Component-Z: Object-Z를 확장한 컴포넌트 정형 명세 언어

(Component-Z: A Formal Specification Language Extended Object-Z for Designing Components)

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

요 약 컴포넌트 기반 소프트웨어 공학(CBSE)은 재사용 가능한 컴포넌트를 조립하여 시스템을 개발하는 방법이다. CBSE는 소프트웨어 개발 비용과 기간을 단축할 수 있는 새로운 패러다임으로 평가되고 있다. 정형 명세를 사용하면 컴포넌트 구성 요소들 사이의 일관성과 정확성을 판단할 수 있기 때문에 컴포넌트 설계의 품질을 높일 수 있다. 현재까지 제안된 컴포넌트 정형 명세 언어들은 인터페이스 간의 계약, 컴포넌트의 구조적인 측면과 동적인 측면, 컴포넌트 기반 시스템, 컴포넌트 간의 결합, 가변성 중 일부만을 명세에 반영한다. 따라서 지금까지 발표된 정형 명세 언어를 컴포넌트 설계 과정의 모든 단계에서 사용하는 것은 적절치 않다. 본 논문에서는 컴포넌트 정형 명세 언어인 Component-Z를 제안한다. Component-Z는 Object-Z를 확장하여 컴포넌트 명세를 위한 새로운 표기법을 추가하고 의미론을 확장한다. Component-Z를 사용하여 인터페이스, 컴포넌트 내부 구조, 컴포넌트 내부 워크플로우와 인터페이스 간의 워크플로우를 명세할 수 있다. 또한 가변점(variation point), 가변치(variant), customization 인터페이스를 사용하여 가변성을 명세할 수 있다. 인터페이스와 컴포넌트 사이의 관계는 매핑 스키마(mapping schema)를 사용하여 정의한다. 또한 병렬 연산자(parallel operator)를 사용하여 컴포넌트 간의 결합을 명세할 수 있고 컴포넌트 기반 시스템을 명세하여 컴포넌트가 배포된 상황을 표현할 수 있다. 따라서 본 논문에서 제안된 정형 명세 언어를 사용하여 컴포넌트 설계에 필요한 모든 요소를 표현 할 수 있다. 사례 연구에서는 은행 계좌 관리 시스템을 명세하여 Component-Z가 컴포넌트 설계의 전 단계에서 사용될 수 있음을 보인다.

키워드 : 컴포넌트 기반 소프트웨어 공학, 정형 명세, 가변성, Object-Z, CSP, 결합

Abstract Component-based software engineering (CBSE) composes reusable components and develops applications with the components. CBSE is admitted to be a new paradigm that reduces the costs and times to develop software systems. The high quality of component designs can be assured if the consistency and correctness among the elements of a component are verified with formal specifications. Current formal languages for components include only some parts of contracts between interfaces, structural aspects and behavioral aspects of component, component-based system, component composition and variability. Therefore, it is not adequate to use current formal languages in all steps of a component design process. In this paper, we suggest a formal language to specify component designs Component-Z. Component-Z extends Object-Z, adds new notations to specify components. It can be possible to specify interfaces, the inner structure of a component, inner workflows, and workflows among interfaces with Component-Z. In addition, Component-Z provides the notations and semantics to specify variability with variation points, variants and required interfaces. The relation between interfaces and components is defined with mapping schemas. Parallel operator is used to specify component composition. It can be possible to describe deployed components with the specifications of component-based systems. Therefore, the formal specification language

· 본 연구는 숭실대학교의 교내 연구비 지원으로 이루어졌음
† 비 회 원 : 숭실대학교 컴퓨터학과
jklee690@disc.co.kr
†† 정 회 원 : 숭실대학교 컴퓨터학과
skshin@otlab.ssu.ac.kr
††† 종신회원 : 숭실대학교 컴퓨터학과 교수
sdkim@ssu.ac.kr
논문접수 : 2003년 9월 18일
심사완료 : 2004년 3월 2일

proposed in this paper can represent all elements to design components. In the case study, we specify an account management system in a bank so that we show that Component-Z can be used in all steps of component design.

Key words : Component-Based Software Engineering (CBSE), Formal Specification, Variability, Object-Z, CSP, Composition

1. 서 론

컴포넌트 기반 소프트웨어 공학(CBSE)은 재사용 가능한 컴포넌트를 조립하여 시스템을 개발하는 방법이다. 전통적인 개발 방법에서는 개발자가 필요한 코드를 구현하여 시스템을 구축한다. 그러나 CBSE에서 개발자는 필요한 컴포넌트를 수집하고 조립하여 시스템을 구축한다. 따라서 CBSE는 소프트웨어 개발 비용과 기간을 단축할 수 있는 새로운 패러다임으로 평가되고 있다[1].

그러나 전통적인 개발 방법론 만으로는 재사용 가능한 컴포넌트를 설계 및 구현하기에 충분하지 않다. 재사용 가능한 컴포넌트를 구현하기 위해는 컴포넌트가 사용될 도메인을 분석하여 공통성과 가변성을 추출해야 한다. 컴포넌트는 인터페이스와 컴포넌트 내부의 객체, 워크플로우, 가변점 등 객체나 모듈보다 복잡하게 구성되어 있다. 또한 컴포넌트 사용자가 컴포넌트를 오류 없이 사용할 수 있도록 정확한 컴포넌트 명세서가 제공되어야 한다. 따라서 컴포넌트 설계는 객체지향 설계보다 많은 노력이 소요된다[2]. 정형 명세를 사용하면 컴포넌트 구성 요소들 사이의 일관성과 정확성을 판단할 수 있기 때문에 컴포넌트 설계의 품질을 높일 수 있다.

컴포넌트 정형 명세에는 다음과 같은 구성 요소들이 포함되어야 한다.

첫째, 인터페이스 명세를 통한 컴포넌트 사용자와 컴포넌트 간의 계약이 포함되어야 한다. 인터페이스 명세에는 오퍼레이션의 입, 출력 타입뿐만 아니라 선행, 후행 조건도 명세해야 한다.

둘째, 컴포넌트의 구조적인 특성을 반영해야 한다. 컴포넌트는 컴포넌트 사용자에게 오직 인터페이스만을 보여주는 블랙박스이므로 인터페이스와 컴포넌트 내부가 명세에서 분리되어야 한다. 컴포넌트 간의 상호 작용은 인터페이스 만을 사용하여 명세 되어야 하며 컴포넌트 내부에 대한 명세와 컴포넌트 내부의 설계 변경은 인터페이스와 인터페이스를 통한 컴포넌트의 결합에 영향을 미쳐서는 안 된다. 또한 대부분의 컴포넌트 내부 구현 기술이 객체로 이루어지는 것을 반영하여 컴포넌트 내부 명세는 객체로 명세 하는 것이 바람직하다.

셋째, 컴포넌트의 동적 명세가 포함되어야 한다. 컴포넌트 기반 시스템에는 인터페이스를 통한 컴포넌트 간의 상호 작용과 동시성이 명세 되어야 한다. 또한 컴포넌트는 내부에 객체를 포함하고 있으므로, 컴포넌트 내부의 객체 상호 작용이 명세되어야 한다.

넷째, 컴포넌트가 배포될 때 특화(customize)되는 상황을 명세에 포함해야 한다. 컴포넌트는 시스템에서 실행될 때, 특화되어 실행된다. 특화는 가변성을 통해 표현된다. 따라서 컴포넌트 명세에는 가변성과 가변성의 값을 설정하는 방법과 배포 후 설정된 가변성 값으로 컴포넌트가 실행되는 상황이 포함되어야 한다.

현재까지 제안된 컴포넌트 정형 명세 언어들은 인터페이스 간의 계약, 컴포넌트의 정적인 측면과 동적인 측면, 컴포넌트 기반 시스템, 컴포넌트 간의 결합, 가변성 중 일부만을 명세에 반영한다. 따라서 현재까지의 정형 명세 언어를 컴포넌트의 설계 과정의 모든 단계에서 사용하는 것은 적절하지 못하다.

본 논문에서는 컴포넌트 설계를 정형 명세하는 언어인 Component-Z를 제안한다. Component-Z는 Object-Z의 표기법과 의미론을 확장한다. Object-Z를 확장하기 위해 본 논문에서는 Object-Z의 표기법에 CSP의 표기법을 추가한다. Component-Z의 의미론에 대한 본 논문의 접근 방법은 Component-Z를 CSPz로 번역하는 것이다. 이 과정에서 동적인 부분은 CSP로 표현되고 정적인 부분은 Object-Z로 표현된다.

Component-Z를 사용하여 인터페이스, 컴포넌트 내부 구조, 컴포넌트 내부 워크플로우와 인터페이스 간의 워크플로우를 명세할 수 있다. 따라서 Component-Z는 컴포넌트의 정적인 구성 요소와 동적인 구성 요소를 모두 명세할 수 있다. 또한 가변점, 가변치, customization 인터페이스를 사용하여 가변성을 명세할 수 있다. Component-Z는 컴포넌트의 특성인 인터페이스와 컴포넌트의 분리, 컴포넌트의 동적인 구성 요소와 정적인 구성 요소의 분리를 반영하고 있다. 이러한 명세 사이의 분리는 복잡한 컴포넌트 명세를 쉽게 작성할 수 있도록 도와준다. 또한 명세 들 사이의 관계는 매핑 스키마(mapping schema)와 상속, 참조자를 사용하여 정의한다. 또한 병렬 연산자(parallel operator)를 사용하여 컴포넌트 간의 결합을 명세할 수 있고 컴포넌트 기반 시스템을 명세하여 컴포넌트가 배포된 상황을 표현할 수 있다. 따라서 본 논문에서 제안된 정형 명세 언어를 사용하여 컴포넌트 설계에 필요한 모든 요소를 표현 할

수 있다. 사례 연구에서는 은행 계좌 관리 시스템을 명세하여 Component-Z가 컴포넌트 설계의 전 단계에서 사용될 수 있음을 보인다.

2. 관련 연구

현재까지 연구된 컴포넌트 정형 명세는 집합을 사용한 정형 명세, 아키텍쳐 명세 언어를 사용한 컴포넌트 명세, 프로세스 기반의 정형 명세, 대수를 이용한 정형 명세, 두 가지 이상의 명세 언어를 사용한 복합 정형 명세로 분류할 수 있다. 또한 본 장에서는 컴포넌트의 특화를 기술하는 정형 명세 기법도 소개한다.

2.1 컴포넌트 정형 명세

집합을 사용한 정형 명세는 대부분 인터페이스를 통한 계약을 명세화하는데 중점을 둔다. Catalysis[3], UML Component[4]는 Object Contraint Language (OCL)을 사용하여 인터페이스의 선행, 후행 조건을 기술한다. 그러나 OCL만으로는 사용하기는 편리하나 기술된 인터페이스에 대한 정확한 의미론을 부여하기 어렵다. 따라서 Z와 Object-Z를 사용하여 인터페이스를 기술하려는 방법이 제안되었다[5,6]. Z와 Object-Z 등의 상태 기반 정형 명세는 인터페이스의 의미론을 정의하여 설계 상의 오류를 검증하기 쉽다. Object-Z를 사용하여 인터페이스뿐 아니라 컴포넌트의 구조적인 면과 동적인 면도 명세 하려는 시도가 있다[7]. 그러나 컴포넌트의 동적인 면은 명세를 실행 한 후에 검증이 가능하며 검증에 많은 시간이 소요된다.

Componentware는 집합과 함수만을 사용하여 인터페이스, 컴포넌트, 컴포넌트 사이의 상호작용을 간결하게 표현한다[8]. 컴포넌트와 인터페이스의 생성과 삭제는 술어(predicate)를 통해 표현되며 메시지의 흐름은 순서쌍을 통해 표현된다. 그러나 Componentware는 명세 검증을 위한 의미론이 정확히 정의되어 있지 않다.

아키텍처 명세 언어는 소프트웨어 시스템의 구조와 행동을 개괄적으로 묘사한다. 아키텍처 명세 언어는 전형적인 경우 컴포넌트, 컨넥터(connector), 구성도(configuration)로 이루어져 있다[9]. 아키텍처에서 컴포넌트는 논리 처리의 단위이고 컴포넌트 인터페이스는 컴포넌트와 주변 환경 사이에 상호 작용이 발생하는 지점이다. 컨넥터는 컴포넌트들 사이의 상호작용을 표현한다. 구성도는 컴포넌트와 컨넥터를 사용하여 시스템이 어떻게 구성되는지를 보여준다.

최근의 연구는 아키텍처 명세 언어가 컴포넌트를 명세하는 데도 유용하다는 것을 보여준다[10]. 어떤 아키텍처 명세 언어는 인터페이스의 선행, 후행 조건, 동기화와 메시지 순서와 같은 시스템의 제약 사항을 표현할 수 있다. 이런 명세 언어의 특징은 컴포넌트를 표현하는 데도 유용하다. 그러나 아키텍처 명세 언어는 단지 컴포넌트의 추상적인 면만을 묘사할 수 있으며 컴포넌트의 또 다른 중요한 특성인 가변성이나 컴포넌트 내부 구조, 인터페이스 등을 명세하기에는 한계가 있다.

Z와 Object-Z 등 상태 기반 정형 명세는 컴포넌트의 동적인 면을 표현하고 검증하는데 한계가 있기 때문에 컴포넌트의 동적 명세는 대부분 프로세스 기반의 정형 명세를 사용한다[11-13]. 그러나 상태 기반 정형 명세가 인터페이스를 통한 계약을 명세할 수 있는 반면 동적 명세는 인터페이스를 사용한 계약, 컴포넌트의 구조적인 면을 표현하기 어렵다는 한계가 있다.

대수를 사용한 정형 명세는 상태 기반 정형 명세와 프로세스 기반 정형 명세의 단점을 극복할 수 있는 방법이다. 특히 카테고리와 시제 논리(temporal logic)을 사용한 정형 명세 언어는 인터페이스를 이용한 계약, 시스템의 동적인 면, 구조적인면을 표현할 수 있고 명세의 결합도 모피즘을 사용하여 표현할 수 있다. 또한 의미론이 수학적으로 명확하기 때문에 오류 검증도 쉽다. 그러나 아직까지는 객체에 인터페이스를 사용한 계약을 추가하거나[14] 모듈의 결합을 명세하거나[15] 서브 시스템을 명세하는 데 대수 기반 정형 명세를 사용하고 있다[16]. 따라서 컴포넌트의 특성을 반영한 정형 명세를 정의하기 위해서는 더 많은 연구가 필요하다.

컴포넌트의 동적, 구조적 특성을 모두 명세하기 위해서는 기존의 정형 명세로는 한계가 있다. 따라서 현재 상태 기반 정형 명세와 프로세스 기반 정형 명세를 통합하는 연구가 진행되고 있다. 통합 정형 명세의 장점은 시스템의 동적인 면은 프로세스 기반 정형 명세의 표기법을 사용하고 시스템의 상태는 상태 기반 정형 명세의 표기법을 사용하기 때문에 표현이 간결하고 명세의 표현력이 좋다는 것이다[17,18]. 복합 정형 명세를 사용하면 컴포넌트의 동적인 특성, 구조적인 특성, 인터페이스를 사용한 계약을 모두 표현할 수 있다. 현재 복합 정형 명세를 컴포넌트에 적용하는 연구가 진행 중이다[19]. 본 논문에서는 복합 정형 명세 중 하나인 CSP-OZ[17]의 표기법과 의미론을 사용하여 컴포넌트를 명세한다.

2.2 컴포넌트 특화

컴포넌트 특화는 컴포넌트 구현 모델인 CORBA Component Model(CCM), Enterprise Java Beans (EJB)에 포함되어 있다. EJB는 XML에 컴포넌트 배포 시 변경되는 부분을 기술하고 컴포넌트는 배포 시에 특화된 부분을 코드로 생성하여 특화 되지 않은 코드와 함께 실행한다. CCM에서는 Configurator 인터페이스를 사용하여 컴포넌트를 특화하는 메커니즘을 제공한다.

몇몇 컴포넌트 개발 방법론들은 컴포넌트 특화를 지원한다. Catalysis에서는 플러그-인 객체를 사용한 컴포

넌트 설계 패턴을 제시한다. KorbA에서는 가변성의 개념을 사용하여 분석 단계에서 가변성을 식별하고 설계 단계의 산출물에 가변성을 명세하고 가변성을 통하여 컴포넌트를 설계하는 단계를 제시한다.

컴포넌트 특화의 구현과 설계 시 산출물의 일관성, 정확성이 보장되어야 한다. Piccola는 PI calculus를 사용하여 컴포넌트 특화에 대한 설계 산출물의 오류를 검증하려고 시도한다[13]. 또한 Petri Net을 사용하여 플러그-인 객체와 특화된 컴포넌트를 표현하고 오류를 검출하려는 시도도 있다[20]. 특화에 대한 보다 일반적인 접근 방법은 connector를 사용하여 컴포넌트를 시스템에 적응(adapt)시키는 방법[21], white box 컴포넌트에 대하여 aspect와 weaver를 사용하여 컴포넌트를 특화하는 방법이다[22]. Abmann은 whitebox 컴포넌트 특화에 대한 메타 모델과 정형 명세, 오류 검증 방법, 구현 방법 등을 제시하였다[23].

현재까지의 컴포넌트 특화에 대한 정형 명세는 컴포넌트 정형 명세와는 별도로 진행되었으며 컴포넌트 정형 명세와의 통합을 시도한 예는 드물다. 본 논문에서는 가변성을 이용한 파라메터 방식의 컴포넌트 특화 명세 방법, CCM의 Configurator 인터페이스를 일반화한 customization 인터페이스의 사용, 컴포넌트 명세와 특화 명세의 통합을 제시한다.

3. Component-Z의 참조 모델

이 장에서는 참조 모델을 통해 컴포넌트가 가져야 할 구성 요소를 제시한다. 컴포넌트 구현 모델과 컴포넌트 개발 방법론마다 컴포넌트에 대한 정의가 다르기 때문에 컴포넌트 정형 명세를 제시하기 전에 컴포넌트에 대한 일반화된 정의가 필요하다. 그림 1은 본 논문에서 제시하는 컴포넌트 참조 모델이다. 3.1절에서 3.5절까지는 그림 1에서 제시한 참조 모델의 각 항목에 대하여 설명하고 3.6절에서는 각 항목 사이의 관계에 대하여 설명한다.

3.1 인터페이스

인터페이스는 컴포넌트가 제공하는 서비스이다. 인터페이스는 오퍼레이션의 집합이며 오퍼레이션은 입력 매개 변수와 출력 매개 변수로 구성된다. 인터페이스는 컴포넌트 사용자와 컴포넌트 사이의 계약이다. 따라서 컴포넌트는 사용자가 올바른 입력 값을 넣었을 때에 올바른 출력 값을 반환한다는 것을 보장해야 한다. 올바른 입력값과 출력 값의 보장은 인터페이스 오퍼레이션의 선행, 후행 조건으로 표현된다.

인터페이스는 provided 인터페이스, required 인터페이스, customization 인터페이스로 분류된다. Provided 인터페이스는 기능성을 제공하는 인터페이스이다. Required 인터페이스는 컴포넌트가 다른 컴포넌트의 기능성을 사

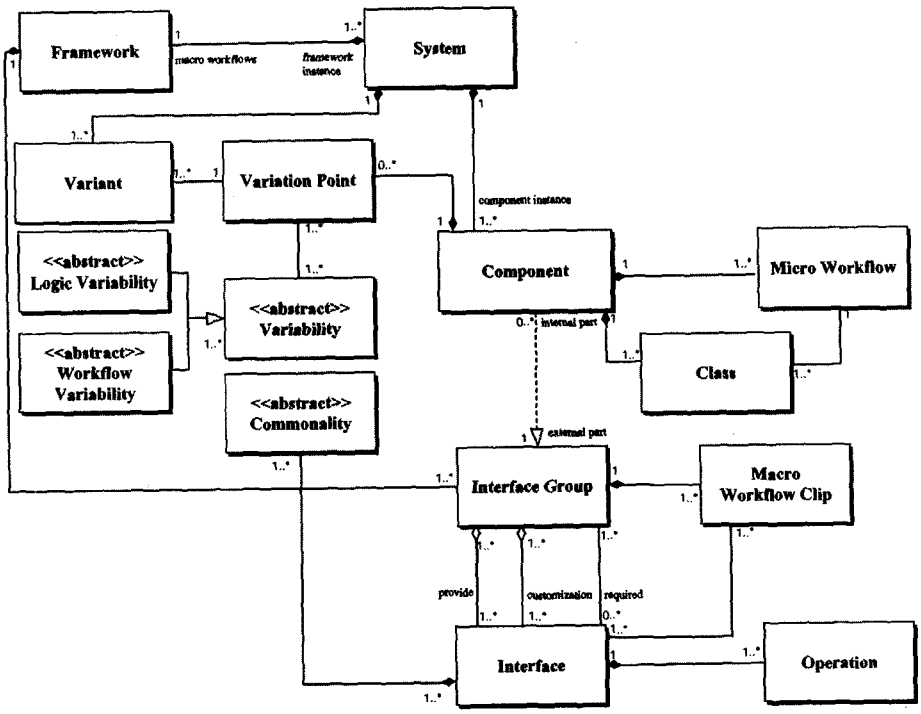


그림 1 컴포넌트 참조 모델

용하는 인터페이스이다. Customization 인터페이스는 가변치를 설정할 때 사용된다.

인터페이스는 컴포넌트와 독립적이다. 한 인터페이스는 여러 컴포넌트에 포함될 수 있고 한 컴포넌트가 여러 개의 인터페이스를 가질 수 있다. 따라서 인터페이스는 컴포넌트와는 독립적으로 명세 되어야 한다.

3.2 컴포넌트

컴포넌트는 복잡한 구조를 가지고 있다. 컴포넌트를 정의하기 위해서 컴포넌트를 기능적, 구조적, 동적 구성 요소로 나누어 설명한다.

3.2.1 기능적 구성 요소

기능적인 관점에서 컴포넌트는 복수 개의 기능을 인터페이스를 통해 제공한다. provided 인터페이스, required 인터페이스, customization 인터페이스는 컴포넌트의 외부적인 구성 요소이다. 이 외부적인 구성 요소를 인터페이스 그룹이라고 부른다.

3.2.2 구조적 구성 요소

내부 관점에서 컴포넌트는 복합 객체이다. 따라서 컴포넌트는 복수 개의 객체로 구성된다. 객체는 인터페이스에서 정의된 기능을 구현한다. 따라서 컴포넌트의 내부 객체는 인터페이스에서 정의된 술어와 제약 사항을 만족해야 한다. 그러나 컴포넌트 내부 객체의 변경이 인터페이스에 영향을 끼칠 수 없다.

컴포넌트는 복수 개의 버전을 가진다. 한 컴포넌트의 버전은 같은 인터페이스 그룹을 공유한다. 컴포넌트의 각 버전은 다른 객체들로 구성될 수 있다. 즉 컴포넌트 개발자는 내부 객체를 수정할 수 있지만 인터페이스 그룹은 수정할 수 없다.

3.2.3 동적 구성 요소

동적인 관점에서 컴포넌트는 객체들의 collaboration이다. 객체 collaboration은 객체 간의 메시지 흐름으로 구성된다. 이런 컴포넌트 내부의 메시지 흐름은 마이크로 워크플로우(micro workflow)라고 부른다. 마이크로 워크플로우는 업무 워크플로우를 구현한다. 즉 컴포넌트는 업무 워크플로우를 캡슐화하며 이를 통해 컴포넌트의 재사용성이 높아진다.

컴포넌트 사용자는 컴포넌트 내부 객체에게 직접 메시지를 보낼 수 없다. 컴포넌트 사용자는 provided/customization 인터페이스를 통해서만 간접적으로 내부 객체에게 메시지를 보낼 수 있다.

컴포넌트 외부에서 보면 인터페이스 그룹은 시스템 워크플로우에 참여한다. 사용자는 provided/customization 인터페이스를 통해 메시지를 보낸다. 그러면, provided/customization 인터페이스를 가지고 있는 컴포넌트가 required 인터페이스를 호출한다. 이런 provided/customization 인터페이스와 required 인터페이스의 메시지 흐름을 macro workflow clip이라고 부른다. Macro workflow clip은 시스템 워크플로우의 일부분이다.

3.3 가변성

컴포넌트는 한 도메인에서 복수 개의 어플리케이션이 공유하는 공통 기능을 가지고 있다. 이 공통 기능의 집합을 공통성(commonality)라고 부른다. 공통성은 여러 어플리케이션에서 컴포넌트를 재사용하기 위한 중요한 개념 중 하나이다. 공통성은 provided 인터페이스, 컴포넌트 내부 객체, 마이크로 워크플로우를 사용하여 구현된다.

그러나 공통성을 컴포넌트에서 구현하는 것 만으로는 재사용을 위해 충분하지 않다. 컴포넌트를 사용하는 특정 어플리케이션은 특정 기능이 컴포넌트에 포함되기를 기대할 수도 있다. 따라서 컴포넌트는 특정 어플리케이션을 위해 특화가 가능해야 한다.

가변성(variability)은 특화를 위한 중요한 개념 중 하나이다[24]. 가변성은 한 도메인의 어플리케이션 마다 각각 다른 특성으로 정의된다. 본 논문에서는 가변성의 개념을 세분화 하여 논리 가변성과 워크플로우 가변성으로 나눈다. 논리 가변성은 업무 규칙, 알고리즘이 어플리케이션마다 다른 것이고 워크플로우 가변성은 워크플로우, 객체 상호 작용이 어플리케이션마다 다른 것이다.

가변성을 구현하기 위해, 컴포넌트는 가변점을 갖는다. 사용자는 인터페이스를 호출하여 가변점에 값을 설정한다. 가변점에 값을 설정하는 인터페이스를 customization 인터페이스라고 부른다. 그리고 설정되는 값은 가변치라고 부른다. 가변치는 배포 시에 컴포넌트에 설정된다. 가변치를 가진 컴포넌트 만이 실행될 수 있다.

3.4 프레임워크

프레임워크는 인터페이스그룹의 결합으로 구성된다. 프레임워크는 컴포넌트 간의 계약, 사용자와 컴포넌트 간의 계약, 컴포넌트로 이루어진 시스템의 동적인 면을 보여준다. 프레임워크는 컴포넌트 내부 구조와 가변성은 포함하지 않는다. 프레임워크에서는 컴포넌트 간의 상호 작용이 표현된다. 프레임워크에서 컴포넌트 간의 상호 작용은 컴포넌트 내부 메시지 흐름은 포함하지 않는다. 프레임워크에서 컴포넌트 간의 상호 작용은 'Macro Workflow'라고 부른다.

3.5 시스템

시스템은 프레임워크에 컴포넌트의 인스턴스가 '플러그-인'된 것이다. 컴포넌트가 '플러그-인'된다는 말은 컴포넌트가 '배포'된다 라고 표현할 수 있다. 컴포넌트가 배포될 때 컴포넌트의 가변점에는 가변치가 설정되어야 한다.

3.6 구성 요소들 사이의 관계

그림 1은 컴포넌트 구성 요소들 사이의 관계를 보여준다. 그림 1에서 'abstract'라는 스테레오 타입을 가진

클래스는 컴포넌트의 요구사항 수집, 분석 단계에서 어떻게 식별되는지를 보여준다. 공통성와 가변성은 컴포넌트가 가져야 할 특성을 보여준다. 공통성은 인터페이스를 통해 컴포넌트 설계에 반영되고 가변성는 가변점을 통해 컴포넌트 설계에 반영된다. 컴포넌트는 internal part와 external part로 나누어진다. 그림 1에서 'Component'는 컴포넌트의 내부이며 'Interface Group'은 컴포넌트의 외부이다. 컴포넌트는 가변점과 클래스, micro workflow로 구성된다. 인터페이스 그룹은 provide 인터페이스와 customization 인터페이스, macro workflow clip으로 구성되며 required 인터페이스를 참조한다. 컴포넌트는 인터페이스 그룹을 구체화한다. 따라서 컴포넌트 내부는 인터페이스의선행, 후행 조건과 macro workflow clip의 메시지 흐름 등의 제약 사항을 만족해야 한다. 프레임워크은 인터페이스 그룹의 결합이다. 인터페이스 그룹은 macro workflow clip을 포함하고 있기 때문에 프레임워크에는 macro workflow가 나타난다. 시스템은 프레임워크, 가변치, 컴포넌트로 구성된다. 시스템에는 컴포넌트가 인스턴스로 포함된다.

4. Component-Z의 문법

이 장에서는 Component-Z의 문법을 설명한다. 추가로 본 논문에서 제시된 언어의 의미도 설명한다. 정확하고 상세한 의미론은 5장에서 설명한다.

Component-Z의 문법은 Object-Z의 문법을 기반으로 한다[25]. Component-Z의 문법은 정의하기 위해 다음과 같은 표기법을 사용한다.

::=	생성한다
|	선택
[x]	x가 없어도 됨
{x}	x가 없거나 여러 개 있음

xList = *x*{,*x*}

선행자(prefix) ℙ는 집합을 가리킨다. 이 집합의 한 원소는 한 라인에 나타나고 다른 원소는 별도의 줄에 나타난다. 후행자(suffix) *Name*은 이전 줄에서 정의된 타입을 가리킨다. 문법 구성 요소의 이름은 후행자에 Name을 사용하여 표시한다.

Component-Z의 문법은 3장의 참조 모델을 반영한다. 다음 문법은 Component-Z의 전체 구조를 보여준다.

CZ는 Component-Z의 약자이다. Component-Z는 여러 개의 단락으로 구성된다. 한 단락은 Z 단락이거나 Object-Z 단락, Component-Z 단락이다. 따라서 Component-Z는 Z와 Object-Z를 포함한다. $Paragraph_{CZ}$는 컴포넌트의 구성 요소를 보여준다. 한 컴포넌트 명세는 여러 개의 인터페이스와 한 개의 인터페이스 그룹과 컴포넌트로 구성된다.

Language::=	ℙ*Paragraph*
Paragraph::=	$Paragraph_Z$ | $Paragraph_{Open\_Z}$ | $Paragraph_{CZ}$
$Paragraph_{CZ}$::=	ℙ*Interface Separator InterfaceGroup Separator Component Separator Framework Separator System*
Separator::=	*NewLine*

그림 2 Component-Z의 전체 구조

4.1 인터페이스

인터페이스는 컴포넌트가제공하는 서비스이다. 문법적으로 인터페이스는 오퍼레이션의 집합이다.

Interface::=	*ProvideInterface* | *CustomizationInterface*
ProvideInterface ::=	*IProvInterfaceName* [ℙ*BaseInterface*] ℙ*Operation*
CustomizationInterface ::=	*ICusInterfaceName* ℙ*Operation*
Operation::=	*OperationName* *Signature* *Semantic* //Z의 표현과 동일함
Signature::=	[*DeltaList Seperator*] [*XiList Separator*] ℙ*Parameter*
DeltaList::=	Δ(*AbstractStateSchemaNameList*)
XiList::=	Ξ(*AbstractStateSchemaNameList*)
Parameter::=	*InputParameter* | *OutputParameter*
BaseInterface::=	*InterfaceName*

그림 3 인터페이스의 문법

위의 문법에서 StateSchemaName은 인터페이스 오퍼레이션에서 참조하는 상태 스키마의 이름을 가리킨다. Semantic은 오퍼레이션의 선행, 후행, 불변 조건을 가리킨다. InputParameter는 입력 매개변수의 선언, OutputParameter는 출력 매개 변수 선언을 가리킨다.

Provided 인터페이스의 이름은 'IProv'로 시작되고 customization 인터페이스의 이름은 'ICus'로 시작된다. 오퍼레이션은 Z 스키마로 표현된다. Provided 인터페이스는 상속을 허용한다. 의미상으로 인터페이스는 객체의 일종이기 때문에 상속이 허용될 수 있다. 또한 인터페이스는 객체의 일종이기 때문에 참조자를 갖는다. Customization 인터페이스는 컴포넌트에 가변치를 설정하는 단순한 기능을 갖고 있으며 상속을 허용하지 않는다. Required 인터페이스는 컴포넌트에서 호출하는 인터페이스로 Provided 인터페이스와 동일하기 때문에 별도의 문법이 필요하지 않다.

인터페이스 오퍼레이션은 오퍼레이션의 의미를 표현하기 위해 술어를 가져야 한다. Z에서 술어를 표현할 때 타입들이 필요하고 일반적으로 Z에서는 술어를 표현하는 타입은 상태 스키마(state schema), 오퍼레이션은 오퍼레이션 스키마(operation schema)로 나누어 표현한다. 그러나 인터페이스는 실제적인 상태를 가질 수 없는 단순한 기능 명세의 집합이다. 인터페이스의 표현과 의미상의 모순을 해결하기 위해 본 논문에서는 abstract

stateschema와 스키마 매핑(schema mapping)의 개념을 도입한다. 인터페이스가 참조하는 상태 스키마는 실제 상태 스키마가 아니다. 이 상태 스키마는 컴포넌트 상태 스키마의 일부일 뿐이다. 그림 2를 보면 사용자는 인터페이스의 abstract state schema를 관찰한다. 이 스키마는 보이지 않는 컴포넌트 내부 상태 스키마의 단면이다. abstract state schema와 컴포넌트 상태 스키마 사이에는 매핑 관계가 성립한다.

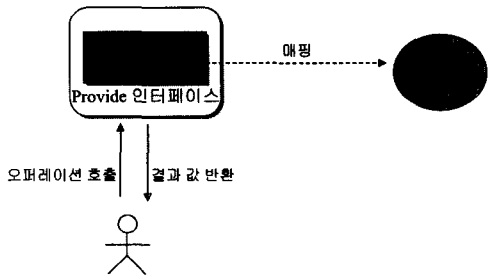


그림 4 인터페이스 모델

그림 4에서는 인터페이스가 어떻게 실행되는지를 보여준다. 사용자는 인터페이스의 참조자를 가진다. 사용자의 관점에서 인터페이스는 객체와 동일하다. 사용자는 인터페이스의 오퍼레이션을 부르고 사용자가 결과 값을 반환할 때까지 기다린다. 사용자가 결과 값을 받을 때 사용자는 abstract state가 변경되었다고 생각한다. 그러나 실제로 변하는 것은 매핑된 컴포넌트의 상태 스키마이다.

Abstract state schema과 컴포넌트 상태 스키마의 매핑은 인터페이스에서 정의되지 않고 컴포넌트에서 정의된다.따라서 인터페이스는 관련된 컴포넌트가 무엇인지 알지 못하며 컴포넌트와 독립적으로 기술된다.

4.2 인터페이스 그룹

외부적으로 컴포넌트는 복수 개의 인터페이스를 갖는다. 컴포넌트가 가지고 있는 인터페이스의 집합을 인터페이스 그룹이라고 부른다. 컴포넌트는 Provided 인터페이스와 customization 인터페이스를 통해 메시지를 받고 required 인터페이스를 통해 외부에 메시지를 보낸다. Provided 인터페이스에 메시지가 들어오는 것은 'action', 컴포넌트 내부의 객체들이 호출된 후 required 인터페이스 가 호출되는 것은 'reaction'이라고 부른다 [12]. 이 인터페이스들 사이의 메시지 흐름을 macro workflow clip이라고 부른다. 인터페이스 그룹은 한 컴포넌트가 가지고 있는 모든 인터페이스 들 사이의 상호작용을 보여준다.

인터페이스 그룹의 이름은 'IG'로 끝난다. 인터페이스 그룹은 provided, required, customization 인터페이스와

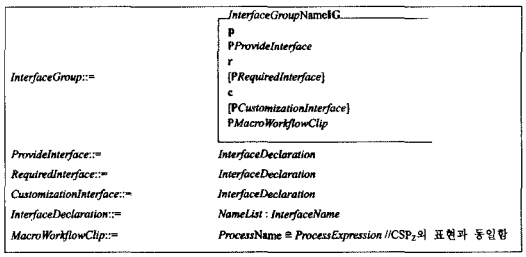


그림 5 인터페이스 그룹의 문법

macro workflow clip으로 구성된다. p, r, c은 세 종류의 인터페이스를 구별하는 구별자이다. macro workflowclip은 action-reaction의 쌍이며 CSP로 표현된다. CSP의 표기법이 다양하기 때문에 본 논문에서는 CSPz [17]의 표기법을 사용한다.

인터페이스 그룹에서 인터페이스 오퍼레이션은 CSP 프로세스에 매핑된다. 오퍼레이션을 프로세스에 매핑한 이유는 provide 인터페이스의 오퍼레이션이 수행되는 동안 required 인터페이스가 호출될 수 있기 때문이다. 오퍼레이션에 대응하는 프로세스는 두 개의 채널을 갖는다. 하나는 입력 값을 받는 채널이고 다른 하나는 출력 값을 보내는 채널이다. 첫 번째 채널은 오퍼레이션의 이름 끝에 b를 붙이고 두 번째 채널은 e를 붙여서 구별한다. 예를 들어 다음과 같이 표현한다.

$$CallingProcessOfOp \mid pi.op_b \ \phi \ pi.op_e \quad (1)$$

Provided 인터페이스의 오퍼레이션이 수행되는 동안 required 인터페이스의 오퍼레이션이 호출된다면 다음과 같이 표현한다.

$$CallingProcessOfOp \mid pi.op_b \phi ri.op_1 \phi pi.op_e \quad (2)$$

$ri.op_1$은 required 인터페이스 ri의 오퍼레이션 op_1을 가리킨다. $ri.op_1$은 $ri.op_{1b} \ \phi \ ri.op_{1e}$를 축약한 것이다. $pi.op_b$는 채널 op_b를 통해 받을 수 있는 모든 이벤트를 가리키며 이 이벤트에는 인터페이스의 참조자 pi와 오퍼레이션의 입력 매개 변수 값이 들어 있다. $pi.op_e$는 채널 op_e를 통해 pi와 출력 매개 변수를 갖고 있는 이벤트를 보낸다.

인터페이스의 오퍼레이션은 인터페이스 그룹의 오퍼레이션과는 다르다. 인터페이스의 오퍼레이션은 인터페이스 그룹의 오퍼레이션을 호출한다. 호출된 인터페이스 그룹의 오퍼레이션은 컴포넌트 내부의 오퍼레이션을 호출한다. 컴포넌트 내부의 오퍼레이션은 컴포넌트의 상태를 변경한다.

그림 6은 인터페이스 그룹의 실행 모델을 보여준다.

그림 4에서 회색 부분이 인터페이스 그룹이다. Provided 인터페이스는 인터페이스 그룹에게 메시지를 보낸다. 인터페이스 그룹은 어떤 required 인터페이스가 호

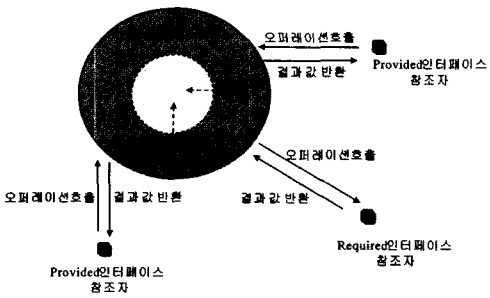


그림 6 인터페이스 그룹 모델

출되어야 하는지를 알고 있다. 인터페이스 그룹은 provided 인터페이스의 메시지가 처리되는 동안 관련된 컴포넌트에게 메시지를 보낸다.

인터페이스 그룹은 관련된 컴포넌트를 알지 못하며 다만 주변 환경에 메시지를 보낼 뿐이다. 따라서 인터페이스 그룹과 컴포넌트도 인터페이스와 컴포넌트의 관계처럼 독립적으로 기술된다.

4.3 컴포넌트

외부적인 관점에서 인터페이스 그룹은 컴포넌트의 동적인 면을 표현한다. 내부적인 관점에서 컴포넌트는 객체들의 상호작용이다. 즉 컴포넌트는 내부에서는 복합 객체로 보인다. 따라서 컴포넌트의 문법은 Object-Z의 문법과 동일하다.

컴포넌트의 이름은 '**COMP**'로 끝난다. *MicroWorkflow*는 컴포넌트 내부의 객체 상호 작용을 표현한다. *LocalDefinitions*, *InitialSchema*, *OperationExpression*는 Object-Z의 문법과 동일하다. *ComponentStateSchema*에서 ω는 Object-Z state schema에는 없는 것

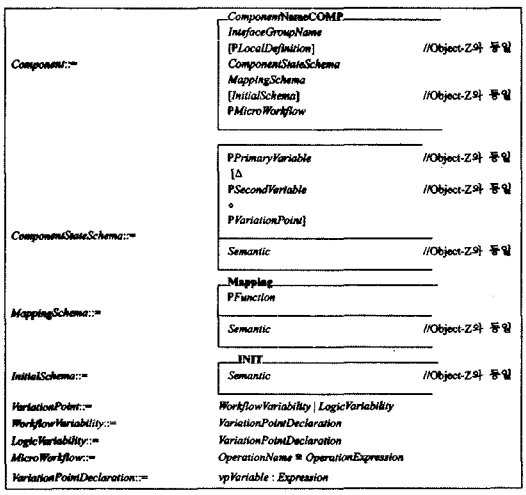


그림 7 컴포넌트의 문법

으로 새로 추가되었다. ω는 가변점을 상태 변수와 구별하는 구별자이다. 가변점은 오퍼레이션의 술어나 *MicroWorkflow*에 사용될 수 있다. 가변점이 오퍼레이션의 술어에서 사용될 경우는 논리 가변성이고 *MicroWorkflow*에서 사용될 경우는 워크플로우 가변성이다. 가변점의 이름은 *vp*로 시작하여 상태 변수와 구별한다.

*MappingSchema*는 인터페이스에서 참조된 abstract state schema와 컴포넌트의 state schema를 매핑한다.

컴포넌트 내부에 포함된 인터페이스 그룹 명은 컴포넌트가 특정 인터페이스 그룹의 제약 사항을 만족한다는 것을 의미한다.

한 컴포넌트는 관련된 한 인터페이스 그룹에서만 사용될 뿐 다른 어디에서도 참조되지 않는다. 따라서 컴포넌트는 참조자를 가질 필요가 없다. 컴포넌트가 참조자를 갖지 않는 복합 객체이므로 컴포넌트 내부 객체도 참조자가 없는 객체로 정의한다.

그림 8은 컴포넌트의 구성과 실행 순서를 보여준다. 컴포넌트는 인터페이스와 인터페이스 그룹, 컴포넌트 내부로 구성된다. 컴포넌트는 인터페이스 그룹을 통해 requied 인터페이스를 호출할 수 있지만 컴포넌트 내부에서 requied 인터페이스를 직접 호출하는 것은 금지된다. 컴포넌트 내부는 컴포넌트의 상태만을 변경할 수 있다.

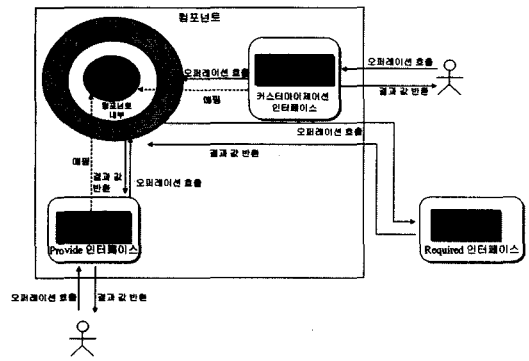


그림 8 컴포넌트 모델

4.4 컴포넌트 프레임워크

컴포넌트 프레임워크는 인터페이스 그룹으로구성된다. 컴포넌트 프레임워크는 시스템의 템플리트이다.

컴포넌트 프레임워크의 이름은 '**FW**'로 끝난다. 컴포

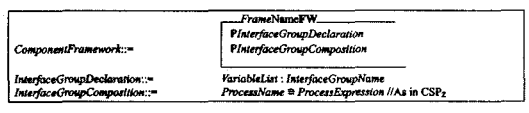


그림 9 컴포넌트 프레임워크

넌트 프레임워크는 인터페이스 그룹의 상화 작용을 명세한다. *InterfaceGroupDeclaration*은 인터페이스 그룹의 한 변수를 선언한다. *InterfaceGroupComposition* 인터페이스 그룹 사이의 상호 작용을 명세한다. *Interface-GroupComposition*와 CSPZ의 표기법을 사용한다.

그림 10은 컴포넌트 프레임워크의 실행 모델을 나타난다. 프레임워크는 시스템의 동적인 면을 타낸다. 프레임워크는 컴포넌트 내부를 숨기고 인터페이스 그룹만을 보여준다. 즉 컴포넌트 프레임워크는 시스템의 추상적인 면을 나타낸다.

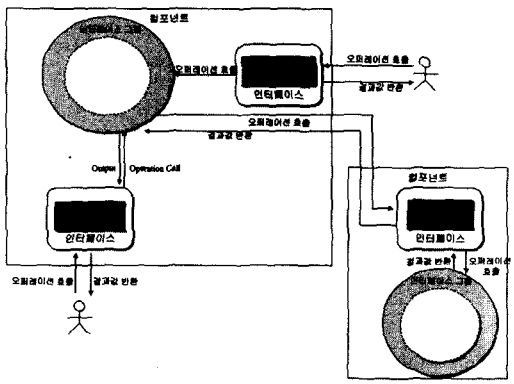


그림 10 컴포넌트 프레임워크

4.5 시스템

시스템 명세는 Z 스키마를 사용한다.

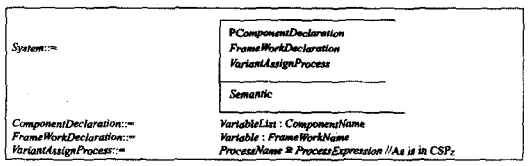


그림 11 시스템의 문법

스키마의 변수들은 컴포넌트 인스턴스와 프레임워크 인스턴스를 나타낸다. *Semantic*에는 컴포넌트 인스턴스가 인터페이스 그룹 변수에 할당된다. 이것은 컴포넌트가 인터페이스의 하위 클래스이기 때문에 가능하다. *VariantAssignProcess*는 customization 인터페이스를 사용하여 가변치를 가변점에 할당한다. *VariantAssign-Process*는 CSPZ 표기법을 사용하여 가변치를 할당하는 과정을 표현한다.

그림 12는 시스템의 한 예를 보여준다. 시스템에서 컴포넌트는 프레임워크에 결합된다. 즉 컴포넌트는 프레임워크에 '플러그-인'된다.

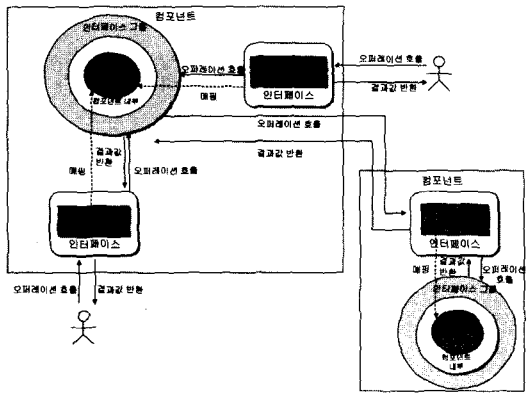


그림 12 컴포넌트 기반 시스템

5. Component-Z의 의미론

Component-Z에서는 CSP와 Object-Z의 표기법이 공존한다. CSP는 컴포넌트의 동적인 면을 표현하기 위해 사용되고 Object-Z는 정적인 면을 표현하기 위해 사용된다. 따라서 Component-Z의 의미론도 CSP의 의미론과 Object-Z의 의미론을 통합한다. CSP와 Object-Z를 결합하는 방법은 다양하다. 본 논문에서는 CSP-OZ의 접근 방법을 사용한다[17]. 표기법은 메타 함수를 사용하고 특정 컴포넌트와 인터페이스의 메타 함수는 컴포넌트와 인터페이스의 이름을 첨자로 사용한다. 상세한 표기법은 [17]의 논문을 참고한다.

Object-Z의 의미론도 다양하다. 4장에서 컴포넌트와 컴포넌트 내부 객체는 참조자가 없는 객체라고 정의하였다. 참조자 없는 value 기반의 객체 의미론은 history 기반의 의미론으로 기술된다[26]. 따라서 본 논문에서는 history 기반의 Object-Z 의미론을 사용한다.

5.1 인터페이스의 의미론

인터페이스는 오퍼레이션의 그룹이다. Component-Z는 인터페이스를 객체로 간주한다. 그러나 인터페이스는 상태 스키마를 캡슐화하지 않기 때문에 일반적인 객체와는 다르다.

인터페이스의 정의는 다음과 같이 표현될 수 있다.

InterfaceName	
$I_1, I_2, I_3, \ldots, I_n$	[상속된 인터페이스]
$op_1, op_2, \ldots, op_m$	[오퍼레이션]

InterfaceName은 INT로 축약하여 표현한다.

오퍼레이션의 집합은 상위 인터페이스의 오퍼레이션과 인터페이스의 오퍼레이션의 합집합이다.

$$\mathbf{ops}_{INT} = \bigcup_{i=1}^{n} \mathbf{ops}_{Ii} \ Y \ \bigcup_{i=1}^{m} op_i \qquad (3)$$

함수 $\mathbf{input}_{INT}$, $\mathbf{output}_{INT}$는 주어진 오퍼레이션의 입

력, 출력 변수를 표현하는 스키마를 계산한다.

$$\mathbf{input}_{INT}(op) = [self : INT]\ f\ \wedge_{i=1}^{n} \mathbf{input}_{Ii}(op) \cdot \mathbf{input}(\mathbf{schema}(op)) \quad (4)$$

$$\mathbf{output}_{INT}(op) = [self : INT]\ f\ \wedge_{i=1}^{n} \mathbf{output}_{Ii}(op) \cdot \mathbf{output}(\mathbf{schema}(op)) \quad (5)$$

schema는 오퍼레이션의 스키마를 계산하는 함수이다. **input**, **output**은 스키마에서 입력 매개변수의 스키마와 출력 매개 변수의 스키마를 반환하는 함수이다. 입력 매개변수와 출력 매개변수에는 항상 인터페이스의 참조자가 포함된다. ·는 scope enrichment 연산자로 INT의 오퍼레이션의 scope를 상위 인터페이스의 오퍼레이션으로 확장한다.

인터페이스의 오퍼레이션은 두 개의 채널을 갖는다. op_b는 입력 값을 받는 채널이고 op_e는 출력 값을 보내는 채널이다. 오퍼레이션은 채널 op_b를 통해 입력 값을 받고 채널 op_e를 통해 출력 값을 보내는 CSP 프로세스다.

$$\mathbf{proO}_{INT}(op) \mid op_b?(\mathbf{input}_{INT}(op)) \phi\ op_e!(\theta\ \mathbf{output}_{INT}(op)) \phi\ \mathbf{procI}_{INT} \quad (6)$$

식 (6)에서 $op_b?(\mathbf{input}_{INT}(op))$는 op_b 채널로 들어오는 이벤트, $op_e!(\mathbf{output}_{INT}(op))$은 op_e 채널로 나가는 이벤트이다. CSP_Z에서 이벤트는 distributed free type이다. 따라서 $op_b?(\mathbf{input}_{INT}(op))$는 함수 op_b에 입력 값으로 스키마 $\mathbf{input}_{INT}(op)$를 넣었을 때 반환되는 값으로 해석한다. θ는 프로세스의 실행을 통해 **output** 스키마에 값이 설정된 것을 의미한다.

인터페이스의 의미는 다음과 같다.

$$\mathbf{procI}_{INT}\ \pi\ op{:}\mathbf{ops}_{INT} \cdot \mathbf{procO}_{INT}(op) \quad (7)$$

즉 인터페이스는 외부 사용자가 오퍼레이션을 선택한 후 그 오퍼레이션에 대응하는 프로세스를 실행하고 다시 외부 사용자가 오퍼레이션을 선택하기를 기다리는 프로세스다.

상속이 없는 customization 인터페이스는 식 (4), (5)에서 $\wedge_{i=1}^{n} \mathbf{input}_{Ii}(op)$, $\wedge_{i=1}^{n} \mathbf{output}_{Ii}(op)$가 생략된다.

Abstract state schema는 식 (7)의 프로세스에 참여하지 않으며 일관성 검증 용도로만 사용된다.

5.2 컴포넌트의 의미론

컴포넌트는 두 개로 나누어진다. 첫째는 인터페이스 그룹으로 컴포넌트의 동적인 부분을 나타내고 둘째는 컴포넌트의 상태 변화를 표현하는 정적인 부분이다. 인터페이스 그룹의 정의는 다음과 같이 표현된다.

InterfaceGroupName

p

$p_1 : IP_1; p_2 : IP_2; \ldots ; p_n : IP_n$

r

$r_1 : IR_1; r_2 : IR_2; \ldots ; r_m : IR_m$

c

$c_1 : IC_1; c_2 : IC_2; \ldots c_r : IC_r$

$P_1 \cong PE_1; P_2 \cong PE_2; \ldots ; P_k \cong PE_k$

$C_1 \cong PE_{k+1}; C_2 \cong PE_{k+2}; \ldots ; C_j \cong PE_{k+j}$

이 절에서는 컴포넌트가 n개의 provided 인터페이스와 m 개의 required 인터페이스, r 개의 customization 인터페이스를 가지고 있다고 가정한다. 또한 provided 인터페이스에 의해 호출되는 macro workflow clip이 k, customization 인터페이스에 의해 호출되는 macro workflow clip이 j 개 있다고 가정한다.

인터페이스 그룹의 프로세스에서 나타나는 reference. operation(*ref.op*)은 다음과 같이 해석된다.

$$\mathbf{input}_{Type(ref)}(ref.op) = [ref : \mathbf{Type}(ref)]\ \wedge_{i=1}^{n} \mathbf{input}_{Ii}(op) \cdot \mathbf{input}(\mathbf{schema}(op)) \quad (8)$$

$$\mathbf{output}_{Type(ref)}(ref.op) = [ref : \mathbf{Type}(ref)]\ \wedge_{i=1}^{n} \mathbf{output}_{Ii}(op) \cdot \mathbf{output}(\mathbf{schema}(op)) \quad (9)$$

식 (8), (9)에서 Type은 ref의 타입을 계산한다. Ii는 **Type**(ref)의 상위 인터페이스를 가리킨다. 모든 입력, 출력 값에는 인터페이스 참조자가 추가된다.

식 (10)의 프로세스는 입력 채널에 들어 오는 이벤트와 출력 채널로 나가는 이벤트를 보여준다.

$$\mathbf{proO}_{Type(ref)}(ref.op) \mid op_b?(\mathbf{input}_{Type(ref)}(ref.op)) \phi\ op_e!(\theta\ \mathbf{output}_{Type(ref)}(ref.op)) \quad (10)$$

인터페이스 그룹의 프로세스 P_1, P_2, ..., P_k는 식 (10)의 프로세스로 구성된다.

인터페이스 그룹 중 provided 인터페이스에 의해 호출되는 프로세스는 다음과 같다.

$$\mathbf{procIGP}_{IG} \triangleq |||_{i=1}^{k} P_i \phi\ \mathbf{procIGP}_{IG} \quad (11)$$

식 (11)에서 IG는 InterfaceGroupName의 약자이다.

인터페이스 그룹 중 cutomization 인터페이스에 의해 호출되는 프로세스는 다음과 같다.

$$\mathbf{procIGC}_{IG} \triangleq |||_{i=1}^{j} C_i \phi\ \mathbf{procIGC}_{IG} \quad (12)$$

컴포넌트의 정의는 다음과 같이 표현된다.

ComponentName

InterfaceGroupName	
$LD_1, \ldots, LD_p$	[지역 변수 정의]
ST	[상태 스키마]
IN	[초기화 스키마]
$op_1, op_2, \ldots, op_q$	[오퍼레이션]

컴포넌트 상태 스키마는 지역 변수 정의와 ST로 구성된다.

$$\mathbf{state}_{com} = \mathbf{state}(LD_1) \cdot \ldots \cdot \mathbf{state}(LD_n) \cdot \mathbf{state}(ST) \quad (13)$$

식 (13)에서 상태 스키마의 scope는 지역 변수로 확장된다.

Initial schema는 다음과 같이 정의된다.

$$\mathbf{init}_{com} = \mathbf{schema}(IN) \quad (14)$$

오퍼레이션의 집합은 다음과 같이 정의된다.

$$\mathbf{ops}_{com} = \bigcup_{i=1}^{q} op_i \quad (15)$$

op의 의미는 채널 op_b를 통해 입력 값을 받고 상태 변화를 schema prefixing를 사용하여 계산하고 채널 op_e로 출력값을 보내는 프로세스다.

$$\mathbf{procO}_{com}(op, \mathbf{state}_{com}) \triangleq op_b?(\mathbf{input}_{com}(op)) \phi \mathbf{schema}(op) \phi op_e!(\theta \mathbf{output}_{com}(op)) \phi \mathbf{procS}_{com}(\theta \mathbf{state}_{com}) \quad (16)$$

$$\mathbf{procS}_{com}(\mathbf{state}_{com}) \triangleq \sqcap op : \mathbf{ops}_{com} \cdot \mathbf{procO}_{com}(op, \theta \mathbf{state}_{com}) \quad (17)$$

식 (16)는 Object-Z의 history 기반 의미론을 내포하고 있다. 컴포넌트의 상태 스키마는 객체로 구성되어 있으며 한 객체는 이 객체를 생성하는 클래스에서 유도된 history 집합의 한 원소이다[26]. 따라서 컴포넌트의 상태 스키마와 오퍼레이션을 통한 state의 변화는 history를 통해 재구성되어야 한다. 즉 컴포넌트의 정의를 history를 통해 재구성한 후 식 (16), (17)를 계산해야 한다. 이 과정을 통해 CSP의 의미론과 Object-Z의 의미론이 통합된다. 본 논문에서는 컴포넌트의 의미론을 직관적으로 이해하기 위해 컴포넌트의 정의를 history를 사용하여 재구성하는 것은 생략한다. 복합 객체의 의미론은 [26]을 참조하면 된다.

컴포넌트의 정적인 부분은 다음과 같이 표현된다.

$$\mathbf{procD}_{com} \triangleq \mathbf{init}_{com} \phi \mathbf{procS}_{com}(\theta \mathbf{state}com) \quad (18)$$

컴포넌트의 의미는 인터페이스 그룹과 컴포넌트의 정적인 부분의 parallel composition이다.

$$\mathbf{proc'}_{com} \triangleq \mathbf{procIGP}_{com} [\{x \varepsilon \in (\{|c: Chans(\mathbf{procIGP}_{com})|\} - \mathbf{required}\{|c:Chans(\mathbf{procIGP}_{com})|\}) \mapsto x \backslash \mathbf{ref}(x)\}] \mathbf{procD}_{com} \quad (19)$$

식 (19)에서 Chans는 프로세스가 가지고 있는 이벤트를 계산한다. $\{|c : Chans(\mathbf{procIGP}_{com})|\}$는 $\mathbf{procIGP}_{com}$의 모든 가능한 채널에서 발생할 수 있는 이벤트를 가리킨다. **ref**(x)는 이벤트 x에 포함된 참조자를 리턴하는 함수이다. x**ref**(x)는 x가 스키마 타입의 이벤트이므로 스키마에서 참조자를 생략한다. [ ]는 linked parallel 연산자로 순서쌍의 첫 번째 이벤트와 두 번째 이벤트가 [] 양쪽의 프로세스에서 동기화 된다. **required**는 이벤트 중 required 인터페이스의 오퍼레이션을 호출하는 이벤트만 반환한다. 따라서 식 (19)는 인터페이스 그룹에서 보낸 이벤트 중 provide 인터페이스에서 발생한 이벤트를 컴포넌트에서 참조자를 제거하고 받는 것을 의미한다.

컴포넌트에서 provide 인터페이스 중 하나를 실행하는 것은 다른 provide 인터페이스를 실행하는 것에 무관하다. 이것은 interleaving 연산자로 표현된다.

$$\mathbf{procP}_{com} \triangleq |||_{i=}^{n} \mathbf{procI}_{pi} \quad (20)$$

식 (20)에서 n은 컴포넌트 com이 가지고 있는 provide 인터페이스의 개수이고 $\mathbf{procI}_{pi}$는 인터페이스 pi의 프로세스로 식 (7)과 동일하다.

컴포넌트의 실행 시 프로세스는 컴포넌트와 provide 인터페이스의 parallel composition이다.

$$\mathbf{procRuntime}_{com} \triangleq \mathbf{proc'}_{com}\{|c:Chans(\mathbf{procIGPcom})|\}|||\{|c:Chans(\mathbf{procPcom})|\}\mathbf{procP}_{com} \phi \mathbf{procRuntime}_{com} \quad (21)$$

인터페이스, 인터페이스 그룹, 컴포넌트는 parallel operator를 통해 메시지를 주고 받는다. 각 오퍼레이션에 대응하는 입력 채널과 출력 채널을 인터페이스, 인터페이스 그룹, 컴포넌트가 공유한다. 컴포넌트는 인터페이스의 참조자를 가지고 있지 않기 때문에 인터페이스와 직접 메시지를 주고 받지 않는다. 식 (21)에 의해 메시지는 인터페이스에서 인터페이스 그룹으로 인터페이스 그룹에서 컴포넌트로 전달된다.

5.3 컴포넌트 프레임워크와 시스템의 의미론

프레임워크는 다음과 같이 정의된다.

FrameworkName	
$g_1 : IG_1; g_2 : IG_2, \ldots; g_n : IG_n$	[인터페이스 그룹]
$P \triangleq PE$	[프로세스]

IG_i는 CSP_Z의 *PROCESS* 타입이며 따라서 g_i는 *PROCESS* 타입의 한 변수이다. g_i와 P의 의미는 방정식 (11)에 의해 정의된다.

$$\mathbf{proc}_{FW} \mid P \quad (22)$$

시스템은 인터페이스 그룹이 컴포넌트에 의해 대치된다는 것만 제외하고는 프레임워크와 의미가동일하다. 이것은 컴포넌트가 인터페이스 그룹을 상속하기 때문에 가능하다. 또한 가변치는 가변점에 할당된다. 시스템은 다음과 같이 정의된다.

$c_1 : COMP_1; c_2 : COMP_2, \ldots, c_n : COM_n$	[컴포넌트]
f : FW	[프레임워크]
b: bindingProcess	[가변치를 가변점에 할당하는 프로세스]
semantic	

시스템 스키마의 술어(Predicate)는 배포시에 적용되며 따라서 가변치는 배포시에 가변점에 할당된다. 가변치가 가변점에 할당되는 프로세스는 컴포넌트 실행 프로세스와 유사하다.

$$\mathbf{procI}_{ri} \triangleq \Box \pi \, op : \mathbf{ops}_{ri} \cdot \mathbf{procO}_{ri}(op) \quad (23)$$

$$\mathbf{procC}_{com} \triangleq |||_{i=}^{r} \mathbf{procI}_{ri} \quad (24)$$

$$\mathbf{proc''}_{com} \triangleq \mathbf{procIGC}_{com}$$

$[\{x \varepsilon (\{|c:Chans(\mathbf{procIGC}_{com})|\}\text{-}uses\{|c:Chans(\mathbf{procIGC}_{com})|\})\ x \backslash \mathbf{ref}(x)\}]\mathbf{procD}_{com}$ (25)

$\mathbf{procDeploytime}_{com} \triangleq \mathbf{proc''com}_{\{|c:Chans(procIGCcom)|\}} \|_{\{|c:Chans(procCcom)|\}} \mathbf{procC}_{com}$ (26)

$\mathbf{proc'}_{sys} \triangleq [semantic] \cdot (\mathbf{procDeploytime}_{c1} \|\| \mathbf{procDeploytime}_{c2} \|\|, ..., \mathbf{procDeploytime}_{cn}) \| b$ (27)

$\mathbf{proc}_{sys} \triangleq proc'_{sys} \rightarrow \mathbf{procf}(c_1/ig_1, c_2/ig_2, ..., c_n/ig_n)$ (28)

방정식 (27)은 semantics에서 정의된 가변치가 각 컴포넌트에 할당된다는 것을 의미한다. Customization 인터페이스의 오퍼레이션을 사용하여 가변치를 가변점에 할당하면 가변치는 식 (26)에 의해 시스템의 각 컴포넌트에 할당된다. 방정식 (28)은 할당 프로세스가 완료된 후에 프레임워크 프로세스가 실행된다는 것을 의미한다. 각 인터페이스 그룹을 대표하는 프로세스는 방정식 (21)에서 정의된 컴포넌트 프로세스로 대치된다.

5.4 의미론을 사용한 명세 검증

Component-Z는 명세의 논리 오류를 찾을 수 있을 뿐 아니라 deadlock과 livelock도 찾을 수 있다.

첫째, 인터페이스의 각 오퍼레이션에 대한 논리 오류는 집합과 함수를 이용한 논리 전개를 사용하여 검증할 수 있다. 즉Z 언어로 작성된 명세의 논리 오류 검증 방법과 동일한 방법을 사용한다.

둘째, 컴포넌트 내부 객체들 간의 deadlock과 livelock은 Object-Z의 History를 이용하여 시제논리로검증할 수 있다[26].

셋째, 컴포넌트 기반 시스템, 컴포넌트, 인터페이스 그룹, 프레임워크의 dealock과 livelock은 5.3 절의 공식을 사용하여 명세를 CSP 프로세스로 변환한 후, CSP의 failure, divergence에 대한 정의와 denotational semantics를 사용하여 검증한다.

넷째, 인터페이스와 컴포넌트 사이의 일관성은 Mapping 함수를 사용하여 인터페이스의 오퍼레이션을 변환한 후, 변환된 오퍼레이션의 선행 조건이 만족되면 대응되는 컴포넌트 오퍼레이션의 선행 조건도 만족되는지, 컴포넌트 오퍼레이션의 후행 조건이 만족되면 변환된 인터페이스 오퍼레이션의 후행 조건도 만족되는지를 검증하여 증명한다.

다섯째, 컴포넌트가 인터페이스 그룹을 상속할 때 오류가 없는지 검증하기 위해서는 인터페이스 그룹과 컴포넌트 사이에 refine 관계가 성립해야 한다. 이것은 History와 CSP 프로세스를 통합한 Failure 모델을 만들고 이 모델을 사용하여 refine 관계를 정의하는 것에 의해 가능하다[27].

여섯째, 가변치가 할당되었을 때의 오류 검증은 컴포넌트에 가변치를 할당 한 후 첫 번째에서 다섯 번째까지의 검증을 수행하면 된다.

6. 사례 연구

Component-Z의 표현력을 보이기 위해 은행 입출금 업무에 대한 정형 명세를 제시한다.

은행 입출금 업무의 요구사항은 다음과 같다.

은행 시스템은 입금, 출금, 이체, 계좌 개설에 대한 서비스를 제공해야 한다. 이 서비스는 계좌 관리라고 불린다. 계좌 관리의 명세는 'AccountManager' 인터페이스를 통해 정의된다. 'AccountManager' 인터페이스는 'openAccount', 'deposit', 'withdraw', 'transfer'로 구성된다. 또한 고객 정보의 검색과 등록, 고객의 사고에 대한 검색과 등록이 가능해야 한다. 고객 관리의 명세는 'CustomerManager' 인터페이스를 통해 정의된다.

'AccountManager' 컴포넌트에는 세 개의 가변점이 있다. 첫째, 이자를 계산하는 논리가 가변적이다. 둘째, 고객이 새로운 계좌를 생성했을 때 계좌 개설이 가능한지를 점검하는 논리도 가변적이다. 셋째, 이체 시 수수료 계산 방식이 가변적이다.

계좌 생성 점검의 첫 번째 가변치는 고객의 연체 횟수가 많거나 신용이 불량이면 계좌 개설을 허락하지 않는다. 두 번째 가변치는 사용자가 사고가 많으면 계좌 개설을 허용하지 않는다.

고객이 이체 할 때 수수료는 두 가지 방법으로 계산된다. 첫 번째 경우는 VIP 고객을 제외한 모든 고객의 수수료를 계산한다. 두 번째 경우는 각 고객의 등급에 따라 수수료 계산 알고리즘이 다르다.

'AccountManager'의 가변치를 설정하는 customization 인터페이스는 '**ICus**Customization'이다.

'AccountManager' 컴포넌트는 CustomerManager 인터페이스를 호출하여 고객 정보를 검색, 생성하고 고객이 발생 시킨 사로도 검색, 생성한다.

그림 13은 은행 시스템에 대한 클래스 도이다.

6.1 기본 타입

이 절에서는 은행 시스템을 명세하기 위해서 필요한 기본 타입을 정의한다.

다음 식은 컴포넌트 명세에서 사용할 기본 타입이다.

$[ACCOUNTID, CUSTOMERID, NAME, ACCIDENTID]$

$GRADE ::= VIP | SPECIAL | GENERAL | BAD$

$ACCIDENT\_TYPE ::= LOSS \mid BAD\_CREDIT \mid BLACKLIST$

$CustomerInfo \triangleq [id : CUSTOMERID\ name : NAME ; grade : GRADE]$

$AccidentInfo \triangleq$

$[accidentId : ACCIDENTID ; customerId : CUSTOMERID : accidentType : ACCIDENT\_TYPE]$

다음 식은 조회 오퍼레이션에서 반환 되는 정보를 담

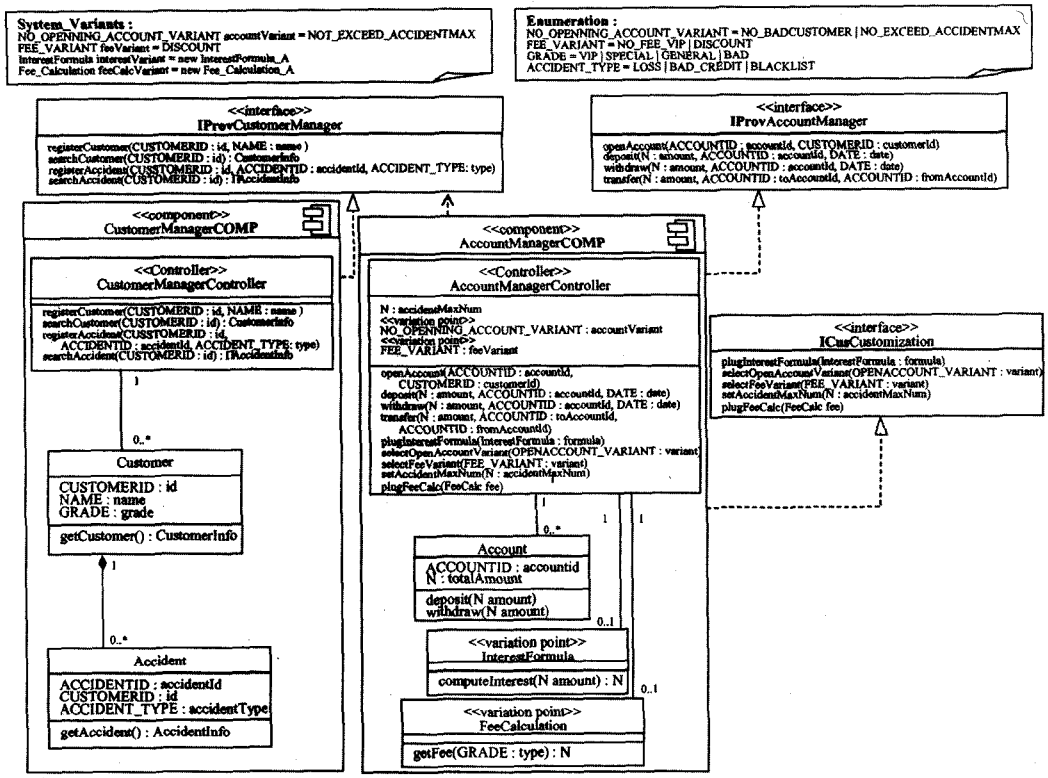


그림 13 은행 계좌 관리 클래스 도

고 있는 스키마이다.

CustomerInfo≙[*id* : *CUSTOMERID*; *grade* : *GRADE*]

AccidentInfo≙

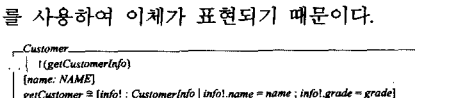
[*accidentId* : *ACCIDENTID*; *customerId*: *CUSTOMERID*; *accidentType* : *ACCIDENT_TYPE*]

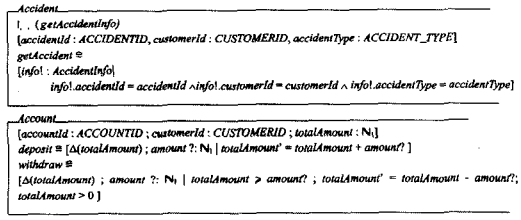
AccountInfo≙[*accountId* : *ACCOUNTID* ; *customerId* : *CUSTOMERID* ; *amount* : ℕ]

ACCOUNTID, *CUSTOMERID*, *ACCIDENTID*는 계정, 고객, 사고의 식별자이다. *GRADE*는 고객의 등급을 가리킨다. *CustomerInfo*와 *AccidentInfo*는 고객과 사고 조회 시 결과값이다.

6.2 클래스

컴포넌트 설계의 첫 번째 단계는 컴포넌트 내부에 들어갈 클래스를 설계하는 것이다. 클래스 설계는 Object-Z의 표기법을 사용하여 명세할 수 있다. 'Account' 클래스에서 'transfer' operation은 생략되었는데 이유는 'transfer' 오퍼레이션이 없어도 'deposit'과 'withdraw'를 사용하여 이체가 표현되기 때문이다.





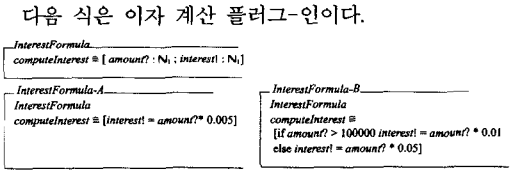
6.3 가변치

클래스 설계가 완성되면 보조적인 역할을 하는 플러그-인 클래스를 설계한다. 플러그-인 클래스는 Object-Z로 명세하고 설정 가능한 가변치는 enumeration으로 명세한다. 다음 식은 컴포넌트에서 설정 가능한 가변치를 보여준다.

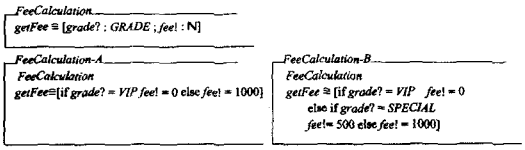
NO_OPENNING_ACCOUNT_VARIANT::=*NOT_BAD CUSTOMER* | *NOT_EXCEED_ACCIDENTMAX*

FEE_VARAINT ::= *NO_FEE_VIP* | *DISCOUNT*

다음 식은 이자 계산 플러그-인이다.



다음 식은 수수료 계산 플러그-인이다.



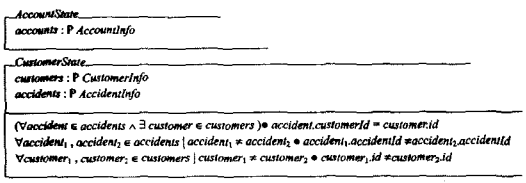
*NO_OPENING_ACCOUNT_VARIANT*는 계좌 개설 시 조건을 점검하는 두 개의 논리 가변성이 존재한다는 것을 보여준다. *NOT_BADCUSTOMER*는 'Account-Manager' 컴포넌트가 *GRADE*가 *BAD*인 고객은 계좌를 개설할 수 없도록 한다. *NOT_EXCEED_ACCIDENTMAX*는 계좌 개설 시 고객의 사고 회수가 어떤 값을 넘으면 계좌 개설을 허용하지 않는다.

*FEE_VARIANT*는 수수료를 계산하는 두 개의 가변성을 보여준다. *NO_FEE_VIP*는 VIP를 제외하고 모든 고객에게 동일한 방법으로 수수료를 부과하도록 한다. *DISCOUNT*는 *FeeCalculation*을 사용하여 수수료를 계산한다. *FeeCalculation*의 하위 클래스가 배포 시 컴포넌트에 삽입된다.

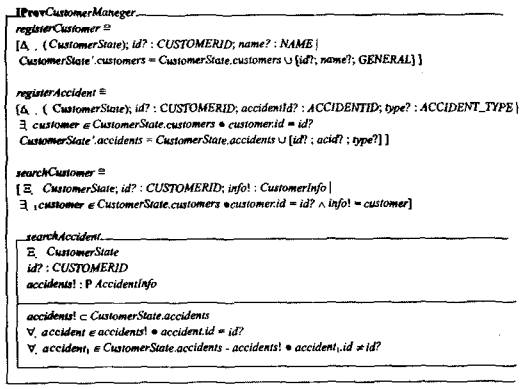
6.4 인터페이스

인터페이스는 Z 언어의 표기법을 사용하여 상태 스키마와 오퍼레이션 스키마를 명세한다.

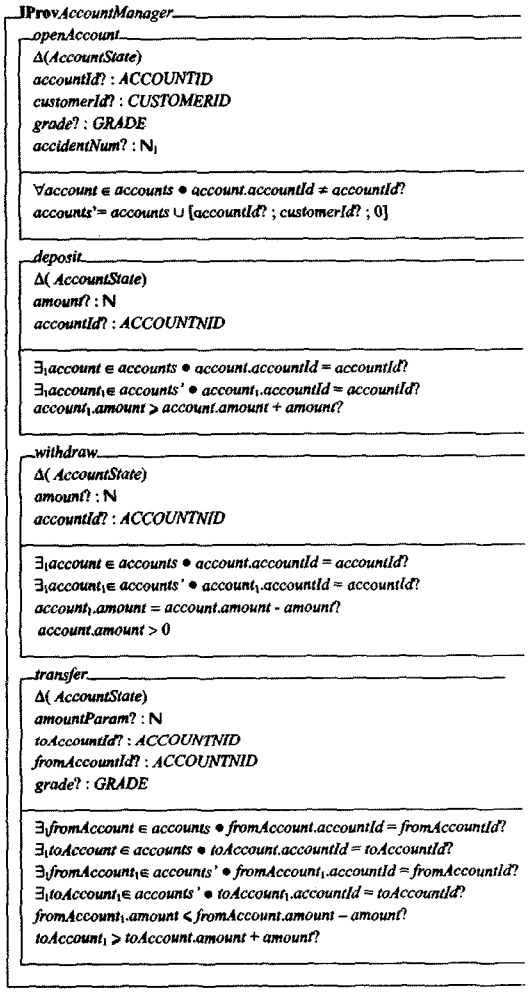
다음 식은 인터페이스에서사용될 abstract state schema이다. Abstract state schema *AccountState*는 **IProv***AccountManager* 인터페이스의 상태 변화를 표현하기 위해 사용된다. *CustomerState*는 **IProv** *CustomerManager* 인터페이스의 상태 변화를 표현하기 위해 사용된다.



다음 식은 고객 관리 인터페이스이다.



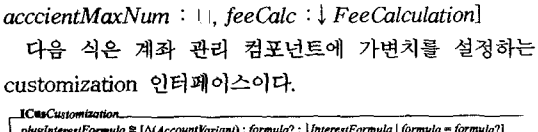
다음 식은 계좌 관리 인터페이스이다.



다음 식은 **ICus***Customization*에서 사용될 abstract state schema이다. *AccountVariant*는 **ICus***Customization* 인터페이스의 상태 변화를 표현하기 위해 사용된다.

AccountVariant ≙
[*formula* : ↓ *InterestFormula, account Variant* : *OPENACCOUNT_ VARIANT, feeVariant* : *FEE_VARIANT acccientMaxNum* : ℕ, *feeCalc* : ↓ *FeeCalculation*]

다음 식은 계좌 관리 컴포넌트에 가변치를 설정하는 customization 인터페이스이다.



인터페이스 명세는 5.1 절의 의미론을 사용하여 다음과 같이 CSP 프로세스로 변환된다.

식 (3)에 의해서 **IProv***AccountManager*의 오퍼레이션의 집합은 다음과 같이 표현된다.

$$\mathbf{ops}_{\mathbf{IProv}AccountManager} = \{openAccount,\ deposit,\ withdraw,\ transfer\} \quad (29)$$

식 (4)에 의해 **IProv***AccountManager*의 *deposit* 오퍼레이션의 입력 스키마는 다음과 같이 표현된다.

$$\mathbf{input}_{\mathbf{IProv}AccountManager}(deposit) \mid [self : \mathbf{IProv}AccountManager;\ amount?: \mathbb{N};\ accountId? : ACCOUNTNID] \quad (30)$$

식 (30)에서 self는 인터페이스에 대응되는 참조 변수로 인터페이스마다 값이 다르도록 정수 값으로 정의된다.

식 (6)에 의해 오퍼레이션 deposit에 대응하는 CSP 프로세스는 다음과 같다.

$$\mathbf{proO}_{\mathbf{IProv}AccountManager}(deposit) \triangleq deposit_b?\ \mathbf{input}_{\mathbf{IProv}AccountManager}(deposit) \rightarrow \mathbf{proO}_{\mathbf{IProv}AccountManager} \quad (31)$$

IProv*AccountManager*의 다른 오퍼레이션에 대응하는 CSP 프로세스는 생략한다. 식 (7)에 의해 **IProv***AccountManager*에 대응하는 CSP 프로세스는 다음과 같이 표현된다.

$$\mathbf{procI}_{\mathbf{IProv}AccountManager} \triangleq \Box op{:}\mathbf{ops}_{\mathbf{IProv}AccountManager} \bullet \mathbf{proO}_{\mathbf{IProv}AccountManager}(op) \quad (32)$$

IProv*CustomerManager*, **ICus***Customization*의 의미론은 생략한다.

6.5 인터페이스 그룹

인터페이스가 컴포넌트의기능성을 나타낸다면 인터페이스 그룹은 컴포넌트의 동적인 면을 나타낸다. 컴포넌트 설계에서 컴포넌트의 동적인 면은 보통 UML의 순차도를 사용하여 표현한다. Component-Z에서는 CSP를 사용하여 컴포넌트의 동적인 면을 표현한다.

```
AccountManagerIG
p
account : IProvAccountManager
r
customer: IProvCustomerManager
c
customizer: ICusCustomization

//openAccountP 프로세스는 실행하기 전 Customer 인터페이스를 사용하여 고객의 신용등급과
사고 횟수를 조회한 후 계좌 개설을 수행한다.
openAccountP ≙ customer.searchCustomer?id!customerInfo→
  if id = NULL customer.registerCustomer ?customerId→
    account.openAccount!accountId?customerId!GENERAL?0→openAccountP
    else customer.searchAccidents!accidents →
    account.openAccount!accountId?customerInfo.grade?!accidents→openAccountP

depositP ≙ account.deposit→ depositP

withdrawP ≙ account.withdraw → withdrawP

//이체시 고객의 등급을 파악하기 위해 고객 검색을 수행해야 한다. 고객 등급을 파악하는
이유는 고객 등급에 따라 수수료가 달라지기 때문이다.
transferP ≙ account.transfer$_b$→customer.searchCustomer!customer!info→
  transfer!info.grade→ account.transfer$_a$→ transferP

// customize, customize$_1$, customize$_2$는 가변치를 설정하는 프로세스이다. 이 프로세스는 여러 개
의 가변치가 충돌을 일으키지 않도록 설정 순서를 정한다.
customize ≙ customizer.plugInterestFormula→ customizer.selectFeeVariant?variant→customize$_1$

customize$_1$ ≙ if variant = DISCOUNT then customizer.plugFeeCalc→customize$_2$ else customize$_2$

customize$_2$ ≙ customizer.selectOpenAccountVariant?variant→
  if variant = NOT_EXCEED_ACCIDENTMAX then customizer.setAccidentMaxNum→ STOP
  else STOP
```

```
CustomerManagerIG
p
customer : ICustomerManager
registerCustomer ≙ customer.registerCustomer → registerCustomer
registerAccident ≙ customer.registerAccident → registerAccident
searchCustomer ≙ customer.searchCustomer → searchCustomer
searchAccident ≙ customer.searchAccident → searchAccident
```

위의 인터페이스 그룹 명세를 보면 인터페이스 그룹 명세는 인터페이스를 참조할 뿐 인터페이스 명세와 통합되어 있지 않다. Object-Z를 사용하는 경우 오퍼레이션 명세와 오퍼레이션 사이의 메시지 흐름을 표현하는 명세는 분리하기 어렵다. Component-Z는 기능적인 명세와 동적 명세를 명확히 분리한다. 인터페이스를 통한 기능적인 명세는 사용자와 컴포넌트 간의 계약이기 때문에 쉽게 바꿀 수 없지만 동적 명세는 설계가 진행되고, 유지보수가 발생할 때 쉽게 변경된다. 따라서 Component-Z를 사용한 기능 명세와 동적 명세의 분리는 컴포넌트 설계에 유용하다.

위의 명세를 보면 복잡한 오퍼레이션 호출 순서가 CSP 표기법을 사용하여 간결하게 표현되었음을 알 수 있다.

*AccountManager***IG**는 일종의 클래스이다. 이 클래스는 세 개의 속성을 갖는다- **IProv***Account-Manager*의 참조자, **IProv***CustomerManager*의 참조자, **ICus***Customization*의 참조자. 또한 이 클래스는 다섯 개의 프로세스를 갖는다. 각 프로세스에서 입출력 값이 다음 프로세스로 넘어가는 경우를 제외하고는 프로세스에 들어오는 입출력 값을 생략한다.

*AccountManager***IG**의 *depositP* 프로세스는 식 (8), (9), (10)에 의해서 다음과 같이 정의된다.

$$\mathbf{proO}_{\mathbf{IProv}AccountManager}(account.deposit) \triangleq deposit_b?[account : \mathbf{IProv}AccountManager\ ;\ amount? : \mathbb{N},\ accountId? : ACCOUNTNID] \quad (33)$$

$$depositP \triangleq \mathbf{proO}_{\mathbf{IProv}AccountManager}(account.deposit) \rightarrow depositP \quad (34)$$

*AccountManager***IG** 중 provide 인터페이스의 실행에 대응하는 CSP 프로세스는 식 (11)에 의해서 다음과 같이 정의된다.

$$\mathbf{procIGP}_{AccountManagerIG} \triangleq openAccountP \,|||\, depositP \,|||\, withdrawP \,|||\, transferP \rightarrow \mathbf{procIGP}_{AccountManagerIG} \quad (35)$$

*AccountManager***IG** 중 customization 인터페이스의 실행에 대응하는 CSP 프로세스와 *CustomerManager*-**IG**에 대응하는 프로세스는 생략한다.

6.6 컴포넌트

이 절에서는 '*CustomerManager*' 컴포넌트와 '*Account-Manager*' 컴포넌트의 명세를 제시한다.

```
CustomerManagerCOMP
CustomerManager1G
[customers : ℙ Customer, accidents : ℙ Accident]

Mapping≙[map : CustomerInfo ⤖ Customer, map1 : AccidentInfo ⤖ Accident |
  dom map = {CustomerState.customers} ∧ ran map = {customers}
  ∧dom map1 = {CustomerState.accidents} ∧ ran map1 = {accidents}
  ∧∀customer ∈ store.setOfCustomer • map(customer).id = customer.id
  ∧ map(customer).name = customer.name ∧ map(customer).grade = customer.grade
  ∧∀accident ∈ accidents • map1(accident).accidentId = accident.accidentId
  ∧map1(accident).customerId=accident.customerId
  ∧map1(accident).accidentType = accident.accidentType]

INIT
  customers = ∅ ∧ accidents = ∅

registerCustomer≙[Δ ( customers), id? : CUSTOMERID, name? : NAME |
  ∀customer ∈ customers • customer.id ≠ id? ∧ let customer : Customer • customer.id = id?
   ∧ customer.name = name? ∧ customer.grade = GENERAL ∧ customers' = customers ∪ customer]
registerAccident
≙[Δ ( accidents), id? : CUSTOMERID, accId? : ACCIDENTID, type? : ACCIDENT_TYPE |
  ∃ customer ∈ customers • customer.id = id? ∧∀ accident ∈ accidents • accident.accId ≠ accId? ∧
  let accident1 : Accident•accident1.customerId = id?∧accident1.accId = accId?∧accident1.type =type?
  ∧ accidents' = accidents ∪ accident1]

searchCustomer ≙[id? : CUSTOMERID, info! : CustomerInfo |
  ∃ customer ∈ customers •customer.id = id?∧ info! = customer]
searchAccident≙[id? : CUSTOMERID, infoSet! : ℙ AccidentInfo | infoSet! ⊂ dacidents ∧
  ∀accident ∈ infoSet • accident.accId = id?∧ ∀accident1 ∈ accidents - infoSet • accident1.accId ≠ id? ]
```

```
AccountManagerCOMP
//AccountManagerCOMP는 AccountManager1G를 상속한다.
AccountManager1G                                          //상속된 인터페이스 그룹

feeAmount : ℕ                                             //지역 변수

feeAmount = 1000

//cvFormula, cvAccountVariant, cvFeeVariant, cvAccidentMaxNum, cvFeeCalc는 가변점이다.
[accounts : ℙ Account ⇸ cvFormula : ↓InterestFormula ;
cvAccountVariant : NO_OPENING_ACCOUNT_VARIANT ;
 cvFeeVariant : FEE_VARIANT ; cvAccidentMaxNum : ℕ ; cvFeeCalc : ↓FeeCalculation]  //상태 스키마

//Mapping은 인터페이스 상태 스키마를 컴포넌트 상태 스키마로 매핑한다.
Mapping                                                   //매핑 스키마
 map : AccountInfo ⤖ Account
 map1 : AccountVariant ⤖ ↓InterestFormula×NO_OPENING_ACCOUNT_VARIANT× FEE_VARIANT
        ×ℕ×↓FeeCalculation

 rm map = accounts
 ∀ account ∈ dom map • map(account).accountId = account.accountId
  map(account).totalAmount = account.totalAmount
 rm map1 = <cvFormula, cvAccountVariant, cvFeeVariant, cvAccidentMaxNum, cvFeeCalc>

INIT
  accounts = ∅

//아래 식은 모두 마이크로 워크플로우임

//openAccount에서는 인터페이스 그룹을 통해 입력 받은 고객의 신용 등급, 사고 횟수, 최대
사고 횟수, 계좌 개설 가변치를 사용하여 계좌 개설 여부를 결정한다.
 openAccount                                              //계좌 개설
  Δ(accounts)
  accountId? : ACCOUNTID
  customerId? : CUTOMERID
  grade? : GRADE
  accidentNum? : N1

  ∃1account : Account • account.accountId = accountId? ; accounts' = accounts ∪ account
  if cvAccountVariant = NOT_BADCUSTOMER grade? = BAD
  if cvAccountVariant = NOT_EXCEED_ACCIDENTMAX accidentNum? < cvAccidentMaxNum
```

```
 checkAccountID ≙                                         //계좌 ID 검사
[accountId? : ACCOUNTID; amount? : N | ∃1account ∈ accounts • account.accountId = accountId?]
checkAccountID1≙ [fromAccountId ?: ACCOUNTID ; toAccountID? : ACCOUNTID ; amountParam? : N ;
  grade? : GRADE ; feeTemp : N ; fromAccount : Account ; toAccount : Account |
  fromAccount ∈ accounts • fromAccount.accountId = fromAccountId? ;
  toAccount ∈ accounts • toAccount.accountId = toAccountId? ]

//계좌에 금액이 올바른지 검사함
checkAmount ≙ [∃1account ∈ accounts • account.accountId = accountId? ; account.amount > amount?]
checkAmount1≙[fromAccount.totalAmount < amount? + feeTemp]

//입금
deposit1 ≙[Δ(accounts) ;dAccount : accounts | dAccount.accountId = accountId?]
•dAccount.deposit(amount?)
deposit ≙checkAccountID • (deposit1 ∧ (cvFormula.computeInterest ⨾deposit1[ amount?/ interest!] ))

//출금
withdraw1 ≙ [Δ(accounts); wAccount : accounts | wAccount.accountId = accountId?]
• wAccount.withdraw(amount?)
withdraw ≙ checkAccountID • checkAmount •withdraw1

//수수료 계산
calculateFee1≙[cvFeeVariant = NO_FEE_VIP ∧ grade? = VIP]•[feeTemp=0]
calculateFee2≙[cvFeeVariant = NO_FEE_VIP ∧ grade? ≠ VIP]•[feeTemp=feeAmount]
calculateFee3≙[cvFeeVariant = DISCOUNT] •cvFeeCalc.getFee![feeTemp=fee!]

//이체
//'transfer' operation은 워크플로우 가변성을 보여준다. 'transfer' 오퍼레이션에는 두 개의 워크
플로우가 존재한다. 두 개의 워크플로우 중 하나를 선택하는 조건은 scope enrichment를 사용
하여 표현한다.
 transfer1≙[fromAccount.withdraw[amountParam?/amount?] ∧ toAccount.deposit[amountParam?/amount?]
  ∧ formula.computeInterest ∧ toAccount.deposit[interest!/amount?]
transfer2≙[fromAccount.withdraw[amountParam?+feeTemp /amount?] ∧
  toAccount.deposit[amountParam?/amount?]
transfer•checkAccountID1•( (calculateFee1[calculateFee2][calculateFee3) ∧
  ([cvFeeVariant = NO_FEE_VIP ∧ grade? = VIP] •transfer1
  ▯[cvFeeVariant = NO_FEE_VIP ∧ grade? ≠ VIP ∨ cvFeeVariant = DISCOUNT] •transfer2)∧
  (formula.computeInterest ⨾ toAccount.deposit[interest!/amount?] ) )

//이자 계산, 수수료 계산 plugin을 삽입하는 워크플로우
//plugInterestFormula는 Customization 인터페이스의 오퍼레이션에 대응하는 컴포넌트 오퍼레이
션으로 이자 계산 객체를 컴포넌트에 plug-in한다. plugFeeCalc도 수수료 계산 객체를 플러그-
인한다.
plugInterestFormula ≙[Δ(cvFormula) ; formula? : ↓InterestFormula| cvFormula = formula?] '
plugFeeCalc ≙ [Δ( cvFeeCalc) ; variant? : ↓FeeCalculation | cvFeeCalc = variant?]

//계좌 개설시 가변치를 설정함
//selectOpenAccountVariant, selecFeeVariant는 가변치 중의 한 값을 컴포넌트에 셋팅한다.
selectOpenAccountVariant ≙ [Δ( cvAccountVariant) ; variant? : NO_OPENING_ACCOUNT_VARIANT |
  cvAccountVariant = variant?]
//수수료 계산 가변치를 설정함
selectFeeVariant ≙ [Δ(cvFeeVariant) ; variant? : FEE_VARIANT | cvFeeVariant = variant?]
//최대 사고수를 설정함
setAccidentMaxNum ≙ [Δ( cvAcccientMaxNum) ; variant? : ℕ | cvAcccientMaxNum = variant?]
```

인터페이스를 사용한 기능 설계와 컴포넌트의 외부 동적인 면에 대한 설계가 완료된 후에는 컴포넌트 내부를 설계해야 한다. 유지 보수 시 컴포넌트가 대치될 수 있으므로 컴포넌트 내부 설계는 인터페이스, 컴포넌트 외부의 동적인 면과 분리되어야 한다. Component-Z는 이 사실을 반영한다.

위의 명세를 보면 컴포넌트 내부의 논리와 구조가 Object-Z를 사용하여 쉽게 표현됨을 알 수 있다. 컴포넌트 내부는 자료 구조를 표현해야 하기 때문에 Object-Z를 사용하는 것이 유용하다.

인터페이스 그룹에 있는 Provided 인터페이스와 customization 인터페이스의 오퍼레이션은 이 인터페이스 그룹을 상속한 컴포넌트의 오퍼레이션에 대응된다. 따라서 'AccountManager' 인터페이스 그룹에서 나타나는 모든 오퍼레이션 이름은 'AccountManager' 컴포넌트에서도 나타나야 한다.

Customization 인터페이스에 대응하는 오퍼레이션이 모두 호출되지 않으면 컴포넌트는 실행될 수 없다. 인터페이스 그룹의 가변성 설정 프로세스는 customization 인터페이스의 오퍼레이션이 충돌을 일으키지 않고 호출되는 것을 보장한다.

매핑 스키마의 두 함수 *map*과 *map*1을 통해 인터페이스와 컴포넌트 사이에 충돌이 발생하지 않는지 점검할 수 있다.

식 (13)에 의해 *AccountManager*COMP의 상태 스키마는 다음과 같이 정의된다.

$$\mathbf{state}_{AccountManagerCOMP} \triangleq [feeAmount : \mathbb{N}] \bullet [accounts : \mathbb{F}\ Account \Rightarrow cvFormula : \downarrow InterestFormula ;\ cvAccountVariant : NO\_OPENING\_ACCOUNT\_VARIANT ;\ cvFeeVariant : FEE\_VARIANT ;\ cvAcccientMaxNum : \mathbb{N};\ cvFeeCalc : \downarrow FeeCalculation] \quad (36)$$

식 (36)에서 *accounts*는 객체 인스턴스의 집합이고, *cvFormula*는 객체 인스턴스이다. 객체 인스턴스는 Object-Z의 history 기반 의미론에 의해 정의된다. 예를 들어 *Account* 클래스의 인스턴스는 *accountId* : *ACCOUNTID*, *customerId* : *CUSTOMERID*, *total Amount* : $\mathbb{N}_1$에서 accountId, customerId, totalAmount가 가질 수 있는 모든 가능한 값들의 집합으로 매핑하는 *State* 함수와 오퍼레이션 deposit, withdraw의 매개변수에 매개변수가 가질 수 있는 모든 값들을 매핑하는 함수에 오퍼레이션의 이름을 순서쌍으로 갖는 Event로 정의된다.

예를 들어 State와 Event는 다음과 같은 값을 가질 수 있다.

$$State = \{\{\text{'}accountId\text{'} \mapsto 100\}, \{\text{'}customerId\text{'} \mapsto 200\}, \{\text{'}totlaAmount\text{'} \mapsto 1000\}, ...\} \quad (37)$$

$$Event = \{\{\text{'}deposit\text{'}, \{\text{'}amount?\text{'} \mapsto 100\}\}, \{\text{'}withdraw\text{'}, \{\text{'}amount?\text{'} \mapsto 100\}\}, ...\} \quad (38)$$

Account 클래스에 대응되는 History 스키마는 다음과 같이 정의된다.

AccountHistory

$states : \mathbf{seq}_{\infty} State$

$events : \mathbf{seq}_{\infty} Event$

$states \neq \langle\rangle$

$\forall i, j : \mathbf{dom}\ states \bullet \mathbf{dom}\ states(i) = \mathbf{dom}\ states(j)$

$\forall i : \mathbb{N}_1 \bullet i \in \mathbf{dom}\ events \Leftrightarrow i+1 \in \mathbf{dom}\ states$

식 (36)의 Π *Account*는 Π AccountHistory와 동일하다. *AccountManager***COMP**의 *openAccount*를 제외한 모든 오퍼레이션들은 각 객체 인스턴스에 대응하는 history를 사용하여 한 오퍼레이션 스키마로 변환될 수 있다. 따라서 *AccountManager***COMP**는 history를 사용하여 단일 객체의 클래스 스키마로 변환된다. 즉 복합 객체의 의미론은 history를 사용하여 단일 객체의 의미론으로 변환된다.

식 (15)에 의해 *AccountManager***COMP**의 오퍼레이션의 집합은 다음과 같이 정의된다.

$$\mathbf{ops}_{AccountManagerCOMP} = \{\text{'openAccount'}, \text{'}checkAccountID\text{'}, \text{'}checkAmount\text{'}, \text{'}checkAmount_1\text{'}, \text{'}deposit_1\text{'}, \text{'}deposit\text{'}, \text{'}withdraw_1\text{'}, \text{'}withdraw\text{'}, \text{'}calculateFee_1\text{'}, \text{'}calculateFee_2\text{'}, \text{'}calculateFee_3\text{'}, \text{'}transfer_1\text{'}, \text{'}transfer_2\text{'}, \text{'}transfer\text{'}, \text{'}plugInterestFormula\text{'}, \text{'}plugFeeCalc\text{'}, \text{'}selectOpenAccountVariant\text{'}, \text{'}selectFeeVariant\text{'}, \text{'}setAccidentMaxNum\text{'}\} \quad (39)$$

*AccountManager***COMP**의 각 오퍼레이션에 대응되는 CSP 프로세스는 식 (16), 식 (17)에 의해서 정의된다. 예를 들어 deposit 오퍼레이션에 대응되는 프로세스는 다음과 같이 정의된다.

$$\mathbf{procO}_{AccountManagerCOMP}(deposit, \mathbf{state}_{AccountManagerCOMP}) \triangleq deposit?[accountId?:ACCOUNTID; dAccount:accounts; amount?:\mathbb{N}_1] \rightarrow \mathbf{schema}(deposit) \rightarrow \mathbf{procS}_{AccountManagerCOMP}(\theta\ \mathbf{state}_{AccountManagerCOMP}) \quad (40)$$

$$\mathbf{procS}_{AccountManagerCOMP}(\mathbf{state}_{AccountManagerCOMP}) \triangleq \Box op : \mathbf{ops}_{AccountManagerCOMP} \cdot \mathbf{procO}_{AccountManagerCOMP}(op, \theta\ \mathbf{state}_{AccountManagerCOMP}) \quad (41)$$

식 (40)에서 **schema**(deposit)은 deposit 오퍼레이션을 나타내는 Z 스키마이다. **schema**(deposit)의 구체적인 내용은 생략한다.

컴포넌트 내부에 대응되는 프로세스는 식 (14), (18)에 의해서 다음과 같이 정의된다.

$$\mathbf{init}_{AccountManagerCOMP} \triangleq [accounts = \varnothing] \quad (42)$$

$$\mathbf{procD}_{AccountManagerCOMP} \triangleq \mathbf{init}_{AccountManagerCOMP}\ \not\phi$$

$$\mathbf{procS}_{AccountManagerCOMP}(\theta\ \mathbf{state}_{AccountManagerCOMP}) \quad (43)$$

컴포넌트 내부와 인터페이스 그룹을 결합한 프로세스는 식 (19)에 의해서 다음과 같이 정의된다.

$$\mathbf{proc}'_{AccountManagerCOMP} \triangleq \mathbf{procIGP}_{AccountManagerCOMP} [\{x \varepsilon (\{|c: Chans(\mathbf{procIGP}_{AccountManagerCOMP})|\} - \mathbf{required}\ \{|c: Chans(\mathbf{procIGP}_{AccountManagerCOMP})|\}) \mapsto x \backslash \mathbf{ref}(x)\}]\ \mathbf{procD}_{AccountManagerCOMP} \quad (44)$$

식 (44)에서 Chans(**procIGP**$_{AccountManagerCOMP}$)은 *AccountManager***IG**가 참조하는 채널 중 provide 인터페이스가 메시지를 보내거나 받을 수 있는 채널이다. 따라서 다음 식이 성립한다.

$$Chans(\mathbf{procIGP}_{AccountManagerCOMP}) = \{register Customer_b, register Customer_e, search Customer_b, search Customer_e, search Accident_b, search Accident_e, open Accounte_b, open Account_e, deposit_b, deposit_e, withdraw_b, withdrawe, transfer_b, transfer_e\} \quad (45)$$

*AccountManager***COMP**의 실행 프로세스는 식 (20)과 식 (21)에 의해 정의된다.

$$\mathbf{procP}_{AccountManagerCOMP} \triangleq \mathbf{procI}_{IProv AccountManager} \quad (46)$$

$$\mathbf{procRuntime}_{com} \triangleq \mathbf{proc}'_{AccountManagerCOMP}(\{|c: Chans(\mathbf{procIGP}_{AccountManagerCOMP})|\} \parallel \{|c: Chans(\mathbf{procP}_{AccountManagerCOMP})|\})\ \mathbf{procP}_{AccountManagerCOMP} \rightarrow \mathbf{procRuntime}_{AccountManagerCOMP} \quad (47)$$

*AccountManager***COMP**의 실행 프로세스가 어떻게 수행되는가를 보기 위해 **IProv***AccountManger*의 *deposit* 오퍼레이션에 *amount*? = 1000, *accountId*? = 1234가 입력되고, **IProv***AccountManger*의 인스턴스 참조자가 44라는 값을 가질 경우 어떤 이벤트가 발생하는지를 계산한다. 먼저 식 (32)에 의해 $deposit_b$.44.1000.1234라는 이벤트가 발생한다. 식 (47)에 의해 $deposit_b$.1000.1234는 식 (44)의 프로세스에 전달된다. 식 (34)와 (35)에 의해 *AccountManager***IG**에 $deposit_b$.44.1000.1234라는 이벤트가 발생한다. 식 (44)에 의해 이 이벤트 중 인터페

이스 참조자 44가 생략되어 $depoist_b$.1000.1234가 컴포넌트 내부에 전달된다. 식 (43)에 의해 식 (41), (42)의 프로세스가 수행된다. 식 (41)에서 deposit 오퍼레이션은 *amount*? = 1000, *accountId*? = 1234을 입력받고 deposit 오퍼레이션의 선행 조건을 만족하는지 검사한 후 만족하면 식 (36)의 상태 스키마를 변경시킨다.

Mapping 스키마를 사용하여 인터페이스 오퍼레이션과 컴포넌트 오퍼레이션의 일관성을 검증하는 과정은 생략한다.

6.7 컴포넌트 프레임워크

컴포넌트 설계가 완료된후에는 컴포넌트를 조립하여 시스템을 만든다. 컴포넌트 기반 시스템은 컴포넌트의 단순한 조합이므로 컴포넌트 프레임워크를 명세하면 된다.

다음 스키마는 *AccountManager*IG와 *CustomerManager*IG를 결합한 컴포넌트 프레임워크를 나타낸다.

*Banking*FW ≙

[*account* : *AccountManager*IG ; *customer* : *Customer ManagerIG*;*main account* ICustomer|| ICustomer *customer*]

식 (22)에 의해 컴포넌트 프레임워크에 대응되는 프로세스는 다음과 같이 정의된다.

$$\mathbf{proc}_{BankingFW} \triangleq account_{ICustomer} ||_{ICustomer}\ customer \quad (48)$$

6.8 계좌 관리 시스템

컴포넌트를 조립한 후에는 컴포넌트가 배포된 상황을 표현해야 한다. 배포시에는 가변성이 설정되고 인터페이스와 인터페이스 그룹의 조건을 만족하는 컴포넌트 인스턴스가 실행된다. 이 상황은 Component-Z에서는 시스템 명세로 표현된다.

다음 식은 *AccountMangaer*COMP와 *CustomerManager*COMP를 결합한 계좌 관리 시스템의 명세이다.

bankingFramework : BankingFW
accountManager : AccountManagerCOMP
customerManager : CustomerManagerCOMP
formula : InterestFormula_A
feeCalc : FeeCalculation_A
main≙*accountManager.customize?formula* → *accountManager.customize?DISCOUNT*→
→*accountManager.customize?feeCalc*→
accountManager.customize?NOT_EXCEED_ACCIDENTMAX→*STOP*

bankingFramework.account = *accountManager*
bankingFramework.customer = *customerManager*

이 스키마에서는 가변치가 컴포넌트에 할당된다. 계좌 관리 시스템에 대응하는 프로세스를 정의하기 위해 식 (23)에서 식 (26)을 사용하여 배포시의 컴포넌트 프로세스를 정의해야 한다. 배포시의 컴포넌트 프로세스는 6.6절에서 정의한 실행 시의 컴포넌트 프로세스와유사하기 때문에 생략한다. 식 (27)에 의해 다음과 같은 프로세스가 정의된다.

$$\mathbf{proc}'_{sys} \triangleq [bankingFramework.account = accountManager \wedge bankingFramework.customer = customerManager] \cdot (\mathbf{procDeploytime}_{AccountManagerCOMP} ||| \mathbf{procDeploytime}_{CustomerManagerCOMP}) || main \quad (49)$$

계좌 관리 시스템에 대응되는 프로세스는 다음과 같이 정의된다.

$$\mathbf{proc}_{sys} \triangleq \mathbf{proc}'_{sys} \rightarrow \mathbf{proc}_{BankingFW}(accountManager/account,\ customerManager/customer) \quad (50)$$

식 (50)에서 프레임워크에 대응되는 프로세스의 각 인터페이스 그룹은 컴포넌트 인스턴스로 대치된다.

7. 결 론

본 논문에서는 컴포넌트를 명세하기 위한 새로운 정형 명세 언어를 제시하였다.

Component-Z는 Object-Z에 기반을 두고 있으며 Object-Z를 확장한다. Object-Z의 확장에는 메시지 순차도와 컴포넌트의 동적인 구성 요소가 포함된다. Object-Z를 확장하기 위해 본 논문에서는 Object-Z의 표기법에 CSP의 표기법을 추가한다. 이런 두 개 언어의 결합은 새로운 것은 아니다. Object-Z에서는 CSP 표기법과 유사한 오퍼레이션 연산자를 가지고 있다. 따라서 본 논문의 접근 방법은 컴포넌트를 명세하기 위해 Object-Z를 개선한다. Component-Z의 의미론에 대한 본 논문의 접근 방법은 Component-Z를 CSPZ로 번역하는 것이다. 이 과정에서 동적인 부분은 CSP로 표현되고 정적인 부분은 Object-Z로 표현된다.

Component-Z는 Object-Z와 CSP 표기법을 모두 사용하였다. 따라서 Component-Z는 Object-Z만을 사용하여 표현하기 어려운 상황도 쉽게 표현할 수 있다. 예를 들어 multi-thread와 동기화도 쉽게 표현할 수 있다. 반대로 CSP만으로는 표현하기 어려운 자료구조는 Object- Z를 사용하여 쉽게 표현할 수 있다.

Component-Z는 컴포넌트를 명세하기에 적절하다. Component-Z는 인터페이스를 컴포넌트와 분리할 뿐 아니라 컴포넌트의 동적인 부분을 정적인 부분과 분리한다. 이러한 분리는 컴포넌트의 중요한 특징이며 Component-Z는 컴포넌트의 특징을 문법에서 반영한다. 또한 Component-Z는product-line engineering의 개념을 반영한다. 가변치, 가변점이 Component-Z에서 명확하게 표현된다.

사례 연구에서는 Component-Z가 컴포넌트 설계의 전 과정에서 사용될 수 있다는 것을 보였다. 컴포넌트는 많은 요소들로 이루어져 있으며 이 요소들 사이의 관계가 복잡하다. 사례 연구는 Component-Z가 컴포넌트의 모든 요소-인터페이스, 컴포넌트, 컴포넌트의 동적인 부분과 정적인 부분, 가변성, 워크플로우-와 컴포넌트 기

표 1 컴포넌트 정형 명세 비교

	Componentware	Heisel, et al[7]	Plasil, et al[28]	Aguirre, et al[16]	Wright	Component-Z
인터페이스를 사용한 계약	집합을 사용하여 표현함.	Object-Z로 표현함	없음	대수를 사용하여 표현함	없음	Object-Z를 변형하여 표현함.
컴포넌트와 인터페이스의 분리	없음	없음	인터페이스와 컴포넌트를 별개의 명세로 표현함.	없음	Port와 Computation을 사용하여 분리함.	인터페이스 명세와 컴포넌트 내부 명세 분리.
동적인 부분에 명세	메시지 순열을 사용함.	오퍼레이션 연산자를 사용함.	메시지 순열을 사용함.	시제 논리를 사용하여 표현함.	CSP를 사용함.	CSP를 사용함.
컴포넌트 구조에 대한 명세	컴포넌트와 서브 컴포넌트의 관계를 명세함.	없음	컴포넌트와 서브 컴포넌트의 관계를 명세함.	미약함	없음	인터페이스, Interface Group, 컴포넌트 내부 명세를 나누어 표현
컴포넌트 결합	집합 기반의 논리로 표현함.	객체 참조자를 이용한 결합.	프로토콜 연산자를 사용함.	모피즘으로 표현함.	Connector를 사용하여 표현함.	병행 연산자를 사용함
컴포넌트 프레임워크	없음	없음	없음	없음	없음	컴포넌트 결합에 대한 명세를 사용함
컴포넌트 배포	없음	없음	없음	없음	Configuration 명세를 사용함.	컴포넌트 기반 시스템에서 컴포넌트 인스턴스와 가변치를 사용하여 표현함.
특화	없음	없음	없음		없음	Customization 인터페이스와 가변점을 사용하여 표현함.
컴포넌트 기반 시스템	집합을 사용하여 표현함	시스템을 Object-Z로 표현함	컴포넌스와 서브 컴포넌트의 관계를 명세함.		Configuration 명세를 사용함.	컴포넌트 프레임워크에 배포된 컴포넌트와 가변치를 사용하여 명세함.

반 시스템을 명세할 수 있으며 요소들 사이의 관계도 상속과 참조 등을 통해 분명하게 표현할 수 있다는 것을 보여준다.

Component-Z와 다른 대표적인 컴포넌트 정형 명세를 표 1에서 비교한다.

ComponentWare는 집합 기반의 명세로 인터페이스와 컴포넌트, 컴포넌트 기반 시스템에 대한 명제만을 표현한다. Heisel, et al[7]의 논문은 Object-Z를 사용하여 컴포넌트를 표현한다. 그러나 객체 참조자를 사용하여 컴포넌트를 표현하기 때문에 인터페이스와 컴포넌트가 분리되지 않는다. Plasil, et al[28]의 논문은 컴포넌트의 동적인 부분에 대해 명세한다. 이 논문의 명세는 컴포넌트의 구조, 시스템, 동적인 부분을 간결하게 표현하지만 인터페이스를 통한 계약은 표현이 어렵다. Aguirre, et al[16]은 시제 논리와 카테고리, 모피즘을 사용하여 컴포넌트를 표현한다. 그러나 인터페이스와 컴포넌트가 명확히 분리되지 않다. Wright는 대표적인 ADL이다. Wright를 사용하면 컴포넌트의 구조, 동적인 부분, 컴포넌트 기반 시스템까지 표현할 수 있지만 인터페이스를 사용한 계약은 표현할 수 없다. Component-Z는 컴포넌트의 중요한 특성을 모두 반영하고 있다는 점에서 다른 컴포넌트 정형 명세에 비해 표현력이 높으며 컴포넌트의 설계 전 과정에 사용 가능하다.

참 고 문 헌

[1] Szyperski, C., Component Softwarebeyond Object-Oriented Programming, pp. 21, Addison Wesely, 2002.

[2] Lders, F., Lau, K.K. and Ho. S.M., Building Reliable Component-Based Software Systems, pp. 2338, Artech House, 2002.

[3] D'Souza, D.F., and Wills, A.C., Objects, Components, and Frameworks with UML: The Catalysis Approach, pp. 91, Addison-Wesley, 1998.

[4] Cheesman, J. and Daniels, J., UML Components: A Simple Process for Specifying Component-Based Software, pp. 50, Addison-Wesley, 2000.

[5] Kreuz, D., "Formal Specification of CORBA Services using Object-Z," Second IEEE International Conference on Formal Engineering Methods, September, 1998.

[6] Huaikou, M., Chuanjiang, Y. and Li, L., "A Formalized Abstract Component Object Model Z-COM," 36th International Conference on Technology of Object-Oriented Languages and Sys-

tems (TOOLS-Asia'00), November, 2000.
[7] Heisel, M., Santen, T. and Souquie'res, J., "Toward a Formal Model of Software Components," International Conference on Formal Engineering Methods(ICFEM) 2002, pp. 57-68, 2002.
[8] Bergner, K., Rausch, A., Sihling, M., Vilbig, A. and Broy. M., Foundations of Component-based Systems, pp. 189-210, Cambridge University Press, 2000.
[9] Medvidovic, N. and Taylor, R.N., "A Classification and Comparison Framework for Software Architecture Description Languages," IEEE Transaction on Software Engineering, Vol.26, No.1, 2000.
[10] Sousa, P.J. and Garlan. D., "Formal Modeling of the Enterprise JavaBeans Component Integration Framework," Information and Software Technology, Vol. 43, No.3, 2001.
[11] Canal, C., Fuentes, L., Pimentel, E., Troya, J.M. and Vallecillo, A., "Adding Roles to CORBA Objects,", IEEE Transaction on Software Engineering, Vol 29, No 3, 2003.
[12] Ivers, J., Sinha, N. and Wallnau, K., "A Basis for Composition Language CL," Technical Note, CMU/ SEI-2002-TN-026, Carnegie Mellon Software Engineering Institute, 2002.
[13] Achermann, F., Lumpe, M., Schneider, J. and Nierstrasz, O., A Survey of Object-Oriented Approaches, pp. 403-426, Cambridge University Press, 2001.
[14] Alencar, P.S.C. and Cowan, D.D., "A Logical Theory of Interfaces and Objects," IEEE Transaction on Software Engineering, Vol.28, No.6, 2002.
[15] Fillipe, J.K., "Foundations of Module Concept for Distributed Object System," PhD Thesis, Mnster University, 2000.
[16] Aguirre, N. and Maibaum, T., "A Logical Basis for the Specification of Reconfigurable Component-Based Systems," Fundamental Approach to Software Engineering(FASE) 2003, pp. 37-51, 2003.
[17] Fischer, C., "Combination and Implementation of Processes and Data: from CSP-OZ to Java," PhD Thesis, University of Oldenburg, 2000.
[18] Mahony, B. and Dong, J.S., "Timed Communicating Object-Z," IEEE Transaction of Software Engineering, Vol.26, No 2, 2000.
[19] Derrick, J, and Boiten, E., "Combining Component Specifications in Object-Z and CSP," Formal Aspects of Computing, Vol.13, pp. 111-137, 2002.
[20] Xie, X. and Shatz, S.M.," An Approach for Modeling Components with Customization for Distributed Systems," International Journal of Informatica, Special issue on Component Based Software Development, Vol. 25, No. 4, pp. 465-474, 2001.
[21] Lopes, A., Wermelinger, M. and Fiadelio, J.L., "Higher-Order Architectural Connector," ACM Transaction on Software Engineering and Methodology, Vol.12, No. 1, 2003.
[22] Deline, R., "Avoiding Packing Mismatch with Flexible Package," IEEE Transactions on Software Engineering, Vol.27, No.2, 2001.
[23] Abmann, U., Invasive Software Composition, pp. 167-187, Springer-Verlag, 2003.
[24] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, O., Uthing, U., Paech, B., Wuest, J. and Zettel, J., Component-based Product Line Engineering with UML, pp. 319, Addison-Wesley, 2001.
[25] Smith, G., The Object-Z Specification Language, pp. 133-142, Kluwer Academic Publishers, 2000.
[26] Smith, G., "An Object-Oriented Approach to Formal Specification," PhD Thesis, University of QueensIand, 1992.
[27] Smith, G. and Derrick, J., "Refinement and Verification of concurrent systems specified in Object-Z and CSP," Proceedings of the 1st International Conference on Formal Engineering Methods, 1997.
[28] Plasil, F. and Visnovsky, S., "Behavior Protocols for Software Components," IEEE Transactions of Software Engineering, Vol.28, No.11, 2002.

이 종 국
1991년 고려대 물리학과 이학사. 1993년 고려대 물리학과 석사. 2004년 현재 대우정보 시스템 기술 연구소 연구원. 2004년 현재 숭실대학교 컴퓨터학과 박사과정. 관심분야는 컴포넌트 개발 방법론, 컴포넌트 정형명세, 소프트웨어 아키텍처, Web Service

신 숙 경
1986년 이화여대 수학교육과 이학사. 1999년 숭실대 정보과학대학원 석사. 2004년 현재 숭실대학교 컴퓨터학과 박사과정 2004년 현재 한국학술진흥재단 학술정보팀장. 관심분야는 컴포넌트 개발 방법론, 객체지향 개발 방법론, 컴포넌트 정형명세

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