객체지향 설계방법에서 오류 검출과 일관성 점검기법 연구

정 기 원† · 조 용 선†† · 권 성 구††

요 약

소프트웨어의 대형화와 사용자 요구사항의 복잡화에 따라, 소프트웨어 개발 초기 단계에서의 산출물의 정확성과 일관성은 중요해지고 있다. 하지만, 객체지향 방법을 기반으로 한 설계 문서에 대한 오류 검출 및 일관성 점검을 위한 기법은 만족할 만한 수준에 이르지 못하고 있다. 본 논문은 UML(Unified Modeling Language)의 다이어그램들에 대한 메타모델을 작성하고, 메타모델의 각 요소들에 대하여 적용할 일반화된 메타규칙을 도출하고, 메타규칙들을 각 다이어그램에 적용한 세부규칙 도출에 활용하였다. 이 방법은 오류 검출과 일관성 점검에 활용할 세부규칙들을 체계적으로 도출하므로, 규칙의 완전성을 도모하고 규칙적용의 자동화를 가능하게 하였다. 또한, 도출된 세부규칙을 적용한 사례를 통하여 그 효용성을 확인하였다.

Detecting Errors and Checking Consistency in the Object-Oriented Design Models

Ki-Won Chong† · Yong-Sun Cho†† · Sung-Goo Kwon††

ABSTRACT

As software size ever increases and user's requirements become more and more sophisticated, the importance of software quality is more and more emphasized. However, we are not satisfied for the present techniques on detecting errors and checking consistency in the object-oriented design model. This paper proposes a systematic approach which produces implementable rules to detect errors and check consistency. At first, the meta-models for UML diagrams are constructed, generalized meta-rules are reduced from the meta-models, and then the meta-rules are applied to produce the implementable rules. This approach enables to pursue the completeness of the rules and the automation of rule application. An example of rule application shows the feasibility of the rule application.

1. 서 론

새로운 소프트웨어를 개발함에 있어서 분석 단계 및 설계 단계와 같은 개발 초기 단계에서의 개발 산출물의 정확성과 일관성은 그 중요함을 아무리 강조해도 부족하지 않다. 이는 소프트웨어 공학에서 이야기하듯이, 부정확한 개발 초기 단계의 개발 산출물은 이후 단계에서 작성되는 개발 산출물들의 부정확성을 증가시키며, 이를 수정하는 것은 개발 후기 단계에서 수정작업이 이루어질수록 그 비용은 급격히 증가하기 때문이다. 특히, 분석 단계에서는 사용자의 요구사항을 파악하고 개발할 소프트웨어의 거시적인 모습을 정의하는 작업이 수행되므로 요구정의 내용의 타당성 검토가 중요하다. 또한, 설계 단계에서는 개발할 소프트웨어의

※ 이 논문은 1997년 한국학술진흥재단의 공모과제 연구비에 의하여 연구되었음.
† 정 회 원 : 숭실대학교 컴퓨터학부 교수
†† 준 회 원 : 숭실대학교 대학원 컴퓨터학과
논문접수 : 1999년 2월 25일, 심사완료 : 1999년 7월 9일

골격 및 세부사항들과 개발 방법이 정의되며 구현 작업의 기본이 될 설계 산출물들이 작성되므로, 설계 산출물들의 정확성 점검 작업의 중요성이 절실해진다. 이러한 이유로, 정확한 소프트웨어를 개발하고 개발 비용을 최소화 하기 위해서는 분석 모델과 설계 모델에서 작성된 각 개발산출물들의 오류 검출과 개발산출물들간의 일관성 점검은 반드시 필요한 작업들이라 할 수 있다.

최근에 개발되는 소프트웨어들은 그 규모가 커지고, 사용자의 요구사항이 복잡해짐에 따라 복잡성은 증가되고 있는 반면, 고품질의 소프트웨어가 요구된다. 그러나, 이러한 복잡한 소프트웨어의 개발에서는 개발 산출물에서 오류가 발생할 확률이 더욱 높아진다. 또한, 개발할 소프트웨어의 대형화에 따라 소프트웨어의 개발은 여러 개발자들이 개발 업무를 분담하여 수행되는 경우가 대부분이나, 이러한 개발 업무의 분담은 대형 소프트웨어의 개발을 좀 더 유용하게 하는 반면, 각 개발자들의 개발 산출물들간의 일관성을 잃게 할 확률을 높이게 된다. 이렇게 개발할 소프트웨어의 대형화 및 복잡화는 오류 발생 확률을 높이며 일관성을 잃게 할 확률을 높이면서도, 점검 대상인 개발 소프트웨어의 복잡함에 비례하여 오류 검출 및 일관성 점검은 더욱 어려워 진다. 특히 객체지향 방법론을 바탕으로 한 소프트웨어개발에서는 오류 검출 및 일관성 점검에 대한 기법들이 아직 많이 연구되지 않아, 대부분의 소프트웨어 개발자들은 그들의 경험이나 주관적인 판단에 의존하여 오류 검출 및 일관성 점검을 수행하고 있는 것이 현실이다. 이러한 이유로, 객체지향 설계방법에서의 오류 검출과 일관성 점검에 대한 기법들은 소프트웨어 개발에 중요한 의미를 가지며, 그 필요성 또한 날로 높아지고 있다.

이러한 배경을 바탕으로, 최근 객체지향 설계 모델로 가장 많이 사용되고 있는 UML을 기반으로 한 설계모델들의 중심이 되는 각각의 UML 다이어그램들을 구성하는 요소들 간의 관계를 나타내는 메타모델(Meta-Model)을 정의하여, 이 메타모델을 통하여 각 다이어그램들의 정확성을 유지하기 위한 일반화된 규칙을 찾아내고, 찾아낸 일반화된 규칙을 다이어그램을 구성하는 특정 요소들에 적용한 세부화된 오류 검출 규칙과 일관성 점검 규칙들을 도출하였다. 또한, 이러한 도출된 규칙을 적용하여 오류 검출과 일관성 점검을 자동적으로 수행할 수 있도록 하였으며, 이를 특정 사례에 적용하였다.

2장에서는 연구의 내용과 관련된 연구들을 소개하였으며, 3장에서는 규칙들의 도출을 위하여 작성된 UML 다이어그램들의 메타모델과 이를 통해 도출된 일반화된 규칙 및 세부 규칙들을 제시한다. 4장에서는 도출된 세부 규칙들을 적용하는 방법을 특정 사례를 통하여 알아보고, 5장에서는 유사한 성격의 연구들과의 비교평가를 수행한다. 6장에서 결론과 향후 연구방향을 제시한다.

2. 관련 연구

2.1 메타모델(Meta-Model)의 설계

메타모델은 모델을 형성하는 다이어그램의 구성 분자들간의 연관관계를 표현한 것으로, 소프트웨어를 모델링(Modeling)하기 위하여 표현하는 다이어그램 등의 개발 산출물과 이를 구성하는 요소들을 다시 모델링한 것이라는 의미에서 메타모델(model of model)이라 한다. 메타모델은 각 다이어그램들과 이를 구성하는 요소들을 일반화하여 정의함으로써, 이들 간의 연관관계를 표현하고 다이어그램들에 대한 이해도를 높이기 위하여 사용된다. 메타모델들의 세부적인 분석과 비교평가는 5장에서 제시한다.

이러한 메타모델은 여러 연구에서 제시하고 있다. OMG에서 제시하고 있는 UML설계 및 이해를 위한 메타모델[18,19], OMT, OOSE와 같은 객체지향 방법론을 작성하는 다이어그램을 기준으로 비교하기 위한 메타모델[10], 아키텍처를 중심으로 새로운 설계 시각을 제시하기 위한 메타모델[17], 그리고 일관성 점검을 위한 메타모델[4] 등이 있다.

2.2 오류 검출 및 일관성 점검 방법

오류 검출 및 일관성 점검은 시스템 개발에 있어서 분석 및 설계 단계에서부터 개발될 시스템의 완전성과 일관성을 확보하기 위하여 반드시 필요한 작업이라고 할 수 있다.

오류 검출 및 일관성 점검 방법에 관한 많은 연구들이 수행되어 왔으며, 또한 수행되고 있다. 요구공학에서 모듈라 페트리 넷(Modular Petri Net)을 사용하여 사용사례를 중심으로 시스템의 완전성과 일관성을 점검하는 방법[21], 산출물의 정보로부터 추론규칙을 도출하여 지식베이스를 구축하고 이를 이용하여 점검하

는 방법[1,5], 모델 제약 언어(MCL : Model Constraint Language)를 이용하여 객체 모델의 일관성을 점검하는 방법[2], 그리고 다이어그램을 구성하는 요소들의 연관관계를 규칙화하여 일관성을 점검하는 방법[3] 등이 있다.

오류 검출 및 일관성 점검 방법에 대한 세부적인 분석과 비교평가는 5장에서 제시한다.

2.3 정형화 기법에 의한 모델 검사

정형화 기법을 사용하여 설계정보의 무결성을 점검하기 위해서는 시스템이 동작할 환경, 시스템에 대한 요구사항, 시스템에 대한 설계정보 등을 표현하는 정형명세와 정형명세를 분석하여 설계상의 오류를 찾아내고, 일관성을 점검하는 정형검증의 두 작업으로 이루어진다. 정형명세는 시스템의 기능적 행위, 시간적 행위, 성능 특성, 내부 구조와 같은 속성을 간략한 수학적인 모델로 표현하는 방법이다. 현재 사용되는 정형 명세 방법에는 Z[7,8], LOTOS[6], VDM[12], Larch[11] 등이 있다. 이런 정형명세 언어들은 순차 시스템의 행위적인 측면에 대한 명세를 잘 지원하고 있지만, 시스템 전체에 대한 명세를 할 수 없으며, 추상화된 형태의 정형명세가 실제 구현된 시스템과는 차이가 있을 수 있다는 단점이 있다.

정형화 기법을 이용하여 시스템을 검증하기 위한 방법은 모델 검사(model checking)와 정리 증명(theorem proving)의 두 가지로 구분할 수 있다[9]. 모델 검사는 시간적인 논리 모델을 만들고 Buchi 오토마타로의 변환을 이용하여 모델이 유한 상태를 가지는지를 검사한다. 이 방식은 UML의 경우 모든 다이어그램을 시간적인 논리 공식으로 변환해야 하며, 시스템의 정적, 기능적인 측면에 대한 검사는 수행하지 못한다는 단점을 가지고 있다. 정리 증명은 시스템과 시스템에 대한 요구사항의 속성을 술어논리식 형태로 표현하고 이를 수학적인 추론규칙을 이용하여 증명하는 방식으로 시스템에 대한 일관성을 점검한다. 하지만 검사를 위해서 시스템의 구성요소에 대한 수학적인 모델을 반드시 생성해야하며, 증명에 걸리는 시간을 예측하기가 어렵고, 자동화가 힘들다는 단점이 있다.

2.4 시나리오 작성방법

분석과 설계에서 오류를 줄이고 일관성을 유지하기 위하여 시나리오의 사용을 제시하는 방법으로, 요구분석시에 편협된 응용영역에 대한 이해를 방지하기 위하여 제시된 방법이다[20]. 여기서는 유효한 시나리오 작성방법과 시나리오를 표현하는 다양한 방법을 제시하고 있지만, 이 방법을 이용하여 다시 시나리오를 작성하는데 노력이 많이 든다. 하지만, 이 방법은 사전에 분석 및 설계 과정에서 오류를 최소화하기 위한 방법이며, 개발된 분석 및 설계 문서에 대한 오류 검출 및 일관성 점검에는 유용하지 못하다.

3. 오류 검출 및 일관성 점검

앞에서도 기술한 바와 같이 개발 초기 단계에서 정확한 개발 산출물을 작성하는 것은 쉽지 않은 일임에도 불구하고 소프트웨어의 개발에 있어 매우 중요한 사항임에 틀림없다. 정확한 개발 산출물의 작성을 위해서는 개발 산출물들의 핵심이 되는 개발할 소프트웨어의 설계 모델 내에서 존재하는 오류를 검출하고, 설계 모델들 간에 존재하는 일관성을 점검하는 작업이 반드시 필요하다.

UML에 정의된 모델들은 다이어그램의 형태로 표현된다. 다이어그램들을 구성하는 요소들에 대한 표현방법에 해당하는 문법적 정의와 의미론적 정의(semantic definition)를 활용하면 오류 검출과 일관성 점검에 사용할 규칙을 식별하는데 크게 도움이 된다. 따라서 다이어그램들을 구성하는 요소들 간의 관계를 메타모델로 표현하면 이 메타모델로부터 규칙을 도출하는 체계적인 방법을 마련할 수 있다. 이러한 메타모델로부터 문법적 정의와 의미론적 정의를 점검할 수 있는 일반화된 규칙을 도출하고, 일반화된 규칙들로부터 다시 세부적인 규칙들을 마련한다. 이러한 세부적인 규칙들을 적용하여 점검할 수 있는 자동화 도구를 마련하여 개발자들이 작성한 설계 모델을 점검하고, 정확성을 유지할 수 있게 한다.

3.1 UML 다이어그램의 메타모델

UML은 개발할 소프트웨어를 분석하고 설계하는데 사용되는 모델링언어로 OMG(Object Management Group)에서 표준으로 채택하였다. UML은 개발할 소프트웨어를 보는 관점에 따라 다른 요소들과 다른 규칙들을 사용하는 9개의 다이어그램을 제시하고, 이들 다이어그램을 작성하는 것으로 개발할 소프트웨어를 표현하며, 이들을 바탕으로 소프트웨어의 개발 공정을

수행할 수 있도록 한다. UML은 현재 객체지향 개발 방법에서 사용되는 개발 소프트웨어의 표기법으로 표준화되었으며, 그 영향력이 갈 수록 커지고 있다. UML의 메타모델은 다이어그램들을 구성하는 요소들의 계층적 관계를 통하여 다이어그램들의 문법적 정의를 표현하는데 중점을 두고 있다. 이에 반해 다이어그램들을 구성하는 요소들 간의 연관관계에 대한 표현은 많지 않아서 메타모델을 통하여 오류 검출 규칙이나 일관성 점검 규칙을 도출하기에는 어려움이 많다.

여기에서 제시하는 메타모델은 다이어그램을 구성하는 요소들의 계층적인 관계보다는 그들간의 연관관계에 초점을 맞추어 작성함으로써, 오류 검출 및 일관성 점검 규칙을 도출하기에 유용하도록 작성하였다. 또한, UML의 각 다이어그램을 그들을 구성하는 요소들의 집합으로 표현하고, 같은 다이어그램에서의 요소들간의 관계는 물론 다른 다이어그램을 구성하는 요소들 중 연관관계가 깊은 것들도 표현하였다.

UML의 문법과 의미는 97년 9월 공개된 UML 1.1 표준에 정의되어 있다. 그러나, UML은 필요한 경우 사용자의 임의적인 확장을 허용하고 있어 현재 나와있는 UML 관련자료[13,16]들마다 표준에는 없는 새로운 항목들을 사용하기도 하며, 표준에 나타나는 다이어그램을 구성하는 요소들의 표기법 및 명칭들을 약간씩 다르게 사용하는 경우도 있다. 제시하는 메타모델은 OMG에서 채택한 UML 1.1 표준을 기준으로 작성하며 표준에 제시되지 않는 항목들은 수용하지 않는 것을 기본 원칙으로 하였다. UML에 정의된 다이어그램은 다음과 같다.[14,15]

- 사용사례도(Use Case Diagram)
- 상태도(State Diagram)
- 클래스도(Class Diagram)
- 활동도(Activity Diagram)
- 객체도(Object Diagram)
- 컴포넌트도(Component Diagram)
- 순차도(Sequence Diagram)
- 전개도(Deployment Diagram)
- 협력도(Collaboration Diagram)

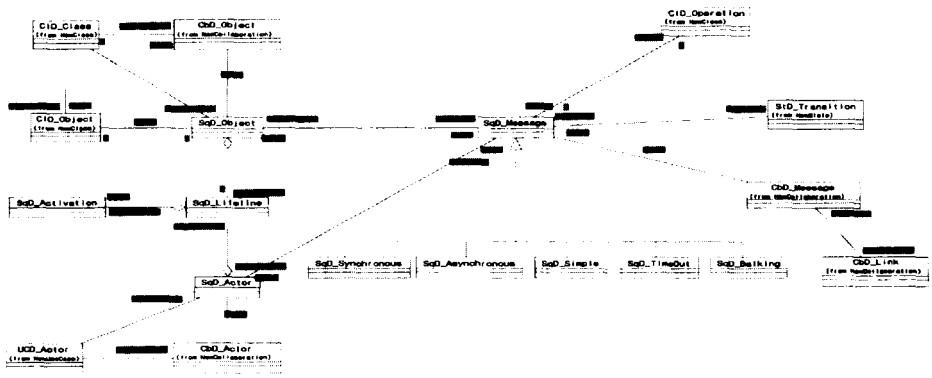
이들 각각에 대한 메타모델을 작성하였다. 메타모델은 UML의 클래스도의 표기법을 이용하여, UML의 각 다이어그램을 구성하는 요소들을 클래스로 표현하고 이들의 관계를 표시하였다. 메타모델상에서 연관관계(association)에 있는 요소들은 연관명(association name), 다중성(multiplicity), 역할명(role name)을 통하여 그들의 연관관계가 표현된다. 집단화(aggregation)관계는 하위 요소들의 집합을 통하여 상위 요소가 구성됨을 표현한다. 일반화(generalization)관계는 하위 요소들이 상위 요소들로부터 상속관계(inheritance)를 가짐을 의미한다. 제시하는 메타모델은 이렇게 UML 다이어그램들을 구성하는 요소들의 관계를 중심으로 표현된다.

이후의 내용에서는 설계 다이어그램들 중 소프트웨어 개발에 핵심이 되는 순차도와 클래스도를 중심으로 설명하고, 다른 다이어그램의 메타모델은 지면관계상 간략히 설명한다. 또한, 순차도와 클래스도의 메타모델에 표현된 요소들의 의미적 특징을 <표 1>과 <표 2>로 정리하였다.

사용사례도를 구성하는 요소는 사용사례(use case), 연관관계(association), 액터(actor), 일반화관계이다. 사용사례는 여러 개의 시나리오를 가지며, 각 시나리오에 대한 순차도와 협력도를 가진다. 사용사례 간, 또는 액터 간의 관계는 반드시 일반화관계로 나타나며 사용사례 사이의 일반화관계는 <<uses>>나 <<extends>> 중 한가지로 제한된다. 액터와 사용사례 간의 관계는 연관관계(<<communication>>)로 나타난다. 또한, 사용사례도의 액터와 동일한 액터가 순차도와 협력도에 존재한다.

순차도를 구성하는 요소는 객체, 액터, 메시지(message), 생명선(lifeline), 활성(activation), 전이시간(transition time), 반복(iteration), 조건(guard condition)이 있다. 메시지는 동기화특성과 응답대기의 특성에 따라 동기(synchronous) 메시지, 비동기(asynchronous) 메시지, 단순(simple) 메시지, 타임아웃(timeout) 메시지, 보킹(balking) 메시지로 나뉜다. 생명선은 객체나 액터의 인스턴스가 생성되는 순간부터 소멸되는 순간까지 존재하며, 한 객체나 액터당 반드시 하나의 생명선이 나타난다. 활성은 생명선이 존재하는 동안에만 나타날 수 있으며, 메시지를 수신할 때 생성된다. 순차도의 객체는 클래스도의 클래스에 대응한다. 하나의 메시지는 클래스도의 대응하는 연산(operation)과 연관관계를 가진다.

다음의 (그림 1)에서 SqD로 시작하는 요소들은 순차도를 구성하는 요소들이며, ClD는 클래스도, CbD는 협력도, UCD는 사용사례도, StD는 상태도를 구성하는 요소들이다.



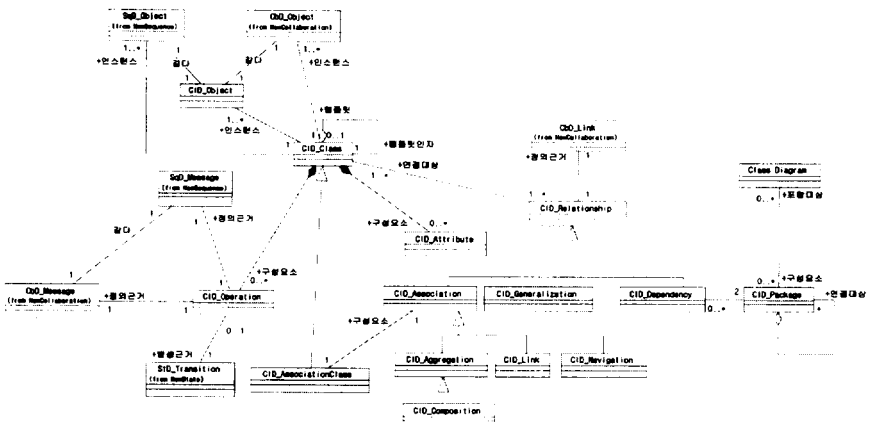
(그림 1) 순차도의 메타모델

협력도를 구성하는 요소는 객체, 액터, 메시지, 링크(link)이다. 메시지의 특성은 조건(guard condition), 반복(iteration), 전이시간(transition time) 등을 통하여 한정된다. 협력도의 액터, 객체, 링크는 각각 순차도의 액터, 객체, 메시지의 전송으로 나타나게 된다. 링크는 객체와 객체 사이 혹은 객체와 액터 사이를 연결하며, 링크를 통하여 객체로 메시지가 전송된다. 링크로 연결된 객체들의 클래스는 클래스도에서 관계(relationship)로 연결된다.

클래스도의 메타모델을 구성하는 요소는 클래스(class), 객체, 인터페이스(interface), 연산(operation), 속성(attribute), 관계(relationship), 패키지(package), 연관클래스(association class) 등이 있다. 관계는 그 특성에 따라 일반화관계, 연관관계, 의존관계(dependency)로 나뉘며, 이 들은 다시 메타모델과 같이 세부적으로 나뉜다. 연관클래스는 연관관계가 속성들을 갖게 될 때, 이를 클래스로 표시한 것이다. 템플릿(template)은 클래스의 한 종류이며, 객체는 순차도와 협력도의 객체를 기반으로 생성되며 클래스가 인스턴스화한 것이다. 연산과 속성은 반드시 클래스에서만 나타나며, 연산은

〈표 1〉 순차도를 구성하는 요소들의 의미적 특징

다이어그램을 구성하는 요소	각 요소들의 의미적 특징
SqD_Message	1. SqD_Message는 순차도의 메시지를 표현한다.
	2. SqD_Message는 CID_Association, CID_Aggregation, 혹은 CID_Composition과 대응한다.
	3. SqD_Message로 연결된 SqD_Object와 SqD_Actor들은 협력도에서 링크로 연결된다.
	4. SqD_Message는 클래스도의 CID_Operation과 대응한다.
SqD_Object	1. SqD_Object는 순차도의 객체를 표현한다.
	2. SqD_Message를 받는 SqD_Object는 그에 대응하는 CID_Operation을 가지고 있어야 한다.
	3. SqD_Message를 주고 받는 SqD_Object들의 클래스는 클래스도에서 CID_Association, CID_Aggregation, 혹은 CID_Composition으로 연결된다.
	4. SqD_Message를 주고 받는 SqD_Object 들은 협력도에서 링크로 연결된다.
SqD_Lifeline	1. SqD_Lifeline은 순차도의 생명선을 표현한다.
	2. SqD_Lifeline은 SqD_Object 인스턴스가 생성되는 순간부터 시작된다.
	3. SqD_Lifeline은 SqD_Object 인스턴스가 소멸되는 시점에서 끝난다.
	4. SqD_Object는 SqD_Lifeline이 존재하는 동안에만 SqD_Message 송수신이 가능하다.
	5. SqD_Actor의 SqD_Lifeline은 생성과 소멸 없이 항상 존재한다.
SqD_Actor	1. SqD_Actor는 순차도에서 액터를 표현한다.
	2. SqD_Actor는 사용사례도의 UCD_Actor, 협력도의 CbD_Actor와 동일하다.
	3. SqD_Actor에 연결된 SqD_Message는 협력도에서 링크로 연결된다.
	4. SqD_Actor간에는 SqD_Message를 주고 받지 않는다.
SqD_Activation	1. SqD_Activation은 순차도에서 SqD_Object의 활성을 표현한다.
	2. SqD_Activation은 SqD_Lifeline이 존재하는 동안에만 생성된다.
	3. SqD_Activation은 처음 메시지를 수신할 때 생성된다.



(그림 2) 클래스도의 메타모델

〈표 2〉 클래스도를 구성하는 요소들의 의미적 특징

다이어그램을 구성하는 요소	각 요소들의 의미적 특징
CID_Class	1. CID_Class는 클래스도의 클래스를 표현한다.
	2. CID_Class는 CID_Operation과 CID_Attribute를 포함하고 있다.
	3. 하나의 CID_Class는 여러 개의 CID_Object를 인스턴스로 생성시킨다.
	4. CID_Class는 CID_Relationship을 통해 다른 CID_Class와 연관관계를 갖는다.
	5. CID_Class는 템플릿의 형태로 표현될 수 있다.
	6. CID_Class의 이름은 유일해야 한다.
CID_Object	1. CID_Object는 클래스도의 객체를 표현한다.
	2. CID_Object는 순차도와 협력도에 SqD_Object, CbD_Object로 존재한다.
	3. CID_Object는 CID_Class의 인스턴스로 존재한다.
CID_Operation	1. CID_Operation은 클래스도 내에 존재하며 연산을 표현한다.
	2. CID_Operation은 SqD_Message, CbD_Message와 대응한다.
	3. 하나의 StD_Transition은 하나의 CID_Operation에 대응된다.
	4. 각 CID_Class 내에서 CID_operation의 서명(signature)는 유일해야 한다.
CID_Attribute	1. CID_Attribute는 클래스의 속성를 표현한다.
	2. 각 CID_Class 내에서 CID_Attribute의 이름은 유일해야 한다.
CID_Relationship	1. CID_Relationship는 클래스도에서 클래스나 객체간의 관계를 표현한다.
	2. CID_Relationship은 CID_Generalization, CID_Association, CID_Dependency의 상위 클래스 이다.
	3. CID_Relationship은 서로 다른 클래스들에 연결관계를 부여한다.
	4. CID_Relationship으로 연결된 클래스들의 인스턴스들 간에 SqD_Message혹은 CbD_Message을 주고 받을 수 있다.
	5. CID_Relationship은 반드시 대응하는 CID_Class나 CID_Package가 있다.
CID_Generalization	1. CID_Generalization은 클래스들간의 일반화관계를 표현한다.
	2. CID_Generalization의 상위 클래스와 하위 클래스는 서로 다른 클래스이어야 한다.
	3. CID_Generalization은 순환될 수 없다.
CID_Association	1. CID_Association은 클래스들간의 연관관계를 표현한다.
	2. CID_Association은 CID_Aggregation의 상위 클래스 이다.
	3. CID_Aggregation은 CID_Composition 의 상위 클래스 이다.
CID_Dependency	1. CID_Dependency는 클래스도에서 CID_Package들 사이의 의존관계를 표현한다.
	2. CID_Dependency는 서로 다른 패키지간을 연결한다.
CID_Package	1. CID_Package는 클래스도의 패키지를 표현한다.
	2. CID_Package는 클래스도를 포함할 수 있으며 클래스도에 표현될 수 있다.
	3. CID_Package는 CID_Package를 포함할 수 있다.
	4. CID_Package의 이름은 유일해야 한다.
CID_AssociationClass	1. CID_AssociationClass는 CID_Association에 나타나는 연관클래스를 표현한다.

순차도의 메시지로부터 정의된다. 또한 연산은 상태도의 상태를 변환하는 근거가 되며 전이를 발생시키는 근거가 된다. 협력도의 링크로부터 정의되는 관계는 연관에는 집단화와 합성 등의 종류를 갖는다. 패키지는 의존(dependency)을 통해 연결하며, 패키지 내에 새로운 클래스도를 포함할 수 있다.

상태도의 메타모델을 구성하는 요소는 상태(state), 의사상태(pseudo state), 전이(transition), 조건, 행동(action), 행동순서(action sequence), 사건(event) 등이 있다. 상태는 복합상태(composite state), 부상태(sub state), 단순상태(simple state)가 있는데, 복합상태는 부상태를 포함하며, 부상태는 내부에 새로운 상태도를 포함할 수 있다. 의사상태는 시작상태(initial state), 종료상태(final state), 결정(decision), 동기화 바(synchronization bar) 등이 있다. 전이는 클래스도의 연산에 의해 발생하며, 사건과 조건, 행동순서를 포함하며 상태간을 연결한다.

활동도의 메타모델을 구성하는 요소는 행동상태(action state), 의사상태, 전이, 객체, 객체흐름(object flow), 조건, 행동, 스윔레인(swimlane), 조작아이콘(control icon)이 있다. 의사상태는 결정, 시작상태, 종료상태, 동기화 바 등으로 나뉜다. 전이는 행동과 조건, 조작아이콘을 통하여 그 특성이 제한되며 상태의 전이를 나타낸다. 조작아이콘은 신호 전송자, 신호 수신자로 나뉜다. 스윔레인은 주로 업무 프로세스(business process)를 설계하는데 사용되는 것으로, 활동상태들을 그 것을 수행하는 객체, 액터, 조직별로 구분한다. 객체흐름은 활동상태가 객체의 상태를 생성, 소멸, 혹은 변화시키는 제어의 흐름을 표시한다.

컴포넌트도의 메타모델을 구성하는 요소는 컴포넌트(component), 인터페이스, 의존관계가 있다. 인터페이스는 컴포넌트에서만 나타나며 의존은 컴포넌트와 인터페이스를 연결한다.

전개도를 구성하는 요소는 패키지, 객체, 노드, 의존관계로 구성되며, 필요한 경우 컴포넌트도의 컴포넌트가 나타나기도 한다. 전개도에서 노드, 객체, 패키지 사이의 관계는 의존관계를 통해 나타난다. 노드는 다른 노드나 컴포넌트도의 컴포넌트를 포함할 수 있다.

3.2 일반화된 규칙

일반화된 규칙은 메타모델로부터 도출된 다이어그램을 구성하는 요소들의 의미적 특징들을 구분하는 작업으로부터 시작한다. 즉, 유사한 종류의 의미적 특징들을 일반화된 규칙들로 묶고, 묶여진 의미적 특징들을 규칙의 형태로 작성하여 일반화된 규칙을 생성한다. 생성된 일반화된 규칙들을 메타모델을 구성하는 요소들 중 적용가능한 요소들에 적용하여 세부적인 규칙들을 도출해낸다. 이렇게 일반화된 규칙을 작성하여 세부 규칙들을 도출해내는 이유는 특정 요소들의 의미적인 특징들을 다른 요소들에도 적용하기 위함이다. 즉, 다이어그램을 구성하는 요소들의 의미적인 특징이 모두 도출되지 않았다는 가정하에 요소들의 의미적인 특징들은 상호보완하면서 새로운 세부규칙의 작성을 돕게 된다. 이러한 일반화된 규칙을 메타규칙이라고 부르기로 한다.

메타규칙 1 : 하나의 다이어그램을 구성하는 요소들 각각의 이름은 유일해야 한다.

각 다이어그램에서 나타나는 사용사례, 클래스, 객체, 패키지, 액터, 상태, 활동, 컴포넌트, 노드 등의 개체의 이름은 다이어그램 내에서 유일해야 한다. 이러한 개체는 개발하려는 소프트웨어의 일부분을 나타내는 것으로, 관점과 추상화 정도에 따라 설계 산출물에서 다른 형태로 나타난다. 같은 이름이 중복되어 사용됨은 설계 산출물상에서의 개발자와 사용자의 혼란을 유발시킴은 물론 결국 개발된 소프트웨어의 오동작의 원인이 된다. 개체들의 이름은 유일하여야 하나, 개체간의 관계나 개체간에 주고 받는 메시지는 동일한 관계가 발생할 수 있으므로, 관계 및 메시지에서는 이름의 유일성이 제약되지 않는다.

메타규칙 2 : 다이어그램을 구성하는 요소는 다른 요소와 관계를 맺지않고 홀로 존재할 수 없다.

메타모델상에서 존재하는 모든 요소들은 다른 요소들과 연관관계를 갖는다. 즉, 독립적으로 존재하는 요소는 존재하지 않는다. 다른 요소들과 연관관계가 없는 요소가 존재하는 것은 문법적인 오류를 발생시키지는 않으나, 이러한 요소는 불필요한 요소가 들어간 경우일 것이며 설계 문서상의 의미적인 오류를 발생시킬 것이다. 그러므로, 개발자들은 설계 문서의 작성시 독립적으로 존재하는 요소가 없도록 설계 작업을 수행하고, 이러한 부분이 있는지 추가적인 점검을 수행하여야 한다.

메타규칙 3 : 각 다이어그램을 구성하는 요소는 해당

다이어그램의 문법적인 특성을 유지하여야 한다.

UML의 모든 다이어그램은 문법적인 특성을 가지고 있다. 문법적인 특성이란 다이어그램을 구성하는 요소들의 종류와 표현법이라고 간단히 말할 수 있다. 다이어그램을 구성하는 요소들의 종류와 표현법은 UML표준의 의미(semantics)와 표기법(notation)을 통하여 자세히 기술되며, 이러한 문법적인 특성들은 앞에서 제시한 각 다이어그램의 메타모델에 표현되었다. 메타모델이 클래스도의 표기법을 사용함으로써 발생하는 한계로 표시되지 않은 몇몇 문법적인 특성은 요소들의 의미적인 특성을 기술하면서 추가하였다. 문법적인 특성은 대부분 모든 다이어그램에 적용되기 보다는 각 다이어그램마다 문법적인 특성을 가진다.

메타규칙 4 : 메타모델상에서 "같다" 관계를 가진 요소들은 서로 다른 다이어그램에서 동일한 요소로 존재한다.

설계 산출물상에서 등장하는 다이어그램을 구성하는 요소들 중 같은 요소가 여러 다이어그램에 등장하는 경우가 있다. 이는 UML의 다이어그램들이 독립적으로 존재하지 않으며, 같은 내용이 여러 다이어그램에서 표현되기 때문이다. 이러한 경우 이런 요소들은 동일한 요소이어야 한다. 이러한 요소들은 다이어그램의 메타모델상에서 "같다"라는 연관관계를 가진다.

메타규칙 5 : 메타모델에 서로 다른 다이어그램 내의 요소들이 연관을 맺고 있는 경우 한 쪽 다이어그램의 요소는 다른 쪽 다이어그램의 요소의 특성을 제약한다.

다이어그램을 구성하는 요소들은 대부분 다른 다이어그램을 구성하는 요소들과 관계를 갖게 된다. 이는 UML이 제공하는 각 다이어그램이 서로 별개의 설계 문서가 아니라 개발할 소프트웨어를 서로 다른 관점에서 본 설계 문서이기 때문이다. 즉, 대상의 상이함이 아니라 관점 및 표현 방법의 상이함일 뿐이므로, 다이어그램간의 관계는 존재하게 된다. 특히, 설계단계에서 핵심 대상인 클래스 및 객체와 그 들의 속성, 연산, 메시지 등은 여러 다이어그램에 걸쳐 다양한 형태로 표현되며, 상호 검증된다. 이러한 상호 연관관계는 각 다이어그램의 메타모델에 표시되었다. 연관관계에 있는 요소들은 메타모델상에 표현된 각자의 역할명(role name)에 따른 특성을 갖게 되며 서로를 제약하는 요소가 된다. 그리고, 다이어그램을 구성하는 요소A와 요소B가 관계가 있고, 요소B와 요소C가 관계가 있을 때 A와 C 또한 관계를 갖는다.

3.3 세부 규칙

메타모델로부터 도출된 각 요소들에 대한 의미적인 특징으로부터 일반화된 규칙을 도출하여 이를 메타규칙이라는 이름으로 제시하였다. 도출된 메타규칙은 일반적인 규칙으로 추상화 되어있기 때문에, 오류 검출과 일관성 점검을 위하여 설계 문서에 적용하기에는 무리가 있다. 이러한 일반화된 규칙들을 실제 설계 문서에 적용가능 하도록 하기 위하여 세부적인 규칙으로 작성한다. 세부 규칙은 일반화된 메타규칙을 다이어그램을 구성하는 요소들에 적용해봄으로써 획득된다. 다이어그램을 구성하는 요소들에 메타규칙을 적용함에 있어, 앞에서 제시한 메타모델을 구성하는 요소들의 의미적인 특징은 물론 해당 요소들이 갖는 문법적인 특징들도 반영하도록 한다.

이러한 방법으로 다이어그램을 구성하는 요소들에 적용된 세부적인 점검 규칙을 마련하였다. 여기서는 순차도와 클래스도의 메타모델에 표현된 요소들에 메타규칙을 적용하여 도출한 세부규칙들을 제시하였다.

메타규칙 1, 메타규칙 2와 메타규칙 3은 메타모델에서 단일 다이어그램에만 나오는 요소들간에 적용하여 세부규칙을 찾아 낸다. 즉, 순차도에 적용했을 때 메타규칙 1, 메타규칙 2와 메타규칙 3은 순차도를 구성하는 요소간의 오류를 검출하는 규칙이다. 메타규칙 4와 메타규칙 5는 메타모델에서 해당 다이어그램과 연관된 다른 다이어그램의 요소를 포함하여 일관성을 점검하는 세부규칙을 찾아 낸다.

3.3.1 순차도에 대한 세부규칙들

메타규칙 1은 하나의 다이어그램을 구성하는 요소들 각각의 이름은 유일하다는 것이다. 메시지는 반복되는 경우는 물론, 동일한 것이 얼마든지 발생할 수 있으므로, 이름의 유일성을 필요로 하지 않는다. 또한 메시지로부터 상속된 동기 메시지, 비동기 메시지, 단순 메시지, 타임아웃 메시지 및 보킹 메시지들도 이름의 유일성이 필요하지 않다. SqD_Lifeline과 SqD_Activation은 이름을 갖지 않기 때문에 메타규칙 1에 적용되지 않는다. 순차도에 메타규칙 1을 적용하여 도출한 세부규칙은 다음과 같다.

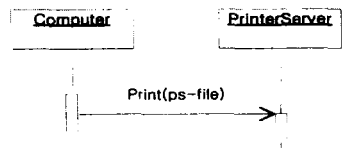
SqD_Rule1 : SqD_Object의 이름은 유일하다.
SqD_Rule2 : SqD_Actor의 이름은 유일하다.

메타규칙 2는 다이어그램을 구성하는 요소는 다른 요소와 관계를 맺지않고 홀로 존재할 수 없다는 것이다. 즉 모든 요소들은 반드시 하나 이상의 다른 요소와 관계를 갖는다. 서로 관계를 갖는 요소들은 메타모델상에서 클래스들간의 관계로 표현하였다. 메타모델상에서도 아무 관계도 갖지 않는 요소들은 존재하지 않는다. 순차도의 액터와 객체는 하나 이상의 메시지와 연결되어야 하며, 메시지는 액터들과 객체들간의 통신방법이 된다. 순차도에 메타규칙 2를 적용하여 도출한 세부규칙은 다음과 같다.

SqD_Rule3 : SqD_Object는 SqD_Message에 의해 연결된다.
SqD_Rule4 : SqD_Actor는 SqD_Message에 의해 연결된다.
SqD_Rule5 : SqD_Message는 SqD_Object나 SqD_Actor에 의해 생성되고 수신된다.

메타규칙 3은 각 다이어그램을 구성하는 요소는 해당 다이어그램의 문법적인 특성을 유지하여야 한다는 것이다. 순차도의 경우 액터는 실세계의 사람이고 객체는 구동중인 소프트웨어의 일부이기 때문에, 이들은 메시지를 송수신하는데 있어서 서로 다른 특징을 갖는다. 이러한 특성들은 각 다이어그램에서 갖는 독특한 특성들로 UML의 문법적인 특성이라고 할 수 있다. 순차도에 메타규칙 3을 적용하여 도출한 세부규칙은 다음과 같다.

SqD_Rule6 : SqD_Message는 송수신하는 SqD_Object나 SqD_Actor의 SqD_Lifeline에 연결된다.
SqD_Rule7 : SqD_Message를 송신하는 시점의 SqD_Object는 SqD_Lifeline에 SqD_Activation이 존재한다.
SqD_Rule8 : SqD_Object에서는 SqD_Message 수신시 요구에 적합한 SqD_Activation이 생성된다.
SqD_Rule9 : SqD_Actor간에는 SqD_Message로 연결되지 않는다.
SqD_Rule10 : SqD_Activation은 SqD_Lifeline의 생존기간 내에서 생성 및 소멸된다.
SqD_Rule11 : SqD_Message를 수신하는 SqD_Object는 그에 대응하는 ClD_Operation을 가지고 있어야 한다.



(그림 3) 순차도 예제

위의 (그림 3)의 순차도 예제와 같이 메시지 print (ps-file)은 활성이 존재하는 중에 발생하여 다른 객체로 전달되며 생명선 사이를 연결한다. 메시지 print (ps-file)를 전달받은 객체 PrintServer는 메시지의 요구에 부합하는 작업을 수행하기 위하여 즉시 활성이 생성된다. 또한, 액터간에는 메시지를 주고 받지 않는다.

메타규칙 4는 메타모델상에서 "같다" 관계를 가진 요소들은 서로 다른 다이어그램에서 동일한 요소로 존재한다는 것이다. 메타모델에서 연관관계의 이름이 "같다"로 정의된 부분을 조사하여 세부규칙을 도출한다. 하나의 다이어그램에 동일한 요소가 중복되어 나타나지는 않으나 동일한 요소가 다른 다이어그램에 나타나게 된다. 순차도에 메타규칙 4를 적용하여 도출한 세부규칙은 다음과 같다.

SqD_Rule12 : SqD_Object와 대응하는 ClD_Object는 동일하다.
SqD_Rule13 : SqD_Object와 대응하는 CbD_Object는 동일하다.
SqD_Rule14 : SqD_Actor와 대응하는 CbD_Actor는 동일하다.
SqD_Rule15 : 모든 SqD_Message에 대하여 동일한 CbD_Message가 존재한다

메타규칙 5는 메타모델에 서로 다른 다이어그램 내의 요소들이 연관을 맺고 있는 경우 한 쪽 다이어그램의 요소는 다른 쪽 다이어그램의 요소의 특성을 제약한다는 것이다. 서로의 특성을 제약하는 요소들은 메타모델상에서 연관관계를 갖는다. 메타모델상에서 연관관계를 갖는 두 요소들을 찾아내고, 이들의 연관관계의 이름과 역할명을 통하여 서로의 관계 및 특성 제약을 파악할 수 있다. SqD_Object와 ClD_Object가 같고 ClD_Object가 ClD_Class와 대응하기 때문에 SqD_Object는 ClD_Class와 대응한다. 또한, SqD_Message와 CbD_Message가 같고 CbD_Message가 CbD_Link

와 대응하므로 SqD_Message와 CbD_Link도 대응한다. 순차도에 메타규칙 5를 적용하여 도출한 세부규칙은 다음과 같다.

SqD_Rule16 : 모든 SqD_Object에 대하여 대응하는 ClD_Class가 존재한다.

SqD_Rule17 : SqD_Message를 송수신하는 두 SqD_Object는 클래스도에서 그 대응하는 ClD_Class들간에 ClD_Relationship을 갖는다.

SqD_Rule18 : SqD_Message를 수신하는 SqD_Object의 클래스인 ClD_Class는 SqD_Message에 대응하는 ClD_Operation을 갖는다.

SqD_Rule19 : StD_Transition이 SqD_Message를 송신하는 경우 같은 SqD_Message가 순차도에 존재한다.

SqD_Rule20 : SqD_Message를 송수신하는 SqD_Object들은 CbD_Link로 연결된다.

3.3.2 클래스도에 대한 세부규칙들

메타규칙 1은 하나의 다이어그램을 구성하는 요소들 각각의 이름은 유일해야 한다는 것이다. 클래스도의 메타모델에서 다른 요소들의 관계를 표현하는 요소들을 제외한 클래스도의 주요요소들은 이름의 유일성이 요구된다. 클래스에 존재하는 속성과 연산의 서명(Signature)은 하나의 클래스 내에서는 이름이 유일해야 하나, 같은 이름이 다른 클래스상에는 존재할 수 있다. 클래스도에 메타규칙 1을 적용하여 도출한 세부규칙은 다음과 같다.

ClD_Rule1 : ClD_Class의 이름은 유일하다.

ClD_Rule2 : ClD_Package의 이름은 유일하다.

ClD_Rule3 : ClD_Object의 이름은 유일하다.

ClD_Rule4 : 각 Class 내에서 ClD_Attribute의 이름은 유일하다.

ClD_Rule5 : 각 Class 내에서 ClD_Operation의 서명은 유일하다.

메타규칙 2는 다이어그램을 구성하는 요소는 다른 요소와 관계를 맺지않고 홀로 존재할 수 없다는 것이다. 클래스도를 구성하는 주요요소는 클래스와 관계이다. 관계는 2개 이상의 클래스들간의 관계를 나타내며, 클래스는 관계를 통하여 다른 클래스들과 상호작용한다. 모든 클래스들은 관계를 통하여 서로 연결되어 있으며, 독립적으로 존재하지 않아야 한다. 마찬가지로, 모든 관계는 클래스들을 연결하며 독립적으로 존재하지 않아야 한다. 이는 UML에서의 문법적인 오류를 발생시키지는 않으나, 개발 소프트웨어의 설계에 의미적인 오류를 일으킨다. 클래스도에 메타규칙 2를 적용하여 도출한 세부규칙은 다음과 같다.

ClD_Rule6 : ClD_Class는 ClD_Generalization, ClD_Association, 또는 ClD_Dependency에 의해 연결된다.

ClD_Rule7 : ClD_Package는 ClD_Dependency에 의해 연결된다.

ClD_Rule8 : ClD_Association은 ClD_Class들을 연결한다.

ClD_Rule9 : ClD_Generalization은 ClD_Class들을 연결한다.

ClD_Rule10 : ClD_Dependency는 ClD_Package를 ClD_Package나 ClD_Class와 연결한다.

ClD_Rule11 : ClD_AssociationClass는 하나의 ClD_Association과 대응한다.

메타규칙 3은 각 다이어그램을 구성하는 요소는 해당 다이어그램의 문법적인 특성을 유지하여야 한다는 것이다. 클래스들간에 일반화 관계는 그 관계에 상위와 하위, 선과 후의 개념을 가지고 있으므로, 순환적이어서는 안 된다. 또한, 속성과 연산은 반드시 하나의 클래스에 포함되어야 하며, 패키지간의 관계는 의존관계만으로 한정된다. 클래스도에 메타규칙 3을 적용하여 도출한 세부규칙은 다음과 같다.

ClD_Rule12 : ClD_Operation과 ClD_Attribute는 하나의 ClD_Class에 포함된다.

ClD_Rule13 : 순환적인 ClD_Generalization은 성립하지 않는다.

ClD_Rule14 : ClD_Package는 ClD_Dependency로만 연결된다.

메타규칙 4는 메타모델상에서 "같다" 관계를 가진 요소들은 서로 다른 다이어그램에서 동일한 요소로 존재한다는 것이다. 클래스의 인스턴스인 객체는 순차도와 협력도에서도 동일한 객체가 출현하며, 이들은 각 다이어그램의 관점에 따라 다른 요소들과의 관계가 기술된다. 클래스도에 메타규칙 4를 적용하여 도출한 세부규칙은 다음과 같다.

ClD_Rule15 : ClD_Object와 대응하는 SqD_Object는

동일하다.

ClD_Rule16 : ClD_Object와 대응하는 CbD_Object는 동일하다.

메타규칙 5는 메타모델에 서로 다른 다이어그램 내의 요소들이 연관을 맺고 있는 경우 한 쪽 다이어그램의 요소는 다른 쪽 다이어그램의 요소의 특성을 제약한다는 것이다. 메타모델상에서 연관관계를 갖는 두 요소들을 찾아내고, 이들의 연관관계의 이름과 역할명을 통하여 서로의 관계 및 특성 제약을 파악하여 세부규칙을 도출한다. ClD_Operation과 연관관계를 갖는 요소들은 ClD_Operation과 합성관계에 있는 ClD_Class와도 연관관계를 갖는다. 또한, ClD_Class와 ClD_Relationship이 대응하고 ClD_Relationship이 CbD_Link와 대응하므로 ClD_Class와 CbD_Link도 대응한다. 클래스도에 메타규칙 5를 적용하여 도출한 세부규칙은 다음과 같다.

ClD_Rule17 : ClD_Class는 ClD_Relationship로 연결되며, ClD_Relationship에 명시된 관계와 역할을 갖는다.

ClD_Rule18 : ClD_Object, SqD_Object, CbD_Object에 대응하는 ClD_Class가 존재한다.

ClD_Rule19 : 순차도에서 SqD_Message를 주고 받는 객체들의 클래스들은 ClD_Association으로 연결된다.

ClD_Rule20 : ClD_Association으로 연결된 ClD_Class의 인스턴스인 SqD_Object들은 협력도에서 CbD_Link로 연결된다.

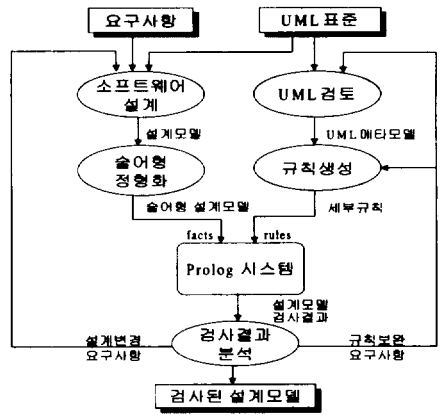
ClD_Rule21 : SqD_Message, CbD_Message와 대응하는 ClD_Operation이 존재한다.

ClD_Rule22 : 상태도의 StD_Transition에 대응하는 ClD_Operation이 존재한다.

4. 설계모델 검사의 구현

지금까지 UML로 표현된 소프트웨어 설계 모델을 검사할 때 적용할 세부규칙을 도출하는 과정과 도출된 규칙을 살펴보았다. 메타모델을 오류검출 및 일관성 점검을 위한 규칙을 찾을 수 있도록 요소들간의 연관관계와 하나의 다이어그램 내의 요소들에 한정하지 않고 여러 다이어그램들에 존재하는 요소들 간의 연관관계도 모형화하였다. 이를 바탕으로 일반화된 규칙을 도출하여 이 일반화된 규칙을 모든 다이어그램 내의 요소에 적용을 하여 규칙을 완전성에 접근하였다.

이러한 규칙들은 UML표준을 준용하여 만들어진 설계 모델이라면 어느 프로젝트의 모델이라도 상관하지 않고 적용할 수 있다. 그런데 설계모델 검사를 실제로 구현하기 위해서는 이들 규칙을 컴퓨터에서 표현하는 방법을 필요로 한다. 또한 설계 모델 자체도 규칙의 표현방법과 연관되어 효과적인 표현방법이 필요하다. UML에 의한 표현은 그래픽 표현이므로 이를 컴퓨터에 저장할 때에 내부 표현의 정형화는 규칙의 표현과 아울러 검사를 효과적으로 수행하는 요건이 된다.



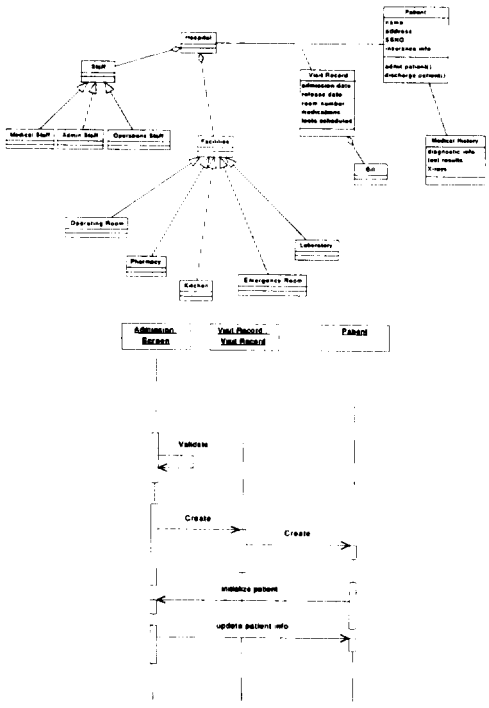
(그림 4) 검증 모델

정형명세언어를 표현하는 방법은 2장에서 거론한 바와 같이 설계 모델의 오류 검출과 일관성 점검에는 아직 기술적인 문제점이 남아 있으므로 여기에서는 (그림 4)에서와 같이 Prolog언어를 활용하여 규칙기반시스템으로 설계모델 검사를 구현하였다[1]. (그림 4)는 설계모델 검사과정을 표현한 것으로 UML 메타모델과 세부규칙 생성과정은 프로젝트마다 수행하지 않고 처음 세부규칙을 마련한 때와 여러 프로젝트에서 세부규칙 적용결과를 분석하여 규칙보완을 필요로 할 때만 수행한다.

Prolog 시스템을 활용하기 위해 설계모델과 세부규칙은 술어(predicate)로 표현한다. 클래스를 술어로 표현하면 class(클래스명, [속성], [연산])이 되어, 클래스임을 선언하는 술어명 class에 클래스명, 속성, 및 연산을 인수(parameter)화 한다[1].

(그림 5)는 Rational사의 CASE 도구 제품인 Rational-

Rose98을 사용하여 작성한 예제인 Hospital의 클래스도와 순차도를 변형하여 순환적 상속을 포함되도록 오류를 삽입한 것이다. 이 클래스도와 순차로를 Prolog코드로 표현하면 (그림 6)과 같다. (그림 6)에는 클래스와 연관클래스, 집단화, 일반화, 및 연관들이 나타나 있으며, (그림 5)의 클래스도의 정보를 모두 포함한다.



(그림 5) Hospital 예제 클래스도 및 순차도

```
% 클래스도의 검사모델.

class(hospital, _, _).
class(staff, _, _).
class(medicalstaff, _, _).
class(adminstaff, _, _).
class(operationsstaff, _, _).
class(facilities, _, _).
class(operationroom, _, _).
class(pharmacy, _, _).
class(kitchen, _, _).
class(emergencyroom, _, _).
class(laboratory, _, _).
class(patient, [name, address, ssno,insurenceinfo], [admit-
    patient, dischargepatient]).
class(medicalhistory, [diagnosticinfo, testresults,xrays], _).
associationclass( visit, visitrecord, [admissiondate, releasedate,
    roomnumber, medications, testsscheduled],_).
aggregation([medicalstaff], staff).
aggregation([adminstaff], staff).
aggregation([operationsstaff], staff).
aggregation([facilities], hospital).
aggregation([bill], visitrecord).
generalization([operationroom], facilities).
generalization([pharmacy], facilities).
generalization([kitchen], facilities).
generalization([emergencyroom], facilities).
generalization([laboratory], facilities).
association(visit, patient, hospital).
association(has, patient, medicalhistory).

% 순차도의 검사모델

sequencediagram(admitpatient).
object(admitpatient, admissionscreen).
object(admitpatient, visitrecord).
object(admitpatient, patient).
message(admitpatient, _, admissionscreen, validdate, admis-
    sionscreen ).
message(admitpatient, _, admissionscreen, create, visitrecord).
message(admitpatient, _, visitrecord, create, patient).
message(admitpatient, _, patient, initializepatient, admis-
    sionscreen).
message(admitpatient, _, admissionscreen, updatepatientinfo,
    patient).
```

(그림 6) Hospital 예제의 클래스도 정형화

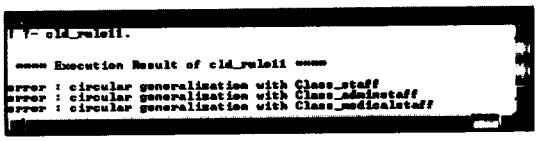
(그림 7)은 클래스간의 일반화 관계가 순환되지 않는다는 세부규칙을 prolog로 표현한 것이다.

```
cld_rule11 :-
  nl,nl, write(' **** Now execution of cld_rule11 **** '), nl,nl,
  retractall(cld_rule11_error),
  generalization(SubClassesId, SuperClassId),
  traverseGeneralization(SubClassesId, SuperClassId, []),
  fail.
cld_rule11 :-
  not(cld_rule11_error),
  write('Generalization hierachies have no circular structure, are
      good'), nl .
cld_rule11.
traverseGeneralization([SubClass|SubClasses], SuperClassId,
      ClassIdList) :-
  SubClass = SuperClassId,
  assert(cld_rule11_error),
  write('error : circular generalization with Class_'), write
      (SuperClassId), nl, !
;
generalization(SubSubClasses, SubClass),
not(member(SubClass, ClassIdList) ),
append([SubClass], ClassIdList, NewClassIdList),
traverseGeneralization(SubSubClasses, SuperClassId, NewClassIdList)
;
    traverseGeneralization(SubClasses, SuperClassId, ClassIdList).
```

(그림 7) 세부규칙의 Prolog 코드 사례

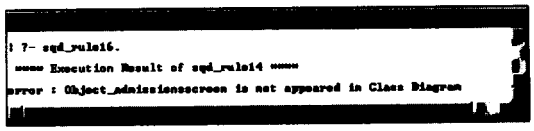
모델을 정형화 내용과 세부규칙의 Prolog 코드를 함께 Consult하여 다음의 결과를 얻을 수 있다. (그림 8)과 (그림 9)는 (그림 6)과 (그림 7)을 적용하여 검사를

수행한 결과를 나타낸 것이다. 일반화 관계에서 상속이 순환적으로 이루어져서는 안 된다는 세부규칙(ClD_Rule11)을 적용한 결과를 (그림 8)에 보였다.



(그림 8) 클래스도의 오류 검사결과

(그림 9)는 모든 순차도의 객체에 대하여 대응하는 클래스도의 클래스가 존재한다는 세부규칙(Sqd_Rule16)을 적용한 결과를 (그림 9)에 보였다. 검사 결과로 (그림 5)에서의 순차도에는 객체 admissionscreen이 있으나, 클래스도에는 admissionscreen이 존재하지 않다는 것을 확인할 수 있다.



(그림 9) 클래스도와 순차도의 일관성 검사결과

(그림 7)에서 나타난 바와 같이 하나의 세부규칙을 Prolog코드로 변환하는 데는 Prolog 프로그래밍 경험을 필요로 한다. 그러나 세부규칙 단위로 Prolog코드를 생성하고 세부규칙 단위로 검사 작업을 수행하므로 세부규칙에 대한 Prolog코드의 개선작업은 개선요구사항이 식별되면 수월하게 수행할 수 있을 것으로 판단된다. 반면에 설계 모델을 Prolog 코드로 변환하는 일은 기계적으로 수행할 수 있어 자동화하는 일도 가능할 것이다.

5. 관련연구와의 비교 평가

설계 모델의 오류 검출 및 일관성 점검을 위한 연구의 접근 방법은 설계 모델의 오류 검출 및 일관성 점검을 위하여 각각의 UML 다이어그램의 요소들의 관계를 통하여 메타모델을 작성하고, 이로부터 일반화된 점검 규칙과 세부적인 점검 규칙들을 차례대로 도출하는 것이다. 이미 수행된 연구에서도 이러한 접근 방법의 일부분들이 수행되었다. 메타모델, 규칙을 통한 오류 검출 및 일관성 점검 방법이나, 정형화 기법 등의 연구가 바로 이들이다. 하지만 이들은 오류 검출을 위해 적합하지 않거나, 오류 검사 수행이 복잡하다. 본 장에서는 이러한 기존의 방법들과의 비교 평가를 수행한다.

5.1 메타모델 설계 방법간의 비교

본 연구에서 메타모델은 오류 검출 및 일관성 점검 규칙들을 빠짐없이 도출하기 위한 하나의 접근방법으로 사용하고 있다. 제시하는 메타모델은 다이어그램을 구성하는 요소들 간의 연관관계를 중심으로 작성하여 규칙 도출에 특성화하였다. 본 절에서는 다른 연구에서 제시하고 있는 메타모델들과 점검 규칙 도출의 측면에서 비교평가를 수행한다.

가. OMG의 UML 메타모델

OMG에서 인증한 UML에서도 메타모델이 제시되고 있다. 그러나, UML에서 제시한 메타모델은 각 다이어그램을 구성하는 요소들에 대한 일반화관계를 중심으로 기술하고 있어 이해도 증진 목적으로는 합당하나 모든 요소들에 대하여 메타모델이 제시되지 않았고, 각 다이어그램 간의 연관관계에 대한 사항은 자세히 제공하지 않으므로 이를 오류 검증과 일관성 점검을 목적으로 활용하기에는 부적합하다[18,19].

나. OMT, OOSE를 위한 메타모델 연구

기존의 OMT, OOSE와 같은 객체지향 방법론에 대해서도 각각의 다이어그램을 구성하는 요소들과 다이어그램들 사이의 관계를 메타모델로 제시한 연구가 있다[10]. 하지만, 이 메타모델은 오류 검증과 일관성 점검과 같은 소프트웨어 설계정보의 무결성을 점검하기 위한 모델이기 보다는 각 개발 방법론을 비교하기 위한 모델로 사용되었다.

다. 아키텍처 중심의 메타모델

소프트웨어의 아키텍처를 중심으로 메타모델을 구성한 시도도 있다[17]. 이는 소프트웨어의 아키텍처를 바라보는 여러 가지 시각을 제시하고, 각각의 관점에 대한 UML의 다이어그램에 대한 메타모델을 정의하고 있다. 이 메타모델은 소프트웨어의 설계정보에 대한 의미를 파악할 수 있다는 장점을 가지고 있다. 하지만 오류 검출과 일관성 점검을 위해서 각각의 시각에 대한 모델을 다시 작성해야 하며, 다이어그램의 문법적인 오류에 대한 검사를 수행하기 어렵다는 단점을 가지고 있다.

라. 일관성 점검을 위한 메타모델의 다른 연구

일관성 검증을 위해 제시된 메타모델도 있다. 이 모델은 UML의 각 다이어그램을 구성하는 요소들을 모형화하고 하나의 다이어그램을 구성하는 요소와 다른 다이어그램을 구성하는 요소를 연결하여 전체적인 메타모델을 제시하고 있으나[4], 이 메타모델은 다이어그램간의 연관관계에 치중되어 있어, 다이어그램의 오류를 검출하기 위한 규칙이나 기법을 도출하기에는 적절치 못하다.

마. 정 리

이상과 같이 제시된 UML을 비롯한 여러 객체지향 설계 방법의 개발 산출물에 대한 메타모델이 있으나, 이들은 오류 검출 및 일관성 점검 기법을 도출하기에 적절치 않다. 그러므로, 이러한 목적에 부합하는 UML 다이어그램에 대한 메타모델을 작성할 필요가 있다. 메타모델을 작성할 때 사용할 표기법으로는 다이어그램을 구성하는 요소들을 객체로 보고, 요소들 간의 연관관계를 모형화하며 하나의 다이어그램 내의 요소들에 한정하지 않고 여러 다이어그램들에 존재하는 요소들 간의 연관관계도 모형화에 포함시켜야 한다.

본 논문에서 제시하는 방법과 위에서 설명한 4가지 메타모델 설계 방법간의 비교 결과를 <표 3>으로 정리하였다.

〈표 3〉 메타모델 설계 방법간의 비교 결과

메타모델 / 목 적		A	B
본 논문	오류 검출, 일관성 점검	×	○
OMG (가)	UML 설계, UML 이해	×	×
OMT, OOSE를 위한 (나)	개발 방법론 비교를 위한 모델	×	×
아키텍쳐 중심 (다)	설계 정보에 대한 새로운 시각	×	×
일관성 점검용(라)	일관성 검증	×	○

A : 오류 검출 규칙 도출 지원
B : 일관성 점검 규칙 도출 지원

5.2 오류 검출 및 일관성 점검 방법간의 비교

본 연구에서는 도출된 오류 검출과 일관성 점검을 위하여 메타모델로부터 도출된 규칙을 사용할 것을 제안하고 있다. 이러한 방법은 메타규칙과 같은 그룹화된 규칙들을 이용하여 해당 그룹의 특성만을 점검하는 것을 가능하게 한다. 또한, 분석 및 설계 모델의 변환 작업과 같은 부가적인 작업이 필요치 않으며, 자동화 도구의 개발도 용이할 것으로 보인다. 본 절에서는 다른 연구에서 제시하고 있는 오류 검출 및 일관성 점검 방법들과의 비교평가를 수행한다.

가. 페트리 넷을 이용한 방법

요구공학에서 모듈라 페트리 넷을 사용하여 사용사례를 분석하고 통합하는 방법이 있다[21]. 여기에서 소프트웨어의 요구사항은 사용사례의 집합처럼 기술된다. 사용사례를 통한 접근은 쉽게 이해되고 기술할 수 있으며 추적가능하기에 좋은 방법이다. 사용사례를 페트리 넷을 이용, 정형화하여 시스템의 완전성과 일관성을 갖는 시스템을 개발할 수 있게 된다. 요구분석시에 사용자의 요구사항이 시스템에 정확하게 반영이 되는지 확인할 수 있고, 완전성과 일관성을 유지하면서 시스템의 행위를 찾아 볼 수 있게 해준다. 그러나, 사용사례에 기술된 시나리오를 분석하는 과정에서 CMPN(Constraint-based Modular Petri Net)로 변화시켜야 하는 불편함이 있다.

나. 지식베이스를 이용한 방법

개발 과정에서 생성되는 산출물의 정보에 대한 규칙을 이용하여 일관성을 점검하는 방법으로 지식베이스를 이용한 방법이 있다[1,5]. 정형명세를 이용하여 정보를 기술하고, 지식베이스에 담겨있는 추론 규칙을 이용하여 분석을 수행한다. 이 방식은 기능모델, 객체모델, 동적모델에 대한 오류 검사를 수행하는데 도움을 준다. 하지만, UML에서 제공하는 모든 다이어그램에 대한 정보를 가지고 있지 못하며, 모델들 사이의 일관성 점검을 수행하기에는 제시된 규칙의 완전성 면에서 부족하다.

다. 제약언어 기반의 방법

객체지향 분석 및 설계의 산출물이 객체 모델의 일관성을 유지하고 품질을 높이기 위한 방법이 있다[2]. 일관성을 유지하기 위해서 주어진 지침과 경험에 의한 관례를 명시하는 모델 제약 언어를 제시하고 있다. 객체 모델을 작성할 수 있는 객체 모델 작성기와 작성된 객체 모델을 명세된 제약언어로 검증할 수 있는 객체 모델 검증기를 포함한 객체 모델 검증 시스템을 제시하고 있다. 그러나, 현재는 OMT 방법론에 근거한 객체 모델만을 지원하고 있고, 동적 모델 및 기능 모델의 작성은 지원하지 않고 있다.

라. 규칙 기반의 방법

UML의 다이어그램들 간의 일관성 검증을 위한 연구가 있다. 하나의 다이어그램의 구성요소와 다른 다이어그램의 구성요소에 대한 연관관계를 규칙으로 제시한다. 하지만 정적, 동적, 기능적인 측면들에 대한 의미적인 구분이 없이 단지 다이어그램의 구성요소에 대한 연관관계만을 표현하고 있으며, 적용하기 위한 규칙이 너무 적고, 자동화가 되어있지 않아서 수작업으로 분석을 해야 한다면 역시 큰 소득을 얻을 수 없다[3].

마. 정 리

이들 방법은 대부분 정형명세나 제약 언어들을 기반으로 하고 있으며, 세부적인 규칙들이 정확히 나열되어 있지 않아 자동화에 어려움이 있다. 또한, UML을 기반으로 하는 오류 검출 및 일관성 점검 방법은 아직 신뢰성 높은 연구 결과가 발표되지 못하고 있다.

이러한 경우를 고려하여, UML에서 제시하는 모든 다이어그램에 대한 메타모델을 작성하고, 이를 바탕으로 일반화된 규칙을 도출한다. 도출된 일반화된 규칙을 바탕으로 다이어그램을 구성하는 각 요소들에 대한 세부규칙을 도출하면, 모든 요소들에게 적용할 규칙들을 빠짐없이 도출할 수 있어 규칙의 완전성에 접근하게 될 뿐만 아니라 점검의 자동화부분 확대에 크게 기여할 수 있다.

본 논문에서 제시하는 방법과 위에서 설명한 4가지 오류 검출 및 일관성 점검 방법간의 비교 결과를 <표 4>로 정리하였다.

〈표 4〉 오류 검출 및 일관성 점검 방법간의 비교 결과

방법	접근방법	A	B	C	D
본 논문	메타모델로부터 세부규칙 도출	○	○	○	○
페트리 넷 기반 (가)	페트리넷으로 사용사례 분석	○	○	×	×
지식베이스 기반 (나)	산출물 정보를 추론 규칙으로 도출	×	○	×	○
제약언어 기반 (다)	모델 제약 언어로 일관성 점검	○	○	×	○
규칙 기반 (라)	다이어그램간 연관 관계로 일관성 점검	×	○	×	○

A : 오류 검출 규칙 도출 지원
B : 일관성 점검 규칙 도출 지원
C : UML 전 다이어그램 지원
D : 변환 과정 없이 점검

6. 결 론

지금까지 UML을 기반으로 한 객체지향 설계방법에서의 오류 검출 및 일관성 점검 기법에 대하여 알아보았다. 오류 검출과 일관성 점검을 위하여 점검 규칙들을 제시하였으며, 점검 규칙을 도출하기 위하여 UML의 각 다이어그램별로 다이어그램을 구성하는 요소들의 관계를 표현하는 메타모델을 작성하였다. 작성된 메타모델로부터 요소들의 의미적 특징을 바탕으로 일반화된 규칙을 추출하였으며, 이를 통하여 개발자들이 설계 문서에 적용할 수 있는 세부적인 규칙들을 도출하였다. 이렇게 체계적인 방법으로 세부규칙을 도출하면 규칙의 완전성, 즉 빠트림 없이 모든 규칙을 찾아내는데 큰 도움을 준다. 도출된 규칙들을 바탕으로 자동화된 프로그램을 작성하여 직접 적용해 보았다. 제시된 규칙들을 통하여 개발자들은 설계문서상의 오류를 검출하고 일관성을 점검할 수 있으며, 이를 통하여 최근의 추세인 복잡한 대형 소프트웨어에 대한 신뢰성 있는 설계 결과를 만들 수 있다.

앞으로의 연구에서는 도출된 규칙들을 더욱 정제하고, 이들을 UML의 다이어그램별로 분류하여 제공할 수 있도록 할 것이다. 또한, 도출된 세부 규칙들을 적용해주는 자동화 도구를 개발하여 모델상의 오류 검출과 일관성 점검을 자동화하도록 추진하여야 할 것이다.

참 고 문 헌

[1] 김도형, 정기원,"객체지향 분석과정에서 오류와 일관성 점검 방법", 정보과학회 논문지(B), 제26권 3호, pp.380-392, 1999. 3.

[2] 김진수, 강권학, 이경환, "제약 언어를 이용한 객체 모델 검증 시스템", 정보처리논문지 제3권 6호, pp.1453-1467, 1996. 11.

[3] 박진호, 김수동, 류성열, "UML 다이어그램들 간의 일관성 검증 방법", 한국정보과학회 '98 봄 학술발표논문집 제25권 1호, 1998.

[4] 심우혁, 김수동, "UML 다이어그램들간의 관계 표현을 위한 메타모델", 한국정보과학회 '98 봄 학술발표논문집 제25권 1호, 1998.

[5] 이광용, "객체지향 분석 모델의 완전성과 일관성 점검을 위한 진단 모델의 분석", 숭실대학교 석사학위논문, 1992.

[6] 천윤식, 조수선, "객체지향 LOTOS의 기본 분석", 한국정보과학회 가을 학술발표논문집 제23권, 제2호, pp.1449~1452, 1996.

[7] Carrington D., Duke D., Duke R., King P., Rose G. and Smith G., "Object-Z: An Object-Oriented Extension to Z," Formal Description Techniques (FORTE '89)-North Holland, 1990.

[8] Duke D. and Duke R., "Towards a Semantics for Object-Z," Proc. of VDM '90 Conf., Lecture Notes in Computer Science 428, pp.244-261, 1990.

[9] Edmund M. Clarke, Jeannette M. Wing, "Formal Methods: State of the Art and Future Directions," ACM Computing Surveys Vol.28 No.4, pp.626-643, December 1996, http://hermes.ifad.dk/products/vdmlangchar.html, 1996.

[10] Esteban A. Pastor, R. T. Trice, "Using Metamodel of Methodologies to determine the needs for reusability support," Proceedings of 1997 Symposium on Software reusability, pp.121-129, 1997

[11] Formal Methods, http://www.comlab.ox.ac.uk/archive/formal-methods.html

[12] IFAD, "Major VDM++ Language Characteristics."

[13] Hans-Erik Eriksson, Magnus Penker, *UML Toolkit*, John Wiley & Sons, Inc., 1998.

[14] James Rumbaugh, Ivar Jacobson, Grady Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley, Inc., 1999.

[15] James Rumbaugh, Ivar Jacobson, Grady Booch, *The Unified Modeling Language User Guide*, Addison Wesley, Inc., 1999.

[16] Martin Fowler, and Kendall Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison Wesley, Inc., 1997.

[17] Phillippe Kruchten, "Software Architecture-A Rational Metamodel," Joint proceedings of the second international software Architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (viewpoint '96) on SIGSOFT '96 workshop, pp.5-7, 1996.

[18] Rational, "UML Semantics version 1.1," http://www.rational.com/uml, 1 Sep. 1997.

[19] Rational, "UML Notation Guide version 1.1," http://www.rational.com/uml, 1 Sep. 1997.

[20] Wolfgang Dzida, Regine Freitag, "Making Use of Scenarios for Validating Analysis and Design," IEEE Transactions on Software Engineering, Vol.24, No.12, pp.1182-1196, December 1998.

[21] Woo Jin Lee, Sung Deok Cha, Yong Rae Kwon, "Integration and Analysis of Use Cases Using Modular Petri Nets in Requirement Engineering," IEEE Transactions on Software Engineering, Vol.24, No.12, pp.1115-1130, December 1998.

정 기 원

e-mail : chong@computing.soongsil.ac.kr

1967년 서울대학교 전기공학과(공학사)

1981년 미국 알라바마 주립대학(전산학 석사)

1983년 미국 텍사스주립대학(전산학 박사)

1969년~1971년 대한전자공업주식회사 자료처리과장

1971년~1975년 한국과학기술연구소 전자계산실

1975년~1990년 국방과학연구소 책임연구원

1990년~현재 숭실대학교 컴퓨터학부 교수

관심분야 : 소프트웨어공학, 모델링/시뮬레이션, 실시간 시스템, CALS/EC

조 용 선

e-mail : yongsuns@it.soongsil.ac.kr

1997년 숭실대학교 소프트웨어공학과(공학사)

1999년 숭실대학교 대학원 전자계산학과(공학석사)

1999년~현재 숭실대학교 대학원 컴퓨터학과(박사과정)

관심분야 : 소프트웨어공학, 개발방법론, 객체기술, 형상관리

권 성 구

e-mail : turtle@it.soongsil.ac.kr

1998년 숭실대학교 전산과(공학사)

1998년~현재 숭실대학교 대학원 컴퓨터학과(석사과정)

관심분야 : 소프트웨어공학, 객체지향 개발방법론, 정형명세