객체지향 설계에서 정형명세를 이용한 컴포넌트 설계로의 변환 기법
(Techniques to Transform Object-oriented Design into Component-based Design Formal Specifications using Formal Specifications)

신 숙 경 [†] 이 종 국 [††] 김 수 동 [†††]
(Suk Kyung Shin) (Jong Kook Lee) (Soo Dong Kim)

요 약 재사용성과 확장성을 높이는 객체지향 개발이 보편화 되면서 새로운 소프트웨어를 개발할 경우 기 개발되어 검증된 객체지향 산출물을 재사용함으로써 개발기간을 단축하고 품질을 향상할 수 있다. 이렇게 성능이 검증된 기 개발된 객체지향 산출물을 이용하여 컴포넌트 기반 모델로 변환하면 짧은 기간에 고품질의 컴포넌트 기반 시스템을 구축할 수 있다.

본 논문에서는 이미 개발되어 있는 객체지향 설계 모델을 이용하여 컴포넌트 기반 설계로 변환하되 변환의 정확성을 위해 정형명세 기법을 사용한다. 컴포넌트 기반 설계를 정형명세하기 위해 컴포넌트 정형명세 언어를 정의한다. 그리고 객체지향 설계의 정적, 동적, 기능적 측면을 정형명세 언어 Object-Z를 사용하여 정형명세하는 기법을 제시한 후, 객체지향 정형명세를 컴포넌트 정형명세로 변환하는 기법을 제시한다. 사례연구는 제시된 변환 기법을 적용하여 객체지향 정형명세가 컴포넌트 기반 정형명세로의 변환과정을 설명한다.

키워드 : 객체지향 설계, 컴포넌트 설계, 정형명세, 프로그램 변환

Abstract As object-oriented development technology that increases extensibility and reusability has been widely spread, it can shorten development period and enhance quality by reusing verified object-oriented artifacts. Thus we can construct high quality component-based system at short time transforming component-based model using verified object-oriented artifacts.

In this paper, we propose techniques to transform available object-oriented design model into component-based model using formal specification techniques in order to increase accuracy of transformation. First, formal specification language for component is defined for formal specification of component-based design. And, techniques for formal specification of object-oriented design using Object-Z, a formal specification language, is proposed in structural, functional, and dynamic aspects. Next, we present techniques for transforming formal specification of object-oriented design into formal specification of component-based design. Through a case study we apply the proposed transformation techniques and show the transformation process of object-oriented formal specification into component-based formal specification.

Key words : Object-oriented design, Component-based design, Formal specification, Program transformation

1. 서 론

1.1 동기

객체지향 기술은 추상화, 캡슐화 및 상속성 등의 개념을 기반으로 확장성과 재사용성을 높이는 핵심 기술로서 현재 소프트웨어 개발 관련 기술의 근간이 되고 있다. 객체지향 기술의 또 다른 특징은 프로그램을 뚜렷하게 구별되는 단위로 분할할 수 있다는 것이다. 구별된

· 본 연구는 숭실대학교의 교내연구비 지원으로 이루어졌음
† 정 회 원 : 숭실대학교 컴퓨터학과
skshin@otlab.ssu.ac.kr
†† 비 회 원 : 숭실대학교 컴퓨터학과
jklee690@disc.co.kr
††† 종신회원 : 숭실대학교 컴퓨터학과 교수
sdkim@ssu.ac.kr
논문접수 : 2003년 9월 1일
심사완료 : 2004년 4월 8일

단위들은 잘 정의된 인터페이스를 이용하여 상호 작용할 수 있다. 이렇게 되면 큰 시스템에 대한 요구를 잘 분할하여 생각해 볼 수 있고 잘 분할된 시스템은 수정할 때 그 영향력이 적어지므로 변경 작업이 용이하다. 코드 재사용에 의하여 프로그램 생산성을 높이고 변경이 쉬워지며 일관된 소프트웨어 개발 모델을 제공하는 객체지향 기술은 소프트웨어 생산 기술에서 매우 중요한 위치를 차지하고 있다. 특히 객체지향 기술에 대한 관심이 프로그래밍에서 분석, 설계로 옮겨지면서 Object Modeling Group(OMG)의 모델링 언어인 Unified Modeling Language(UML)이 널리 보편화되어 사용되고 있다.

객체지향 패러다임에 뒤이어 등장한 컴포넌트 기반의 개발(Component-Based Development, CBD)은 블럭을 조립하듯 소프트웨어 시스템을 조립하자는 개념이다. 컴포넌트 기반의 소프트웨어 개발을 일컫는 CBD는 이미 개발된 소프트웨어 컴포넌트를 조립하거나 재사용해 소프트웨어 시스템을 개발하는 방법으로, IT 자산의 재활용성을 높이고 기업의 투자대비 수익을 높여주는 개념으로 최근 각광받고 있다.

컴포넌트는 소프트웨어 개발과 유지보수의 비용을 줄이고 개발 생산성을 향상시킬 수 있는 효과적인 방법으로 평가받고 있다[1]. 컴포넌트의 장점은 소프트웨어 개발의 복잡성을 효과적으로 관리할 수 있다는 것과 빠른 시간에 제품을 출시해야 하는 상황에 적합하다는 것, 개발이 계속 될수록 생산성이 향상된다는 것, 품질이 향상된다는 것, 그리고 재사용성이 좋다는 것들이다[2]. 이러한 컴포넌트의 특성을 성공적으로 적용한 사례들이 계속 발표되고 있다[3,4]. 그러나 컴포넌트가 소프트웨어 개발에 성공적으로 적용되기 위해서는 해결해야 할 과제들이 많다. 첫째는 컴포넌트 개발 비용이 기존의 소프트웨어 개발 비용보다 크다는 것이다. 두 번째 문제는 컴포넌트는 소스 코드를 알 수 없기 때문에 명확한 컴포넌트 명세서가 필요하다는 것이다. 명확하고 이해하기 쉬운 컴포넌트 명세서는 컴포넌트에 대한 신뢰성을 높이고 컴포넌트의 조립을 통한 시스템 개발을 용이하게 한다.

그러므로, 위에서 지적한 문제점을 해결하고 효율적인 컴포넌트 기반 시스템을 설계하기 위해서는 기 개발되어 있는 객체지향 설계 모델을 이용하여 컴포넌트 기반 설계로 변환하는 기술이 요구된다. 기 개발되어 있는 객체지향 산출물을 재사용하면 이미 품질이 검증되었기 때문에 설계의 정확성이 높고, 시스템의 개발기간을 단축할 수 있다. 그리고 설계 명세의 정확성을 통하여 에러를 찾아낸다는 것은 시스템의 전체 품질에 매우 중요한 의미를 갖는다. 객체지향 설계 산출물과 컴포넌트 산출물의 정확성을 기하고 변환작업의 정확성을 기하기 위해서는 정형명세 기법을 사용하여야 한다.

1.2 논문의 범위와 구성

본 논문의 접근 방법은 그림 1과 같다. 객체지향 설계의 구성요소를 제시한 후, 정형명세 언어인 Object-Z를 이용하여 객체지향 설계를 정형적으로 명세한다. 그리고, 객체지향 설계의 정형명세(Formal Specification of Object-Oriented Design)를 이용하여 컴포넌트 기반 설계의 정형명세(Formal Specification of Component Based Design)로 변환하는 기법을 제시한다. 이때, 앞에서 정의한 컴포넌트 정형명세 문법을 기준으로 변환한다. 사례연구는 제안한 변환기법을 적용하여 객체지향 정형명세를 컴포넌트 정형명세로 변환한다. 이렇게 생성된 컴포넌트 기반 설계의 정형명세는 그림 1의 점선부분처럼 추후 컴포넌트 기반 설계로 변환하는 과정을 거쳐 EJB, CORBA 등의 컴포넌트 기반 시스템을 구현하는 단계까지 확장할 수 있다.

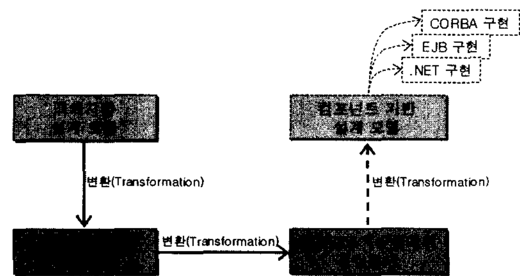


그림 1 논문의 접근 방법

본 논문의 범위는 다음과 같다. 2장에서는 관련 연구 분야로서 객체모델과 정형명세 언어, 컴포넌트 참조 모델 등에 대해 알아본다. 3장은 컴포넌트 기반 설계의 구성요소를 기준으로 컴포넌트 정형명세 언어를 정의한다. 4장은 객체지향 설계의 구성요소를 기준으로 정형명세 언어인 Object-Z를 이용하여 객체지향 설계를 정형명세하는 기법을 제시한다. 5장은 객체지향 설계의 정형명세를 컴포넌트 기반 설계의 정형명세로 변환하는 기법을 제시한다. 6장은 변환 툴을 소개하고, 7장은 사례 연구로서 객체지향의 정형명세를 기준으로 5장에서 제시한 변환기법에 의해 어떻게 컴포넌트 기반 정형명세로 변환되는지를 확인해 본 후, 8장에서 결론을 도출한다.

2. 기반 연구

2.1 객체지향 설계모델

객체지향 개발 방법은 객체의 정적인 정보에 객체의 동작을 추가시켜 객체를 완벽하게 기술하고 구현하는 방법으로 Coad, Yourdon, Jacobson 등 여러 가지 방법론이 있으나 Rumbaugh의 객체 모델링 기법(Object

Modeling Technique)에 의하면 요구사항을 분석하기 위해 객체의 서로 다른 관점인 정적인 측면과, 동적인 측면, 그리고 기능적인 측면에서 분석한다.

1) 정적 모델링: 시스템에서 요구되는 객체를 찾아내어 객체들의 특성과 객체들 사이의 관계를 규명하는 것으로 UML의 클래스 다이어그램 및 컴포넌트 다이어그램으로 표현한다.

2) 동적 모델링: 정적 모델링에서 규명된 객체들의 행위와 객체들의 상태를 포함하는 라이프 사이클을 보여주는 단계로 UML의 시퀀스 다이어그램 및 상태 다이어그램으로 표현한다.

3) 기능 모델링: 각 객체의 형태 변화에서 새로운 상태로 들어갔을 때 수행되는 동작들을 기술하는 데 사용하며 UML의 유즈케이스 다이어그램과 활동 다이어그램으로 표현한다.

객체지향 설계모델은 앞에서 언급한 세가지 모델링 측면을 단계별로 적용하여 그 결과를 통합하여 구성한다.

2.2 정형명세 언어

컴포넌트를 정형적으로 명세하는 언어는 여러 종류가 있다. 아키텍처 명세 언어(Architecture Description Language)는 소프트웨어 시스템의 구조와 행동을 개괄적으로 묘사하여 컴포넌트를 명세하는 데도 유용하다[5]. 그러나 아키텍처 명세 언어는 단지 컴포넌트의 추상적인 면만을 묘사할 수 있으며 컴포넌트의 중요한 특성인 가변성이나 컴포넌트 내부 구조, 인터페이스를 통한 컴포넌트 개발자와 사용자 사이의 계약 등을 명세하기에는 한계가 있다.

Componentware에서 제시한 정형 명세 언어는 컴포넌트와 인터페이스, 다른 컴포넌트, 주고 받는 메시지 사이의 관계를 기술하는데 중점을 두고 있다. 그러나 Componentware는 컴포넌트가 가져야 할 선행, 후행 조건이나 불변 조건을 기술하기 위한 장치가 부족하다.

Piccola는 PI calculus와 PI-L calculus를 기초로 만들어졌다[6]. Piccola는 컴포넌트의 컴포지션(Composition)과 인터페이스의 확장성을 명세하기에 좋은 언어이다. 그러나 Piccola는 컴포넌트를 명세하기 위한 기본적인 언어 장치만을 제공하고 있으며 컴포넌트와 인터페이스를 어떻게 명세할 것인지는 언급하고 있지 않다.

Object-Z 언어는 Z 언어[7]를 확장하여 클래스뿐 아니라 컴포넌트를 기술하는 데도 적합한 언어이다[8,9]. Object-Z에서는 스키마를 포함하는 클래스를 명세하며 클래스를 포함하는 클래스를 명세할 수 있다. 클래스를 포함하는 컴포넌트는 상속의 개념을 사용하여 간결하게 명세할 수 있다. 또 하나의 장점은 객체 지향 명세가 Enterprise Java Beans(EJB)나 .NET, COM 등의 구현 언어로 쉽게 매핑될 수 있다는 것이다. 그러나 Object-Z는 인터페이스와 컴포넌트의 분리를 명확히 보여줄 수 없으며 컴포넌트 가변성을 표현하기 어렵다.

그러므로, Object-Z는 객체지향 시스템을 정형명세하기에 적합하게 만들어진 언어이나 컴포넌트 기반 시스템을 명세하기에는 한계를 가지고 있다. 컴포넌트 정형명세 언어는 컴포넌트와 인터페이스, 컴포넌트 간의 관계를 기술할 수 있을 뿐 아니라 객체의 추상화와 상속의 속성을 지원하여 객체지향 정형명세와도 매핑이 용이하여야 하므로 본 논문에서 정의하여 표현한다.

2.3 CORBA 컴포넌트 모델(CORBA Component Model, CCM)

CORBA 컴포넌트는 CORBA3의 3가지 주요 항목에, 인터넷 통합 서비스 제어의 높은 질, 코바 컴포넌트 구조와 함께 포함된다. CORBA 컴포넌트는 기존의 EJB와 비슷한 모형을 가지고 있다. 컴포넌트의 생성, 서비스, 소멸의 전 단계를 통제하는 컨테이너(Container) 안에서 홈 객체에 의해서 객체들이 생성되고 컨테이너에서 관리되며, 각각의 컴포넌트들은 인터페이스만을 통하여 서로 호출할 수 있다.

CORBA에서 컴포넌트는 메타 타입(Meta-Type)이다. 컴포넌트 메타 타입은 객체 메타 타입의 확장이며 상속된 것이다. 컴포넌트라 해서 특별한 사용 방법이 있는 것이 아니고 객체를 사용하는 방법과 같다. 컴포넌트 타입들은 클라이언트와 서버간의 인터페이스를 정의한 Interface Definition Language(IDL)에 명세되어 있으며 인터페이스 저장소(Interface Repository)에 저장된다. 인터페이스 저장소에는 컴포넌트가 제공하는 메소드들에 관한 정보들이 저장된다. 향후에 웹 서비스(Web Service)를 할 때 중요하다. 네트워크를 통해 인터페이스 저장소에 있는 정보를 보고 웹 서비스가 제공하는 서비스를 제공받을 수 있기 때문이다.

CCM의 컴포넌트 모델은 컴포넌트 모델이 단순하면서도 쉽게 확장될 수 있는 모델이기 때문에 본 논문의 컴포넌트 메타모델 정의시 참조한다.

3. 컴포넌트 기반 설계의 정형명세 언어

본 장에서는 컴포넌트 기반 설계의 주요한 산출물인 컴포넌트의 구성요소를 정의하고, 구성요소를 중심으로 컴포넌트 기반 설계를 정형적으로 명세할 언어를 제안한다. 분류된 구성요소는 기존의 객체 지향 설계의 산출물을 이용하여 컴포넌트를 명세하는 데 필요한 요소가 된다. 그림 2는 컴포넌트 기반 설계의 메타 모델을 보여준다. 이는 설계단계에서의 관점을 중심으로 시스템을 구성하는 요소들 간의 관계를 표현한 UML의 클래스 다이어그램이다. 그림 2에서 점선 안의 모델은 컴포넌트

메타 모델로 컴포넌트를 구성하는 요소들을 나타낸다.

그림 2를 보면 컴포넌트는 클래스(Class), 인터페이스(Interface), 가변성(Variability), 워크플로우(Workflow) 요소로 이루어진다. 컴포넌트는 하나 이상의 클래스로 구성된다. 구성된 클래스가 컴포넌트의 명세부분에서 선언된 인터페이스를 구현한다. 클래스는 관련된 속성(Attribute)과 오퍼레이션으로 구성된다. 따라서, 클래스를 통해 컴포넌트의 구현이 이루어지면, 컴포넌트의 내부 구현은 숨긴 채, 인터페이스를 통해서 클라이언트에게 컴포넌트의 기능 혹은 서비스를 제공한다.

인터페이스는 Provide 인터페이스와 Required 인터페이스, 그리고 Uses 인터페이스로 구성된다. Provide 인터페이스는 컴포넌트가 다른 컴포넌트 즉, 클라이언트에게 서비스를 제공하는 역할을 한다. Required 인터페이스는 컴포넌트의 가변성을 설정한다. 가변성이란 사용자의 설정에 의해서 컴포넌트 내부 구성이나 워크플로우가 변하는 것을 의미한다. 따라서 컴포넌트의 가변성은 컴포넌트 내부 클래스에 표현될 것이다. Required 인터페이스의 호출에 의해 가변성이 설정되면 컴포넌트 내부 클래스 중 실행 가능한 클래스가 일부분으로 제한될 것이다. Uses 인터페이스는 서비스를 제공하는 컴포넌트로부터 필요한 서비스를 요청하는 역할을 한다.

가변성이란 제품 패밀리 멤버들 간에 존재하는 차이로서[10], 컴포넌트들 사이의 차이나 컴포넌트 기반 애플리케이션 사이의 차이라고 할 수 있다. 가변성의 종류는 속성과 로직(Logic), 그리고 워크플로우가 있다. 여기에서 속성 가변성은 의미적으로 로직 가변성에 포함될 수 있으므로 본 논문에서는 로직과 워크플로우 가변성에 대해서만 정의한다. 컴포넌트 기반 애플리케이션에서 가변성이 일어나는 위치에 대한 식별자를 가변점(Variation Point)이라 한다. 가변점에 통합되어질 수 있는 해결책으로, 하나의 가변점에 대해서 가변값(Variant)은 여러 개 존재한다.

컴포넌트 내부 객체 간의 흐름을 나타내는 워크플로우는 컴포넌트 인터페이스의 단위로 설계된다. 이렇게 비즈니스 오퍼레이션을 수행하기 위해서 하나의 컴포넌트 내부 객체들 간의 메시지 흐름을 마이크로 워크플로우(Micro Workflow)라고 정의한다. 컴포넌트 간의 흐름은 컴포넌트 기반 시스템에서 컴포넌트들 간에 인터페이스의 오퍼레이션을 호출하는 것으로 설계된다. 이렇게 시스템 오퍼레이션을 수행하기 위한 컴포넌트들 간의 메시지 흐름을 매크로 워크플로우(Macro Workflow)라 한다.

컴포넌트 정형명세 언어는 Object-Z 언어의 문법에 기초하고 있다. 컴포넌트 기반 시스템은 인터페이스와 인터페이스 그룹(Interface Group), 컴포넌트, 시스템으로 구성된다.

3.1 인터페이스의 명세

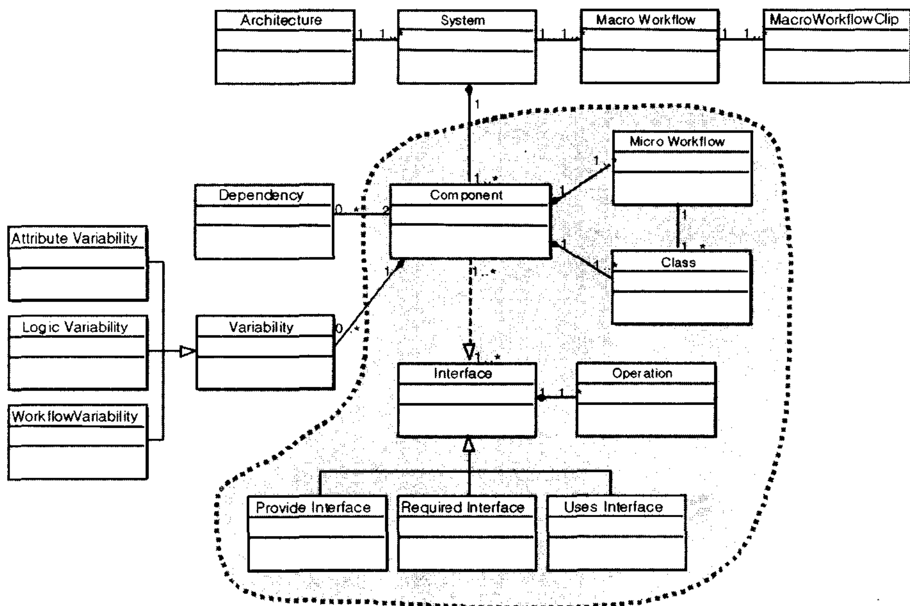
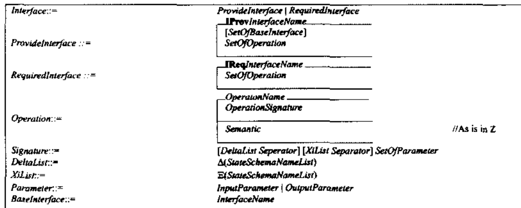


그림 2 컴포넌트 기반 설계의 메타 모델

컴포넌트 인터페이스는 컴포넌트에 의해 제공되는 서비스로, 오퍼레이션의 집합이다. 그러므로 인터페이스 스키마는 Z 언어의 오퍼레이션 스키마(Operation Schema) 형태로 정의된다.



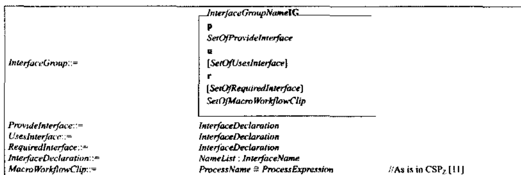
위의 문법에서 *StateSchemaName*은 인터페이스 오퍼레이션에서 참조하는 상태 스키마의 이름을 가리킨다. *Semantic*은 오퍼레이션의 선행, 후행, 불변 조건을 가리킨다. *InputParameter*는 입력 매개변수의 선언, *OutputParameter*는 출력 매개 변수 선언을 가리킨다.

Provide 인터페이스는 컴포넌트와 독립적으로 기술된다. Provide 인터페이스는 Z 스키마로 정의되고, 다른 스키마와의 구분을 위하여 스키마명 앞에 "IProv"를 붙인다.

Required 인터페이스는 가변성을 기술하기 위해 어떤 타입의 입력 매개변수를 받을 수 있는지를 기술한다. Required 인터페이스 또한 Z 스키마로 정의되고, 다른 스키마와의 구분을 위하여 스키마명 앞에 "IReq"를 붙인다.

3.2 인터페이스 그룹의 명세

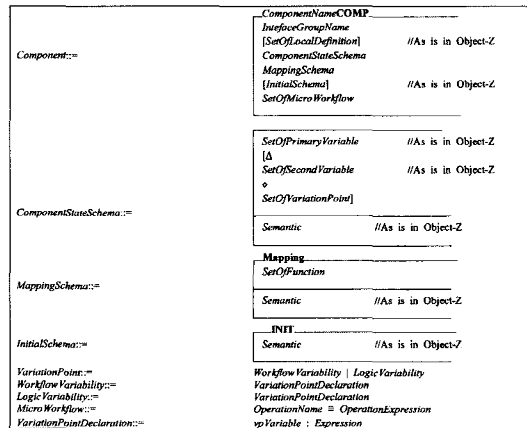
인터페이스 그룹은 하나의 컴포넌트와 연관된 Provide 인터페이스와 Required 인터페이스, 그리고 다른 컴포넌트의 기능을 사용하기 위해 필요한 Uses 인터페이스 목록을 정의한다. **p, u, r**은 세 종류의 인터페이스를 구별하는 구분자이다. 또한 컴포넌트 간의 메시지흐름을 표현하는 매크로 워크플로우(*MacroWorkflowClip*)를 정의하는데, 이것은 Object-Z에 CSP를 접목하여 프로세스의 흐름을 표현한다.



3.3 컴포넌트의 명세

인터페이스가 컴포넌트의 외부에서 컴포넌트의 행위를 기술하는 반면, 컴포넌트에 대한 명세는 컴포넌트의 내부를 기술한다. 컴포넌트 내부 명세는 Object-Z의 명세와 유사하게 속성에 대한 선언과 오퍼레이션 스키마로 이루어진다. 내부 컴포넌트는 컴포넌트에서 정한 Provide 인터페이스의 모든 오퍼레이션을 오퍼레이션 스키마에서 명세해야 한다. 컴포넌트 내부에 나타나는 오퍼레이션 스키마는 시그너처(Signature)가 인터페이스 명세의 오퍼레이션 스키마와 동일해야 한다.

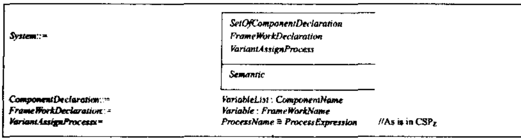
컴포넌트의 이름은 **'COMP'**로 끝난다. *MicroWorkflow*는 컴포넌트 내부의 객체 상호 작용을 표현한다. *LocalDefinitions, InitialSchema, OperationExpression*는 Object-Z의 문법과 동일하다. *ComponentStateSchema*에서 ◇는 가변점을 상태 변수와 구별하는 구분자이다. 가변점은 오퍼레이션의 술어나 *MicroWorkflow*에 사용될 수 있다. 가변점이 오퍼레이션의 술어에서 사용될 경우는 논리 가변성이고 *Microworflow*에서 사용될 경우는 워크플로우 가변성이다. 가변점의 이름은 *vp*로 시작하여 상태 변수와 구별한다. 인터페이스와 내부 객체들간의 관계는 매핑 스키마(*MappingSchema*)로 기술하고 매핑 스키마에서 인터페이스에서 사용한 타입, 변수와 컴포넌트에서 사용하는 타입, 변수와의 관계도 정의한다. 매핑 스키마는 한 인터페이스 스키마의 변수와 속성을 컴포넌트 명세에 연결한다.



3.4 시스템의 명세

시스템은 개발하려는 대상 애플리케이션으로 컴포넌트 기반 설계 메타모델에서 구성요소를 모두 포함하는 명세의 최상 단위이다. 또한 시스템은 Namespace의 역할을 한다.

시스템은 컴포넌트와 인터페이스를 통해 기술되며 시스템이 클래스를 직접 사용하는 경우는 없다. 또한 컴포넌트는 내부에 클래스를 포함하며 컴포넌트가 내부에 컴포넌트를 포함하는 경우는 없다. *FrameWorkDeclaration*은 인터페이스 그룹간의 상호작용을 명세하고, *VariantAssignProcess*는 Required 인터페이스의 오퍼레이션을 사용하여 가변점에 가변값을 할당해주는 시퀀스(Sequence)를 표현한다.



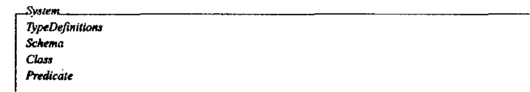
4. 객체지향 설계의 정형명세화 기법

객체 지향 방법론에서는 시스템의 요구사항을 분석하기 위해서 정적인 측면, 동적인 측면, 그리고 기능적인 측면을 분석한다. 이와 같이 세가지 관점을 고려하여 객체지향 설계의 구성요소를 추출하기 위한 메타모델을 표현하면 그림 3과 같다. 세가지 관점을 나타내는 UML 다이어그램은 여러 형태가 있으나, 대표적으로 기능적인 측면은 유즈케이스 다이어그램으로 표현하고, 정적인 측면은 클래스 다이어그램, 그리고 동적인 측면은 시퀀스 다이어그램으로 표현한다.

4.1 시스템의 명세

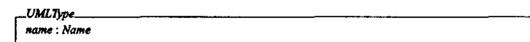
시스템은 개발하려는 대상 애플리케이션을 가리킨다. 시스템은 그림 3의 메타 모델의 구성요소를 모두 포함하는, 명세의 최상 단위이다. Object-Z에서는 스키마를 포함하는 클래스를 명세하고, 클래스를 포함하는 클래스를 명세할 수 있기 때문에, 명세 대상이 무한히 커질 수 있다. 그러나 명세의 간결함을 위해서 시스템을 명세한 후 클래스와 스키마 등을 명세한다.

시스템은 Namespace의 역할을 하며 한 시스템 안에서는 스키마와 클래스, 변수명이 중복될 수 없다. 시스템의 구조는 Object-Z 언어와 같은 구조를 가지며, 그 형태는 다음과 같다.

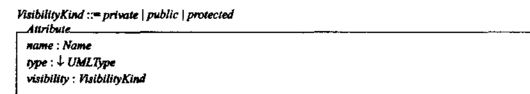


4.2 객체 모델(Structural Model)의 명세

Object-Z 안에서 클래스의 구성요소들을 표현하면 그림 4와 같다. 이것은 [12]에서 제시한 모델을 참조로 하여 메타모델로 표현한 것이다. UMLType은 UML에서 가능한 모든 타입을 상속하는 추상 클래스이다. 클래스도 UMLType의 한 형태이기 때문에 UMLType으로부터 상속 받는다. 이를 Object-Z로 표현하면 다음과 같다. OtherType은 Object-Z에서 제공하는 멱집합(Power Set)이나, 카테션(Cartesian), 스키마(Schema) 등과 같은 형태를 말한다.



속성은 이름과 타입, Visibility로 구성되어 있다. Visibility의 종류는 Private, Public, Protected가 있다.



파라미터는 이름과 타입으로 구성되고, 속성처럼 다음

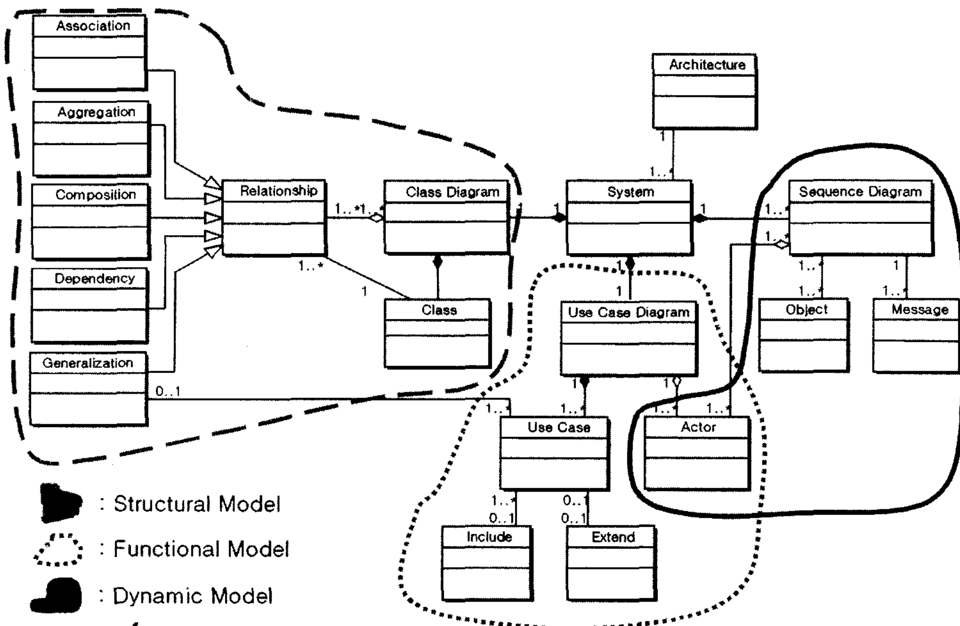


그림 3 객체지향 메타 모델

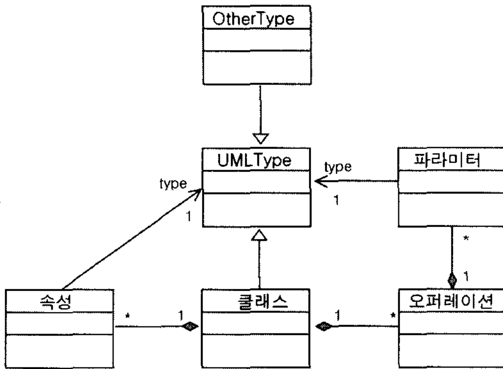


그림 4 Object-Z에서 클래스 구성요소의 관계

과 같이 정의된다.

```
┌ Parameter ──────────────────────
│ name : Name
│ type : ↓ UMLType
└─────────────────────────────────
```

오퍼레이션은 이름과 Visibility, 파라미터를 갖는다. 한 오퍼레이션 내에서 파라미터는 유일한 이름을 가져야 한다.

```
┌ Operation ──────────────────────
│ name : Name
│ visibility : VisibilityKind
│ parameters : seq Parameter ©
├─────────────────────────────────
│ ∀p1,p2 : ran parameters • p1.name = p2.name ⇒ p1 = p2
└─────────────────────────────────
```

Object-Z에서 Containment 관계를 표현하는 ⓒ는 그 타입에 소속되는 것으로, *Parameter*ⓒ는 파라미터 타입에 속하는 것을 말한다.

클래스는 UmlType으로부터 상속받은 타입으로 이름과 속성, 오퍼레이션으로 구성된다. 클래스 안에 정의된 속성 이름은 서로 달라야 하고, 오퍼레이션도 서로 다른 시그너처를 갖는다. 이런 특성을 Object-Z로 표현하면 다음과 같다.

```
┌ Class ──────────────────────────
│ UMLType
│ attributes : 𝔽Attribute ©
│ operations : 𝔽Operation ©
├─────────────────────────────────
│ ∀a1,a2 : attributes • a1.name = a2.name ⇒ a1 = a2
│ ∀op1,op2 : operations •
│     ( op1.name = op2.name ⇒ a1 = a2 ∧ #op1.parameters = #op2.parameters ∧
│     ∀i : 1..#op1.parameters •
│         op1.parameters(i).name = op2.parameters(i).name ∧
│         op1.parameters(i).type = op2.parameters(i).type) ⇒ op1 = op2
└─────────────────────────────────
```

클래스의 구성요소들을 Object-Z 클래스 스키마 정의와 매핑하면 그림 5와 같다.

위 구조를 사용하여 클래스를 Object-Z로 정형명세하기 위해 변환하는 규칙은 다음과 같다.

1) Object-Z 클래스명은 클래스명과 동일하게 사용한다.
2) Object-Z의 Visibility List에는 UML에서 Visibility가 Public 성격을 가진 속성이나 오퍼레이션만 매핑된다.
3) Inherited Class는 UML에서 상속하는 Super Class

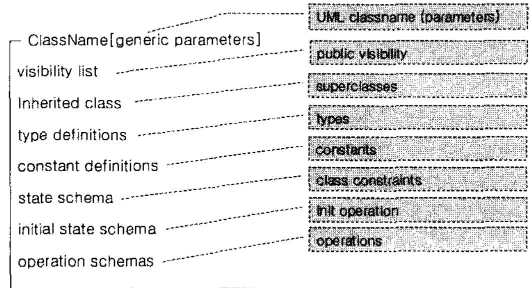


그림 5 Object-Z 클래스와 클래스 구성요소의 매핑

와 매핑된다.

4) Type Definitions과 Constant Definitions은 클래스 안에서 사용되는 타입과 상수를 정의한다. Type Definitions과 매핑되는 것은 클래스에서 속성의 타입 정의 부분이다.
5) State Schema는 선언부와 Predicate부로 나뉜다. 선언부에서는 상태변수를 정의하므로 클래스의 속성과 매핑되고, Predicate부에는 변수의 제한사항을 정의하므로 클래스의 속성이나 클래스 제한사항과 매핑된다.
6) Initial State 스키마는 클래스의 초기상태를 명시하는 것으로 클래스의 Init 오퍼레이션과 매핑된다.
7) 오퍼레이션 스키마는 클래스의 오퍼레이션과 매핑된다.

4.3 기능적 모델(Functional Model)의 명세

Booch[13]에 따르면 액터는 유즈케이스와 상호작용할 때 유즈케이스의 이용자가 행동하는 역할과 밀착된 집합을 나타낸다고 했다. 유즈케이스 메타모델은 액터와 유즈케이스, 그리고 그들 간의 관계를 보여준다. 유즈케이스 다이어그램은 하나 이상의 액터와 유즈케이스로 구성된다. 하나의 액터는 하나 이상의 유즈케이스와 관계를 갖고, 하나의 유즈케이스 역시 하나 이상의 액터와 관계를 갖는다. 액터와 유즈케이스의 관계는 Association으로 표현하고, 유즈케이스와 유즈케이스의 관계는 Include와 Extend가 있다. Include 관계는 공통되는 기능을 재사용하기 위한 방법으로 상속관계의 일종이라고 할 수 있다. Extend 관계는 기존의 유즈케이스를 확장하기 위해 사용된다. [6]에 나타난 메타모델을 참조하여 유즈케이스 다이어그램의 메타모델을 표현하면 그림 6과 같다.

유즈케이스 다이어그램은 액터와 유즈케이스, 그리고 그들간의 관계로 구성되므로 다음과 같이 나타낸다.

```
┌ UseCaseDiagram ─────────────────
│ actor : ACTOR
│ usecase : USECASE
├─────────────────────────────────
│ actor <relation> usecase
└─────────────────────────────────
```

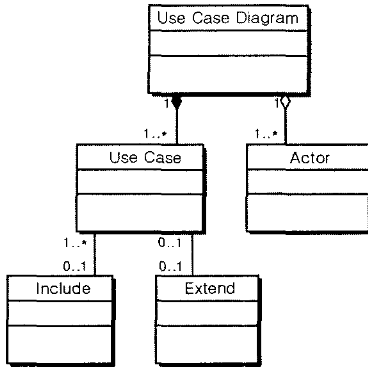


그림 6 유즈케이스 다이어그램의 메타모델

유즈케이스는 시스템이 수행하는 일련의 액션들의 집합으로 액터와의 관계를 표현하고 유즈케이스 명세(Description)를 이용하여 Pre-Condition을 스키마 안에 Predicates로 표현할 수 있다. 각 유즈케이스 명세는 클래스 스키마에 표현된 오퍼레이션 스키마와 중복되므로, 각 유즈케이스에 대한 스키마명 뒤에 "UC"를 붙여 클래스 스키마와 구분한다.

```
UseCaseUC
actors : ℙ ACTOR
scenarios : ℙ seq Action
────────
pre-condition
```

유즈케이스 간의 관계인 Include와 Extend를 [14]를 참조하여 정형명세하면 다음과 같다.

```
Include
include : UseCase ↔ UseCase
scenarios : ℙ seq Action
────────
∀ u1, u2 : UseCase • u1 include u2 ⇔
    ∀ d2 : u2.scenarios • ∃ d1 : u1.scenarios • d2 in d1
```

```
Extend
extend : UseCase ↔ UseCase
scenarios : ℙ seq Action
────────
∀ u1, u2 : UseCase • u1 extend u2 ⇔
    ∀ d2 : u2.scenarios • ∃ d1 : u1.scenarios • d2 isSubseqOf d1
```

4.4 동적 모델(Dynamic Model)의 명세

동적 모델을 대표적으로 표현하는 시퀀스 다이어그램은 시간적인 순서로 객체들 사이에 보내지는 메시지의 상호작용을 보여준다. 시퀀스 다이어그램은 하나의 유즈케이스 안에서 기능을 수행하기 위해 객체들 간에 어떻게 상호작용 하는지를 나타내고, 클래스의 기능과 인터페이스를 결정하기 위한 기본적인 자료를 제공한다. 시퀀스 다이어그램은 액터와 객체, 메시지, 시간적 처리흐름으로 구성되어 있다. 이를 메타모델로 표현하면 다음 그림 7과 같다.

시퀀스 다이어그램은 객체 사이의 관계를 메시지 순열로 표현하므로 다음과 같이 정형명세된다.

```
SequenceDiagram
object : OBJECT
message : MESSAGE
interaction : seq object.message
```

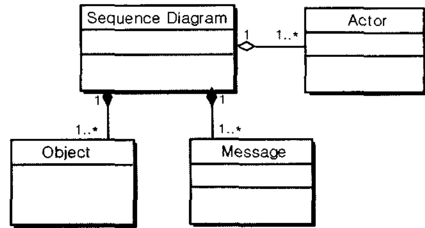


그림 7 시퀀스 다이어그램의 메타모델

객체와 객체 사이의 상호작용은 메시지 전달을 통하여 이루어진다. 메시지의 흐름은 시퀀스로 표현하고 메시지와 메시지의 구분은 Sequential Composition Operator ⨟로 표현한다. 그러므로 각 유즈케이스에 대한 시퀀스는 다음과 같이 표현된다. 이 명세는 각 유즈케이스 명세서와 이름이 중복되는 것을 방지하기 위해서 스키마명 뒤에 "SD"를 붙여 구분한다.

```
UseCaseNameSD
object : ℙ OBJECT
message : ℙ MESSAGE
────────
object1.message1 ⨟ object2.message2 ⨟ object3.message3
```

5. 객체지향 정형명세에서 컴포넌트 정형명세로의 변환 기법

3장에서 컴포넌트 기반 설계의 메타모델 구성요소를 기준으로 정형명세 언어를 제시하였고, 4장에서는 객체지향 설계의 메타모델 구성요소를 기준으로 정형명세화 기법을 제시하였다. 본 장에서는 3장의 컴포넌트 정형명세 문법을 기준으로 객체지향 정형명세에서 컴포넌트 정형명세로의 변환기법을 제시한다.

객체지향 설계를 이용하여 컴포넌트 기반 설계로 변환하기 위해서는 우선 컴포넌트를 식별하여 생성하고, 인터페이스 식별, 컴포넌트 내부 클래스 설계 등의 절차를 거쳐야 된다. 객체지향 설계를 컴포넌트 기반 설계로 변환하기 위해서는 컴포넌트를 추출하는 작업이 추가된다. 객체들간의 결합을 유도하여 재사용을 강조하는 객체지향 모델은 컴포넌트들 간의 독립성을 강조하는 컴포넌트 기반의 모델과 사상이 달라 적절한 컴포넌트 추출을 어렵게 만드는 문제가 발생한다. 독립적인 컴포넌트를 추출하기 위해 객체들 간의 상속을 포함한 관계들을 조정할 필요가 있다.

본 논문에서는 컴포넌트의 식별이나 인터페이스 추출 등 구체적인 컴포넌트 기반 설계방법론을 제시하는 것은 제외한다. 다만, 변환기법의 설명을 위해 다음과 같은 규칙을 정의한다.

우선 그림 8과 같이 객체지향 설계로부터 추출된 객체가 컴포넌트로 식별되었다고 하자.

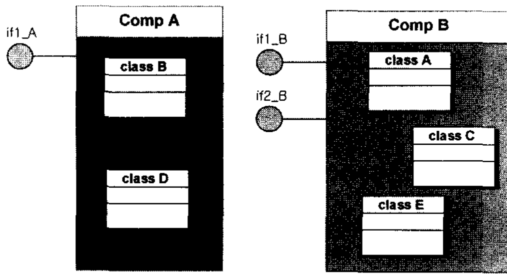


그림 8 컴포넌트 식별

객체의 집합을 *SetofObjects*라 하면 다음과 같이 정의된다.

SetofObjects = {*classA*, *classB*, *classC*, *classD*, *classE*}

객체의 집합에서 컴포넌트(*Comp*)를 식별하는 함수를 *Group*이라 하면 다음과 같다.

Group(*Comp*) = { {*classB*, *classD*}, {*classA*, *classC*, *classE*} }

식별된 컴포넌트에서 인터페이스를 식별하는 함수를 *Interface*라 하면 다음과 같이 정의된다.

Interface(*Comp*) = { {*if1_A*}, {*if1_B*, *if2_B*} }

위와 같은 규칙을 이용하여 객체지향 설계의 정형명세를 컴포넌트 기반 정형명세로 변환하는 기법을 제시한다. 우선 각 정형명세의 요소를 추출하여 변환 규칙을 정의한 후, 변환 함수를 통하여 변환되는 것을 보여준다. 이를 그림으로 표현하면 그림 9와 같다.

그림 9는 객체지향 설계의 정형명세(Object-Oriented Formal Specifications, OOFS) 스키마의 구성요소와 컴포넌트 기반 설계의 정형명세(Component-Based Formal Specifications, CBFS) 스키마의 구성요소 간의 관계를 나타낸 것이다. CBFS 스키마는 타입을 정의하는 *Type definition*과 *Interface* 스키마, 그리고 *Component* 스키마로 구성되어 있다. OOFS 스키마는 *Type definition*과 *Class* 스키마, 기능적 모델을 나타내는 *UseCaseDiagram* 스키마, 그리고 동적 모델을 나타내는 *SequenceDiagram* 스키마로 구성된다.

CBFS의 *Type definition*에서는 OOFS에서 정의된 타입(*Type definition*)을 모두 그대로 명세한다. CBFS의 *Provide Interface*는 OOFS의 *Class* 스키마 구성요소인 *visibility list*에 정의된 오퍼레이션과 *UseCaseDiagram* 스키마에서 정의된 오퍼레이션을 이용하여 명세된다. *Component* 스키마의 *local definition*은 OOFS의 *Class* 스키마 구성요소인 *local definition*에서 추출되고, *component state schema*와 *initial state schema*는 *Class* 스키마의 *state schema*와 *initial state schema*에서 각각 추출된다. CBFS의 *class*는 OOFS의 *Class* 스키마를 이용하여 명세되고, 컴포넌트 내부 워크플로우를 나타내는 *set of micro workflow*는 *SequenceDiagram* 스키마를 이용하여 명세된다. 여기에서 객체지향 설계에서 지원하지 않는 부분인 Required 인터페이스나 가변성, 컴포넌트 간 워크플로우인 매크로 워크플로우 등에 관한 변환기법은 제외한다. 그러나, 이 구성요소들에 관한 명세는 3장의 컴포넌트 정형명세 언어에서 제시하고 있으므로 이 표기법에 따라 명세한다.

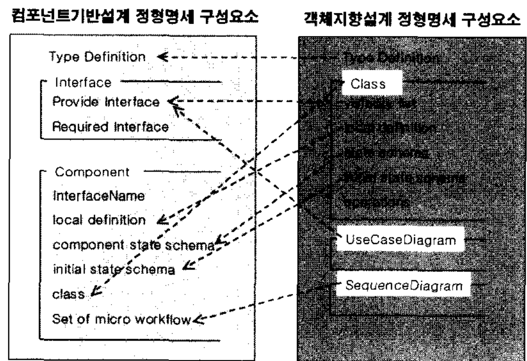


그림 9 정형명세 구성요소간 매핑

5.1 타입 정의(Type Definition) 변환

객체지향 설계에서 정의한 타입은 그대로 컴포넌트 기반 설계의 타입 정의부분으로 변환되므로 그림 10과 같이 표현된다.

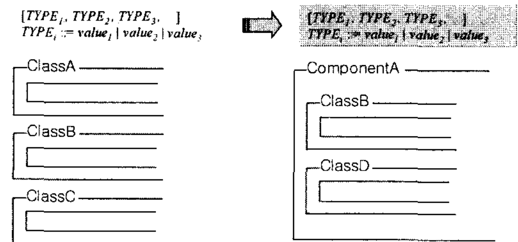


그림 10 타입 정의(Type Definition) 변환

[규칙1] 객체지향 정형명세에서 정의된 타입은 모두 컴포넌트 기반 정형명세의 타입 정의로 각각 매핑된다.

객체지향설계 정형명세의 타입은 기본 타입(Basic Type)과 사용자가 지정한 자유 타입(Free Type)이 있고, 상수값 정의 등으로 구성된다. 타입 정의에서 정의가능한 모든 타입의 집합을 *OTYPE*이라 하자.

$OTYPE = \{ TYPE_1, TYPE_2, TYPE_3, ..., TYPE_n \}$

컴포넌트 기반 설계에서 정의가능한 모든 타입을 *CTYPE*이라 한다면, 타입 변환함수 f는 *OTYPE*에서 *CTYPE*으로 매핑하는 함수이다. *OTYPE*에 속하는 모

든 원소 o에 대해 *CTYPE*에 속하는 원소 c가 하나 이상 존재하여 모든 객체 타입의 집합은 컴포넌트 타입의 집합에 속한다. 그리고, *OTYPE* 집합의 모든 원소 o1, o2에 대해 변환함수 *f*로 매핑하면 f(o1) = c1, f(o2) = c2로 각각 매핑된다.

$f : OTYPE \rightarrow CTYPE$

$\forall o \in OTYPE \bullet \exists c \in CTYPE \Rightarrow f(o) \subset f(c)$

let $\forall o1, o2 : \mathrm{dom}\ OTYPE \bullet f(o1) = c1 \in CTYPE \wedge f(o2) = c2 \in CTYPE$

5.2 인터페이스 명세 변환

컴포넌트 기반 설계의 정형명세에서 인터페이스는 세 가지로 나뉜다. Provide 인터페이스와 Required 인터페이스, Uses 인터페이스가 있다. 이 중 Provide 인터페이스는 객체지향 설계의 정형명세에서 클래스의 퍼블릭 오퍼레이션과 유즈케이스 다이어그램의 유즈케이스를 이용하여 추출한다. Required 인터페이스와 Uses 인터페이스는 객체지향 설계에서는 지원되지 않는 개념이므로 변환대상에서 제외하고 Provide 인터페이스의 변환에 대해서만 언급하기로 한다.

객체지향 설계에서 각 클래스 정형명세 중 Visibility List에 정의된 오퍼레이션은 컴포넌트 기반 설계의 인터페이스 오퍼레이션의 대상이 된다. 그리고, 유즈케이스 다이어그램에서 액터와 관계를 갖는 유즈케이스 오퍼레이션도 컴포넌트 기반 설계의 인터페이스 대상이다. 그러므로, 객체지향 설계의 Visibility List에 나타난 오퍼레이션 중에서 유즈케이스 다이어그램에서 액터와 상호작용하는 유즈케이스 오퍼레이션은 컴포넌트기반 설계의 인터페이스 대상이 된다. 이 유즈케이스 오퍼레이션이 모두 인터페이스로 추출되지 않을 수도 있으나, 이는 개발 시스템의 정책에 해당되므로 본 논문에서는 모두 인터페이스 오퍼레이션으로 추출된다고 가정한다.

[규칙2] 클래스 정형명세의 Visibility List에 정의된 오퍼레이션 중에서 유즈케이스 다이어그램에서 액터와 상호작용하는 유즈케이스와 중복되는 오퍼레이션은 컴포넌트 기반 설계의 Provide 인터페이스 오퍼레이션으로 변환된다.

그림 11처럼 객체지향 정형명세에서 Visibility List 중 액터와 관계를 갖은 유즈케이스와 동일한 오퍼레이션명들은 컴포넌트 기반 설계의 Provide 인터페이스 오퍼레이션으로 매핑되고, Op_1, Op_2, Op_3와 같은 각 오퍼레이션 스키마는 컴포넌트 명세 안에 해당 클래스의 오퍼레이션 스키마로 매핑된다. 여기에서 인터페이스에 나타나는 오퍼레이션 스키마는 컴포넌트에 있는 오퍼레이션 스키마와 대응하는 것으로 동일한 스키마명이어야 한다. 인터페이스 안의 오퍼레이션 스키마는 오퍼레이션의 의미를 표현하기 위해 Predicate을 포함해야 하나, 인터페이스는 단지 컴포넌트의 기능성을 보여주는 것이기 때문에 실제 State 스키마를 가질 수 없다. 그러므로 인터페이스의 오퍼레이션 스키마는 컴포넌트 안의 오퍼레이션 스키마와 스키마명은 동일하여야 하나, 스키마 내부의 State와 Predicate은 컴포넌트의 기능 중 API와 관련된 명세만을 표현하므로 동일하지 않을 수 있다. 그러나, 이 부분은 시스템의 개발 정책(Contract)과 관련된 부분이므로 여기에서는 언급하지 않는다.

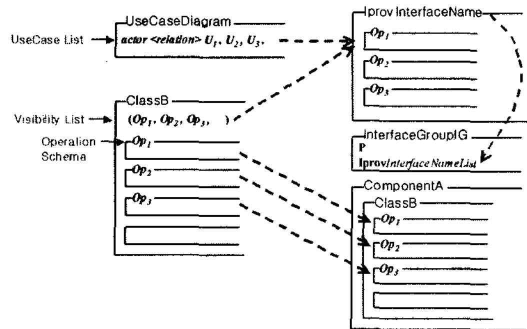


그림 11 Provide 인터페이스 변환

그림 8에서 정의한 컴포넌트 CompA에 속하는 클래스와 인터페이스는 다음과 같다.

SetofObjects = {classA, classB, classC, classD, classE}

CompA = { classB, classD }

클래스B와 클래스D에서 정의된 Visibility List 중 액터와 관계된 유즈케이스가 다음과 같다고 하자.

$Visibility(classB, classD) = \{ classB.Op_1, classB.Op_2, classB.Op_3, classD.Op_4, classD.Op_5 \}$

그렇다면, 컴포넌트 CompA의 인터페이스는 클래스B와 클래스D의 Visibility를 인터페이스 후보들의 집합 I로 추출한다. 후보 인터페이스 I 중에서 유즈케이스 다이어그램에서 액터와 상호작용하는 유즈케이스의 집합 U 안에 존재하는 오퍼레이션과 중복되는 오퍼레이션이 Provide 인터페이스로 추출되므로 이를 수식으로 나타내면 다음과 같다.

$transV : VISIBILITY(ClassB, ClassD) \rightarrow I$

$\forall op \in VISIBILITY(ClassB,\ ClassD) \bullet \exists i \in I \Rightarrow transV(op) = i$

$\exists op1 \in U \bullet op1 \in VISIBILITY(ClassB, ClassD) \Rightarrow op1 \in I$

5.3 컴포넌트 명세 변환

컴포넌트 스키마를 구성하는 요소는 Local Definition, Component State Schema, Initial State Schema, Class 등이다. 객체지향 설계의 정형명세에서 컴포넌트를 구성하는 구성요소별로 변환 규칙 및 변환함수를 제

시한다.

5.3.1 Local Definition 변환

객체지향 설계의 정형명세 중 클래스의 Local Definition에서 정의한 타입은 그림 12처럼 컴포넌트 스키마 안에 있는 클래스의 Local Definition으로 그대로 매핑된다.

[규칙3] 객체지향 정형명세에서 클래스 안에 정의된 Local Definition은 모두 컴포넌트 스키마 명세 안에 클래스의 Local Definition으로 그대로 변환된다.

객체지향 정형명세의 클래스 Local Definition에서 정의가능한 모든 타입의 집합을 *LTYPE*이라 하자.

$LTYPE = \{ LTYPE_1, LTYPE_2, LTYPE_3, ..., LTYPE_n \}$

컴포넌트 안에 존재하는 클래스의 Local Definition에서 정의가능한 타입을 *LcTYPE*이라 한다면, 타입 변환함수 g 는 *LTYPE*에서 *LcTYPE*으로 매핑하는 함수이다. *LTYPE*에 속하는 모든 원소 i에 대해 *LcTYPE*에 속하는 원소 c가 하나 이상 존재하여 객체지향 클래스의 타입의 집합은 컴포넌트 내의 클래스 타입의 집합에 속한다. 그리고, *LTYPE* 집합의 모든 원소 i1, i2에 대해 변환함수 g 로 매핑하면 g(i1) = c1, g(i2) = c2로 각각 매핑된다.

$g : LTYPE \rightarrow LcTYPE$

$\forall i \in LTYPE \bullet \exists c \in LcTYPE \Rightarrow g(i) \subset g(c)$

let $\forall i1, i2 : \text{dom } LTYPE \bullet g(i1) = c1 \in LcTYPE \wedge g(i2) = c2 \in LcTYPE$

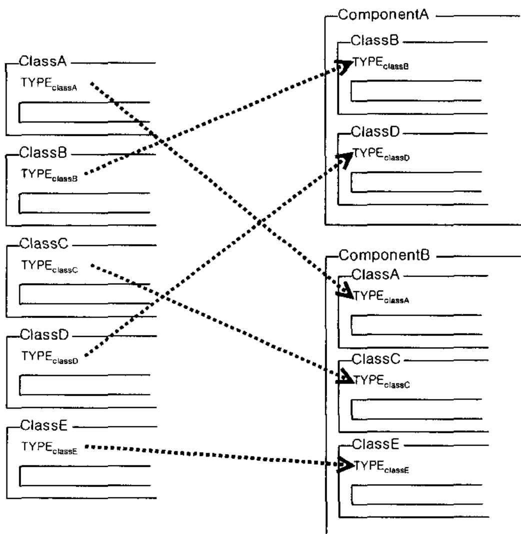


그림 12 Local Definition 명세 변환

5.3.2 Component State Schema 변환

Component State Schema는 객체지향 설계에서 명세한 State Schema와 컴포넌트에서의 가변점(Variation Point)을 명세하는 스키마이다. 여기에서 가변점은 객체지향 설계에서 없는 부분이므로 State Schema의 변환에 대한 것만 언급한다.

State Schema는 스키마명이 없는 박스로 표현되거나, 수평형태로 [ ] 안에 표현된다. 객체지향설계 정형명세 중 클래스의 State Schema는 컴포넌트 기반 설계 명세에서 컴포넌트 안의 Component State Schema로 그대로 매핑된다.

[규칙4] 객체지향 정형명세에서 클래스 안에 정의된 State Schema는 모두 컴포넌트 명세 안의 Component State Schema로 그대로 변환된다.

객체지향설계 클래스 정형명세 중 State Schema에서 정의가능한 모든 State의 집합을 *STATE* 이라 하자.

$STATE = \{ STATE_1, STATE_2, STATE_3, ..., STATE_n \}$

컴포넌트 명세에서 정의가능한 모든 State의 집합을 *CSTATE*라 한다면, 타입 변환함수 h는 *STATE*에서 *CSTATE*로 매핑하는 함수이다. *STATE*에 속하는 모든 원소 s 에 대해 *CSTATE*에 속하는 원소 c 가 하나 이상 존재하여 모든 클래스 State의 집합은 컴포넌트 State의 집합에 속한다. 그리고, *STATE* 집합의 모든 원소 s1, s2에 대해 변환함수 h로 매핑하면 h(s1) = c1, h(s2) = c2로 각각 매핑된다.

$h : STATE \rightarrow CSTATE$

$\forall s \in STATE \bullet \exists c \in CSTATE \Rightarrow h(s) \subset h(c)$

let $\forall s1, s2 : \text{dom } STATE \bullet h(s1) = c1 \in CSTATE \wedge h(s2) = c2 \in CSTATE$

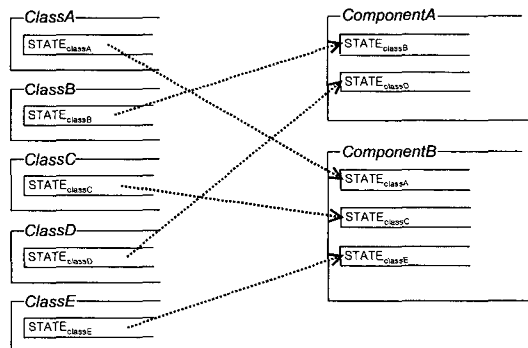


그림 13 State Schema 명세 변환

5.3.3 Initial State Schema 변환

Initial State Schema는 객체지향 설계에서 명세한 State Schema의 초기값을 설정하는 스키마로 스키마명이 "INIT"로 표현된다. 객체지향 정형명세 중 클래스의 Initial State Schema는 컴포넌트 기반 설계 명세에서

컴포넌트 안의 Initial State Schema로 그대로 매핑된다.

[규칙5] 객체지향 정형명세에서 클래스 안에 정의된 Initial State Schema는 모두 컴포넌트 명세 안의 Initial State Schema로 그대로 변환된다.

객체지향 클래스 정형명세 중 Initial State Schema에서 정의가능한 모든 초기값의 집합을 *INIT*이라 하자.

$INIT = \{ INIT_1, INIT_2, INIT_3, ..., INIT_n \}$

컴포넌트 명세에서 정의가능한 모든 Initial State의 집합을 *CINIT*라 한다면, 초기값 변환함수 *j*는 *INIT*에서 *CINIT*로 매핑하는 함수이다. *INIT*에 속하는 모든 원소 i에 대해 *CINIT*에 속하는 원소 c가 하나 이상 존재하여 모든 클래스 Initial State의 집합은 컴포넌트의 Initial State의 집합에 속한다. 그리고, *INIT* 집합의 모든 원소 i1, i2에 대해 변환함수 *j*로 매핑하면 j(i1) = c1, j(i2) = c2로 각각 매핑된다.

$j : INIT \rightarrow CINIT$

$\forall i \in INIT \bullet \exists c \in CINIT \Rightarrow j(i) \subset j(c)$

let $\forall i1, i2 : \text{dom } INIT \bullet j(i1) = c1 \in CINIT \wedge j(i2) = c2 \in CINIT$

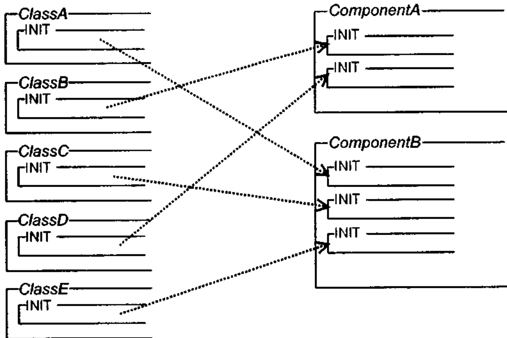


그림 14 Initial State Schema 명세 변환

5.3.4 Class 명세 변환

객체지향 설계에서 명세한 클래스는 컴포넌트 기반 설계 명세에서 컴포넌트 안의 클래스로 그대로 매핑된다.

[규칙6] 객체지향 정형명세의 클래스 스키마는 모두 컴포넌트 명세 안의 클래스 스키마로 그대로 변환된다.

객체지향 클래스 명세에서 컨트롤러의 역할을 하는 컨트롤 클래스는 그 기능이 컴포넌트 기반 명세에서 인터페이스와 컴포넌트 명세로 나뉜다. 인터페이스의 기능을 표현한 API 부분은 변환대상에서 제외해야 하므로, 컨트롤 클래스 스키마명과 스키마 내부에서 Visibility List 부분은 제외하고 나머지 명세 부분만 컴포넌트 명세 안으로 변환되어야 한다.

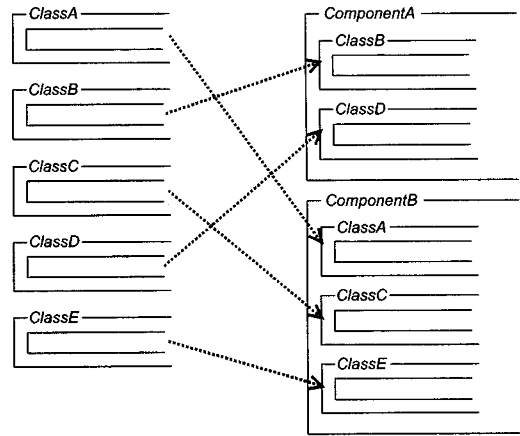


그림 15 Class 명세 변환

객체지향 클래스 정형명세에서 정의 가능한 모든 클래스의 집합을 *CLASS*라 하자.

$CLASS = \{ CLASS_1, CLASS_2, CLASS_3, ..., CLASS_n \}$

컴포넌트 안에 정의가능한 모든 클래스의 집합을 *CCLASS*라 한다면, 변환함수 *k*는 *CLASS*에서 *CCLASS*로 매핑하는 함수이다. *CLASS*에 속하는 모든 원소 c에 대해 *CCLASS*에 속하는 원소 cc가 하나 이상 존재하여 모든 객체지향 클래스의 집합은 컴포넌트 안에 존재하는 클래스의 집합에 속한다. 이때 클래스 cc는 컴포넌트 명세에 따라 컴포넌트 CompA에 속하거나 컴포넌트 CompB에 속한다. 이를 수식으로 나타내면 다음과 같다.

$k : CLASS \rightarrow CCLASS$

$\forall c \in CLASS \bullet \exists cc \in CCLASS \Rightarrow k(c) = cc$

$\exists cc: CCLASS \Rightarrow cc \in CompA \vee cc \in CompB$

5.4 워크플로우 명세 변환

컴포넌트 기반 설계에서 워크플로우는 한 컴포넌트 내에서 객체간의 상호작용을 나타내는 마이크로 워크플로우와 컴포넌트 간의 상호작용을 나타내는 매크로 워크플로우로 구분된다. 여기에서는 객체지향 명세를 변환하는 것이므로 마이크로 워크플로우의 변환에 대해서만 언급하기로 한다.

마이크로 워크플로우는 인터페이스와 컴포넌트 내부 객체의 오퍼레이션과의 매핑이다. 인터페이스는 객체지향 설계의 Visibility List에 나타난 오퍼레이션 중에서 유즈케이스 다이어그램에서 액터와 상호작용하는 유즈케이스와 중복되는 오퍼레이션이 대상이 되므로, 유즈케이스 다이어그램과 시퀀스 다이어그램의 명세를 이용하여 마이크로 워크플로우를 추출한다.

[규칙7] 객체지향 설계에서 유즈케이스 다이어그램 정형명세 중 액터와 상호작용하는 유즈케이스에 해당하

는 시퀀스 다이어그램 정형명세는 컴포넌트 기반 설계의 마이크로 워크플로우로 변환된다.

그림 16처럼 객체지향 정형명세의 유즈케이스 다이어그램에서 액터와 관계를 갖는 유즈케이스에 대한 각각의 시퀀스 다이어그램 정형명세가 존재한다. 이 정형명세의 메시지 순서는 컴포넌트 기반 설계의 컴포넌트 명세 안에 마이크로 워크플로우 명세로 매핑된다. 여기에서 컴포넌트 명세에 나타나는 메시지 오퍼레이션 스키마는 인터페이스에 있는 오퍼레이션 스키마와 연결되어야 하므로 유즈케이스 다이어그램에서 추출된 인터페이스의 오퍼레이션 스키마를 먼저 명세한 후 메시지 시퀀스를 명세한다.

유즈케이스 다이어그램에서 액터와 관계를 갖는 유즈케이스의 집합을 *UList*라 하자.

$UList = \{ U_1, U_2, U_3 \}$

*UList*에 나타난 각 유즈케이스의 시퀀스 명세는 다음과 같이 표현된다.

$Sequences(U_1) \triangleq object1.message1;\ \ object2.message_2\ ;\ object3.message3$

$Sequences(U_2) \triangleq object4.message4; object5.message5$

$Sequences(U_3) \triangleq object6.message6; object7.message7; object8.message8$

그렇다면, 컴포넌트 CompA의 마이크로 워크플로우 W는 인터페이스 I의 오퍼레이션과 오퍼레이션의 시퀀스 명세의 Composition으로 구성되므로 수식으로 나타내면 다음과 같다.

$W(CompA) = \{Workflow.op1,\ Workflow.op2,\ Workflow.op3\}$

$\forall op \in W(CompA) \bullet \exists i : I \wedge \forall Ui : U,\ \ i : N \Rightarrow W(CompA) = i.op;\ \Sigma Sequences(Ui)$

$\exists op1 \in U \bullet op1 \in VISIBILITY(ClassB, ClassD) \Rightarrow op1 \in I$

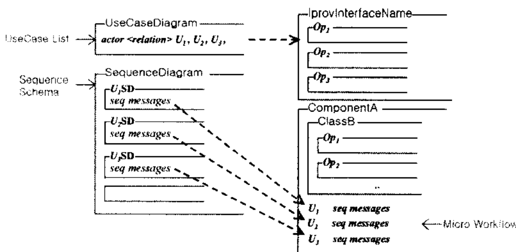


그림 16 워크플로우 명세 변환

6. 변환 툴(Transformation Tool)

5장에서 제시한 변환기법을 적용하여 변환하는 과정을 자동화 할 수 있도록 변환 툴을 개발하여 제공한다. 변환 툴은 객체지향 정형명세를 컴포넌트 기반 정형명세로 자동으로 변환되도록 지원해 주는 도구로서 수작업으로 변환 시 발생할 수 있는 에러를 방지해 준다. 그러나, 변환 과정 중 컴포넌트 식별이나 인터페이스의 추출 등 개발 시스템의 정책이나 개발자에 의해 수작업으로 조정되는 부분은 자동으로 변환될 수 없기 때문에, 이 변환 툴은 자동 변환이 가능한 부분에 대한 것만 지원한다.

변환 툴의 구조는 그림 17과 같다. 객체지향 정형명세를 시스템과 타입, 클래스, 유즈케이스, 시퀀스 등으로 구분하여 정의하면 5장에서 제시한 변환 기법에 따라 컴포넌트 기반 정형명세가 생성된다.

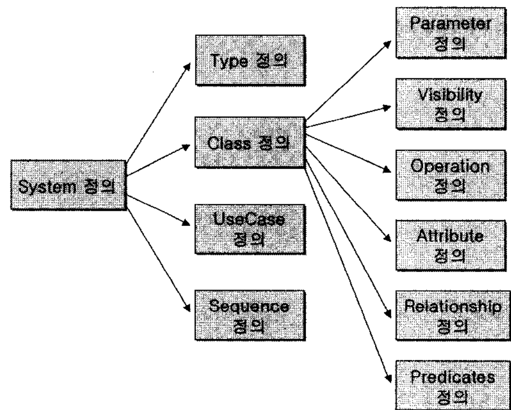


그림 17 변환 툴의 구조

타입 정의 화면에서는 시스템명을 입력하고, 타입의 종류와 변수를 정의한다. 상수가 있을 경우 상수도 정의한다. Operation 정의에서는 시스템명을 선택하고 Operation명을 지정한다. 그리고 Visibility와 파라미터, Predicates를 정의한다. 클래스 정의 화면에서는 시스템명과 클래스명을 정의한다. 그리고, 파라미터와 Visibility를 정의하고 상속 클래스가 있을 경우 상속 클래스 스키마를 정의한다. 지역 타입과 상수를 정의하고 속성 정의, 오퍼레이션 정의, 그리고 Predicates를 정의한다.

오퍼레이션 정의가 끝나면 유즈케이스와 시퀀스를 정의한다. 유즈케이스는 액터와 유즈케이스, 그리고 관계를 정의한다. 시퀀스는 객체와 메시지, 그리고 객체와 메시지의 상호작용에 대해 정의한다. 이와 같은 방법으로 정의하여 컴포넌트 기반 정형명세를 자동으로 생성한다.

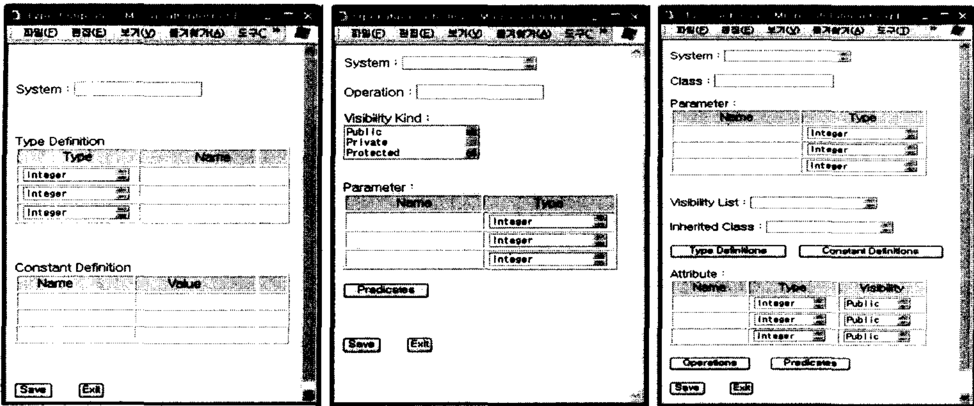


그림 18 타입과 클래스, 오퍼레이션 정의 화면

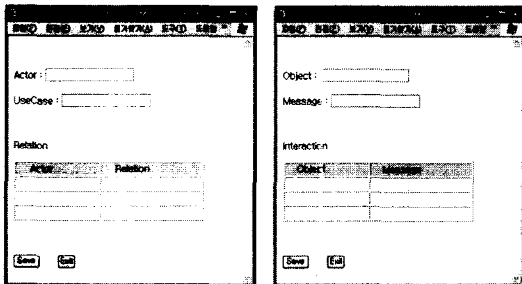


그림 19 유즈케이스와 시퀀스 정의 화면

7. 사례 연구

4장에서 정의한 객체지향 설계의 정형명세를 기준으로 5장에서 제시한 변환 기법을 적용하여 변환하는 과정을 사례 연구를 통하여 확인함으로써 변환기법의 적절성을 검증한다.

7.1 객체지향 설계의 정형명세

객체지향 설계의 정형명세를 위해 간단한 뱅킹 시스템을 구현하여 증명한다. 뱅킹 시스템은 고객관리와 계좌관리의 두 가지 기능이 있다. 고객이 우선 시스템을

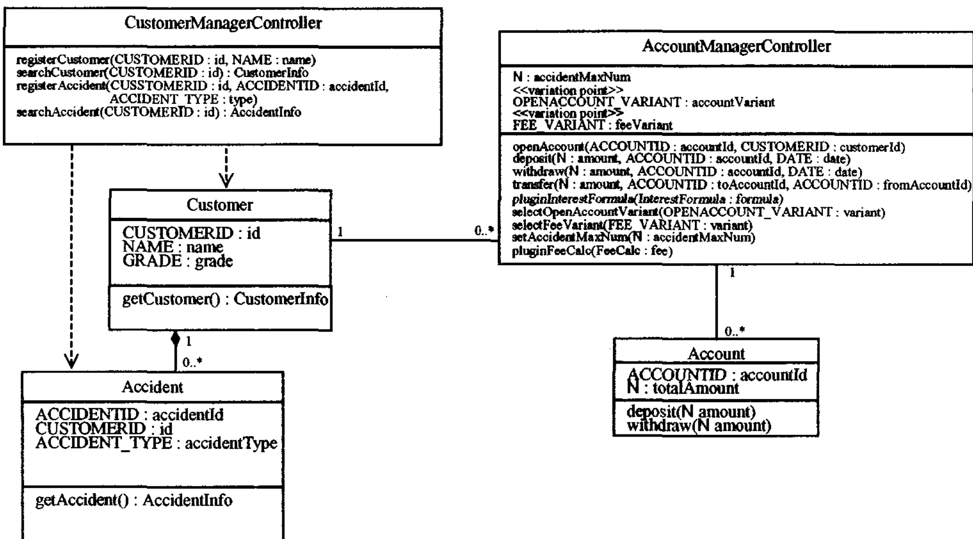


그림 20 뱅킹 시스템의 객체지향 설계구조도

사용하기 위해서는 시스템에 고객정보를 등록해야한다. 만약 고객이 연체와 같은 문제가 있을 경우 시스템 관리자는 시스템에 사건을 기록한다. 또한, 고객과 사건의 조회는 고객관리에 필요한 기능이다.

시스템의 명세는 고객관리의 계약을 포함한다. 은행에서 고객과 텔러는 시스템에 이런 작업을 위해서 계약을 하고, 이는 'CustomerManager'라는 컨트롤 클래스에 묘사된다. 뱅킹 시스템은 고객을 위해 예금, 인출, 이체 서비스를 제공한다. 이런 서비스를 계좌관리라고 한다. 계좌관리의 계약은 'AccountManager'라는 컨트롤 클래스로 구현된다. 그림 20은 뱅킹 시스템의 설계구조도를 보여준다. 이를 이용하여 고객관리 부분만 정형명세한다.

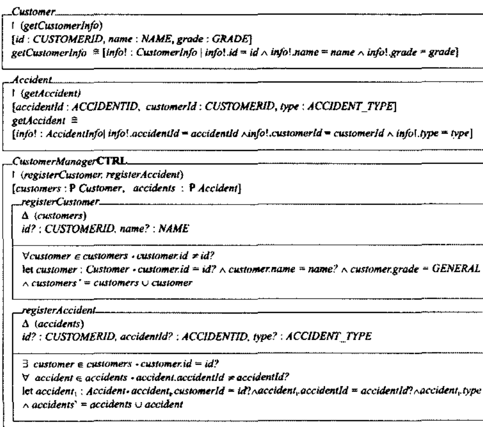
7.1.1 Type Definition

뱅킹 시스템에서 고객관리를 명세하기 위한 기본 타입을 정의한 것이다. *GRADE*는 고객의 신용등급을 나타내는 것으로 최우수고객(VIP)과, 우수고객(SPECIAL), 일반고객(GENERAL), 신용불량고객(BAD)으로 나뉜다. *CustomerInfo*는 고객정보를 나타내는 타입으로 고객번호와 성명, 신용등급의 정보를 갖는다.

[ACCOUNTID, CUSTOMERID, NAME, ACCIDENTID]
GRADE ::= VIP | SPECIAL | GENERAL | BAD
ACCIDENT_TYPE ::= LOSS | BAD_CREDIT | BLACKLIST
CustomerInfo ≙ [id : CUSTOMERID ; name : NAME ; grade : GRADE]
AccidentInfo ≙ [accidentId : ACCIDENTID ; customerId : CUSTOMERID : accidentType : ACCIDENT_TYPE]

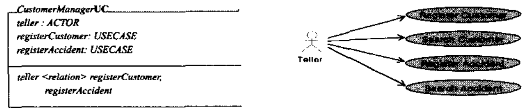
7.1.2 Class

뱅킹 시스템에서 고객관리 클래스는 '*CustomerManagerCTRL*', '*Customer*', '*Accident*'로 구성되어 있다. '*CustomerManager*CTRL'는 고객관리 컨트롤 클래스로 고객정보를 등록하는 오퍼레이션인 '*registerCustomer*' 스키마를 갖고, 고객의 사건기록을 등록하는 오퍼레이션인 '*registerAccident*' 스키마 가진다.



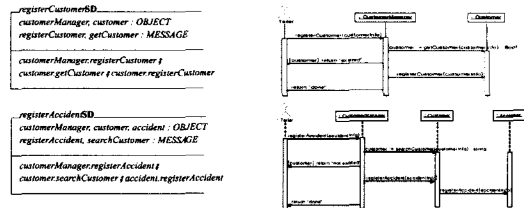
7.1.3 Use Case Diagram

뱅킹 시스템에서 고객관리 유즈케이스 다이어그램을 명세한 것이다. 텔러(*Teller*)는 유즈케이스 오퍼레이션 *registerCustomer, registerAccident* 와 관계를 갖는다.



7.1.4 Sequence Diagram

뱅킹 시스템에서 각 유즈케이스에 대한 시퀀스 다이어그램을 명세한 것이다. *registerCustomer*SD 스키마는 *registerCustomer*라는 유즈케이스에 대한 시퀀스 다이어그램 스키마로, 메시지가 전달되는 순서대로 스키마를 명세한다. 명세시 스키마명은 *customerManager. registerCustomer* 처럼 스키마명 앞에 수행 주체인 객체명을 명시하여 <객체명>.<스키마명> 형태로 표시한다. 그리고, 메시지의 순서는 스키마명 사이에 ⨟ 연산자를 사용하여 표시한다.



7.2 컴포넌트 정형명세로의 변환

이 사례 연구는 그림 21의 뱅킹 시스템에 대한 컴포넌트 기반 설계 구조도를 기준으로 6.1절의 객체지향 정형명세를 5장에서 제시한 변환기법을 적용하여 컴포넌트 정형명세로 변환한다.

7.2.1 타입 정의 변환

객체지향 정형명세에서 정의된 타입정의 부분은 그대로 컴포넌트 기반 정형명세의 타입정의 부분으로 변환되었다. 사용자 타입 정의인 *ACOUNTID*나 *CUSTOMERID*도 그대로 옮겨지고, 자유 타입 정의 부분도 그대로 옮겨진다. 컴포넌트 설계에서 타입 정의 부분에는 객체지향에 없는 가변성에 관한 정의를 추가할 수 있다.

[ACCOUNTID, CUSTOMERID, NAME, ACCIDENTID] *GRADE ::= VIP | SPECIAL | GENERAL | BAD* *ACCIDENT_TYPE ::= LOSS | BAD_CREDIT | BLACKLIST* *CustomerInfo ≙ [id : CUSTOMERID ; name : NAME ; grade : GRADE]* *AccidentInfo ≙ [accidentId : ACCIDENTID ; customerId : CUSTOMERID : accidentType : ACCIDENT_TYPE]*	⇒	*[ACCOUNTID, CUSTOMERID, NAME, ACCIDENTID]* *GRADE ::= VIP | SPECIAL | GENERAL | BAD* *ACCIDENT_TYPE ::= LOSS | BAD_CREDIT | BLACKLIST* *CustomerInfo ≙ [id : CUSTOMERID ; name : NAME ; grade : GRADE]* *AccidentInfo ≙ [accidentId : ACCIDENTID ; customerId : CUSTOMERID : accidentType : ACCIDENT_TYPE]*

7.2.2 인터페이스 명세 변환

클래스 정형명세의 Visibility List에 정의된 오퍼레이션 중에서 유즈케이스 다이어그램에서 액터와 상호작용하는 유즈케이스와 중복된 오퍼레이션은 컴포넌트 기반 설계의 Provide 인터페이스 오퍼레이션으로 변환되었다. *CustomerManager*CTRL 스키마안의 Visibility List에 정의된 오퍼레이션 중에서 *CustomerManager*UC 스키마에서 액터 *Teller*와 상호작용하는 오퍼레이션은 *registerCustomer, registerAccident* 이다. 이 오퍼레이션들은 Provide 인터페이스로 추출되어 *Customer*-

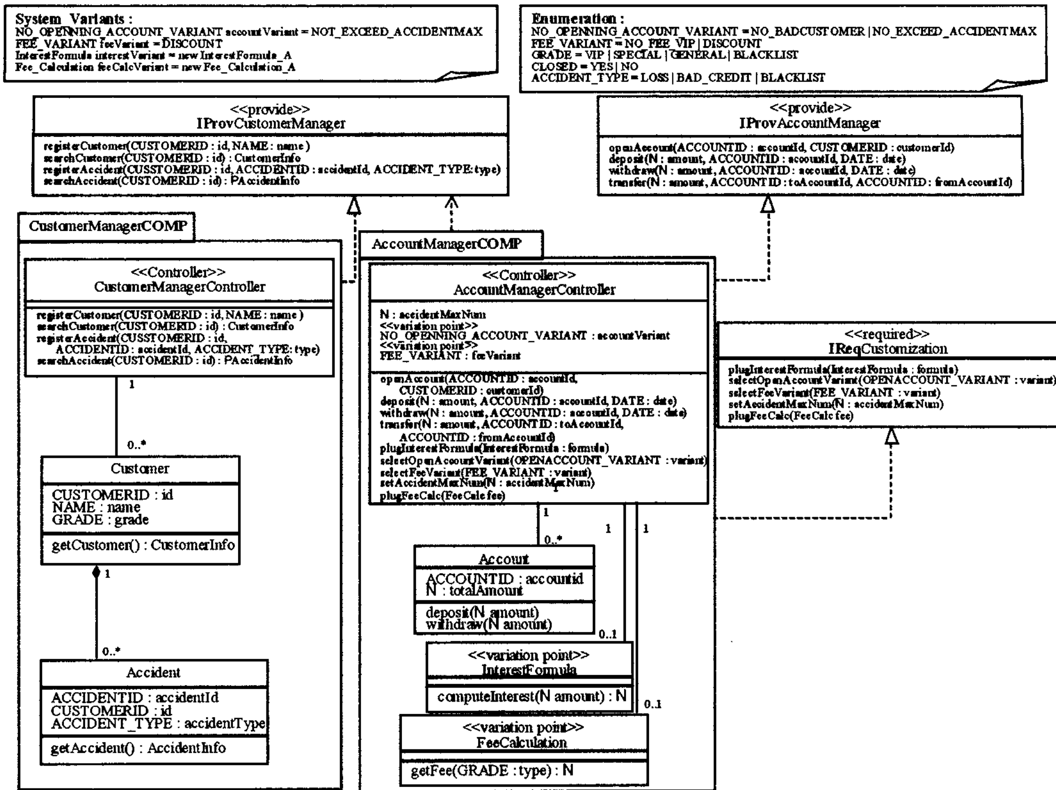
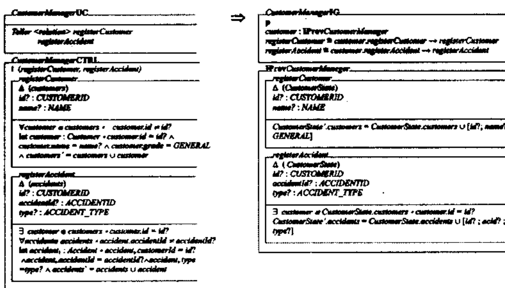


그림 21 뱅킹 시스템의 컴포넌트 기반 설계구조도

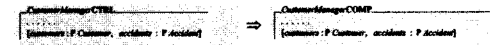
*Manager*IG 인터페이스 그룹에 Provide 인터페이스로 명시되고, **Iprov***CustomerManager* 인터페이스의 각 스키마 명세는 *CustomerManager***CTRL** 스키마에 명세된 것으로 변환된다.



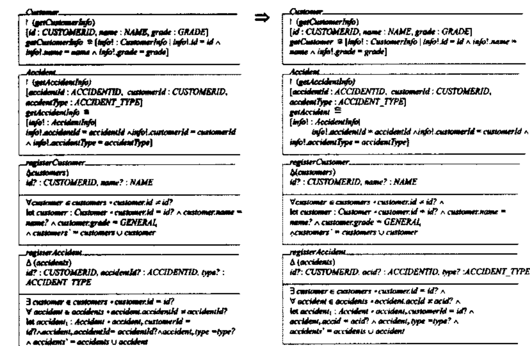
7.2.3 컴포넌트 명세 변환

컴포넌트 스키마를 구성하는 요소는 Local Definition, Component State Schema, Initial State Schema, Class 등이다. 우선 객체지향의 정형명세 중 Local Definition과 Initial State Schema 부분은 정의된 것이 없으므로 이것의 변환은 생략한다.

객체지향 정형명세 중 클래스 안에 정의된 State Schema는 모두 컴포넌트 명세 안의 Component State Schema로 그대로 변환되었다.



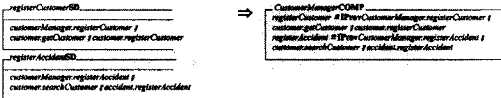
객체지향 정형명세 중 클래스 스키마인 *Customer*, *Accident*, *registerCustomer*, *registerAccident*는 모두 컴포넌트 *CustomerManager***COMP** 안의 클래스 스키마로 그대로 변환되었다.



7.2.4 워크플로우 변환

객체지향설계에서 액터와 상호작용하는 유즈케이스에 해당하는 시퀀스 다이어그램 정형명세는 컴포넌트 기반

설계의 마이크로 워크플로우로 변환되었다. *registerCustomer*SD 스키마는 컴포넌트 스키마 안으로 옮겨지면서 *customerManager* 객체는 **IProv***CustomerManager*로 바뀌어 변환되므로, *customerManager.registerCustomer*는 **IProv***CustomerManager.registerCustomer*로 표현된다.



7.3 변환 검증

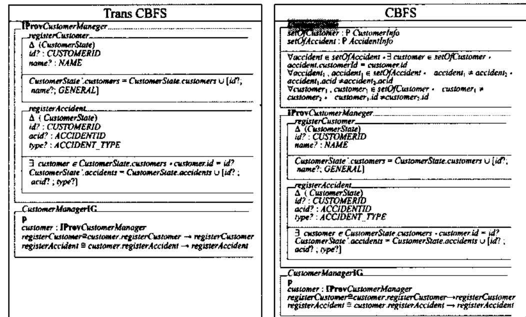
7.2절의 사례연구에서 객체지향 정형명세를 5장의 변환기법을 적용하여 변환한 컴포넌트 정형명세(Transformed Component-Based Formal Specifications, Trans CBFS)가 제대로 되었는지에 대한 검증 방법으로 컴포넌트 설계 모델을 이용하여 컴포넌트 정형명세(Component-Based Formal Specifications, CBFS)를 수행한 후, 7.2절에서 변환된 명세와 비교하여 변환기법의 정확성을 검증하고 문제점을 찾아본다. 컴포넌트 설계의 정형명세를 위해 7.2절의 그림 21과 같은 도메인에 해당하는 컴포넌트 설계를 이용하여 정형명세 한다. 그림 21은 뱅킹 시스템의 컴포넌트와 인터페이스, 클래스, 가변성 설정을 보여준다. 이를 이용하여 고객관리 부분만 정형명세하여 변환된 명세와 비교하여 확인한다.

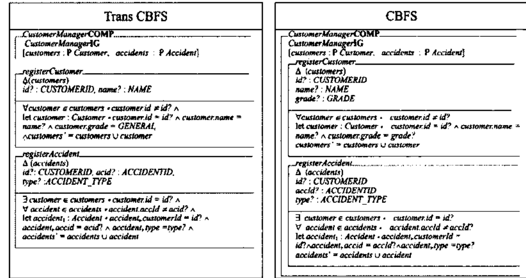
뱅킹 시스템을 명세하기 위한 기본 타입 정의에 대한 명세 부분은 7.2.1절의 변환된 명세와 컴포넌트 기반 설계에서 정의한 명세가 일치하며 그 명세는 다음과 같다.

```
[ACCOUNTID, CUSTOMERID, NAME, ACCIDENTID]
GRADE ::= VIP | SPECIAL | GENERAL | BAD
ACCIDENT_TYPE ::= LOSS | BAD_CREDIT | BLACKLIST
CustomerInfo ≙ [id : CUSTOMERID ; name : NAME ; grade : GRADE]
AccidentInfo ≙ [accidentId : ACCIDENTID ; customerId : CUSTOMERID : accidentType : ACCIDENT_TYPE]
```

인터페이스 명세 부분은 7.2.2절의 변환된 명세(Trans CBFS)와 컴포넌트 기반 설계에서 정의한 인터페이스 명세(CBFS)가 *CustomerState* 부분을 제외하고는 동일하다. 이것은 인터페이스의 기능을 정의하는 state 스키마로 IProv*CustomerManager* 인터페이스의 상태 변화를 나타내는 데 사용된다. 이 state 스키마는 *Customer* 클래스 스키마에서 state 부분을 참조하여 작성한다. 그러나, 작성방법은 개발시스템의 정책에 따라 달라지기 때문에 변환 범위에 포함되지 않는다. 따라서 이 부분은 3장에서 정의한 인터페이스 명세 문법에 따라 명세한다. 인터페이스 그룹의 명세 부분은 7.2.2절의 변환된 명세(Trans CBFS)와 컴포넌트 기반 설계에서 정의한 인터페이스 그룹 명세(CBFS)가 일치한다.

컴포넌트를 명세한 부분은 7.2.3절의 변환된 명세(Trans CBFS)와 컴포넌트 기반 설계에서 정의한 컴포넌트 명세(CBFS)가 항목별로 일치한다.





8. 결 론

확장성과 재사용성을 높이는 객체지향 패러다임의 뒤를 이은 컴포넌트 기반 개발은 이미 개발된 소프트웨어 컴포넌트를 재사용하여 조립함으로써 소프트웨어 시스템을 개발하는 방법이다. 본 논문에서는 효율적인 컴포넌트 기반 시스템을 개발하기 위해 기 개발된 객체지향 설계 모델을 이용하여 컴포넌트 기반 설계로 변환하는 기법을 제시하였고, 변환의 정확성을 위해 정형명세 기법을 사용하였다. 우선 컴포넌트 기반 설계를 정형명세 하기 위해서 컴포넌트 정형명세 문법을 정의하였다. 그리고 객체지향 설계의 정적, 동적, 기능적 측면을 정형명세 언어 Object-Z를 사용하여 정형명세하는 기법을 제시하였다. 또한, 앞에서 작성된 객체지향 정형명세를 컴포넌트 정형명세로 변환하는 기법을 제시하였고, 변환툴을 이용하여 자동 변환 부분을 변환하였다. 사례연구는 제시된 변환 기법을 적용하여 객체지향 정형명세가 컴포넌트 기반 정형명세로의 변환과정을 설명하였다.

향후 연구과제로는 컴포넌트 기반 설계의 정형명세를 컴포넌트 기반 설계로 변환하는 기법까지 확장함으로써 기 개발된 객체지향 설계 모델을 이용하여 컴포넌트 기반 시스템을 구현하는 것이 효율적인 개발 방법임을 검증한다.

참 고 문 헌

[1] Szyperski, C., *Component Software- beyond Object-Oriented Programming*, Addison Wesley, 1997.

[2] Brown, A., Large-Scale, Component-Based Development, Prentice Hall, 2000.
[3] Larsson, M. and Crnkovic, I., "Development Experiences of a Component-Based System," *Seventh IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2000.
[4] McGibbon, B., "Status of CBSE in Europe," *Component-Based Software Engineering*, Addison-Wesley, 2001.
[5] Sousa, P. and Garlan, D., "Formal Modeling of the Enterprise JavaBeans Component Integration Framework," *Information and Software Technology*, Vol.43, 2001.
[6] OMG, UML Specification v1.4, September, 2001.
[7] Spivey, J., *The Z Notation A Reference Manual*, URL : http://spivey.oriel.ox.ac.uk/~mike/zrm/, 1992.
[8] Smith, G., *The Object-Z Specification Language*, Kluwer Academic Publishers, 2000.
[9] Smith, G., "An Object-Oriented Approach to Formal Specification," Doctoral Thesis, University of Queensland, 1992.
[10] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Uthing, D., Paech, B., Wuest, J. and Zettel, J., *Component-based Product Line Engineering with UML*, Addison-Wesley, 2001.
[11] Fischer, C., "Combination and Implementation of Processes and Data: from CSP-OZ to Java," Doctoral Thesis, University of Oldenburg, January 2000.
[12] Soon-Kyeong Kim and David Carrington, "A Formal Mapping between UML Models and Object-Z Specifications," International Conference of B and Z Users, Springer Verlag, Berlin, 2000.
[13] Booch, G., Rumbaugh, J. and Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.
[14] Butler, G., Grogono, P. and Khendek, F., "A Z Specification of Use Cases: A Preliminary Report," *4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC '97 / ICSC '97)*, IEEE Computer Society Press, Los Alamitos, CA, 1997.

신 숙 경
정보과학회논문지 : 소프트웨어 및 응용
제 31 권 제 5 호 참조

이 종 국
정보과학회논문지 : 소프트웨어 및 응용
제 31 권 제 5 호 참조

김 수 동
정보과학회논문지 : 소프트웨어 및 응용
제 31 권 제 1 호 참조