UML 분석 모델의 구조적 제약사항에 대한 OCL 기반의 명세 및 검증

(OCL Based Specification and Verification of Structural Constraints for UML Analysis Models)

채 흥 석 † 염 근 혁 ††
(Heung Seok Chae) (Keunhyuk Yeom)

요 약 분석 모델은 오직 시스템의 기능적 요구사항에 초점을 두며, 비 기능적인 요구사항과 구현과 관련된 이슈들은 이후의 설계 작업이 착수될 때까지 미루어진다. 설계 활동은 분석 모델을 상세화하고 명확하게 하는 방식으로 수행된다. 따라서, 분석 모델의 품질은 설계 모델에 지대한 영향을 미친다. 그러므로, 정확한 분석 모델을 구축하기 위하여 많은 노력을 기울여야 한다.

본 논문에서는 전형적인 객체지향 개발 방법론의 분석 모델이 항상 충족해야 하는 구조적 제약 사항을 제안한다. 제약 사항은 개별 클래스에 관한 제약사항, 클래스 간의 관계에 대한 제약 사항과 클래스의 사용에 대한 제약사항으로 분류된다. 각 제약사항 별로 정형적인 정의와 OCL을 이용한 기술이 제공된다. 또한, 2개의 산업체 프로젝트를 대상으로 수행된 사례 연구를 통하여 제안된 기법이 객체지향 분석 모델에 존재하는 심각한 오류를 발견하고 이를 수정하는 데 도움을 줄 수 있음을 보여 준다.

키워드 : 객체지향 분석 모델 검증, OCL

Abstract Analysis model focuses only on functional requirements and postpones nonfunctional requirements and implementation specific issues until subsequent design activities are undertaken. Based on the analysis models, the design activities are performed by refining and clarifying the analysis models. Thus, the quality of analysis models has a vast impact on the design models. Therefore, much effort should be taken to build correct analysis model.

In this paper, we propose a set of structural constraints that analysis models of typical object-oriented development methods should satisfy. Three kinds of constraints are proposed: class related constraints, relation related constraints, and usage related constraints. For each constraint, formal definition and description with OCL are provided. In addition, through a case study with two medium-sized industrial systems, we demonstrated that the proposed approach can help to identify and correct serious deficiencies in object-oriented analysis models.

Key words : Object-Oriented Analysis Model Validation, OCL

1. 서 론

분석 모델은 객체지향 소프트웨어 개발 프로세스 중에서 분석 단계에서 작성되는 객체 모델을 의미한다. 분석 모델은 요구사항을 보다 구체화하고, 시스템에 대한 보다 명확한 이해와 용이한 유지보수의 이점을 제공하는 데 중요한 역할을 한다. 또한, 분석 이후의 설계 단계 및 구현 단계의 수행 시 입력으로 사용된다[1]. 분석 모델이 가지는 가장 중요한 특성 중의 하나는 독립성이다. 다시 말하면, 분석 모델은 운영체제, 미들웨어, 프레임워크, 레거시 시스템, 프로그래밍 언어 등과 같은 구현 환경에 독립적으로서 이들 요소에 무관하게 개발된다. 또한, 분석 모델은 오직 시스템에 대한 기능적인 요구사항에 초점을 두며, 성능, 신뢰도, 유지보수성, 견고성 등의 비 기능적인 요구사항에 대한 사항은 설계 단계에서 다루어진다. 이와 같이 분석 단계와 설계 단계를 구분하여 시스템 개발 중의 이슈들을 다루는 것은 바로

· 이 논문은 교육인적자원부 지방연구중심대학육성사업(차세대물류 IT 기술연구사업단)의 지원에 의하여 연구되었음
† 정 회 원 : 부산대학교 컴퓨터공학과 교수
hschae@pusan.ac.kr
†† 종신회원 : 부산대학교 컴퓨터공학과 교수
yeom@pusan.ac.kr
논문접수 : 2004년 11월 4일
심사완료 : 2005년 12월 13일

소프트웨어 공학의 기본 원칙인 separation of concern에 해당된다.

객체지향 분석 모델의 독립성은 경계 클래스(boundary class), 제어 클래스(control class), 개체 클래스(entity class)라는 세가지 유형의 클래스를 통하여 얻어진다. 각 유형의 클래스는 특정한 목적으로 사용되는 클래스를 지칭하며 구현 환경에 독립적인 분석 모델을 작성하는 데 중요한 도구로서 사용된다. 경계 클래스는 시스템과 액터(사용자, 외부 시스템 또는 장치)과의 상호작용을 모델링 하는 데 사용되며, 사용자 인터페이스를 위한 화면 객체, 외부 시스템 및 장치와의 통신을 위한 인터페이스 등이 경계 클래스에 해당된다. 제어 클래스는 다른 객체들에 대한 조율, 트랜잭션, 및 제어 기능을 제공하는 클래스로서 일반적으로 특정 유즈케이스의 비즈니스 로직을 제공하는 데 사용된다. 예를 들면, 트랜잭션 관리자, 자원 관리자, 에러 핸들러 등이 제어 클래스에 해당된다. 개체 클래스는 장기간 유지 및 관리가 되는 정보 및 그 정보에 대한 조작 기능을 모델링 하는 데 사용된다.

객체지향 분석 모델은 구현 환경에 독립적이기 때문에, 상이한 구현 환경이지만 동일한 요구사항을 가지는 다른 시스템을 개발할 때 사용될 수 있다. 예를 들어, .NET 플랫폼을 대상으로 개발된 신용 평가 시스템의 분석 모델은 동일한 요구사항이라고 한다면 J2EE 플랫폼의 신용 평가 시스템의 개발에도 사용될 수가 있다. 이처럼 하나의 분석 모델이 여러 시스템의 개발에서 사용되는 것은 그 분석 모델의 품질이 여러 시스템의 구축에 직접적인 영향을 미칠 수 있음을 뜻한다. 따라서, 정확한 분석 모델은 전체 시스템 개발 관점에서 매우 중요한 역할을 한다.

뿐만 아니라, 분석 모델은 Model Driven Architecture(MDA)[2]의 Platform Independent Model에 비유될 수 있다. MDA는 Platform Independent Model (PIM)과 Platform Specific Model(PSM)에 바탕을 둔다. PIM은 상위 수준의 추상화를 지원하는 모델로서 구현 기술에 무관하다. 그리고, PIM은 특정 구현 기술을 적용하여 PSM으로 변환된다. 분석 모델과 마찬가지로 하나의 PIM은 다른 구현 환경에 맞춘 여러 PSM으로 변환될 수 있기 때문에, 정확한 PIM의 구축은 매우 중요한 작업이다.

본 논문에서는 정확한 객체지향 분석 모델을 구축하기 위하여 구조적인 제약 사항을 바탕으로 한 객체지향 분석 모델의 검증 방법을 제안한다. 우선, 객체지향 분석 모델이 항상 충족시켜야 하는 구조적 제약 사항을 소개한다. 그리고, 분석 모델에서 제안된 구조적 제약 사항을 위반하는 부분을 찾고 이를 수정함으로써 객체지향 분석 모델의 정확성을 향상시키는 방법을 제시한다. 이 방법은 모든 객체지향 분석 모델에서 충족시켜야 하는 기본적인 그리고 구문적인 특성에 바탕을 두고 있으므로 분석 모델이 기능적인 측면의 검증은 아니지만 정확한 모델을 구축하는 데 기여할 것으로 판단된다.

본 논문에서 제안하는 제약사항은 객체지향 분석 모델 고유의 구조적인 특성으로부터 유도된다. 즉, 분석 모델을 구성하는 세가지 유형의 클래스 고유의 의미를 바탕으로 제약 사항을 제안한다. 예를 들면, 각 제어 클래스는 하나 이상의 경계 클래스와 연관 관계를 맺어야 한다. 제어 클래스는 시스템의 비즈니스 로직을 전담하며 경계 클래스는 시스템의 외부 즉 사용자 또는 외부의 다른 시스템과의 인터페이스를 전담한다. 그리고, 사용자 또는 외부의 시스템은 경계 클래스를 통하여 제어 클래스가 제공하는 비즈니스 로직을 접근할 수 있다. 따라서, 경계 클래스와 연관 관계가 없는 제어 클래스는 사용자(또는 외부 시스템)에 의해서 그 기능이 이용될 수 없게 된다. 그러므로, 각 제어 클래스가 하나 이상의 경계 클래스와 연관 관계를 맺는 것은 정확한 모델로서의 기본적인 제약 사항이 된다.

본 논문에서는 제안된 제약 사항을 표현하기 위하여 Object Constraint Language(OCL)[3,4]을 사용한다. OCL은 UML 표준의 일부로서 UML 모델의 특정 문맥(context)하에서의 다양한 제약 사항을 기술할 때 효과적인 언어이다. 예를 들어, OCL은 클래스의 불변식(invariant)과 각 연산에 대한 선행 조건 및 후행 조건을 명세할 때 이용될 수 있다. 뿐만 아니라, OCL은 모델의 검증[5-17], 테스트 케이스 생성[18] 등의 연구에도 사용되고 있다.

본 논문은 객체지향 분석 모델이 충족시켜야 하는 제약 사항을 OCL로 표현하고 이 제약 사항에 대한 충족 여부를 확인하는 방식으로 분석 모델을 검증하는 기법을 소개한다. 뿐만 아니라, 두 개의 실제 프로젝트의 분석 모델에 대해서 수행된 사례 연구 결과를 소개한다. 이 사례 연구에서는 제약 사항을 위반하는 많은 사례를 발견하였으며 객체지향 분석 모델을 개선시킬 수 있도록 개발자에게 권고하였다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문에서 검증 대상으로 다루는 객체지향 분석 모델에 대하여 간략하게 소개한다. 3장에서는 객체지향 분석 모델에 대해서 검증할 수 있는 구조적 제약 사항을 제안한다. 그리고, 4 장에서는 본 논문에서 제안한 기법을 실제 산업체의 2개의 프로젝트에 적용한 사례 연구를 소개한다. 5장과 6장에서는 본 연구와 관련된 기존의 연구 및 향후 연구 방향을 각각 설명한다.

2. 객체지향 분석 모델

객체지향 분석 모델은 시스템 개발 프로세스 중에서 분석 단계의 산출물로서 실제 구현 환경에 독립적인 시스템을 구성하는 데 초점을 둔다. 따라서, 객체지향 분석 모델을 통하여 시스템 생명 주기 전체에 걸쳐서 보다 안정적이고 유지보수가 용이한 시스템 구조를 표현할 수 있다. 본 논문에서 언급하는 객체지향 분석 모델은 시스템을 구성하는 논리적인 클래스 및 그들 간의 관계로 한정하며 UML 클래스 다이어그램으로 표현한다. 예를 들어, 그림 1은 객체지향 분석 모델의 예를 보여 준다.

객체지향 분석 모델을 구성하는 클래스는 경계 클래스, 제어 클래스, 개체 클래스의 세가지 유형으로 구분된다. 각 유형의 클래스는 명확한 의미를 가지며 클래스 다이어그램에서는 <<boundary>>, <<control>>, <<entity>>와 같은 스테레오 타입으로 구분된다. 이와 같은 세가지 유형의 클래스에 바탕을 둔 분석 모델의 개념은 Ivar Jacobson의 Use Case Driven Approach[18]에서 그 출처를 찾을 수 있으며 Unified Process[19,1], ICONIX Process[20] 등과 같은 객체지향 및 컴포넌트 기반 방법론에서 일반적으로 이용되고 있다. 심지어, 이 세가지 스테레오 타입은 UML 언어에서 표준으로 정의되었다[3]. 그림 1의 클래스 다이어그램은 세가지 유형의 클래스 뿐만 아니라 액터도 포함하고 있다. 액터는 분석 모델이 아니라 요구사항 정의 단계에서 작성되는 유즈케이스 모델에 속한다. 그러나, 본 논문에서 제안하는 구조적 제약 사항은 경계 클래스와 액터 간의 관련성도 포함하고 있으므로, 분석 모델에 액터도 함께 표현한 것이다. 3장에서 구조적 제약 사항을 소개하기에 앞서서 제약 사항의 배경에 대한 이해를 도울 수 있도록 액터를 포함하여 세가지 스테레오 타입의 클래스를 간략히 소개 한다.

- 액터는 시스템과 상호작용을 하는 시스템 외부의 대상을 나타낸다. 시스템의 사용자, 시스템에 연동하는 외부의 다른 시스템, 시스템에 감시 또는 제어하는 장치 등이 액터의 전형적인 예이다. 예를 들어, 그림 1의 분석 모델에는 2개의 액터가 있다. Customer 액터는 이 시스템을 이용하는 사용자 유형의 액터이고, ValidationSystem 액터는 이 시스템과 연동되는 또 다른 시스템을 나타낸다.
- 경계 클래스는 시스템과 액터 간의 경계 즉 인터페이스에 대한 표현을 전담하는 클래스를 뜻한다. 사용자와의 인터페이스를 위하여 사용되는 화면(윈도우 또는 폼), 다른 시스템과의 통신을 전담하는 클래스, 그리고 프린터, 센서 등의 장치와의 인터페이스를 전담하는 클래스가 경계 클래스에 해당한다. 예를 들어, 그림 1의 분석 모델은 5개의 경계 클래스를 가지고 있다. LoginForm, MenuForm, OrderForm, BannerForm 클래스는 Customer 액터와의 사용자 인터페이스를 위한 화면을 나타내는 경계 클래스이다. 그리고, ValidatingSystemAgent 클래스는 ValidationSystem 액터와의 통신 기능을 제공하는 경계 클래스이다.
- 개체 클래스는 시스템에서 영속적으로 유지할 정보와 그 정보에 대한 관리 기능을 제공하는 클래스이다. 따라서, 정보에 대한 생성, 조회, 검색, 삭제 등이 개체

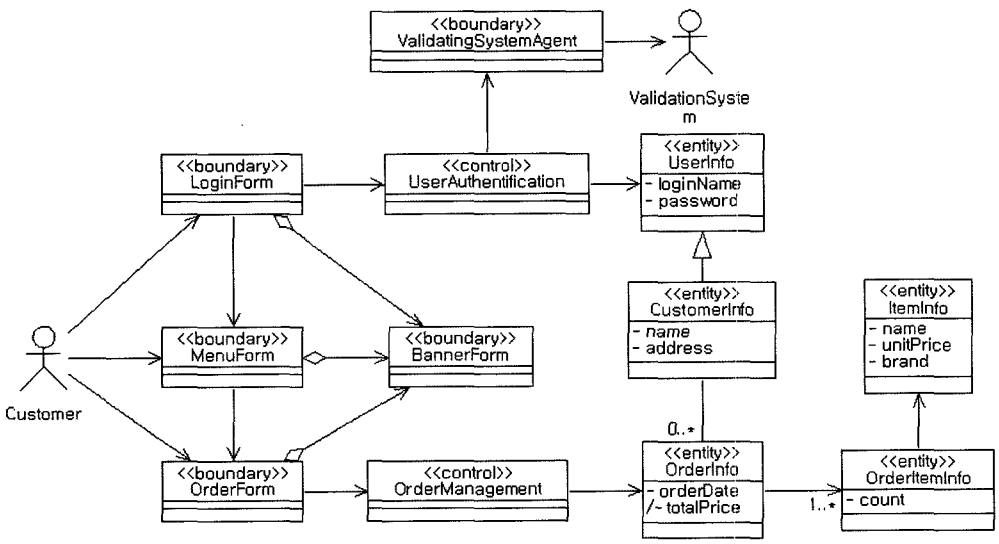


그림 1 객체지향 분석 모델의 예

클래스의 주요 기능이다. 그림 1의 분석 모델에는 5개의 개체 클래스가 있다. UserInfo, CustomerInfo, OrderInfo, OrderItemInfo, ItemInfo 클래스는 각각 사용자 정보, 고객 정보, 주문 정보, 주문 상품 정보, 상품 정보와 각 정보에 대한 관리 기능을 제공하는 개체 클래스이다.

- 제어 클래스는 시스템이 제공하는 제어 로직 및 비즈니스 로직을 나타낸다. 일반적으로 제어 클래스는 한 개 또는 그 이상의 유즈케이스에 한정된 특정한 제어 로직을 제공한다. 제어 클래스는 액터와의 상호작용과 영속적인 정보에 대한 관리와 관련된 기능을 제공하지 않는다. 액터와의 상호작용은 경계 클래스에 의해서 표현되며 영속적인 정보에 대한 관리는 개체 클래스가 담당한다. 다시 말하면, 제어 클래스는 세부적인 입출력 방식과 정보 관리 로직에 무관한 순수한 비즈니스 로직만을 제공한다. 그림 1의 분석 모델에는 2개의 제어 클래스가 있다. UserAuthentification 클래스는 사용자 인증 기능을 제공하는 제어 클래스이고 OrderManagement 클래스는 주문에 대한 처리 기능을 제공하는 제어 클래스이다.

3. 객체지향 분석 모델의 구조적 제약 사항

이 장에서는 객체지향 분석 모델이 충족해야 할 구조적 제약 사항을 소개한다. 제약 사항을 구체적으로 소개하기에 앞서 제약 사항의 명확한 기술을 위한 기본적인 정의를 몇 가지 설명한다.

3.1 기본 정의

정의 1. CL은 시스템을 구성하는 모든 분류자(classifier)의 집합을 나타낸다. 그리고, CS와 AT는 각각 시스템을 구성하는 모든 클래스와 액터를 나타낸다

UML에서는 클래스와 액터는 분류자의 일종이다. 따라서, $CS \subseteq CL$, $AT \subseteq CL$, $CS \cap AT = \emptyset$ 이다. 예를 들어, 그림 1의 분석 모델에서 CL = {Customer, LoginForm, MenuForm, OrderForm, BannerForm, UserAuthentification, OrderManagement, UserInfo, CustomerInfo, OrderInfo, OrderItemInfo, ItemInfo} 이다. 그리고, AT = {Customer, ValidationSystem} 이며, CS는 CL에서 AT를 뺀 나머지에 해당된다.

정의 2. st(c)는 분류자 c에 대한 스테레오 타입을 나타낸다. 그리고, 편의상 액터의 스테레오 타입은 "actor"로 정의한다.

그림 1에서 st(Customer) = st(ValidationSystem) = "actor"이고, st(MenuForm) = "boundary"이다. 그리고, st(OrderInfo) = "entity" 이고 st(OrderManagement) = "control"이다. UML에서는 분류자는 한 개 이상의 스테레오 타입을 가질 수가 있다. 그러나, 본 논문에서 대상으로 하는 분석 모델은 클래스가 <<boundary>>, <<control>>, <<entity>> 스테레오 타입 중의 오직 하나를 가지기 때문에 st(c)가 오직 하나의 스테레오 타입을 지정하는 것으로 가정한다.

뿐만 아니라, BD, CT, ET를 각각 경계 클래스, 제어 클래스, 개체 클래스의 집합을 나타내기 위하여 사용한다.

정의 3. 분류자의 유형. 경계 클래스의 집합(BD), 제어 클래스의 집합(CT), 개체 클래스의 집합(ET)는 다음과 같이 정의된다.

$BD = \{c \in CS \mid st(c) = \text{"boundary"}\}$

$CT = \{c \in CS \mid st(c) = \text{"control"}\}$

$ET = \{c \in CS \mid st(c) = \text{"entity"}\}$

클래스 다이어그램에서는 두 클래스 간에 연관(association), 집합(aggregation), 일반화(generalization), 의존(dependency) 관계가 존재할 수 있다.

정의 4. 분류자 간의 관계. 분류자 *ci*와 *cj*간의 연관 관계, 집합 관계, 일반화 관계, 의존 관계를 가지며, 다음과 같이 표현된다.

$ci \rightarrow_s cj$: *ci 에서 cj 로의 연관 관계가 있다.*

$ci \rightarrow_a cj$: *ci 에서 cj 로의 집합 관계가 있다*

$ci \rightarrow_g cj$: *ci 와 cj 간에 일반화 관계가 있다.*

$ci \rightarrow_d cj$: *ci 에서 cj 로의 의존 관계가 있다.*

예를 들어, 그림 1의 분석 모델에서 LoginForm $\rightarrow_s$ UserAuthentification, LoginForm $\rightarrow_a$ BannerForm, UserInfo $\rightarrow_g$ CustomerInfo 이다.

그리고, 각 기본적인 관계를 바탕으로 분석 모델의 두 분류자 간의 인접 여부를 정의한다.

정의 5. 분류자 간의 인접. $ci \rightarrow_s cj$, $ci \rightarrow_a cj$, 또는 $ci \rightarrow_g cj$ 일 때 분류자 *ci* 가 *cj* 에 인접하다고 정의하며 $ci \Rightarrow cj$ 로 표현한다.

한 분류자가 다른 분류자에 인접한 것은 두 분류자 간에 순 방향의 연관 관계(집합 관계도 포함) 또는 일반화 관계가 있을 때로 정의된다. 예를 들면, 그림 1의 분석 모델에서 Customer 액터에서 LoginForm 클래스 방향으로의 연관 관계가 존재하기 때문에Customer $\Rightarrow$ LoginForm 이다. 마찬가지로, LoginForm $\Rightarrow$ BannerForm 도 성립된다. 그러나, BannerForm과 LoginForm 사이의 연관 관계가 LoginForm에서 BannerForm으로의 방향이기 때문에 BannerForm $\Rightarrow$ LoginForm은 성립되지 않는다.

연관 관계(집합 관계도 포함)와 달리 일반화 관계의 방향은 인접 관계에서 고려되지 않는다. 즉, 일반화 관계를 가진 두 클래스는 양 방향으로 인접한 것으로 정의한다. 예를 들어, UserInfo $\Rightarrow$ CustomerInfo 과 CustomerInfo $\Rightarrow$ UserInfo 은 모두 성립된다. 그리고,

분석 모델에서는 의존 관계는 사용되지 않는 것이 일반적이므로 두 분류자 가의 인접 관계는 의존 관계를 포함하지 않도록 정의되었다.

두 분류자는 직접적으로 인접하는 대신에 다른 분류자와의 인접 관계를 조합함으로써 간접적으로 연결이 될 수 있다. 우선 한 분류자에서 다른 분류자로의 연결 경로 개념을 정의하도록 한다.

정의 6. 분류자 간의 연결 경로. 분류자 *ci* 에서 *cj* 로의 연결 경로는 $< c_{p0}(= c_i), c_{p1}, c_{p2}, \cdots, ; c_{pn}(= c_j) >$로 *표현되며* $c_{pi} \Rightarrow c_{pi+1}(0 \leq i < n, 0 \leq n)$이어야 한다.

예를 들어, Customer ⇒ MenuForm and MenuForm ⇒ BannerForm이기 때문에 < Customer, MenuForm, BannerForm >은 Customer에서 BannerForm으로의 연결 경로가 될 수 있다. 두 분류자 간에는 2개 이상의 연결 경로가 존재할 수 있다. < LoginForm, UserAuthentification, UserInfo, CustomerInfo > 와 < LoginForm, MenuForm, OrderForm, OrderManagement, OrderInfo, CustomerInfo > 은 모두 LoginForm에서 CustomerInfo로의 연결 경로가 된다.

두 분류자 간의 연결은 연결 경로의 개념에 추가적으로 연결 경로에 포함된 분류자의 스테레오 타입에 대한 제약을 줌으로서 정의된다.

정의 7. 분류자 간의 연결. 분류자 *ci* 에서 *cj* 로의 연결은 $ci \Rightarrow^* cj$ 로 표현되며 *ci* 에서 *cj* 로의 연결 경로 $< c_{p0}(= ci), c_{p1}, c_{p2}, \cdots, c_{pn}(= cj) >$가 존재하고 $k \leq l$ 인 모든 k에 대해서 $st(ci) = st(c_{pk})$이고 $m > l$ 인 모든 m에 대해서 $st(cj) = st(c_{pm})$할 수 있는 l $(0 \leq l < n)$이 있어야 한다.

$ci \Rightarrow^* cj$ 연결 관계는 *ci* 에서 *cj* 로 연결 경로가 있는 것뿐만 아니라 연결 경로에 속한 각 분류자의 스테레오 타입이 특정한 패턴을 가지도록 제약한다. 즉, 연결 경로에 속한 각 분류자의 스테레오 타입은 *ci* 분류자의 스테레오 타입으로 시작해서 *cj* 분류자의 스테레오 타입으로 끝나야 하며, 이 순서가 바뀌어서는 안 되며 중간에 다른 유형의 스테레오 타입이 있어도 안 된다. 예를 들어, *ci* 와 *cj* 의 스테레오 타입을 *sti* 와 *stj* 라고 하면, *sti*, ⋯, *sti*, *stj*, ⋯, *stj* 패턴의 스테레오 타입만이 허용하며 *sti*, *sti*, *stk*, *stj* 경로와 *sti*, *stj* , *sti*, *stj* 경로는 허용되지 않는다. 전자는 *ci* 와 *cj* 의 스테레오 타입에 해당하지 않는 스테레오 타입 *stk* 가 경로에 있기 때문이고, 후자는 *stj*, *sti* 와 같이 두 스테레오 타입의 순서가 바뀐 형태로 나타내기 때문이다.

예를 들어, 그림 1의 분석 모델에서 LoginForm에서 OrderManagement로 연결 경로 < LoginForm, MenuForm, OrderForm, OrderManagement > 가 존재하며 스테레오 타입의 패턴이 boundary(= st(LoginForm)), boundary(= st(MenuForm)), boundary(= st(OrderForm)), control(= st(OrderManagement)) 이기 때문에 LoginForm ⇒* OrderManagement은 성립된다. 반면에, LoginForm ⇒ UserInfo은 성립되지 않는다. 연결 경로 < LoginForm, UserAuthentification, UserInfo >가 LoginForm에서 UserInfo로의 유일한 경로이면서 스테레오 타입은 boundary, control, entity와 같이 LoginForm의 스테레오 타입인 boundary와 UserInfo의 스테레오 타입인 entity 에 해당하지 않는 control 스테레오 타입을 UserAuthentification이 가지기 때문이다.

3.2 제약 사항

이 절에서는 객체지향 분석 모델이 충족시켜야 할 구조적 제약 사항을 소개한다. 본 논문에서 제안하는 구조적 제약 사항은 클래스 관련 제약사항, 관계 관련 제약사항, 사용 관련 제약 사항의 세가지 범주로 분류된다. 각 범주 속에는 한 개 이상의 구체적인 제약 사항이 정의된다. 각 세부적인 제약 사항에 대해서는 지금까지 소개한 기본 정의를 이용하여 제약 사항을 명확하게 기술한 후에 제약 사항의 배경을 설명한다. 그리고, 분석 모델에서 제약 사항을 검증할 수 있는 OCL 식을 소개한다. 뿐만 아니라, 그림 2의 분석 모델을 예로 하여 해당 제약 사항을 위반하는 사례를 설명한다. 그림 2의 분석 모델은 제약 사항을 위반하도록 그림 1의 분석 모델을 바탕으로 변형시킨 것이다.

3.2.1 클래스 관련 제약 사항

분석 모델을 구성하는 각 클래스가 충족시켜야 하는 제약 사항으로서 각 클래스의 스테레오 타입에 대한 제약 사항을 제안한다.

3.2.1.1 스테레오 타입 제약 사항

표현. $\forall c \in CS, st(c) =$ *"boundary" or "control" or "entity"*

설명. 각 분석 클래스는 boundary, control, entity 중의 오직 하나의 스테레오 타입을 가진다. 분석 단계에서 도출된 모든 클래스는 세가지 유형 중의 하나로 분류된다. 이는 2장. 분석 모델에서 설명한 것처럼 구현 환경에 독립적인 시스템에 대한 모델을 구축하는 가장 기본적인 틀을 제공한다.

그림 3은 스테레오 타입 제약 사항을 OCL 식으로 표현한 것을 보여 준다. 스테레오 타입 제약 사항은 각 분석 클래스에 대해서 항상 적용된다. 따라서, 이 제약 사항은 Class 문맥에서 불변식으로 표현되었다. 이 OCL 식은 각 클래스는 오직 하나의 스테레오 타입을 가지며 그 스테레오 타입은 "boundary", "control", "entity" 중의 하나 임을 표현한 것이다. **self**는 OCL에서 정의된 키워드로서 주어진 문맥에 일치하는 현재의 객체를 나

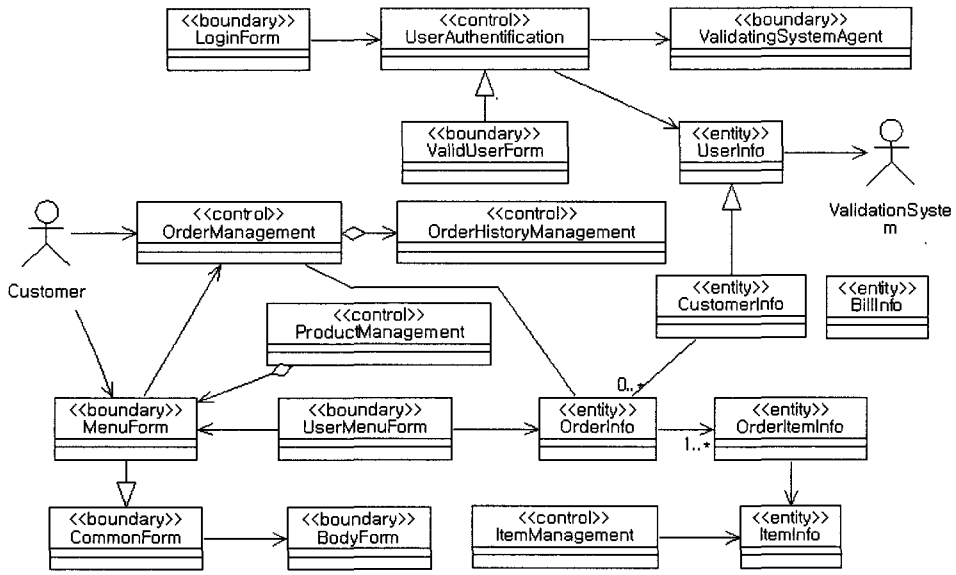


그림 2 제약 사항을 위반하는 객체지향 분석 모델의 예

```
inv StereotypeConstraint:
    let st(c:Classifier) : String =
        if c.oclIsTypeOf(Actor) then "actor"
        else c.stereotype->asSequence()->first.name.toLower
        endif
    self.stereotype->size() = 1 and
    Set{"boundary", "control", "entity"}->includes(st(self))
```

그림 3 스테레오 타입 제약 사항에 대한 OCL 식

타내며 Java 언어 또는 C++ 언어의 this 에 해당한다. st(Classifier) 정의 2에서 설명한 것처럼 인자로 주어진 분류자의 스테레오 타입을 구한다. 이 제약 사항은 액터가 아니라 클래스에만 적용되므로 액터에 대해서는 스테레오 타입을 기술할 필요가 없다. 즉, st(Classifier)가 아니라 st(Class)로 충분하다. 그러나, 본 논문에서는 연관 관계에 대한 제약 사항 등에서 st(Classifier)가 재사용될 수 있도록 클래스에 대해서만이 아니라 액터를 포함하도록 분류자에 대해서 st()를 정의하였다.

스테레오 타입 제약 사항 관점에서 그림 2의 분석 모델을 살펴 보면, 이 클래스 다이어그램의 모든 클래스는 모두 세가지 유형 중의 하나의 스테레오 타입을 가지고 있다. 그러므로, 이 분석 모델은 스테레오 타입 제약 사항을 충족시키고 있다.

3.2.2 관계 관련 제약 사항

클래스 간에는 연관 관계, 집합 관계, 일반화 관계, 의존 관계가 존재할 수 있다. 각 유형의 관계 별로 분석 모델이 충족시켜야 할 구조적 제약 사항을 소개한다.

3.2.2.1 집합 관계 제약 사항

표현. $\forall c_1, c_2 \in CS, c_1 \rightarrow_a c_2$ implies $st(c_1) = st(c_2)$

설명. 집합 관계를 맺고 있는 두 클래스는 동일한 스테레오 타입을 가진다. 집합 관계는 전체와 부분을 나타내는 두 클래스 간의 물리적인 또는 개념적인 포함의 의미가 성립될 때 사용된다. 따라서, 집합 관계를 이루는 두 클래스 간에는 클래스의 성격 면에서 유사성이 있어야 한다. 이를 분석 모델 관점에서 설명하면 두 클래스가 동일한 스테레오 타입을 가진다 라고 볼 수 있다. 예를 들어, 하나의 화면(경계 클래스)가 다른 화면(경계 클래스)을 부분으로서 가지는 것은 타당하다. 그러나, 비즈니스 로직 객체(제어 클래스)가 사용자 인터페이스 화면(경계 클래스) 또는 영속적인 정보(개체 클래스)를 부분으로서 포함하는 것은 자연스럽지 않은 관계이다.

```
context Association
inv AggregationConstraint:
    let isAggregation(as:Association) : Boolean =
        as.connection->exists(ae:AssociationEnd | ae.aggregation <> #none)
    let ptWithAs(as:Association) : Set(Classifier) = as.connection->asSet()
    isAggregation(self) implies ptWithAs(self)->forAll( c1, c2 | st(c1) = st(c2) )
```

그림 4 집합 관계 제약 사항에 대한 OCL 식

그림 4는 집합 관계 제약 사항에 대한 OCL 식이다. UML에서 집합 관계는 연관 관계의 일종이므로 Association 문맥 하에서 제약 사항이 불변식으로 표현된다. isAggregation(Association)은 주어진 연관 관계가 집합 관계에 해당하는 지를 조사한다. ptWithAs(Association)

은 주어진 연관 관계를 통하여 연관된 모든 분류자를 구한다. 최종적인 식은 주어진 연관 관계가 집합 관계인 경우에 연관된 모든 분류자의 스테레오 타입이 동일함을 표현한 것이다. 위의 OCL 식에서는 st(Classifier)가 사용되고 있지만 정의되지는 않았다. st(Classifier)의 정의는 그림 3에서와 동일하며 지면 관계 상 생략된 것이다.

그림 2의 분석 모델에는 두 개의 집합 관계가 있다. OrderManagement 클래스와 OrderHistoryManagement 클래스는 모두 "control" 스테레오 타입을 가지므로 이 제약 사항을 충족시킨다. 이 집합 관계는 전체 클래스인 OrderManagement가 제공할 기능의 일부가 부분 클래스인 OrderHistoryManagement로 위임되는 것으로 해석될 수 있다. 그러나, 제어 클래스인 ProductManagement가 경계 클래스인 MenuForm을 부분으로 가지는 것은 자연스럽지 않다. 연관 관계에 대한 제약 사항에서 설명을 하겠지만, 제어 클래스와 경계 클래스 간의 관계는 집합 관계보다는 연관 관계로 표현하는 것이 타당하다.

3.2.2.2 일반화 관계 제약 사항

표현. $\forall c_1, c_2 \in CS, c_1 \rightarrow_g c_2$ implies $st(c_1) = st(c_2)$

설명. 일반화 관계는 동일한 스테레오 타입의 클래스 간에만 성립된다. 일반화 관계는 보다 일반적 대상을 나타내는 분류자와 보다 구체적인 대상을 나타내는 분류자 간의 관계로 UML에서는 정의되고 있다. 집합 관계에 관련된 두 대상이 유사한 성격을 가지는 클래스이어야 하는 것처럼 일반화 관계를 가지는 부모 클래스의 자식 클래스는 유사한 성격 즉 동일한 스테레오 타입을 가져야 한다. 이 제약 사항은 UML 표준에도 well-formedness rule로서 포함되고 있다.

```
context Generalization
inv GeneralizationConstraint:
    let childClass : Class = self.child.oclAsType(Class)
    let parentClass : Class = self.parent.oclAsType(Class)
    st(childClass) = st(parentClass)
```

그림 5 일반화 관계 제약 사항에 대한 OCL 식

그림 5는 일반화 스테레오 타입 제약 사항을 OCL 식으로 표현한 것이다. 일반화 제약 사항의 OCL 식은 Generalization 문맥 하에서 부모 클래스와 자식 클래스의 스테레오 타입이 동일함을 나타낸다. 위 OCL 식에서 childClass와 parentClass는 일반화 관계를 맺고 있는 자식 클래스와 부모 클래스를 나타낸다.

그림 2의 분석 모델에는 세 개의 일반화 관계가 존재한다. MenuForm 클래스와 CommonForm 클래스 간의 일반화 관계는 두 클래스가 모두 경계 클래스이므로 이 제약 사항을 준수한다. 마찬가지로, UserInfo 클래스와 CustomerInfo 클래스는 모두 개체 클래스이므로 제약 사항을 준수하고 있다. 그러나, UserAuthentification 클래스와 ValidUserForm 클래스는 서로 다른 스테레오 타입을 가지므로 두 클래스 간의 일반화 관계는 제약 사항을 위반하고 있다.

3.2.2.3 연관 관계 제약 사항

집합 관계와 일반화 관계가 동일한 스테레오 타입의 클래스 간에만 성립되는 것과 달리 연관 관계는 다른 스테레오 타입의 클래스 간에도 존재할 수가 있다. 그러나, 연관 관계는 분석 클래스의 종류에 따라서 연관 관계의 존재가 불허되거나 연관 관계의 방향성이 제한될 수 있다. 예를 들면, 경계 클래스와 개체 클래스 사이에는 연관 관계가 존재하지 않는다. 그리고, 제어 클래스에서 개체 클래스로의 연관 관계는 허용되지만 반대로 개체 클래스에서 제어 클래스로의 연관 관계는 성립되지 않는다.

표현.

AC(1) $\forall a \in AT, c \in CT \cup ET$, not $a \rightarrow_s c$ and not $c \rightarrow_s a$

AC(2) $\forall b \in BD, e \in ET$, not $b \rightarrow_s e$ and not $e \rightarrow_s b$

AC(3) $\forall c \in CT, e \in ET$, not $e \rightarrow_s c$

설명. AC(1)은 액터는 제어 클래스 및 개체 클래스와 연관 관계를 가질 수 없음을 뜻한다. 즉, 액터는 다른 액터 또는 경계 클래스 와만 연관을 맺을 수가 있다. 이는 분석 모델에서 경계 클래스가 시스템 외부 즉 액터와의 인터페이스를 전담하고 다른 유형의 클래스(제어 클래스와 개체 클래스)는 경계 클래스를 통해서만 액터와 상호 작용할 수 있다는 원칙에 따른 것이다. AC(2)는 경계 클래스와 개체 클래스 간에는 연관 관계가 허용되지 않음을 나타낸다. 경계 클래스는 외부와의 인터페이스 즉 입력/출력 만을 담당하며 개체 클래스를 접근하는 로직을 제공하지 않기 때문이다. 개체 클래스는 비즈니스 로직을 제공하는 제어 클래스를 통하여 접근된다. AC(3)은 개체 클래스에서 제어 클래스 방향으로의 연관 관계가 없음을 뜻한다. 개체 클래스는 수동적인 클래스로서 제어 클래스로부터 요청 받은 데이타 조작 관련된 개체 클래스의 연산만을 수행하며 제어 클래스에게 어떤 행위를 수행하도록 요청하지는 않기 때문이다[3].

그림 6는 연관 관계 제약 사항을 OCL 식으로 표현한 것이다. stWithAs(Association)은 주어진 연관 관계를 맺고 있는 모든 분류자의 스테레오 타입 집합을 구한다. ptWithAs(Association)은 그림 4에서 정의된 것으로서 주어진 연관 관계의 모든 분류자 집합을 구하는 것이다.

```
context Association
inv AssociationConstraint:
    let stWithAs(as:Association) : Set(String) = ptWithAs(as)->iterate(
        c : Classifier ; result : Set(String) = Set{} |
        if c.oclIsTypeOf(Class) then result->including(st(c))
        else result
        endif
    )
    let AC1 = ptWithAs(self)->exists(oclIsTypeOf(Actor)) implies
        stWithAs->excludesAll(Set{"control", "entity"})
    let AC2 = not stWithAs(self)->includesAll(Set{"boundary", "entity"})
    let AC3 = stWithAs(self)->includesAll(Set{"control", "entity"}) implies
        self.connection->select(isNavigable=true)->forAll(st(participant) <> "control")
    AC1 and AC2 and AC3
```

그림 6 연관 관계 제약 사항에 대한 OCL 식

AC1은 연관된 분류자 중에서 액터가 있을 때는 동일한 연관 관계에는 제어 클래스와 개체 클래스가 연관되어 있지 않음을 뜻한다. AC2는 경계 클래스와 개체 클래스가 함께 관련된 연관 관계가 없음을 나타낸다. AC3는 제어 클래스와 개체 클래스가 참여하는 연관 관계가 있을 때 연관이 방향이 제어 클래스가 아님을 의미한다. 그리고, 연관 관계 제약 사항은 이 세가지 제약 사항을 조합하여 정의된다.

그림 2의 분석 모델에 존재하는 많은 수의 연관 관계 중에서 네 개의 연관 관계는 이 제약사항을 위반하고 있다. 액터에서 제어 클래스로의 연관은 허용되지 않기 때문에 Customer 액터에서 OrderManagement 제어 클래스로의 연관 관계는 AC(1)을 위반한다. 마찬가지로 UserInfo 개체 클래스에서 ValidationSystem 액터로의 연관 관계도 AC(1)을 위반한다. UserMenuForm 클래스에서 OrderInfo 클래스로의 연관 관계는 AC(2)를 위반한 예이다. 그리고, OrderManagement 클래스와 OrderInfo 클래스 사이의 연관 관계는 AC(3)을 위반하고 있다.

3.2.2.4 의존 관계 제약 사항

표현. $\forall c_1, c_2 \in CS$, not $c_1 \rightarrow_d c_2$

설명. 분석 모델에서는 클래스 간의 의존 관계가 사용되지 않는다. 의존 관계는 설계 모델에서 연산의 인자로 사용되는 클래스, 연산 내부의 지역 객체를 생성할 때 사용되는 클래스와의 관계를 표현할 때 사용된다. 그리고, 분석 모델에서는 클래스 사이에서가 아니라 패키지 간에 사용되며 이는 한 패키지의 클래스가 다른 패키지의 클래스를 접근함을 의미한다. 요약하면, 클래스 간의 의존 관계는 설계 모델에서 사용되며 분석 모델에서는 사용되지 않는다.

그림 7은 의존 관계 제약 사항에 대한 OCL 식을 보여 준다. 이 OCL 식은 의존 관계에 참여하는 모델 요소 중에는 클래스가 존재하지 않음을 나타낸다.

```
context Dependency
inv DependencyConstraint:
    supplier->forAll(me:ModelElement | not me.oclIsTypeOf(Class)) and
    client->forAll(me:ModelElement | not me.oclIsTypeOf(Class))
```

그림 7 의존 관계 제약 사항에 대한 OCL 식

3.2.3 사용 관련 제약 사항

시스템의 기능은 분석 클래스 간의 협력을 통하여 제공되며, 이를 유즈케이스 실현(Use Case Realization)이라고도 부른다. 이런 측면에서 보면, 분석 클래스는 시스템의 기능을 실현하는 데 반드시 기여를 할 수 있어야 한다. 시스템 기능에 대한 분석 클래스의 기여 측면에서 네 가지의 사용 제약 사항을 소개한다.

3.2.3.1 액터 사용 제약 사항

표현. $\forall a \in AT, \exists b \in BD, a \Rightarrow^* b$ or $b \Rightarrow^* a$

설명. 액터는 반드시 한 개 이상의 경계 클래스와 연결되어야 한다. 액터는 시스템과 상호작용을 하는 시스템 외부의 존재이고 경계 클래스는 시스템의 다른 클래스를 대표하여 액터와의 상호작용을 구현하는 클래스이다. 따라서, 액터는 반드시 한 개 이상의 경계 클래스와 직접적 또는 간접적으로 연관 관계를 맺어야 한다. 그렇지 않다면, 액터는 시스템의 기능과는 아무런 관련이 없으므로 모델에서 제거되어야 한다.

그림 8은 액터 사용 제약 사항에 대한 OCL 식을 보여 준다. associatedSet()은 주어진 분류자와 연관 관계를 가지면서 연관 관계의 방향성(outgoing), 제외될 분류자 집합(visited), 그리고 스테레오 타입(startingST와 endingST)를 만족시키는 분류자 집합을 구한다. outGoing 인자는 주어진 분류자로부터의 연관 관계를 대상으로 하는 지 또는 주어진 분류자 방향으로의 연관 관

```
context Classifier
inv ActorUsage:
  let associatedSet(c:Classifier, outGoging:Boolean, visited:Set(Classifier),
    startingST:String, endingST:String) : Set(Classifier) =
    let targetSTs : Set(String) =
      if st(c) = startingST then Set{startingST, endingST}
      else Set{endingST}
      endif
    if outGoging = true then
      c.association.association.connection->select(isNavigable=true)->
        collect(participant)->select(ac:Classifier|ac <> c and
        visited->excludes(ac) and targetSTs->includes(st(ac)))->asSet()
    else
      c.association->select(isNavigable=true).association.connection->
        collect(participant)->select(ac:Classifier|ac <> c and
        visited->excludes(ac) and targetSTs->includes(st(ac)))->asSet()
    endif
  let generalizedSet(c:Classifier, visited:Set(Classifier)) : Set(Classifier) =
    c.generalization->collect(parent->oclAsType(Classifier))->asSet()->
      union(c.specialization->collect(child->oclAsType(Classifier))->asSet())->
      select(gc:Classifier|gc <> c and visited->excludes(gc))
  let adjacentSet(c:Classifier, outgoing:Boolean, visited:Set(Classifier),
    startingST:String, endingST:String) : Set(Classifier) =
    let as = associatedSet(c, outgoing, visited, startingST, endingST)
    let gs = generalizedSet(c, visited)
    as->union(gs)
  let connectedSet(c:Classifier, outgoing:Boolean, visited:Set(Classifier),
    startingST:String, endingST:String) : Set(Classifier) =
    let adjacent = adjacentSet(c, outgoing, visited, startingST, endingST)
    if adjacent->size() = 0 then visited->including(c)
    else
      let newVisited = visited->including(c)
      adjacent->iterate(adc:Classifier ;
      resultSet : Set(Classifier) = visited->including(c) |
        let adj2 = connectedSet(adc, outgoing, newVisited, startingST, endingST)
        resultSet->union(adj2) )
    endif
  st(self) = "actor" implies
    connectedSet(self, true, Set{}, "actor", "boundary")->
      exists(c:Classifier|st(c) = "boundary") or
    connectedSet(self, false, Set{}, "actor", "boundary")->
      exists(c:Classifier|st(c) = "boundary")
```

그림 8 액터 사용 제약 사항에 대한 OCL 식

계를 대상으로 할지를 지정한다. visited 인자는 이미 조사된 분류자를 중복해서 조사하지 않도록 제외될 분류자 집합이다. generalizedSet()은 주어진 분류자와 일반화 관계를 맺고 있는 분류자를 구한다. associatedSet()과 마찬가지로 이미 조사된 분류자들을 제외시키기 위해서 visited가 인자로 주어진다. 그리고, 일반화 관계의 두 클래스는 동일한 스테레오 타입을 가진다는 일반화 관계 제약을 가정하여 스테레오 타입에 대한 조사는 생략하였다. adjacentSet()은 associatedSet()과 generalizedSet()을 이용하여 주어진 분류자와 연관 관계 또는 일반화 관계를 맺고 있는 분류자들을 구한다. 분석 모델에서는 클래스 간에 의존 관계가 사용되지 않

으므로 adjacentSet()에서는 클래스 간의 의존 관계를 포함하지 않는다. connectedSet()은 adjacentSet()을 반복적으로 적용하면서 주어진 분류자와 간접적으로 연결된 모든 분류자를 구한다. 마지막 OCL 식은 현재 분류자가 액터인 경우에는 액터를 시점으로 직/간접적으로 연결된 경로에 있는 분류자에 경계 클래스가 있거나 액터를 종점으로 직/간접적으로 연결된 경로에 경계 클래스가 있음을 의미한다. 즉, 주어진 액터에서 경계 클래스로의 직/간접적인 연결이 있거나, 어떤 경계 클래스에서 주어진 액터로의 직/간접적인 연결이 있음을 뜻한다.

그림 2의 분석 모델에는 두 개의 액터가 있다. Customer 액터는 MenuForm 경계 클래스와는 직접적으로 연결되고, CommonForm, BodyForm 경계 클래스와는 MenuForm을 통하여 간접적으로 연결되었다. 따라서, Customer 액터는 액터 사용 제약 사항을 만족시킨다. 반면에, ValidationSystem 액터는 연결된 경계 클래스를 가지고 있지 않다. <LoginForm, User-Authenitification, UserInfo, ValidationSystem>의 경로가 있지만, UserAuthentification과 UserInfo 클래스가 스테레오 타입에 대한 조건을 충족시키지 않으므로 LoginForm $\Rightarrow^*$ ValidationSystem이 성립되지 않는다. ValidationSystem 액터가 연결된 경계 클래스가 없는 것은 해당 액터가 시스템과 무관한 것이므로 제외되든가 아니면 액터와의 인터페이스를 담당하는 경계 클래스가 누락된 것이므로 적절한 경계 클래스를 추가하든지 해야 한다.

3.2.3.2 경계 클래스 사용 제약 사항

표현. $\forall b \in BD,\ \exists a \in AT,\ b \Rightarrow^* a$ or $a \Rightarrow^* b$

설명. 경계 클래스는 하나 이상의 액터와 연결되어야 한다. 경계 클래스는 외부 액터와의 인터페이스를 제공한다. 따라서, 경계 클래스는 인터페이스의 대상이 되는 하나 이상의 액터와 관계를 맺어야 한다. 그렇지 않다면, 인터페이스를 제공할 외부 대상이 없으므로 해당 경계 클래스는 무의미해진다.

그림 9의 BoundaryClassUsage 불변식은 경계 클래스 사용 제약 사항을 OCL 식으로 표현한 것이다. 현재 클래스가 경계 클래스인 경우에 한 개 이상의 액터로부터의 연결이 있거나 또는 한 개 이상의 액터로의 연결이 있음을 의미한다. 이 OCL 식에서 connectedSet()은 액터 사용 제약 사항의 OCL 식에서 정의된 것과 동일하다. 예를 들어, 그림 2의 분석 모델에서 MenuForm 경계 클래스는 Customer 액터로부터의 직접적인 연관 관계를 가진다. 연결의 정의에 따라서 경계 클래스와 액터 간의 간접적인 관계도 포함된다. 예를 들어, CommonForm과 BodyForm 경계 클래스는 MenuForm 경계 클래스를 통하여 Customer 액터로부터의 연결이 성립된다. 그러나, UserMenuForm과 MenuForm 간의 연관 관계의 방향성 때문에 UserMenuForm은 간접적으로 Customer 액터로부터 연결이 될 수가 없다. 또한, LoginForm, ValidatingSystemAgent, ValidUserForm 경계 클래스는 연결된 액터가 없으므로 모두 경계 클래스 사용 제약 사항을 위반하고 있다.

```
context Class
inv BoundaryClassUsage:
  st(self) = "boundary" implies
    connectedSet(self, false, Set{}, "boundary", "actor")
      ->exists(c:Classifier|st(c) = "actor") or
    connectedSet(self, true, Set{}, "boundary", "actor")
      ->exists(c:Classifier|st(c) = "actor")

inv ControlClassUsage:
  st(self) = "control" implies
    connectedSet(self, false, Set{}, "control", "boundary")
      ->exists(c:Classifier|st(c) = "boundary")

inv EntityClassUsage:
  st(self) = "entity" implies
    connectedSet(self, false, Set{}, "entity", "control")
      ->exists(c:Classifier|st(c) = "control")
```

그림 9 경계/제어/개체 클래스 사용 제약 사항에 대한 OCL 식

3.2.3.3 제어 클래스 사용 제약 사항

표현. $\forall c \in CT,\ \exists b \in BD,\ b \Rightarrow^* c$

설명. 제어 클래스는 반드시 한 개 이상의 경계 클래스에 의해서 연결되어야 한다. 제어 클래스는 시스템이 제공할 제어 로직 또는 비즈니스 로직을 실제로 구현하는 클래스이다. 그리고, 액터는 경계 클래스를 통하여 제어 클래스가 제공하는 기능을 이용한다. 만약 제어 클래스로 연결된 경계 클래스가 없다면 해당 제어 클래스는 시스템에서 이용되지 않는 고립된 클래스가 된다. 따라서, 제어 클래스가 시스템의 기능을 제공하는 데 기여를 하기 위해서는 한 개 이상의 경계 클래스로부터의 연결이 존재해야 한다.

그림 9의 ControlClassUsage 불변식은 제어 클래스 사용 제약 사항을 OCL 식으로 표현한 것이다. 이 불변식은 현재 클래스가 제어 클래스인 경우에 제어 클래스의 방향으로 들어오는 직/간접적으로 연결된 경계 클래스가 존재함을 뜻한다. 예를 들어, 그림 2의 분석 모델에는 5개의 제어 클래스가 존재한다. UserAuthentification과 OrderManagment 제어 클래스는 각각 LoginForm과 MenuForm 경계 클래스로부터의 직접적인 연결이 존재한다. 그리고, OrderHistoryManagement 제어 클래스는 OrderManagement 제어 클래스를 경유

하여 MenuForm 경계 클래스로부터의 간접적인 연결이 성립된다. 그러나, ProductManagement 클래스에서 MenuForm 클래스 방향으로의 집합 관계이기 때문에 ProductManagement 제어 클래스는 자신을 이용하는 경계 클래스가 없으므로 제어 클래스 사용 제약 사항을 위반한다. 또한, ItemManagement 제어 클래스는 연결된 경계 클래스가 없으므로 이 제약 사항을 위반한다.

3.2.3.4 개체 클래스 사용 제약 사항

표현. $\forall e \in ET, \exists c \in CT, c \Rightarrow^* e$

설명. 개체 클래스는 반드시 한 개 이상의 제어 클래스에 의해서 연결되어야 한다. 개체 클래스는 자신이 나타내는 영속적인 정보에 대한 관리 기능을 제공한다. 그리고, 개체 클래스는 경계 클래스에 의해서 직접적으로 접근되지 않고 제어 클래스를 통하여 이용된다. 즉, 제어 클래스가 비즈니스 로직을 제공하는 과정에서 영속적인 데이타 관리 부분은 해당 개체 클래스를 통하여 수행하는 것이다. 그러므로, 만약 개체 클래스를 이용하는 제어 클래스가 없다면 그 개체 클래스는 시스템의 기능에 전혀 기여하지 않는 무의미한 클래스가 된다. 따라서, 개체 클래스는 반드시 한 개 이상의 제어 클래스로부터의 연결이 존재하여 시스템의 기능에 기여할 수 있어야 한다.

그림 9의 EntityClassUsage 불변식은 개체 클래스 사용 제약 사항을 OCL 식으로 표현한 것이다. 이 불변식은 현재 클래스가 개체 클래스인 경우에 개체 클래스의 방향으로 들어오는 직/간접적으로 연결된 제어 클래스가 존재함을 뜻한다. 예를 들어, 그림 2의 분석 모델에는 6개의 개체 클래스가 존재한다. UserInfo, OrderInfo, ItemInfo 개체 클래스는 각각 UserAuthentification, OrderManagement, ItemManagement 제어 클래스로부터 직접적으로 연결된다. 그리고, CustomerInfo와 OrderItemInfo 개체 클래스는 다른 개체 클래스를 통하여 간접적으로 제어 클래스로부터의 연결이 성립된다. 그러나, BillInfo 개체 클래스는 연결된 제어 클래스가 존재하지 않으므로 이 제약 사항을 위반하고 있다.

4. 사례 연구

이 장에서는 본 논문에서 제시한 방법을 실제 산업체에서 수행된 2개의 프로젝트에 적용한 사례를 소개한다. 우선, 사례 연구를 수행한 환경을 설명한 후에 사례 연구의 결과를 소개한다.

4.1 사례 연구 환경

사례 연구는 한 SI 업체에서 실제로 수행한 2개의 정보 시스템을 대상으로 하였다. 두 개의 정보 시스템은 모두 대규모 은행의 방카슈랑스 시스템 구축 프로젝트로서 시스템 개발 중의 분석 모델을 수집하였다. 표 1은 2개의 대상 시스템에 대한 개요를 보여 준다. 편의상 두 개의 시스템을 시스템-A와 시스템-B로 부르겠다. 각 시스템의 분석 모델로부터 액터, 경계 클래스, 제어 클래스, 개체 클래스의 수로부터 시스템의 규모를 짐작할 수 있다. 그리고, 분석 모델은 구현 플랫폼에 무관하지만 시스템-A는 .NET 플랫폼에서 구축하였으며 시스템-B는 J2EE 플랫폼에서 구축하였음을 기술하였다.

표 1 대상 시스템 개요

시스템	액터 수	경계 클래스 수	제어 클래스 수	개체 클래스 수	플랫폼
시스템-A	27	76	51	55	.NET
시스템-B	19	110	71	83	J2EE

주어진 OCL 식을 분석 모델에 대해서 평가하기 위하여 OCL Evaluator 도구[22]를 사용하였다. OCL Evaluator 도구는 Babes-Bolyai 대학의 LCI에서 개발한 도구로서 OCL 식에 대한 평가를 지원하는 도구이다. OCLE 도구는 XMI 형식의 UML 모델과 OCL 식을 읽어서 OCL 식이 주어진 UML 모델에서 만족되는 지를 조사한다. 그리고, 만족되지 않는 OCL 식 목록을 출력해 준다.

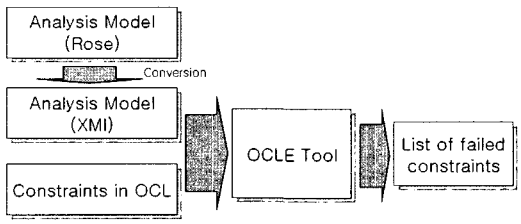


그림 10 OCLE를 이용한 제약 사항의 평가

그림 10은 OCL Evaluator 도구를 이용하여 사례 연구의 대상 분석 모델에 대한 검증을 수행한 과정을 보여 준다. 프로젝트에서는 UML 모델을 작성하기 위하여 Rational Rose를 사용하였다. 그리고, OCL Evaluator 도구는 Rose 모델을 직접적으로 읽지 못하기 때문에 Rose 모델을 XMI 형식으로 전환해야 한다.[1)]

그림 11은 OCL Evaluator 도구를 이용하여 그림 2의 분석 모델을 대상으로 구조적 제약 사항을 검증하는 모습을 보여 준다. OCL Evaluator 도구의 우측의 클래스 다이어그램에는 그림 2의 분석 모델이 표시되어 있고 텍스트 편집 창에는 본 논문에서 제시한 제약 사항

1) Rose의 Add-In 인 Unisys의 XMI export 모듈은 한글을 지원하지 않는다. 그래서, 본 사례 연구에서는 Unisys의 XMI export 모듈을 사용하는 대신에 Rational XDE 도구를 이용하여 Rose 모델을 XMI 파일로 변환하였다.

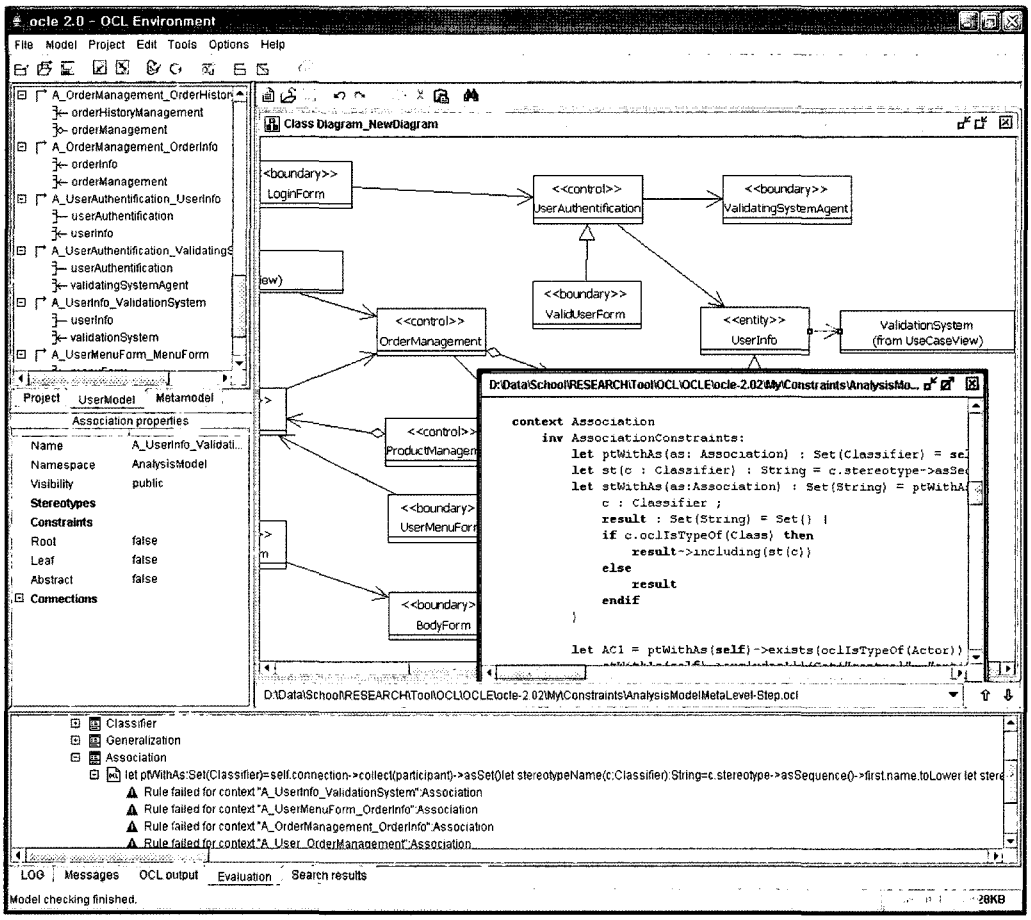


그림 11 OCL Evaluator 도구를 이용한 제약 사항 검증

에 대한 OCL 식이 있다. 화면의 하단의 evaluation 창에는 충족되지 않은 OCL 식을 문맥 별로 구분하여 보여 준다. Evaluation 창에서 OCL 식을 위반한 실제 모델 요소를 선택하면 좌측 상단의 UserModel 창과 우측의 클래스 다이어그램은 해당 모델 요소가 부각되어 표시된다.

4.2 사례 연구 결과

2개의 대상 시스템에 대하여 수행된 사례 연구의 결과는 표 2와 같다. 이 표는 각 시스템에서 제약 사항 별로 제약 사항의 위반 횟수를 보여 준다.

- 스테레오 타입 제약 사항

  두 개의 시스템에서는 모두 스테레오 타입 제약 사항의 위반 사례를 발견하지 못하였다. 즉, 두 시스템의 분석 모델에서는 경계 클래스, 제어 클래스, 개체 클래스 유형의 클래스만을 사용하였다.

- 관계 관련 제약 사항

  두 시스템은 집합 관계 제약 사항과 연관 관계 제약 사항은 만족시킨다. 즉, 동일한 스테레오 타입의 두 클래스 사이에서만 집합 관계를 사용하였다. 그리고, 연관 관계는 본 논문에서 제시한 대로 경계 클래스와 제어 클래스 사이에 그리고 제어 클래스에서 개체 클래스로의 연관 관계를 사용하였다. 표 2에 따르면 스테레오 타입을 포함한 집합 관계와 연관 관계에서 오

표 2 사례 연구 결과 요약

제약 사항	시스템-A	시스템-B
스테레오 타입 제약 사항	0	0
집합 관계 제약 사항	0	0
일반화 관계 제약 사항	12	0
연관 관계 제약 사항	0	0
의존 관계 제약 사항	46	10
액터 사용 제약 사항	13	19
경계 클래스 사용 제약 사항	76	107
제어 클래스 사용 제약 사항	2	6
개체 클래스 사용 제약 사항	0	11

류가 발견되지 않았다. 이에 대한 이유는 두 시스템의 개발 시 적용된 개발 방법론이 RUP를 바탕으로 하고 있으며, 이 제약 사항은 해당 SI 업체의 개발 방법론에서 기존으로 강조되는 부분이었기 때문에 이 부분에 대해서는 오류가 발견되지 않은 것으로 추정된다. 반면에 일반화 관계 제약 사항과 의존 관계 제약 사항을 위반하는 많은 사례가 발견되었다. 시스템-A의 분석 모델을 조사한 결과 시스템-A의 개발자는 Rose 도구에 익숙하지 않아서 부정확한 일반화 관계를 삭제할 때 분석 모델에서 완전히 삭제한 것이 아니라, 클래스 다이어그램 상에서만 삭제한 것에 기인함을 알았다. 이것이 개발자가 도구에 대한 경험이 없어서 범한 단순한 실수라고 볼 수도 있지만, 본 논문에서 제시한 방법을 통하여 부적절한 일반화 관계를 파악하였고 이를 제거하여 모델을 수정하도록 개발자에게 권하였다. 두 시스템에서 의존 관계 제약 사항의 위반 사례가 많이 발견되었다. 분석 모델을 살펴본 결과 개발자가 UML 모델링에 대한 경험이 부족하여 연관 관계를 사용해야 하는 곳에 의존 관계를 사용한 것임을 알았다. 이것 또한 오류를 수정하도록 개발자에게 권고하였다.

- 사용 관련 제약 사항

표 1에 표시된 액터 및 경계 클래스의 수와 비교하면 대부분의 액터와 경계 클래스가 사용 제약 사항을 위반하였다. 즉, 대부분의 액터에 연결된 경계 클래스가 없었으며 경계 클래스의 경우에도 연결된 액터를 가지지 않은 경우가 많았다. 두 시스템의 분석 모델을 조사해본 결과 개발자들이 액터와 경계 클래스 간의 연관 관계를 클래스 다이어그램에 명시적으로 표현하는 대신에 각 유즈케이스 별 실현 모델 즉 시퀀스 다이어그램에 액터와 경계 클래스 간의 메시지 전송을 표현함을 알았다. 따라서 분석 모델에서 액터와 경계 클래스 간의 연결이 없다고 실제로 액터와 경계 클래스 간의 연결에 대한 제약을 위반하는 것으로 판단하는 것은 부적절하다. 따라서, 액터 사용 제약 사항과 경계 클래스 사용 제약 사항은 시퀀스 다이어그램까지 고려하도록 확장될 필요가 있음을 알았다. 반면에 제어 클래스 제약 사항과 개체 클래스 제약 사항에 대한 위반은 분석 모델 상의 심각한 오류로 판단될 수 있다. 예를 들어, 시스템-B에서는 이용되지 않는 제어 클래스가 6개 그리고 개체 클래스가 11개 있었으며, 이는 모델링 상의 명백한 오류로 간주되었다.

두 개의 실제 프로젝트에 본 논문에서 제시한 기법을 적용한 사례 연구에서 발견된 제약 사항의 위반은 일부분은 개발자의 실수 또는 개발자의 Rose 도구에 대한 경험 부족에 기인하거나 또는 액터/경계 클래스 사용 제약 사항과 같이 너무 엄격한 제약 사항을 적용했기 때문인 면도 있다. 그러나, 제어/개체 클래스 사용 제약 사항의 위반은 명백한 오류로 간주될 수 있다. 본 사례 연구에서는 분석 모델을 대상으로 제시된 제약 사항을 검증하였으며, 발견된 제약 사항에 대한 위반 사항을 개발자에게 통보하여 이를 수정하도록 요구하였다.

5. 관련 연구

OCL을 이용한 UML 모델의 검증과 관련된 여러 연구들이 존재한다. Richters와 Gogolla[7,9]는 UML 모델의 OCL 제약 검증을 위하여 애니메이션 기반 기법을 제안하였다. 그들은 USE 도구를 사용하였으며, USE 도구는 UML 모델을 시뮬레이션하기 위한 애니메이션 기능과 제약 사항 검사를 위하여 OCL 인터프리테이션 기능을 지원한다. 그들은 USE 도구를 UML 1.3 메타모델과 well-formedness rule에 적용하였으며, UML 1.3 메타모델 상의 몇 가지 오류를 지적하였다. Ziemann과 Gogolla[11]는 USE 도구를 이용하여 OCL로 기술된 safety properties를 검증하는 방법을 제안하였다. 그리고, Ol'kovich 등[8]은 OCL 식에 대한 검증을 지원하는 프로그램 모듈을 자동으로 생성하는 연구를 하였다.

스테레오 타입을 정형화하고 이를 UML 모델 검증에 적용한 연구도 있다. Schleicher와 Westfechtel[23]은 스테레오 타입의 활용 방법 중의 하나로서 OCL 식으로 제약사항을 부가적으로 기술하는 제한적인 스테레오타입에 대해서 소개하였다. 또한 UML 메타모델을 바탕으로 스테레오 타입의 의미를 OCL 식으로 정형화를 하는 연구도 있었다[5,6]. 그리고, Ziadi[24]등은 플랫폼에 무관한 컴포넌트 모델을 제안할 때 컴포넌트 모델의 아키텍처 측면에서의 제약 사항을 OCL 식으로 기술하였다.

또한, 일관성(consistency)에 초점을 둔 많은 연구[15-18]들이 존재한다. 특히, [11,13]에서는 본 논문에서 다룬 Unified Process를 대상으로 UML 모델 간의 일관성을 OCL로 기술하고 검증하는 방법을 소개하고 있다. Unified Process를 대상으로 OCL을 이용한다는 측면에서는 본 논문과 유사하다. 그러나, [12,13]은 분석 모델 뿐만 아니라 시스템 모델과 유즈케이스 모델도 대상으로 하기 때문에, 본 논문에서 제시된 것과 같이 분석 모델이 준수해야 하는 비교적 상세한 수준의 제약 사항을 제시하진 못한다.

본 논문에서 사용한 OCL 기반의 모델 검증 기법이 새로운 시도는 아니며, 기존에 제안된 OCL을 이용한 UML 모델 검증 기법과 공통되는 면이 있다. 그러나, 본 논문은 제시한 일련의 구조적 제약 사항은 분석 모델을 대상으로 하며 구현 환경에 독립적인 분석 모델은 다양한 구현 환경의 시스템을 구축할 때 재사용될 수

있는 이점이 있다. 따라서, 제시된 구조적 제약 사항은 분석 모델의 기능적 측면의 정확성을 검증하는 것은 아니지만, 특정 기능에 대한 구체적인 검증 수준은 아니지만, 분석 모델 고유의 특성을 반영한 제약 사항을 통하여 분석 모델에 대한 기본적인 검증을 수행하였다. 또한, 본 논문에서 제시된 제약 사항은 일반적인 객체지향 개발 방법론에서 사용되는 분석 모델에 적용될 수 있도록 고안되었다. 특히 정보 시스템에 적합할 수가 있다. UML에서는 분석 모델에 대한 표준으로서 경계 클래스, 개체 클래스, 제어 클래스에 대한 정의는 하고 있지만 이들의 의미, 즉 적절한 사용 방식을 본 논문에서 제시된 제약 사항으로서 정의하고 있지 않다.[2] 또한, 기존의 OCL을 이용한 UML 모델 검증에 관한 연구들은 대부분 예제 수준의 모델을 대상으로 수행되었으며 실제로 구축된 규모 있는 시스템을 대상으로 적용된 사례 연구를 소개하고 있지 않다. 그러나 본 논문에서는 실제 SI 업체에서 수행한 시스템 구축 프로젝트에 적용하여 제시된 구조적 제약 사항의 역할을 파악하려는 시도를 하였다.

6. 결론 및 향후 연구 방향

본 논문에서는 다양한 시스템에서 일반적으로 사용될 수 있는 플랫폼 독립적인 분석 모델의 검증을 위하여 객체지향 분석 모델이 충족시켜야 할 구조적 제약 사항을 OCL로 기술하고 이를 조사하는 기법을 소개하였다. 본 논문에서 제안한 제약 사항은 객체지향 분석 모델 고유의 특성을 반영하였으며, 정확한 분석 모델의 구축을 위한 최소한의 기준으로 사용될 수 있다. 또한, 실제 시스템에 대한 사례 연구를 통하여 제안된 구조적 제약 사항이 분석 모델의 오류를 발견하는 데 도움을 줄 수 있음을 확인하였다.

본 논문에서 소개한 방법은 다음과 같은 방향으로 연구가 발전될 수 있다. 우선 제시된 제약 사항에 대한 검증 기능을 UML 모델링 도구에 내장함으로써 개발자가 분석 모델을 작성하는 중에 제약 사항을 위반하는 상황을 탐지하고 경고를 함으로써 분석을 완성한 후에 검증하는 대신에 모델 작성 중에 제약 사항을 강제화할 수가 있다. 또한, 본 논문에서 제시한 기법을 분석 모델이 아니라 다른 도메인의 모델을 대상으로 적용하는 것도 가능하다. 예를 들면, 임베디드 소프트웨어의 경우에는 장치 드라이버, 태스크 등과 같은 전형적인 클래스를 스테레오 타입으로 정의할 수 있다. 그리고, 그 전형적인 클래스의 고유의 특성을 파악하여 OCL로 기술할 수가 있다. 특정 도메인에 대하여 스테레오 타입을 도출하고 제약사항을 결정하기만 하면 이를 OCL로 기술하고 검증하는 것은 본 논문에서 소개한 절차와 동일하다.

2) 일반화 관계에 대한 제약 사항을 UML에 포함되어 있다.

참 고 문 헌

[1] I. Jacobson, et al, The Unified Software Development Process, Addison-Wesley, 1999.

[2] OMG. MDA Guide Version 1.0.1. OMG Document formal/03-06-01, 2003.

[3] OMG. UML 1.4 Specification. OMG Document formal/04-07-02, 2002.

[4] J. Warmer and A. Kleppe, The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 1998.

[5] M. Gogolla, "Using OCL for Defining Precise, Domain-Specific UML Stereotypes," Proc. 6th Australian Workshop on Requirement Engineering, 2001.

[6] M. Gogolla and B. Henderson-Sellers, "Analysis of UML Stereotypes within the UML Metamodel," Proc. 5th Conf. Unified Modeling Language, pp. 84-99, 2002.

[7] M. Gogolla, Jörn and M. Richters, "Validation of UML and OCL Models by Automatic Snapshot Generation," Proc. 6th Conf. Unified Modeling Language, pp. 265-279, 2002.

[8] L. Ol'khovich and D. V. Koznov, "OCL-Based Automated Validation Method for UML Specifications," Programming and Computer Science, pp. 323-327, 2003.

[9] M. Richters and M. Gogolla, "Validating UML Models and OCL Constraints," Proc. 3rd Int. Conf. Unified Modeling Language, pp. 265-277, 2000.

[10] P. Selonen and J. Xu, "Validating UML Models Against Architecture Profiles," ESEC/FSE '03, pp. 58-67, 2003.

[11] P. Ziemann and M. Gogolla, "Validating OCL Specifications with the USE Tool - An Example Based on the BART Case Study," Proc. 8th Int. Workshop on Formal Methods for Industrial Critical Systems, 2003.

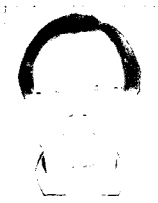
[12] B. Hnatkowska, Z. Huzar, L. Kuzniarz, and L. Tuzinkiewics, "A Systematic Approach to Consistency Within UML based Software Development Process," Workshop on Consistency Problems in UML-based Software Development, Oct. 2002.

[13] B. Hnatkowska and A. Walkowiak, "Consistency Checking of USDP Models," Workshop on Consistency Problems in UML-based Software Development, Oct. 2004.

[14] I. Kuzniarz, G. Reggio, J. Sourrouille, and Z. Huzar, Workshop on Consistency Problems in UML-based Software Development, UML 2002, 2002.

[15] I. Kuzniarz, G. Reggio, J. Sourrouille, Z. Huzar, and M. Staron, Workshop on Consistency

Problems in UML-based Software Development II, UML 2003, 2003.
[16] Z. Huzar, I. Kuzniarz, G. Reggio, and J. Sourrouille, Workshop on Consistency Problems in UML-based Software Development III, UML 2004, 2004.
[17] M. Elaasar and L. Briand, "An Overview of UML Consistency Management," Technical Report SCE-04-18, Dept. of Systems and Computer Engineering, Carleton University, Canada.
[18] M. Benattou, J-M. Bruel, and N. Hameurlain, "Generating Test Data from OCL Specification," Workshop on Integration and Transformation of UML Models, 2002.
[19] I. Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992.
[20] Jim Arlow and Ila Neustadt, UML and the Unified Process: Practical Object-Oriented Analysis & Design, Addison-Wesley, 2002.
[21] D. Rosenberg and K. Scott, Applying Use Case Driven Object Modeling with UML: An Annotated E-Commerce Example, Addison-Wesley, 2001.
[22] OCL Evaluator http://lci.cs.ubbcluj.ro/ocle
[23] A. Schleicher and B. Westfechtel, "Beyond Stereotyping: Metamodeling Approaches for the UML," Proc. of 34th Hawaii Int. Conf. on System Sciences, 2001.
[24] T. Ziadi, B. Traverson and J-M. Jézéquel, "From a UML Platform Independent Component Model to Platform Specific Component Models," Workshop in Software Model Engineering, 2002.



채 홍 석

1994년 서울대 원자핵공학 학사. 1996년 한국과학기술원 전산학 석사. 2000년 한국과학기술원 전산학 박사. 2000년~2003년 (주)동양시스템즈 기술연구소 선임연구원. 2003년~2004년 한국과학기술원 전산학과 초빙교수. 2004년~현재 부산대학교 컴퓨터 공학과 전임강사. 관심분야는 객체지향 방법론, 소프트웨어 테스팅, 소프트웨어 메트릭, 소프트웨어 유지보수

염 근 혁

1985년 2월 서울대학교 계산통계학과(학사). 1992년 8월 Univ. of Florida 컴퓨터공학과(석사). 1995년 8월 Univ. of Florida 컴퓨터공학과(박사). 1985년 1월~1988년 2월 금성반도체 컴퓨터연구실 연구원. 1988년 3월~1990년 6월 금성사 정보기기연구소 주임연구원. 1995년 9월~1996년 8월 삼성 SDS 정보기술연구소 책임연구원. 1996년 8월~현재 부산대학교 컴퓨터공학과 부교수. 부산대학교 컴퓨터 및 정보통신 연구소 연구원. 관심분야는 소프트웨어 재사용, 프로덕트라인 공학, 소프트웨어 아키텍쳐, 컴포넌트 기반 소프트웨어 개발, 적응형 소프트웨어 개발, RFID기반 미들웨어 등임