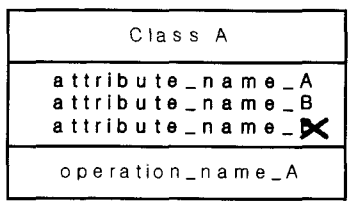
● **<정확성 기본규칙2 [구성요소간 관계의 적합성]** : 하나의 다이어그램 내에서 구성요소 간에는 관계가 존재한다> 이 규칙은 다이어그램을 구성하는 있는 구성요소 사이의 관계가 적절하게 이루어져 있는가를 판단하는 것이다. 예를 들어 (그림 8)과 같이 클래스와 클래스간에는 일반화, 연관, 의존과 같은 관계는 가질 수 있지만 실체화 관계는 가질 수 없다는 것이다. 클래스 다이어그램의 경우에는 다음과 같은 세부규칙이 유도된다.



(그림 8) 구성요소간 관계의 적합성

- 세부규칙 21 : 클래스와 클래스 사이에는 의존, 연관, 일반화 관계가 존재한다.
- 세부규칙 22 : 클래스와 인터페이스 사이에는 실체화 관계가 존재한다.
- 세부규칙 23 : 클래스와 패키지 사이에는 집합연관이 존재한다.
- 세부규칙 24 : 인터페이스와 인터페이스 사이에는 의존, 일반화 관계가 존재한다.

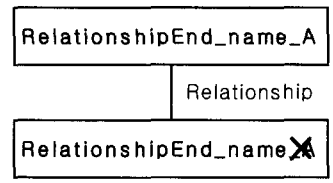
● **<정확성 기본규칙3 [구성요소의 유일성]** : 하나의 다이어그램 내에서 각각의 구성요소를 포함(consists)하는 상위 구성요소 내에서는 같은 종류의 구성요소일 경우 각 구성요소의 이름은 유일해야 한다> 이 규칙은 각 다이어그램을 구성하는 구성요소의 종류별로 유일한 이름을 가지는가를 판단하기 위한 규칙이다. (그림 9)와 같이 하나의 클래스 내에서 같은 이름의 속성을 가질 수 없다는 것이다. 클래스 다이어그램의 경우 다음과 같은 세부규칙을 유도할 수 있다.



(그림 9) 구성요소의 유일성

- 세부규칙 31 : 클래스 다이어그램 안에서 클래스 이름은 유일하다.
- 세부규칙 32 : 클래스 다이어그램 안에서 인터페이스 이름은 유일하다.
- 세부규칙 33 : 클래스 다이어그램 안에서 패키지 이름은 유일하다.
- 세부규칙 34 : 클래스 다이어그램의 클래스 안에서 속성 이름은 유일하다.

● **<정확성 기본규칙4 [관계요소의 유일성]** : 하나의 다이어그램 내에서 관계를 구성하는 두 요소의 이름은 유일해야 한다> 이 규칙은 각 다이어그램의 관계 즉 일반화, 의존, 연관, 실체화 관계를 가지는 양쪽 요소의 이름이 유일한가를 판단하는 규칙이다. (그림 10)과 같이 어떤 관계에서 양끝 요소의 이름은 같지 않아야 한다는 것이다. 클래스 다이어그램의 경우 다음과 같은 세부규칙을 가진다.

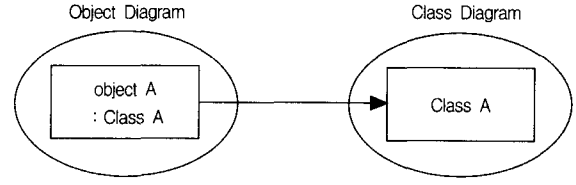


(그림 10) 관계요소의 유일성

- 세부규칙 41 : 일반화 관계를 구성하는 두 요소의 이름은 같지 않아야 한다.
- 세부규칙 42 : 의존 관계를 구성하는 두 요소의 이름은 같지 않아야 한다.
- 세부규칙 43 : 실체화 관계를 구성하는 두 요소의 이름은 같지 않아야 한다.

일관성 검증규칙은 앞장에서 메타모델을 통해 파악된 다이어그램간의 대응요소를 파악함으로써 유도할 수 있다. 검증규칙은 주로 이러한 대응요소가 해당 다이어그램에 포함되어 있는가를 판단하는 것으로 구성된다. 정확성 검증규칙과 마찬가지로 기본규칙을 정하고 이를 기초로 하여 세부규칙을 유도한다.

● **<일관성 기본규칙1 [다이어그램간 대응요소의 존재]** : 대응하는 다이어그램 사이에는 각 다이어그램 안에 대응하는 구성요소가 존재한다> 이 규칙은 다이어그램간 대응요소의 존재여부를 판단하는 규칙으로서 (그림 11)과 같이 객체 다이어그램의 경우 각 객체에 해당하는 클래스가 클래스 다이어그램에는 존재하여야 한다는 것이다. 클래스 다이어그램과 관계되는 일관성 규칙을 유도해보면 다음과 같다.



(그림 11) 다이어그램간 대응요소의 존재여부

- 세부규칙 51 : 객체 다이어그램의 각 객체에 대응하는 클래스가 클래스 다이어그램 내에 존재하여야 한다.
- 세부규칙 52 : 쓰임새 다이어그램의 쓰임새에 해당하는 오퍼레이션이 클래스 다이어그램 내에 존재하여야 한다.
- 세부규칙 53 : 컴포넌트 다이어그램의 컴포넌트에 포함되어 있는 클래스에 대응하는 클래스가 클래스 다이어그램 내에 존재하여야 한다.

5. 검증규칙의 정형명세

메타모델을 통해 유도된 텍스트 형태의 규칙을 좀 더 명확히 표현함으로써 다이어그램의 정확성을 파악하는데 용이함을 주고 나아가 검증작업의 자동화를 손쉽게 할 수 있으므로 정형적인 명세로의 변환이 요구된다. 즉 텍스트 형태의 규칙을 자동화 도구가 이해할 수 있는 형태로 좀 더 정형적으로 변환하는 작업이 필요하다. 기존의 전형적인 명세기법은 비정형 언어가 갖는 단점을 보완하기 위하여 대수적, 공리적으로 명세하는 방법을 취해왔으며, 이러한 전형적 명세기법[23]을 기반으로 객체지향 소프트웨어의 특징인 정보은닉, 상속성 등을 반영한 기법인 Object-Z[24], LOTOS[25], VDM++[26]과 같은 객체지향 명세기법들이 나타나게 되었다. 이와 같은 정형적 명세기법들은 수학적 지식에 바탕을 둔 것들로서 다이어그램의 논리적 정확성을 검증하는데는 도움을 줄 수 있으나 이해하기가 쉽지 않고 UML 다이어그램을 적절하게 표현하는데도 많은 제약사항이 따른다. 따라서 본 연구에서는 정형적 명세언어로서 UML 표준의 제약언어로 사용되고 있는 OCL(Object Constraints Language)을 이용하여 명세한다.

OCL은 UML 표준에 부속된 제한언어로서 UML의 구성요소를 표현하는데 적합하고, 정규문법이 정해져 있어서 Parser를 구현하기가 용이하고, 나아가 검증작업의 자동화에도 적합하다는 장점을 가지므로 본 연구에서는 OCL을 사용한다. OCL은 다이어그램 설계시 부수적인 제한조건(constraints)을 첨가하고자 할 때 사용되는 언어로서 다이어그램이 항상 유지되어야할 조건인 불변조건(invariants), 메소드의 실행으로 인한 다이어그램의 사전, 사후 조건을 기술하는 선후행조건(pre-/postconditions)을 표현할 수 있다. 본 연구에서는 유도한 검증규칙을 OCL의 불변조건 표현방법을 이용하여 명세한다. OCL의 불변조건 표현형식은 다음과 같다.

```
context verified_element inv rule_name :
verification_condition
```

*verified_element*는 검증하고자 하는 다이어그램의 요소이며 *rule_name*은 검증규칙에 부여하는 고유이름이다. *virification_condition*은 검증규칙을 표현하는 부분이다. 이는 검증하고자 하는 요소가 조건에 맞도록 표현되었는지를 판단하는 것으로 true 또는 false를 결과 값으로 가진다.

위의 구조로 앞장에서 유도한 검증규칙을 OCL로 표현한다. 구성요소 포함여부를 검증하는 검증규칙의 1의 세부규칙의 경우 다음과 같이 OCL로 명세할 수 있다.

```
context ClassDiagramElement inv rule11 :
self.allContents ( ) -> forAll (c | c.oclIsKindOf (Class) or
c.oclIsKindOf (Interface) or c.oclIsKindOf (Package) or
c.oclIsKindOf (Relationship))

context Relationship inv rule 12 :
self.allContents ( ) -> forAll (c | c.oclIsKindOf (Dependency) or
c.oclIsKindOf (Association) or
c.oclIsKindOf (Generalization) or
c.oclIsKindOf (Realization))

context Class inv rule 13 :
self.allContents ( ) -> forAll (c | c.oclIsKindOf (Attribute) or
c.oclIsKindOf (Operation))

context Package inv rule 14 :
self.allContents ( ) -> forAll (c | c.oclIsKindOf (Class) or
c.oclIsKindOf (Interface))

context Interface inv rule 15 :
self.allContents ( ) -> forAll (c | c.oclIsKindOf (Operation))
```

각 규칙은 *allContents*()라는 메소드를 통하여 'ClassDiagramElement,' 'Relationship,' 'Class,' 'Package'와 같은 요소가 가지고 있는 하위 구성요소를 파악한다. 또한 *forAll*이라는 OCL의 표준 오퍼레이션을 통해 각 구성요소의 존재여부를 검증하는 것이다. 각 표현의 결과는 Boolean 값을 가지므로 true 또는 false를 리턴한다. rule 11의 경우 *allContents*()는 다음과 같이 표현할 수 있다.

```
contents ( ) : Set (ClassDiagramElement) = self.r_package ;
allContents ( ) : Set(ClassDiagramElement) = self.contents ( ) ->
                  union (self.r_class) -> union (self.r_interface) ->
                  union (self.r_relationship) ;
```

구성요소간의 관계를 검증하는 정확성 기본규칙 2의 각 세부규칙의 경우는 다음과 같이 명세할 수 있다.

```
context Relationship inv rule 21 :
self.allInstance ( ) ->
forAll (r | r.oclIsKindOf (Dependency) or
r.oclIsKindOf (Association) or
r.oclIsKindOf (Generalization))

context Relationship inv rule 22 :
self.allInstance ( ) ->
forAll(r | r.oclIsKindOf (Realization))

context Relationship inv rule 23 :
self.allInstance ( ) ->
forAll (r | r.oclIsKindOf (Association))

context Relationship inv rule 24 :
self.allInstance ( ) ->
forAll (r | r.oclIsKindOf (Dependency) or
r.oclIsKindOf (Generalization))
```

위의 각 규칙은 *allInstance*()라는 메소드를 통하여 조건에 맞는 인스턴스를 구한다. 마찬가지로 forAll 오퍼레이션을 통하여 존재여부를 검증한다. Rule 21의 *allInstance*()는의 다음과 같이 표현할 수 있다.

```
instance ( ) : Set (Relationship) = self.r_generalization ->
select (r | r.relation1 = 'class' and r.relation2 = 'class')
```

```
allInstance ( ) : Set (Relationship) = self.instance( ) ->
union (self.r_dependency -> select(r|r.relation1 = 'class' and
                                   r.relation2 = 'class')) ->
union (self.r_association -> select(r|r.relation1 = 'class' and
                                   r.relation2 = 'class')) ->
union (self.r_realization -> select (r | r.relation1 = 'class' and
                                   r.relation2 = 'class'))
```

구성요소의 유일성을 검증하는 정확성 기본규칙 3의 각 세부규칙의 경우는 다음과 같이 명세할 수 있다.

```
context ClassDiagramElement inv rule31 :
self.r_class -> select (a | a.oclIsKindOf (Class)) ->
forAll (p, q | p.cname = q.cname implies p = q)

context ClassDiagramElement inv rule32 :
self.r_interface -> select (a | a.oclIsKindOf (Interface)) ->
forAll (p, q | p.iname = q.iname implies p = q)

context ClassDiagramElement inv rule33 :
self.r_package -> select (a|a.oclIsKindOf (Package)) ->
forAll (p, q | p.pname = q.pname implies p = q)

context Class inv rule34 :
self.r_attribute -> select (a | a.oclIsKindOf (Attribute)) ->
forAll (p, q | p.aname = q.aname implies p = q)
```

위의 각 규칙은 'ClassDiagramElement', 'Class'와 같은 요소와 연관 관계를 가지는 대응요소를 self.role_name을 통하여 파악하고 forAll 오퍼레이션을 통하여 이름의 유일성을 파악하는 것이다.

관계의 유일성에 관한 정확성 기본규칙 4의 각 세부규칙을 OCL로 명세하면 다음과 같다.

```
context Relationship inv rule 41 :
self.r_generalization -> select (a | a.oclIsKindOf (Generalization))->
    forAll(r1, r2 | r1.r1name = r2.r2name implies r1 = r2)

context Relationship inv rule 42 :
self.r_dependency -> select (a | a.oclIsKindOf (Dependency)) ->
    forAll (r1, r2 | r1.r1name = r2.r2name implies r1 = r2)

context Relationship inv rule 43 :
self.r_realization -> select (a | a.oclIsKindOf (Realization)) ->
    forAll (r1, r2 | r1.r1name = r2.r2name implies r1 = r2)
```

위의 각 규칙은 관계의 각 종류별 해당요소를 구별해내고 forAll 오퍼레이션을 통하여 관계요소의 이름이 같은지를 판단하는 것이다.

클래스 다이어그램과 관계를 가지는 다이어그램과의 일관성 검증을 체크하기 위한 일관성 기본규칙 1의 세부규칙을 OCL로 개략적으로 표현하면 다음과 같다.

```
context DiagramElement inv rule 51 :
self.objectdiagram.object_role_name -> exist (c | c.oname =
self.classdiagram.class_role_name.cname)

context DiagramElement inv rule 52 :
self.componentdiagram.component_role_name ->
exist (c | c.coname = self.classdiagram.class_role_name.cname)

context DiagramElement inv rule 53 :
self.usecasediagram.usecase_role_name -> exist (c | c.uname =
self.classdiagram..class_role_name.cname)
```

6. 규칙의 검증 및 평가

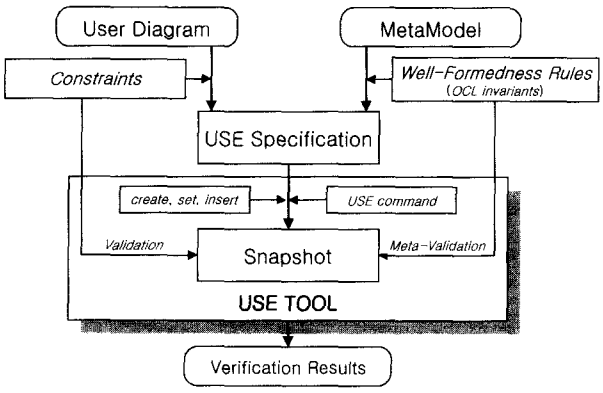
OCL을 이용하여 세부규칙을 표현함으로써 얻는 이점 중의 하나는 OCL의 정규문법이 정의되어 있으므로 Parser를 만들기가 용이하고 따라서 이를 이용하여 검증자동화 도구를 구현하기가 용이하다는 것이다. 현재 OCL을 처리하는 도구로는 OCL 자체의 문법적 정확성을 체크하는 도구가 주를 이루고 있으며, 사용자가 작성한 다이어그램과 OCL을 입력으로 받아 다이어그램의 정확성을 검증하는 도구는 소수에 불과하다. OCL을 입력으로 받아들여 문법적 오류와 제한조건을 자동적으로 검증해주는 도구로는 IBM의 OCL parser[28], Dresden OCL Toolkit[27], TU Munich[20], ModelRun[21], USE tool[22]과 같은 것이 있다. <표 5>는 OCL 관련 도구의 기능을 비교 분석한 것이다. IBM의 OCL parser는 OCL의 문법적 정확성을 분석할 수 있는 도구로서 가장 먼저 개발된 도구이다. 이 도구는 OCL 명세의 문법 분석 기능 및 타입 체크 기능을 가지고 있으며 다른 도구 발전의 기초가 되고 있다. 현재는 Klasse Objecten에서 관리되고 있다. Dresden OCL Toolkit은 OCL compiler 기능을 가지는 것으로 OCL에 관한 문법 체크 기능이 UML 모

〈표 5〉 OCL 관련 도구의 기능 비교

기능	도 구				
	IBM Parser	Dresden OCL Toolkit	TU Munich	ModelRun	USE
syntactical analysis	•	•	•	•	•
type checking	•	•	•	•	•
logical consistency checking	-	-	-	-	-
dynamic invariant validation	-	-	•	•	•
dynamic pre-/post-condition validation	-	-	-	-	•
test automation	-	-	-	-	•
code verification & synthesis	-	-	-	-	-

델링 도구인 Argo/UML[29]에 삽입되어 사용되고 있다. TU Munich은 OCL 메타모델을 기반으로 OCL의 문법을 체크하고 불변조건을 검증하는 기능을 가지고 있다. Model-Run은 상업적인 도구로서 다이어그램의 스냅샷에 대하여 불변조건을 검증해주는 기능을 가진다. USE는 보다 발전된 형태의 도구로서 문법적 오류뿐만 아니라 OCL의 TYPE을 체크하는 기능, 스냅샷에 대한 불변조건, 선후행조건을 검증하는 기능을 가지고 있다. 따라서 USE는 OCL 관련 도구 중 보다 발전된 형태로 파악된다[7].

USE는 사용자가 작성한 다이어그램을 클래스 형태의 USE 명세로 변형하고, USE 명령어를 통하여 USE 명세의 인스턴스인 객체 다이어그램을 생성하고 여기에 제한조건, 선후행조건을 부가하여 다이어그램의 정확성을 검증한다. USE의 다이어그램 검증 과정은 (그림 12)와 같다.



(그림 12) USE 도구의 검증 과정

USE의 이용방법은 Validation과 Meta-Validation으로 구분할 수 있다. Validation은 사용자 다이어그램에 제한조건을 설정하여 입력하고 이에 대한 인스턴스를 생성하여 인스턴스가 제한조건에 맞는지를 검증하는 방법이다. 이 방법은 인스턴스가 갖는 "내부데이터의 정확성"을 검증하는데 주로 사용된다. Meta-validation은 메타모델에 Well-Formedness Rule을 설정하여 입력하고 이에 대한 인스턴스 모델을 생성하여 인스턴스 모델이 Well-Formedness Rule에 적합한지를 검증하는 방법이다. 이 방법은 "모델 자체의 정확성"을 검증하는데 주로 사용한다. 따라서 본 연구에서는 USE의 Meta-Validation 방법을 이용하여, 다이어그램별 메타모델을 USE 명세로 표현하여 입력하고 유도한 검증규칙을 Well-Formedness Rule과 같이 표현하고 사용자 다이어그램을 메타모델의 인스턴스와 같이 생성하여 검증하는 방법을 취한다.

USE를 이용하여 사용자가 작성한 다이어그램을 검증하기 위해서는 우선 메타모델을 USE 명세로 변환하는 작업이 필요하다. USE 명세는 다음과 같은 구조를 가진다.

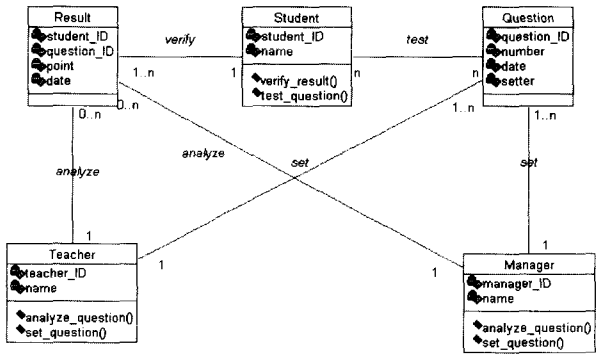
```
model model_name
class class_name
attributes
 attributes_name
operation
model model_name
class class_name
attributes
 attributes_name
operation
 operation_name
end
association association_name
between
 class_name1 role role_name1
 class_name2 role role_name1
end
constraints
context verified_element inv rule_name :
verification_condition
```

검증의 기초가 되는 메타모델의 이름을 *model_name*으로 정하고 'class-end' 구조로 다이어그램을 구성하는 요소들을 표현한다. 각 요소에 따르는 제한 조건은 '**constraints**' 부분에 표현한다. '**constraints**' 부분은 앞장에서 언급한 것과 같이 invariant 표현형식을 이용한다. 이렇게 표현한 USE 명세를 USE 도구의 입력으로 하여 USE 문법의 정확성과 OCL 문법의 정확성을 검증하게 된다. 입력된 명세의 문법적 체크가 끝나면 '**constraints**'를 검증하기 위하여 입력된 다이어그램의 스냅샷을 생성한다. 이는 입력된 클래스 형태의 명세로부터 인스턴스를 생성하는 것으로 객체 다이어그램 또는 순차 다이어그램 형태로 표현된다. 생성된 객체 다이어그램 또는 순차 다이어그램에 USE 명세에서 정한 '**constraints**'를 적용하여 정확성 결과를 얻어내는 것이다. 클래스 다이어그램의 메타모델의 경우 다음과 같이 명세할 수 있다.

```
model ClassDiagramMetaModel
-- classes

abstract class ClassDiagramElement
attributes
  ename : String ;
operations
  contents ( ) : Set (ClassDiagramElement) = self.r_package ;
  allContents ( ) : Set (ClassDiagramElement) = self.contents ( ) ->
union (self.r_class) -> union (self.r_interface) ->
                        union (self.r_relationship) ;
end
class Package < ClassDiagramElement
attributes
  pname : String ;
operations
  contents ( ) : Set (Package) = self.r_interface2 ;
allContents ( ) : Set (Package) = self.contents ( ) ->
 union (self.r_class2) ;
end
class Class < ClassDiagramElement, Package
attributes
 cname : String ;
operations
 contents ( ) : Set (Class) = self.r_attribute ;
```

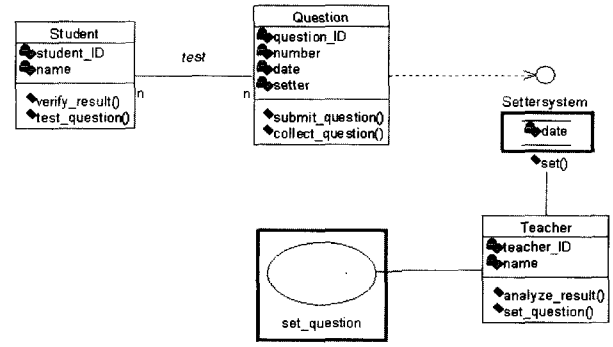
```
 allContents ( ) : Set (Class) = self.contents ( ) ->
 union (self.r_operation) ;
end
...
-- associations
association CD_C
between
  ClassDiagramElement [0..1] role r_classdiagramelement
  Class [*] role r_class
end
association CD_P
between
 ClassDiagramElement [0..1] role r_classdiagramelement2
 Package [*] role r_package
end
...

constraints
context ClassDiagramElement inv rule11 :
self.allContents ( ) -> forAll (c | c.oclIsKindOf (Class) or
c.oclIsKindOf (Interface) or c.oclIsKindOf (Package) or
c.oclIsKindOf (Relationship))
...
```

5장에서 유도한 검증규칙을 검증하기 위하여 (그림 13)과 같은 간단한 모델을 가정한다. 가정한 모델은 웹기반의 학습평가시스템으로 WEB 상에서 교수자와 학습자가 상호작용하면서 학습할 수 있는 공간을 제공하는 것으로 교수자는 학습자를 위하여 문제를 출제하고 학습자는 출제한 문제를 풀이하여 풀이한 결과를 바탕으로 각종 교육통계자료를 생산해내는 시스템이다. 관리자는 학습자와 교수자가 모두 이용할 수 있는 문제은행식의 문제를 출제하고 관리한다. 시스템의 일부분을 (그림 13)과 같이 클래스 다이어그램 형태로 나타낸다.



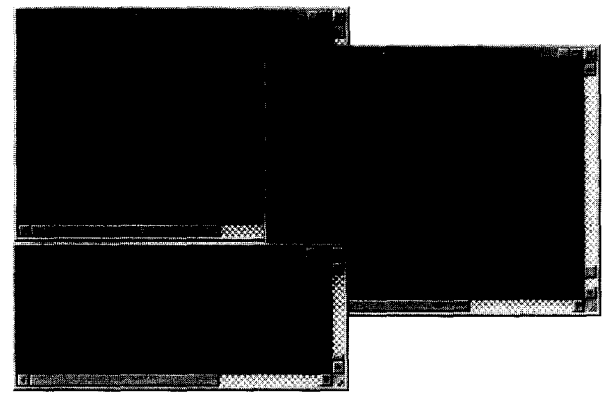
(그림 13) 학습평가 시스템의 클래스 다이어그램

클래스 다이어그램의 정확성 검증규칙 중 세부규칙 11~세부규칙 15를 검증하기 위하여 (그림 14)와 같은 예제다이어그램을 가정한다. 가정한 다이어그램은 그림에서 표시한 부분과 같이 두 가지의 오류를 가지고 있다. 즉 클래스 다이어그램이 포함할 수 없는 유즈케이스 구성요소를 포함하고 있으며, 'Settersystem'이라는 인터페이스도 속성을 포함할 수 없음에도 불구하고 포함하고 있는 오류를 가지고 있다. 따라서 이 모델은 세부규칙 11과 세부규칙 15에 어긋난다.



(그림 14) 세부규칙 11~세부규칙 15 검증을 위한 예제 다이어그램

위와 같은 오류가 있는 다이어그램을 검증하기 위하여 (그림 15)와 같은 명령어를 통하여 인스턴스 검증모델을 생성한다. 구성요소, 구성요소 사이의 관계, 구성요소의 값을 생성하는 명령이 실행된다.

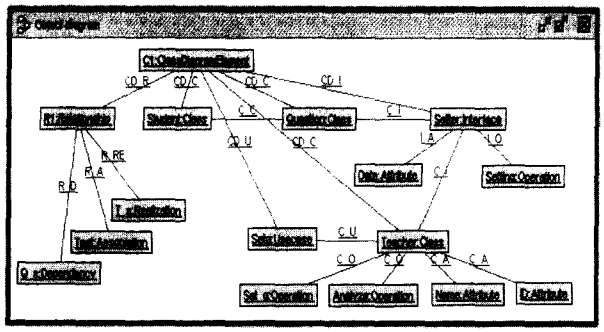


(그림 15) (그림 14) 예제 다이어그램 생성을 위한 USE 명령

인스턴스 모델을 생성하는 명령은 create, insert, set과 같은 명령어가 있다. create는 인스턴스 모델의 구성요소를 생성하는 명령이고, insert는 구성요소 사이의 관계를 설정하여주는 명령이다. set 명령은 구성요소의 속성에 값을 설정하여주는 명령이다. 명령어의 형식은 다음과 같다.

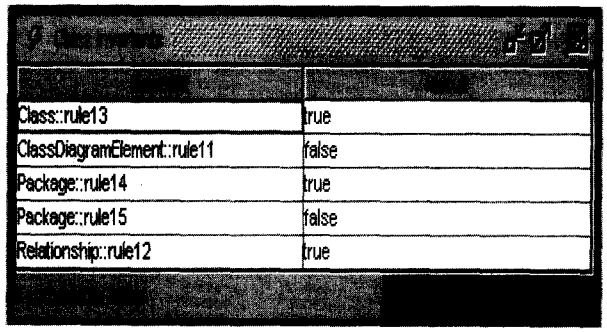
```
!create component_name : component
!insert (component_name, component_name) lass_name
        into association_name
!set acomponent_name.attributes_name = 'name'
```

생성된 인스턴스 모델은 (그림 5)~(그림 9)와 같이 객체 다이어그램 형태로 표현된다. 사각형으로 표현되는 객체와 그들 사이의 관계를 표현하고 있다. 객체는 메타모델에서 클래스로 표현한 각 구성요소의 인스턴스이다. 객체들 사이의 관계는 통합 USE 명세의 'association'에서 정의한 관계에 대한 인스턴스이다. 이렇게 객체 다이어그램으로 표현되는 인스턴스 모델에 통합 USE 명세의 'constraints' 부분에 정의된 각 규칙이 적용된다. 이 경우는 세부규칙 11~세부규칙 15가 적용된다.



(그림 16) (그림 14) 예제다이어그램 검증을 위한 인스턴스 모델

인스턴스 모델에 대한 검증규칙의 적용결과는 (그림 17)과 같이 나타난다. 예제다이어그램의 경우 위에서 언급한 것과 같이 세부규칙 11과 세부규칙 15에 어긋나므로 해당항목에 'false' 결과가 나타난다.



Class::rule13	true
ClassDiagramElement::rule11	false
Package::rule14	true
Package::rule15	false
Relationship::rule12	true

(그림 17) (그림 14) 예제다이어그램에 대한 검증규칙의 적용 결과

USE는 OCL을 이용한 다이어그램의 검증에 사용되는 도구로서 가장 발전된 형태로 파악이 되지만 적용 후 다음과 같은 몇 가지 문제점을 발견하였다. 첫째, 검증을 위해 USE 명세라는 정형적 형태로 표현해야 하는 불편함이 있다. 이러한 문제점은 현실에서 아주 중요한 문제이다. 다이어그램의 정확성을 검증하기 위해 작성한 다이어그램을 또 다른 형태로 변형해야하는 작업은 굉장히 번거로운 것이다. 따라서 향후 UML 도구와의 연계를 통하여 사용자가 작성한 다이어그램이 전환과정 없이 검증될 수 있도록 하는 것이 필요하다. 둘째, 전체적인 검증 메커니즘이 다소 복잡하고 제한적이다. 즉 사용자가 작성한 다이어그램을 클래스 다이어그램 형태의 USE 명세로 표현하고 명령어를 통하여 다이어그램의 인스턴스를 생성하여 그 인스턴스에 제한조건을 적용하는 구조로 되어 있다. 즉 9가지 종류의 다이어그램 모두를 적용할 수 있을 만큼 폭넓은 명세규칙을 가지고 있지 못하고 단지 클래스 다이어그램 형태의 구조로만 명세될 수 있다는 것이다. 또한 인스턴스의 형태는 단지 객체 다이어그램 또는 순차 다이어그램 형태로만 표현될 수 있기 때문에 모든 다이어그램을 수용하기에 제한적이라고 할 수 있다. 셋째, 제한조건을 이용하여 사용자가 작성한 다이어그램의 정확성을 검증하고 나아가서 UML의 Core와 같은 메타모델을 입력으로 하여 Well-Formedness Rule을 검증할 수는 기능은 가지고 있으나 두 가지의 기능 즉 모델에 대한 제한조건과 Well-Formedness Rule을 같이 적용하기에는 부족한 면이 있다. 본 연구에서는 도출한 다이어그램별 메타모델을 입력으로 하고 검증규칙을 Well-Formedness Rule로 표현하여 부가하고 사용자 다이어그램을 메타모델의 인스턴스로 생성하여 검증하고자 시도하였다. 그러나 (그림 15)의 표시된 부분과 같이 다이어그램의 형태가 (그림 13)의 원래 작성한 다이어그램과는 달리 메타모델과 같은 형태로 검증할 수밖에 없었다. 제한조건은 통상적으로 다이어그램의 속성 또는 오퍼레이션의 조건을 부가하는데 사용되며 Well-Formedness Rule은 메타모델의 건전성을 판단하기 위해 꼭 지켜져야 하는 규칙을 표현하는데 사용되는 것으로서 성질이 다르기는 하지만 두 가지 제한규칙의 통합된 적용이 요구된다. 넷째, 단독 다이어그램의 정확성 검증 기능은 구현할 수 있으나 몇 개 다이어그램간의 일관성을 검증하기 위한 환경은 제공하지 못한다는 것이다. 일관성이란 다이어그램간의 상관요소의 존재여부 등을 검증하는 것이다. 따라서 몇 개 다이어그램을 폭넓게 수용하여 검증할 수 있는 환경이 요구된다.

〈표 6〉 본 연구에서 유도한 검증규칙의 적용여부

검증규칙	UML 모델링 도구				
	Rose	Together	ArgoUML	Plastic	DEBUT
세부규칙 11	△	•	•	△	•
세부규칙 12	•	△	•	•	•
세부규칙 13	•	•	•	•	•
세부규칙 14	•	•	•	-	•
세부규칙 15	-	-	•	-	•
세부규칙 21	-	•	•	-	•
세부규칙 22	△	-	△	-	•
세부규칙 23	-	-	-	-	-
세부규칙 24	△	△	•	-	-
세부규칙 31	•	•	-	-	△
세부규칙 32	•	•	-	-	△
세부규칙 33	•	•	-	-	△
세부규칙 34	•	•	-	-	△
세부규칙 41	•	•	-	-	-
세부규칙 42	•	•	-	-	-
세부규칙 43	•	-	-	-	•

• : 적용 △ : 부분적용 - : 적용않음

본 연구에서는 다이어그램의 정확성을 검증하기 위한 방법으로 16개의 정확성 검증규칙을 유도하고 이를 OCL의 invariants로 표현한 다음 USE를 이용하여 다이어그램의 정확성을 검증한다. 유도한 검증규칙의 유용성을 평가하기

위하여 보편적으로 사용되고 있는 UML 모델링 도구를 선정하여 본 연구에서 유도한 검증규칙이 적용되고 있는가를 조사해본다. 조사대상이 되는 모델링 도구는 가장 보편적으로 사용되고 있는 도구인 Rational Rose ver. 2001, Together ver. 4.2, ArgoUML ver. 0.10, Plastic ver. 2.0, MaRMI-Ⅱ DEBUT이다. 조사결과는 <표 6>과 같다.

조사 결과 기존 도구는 본 연구에서 유도한 검증규칙을 제대로 적용하지 않는 것으로 나타났다. 도구에 따라 차이가 있지만 일부 세부규칙 만을 검증하고 있었으며, 특히 구성요소의 유일성과 관계요소의 유일성에 관한 세부규칙은 제대로 검증되지 않고 있었다. 따라서 본 연구에서 유도한 검증규칙은 향후 유용한 검증기준으로서 도구에 적용가능성이 있다고 판단된다.

7. 결론 및 향후 연구 방향

본 연구에서는 사용자가 작성한 다이어그램의 일관성 및 정확성을 검증하기 위한 방법으로 UML 표준에 부속되어 있는 모델제약 언어인 OCL을 이용하여 검증하는 방법을 제시하였고, 이에 따라 메타모델을 유도하고 그것으로부터 검증규칙을 유도하였다. 메타모델은 UML Specification을 기반으로 다이어그램별로 구성요소와 관계를 고려하여 작성하였다. 특히 일관성 검증을 위하여 다이어그램간의 관계요소를 적절히 표현하였다. 작성된 메타모델을 기반으로 정확성과 일관성 검증을 위한 검증규칙을 유도하였다. 유도된 검증규칙은 다이어그램의 정확성을 파악하는데 용이함을 주고 검증작업의 자동화를 용이하게 하기 위하여 정형적으로 명세하였다. 본 연구에서는 특징적으로 UML 준의 모델제약언어로 사용되고 있는 OCL을 사용하여 검증규칙을 정형적으로 명세하였다. 검증규칙은 OCL의 문법적 정확성과 불변조건, 선후행조건을 검증할 수 있는 가장 발전된 형태의 도구인 USE를 이용하여 유용성을 검증하였다.

USE는 OCL을 사용하여 검증조건을 표현함으로써 다이어그램의 구성요소를 보다 정확하게 표현할 수 있으며, 검증조건을 보다 명확하게 기술할 수 있다는 장점을 가진다. 그러나 USE는 이미 제시한 바와 같이 몇 가지 단점을 가진다. 따라서 향후 연구 과제로 본 연구에서 제시한 검증규칙을 모두 수용하고 제시한 문제점을 해결할 수 있는 자동화된 검증 시스템에 대한 연구가 필요하다.

본 연구에서 유도한 검증규칙의 유용성을 검증하기 위해 보편적으로 사용되고 있는 UML 모델링 도구를 선정하여 세부규칙의 적용여부를 조사하였다. 그 결과 대부분의 도구에서 세부규칙을 만족스럽게 적용하고 있지는 않는 것으로 나타났다. 따라서 본 연구에서 제시한 검증규칙이 보다 새로운 검증기준으로 사용될 가능성이 있는 것으로 나타났다. 그러나 본 연구에서 유도한 검증규칙이 다이어그램을 완벽하게 검증할 수 있는 것은 아니다. Class Diagram의 reflexive association이나 N-ary association에 대해서는 적용할 수 없다는 문제점 등을 가진다. 추후 보다 완벽한 검증을 위한 검증규칙에 대한 연구가 필요하다.

참 고 문 헌

[1] OMG, "Object Constraint Language Specification," OMG Unified Modeling Language Specification Version 1.4, 2001.

[2] OMG, "UML Semantics," OMG Unified Modeling Language Specification Version 1.4, 2001.

[3] OMG, "OMG Unified Modeling Language Specification Version 1.4," Object Management Group Inc. 2001, Internet, http://www.omg.org.

[4] G. Booch, J. Rumbaugh and I. Jacobson, "The Unified Modeling Language User Guide," Addison-Weslay, 1999.

[5] M. Fowler and K. Scott, "UML Distilled," Addison - Wesley, 1999.

[6] J. B. Wormer and A. G. Kleppe, "The Object Constraint Language," Addison-Wesley, 1999.

[7] M. Richters, "A Precise Approach to Validating UML Models and OCL Constraints," PhD thesis, University Bremen, Logos Verlag, Berlin, BISS Monographs, No.14, 2002.

[8] B. Hnatkowska, Z. Huzar and J. Magott, "Consistency Checking in UML Models," Proc. of 4th International conference on Information Systems Modeling ISM '01, 2001.

[9] M. Richters and M. Gogolla, "Validating UML models and OCL Constraints," Proc. of UML2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, Vol.1939 of LNCS, pp.265-277, Oct., 2000.

[10] A. Tsiolakis and H. Ehrig, "Consistency analysis of UML class and sequence diagrams using attributed graph grammars," Proc. of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, Mar., 2000.

[11] A. Tsiolakis, "Consistency Analysis of UML Class and Sequence Diagrams based on Attributed Typed Graphs and their Transformation," Technical Report 2000/3, Technical University of Berlin, Mar., 2000.

[12] T. Sunetnanta and A. Finkelsteing, "Automated Consistency Checking for Multiperspective Software Specifications," Proc. of the 23rd Internation Conference on Software Engineering, ICSE2001, May, 2001.

[13] P. Krishnan, "Consistency Checks for UML," Proc. of the 7th Asia-Pacific Software Engineering Conference, pp.162-169, Dec., 2000.

[14] P. Bottoni, M. Koch, F. Parisi-Presicce and G. Taentzer, "Consistency Checking and Visualization of OCL Constraints," Proc. of UML2000 The Unified Modeling Lan-

guage. Advancing the Standard. Third International Conference, Vol.1939 of LNCS, pp.294-308, Oct., 2000.

[15] P. Anre, A. Romanczuk, J. Royer and A. Vasconcelos, "Checking the Consistency of UML class Diagram Using Larch Prover," Proc. of Third Workshop on Rigorous Object-Oriented Methods, ROOM '2000, Jan., 2000.

[16] 김진수, 강권학, 이경환, "제약언어를 이용한 객체 모델 검증 시스템", 정보처리학회논문지, 제3권 제6호, 1996.

[17] 조진형, 배두환, "UML 객체지향 분석 모델의 완전성 및 일관성 진단을 위한 시나리오 검증기법", 정보과학회논문지, 제28권 제3호, 2001.

[18] 정기원, 조용선, 권성구, "객체지향 설계방법에서 오류 검출과 일관성 점검기법 연구", 정보처리논문지, 제6권 제8호, 1999.

[19] 김도형, 정기원, "객체지향 분석과정에서 오류와 일관성 점검방법", 정보과학회논문지(B), 제26권 제3호, 1999.

[20] M. Wittmann, "Ein Interpreter fur OCL," Diplomarbeit, Ludwig-Maximilians-Universitat Munchen, 2000.

[21] BoldSoft, "Modelrun," 2000, Internet, http://www.boldsoft.com/products/modelrun/index.html.

[22] M. Richters, "The USE tool : A UML-based specification environment," 2001, Internet, http://www.db.informatik.uni-bremen.de/projects/USE/.

[23] J. M. Spivey, "The Z Notation : A Reference Manual," 2Ed., Programming Research Group University of Oxford, 1998.

[24] R. Duke, P. King, G. Rose and G. Smith, "The Object-Z Specification Language," Ver.1, 1991.

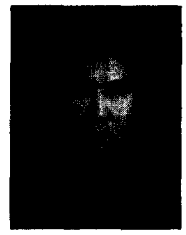
[25] T. Bolognesi and Ed Brinksma, "Introduction to the ISO Specification Language LOTOS," Computer Networks and ISDN Systems, 1987.

[26] E. H. Durr and J. V. Katwijk, "VDM++ - A Formal Specification Language for Object-oriented Designs," Proc. of CompEuro '92, pp.214-219, 1992.

[27] H. Hussmann, B. Demuth and F. Finger, "Modular architecture for a toolset supporting OCL," Proc. of UML2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, Vol.1939 of LNCS, pp.278-293, Oct., 2000.

[28] IBM, "OCL Parser," ver.o.3, Internet, http://www-3.ibm.com/software/ad/library/standards/ocl.html.

[29] J. Robbins et al., Argo/UML CASE tool, 2001, Internet, http://www.argouml.org.



하 일 규

e-mail : ilkyuha@yumail.ac.kr

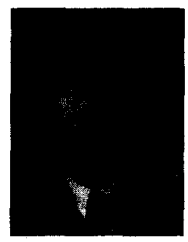
1992년 영남대학교 공과대학 전산공학과 (학사)

2001년 영남대학교 정보처리교육전공(석사)

2003년 영남대학교 대학원 컴퓨터공학과 (박사)

1992년~1995년 증권감독원 전산업무실

관심분야 : UML, OCL, 정형명세기법, 소프트웨어공학, 원격교육



강 병 욱

e-mail : bwkang@yu.ac.kr

1970년 영남대학교 공과대학 전기공학과 (학사)

1977년 영남대학교 대학원 전자공학과 (석사)

1994년 경북대학교 대학원 전자공학과 (박사)

1973년~1976년 영남대학교 전자계산연구소(SE)

1977년~1978년 영남전문대학 전자과 전임강사

1979년~현재 영남대학교 공과대학 전자정보공학부 교수

관심분야 : UML, OCL, 정형명세기법, 소프트웨어공학, 프로그래밍언어, 원격교육