컴포넌트 워크플로우 가변성의 정형 명세 및 모델링 기법

(Formal Specification and Modeling Techniques of Component Workflow Variability)

이 종 국 † 조 은 숙 †† 김 수 동 †††

(Jong Kook Lee) (Eun Sook Cho) (Soo Dong Kim)

요 약 컴포넌트는 소프트웨어 개발의 복잡성을 감소시키는 효과적인 방법으로 평가되고 있다. 그러나 소프트웨어 개발 시 컴포넌트를 사용하여 기간 단축과 비용 절감 효과를 얻기 위해서는 컴포넌트의 재사용성이 향상되어야 한다. 업무 단위로 컴포넌트를 설계하여 컴포넌트 안에 업무 워크플로우를 포함하는 것은 컴포넌트의 재사용성을 향상시키는 효과적인 방법이다. 워크플로우가 내장된 컴포넌트는 업무 단위로 재사용되기 때문에 개발 기간 단축과 비용 절감 효과가 크다. 몇 몇 컴포넌트 방법론에서 워크플로우를 내장한 컴포넌트 설계 기법의 필요성을 제시했다. 그러나 컴포넌트 개발에 적용하기 위해서는 좀 더 실용적이고 구체적인 기법이 요구된다. 본 논문에서는 컴포넌트를 통한 워크플로우의 재사용을 위해 패밀리 멤버간의 가변적인 워크플로우를 컴포넌트에 내장하여 재사용성을 높이는 기법을 제안한다. 제시된 기법은 워크플로우와 워크플로우 가변성에 대한 정형명세를 통해 복잡한 워크플로우의 설계를 단순화한다. 또한 정형 명세를 통해 워크플로우 가변성 간의 불일치를 해결하고 가변성의 결합도를 낮춘다. 정형 명세와 UML을 사용한 컴포넌트 모델링의 산출물은 컴포넌트 구현 소스 코드를 자동으로 생성하는 것을 돕는다. 따라서 제시된 설계 기법은 개발자의 생산성을 높이고 컴포넌트의 재사용성을 향상시킨다. 본 논문에서는 설계 기법과 함께 예제를 통해 컴포넌트 워크플로우 명세와 설계 기법의 타당성을 입증한다.

키워드 : 컴포넌트 가변성, 워크플로우, 워크플로우 가변성, Provide Interface, Required Interface

Abstract It is well recognized that component-based development (CBD) is an effective approach to manage the complexity of modern software development. To achieve the benefits of low-cost development and higher productivity, effective techniques to maximize component reusability should be developed. Component is a set of related concepts and objects, and provides a particular coarse-grained business service. Often, these components include various message flows among the objects in the component, called 'business workflow'. Blackbox components that include but hide business workflow provide higher reusability and productivity. A key difficulty of using blackbox components with business workflow is to let the workflow be customized by each enterprise. In this paper, we provide techniques to model the variability of family members and to customize the business workflow of components. Our approach is to provide formal specification on the component variability, and to define techniques to customize them by means of the formalism.

Key words : Component Variability, Workflow, Workflow Variability, Provide Interface, Required Interface

† 비 회 원 : 숭실대학교 컴퓨터학과
jklee690@selab.soongsil.ac.kr
†† 정 회 원 : 동덕여자대학교 정보학부 교수
escho@dongduk.ac.kr
††† 종신회원 : 숭실대학교 컴퓨터학과 교수
sdkim@comp.ssu.ac.kr
논문접수 : 2002년 3월 12일
심사완료 : 2002년 7월 15일

1. 서 론

컴포넌트는 소프트웨어의 복잡성을 해결하고 개발 생산성을 높일 수 있는 방법으로 평가되고 있다[1,2,3]. 컴포넌트를 이용하여 개발 생산성을 높이기 위해서는 컴포넌트의 재사용성에 대한 연구가 선행되어야 한다[4]. 현재 컴포넌트의 재사용성을 향상시키는 연구들이 활발

히 진행되고 있다. 그러나 실제 프로젝트에서 적용 하기 위해서는 더 많은 연구가 필요하다.

컴포넌트의 재사용성을 높일 수 있는 방법으로 현재 제안되고 있는 기법은 Product-Line에서 사용되는 공통성/가변성 분석 기법을 사용하여 한 도메인의 패밀리 멤버들의 요구사항을 컴포넌트에 반영하는 것이다[5]. 또한 컴포넌트의 재사용성을 높이기 위해서는 소규모의 라이브러리 성격의 컴포넌트보다는 업무 단위의 컴포넌트를 사용해야 한다[6]. 업무 단위의 컴포넌트는 컴포넌트 안에 업무 워크플로우를 포함해야 한다. 따라서 공통성/가변성과 업무 워크플로우를 포함하는 컴포넌트를 개발하려면 여러 패밀리 멤버의 업무 워크플로우를 컴포넌트에 포함시키고 워크플로우를 사용자의 요구사항에 따라 변경하여 사용할 수 있도록 해야 한다. 이렇게 개발된 컴포넌트는 업무 단위의 컴포넌트로 다양한 사용자의 요구사항을 반영할 수 있기 때문에 재사용성이 높다.

현재까지 컴포넌트를 개발하는 다양한 프로세스가 제안되고 있으나, 실제 프로젝트에 적용할 수 있는 구체적인 기법을 제시하기 위해서는 더 많은 연구가 필요하다[7,8,9,10]. 컴포넌트 안에 워크플로우의 공통성과 가변성을 포함시키는 기법의 연구가 활성화되지 못한 이유는 업무 워크플로우가 매우 다양하여 한 컴포넌트에 포함되도록 설계하기가 어렵기 때문이다.

본 논문에서는 워크플로우의 공통성과 가변성을 기술하기 위해 정형 명세와 UML을 함께 사용한다. 본 논문에서 사용된 정형 명세는 워크플로우를 정확하게 기술할 수 있을 뿐 아니라, 구현 코드로 쉽게 변환된다. 또한 제시된 모델링 기법은 워크플로우 설계시의 복잡성을 감소시키고, 오류 발생 가능성을 줄여준다.

본 논문은 2장에서 컴포넌트 워크플로우에 대한 관련 연구를 제시하고 3장에서는 컴포넌트, 인터페이스, 워크플로우, 워크플로우 가변성을 정형적으로 정의한다. 4장에서는 컴포넌트에 대한 정형적 정의를 통해 컴포넌트 워크플로우와 워크플로우를 설계하는 기법을 제시한다. 5장에서는 제시된 기법의 타당성을 검증한다.

2. 관련 연구

이 장에서는 컴포넌트 개발 방법론, 컴포넌트 정형 명세, 워크플로우에 대한 관련 연구를 제시한다.

2.1 컴포넌트 개발 방법론

이 절에서는 대표적인 컴포넌트 개발 방법론에 대해 언급하고 컴포넌트 워크플로우와 워크플로우 가변성 설계 기법에 대한 연구가 어디까지 진행되었는지를 설명한다.

2.1.1 Business Component Factory

Business Component Factory는 컴포넌트의 단위를 독립적인 업무 개념으로 정의한다[6]. 따라서 Business Component Factory에서 정의한 컴포넌트에는 업무 워크플로우가 내장되어 있으며 업무 워크플로우를 재사용함으로써 개발과 유지보수의 생산성이 향상된다. 업무 워크플로우를 컴포넌트에 포함시키기 위해 워크플로우 관리 방식과 규칙 기반 모델링 방식의 두 가지 기법이 제시된다. 워크플로우 관리 기법에서는 워크플로우를 실행하기 위해 별도의 워크플로우 스크립트를 정의하여 워크플로우를 변경하기 쉽도록 한다. 컴포넌트는 내부에 워크플로우의 일부를 포함하고 있어서 워크플로우의 변경이 용이해진다.

Business Component Factory에서는 컴포넌트를 업무 단위에서 정의하고 컴포넌트의 재사용성과 유지보수성을 높이기 위한 구체적인 방법을 제시하고 있다. 그러나 Business Component Factory는 어떻게 컴포넌트 내부에 워크플로우를 효과적으로 내장하면서 설계의 복잡성을 줄일 것인가에 대하여 일반적으로 알려진 상태 전이도를 사용하는 방법만을 제시하고 있다. 따라서 컴포넌트 내부의 워크플로우를 설계하는 좀 더 효율적이고 설계의 복잡성을 줄일 수 있는 방법이 필요하다. 또한 워크플로우 설계 이전에 도메인 분석을 통해 워크플로우의 변경 가능성이나, 워크플로우의 재사용성을 분석해야 한다.

2.1.2 Software Reuse 방법론

Software Reuse 방법론에서는 가변성을 사용하여 컴포넌트의 재사용성을 향상시키는 방법을 사용한다[4]. 가변성은 컴포넌트를 모델링 할 어플리케이션의 집합인 패밀리에서 패밀리 멤버 간의 기능성의 차이로 인해 발생한다. 따라서 가변성이 포함된 컴포넌트는 한 패밀리에서 사용될 수 있기 때문에 재사용성이 높다. 가변성은 Use-Case에서 처음 식별되며 가변성이 출현하는 지점을 variable point라고 부른다. 가변성은 가변성 구현 장치에 의해 상속, 확장, 사용, 선택, 매개변수화, 템플릿 사용, 일반화로 분류된다. 가변성 구현 장치에 의해 클래스도와 순차도에서 가변성이 설계된다.

Software Reuse 방법론에서는 기존에 알려진 기법을 사용하여 가변성을 설계하였으며 객체 모델링 방법의 일부분으로서 가변성 설계를 사용하였다. 따라서 가변성 설계는 객체 모델링과 효과적으로 통합된다. 그러나 가변성의 조합의 수가 많으면 제시된 가변성 구현 장치와 객체 모델링 기법만으로는 설계시의 복잡성을 감소시킬 수 없다. 또한 Software Reuse 방법론은 컴포넌트를

커스터마이제이션하기 위한 Required Interface의 설계 및 구현 방법에 대해서는 큰 비중을 두고 있지 않다. 따라서 Software Reuse의 가변성 설계가 구체화되기 위해서는 좀 더 체계적인 워크플로우 가변성 설계 지침이 제시되어야 한다.

2.1.3 Paul Allen의 컴포넌트 개발 방법론

Paul Allen은 전자 상거래를 위한 컴포넌트를 개발하기 위해 Business Component Factory와 유사한 컴포넌트 개념을 제시하고 있다[11]. Paul Allen의 방법론에서는 컴포넌트를 인터페이스를 통한 업무 기능을 제공하는 서비스로 정의하고 있다. 따라서 인터페이스를 통해 업무 기능을 제공하기 위해서는 컴포넌트 내부에 워크플로우가 내장되어야 한다. 컴포넌트를 모델링하기 위해서 업무 워크플로우는 활동도를 통해 모델링하며 활동도를 사용하여 워크플로우에 참여하는 컴포넌트를 식별한다. 컴포넌트를 식별한 후에는 동적 모델링을 통해 컴포넌트의 워크플로우를 정제한다. 동적 모델링 중 순차도의 패턴은 컨트롤 클래스를 사용하는 Fork Structure 방식과 컨트롤 클래스를 사용하지 않는 Stair Structure 방식으로 구별한다. 워크플로우 설계 지침으로는 인터페이스와 컴포넌트 사이의 결합도를 최소화하도록 권장한다.

Paul Allen의 방법론은 UML을 사용한 컴포넌트 모델링 방법을 제시하고 있으며 적용하기가 어렵지 않다. 그러나 컴포넌트의 재사용성을 위한 가변성이 고려되지 않았기 때문에 재사용성을 위해서는 방법론이 확장되어야 한다. 또한 컴포넌트 내부 워크플로우 설계를 위해 좀 더 구체적인 지침이 제공되어야 한다.

2.1.4 Rational Unified Process

RUP에서는 컴포넌트 내부 워크플로우 설계를 위한 지침을 제공한다[12]. RUP의 Use Case Realization이라는 기법은 Use Case에서 분석 단계의 클래스와 순차도를 도출하는 기법이다. 컴포넌트 내부의 워크플로우 설계도 Use Case Realization의 지침을 따른다. 한 컴포넌트의 Provide Interface는 Use Case에 대응되며 Provide Interface에서 내부 클래스와 순차도를 도출하는 과정은 Use Case Realization과 대응된다. 컴포넌트 내부의 순차도가 복잡한 경우는 순차도를 축약하여 추상적으로 표현할 수 있다. 또한 순차도는 Paul Allen의 방법론처럼 Centralized와 Decentralized 패턴으로 설계된다.

RUP는 다른 방법론들에 비해 구체적인 컴포넌트 워크플로우 설계 기법을 제시하고 있다. 그러나 RUP는 워크플로우 가변성을 고려하지 않았다. 따라서 RUP는 컴포넌트의 재사용성을 고려하여 기법을 확장되어야 한다. 워크플로우의 가변성을 고려한다면 Use-Case Realization만을 사용하여 컴포넌트 워크플로우를 설계하기는 어렵다. 이유는 Use-Case Realization만을 사용한다면 워크플로우 가변성 사이에 발생하는 불일치를 찾아내기 어렵기 때문이다. 따라서 컴포넌트 워크플로우 설계는 Use-Case Realization외에 가변성에 대한 분석 단계와 가변성 설계 지침이 추가되어야 한다.

2.2 컴포넌트 정형 명세

이 절에서는 컴포넌트 정형 명세 언어에 대한 관련 연구를 제시한다.

2.2.1 Componentware

Componentware는 컴포넌트를 기술하기 위한 정형 명세 언어와 컴포넌트 설계 프로세스를 제시하고 있다[13]. Componentware의 정형 명세 언어는 컴포넌트와 인터페이스, 다른 컴포넌트, 주고 받는 메시지 사이의 관계를 기술하는데 중점을 두고 있다. 컴포넌트의 정적인 측면에 대한 기술은 컴포넌트와 컴포넌트의 관계, 컴포넌트와 인터페이스의 관계를 함수를 사용하여 기술한다. 컴포넌트의 동적인 측면에 대한 기술은 인터페이스에 입력되는 메시지 수열(sequence)과 출력되는 메시지 수열을 사용하여 기술한다. 정형 명세는 집합과 함수만을 사용하여 기술한다. Componentware는 명세가 단순하며 컴포넌트들 사이의 정적인 관계와 상호작용을 기술하기가 용이하다. 그러나 Componentware는 컴포넌트가 가져야 할 선행, 후행 조건이나 불변 조건을 기술하기 위한 장치가 부족하다. 또한 컴포넌트 내부의 워크플로우나 워크플로우 가변성은 고려하지 않았다.

2.2.2 Piccola

Piccola는 PI calculus와 PI-L calculus를 기초로 만들어졌다[14,22]. Piccola는 객체의 추상화, 확장 장치인 상속과 객체 타입의 다형성을 사용하지 않는다. 대신 Piccola는 generic parameter를 사용하여 Haskell과 유사한 방식으로 함수 다형성을 제공한다. Piccola에서는 generic parameter 대신 form이라는 용어를 사용한다. Piccola는 컴포넌트의 컴포지션과 인터페이스의 확장성을 명세하기에 좋은 언어이다. Piccola를 사용하면 plug-in을 표현하기 쉽고, 컴포넌트의 컴포지션도 추상화하기 쉽다.

그러나 Piccola는 컴포넌트를 명세하기 위한 기본적인 언어 장치만을 제공하고 있으며 아직 컴포넌트와 인터페이스를 어떻게 명세할 것인지는 언급하고 있지 않다. 또한 form에 기초한 컴포넌트의 확장과 추상화가 이루어 질 때, 컴포넌트가 가져야 할 불변 조건은 어떻게 유

지할 것인지, 컴포넌트의 동적인 면은 어떻게 명세해야 할 것인지에 대한 연구가 진행되어야 한다.

2.2.3 Acme

Acme는 객체 모델링에 기초한 컴포넌트 정형 명세이다[15]. Acme는 컴포넌트와 인터페이스, 컴포넌트와 컴포넌트의 관계를 기초로 시스템을 표현하려는 점에서 명세의 목적은 componentware와 동일하다. 그러나 Acme는 표기법이 단순하여 componentware에 비해 이해하기 쉽다. pre, post, 불변 조건은 FOPL이라는 언어를 사용하여 기술하는데 FOPL는 OCL과 비슷하게 컴포넌트와 컴포넌트의 관계에 기초하여 제약사항을 명세한다.

그러나 Acme는 컴포넌트의 동적인 면을 기술하기 위한 장치가 부족하며 가변성, 확장성, 추상화를 위한 장치를 아직 제시하고 있지 못하다. 따라서 명세가 쉽고 객체지향 언어에 직관적으로 매핑되는 장점을 유지하면서 동적인 면에 대한 명세를 추가하는 연구가 진행되야 한다.

2.3 워크플로우

이 절에서는 객체 간의 메시지 흐름인 워크플로우를 명세하는 대표적인 방법들을 소개하고 제시된 명세 기법들이 컴포넌트 워크플로우에 적용 가능한지를 검토한다.

2.3.1 UML 순차도

UML 순차도는 분석 단계에서 메시지의 순서만을 보여줄 수도 있고 구현 수준에서 분기문이나 루프도 기술할 수 있다[16,17]. 또한 OCL을 추가하면 워크플로우가 만족해야 하는 조건을 기술 할 수 있다[18]. 따라서 UML 순차도는 컴포넌트의 워크플로우를 기술하기에도 적합하다. 그러나 순차도에는 워크플로우 가변성을 추가하여 확장할 수 있는 장치가 마련되어 있지 못하다. 또한 OCL은 객체의 동적인 면보다는 정적인 면을 기술하기 때문에 OCL을 사용하여 워크플로우 가변성을 추가했을 때의 에러 검사나 일관성 검사가 어렵다. 따라서 워크플로우 가변성을 명세하기 위해 OCL의 확장이 요구된다.

2.3.2 Object-Z

Object-Z는 객체 오퍼레이션 사이에 composition 연산자를 사용하여 한 객체에서 참조되는 다른 객체를 호출하는 상황을 표현할 수 있다[19,20,21]. Object-Z는 클래스 중심으로 기술되기 때문에 컴포넌트의 워크플로우를 명세하기 위해서는 컴포넌트에 대한 클래스를 먼저 명세해야 한다. 또한 워크플로우를 명세하기 위해서는 Provide Interface, Required Interface, 워크플로우를 어떻게 명세해야 할 것인지를 결정해야 한다. 즉 Object-Z로 컴포넌트를 어떻게 명세해야 할 것인지에 대한 연구가 워크플로우 명세 이전에 선행되어야 한다.

3. 컴포넌트의 정형적 정의

3.1 컴포넌트 워크플로우에 대한 정형 명세의 필요성

UML을 사용한 모델링 기법만으로는 컴포넌트 워크플로우 설계의 복잡성을 줄이는 데는 한계가 있다. 다음과 같은 변수를 정의하여 워크플로우 설계의 복잡성을 보인다.

- 컴포넌트의 가변성의 개수 : *CountOfVar*
- 컴포넌트의 오퍼레이션의 개수 : *CountOfOp*
- 워크플로우의 수 : *CountOfWorkflow*

가변성이 컴포넌트에 설정될 수도 있고 안될 수도 있기 때문에 컴포넌트에 설정 가능한 가변성의 갯수는 $2^{CountOfVar}$이다. 가변성의 값들이 정해 졌을 때는 정해진 가변성에 대해 워크플로우를 설계해야 한다. 단순한 방법으로 워크플로우를 설계할 경우 설계해야 하는 워크플로우의 수는 다음과 같다.

$$CountOfWorkflow = 2^{CountOfVar} * CountOfOp$$

한 컴포넌트가 10개의 가변성을 가지고 있다면 가변성의 값들의 조합은 $2^{10}=1024$개이다. 컴포넌트가 다섯 개의 오퍼레이션을 가지고 있다면 설계해야 하는 총 워크플로우의 최대 개수는 1024*5=5120 개이다. 따라서 컴포넌트에서 생성될 수 있는 모든 워크플로우를 UML의 순차도를 사용하여 설계하는 것은 불가능하다. 따라서 워크플로우 가변성과 워크플로우 가변성이 포함되어 있지 않는 워크플로우를 별도로 기술하고 가변성이 설정되었을 때의 워크플로우의 변화는 자동화 도구를 사용하여 기계적으로 도출한다. 이런 방법으로 사용하면 기술해야 하는 산출물의 수는 *CountOfVar*+i *CountOfOp*이다. 한 컴포넌트가 10개의 가변성과 다섯개의 오퍼레이션을 가지고 있을 경우에는 15개의 산출물만 생성하면 된다. 따라서 워크플로우 설계의 복잡성은 크게 줄어든다.

따라서 워크플로우 설계의 복잡성을 감소시키기 위해서는 자동화 도구를 사용해야 하며 자동화 도구를 사용하기 위해서는 컴포넌트를 정형 명세로 기술해야 한다. 정형 명세는 자동화 도구의 입력으로 사용되어, 컴포넌트에 대한 설계를 자동화시키고, 구현 코드도 생성한다.

지금까지 컴포넌트에 대한 다양한 정형 명세가 제시되었다[18,13]. 그러나 정형 명세를 사용한 설계 기법은 학습하는데 많은 시간이 소요되며 개발자들이 설계된 산출물을 이해하기도 어렵다. 따라서 정형 명세는 반드시 필요한 부분에서만 사용하고 UML의 표기법을 병행하여 기술한다. Catalysis와 UML Component에서는

컴포넌트 명세를 위해 OCL와 UML을 병행하여 사용하였다[7]. 본 논문에서도 최소한의 정형 명세만을 제시하고 UML 모델링과 병행하여 컴포넌트를 설계한다. 이 장에서는 컴포넌트와 인터페이스, 컴포넌트 워크플로우, 워크플로우 가변성을 정형적으로 정의하고 설계 기법에서 사용될 표기법을 제시한다. 앞으로 전개할 정형 명세에서는 Z-Notation-A Reference Manual 뒷장에 나오는 집합과 함수에 대한 표기법을 사용하며 추가적으로 필요한 표기법을 정의한다[23].

3.2 컴포넌트와 인터페이스 정의

이 절에서는 컴포넌트와 인터페이스, Provide Interface, Required Interface에 대한 정의를 제시한다. 객체 모델링인 Use-Case와 클래스도, 순차도에서 여러 개의 컴포넌트를 추출 할 수 있다. 이 컴포넌트 들은 인터페이스를 통해 호출된다. 추출된 컴포넌트들의 집합은 *SetOf Components*로 표기한다.

컴포넌트의 집합 $SetOfComponents = \{comp_i \mid comp_i$는 추출된 컴포넌트 중의 하나 $\wedge\ 1 \le I \le NumOf\ Component\ \wedge\ NumOfComponent$는 추출된 컴포넌트의 개수$\}$

*SetOfComponents*에 속한 한 컴포넌트는 Provide Interface와 Required Interface, 컴포넌트 내부 클래스들의 집합으로 구성된다. Provide Interface는 컴포넌트 외부에서 유일하게 컴포넌트에 접속하여 컴포넌트를 실행할 수 있는 수단이다. Required Interface는 컴포넌트의 가변성 중 하나를 설정하여 컴포넌트를 커스터마이제이션하는 인터페이스이다. Required Interface는 선택사항으로 컴포넌트에 포함되지 않을 수도 있다. 컴포넌트는 정적인 관점에서 클래스의 집합을 포함한다. 동적인 관점에서는 컴포넌트가 실행될 때 객체의 호출 순서인 컴포넌트 워크플로우를 포함한다. Provide Interface의 오퍼레이션이 호출되면 한 워크플로우가 실행되므로 컴포넌트는 Provide Interface의 오퍼레이션의 수만큼의 컴포넌트 워크플로우를 포함할 것이다. 또한 컴포넌트 내부에서는 객체들이 메시지를 통해 상호작용하므로 컴포넌트는 메시지의 집합을 갖는다.

$\forall comp_i \in SetOfComponents \bullet$

$comp_i = \{ProvideInterface_i, [RequiredInterface_i], SetOfClasses_i, SetOfMessages_i, SetOfWorkflows_i\}$

윗식에서 $RequiredInterface_i$는 선택 사항이라는 의미로 []를 사용했다.

컴포넌트는 오직 하나의 Provide Interface를 갖는다. Provide Interface는 클래스와는 달리 속성을 갖지 않고 오퍼레이션 만으로 이루어진 집합이다.

$\forall comp_i \in SetOfComponents \bullet$

$\exists_1 ProvideInterface_i =$

$\{OpOfPInf_{i,j} \mid OpOfPInf_{i,j}$는 *Provide Interface*의 한 오퍼레이션 $\wedge\ 1 \le j \le NumOfOpPInf_i$, $NumOfOp PInf_i$는 $ProvideInterface_i$의 오퍼레이션의 개수$\}$

Required Interface는 컴포넌트의 가변성 중 하나를 설정하여 컴포넌트를 커스터마이제이션하는 인터페이스이다. Required Interface도 Provide Interface와 같은 방법으로 정의된다.

$\forall comp_i \in SetOfComponents \bullet$

$\exists_1 RequiredInterface_i =$

$\{OpOfRInf_{i,k} \mid OpOfRInf_{i,k}$는 컴포넌트 $comp_i$에 속하는 $RequiredInterface_i$의 한 오퍼레이션 $\wedge\ 1 \le k \le NumOfOpRInf_i \wedge NumOfOpRInf_i$는 $RequiredInterface_i$의 오퍼레이션의 개수$\}$

컴포넌트는 내부에 클래스들을 포함한다.

$\forall comp_i \in SetOfComponents \bullet$

$\exists_1 SetofClasses_i =$

$\{ class_{i,l} \mid class_{i,l}$은 $comp_i$의 한 클래스 $\wedge\ 1 \le l \le NumOfClasses_i \wedge NumOfClasses_i$은 $comp_i$의 클래스의 개수$\}$

앞에서 정의된 컴포넌트, Provide Interface, Required Interface, 클래스 사이의 관계를 그림으로 나타내면 그림 1과 같다.

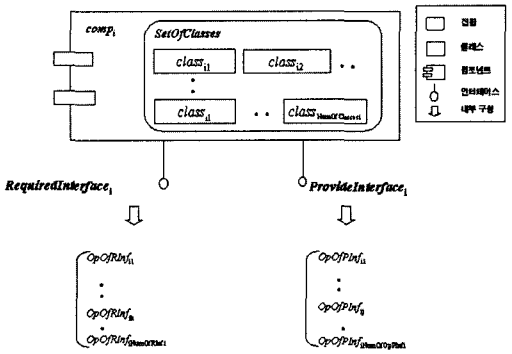


그림 1 컴포넌트의 구성

3.3 컴포넌트 워크플로우의 정의

3.2절에서는 컴포넌트의 정적인 구조를 정의하였다. 이 장에서는 컴포넌트의 동적인 구조를 정의한다. 컴포넌트의 동적인 구조는 메시지와 워크플로우로 표현된다. 메시지는 컴포넌트 내부의 객체들이 상호작용하는 수단

이다. 이 논문에서는 메시지를 객체의 오퍼레이션으로 간주한다. 따라서 컴포넌트의 구성 요소 *SetOfMessages*는 다음과 같이 정의된다.

$\forall comp_i \in SetOfComponents \Rightarrow$

$\exists_1 SetOfMessages_i =$

$\{message_{i,p} \mid message_{i,p}$는 $comp_i$에 속하는 클래스의 한 오퍼레이션 $\wedge\ 1 \le p \le NumOfOp_i \wedge NumOfOp_i$는 $comp_i$에 속하는 클래스의 오퍼레이션의 총수$\}$

Provide Interface의 한 오퍼레이션이 호출되면 컴포넌트 내부에서 객체의 오퍼레이션이 차례로 메시지를 받는다. 한 컴포넌트 내부에서 발생할 수 있는 가능한 모든 메시지 수열의 집합*SetOfMessageSequence*은 자연수와 메시지의 순서쌍의 집합 { (1, m_1), (2,m_2) }로 표현된다.

$\forall comp_i \in SetOfComponents \Rightarrow$

$SetOfMessageSequence_i =$

$\{seqOfMessage : N_1 \mapsto SetOfMessages_i \mid (\exists n : N \cdot \mathrm{dom}\ seqOfMessage = 1..n)\}$

윗 식에서 $\mapsto$은 partial function을 나타내고 N_1은 자연수를 나타낸다. 메시지 수열은 다른 방법으로는 <m_1, m_2, m_3>로 표현한다. 메시지 하나가 수열을 나타낼 때는 <m_1>으로 표현한다.

Provide Interface의 한 오퍼레이션이 호출되었을 때 발생할 수 있는 메시지의 수열은 *SetOfMessage Sequence*$_i$의 부분 집합이다. 이것을 워크플로우라고 부른다. 워크플로우는 다음과 같이 표현된다.

$\forall comp_i \in SetOfComponents \bullet \forall OpOfPInf_{i,j} \in ProvideInterface_i \bullet$

$workflow_{i,j} = \{seqOfOpPInf_{i,j} \mid seqOfOpPInf_{i,j} \in SetOfMessageSequence_i\}$

컴포넌트의 모든 워크플로우의 합집합은 *SetOf MessageSequence*이다. 또한 각기 다른 두 개의 오퍼레이션이 같은 메시지 수열을 생성 할 수 없으므로 워크플로우 사이에는 교집합이 없다. 이것을 식으로 표현하면 다음과 같다.

$\forall comp_i \in SetOfComponents \bullet$

$\bigcup_{j=1}^{j=NmOfOpPInf_i} workflow_{i,j} = SetOfMessageSequence_i$

$\forall comp_i \in SetOfComponents \bullet$

$(\forall j, k < NumOfOpPInf_i \bullet j \neq k) \bullet$

$workflow_{i,j} \cap workflow_{i,k} = \varnothing$

3.2절의 컴포넌트 구성 요소 중 *SetOfWorkflows*는 컴포넌트의 모든 워크플로우를 원소로 갖는 집합이다.

$\forall comp_i \in SetOfComponents \bullet$

$\exists_1 SetOfWorkflows_i =$

$\{workflow_{i,j} \mid 1 \le j \le NumOfOpPInf_i\}$

워크플로우가 메시지 수열 하나에 대응하는 경우도 있지만 Provide Interface의 오퍼레이션을 호출했을 때 매개변수의 값이나 required interface의 호출에 의해 메시지 수열이 달라질 수도 있다. 이 경우 워크플로우는 여러 메시지 수열을 포함하는 집합이 된다. Required interface가 호출된 후 Provide interface의 한 오퍼레이션을 호출했을 때 발생할 수 있는 워크플로우와 Required Interface가 호출되지 않고 Provide interface의 한 오퍼레이션을 호출했을 때 발생하는 워크플로우가 다를 때 워크플로우 가변성이라고 한다. Provide interface의 한 오퍼레이션과 required interface의 한 오퍼레이션에서에서 *SetOfWorkflows*에 대응하는 함수를 *FuncOf-ComponentWorkflow*라고 하면 다음과 같이 정의된다.

$\forall comp_i \in SetOfComponents \bullet$

$\exists_1 FuncOfCompnentWorkflow : ProvideInterface_i \times RequiredInterface_i \rightarrow \mathbb{P}\ SetOfMessage_i$

required interface를 호출하지 않고 provide interface의 오퍼레이션만 호출한 경우는 생성 가능한 워크플로우 중 일부만 호출된다. 따라서 워크플로우는 Provide Interface의 한 오퍼레이션에서 메시지 수열에 대응하는 함수이다. 이 함수를 FuncOfComponentWorkflow1이라고 하면 다음과 같이 정의된다.

$\forall comp_i \in SetOfComponents \bullet$

$\exists_1 FuncOfComponentWorkflow1 : ProvideInterface_i \rightarrow \mathbb{P}\ SetOfMessage_i$

Provide interface의 어떤 오퍼레이션에 대하여 *Func OfComponentWorkflow*와 *FuncOfComponentWorkflow*1이 다를 때 워크플로우 가변성이 존재한다. 이 문장은 다음과 같이 포현된다.

$\forall comp_i \in SetOfComponents \bullet \exists OpOfPInf \in ProvideInterface_i \bullet \exists OpOfRInf \in RequiredInterface_i \bullet FuncOfComponentWorkflow(OpOfPInf, OfOfRInf) \neq FuncOfComponentWorkflow1(OpOfPInf)$

컴포넌트 워크플로우가 발생하는 경우는 네 가지로 분류할 수 있다.

- 컴포넌트 워크플로우에 나타나는 메시지의 순서가 다른 경우 : <*a, b, c*>≠<*a, c, b*>
- 컴포넌트 워크플로우에 있는 메시지나 메시지의 흐름이 추가되는 경우 : <*a, b, c* >≠<*a, b, c, d*>
- 컴포넌트 워크플로우에 있는 메시지나 메시지의 흐름이 삭제되는 경우 : <*a, b, c, d* >≠<*a, b, c*>

컴포넌트 가변성의 세 가지 경우는 컴포넌트 가변성

을 명세하는 데 사용된다. 컴포넌트 가변성에 대한 식을 그림으로 표시하면 다음과 같다.

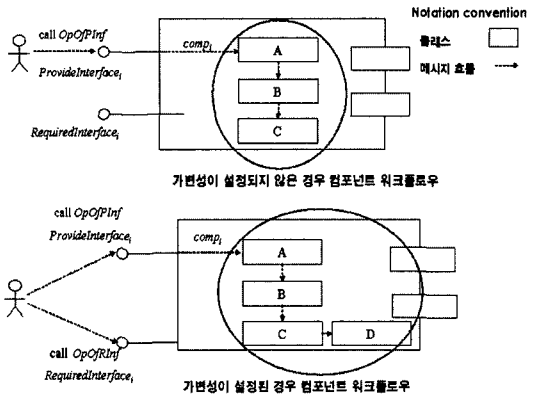


그림 2 컴포넌트 워크플로우와 워크플로우 가변성의 예

3.4 컴포넌트 워크플로우에 대한 표기법

워크플로우는 정형적으로는 메시지 수열의 집합이며 UML의 순차도에 대응된다. 워크플로우는 집합 기호를 사용하여 표기할 수도 있지만, UML의 순차도와의 대응 관계를 고려한 새로운 표기법을 제안한다. 이 표기법을 사용하면 순차도를 정형 명세로 변형하기가 쉽다.

워크플로우는 메시지 수열의 집합이지만 특정 메시지 수열은 Provide Interface의 오퍼레이션의 매개변수 값이나 Required Interface의 호출에 의해 실행된다. 따라서 메시지 수열의 집합은 조건문을 사용하여 메시지 수열을 선택하는 것과 문법적으로 동등하다. 예를 들어 외부 매개변수 i의 값이 0 일 때의 메시지 수열이 <*a, b, c*>이고 0이 아닐 때는 <*a,b,d*>라면 워크플로우는 집합 기호를 사용하면 다음과 같이 표시된다.

{<*a,b,c*>, <*a,b,d*>}

특정 조건을 명시하고 메시지 사이의 순서가 표시되도록 (를 사용하면 다음과 같이 표기된다.

$a \rightarrow b \rightarrow (c)\{\text{if i = 0 }\} \mid (d)\{\text{else}\}$

그러나 의미상으로는 특정 조건을 명시하기 때문에 워크플로우의 의미가 더 명확해진다. 위의 표기법은 문법적으로는 집합 기호를 사용한 식과 문법적으로 동등하다. 따라서 다음 세 식은 문법적으로 동일하다.

$\{ (1,a), (2,b), (3,c) \} = < a, b, c> = a \rightarrow b \rightarrow c$

다음 장부터 이 표기법을 사용하여 워크플로우를 명세한다. 이 표기법을 사용하면 워크플로우 가변성도 명확하게 표기된다. 표 1은 모든 가능한 UML 순차도가 워크플로우에 어떻게 명세되는 가를 보여준다.

4. 컴포넌트 워크플로우 가변성 설계

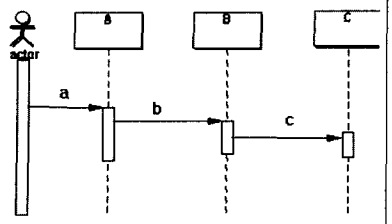
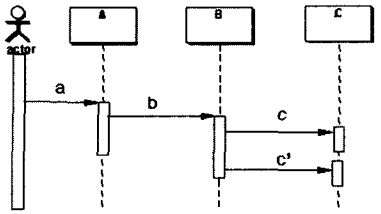
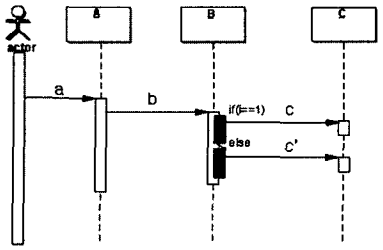
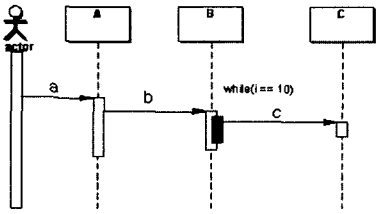
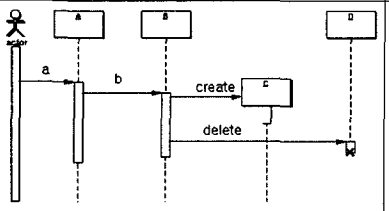
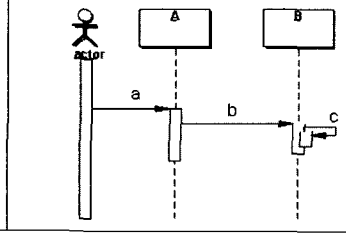
이 장에서는 앞장에서 제시한 컴포넌트의 정형적 정의와 명세 방법을 사용하여 컴포넌트 워크플로우 가변성을 설계한다. 컴포넌트 워크플로우를 설계하기 전에 개략 클래스도와 순차도가 완성되었고 컴포넌트도 식별되었으며 컴포넌트의 인터페이스도 설계되었다고 가정한다.

컴포넌트 워크플로우 설계는 순차도를 작성하여 어플리케이션의 동적인 측면을 설계하는 경우와는 차이가 있다. 첫째, 컴포넌트 워크플로우에는 어플리케이션에는 없는 가변성이 포함된다. 이 가변성은 선택적일 수도 있고 컴포넌트 외부에서 가변성을 Plug-in으로 삽입할 수도 있다. 둘째, 컴포넌트 워크플로우는 Provide Interface가 제공하는 한 오퍼레이션에 대하여 pre, post 조건을 만족해야 한다. 더 나아가서 컴포넌트 워크플로우는 오퍼레이션의 제약 사항을 만족해야 한다.

컴포넌트 워크플로우에 가변성을 포함시키는 첫 번째 방법은 컨트롤러 객체를 사용하는 것이다. 컨트롤러가 컴포넌트 워크플로우를 제어한다면 컨트롤러 객체를 상속하여 가변적인 워크플로우를 생성할 수 있다. 두 번째 방법은 워크플로우에 참여하는 각 객체를 상속하는 방법이다. 1장에서 컴포넌트 워크플로우 설계의 복잡성을 설명하였다. 컴포넌트 워크플로우 설계가 복잡한 또 한 가지 이유는 한 객체가 여러 워크플로우에 참여한다는 것이다. 예를 들어 한 컨트롤러 객체가 여러 워크플로우를 제어할 수 있다. 이 경우 가변성을 포함시키기 위해 컨트롤러 객체를 다른 객체로 대체한다면 가변성과 무관한 다른 워크플로우에서는 에러가 발생할 수 있다. 에러가 발생하지 않더라도 호출된 Provide Interface의 오퍼레이션의 pre, post 조건을 만족하지 않을 경우도 있다.

컴포넌트 워크플로우 설계상의 복잡성을 피하기 위해, 워크플로우 설계 시 정형 명세를 사용한다. 정형 명세를 사용하면 워크플로우간의 불일치와 pre, post 조건을 검사할 수 있다. 제시된 워크플로우 명세는 간단하기 때문에 순차도를 워크플로우 명세로 옮기거나 워크플로우 명세를 순차도로 변환하는 것이 가능하다. 컴포넌트 설계자는 순차도를 작성한 후, 정형 명세로 옮겨서 문제점을 점검하고 순차도를 재설계할 수 있다. 또한 본 논문에서는 컴포넌트 워크플로우를 설계하기 위해 컴포넌트 워크플로우와 워크플로우 가변성을 분리하여 기술하고 다시 통합하는 방법을 사용한다. 컴포넌트 워크플로우 설계의 절차는 그림 3과 같다.

표 1 컴포넌트 워크플로우 표기법

Case	그림	표기법	집합 기호를사용한 표기법
연속		A.a → B.b → C.c	{<A.a,B,b,C.c>}
한 메시지에 대하여 두 개 이상의 오퍼레이션이 수행될 때		A.a → B.b → C.c → C.c'	{<A.a,B.b,C.c,C.c'>}
조건		A.a → B.b → (C.c){if i==1} | (C.c'){else}	{<A.a,B.b,C.c>,<A.a,B.b,C.c'>}
반복		A.a → B.b → RoutineOfC RoutineOfC = RouinteOfC(0) RoutineOfC(0) = C.c → RoutineOfC(1) RoutineOfC(i) = C.c → RoutineOfC(i+1) RoutineOfC(10) = C.c	{<A.a,B.b,C,c,C,c,C,c,C,c,C,c, C.c,C.c,C.c,C.c,C.c,C.c>}
객체의 생성과 삭제		A.a → B.b → C.create → D.delete	{<A.a,B.b,C.create,D.delete>}
재귀 호출		A.a → B.b → RecursionOfB RecursionOfB = B.c → *RecursionOfB*	{<A.a,B.b,C.c..>}

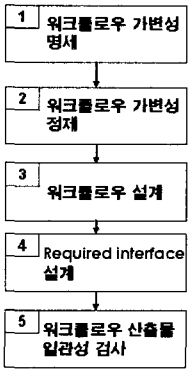


그림 3 컴포넌트 워크플로우 설계 절차

컴포넌트 워크플로우 설계 절차와 산출물과의 관계는 그림 4와 같다. 패밀리 요구사항 명세서는 컴포넌트 워크플로우와 워크플로우 가변성에 대한 정형 명세로 변환되고 부수적으로 Required Interface에 대한 명세도 도출된다. 정형 명세는 다시 가변성을 실행할 수 있는 클래스를 포함한 순차도와 클래스 도로 변환된다. 최종 산출물인 순차도와 클래스도는 컴포넌트 구현 모델로 변환된다.

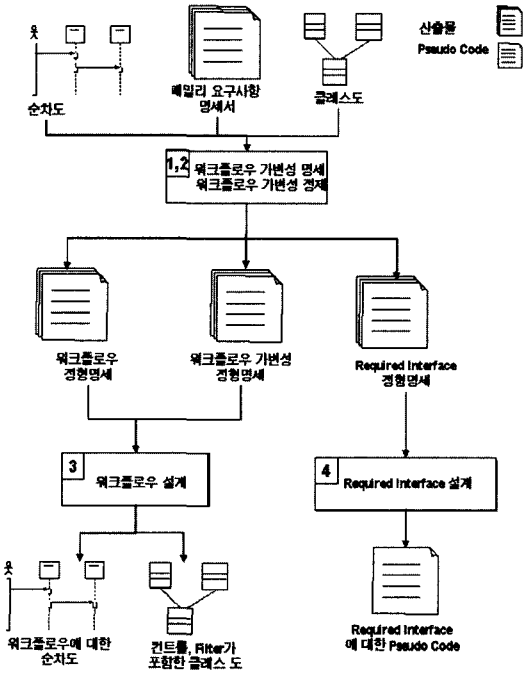


그림 4 워크플로우 설계 단계와 산출물 사이의 관계

워크플로우 설계의 예제로는 비디오 대여 컴포넌트를 사용한다. 비디오 관리에 대한 컴포넌트는 그림 5와 같다. 이 클래스 도는 비디오 대여, 예약, 반납 기능에 대하여 설계되었다. 비디오는 비디오 정보를 관리하는 클래스와 비디오를 개별적으로 관리하는 클래스로 나누어진다. 비디오 대여는 Rent 클래스로 모델링된다.

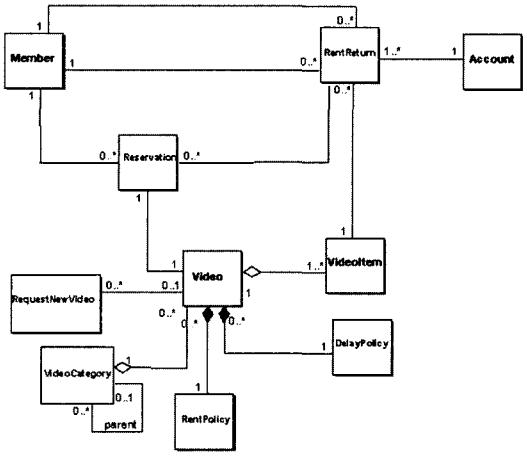


그림 5 비디오 관리 시스템의 클래스 도

비디오 대여는 거의 모든 대여점이 동일한 기능을 가지고 있지만, 비디오 대여 정책에 따라 워크플로우에 차이가 있다. 비디오 대여의 기본적인 흐름은 회원 여부 검사 → 회원 정보 조회 → 비디오 상태 검사 → 대여료 계산 → 반납일 계산으로 이루어진다. 10개의 비디오 대여 점을 조사한 결과 다음과 같은 대여 워크플로우 상에 차이가 있었다.

표 2는 11개 대여점의 워크플로우 가변성 차이를 보여준다. 만일 11개의 가변성 차이만 Required Interface에 반영한다면 11개의 경우만 컴포넌트를 사용할 수 있다. 그러나 11개의 경우에 나타나는 가변성은 9개이다. 9개의 가변성을 Required Interface에서 독립적으로 실행한다면 총 29 개의 case에 컴포넌트를 사용할 수 있으므로 컴포넌트의 재사용성이 극대화된다. 그러나 9개의 서로 독립적이 아니다. 예를 들어 I 대여점의 가변성과 J 대여점의 가변성은 동시에 실행될 수 없다. 따라서 재사용성을 극대화하기 위해서는 워크플로우 가변성 조합의 복잡성과 불일치를 해결해야 한다.

4.1 단계 1. 워크플로우와 워크플로우 가변성 명세

이 단계에서는 각 패밀리 멤버의 요구사항을 분석하여 워크플로우와 워크플로우 가변성을 찾고 명세한다.

표 2 대여 워크플로우의 패밀리 멤버 간의 차이

패밀리 멤버	워크플로우 상의 차이
A 대여점	대여료를 계산하고, 고객의 비디오 이용 횟수를 계산 한 후, 이용 횟수에 따라 대여료를 감면한다.
B 대여점	반납일을 계산하고, 고객의 비디오 이용 횟수를 계산 한 후, 이용 횟수에 따라 대여 기간을 연장한다.
C 대여점	반납일은 컴포넌트에서 자동으로 판단 할 수 없으며 현재 비디오 회수율에 근거하여 직원이 판단한다. 따라서 컴포넌트에는 반납일 계산이 제외된다.
D 대여점	회원 여부를 검사한 후 회원이 아니면 자동으로 회원으로 등록하는 과정이 포함된다. 반납일 계산과 대여료 계산이 반대로 이루어진다. 반납일을 계산 한 후 그 결과에 따라 대여료를 계산한다.
E 대여점	회원 여부를 검사한 후 회원이 아니면 자동으로 회원으로 등록하는 과정과 대여료 계산 후 대여료 감면하는 과정이 포함된다.
F 대여점	대여료 계산하고 대여료 감면하는 과정과 반납일 계산 후 반납일 연장하는 과정이 포함된다.
G 대여점	비디오 상태를 검사한 후 비디오가 불량이거나, 대출 중이면 고객의 취향을 분석 한 후 새로운 비디오를 추천하는 과정이 포함된다. 비디오를 추천한 후에 워크플로우는 종료된다.
H 대여점	회원 정보 조회 후에는 회원 상태에 따라 고객 신용도를 계산한다. 비디오 상태를 검사 한 후에는 비디오가 일정 기간이 지났거나, 비디오의 대출율이 낮으면 비디오 상태를 변경한다. 대여료 계산 후에는 대여료 감면이 이루어지고 반납일 계산 후에는 반납일 연장이 이루어진다.
I 대여점	반납일 계산과 대여료 계산이 반대로 이루어진다. 반납일을 계산 한 후 그 결과에 따라 대여료를 계산한다.
J 대여점	비디오 상태 검사가 반납일 계산 후에 이루어진다. 비디오에 대한 정보를 가져올 때 현재 대여 가능한 비디오의 개수와 예약 되지 않은 비디오의 ID도 가져온다.

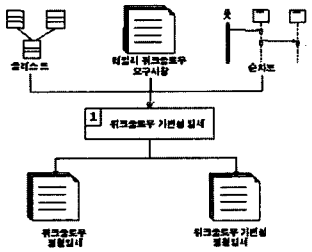


그림 6 단계1의 입력, 출력 산출물

그림 6을 보면 단계 1에서는 클래스도와 패밀리별 워크플로우 요구사항과 순차도를 입력 받는다. 이 입력 산출물을 기초로 컴포넌트 워크플로우를 식별하고 워크플로우 가변성을 찾아서 정형 명세로 명세한다. Provide Interface의 오퍼레이션 op_1(비디오 대여)에 대한 워크플로우와 워크플로우 가변성을 명세하기 위해 표 3의 각 패밀리 멤버의 워크플로우를 메시지의 시퀀스로 변환한다.

표 3 패밀리 멤버 워크플로우의 예

패밀리 멤버	패밀리 멤버의 워크플로우
패밀리 멤버 A	대여→회원 여부 검사→대여 횟수 조회→연체 횟수 조회→미납 비디오 조회→대출 가능 여부 조회→비디오 출시 연도조회→비디어 등급 조회→비디오 상태 조회→대여료 계산→대여료 감면→반납일 계산 $\{m_{1,1}\rightarrow m_{1,2}\rightarrow m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,10}\rightarrow m_{1,11}\rightarrow m_{1,12}\}$
패밀리 멤버 B	$\{m_{1,1}\rightarrow m_{1,2}\rightarrow m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,10}\rightarrow m_{1,11}\rightarrow m_{1,16}\}$
패밀리 멤버 C	$\{m_{1,1}\rightarrow m_{1,2}\rightarrow m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,10}\}$
패밀리 멤버 D	$m_{1,1}\rightarrow m_{1,2}\rightarrow (m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,11}\rightarrow m_{1,10}$ {회원인 경우} | $m_{1,13}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,11}\rightarrow m_{1,10}$ {회원이 아닌 경우})
패밀리 멤버 E	$m_{1,1}\rightarrow m_{1,2}\rightarrow$ ($m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,10}\rightarrow m_{1,12}\rightarrow m_{1,11}$ {회원인 경우} | $m_{1,13}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,10}\rightarrow m_{1,12}\rightarrow m_{1,11}$ {회원이 아닌 경우})
패밀리 멤버 F	$\{m_{1,1}\rightarrow m_{1,2}\rightarrow m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,10}\rightarrow m_{1,12}\rightarrow m_{1,11}\rightarrow m_{1,16}\}$
패밀리 멤버 G	$m_{1,1}\rightarrow m_{1,2}\rightarrow m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow$ ($m_{1,10}\rightarrow m_{1,11}$ {~비디오 불량 ∧ ~대출 중} | $m_{1,14}\rightarrow m_{1,19}$ { 비디오 불량 ∨ 대출 중})
패밀리 멤버 H	$\{m_{1,1}\rightarrow m_{1,2}\rightarrow m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,20}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,15}\rightarrow m_{1,10}\rightarrow m_{1,12}\rightarrow m_{1,11}\rightarrow m_{1,16}\}$
패밀리 멤버 I	$\{m_{1,1}\rightarrow m_{1,2}\rightarrow m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,11}\rightarrow m_{1,10}\}$
패밀리 멤버 J	$\{m_{1,1}\rightarrow m_{1,2}\rightarrow m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,21}\rightarrow m_{1,22}\rightarrow m_{1,10}\rightarrow m_{1,11}\rightarrow m_{1,9}\}$

위의 테이블에서 어떤 워크플로우는 집합 기호를 사용했는데 이유는 메시지 수열과 워크플로우를 구별하기 위해서이다. 위의 테이블에서 공통성과 가변성을 찾을 수 있다. 컴포넌트 워크플로우에서 메시지 흐름의 한 부분을 워크플로우 Clip이라고 부른다. 공통성과 가변성은 워크플로우 Clip을 사용하여 기술할 수 있다. Required Interface의 호출 없이 컴포넌트에서 실행되는 워크플로우는 표 3에서 $\{m_{1,1}\rightarrow m_{1,2}\rightarrow m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,10}\rightarrow m_{1,11}\}$ 이다.

따라서 $SetOfComponentWorkflow_1$은 다음과 같이 기술할 수 있다.

$SetOfComponentWorkflow_1 = \{ Workflow_{11}, Workflow_{12}, Workflow_{13} \}$

($Workflow_1$= 대여 워크플로우, *Workflow2* = 반납 워크플로우, *Workflow3* =예약 워크플로우)

$Workflow_{11}$ = $\{m_{1,1}\rightarrow m_{1,2}\rightarrow m_{1,3}\rightarrow m_{1,4}\rightarrow m_{1,5}\rightarrow m_{1,6}\rightarrow m_{1,7}\rightarrow m_{1,8}\rightarrow m_{1,9}\rightarrow m_{1,10}\rightarrow m_{1,11}\}$

각 패밀리 멤버의 워크플로우는 기본 워크플로우의 변형이므로, 기본 워크플로우를 사용하여 패밀리 멤버 워크플로우로의 변형 과정을 기술한다. 1장의 워크플로우 가변성 정의 시 워크플로우 가변성을 기본 워크플로우의 추가, 삭제, 대체로 정의하였다. 기본 워크플로우에 워크플로우 Clip을 추가, 삭제, 대체하는 것을 명세하기 위해 연산자를 사용한다.

WorkflowClip은 인덱스를 사용하여 표시할 수 있다. 예를 들어 수열 $<m_1, m_2, m_3>$의 workflowclip인 $<m_2, m_3>$는 인덱스 2,3을 이용하여 표시할 수 있다. 이것을 /라는 연산자를 사용하여 표시하면 $<m_1, m_2, m_3>/<m_2, m_3> = \{ <m_1, m_2, m_3>, \{2,3\} \}$으로 표시할 수 있다. /의 정확한 정의를 위해 수열에 적용되는 오른쪽 이동 연산자 >>를 정의한다.

X가 어떤 집합이라고 할 때

$$\_>>\_ : \text{seq } X \times \mathbb{N} \rightarrow \text{seq } X$$

$$\forall n \in \mathbb{N} \bullet (1 \leqslant n \leqslant \#\,\text{seqX}) \Rightarrow (\text{seq } X >> n)(i+n) = \text{seq } X(i)$$

즉 이동 연산자는 수열의 인덱스를 주어진 숫자만큼 이동한다. 같은 방법으로 왼쪽 이동 연산자 <<도 정의할 수 있다.

X가 어떤 집합이라고 할 때

$$\_<<\_ : \text{seq } X \times \mathbb{N} \rightarrow \text{seq } X$$

$$\forall n \in \mathbb{N} \bullet (1 \leqslant n \leqslant \#\text{seqX}) \Rightarrow (\text{seqX}>>n)(i-n) = \text{seqX}(i)$$

오른쪽 이동 연산자와 왼쪽 이동 연산자를 사용하면 어떤 수열이 한 수열의 workflowclip이라는 것의 의미를 정의할 수 있다. 예를 들어 $<m_2, m_3>$가 $<m_1, m_2, m_3>$의 workflowclip이라는 것은 다음과 같은 의미로 정의한다.

$<m_1, m_2, m_3> = \{ (1,m_1), (2,m_2), (3,m_3) \}$

$\mathbb{P}<m_1, m_2, m_3> = \{ \{ (1,m_1), (2,m_2) \}, \{ (2,m_2), (3,m_3)\}, \{ (1,m_1), (3,m_3)\}, ..\}$

$<m_2, m_3> >> 1 = \{ (1,m_2), (2,m_3) \} >>1 = \{ (2,m_2), (3,m_3) \}$

따라서 $(<m_2, m_3>>> 1) \subset <m_1,m_2,m_3>$

즉 어떤 수열이 한 수열의 workflowclip이라는 것은 수열을 어떤 숫자만큼 왼쪽 이동이나 오른쪽 이동을 수행했을 때 한 수열의 부분집합이 된다는 의미이다. 이제 연산자 /는 이동 연산자를 사용하여 다음과 같이 정의된다.

어떤 집합 X, Y에 대하여

$$\_ / \_ : \text{seq } X \times \text{seq } Y \rightarrow (\text{seq } X, \mathbb{P}(N_1 \times N_1))$$

$$\_/\_ = (\text{seq } X, \{(n_1, \#\text{seqY}+n_1-1) \mid \text{seq } Y >> n_1-1 \subset \text{seq } X \vee \text{seq } Y << n_1-1 \subset \text{seq } X\})$$

만일 seq Y가 seqX의 workflowclip이 아니라면 연산의 결과는 (seqX, ∅)이 된다.

어떤 수열에 다른 수열을 삽입하는 연산자는 다음과 같이 정의된다.

$$\_+\_ : \text{seq } X / \text{seq } Y \times \text{seqZ} \rightarrow \text{seq } (X \cup Z)$$

if second(seqX/seqY) = ∅ **then**

$\_+\_$ = seqX

else

let addPoint : $\mathbb{N}_1 \rightarrow$ ran(second(seqX/seqY))

•addPoint(i) < addPoint(i+1) <addPoint(i+2)

let addShiftPoint = { $(n_1, n_2) \mid n_1, n_2 \in \mathbb{N}_1, n_1$

= addPoint(i) + (i-1)*#seqZ+1, n_2

= addPoint(i) + i*#seqZ,

1< i < #dom addPoint}

let subSeq={ a : $\mathbb{N}_1 \rightarrow$ran(_+_) |

b ∈ addShiftPoint, first(b)

< first(a) < second(b), second(a) = _+_(first(a)) }

∀seq ∈ subSeq•seq << min(dom seq) -1 = seq Z ∧

let map : $\mathbb{N}_1 \rightarrow$ dom subSeq ⊳(_+_)•

∀i ∈$\mathbb{N}_1$•(1 < i < #dom subSeq⊳(_+_))

⇒map(i-1) <map(i) <map(i+1)

map∘(subSeq ⊳(_+_)) = seqX

윗 식에서 first는 순서쌍의 첫번째 부분,second은 순서쌍의 두번째 부분을 의미한다. ⊳는 함수에서 도메인을 제한한다는 것을 의미한다. ran은 함수의 range를 의미한다.

어떤 수열에서 다른 수열을 삭제하는 연산자는 다음과 같이 정의된다.

$$\_ - \_ : \text{seq } X \times \text{seq } Y \rightarrow \text{seq } X$$

if second(seqX/seqY) = ∅ **then**

$\_-\_$ = seqX

else

let deletePoint = { a | a= seq Y >> n}}}, n_2

= n_1+#seq Y-1, n ∈ dom(second(seqX/seqY))

let map : $\mathbb{N}_1 \rightarrow$ dom(seqX-deletePoint) •

∀i ∈$\mathbb{N}_1$•(1 < i < #dom subSeq⊳(_-_))

⇒map(i-1) <map(i) <map(i+1)

map∘(seqX-deletePoint) = _- _

어떤 수열의 일부분을 workflowclip로 대체하는 연산자는 다음과 같이 정의된다.

_ ↔ _ : seq X / seq Y × seqZ → seq (X ∪ Z)

---

if second seqX \ seq Y = ∅ **then**

seqX/seqY ↔ seqZ = seqX

else

let shiftSeq = { a | a ∈ (seqZ >> n), n ∈ ran(second(seqX/seqY)) }

let replaceSet = (seqX \seqY) ∪ shiftSeq

let map : N_1↣ dom repalceSet •

∀i ∈N_1•(1 ≤ i ≤ #dom replaceSet)⇒map(i-1) ≤map(i) ≤map(i+1)

mapºreplaceSet = _ ↔ _

지금까지는 수열에 대하여 연산자를 정의했다. 워크플로우는 수열의 집합이므로 워크플로우에 연산자를 정의한 것은 워크플로우의 각 원소에 연산자를 정의한 것과 동일한 것으로 간주한다.

workflow/sequence
= { seq_1/*sequence* | seq_1∈*workflow*}
workflow/seq+*sequence*
= { seq_1/*seq*+*sequence* | seq_1∈*workflow*}
workflow−*sequence*
= { seq_1−*sequence* | seq_1∈*workflow*}
workflow/seq(*sequence*
= { seq_1(*sequence* | seq_1∈*workflow*}

정의된 연산자에 대해 다음과 같은 법칙이 성립한다.

(seq_1/seq_2+seq_3)/seq_4+seq_5
=(seq_1/seq_4+seq_5)/(seq_2/seq_4+seq_5)
+(seq_3/seq_4+seq_5) ..... (1)

(seq_1/seq_2+seq_3)−seq_5
=(seq_1−seq_5)/(seq_2-seq_5)+(seq_3−seq_5) ..... (2)

(seq_1/seq_2+seq_3)/seq_4 ↔ seq_5
=(seq_1/seq_4(seq_5)/(seq_2/seq_4(seq_5)
+(seq_3/seq_4(seq_5) ..... (3)

(seq_1−seq_2)/seq_3+seq_4
=(seq_1/seq_3+seq_4)−(seq_2/seq_3+seq_4) ..... (4)

(seq_1−seq_2)−seq_3
=(seq_1−seq_3)−(seq_2-seq_3) ..... (5)

(seq_1−seq_2)/seq_3 ↔ seq_4
=(seq_1/seq_3(seq_4)−(seq_2/seq_3↔seq_4) ..... (6)

(seq_1/seq_2 ↔ seq_3)/seq_4+seq_5
=(seq_1/seq_4+seq_5)/(seq_2/seq_4+seq_5)
↔ (seq_3/seq_4+seq_5) ..... (7)

(seq_1/seq_2 ↔ seq_3) − seq_4
=(seq_1−seq_4)/(seq_2−seq_4) ↔ (seq_3−seq_4) .... (8)

(seq_1/seq_2 ↔ seq_3)/seq_4 ↔ seq_5
=(seq_1/seq_4 ↔ seq_5)/(seq_2/seq_4 ↔ seq_5)
↔ (seq_3/seq$_4$(seq_5) ..... (9)

위 식은 연산에 대한 정의에서 유도될 수 있다. 위 식의 증명은 부록에서 다룬다.

3.4의 워크플로우에 대한 표기법에 대해서는 다음과 같은 법칙이 성립된다.

• 조건문이 포함되어 있을 경우

이 경우에는 if와 else 문이 별도의 수열이 된다. 워크플로우의 경우에는 연산자가 각각의 수열에 적용되므로 if와 else에도 각각 연산이 적용된다.

seq_1 ↔ seq_2{if..} | seq_3(else..) /seq_4+seq_5
=(seq_1/seq_4+seq_5) (seq_2/seq_4+seq_5{if..}|
(seq_3/seq_4+$seq_5){else..}

• 반복문. 재귀호출의 경우

이 경우에는 반복되는 수열에 모두 연산이 적용된다.

(seq_1 → RoutineOfC, RoutineOfC
=RouinteOfC(0), RoutineOfC(0)
=seq_2 → RoutineOfC(1)
RoutineOfC(i)=seq_2→ RoutineOfC(i+1),
RoutineOfC(10)=seq_2)/seq_3+seq_4
=(seq_1 /seq_3+seq_4)→ RoutineOfC,
RoutineOfC=RouinteOfC(0),
RoutineOfC(0)=seq_2/seq_3+seq_4→ RoutineOfC(1)
RoutineOfC(i)=seq_2/seq_3+seq_4→ RoutineOfC(i+1),
RoutineOfC(10)=seq_2/seq_3+seq_4

정의된 연산자를 사용하여 워크플로우 가변성을 간략하게 명세할 수 있다. 단계 1에서 완성된 컴포넌트 워크플로우에 대한 명세서를 요약하면 표 4와 같다. 표 4는 비디오 관리 시스템에 어떤 컴포넌트가 포함되어 있으며 한 컴포넌트는 몇 개의 클래스로 구성되어 있고 Provide Interface의 한 오퍼레이션이 어떤 워크플로우를 발생시키는지를 명확하게 보여준다.

예제에서 패밀리 멤버의 워크플로우 가변성을 세 개의 연산자를 이용하여 기술하면 다음과 같다.

표 5는 워크플로우 가변성을 설정하기 위해 Required Interface의 오퍼레이션에서 수행해야 할 일을 기술한다. 예를 들어 패밀리 멤버 A의 워크플로우를 실행하려면 Required Interface에서 새로운 메시지를 삽입해야 함을 알 수 있다. 패밀리 멤버 G의 워크플로우를 실행하려면 워크플로우 대체와 삽입을 수행해야 한다. 표 5는 나중에 Required Interface에 대한 pseudo code로 변환된다.

표 4 컴포넌트 워크플로우의 명세 예

$SetOfComponents$={$comp_1$, $comp_2$} ($comp_1$=비디오 관리, $comp_2$=자금 관리)
$comp_1$={$ProvideInterface_1$, $RequiredInterfacecomp_1$, $SetofClasses_1$, $SetOfMessages_1$, $SetOfComponentWorkflows_1$) }
$ProvideInterface_1$={$op_{1,1}$, $op_{1,2}$, $op_{1,3}$} ($op_{1,1}$=비디오 대여, $op_{1,2}$=비디오 반납, $op_{1,3}$=비디오 예약)
$SetofClasses_1$={$class_{1,1}$, $class_{1,2}$, $class_{1,3}$, $class_{1,4}$, $class_{1,5}$, $class_{1,6}$, $class_{1,7}$, $class_{1,8}$, $class_9$, $class_{1,10}$, $class_{1,11}$}
($class_{1,1}$=Member, $class_{1,2}$=AllTransaction, $class_{1,3}$=RentReturn, $class_{1,4}$=Account,$class_{1,5}$=Reservation, $class_{1,6}$=VideoItem, $class_{1,7}$=VideoWaste, $class_{1,8}$=RequestNewVideo, $class_{1,9}$=VideoCategory, $class_{1,10}$ =DelayPolicy, $class_{1,11}$=RentPolicy)
$SetMessages_1$={$m_{1,1}$, $m_{1,2}$, $m_{1,3}$, $m_{1,4}$, $m_{1,5}$, $m_{1,6}$, $m_{1,7}$, $m_{1,8}$, $m_{1,9}$, $m_{1,10}$, $m_{1,11}$, $m_{11,12}$, $m_{11,13}$, $m_{1,14}$, $m_{11,15}$, $m_{1,16}$, $m_{1,17}$, $m_{1,18}$, $m_{1,19}$, $m_{1,20}$, $m_{1,21}$, $m_{1,22}$}
($m_{1,1}$=대여, $m_{1,2}$=회원 여부 검사, $m_{1,3}$=대여 횟수 조회, $m_{1,4}$=연체 회수 조회, $m_{1,5}$=미납 비디오 조회, $m_{1,6}$=대출 가능 여부 조회, $m_{1,7}$=비디오 출시 연도 조회, $m_{1,8}$=비디오 등급 조회, $m_{1,9}$=비디오 상태 조회, $m_{1,10}$=대여료 계산, $m_{1,11}$=반납일 계산, $m_{1,12}$=대여료 감면, $m_{1,13}$=회원 등록, $m_{1,14}$=고객 취향 분석, $m_{1,15}$ =비디오 상태 변경, $m_{1,16}$=반납일 연장, $m_{1,17}$=연체료 계산, $m_{1,18}$=비디오 예약, $m_{1,19}$=새로운 비디오 추천, $m_{1,20}$=고객 신용도 계산, $m_{1,21}$=대여 되지 않은 비디오 Item 가져옴, $m_{1,22}$=예약되지 않은 비디오 Item 가져옴)
$SetComponentWorkflow_1$={$Workflow_{1,1}$, $Workflow_{1,2}$, $Workflow_{1,3}$}
($Workflow_{1,1}$=대여 워크플로우, $Workflow_{1,2}$=반납 워크 플로우, $Workflow_{1,3}$=예약 워크플로우)
$Workflow_{1,1}$={$m_{1,1}$→$m_{1,2}$→$m_{1,3}$→$m_{1,4}$→$m_{1,5}$→$m_{1,6}$→ $m_{1,7}$→ $m_{1,8}$→$m_{1,9}$→$m_{1,10}$→$m_{1,11}$}

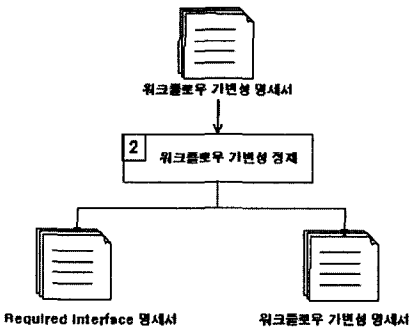
표 5 패밀리 멤버의 워크플로우 가변성 기술 예

패밀리 멤버	패밀리 멤버의 워크플로우 가변성
패밀리 멤버 A	$Workflow_{1,1}$/<$m_{1,10}$>+<$m_{1,12}$>
패밀리 멤버 B	$Workflow_{1,1}$/<$m_{1,10}$>+<$m_{1,16}$>
패밀리 멤버 C	$Workflow_{1,1}$-<$m_{1,11}$>
패밀리 멤버 D	($Workflow_{1,1}$/<$m_{1,11}$>↔<$m_{1,10}$>)/<$m_{1,2}$> +<$m_{1,13}$>{회원이 아닌 경우}
패밀리 멤버 E	$Workflow_{1,1}$/<$m_{1,10}$>+<$m_{1,12}$>{회원인 경우}| ($Workflow_{1,1}$/<$m_{1,3}$>+<$m_{1,13}$>)/< $m_{1,10}$> +<$m_{1,12}$>{회원이 아닌 경우}
패밀리 멤버 F	(($Workflow_{1,1}$/<$m_{1,1}$0>+<$m_{1,12}$>)/<$m_{1,10}$> +<$m_{1,12}$>)/< $m_{1,11}$> +<$m_{1,16}$>
패밀리 멤버 G	$Workflow_{1,1}$(~비디오 불량 ∧ ~대출중} | $Workflow_{1,1}$(<$m_{1,10}$→$m_{1,11}$>)↔<$m_{1,14}$→$m_{1,19}$)) { 비디오 불량 ∨ 대출 중}
패밀리 멤버 H	(($Workflow_{1,1}$/< $m_{1,5}$>+<$m_{1,20}$>)/<$m_{1,9}$> +<$m_{1,15}$>)/<$m_{1,11}$>+<$m_{1,16}$>
패밀리 멤버 I	$Workflow_{1,1}$/<$m_{1,10}$>↔<$m_{1,11}$>
패밀리 멤버 J	(($Workflow_{1,1}$-<$m_{1,9}$>)/<$m_{1,11}$> +<$m_{1,9}$>) /<$m_{1,8}$>+$m_{1,21}$→$m_{1,22}$

4.2 단계 2. 워크플로우 가변성 정제

단계 2에서는 단계 1의 워크플로우 가변성 명세서를 입력 받아 워크플로우 가변성 간의 모순을 검사하고 워크플로우 가변성 검사 테이블을 출력한다. 단계 2에서 추출된 워크플로우 가변성은 아직 정제가 이루어지지 않았다. 각 패밀리 멤버마다의 가변성만 컴포넌트에서 사용한다면 가변성의 정제는 필요치 않다. 그러나 패밀리 멤버에서 추출된 가변성을 동시에 컴포넌트에서 사용할 경우는 정제가 필요하다. 비디오 컴포넌트의 예제에서는 9개의 가변성을 가지고 있으며 모든 가변성을 동시에 사용할 수 있도록 한다면 컴포넌트의 재사용 범위가 넓어질 것이다. 그러나 9개의 가변성을 컴포넌트에서 사용하기 위해서는 가변성들 사이에 불일치가 발생

그림 7 단계 2의 입력, 출력 산출물



하는지를 검사해야 한다. 비디오 컴포넌트 예제에서는 패밀리 멤버 H와 패밀리 멤버 J의 가변성을 동시에 사용할 수 없다. 패밀리 멤버가 작을 경우에는 다음과 같은 지침을 사용하여 가변성들 사이에 불일치를 검사할 수 있다.

- 두 가변성에서 한쪽은 Workflow Clip을 Add하고 다른 쪽은 Workflow Clip을 Delete하는 경우 에러가 발생한다.
- 두 가변성의 한쪽은 Workflow Clip을 Replace하고 다른 쪽은 Workflow Clip을 Delete하는 경우 에러가 발생한다.
- 두 가변성에서 모두 같은 워크플로우의 위치에 Workflow Clip을 Add할 경우, Add한 Workflow Clip에 순서가 필요하다면 에러가 발생할 가능성이 있다.

첫 번째, 두 번째 불일치는 워크플로우에 대한 정형명세와 /, +, −, ↔에 대한 법칙을 사용하여 검사할 수 있다. 만일 Add하거나 Replace한 워크플로우 clip의 일

부분이라도 다른 연산을 함께 적용한 결과 삭제되었다면 모순이 발생한 것이다. 따라서 Add나 Replace한 워크플로우 clip에 대해 오른쪽, 왼쪽 이동 연산자를 적용하여 불일치를 검사할 수 있다.

$\exists n \in \mathbf{N},\ seq_1 \ll n \subset workflow \vee seq_1 \gg n \subset workflow$

Add나 Replace된 workflowclip이 다른 연산을 조작한 결과 만들어진 워크플로우에 대하여 위 식을 만족하지 않는다면 불일치가 발생한 것이다. 좀 더 자동화된 검사를 위해서는 워크플로우에 대한 표기법을 모두 수열의 집합으로 바꾸고 워크플로우 clip이 위 조건을 만족하는지 검사하면 된다.

세 번째의 경우는 워크플로우의 의미를 살펴보고 의미의 변형이 발생했는지 검사해야 한다.

첫 번째, 두 번째의 불일치를 검사하는 또 다른 방법은 명세를 입력 받아 검사하는 알고리즘을 사용하는 것이다.패밀리 멤버의 개수가 증가하면 워크플로우 가변성 간의 불일치를 지침만을 사용하여 검사하는 것은 많은 시간을 필요로 한다. 본 논문에서는 워크플로우 가변성의 명세를 입력하여 가변성 간의 불일치를 검사할 수 있는 알고리즘을 제시한다.

워크플로우는 자료구조 중 tree를 사용하여 표현하고, +, ↔, −는 tree를 조작하는 함수로 생각할 수 있다. 따라서 tree를 조작하는 세 개의 함수는 다음과 같은 pseudo code로 표현된다.

```
Add( tree, message, WorkflowClip)
   index = tree->search(message)
   if( index < 0 )
         return ERROR
   else
         tree->add(index, WorkflowClip)

Delete( tree, WorkflowClip)
   index = tree->search(WorkflowClip)
   if( index < 0 )
         return ERROR
   else
         tree->delete(WorkflowClip)

Replace(tree, WorkflowClip1, WorkflowClip2)
   index1 = tree->getParent(WorkflowClip1)
   index2 = tree->getParent(WorkflowClip2)
   Message message1 = tree->getMessage(index1);
   Message message2 = tree->getMessage(index2);
   if( Delete(tree, WorkflowClip1) != ERROR &&
      Delete(tree, WorkflowClip2) )
         if( tree->Add(message1, WorkflowClip2) )
               if( tree->Add(message2, WorkflowClip1)
               else
                     return ERROR
         else
               return ERROR
   else
         return ERROR
```

또한 두 개의 워크플로우 가변성은 적용 순서에 따라 다른 결과를 만들어서는 안된다. 이것은 워크플로우를 가변적으로 만드는 두 개의 함수가 서로 교환 가능해야 한다는 것을 의미한다. 예를 들어 어떤 워크플로우에 work flowclip 두 개를 더하는 것은 다음과 같이 표시한다.

$(workflow/seq_1+seq_2)/seq_3+seq_4$

윗 식에서 $/seq_1+seq_2$는 워크플로우를 변형시키는 함수로 생각할 수 있다. 따라서 함수의 합성 기호를 사용하여 워크플로우에 두 개의 workflowclip을 추가하는 함수는 다음과 같이 표현된다.

$(/seq_1+seq_2)^{\circ}(/seq_3+seq_4)$

워크플로우 가변성이 독립적이라는 것은 위의 두 함수가 교환 가능하다는 것을 의미한다.

$(/seq_1+seq_2)^{\circ}(/seq_3+seq_4)$
$=(/seq_3+seq_4)^{\circ}(/seq_1+seq_2)$

위 식이 교환 가능할 조건은 연산자에 대한 공식 (1)에서 유도될 수 있다.

$(workflow/seq_1+seq_2)/seq_3+seq_4$
$=(workflow/seq_3+seq_4)/(seq_1/seq_2+seq_4)$
$+(seq_2/seq_3+seq_4)$

위 식에 다음과 같은 조건은 대입하면

$seq_1/seq_2+seq_4=seq_1 \wedge seq_2/seq_3+seq_4$

식은 다음과 같이 변형된다.

$(workflow/seq_1+seq_2)/seq_3+seq_4$
$=(workflow/seq_3+seq4)/seq_1+seq_2$

따라서 조건을 만족하면 +연산자에 의한 가변성 함수는 교환 가능하다. 따라서 위의 조건을 만족하는지를 검사하면 가변성이 독립적인지를 검사할 수 있다. 나머지 연산자에 대한 독립성의 조건도 연산자에 대한 식에서 쉽게 유도할 수 있다. +, −, ↔에 대한 연산의 독립성에 대한 조건을 표로 요약하면 다음과 같다.

조건을 적용하여 예제에서 찾은 워크플로우 가변성의 불일치와 독립성은 표 6과 같다.

패밀리 멤버의 워크플로우 가변성을 분석한 결과로 컴포넌트에 포함시킬 수 있는 워크플로우 가변성이 어떤 것인지 알 수 있었다. 컴포넌트에 설정 가능한 최소의 워크플로우 가변성을 단위 워크플로우 가변성이라 하자. 비디오 대여점 예제에서는 9개의 단위 워크플로우 가변성을 도출하였다. 그러나 포함시킬 수 있는 단위 워크플로우 가변성이 너무 많다면 가변성에 대한 명세와 컴포넌트에 대한 명세도 복잡할 것이고 컴포넌트 사용자가 워크플로우 가변성을 설정하는 것도 복잡할 것이다. 컴포넌트 사용자의 사용 편리성을 위해 단위 워크플

표 6 +, −, ↔에 대한 연산의 독립성 조건

	+	−	↔
+	second(seq_1/seq_3)=∅ ∧ second(seq_2/seq_3)=∅ ∧ second(seq_3/seq_1)=∅ ∧ second(seq_4/seq_1)=∅⇒ $(/seq_1+seq_2)^*(/seq_3+seq_4)$ $=(/seq_3+seq_4)^*(/seq_1+seq_2)$	second(seq_1/seq_3)=∅ ∧ second(seq_2/seq_3)=∅ ∧ second(seq_3/seq_1)=∅⇒ $(/seq_1+seq_2)^*-seq_3=$ $-seq_3{}^*(/seq_1+seq_2)$	second(seq_1/seq_3)=∅ ∧ second(seq_2/seq_3)=∅ ∧ second(seq_3/seq_1)=∅ ∧ second(seq_4/seq_1)=∅⇒ $(/seq_1+seq_2)^*(/seq_3\leftrightarrow seq_4)=$ $(/seq_3\leftrightarrow seq_4)^*(/seq_1+seq_2)$
−		second(seq_1/seq_2)=∅ second(seq_2/seq_1)=∅ $-seq_1{}^*-seq_2=-seq_2{}^*-seq_1$	second(seq_1/seq_3)=∅ ∧ second(seq_2/seq_3)=∅ ∧ second(seq_3/seq_1)=∅⇒ $-\text{seq1}^*(/seq_2\leftrightarrow seq_3)=$ $(/seq_2\leftrightarrow seq_3)^*-seq_1$
↔			second(seq_1/seq_3)=∅ ∧ second(seq_2/seq_3)=∅ ∧ second(seq_3/seq_1)=∅ ∧ (second(seq_4/seq_1)=∅⇒ $(/seq_1\leftrightarrow seq_2)^*(/seq_3\leftrightarrow seq_4)=$ $(/seq_3\leftrightarrow seq_4)^*(/seq_1\leftrightarrow seq_2)$

표 7 워크플로우 가변성 검사 테이블의 예

불일치 없음: ○ 불일치 있음: × 독립적임: ◇ 독립적이지 않음: ⊗

	A	B	C	D	E	F	G	H	I	J
A		○, ◇	○, ◇	○, ◇	○, ◇	○, ◇	○, ◇	○, ◇	○, ◇	○, ◇
B			○, ◇	○, ◇	○, ◇	○, ◇	○, ◇	○, ◇	○, ◇	○, ◇
C				×, ⊗	○, ◇	×, ⊗	×, ⊗	×, ⊗	×, ⊗	×, ⊗
D					○, ◇	○, ◇	○, ◇	○, ◇	○, ◇	○, ◇
E						○, ◇	○, ◇	○, ◇	○, ◇	○, ◇
F							○, ◇	○, ◇	○, ◇	○, ◇
G								○, ◇	○, ◇	○, ◇
H									○, ◇	○, ◇
I										○, ◇
J										

로우 가변성을 조합하여 Required Interface의 호출에 의해 한번에 설정될 수 있도록 해야 한다. 따라서 Required Interface의 가변성 설정이 사용자 편리성을 제공하면서 불일치가 없도록 설계되어야 한다. 워크플로우 가변성의 재설계는 다음과 같은 경우로 나누어 설계된다.

• Case 1. 단위 워크플로우 가변성들이 불일치가 없고 독립적인 경우

이 경우는 단위 워크플로우 가변성을 개별적으로 컴포넌트 워크플로우에서 실행되도록 하고 단위 워크플로우의 조합은 Required Interface에서 제어한다. 단위 워크플로우의 조합이 Provide Interface의 한 오퍼레이션의 pre, post 조건을 만족하는지를 검사해야 한다. pre, post 조건을 만족하지 않는 조합은 실행될 수 없도록 해야 한다. 비디오 대여 예제에서 불일치가 없고 독립적인 단위 워크플로우 가변성은 다음과 같다. 이제 가변성은 연산자를 결합한 함수로 표현하고 워크플로우는 표기하지 않는다.

표 8 비디오 컴포넌트의 불일치가 없고 독립적인 워크플로우 가변성

워크플로우 가변성
$/<m_{1.10}>+<m_{1.12}>$
$/<m_{1.10}>+<m_{1.16}>$
$/<m_{1.11}>\leftrightarrow<m_{1.10}>$
$/<m_{1.2}>+<m_{1.13}>$
$/<m_{1.9}>+ m_{1.14}\rightarrow m_{1.19}${비디오 불량 ∨ 대출 중}
$/ <m_{1.5}>+<m_{1.20}>$
$/<m_{1.9}>+<m_{1.15}>$
$/<m_{1.11}>+<m_{1.16}>$
$/<m_{1.8}>+ m_{1.21}\rightarrow m_{1.21}$

• Case 2. 단위 워크플로우 가변성들이 불일치가 없고 독립적이지 않은 경우

이 경우는 독립적이지 않은 가변성을 조합하고 조합된 가변성들끼리는 독립적이도록 한다. 독립적이지 않은 단위 워크플로우 가변성이 *var1*, *var2* 라면 가변성의

조합은 *var1*°*var2*, *var21*°*var1*이다.

- Case 3. 단위 워크플로우 가변성들이 불일치가 있고 독립적이지 않은 경우

불일치가 있는 가변성들은 컴포넌트에서 함께 실행될 수 없도록 Required Interface 설계 시 에러 체크 로직을 설계한다.

단계 3을 사용하여 정제된 워크플로우 가변성은 1장에서 제시한 *FuncOfCompnentWorkflow*와 Required Interface에 대한 기술로 명세된다. 는 비디오 컴포넌트에 대하여 정제된 컴포넌트 워크플로우 가변성에 대한 명세이다.

표 9 비디오 컴포넌트 워크플로우 가변성 명세

SetOfWorkflowVariability = {*var1*, *var2*, *var3*, *var4*, *var5*, *var6*, *var7*, *var8*, *var9*, *var10*}
var1 = /<$m_{1,10}$>+<$m_{1,12}$>
var2 = /<$m_{1,10}$>+<$m_{1,16}$>
var3 = - <$m_{1,11}$>
var4 = /<$m_{1,11}$>+<$m_{1,10}$>
var5 = /<$m_{1,2}$>+<$m_{1,13}$>{회원이 아닌 경우}
var6 = /< $m_{1,11}$>+<$m_{1,16}$>
var7 = /<$m_{1,9}$>+ $m_{1,14}$→$m_{1,19}$
var8 = (/< $m_{1,9}$>+< $m_{1,15}$>)°(/< $m_{1,5}$>+<$m_{1,20}$>)
var9 = (/<$m_{1,11}$>+<$m_{1,9}$>)°(-<$m_{1,9}$>)
var10 =(/< $m_{1,21}$>+<$m_{1,22}$>)°(/< $m_{1,8}$>+<$m_{1,21}$>)
FuncOfComponentWorkflow : *ProvideInterface*$_i$ ×*RequiredInterface*$_i$ → **P** *SetOfMessage*$_i$
(*rent*, *RequiredInterface*::*setWorkflowVariability* (*var*$_3$, *var*$_4$)) → ∅
(*rent*, *RequiredInterface*::*setWorkflowVariability* (*var*$_3$, *var*$_7$)) → ∅
(*rent*, *RequiredInterface*::*setWorkflowVariability* (*var*$_3$, *var*$_7$)) → ∅
(*rent*, *RequiredInterface*::*setWorkflowVariability* (*var*$_3$, *var*$_{10}$)) → ∅
(*rent*, *RequiredInterface*::*setWorkflowVariability* (*var*$_8$, *var*$_{10}$)) → ∅

위 테이블은 첫번째 줄에서 설정 가능한 워크플로우 가변성을 보여준다. var9와 var10은 두개의 단위 워크플로우 가변성의 조합이다. FuncOfComponentWorkflow은 모든 가능한 워크플로우의 조합을 보여줄 수 없으므로 에러가 발생하는 조합만을 표시한다. FuncOfComponentWorkflow의 명세는 Required Interface를 설계할 때 사용한다.

4.3 단계 3. 컴포넌트 워크플로우 설계

단계 3에서는 명세된 컴포넌트 워크플로우와 워크플로우 가변성을 통합한다. 컴포넌트 워크플로우와 워크플로우 가변성을 통합하기 위해 컨트롤 클래스와 Filter

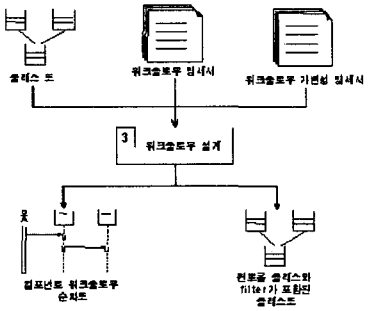


그림 8 단계 3의 입력, 출력 산출물

클래스를 사용한다.

워크플로우는 두 가지 방식으로 설계된다. Centralized 패턴은 한 컨트롤러 클래스를 통해 워크플로우를 제어한다. Decentralized 패턴은 메시지를 받은 객체가 다른 객체를 호출함으로써 워크플로우가 성립된다.

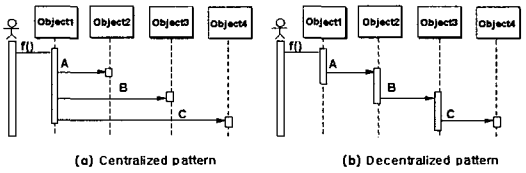


그림 9 컴포넌트 워크플로우 설계 패턴

Centralized 패턴을 사용하는 경우는 컴포넌트의 특성이 Transaction 처리 위주일 때 사용한다. Centralized 패턴의 장점은 워크플로우가 컨트롤러에게 집중되어 있으므로 워크플로우의 순서를 바꾸는 것이 쉽다는 것이다. 컨트롤러 객체를 상속하여 대체하면 워크플로우의 순서를 바꿀 수 있고 워크플로우 가변성 설계도 쉬워진다. 또한 기존의 워크플로우 외에 새로운 워크플로우를 추가하는 것도 어렵지 않다.

Decentralized 패턴을 사용할 경우는 워크플로우에 참여하는 객체들 사이에 Aggregation 관계나 Composition 관계가 있을 때 사용한다. 또는 객체들 사이에 계층 구조의 관계가 있을 때 사용한다. 예를 들어 객체가 회사, 부서, 직원의 계층 구조를 가지고 있다면 Decentralized 패턴을 사용해야 한다. 또는 시간 순서대로 실행 되야 할 경우, 예를 들어 주문, 선적, 지불 객체일 경우는 Decentralized 패턴을 사용한다. 또한 개념적인 상속 관계 일 때도 Decentralized 패턴을 사용한다. 이 경우는 컨트롤 객체가 없기 때문에 워크플로우에 참여하는 객체를 상속 받거나, 가변성에 대한 코드를 객체

에 구현해야 한다.

Centralized 패턴과 Decentralized 패턴은 한 워크플로우에 함께 나타날 수도 있다. 예를 들어 비디오 대여 예제에서는 전체적으로는 대여 컨트롤러를 사용하는 것이 좋지만, 부분적으로 고객과 비디오 정보, 대여 정보의 관계는 Decentralized 패턴으로 처리된다. 따라서 단계 3의 컨트롤러를 사용하는 경우와 객체 대체를 사용하는 경우는 완전히 분리된 것이 아니며 한 워크플로우와 워크플로우 가변성을 설계할 때 두 가지를 모두 고려해야 하는 경우가 대부분이다.

컴포넌트 워크플로우를 설계하기 위한 첫 번째 스텝은 워크플로우를 제어할 수 있는 컨트롤클래스를 찾는 것이다. 컨트롤 클래스는 워크플로우의 패턴을 Centralized 패턴이나 Decentralized 패턴을 사용하여 분류함으로써 찾는다. 혹은 워크플로우가 두 개의 패턴을 복합적으로 가지고 있다면 워크플로우를 몇 개의 워크플로우 Clip으로 분리하여 워크플로우 Clip의 패턴을 분류한다. 컨트롤러 클래스를 찾고 설계하는 지침은 OOSE와 RUP를 비롯하여 여러 객체 방법론에서 언급되었다. 컨트롤 클래스를 찾기 위한 추가적인 지침은 다음과 같다.

워크플로우를 제어하기 위해 Provide Interface는 하나의 컨트롤러 클래스를 반드시 가지며 Provide Interface의 오퍼레이션은 컨트롤 클래스의 오퍼레이션에 대응된다.

컨트롤 클래스를 찾기 위해서는 클래스의 성격을 Boundary, Control, Entity로 분류하고 Collaboration diagram을 작성하면 컨트롤 클래스를 보다 쉽게 찾을 수 있다.

하나의 워크플로우에 하나의 컨트롤러 클래스를 갖는 것이 대부분이지만 워크플로우가 실행되는 동안 워크플로우의 성격과 관계 없는 별도의 워크플로우 Clip을 실행한다면 이 워크플로우 Clip을 제어하는 컨트롤 클래스가 필요하다.

트랜잭션을 관리하는 컨트롤 클래스가 필요하다. 예를 들어 대여 컴포넌트에서는 대여, 반납 트랜잭션을 처리하는 컨트롤 클래스가 필요하다.

비디오 컴포넌트 예제에서 식별된 컨트롤 클래스는 다음과 같다.

표 10 비디오 컴포넌트의 컨트롤 클래스

컨트롤 클래스	내용
대여 클래스	대여 전체 워크플로우를 관리한다.
반납 클래스	반납 전체 워크플로우를 관리한다.
예약 클래스	예약 전체 워크플로우를 관리한다.

대여 클래스는 대여 워크플로우 뿐만 아니라 반납 워크플로우와 예약 워크플로우에서도 사용된다. 반납 워크플로우는 반납시 대여 클래스를 통해 대여된 비디오를 조회한다. 예약 시에는 예약하려는 비디오가 이미 대여되었는지를 대여 클래스를 사용하여 조회한다. 예약 클래스는 대여 워크플로우에서 대여하려는 비디오가 예약된 것인지를 조회할 때 사용한다.

두 번째 스텝에서는 식별된 컨트롤 클래스를 사용하여 가변성이 없는 기본적인 워크플로우에 대한 순차도를 작성한다. 비디오 컴포넌트의 대여에 대한 순차도는 다음과 같다.

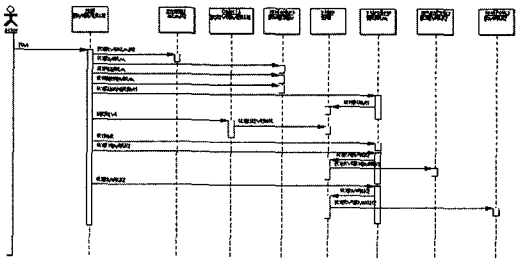


그림 10 비디오 컴포넌트의 대여에 대한 순차도

그림 10을 보면 대여 워크플로우 전체는 Rent Controller가 제어하지만 Video, VideoItem, RentPolicy, ReturnPolicy 사이에는 Aggregation, Composition 관계 때문에 컨트롤 클래스로 제어 되지 않는 것을 알 수 있다. 또한 RentController는 부 워크플로우인 예약 확인을 실행하기 위해 ReserveController를 호출한다.

세번째 스텝에서는 식별된 컨트롤 클래스를 사용하여 워크플로우 가변성을 제어할 수 있는지 판단한다. 컨트롤 클래스와 워크플로우 가변성 사이에 테이블을 작성하여 제어 여부를 검사한다. 표 11의 컬럼명은 표 9을 사용하였다.

표 11 컨트롤 클래스와 워크플로우 가변성 연관 테이블

	var1	*var2*	*var3*	*var4*	*var5*	*var6*	*var7*	*var8*	*var9*	*var10*
대여	√	√	√	√	√	√		√	√	
반납								√		
예약							√			√

표 11을 보면 var7은 고객의 비디오 대여 취향을 분석하는 것으로 대여 취향 분석 워크플로우 Clip을 대여 컨트롤 클래스에 통합하기 어려워서 체크하지 않았다.

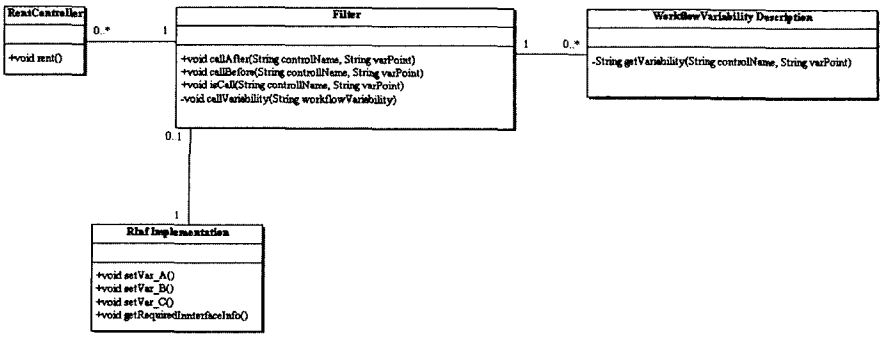


그림 11 Filter 클래스와 관련 클래스 들 사이의 관계

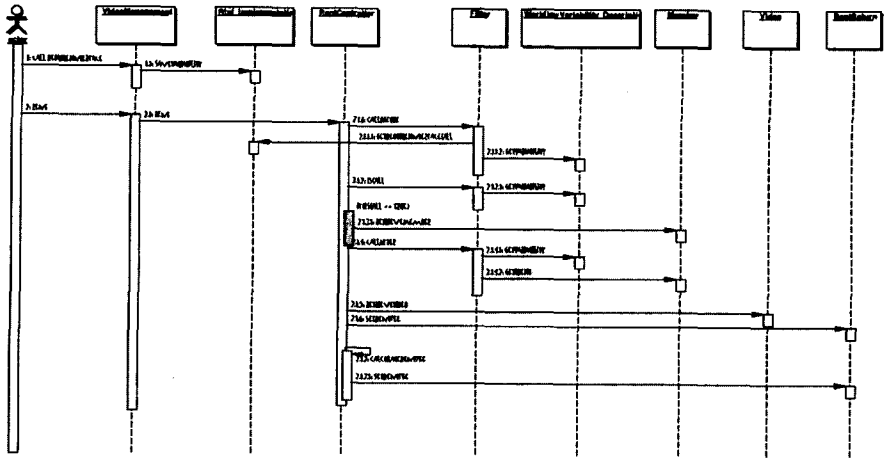


그림 12 Filter 클래스의 워크플로우 가변성 실행 장치

비디오 정보를 가져올 때 기본 워크플로우에서는 비디오 출시 연도와 비디오 등급을 가져오지만 var10의 경우는 관련된 비디오 아이템 중 대출 가능한 아이템을 가져오고 예약되었는지를 검사한다. var10의 경우는 대여 컨트롤 클래스에서 제어하지 않고 비디오와 비디오 아이템 클래스에서 워크플로우 가변성을 직접 제어한다.

네 번째 스텝에서는 컨트롤 클래스와 워크플로우 가변성 연관 테이블을 참고하여 컨트롤 클래스를 설계한다. 총 10개의 워크플로우 가변성 중 컨트롤 클래스에서 8개의 워크플로우 가변성을 제어할 수 있다. 컨트롤 클래스를 상속하여 워크플로우 가변성을 설계한다면 8개의 가능한 가변성을 모두 실행하려면 $2^8 = 256$ 개의 상속된 컨트롤 클래스를 만들어야 한다. 컴포넌트에 너무 많은 클래스를 넣으면 유지보수 하기 어렵다. 따라서 컨트롤의 기능을 확장할 수 있는 다른 장치가 필요하다. 본 논문에서는 컨트롤 클래스의 메시지를 대신 전달 받고 메시지를 해석하여 컴포넌트 안에 객체들을 처리하는 Filter 클래스의 설계를 제안한다. Filter 클래스는 가변성이 check 되었다면 가변성에 대한 규칙을 입력으로 받아 컨트롤 클래스의 명령을 재해석하여 실행한다. 본 논문에서 제안한 방식은 일종의 Rule-Base 방식이다. Rule-Base 방식의 특징은 규칙에 대한 문법만 정해지면 얼마든지 다양한 처리를 할 수 있다는 것이다. 따라서 가변성의 경우의 수가 많은 컴포넌트 워크플로우 설계에 적합하다. 또한 Filter 클래스가 읽어야 하는 rule은 가변성에 대한 정형명세에서 도출될 수 있다. Filter 객체와 컨트롤 클래스, 가변성의 명세, Required Interface의 호출 사이의 관계는 다음과 같다.

그림 12를 보면 컴포넌트 사용자는 Required Interface를 호출한다(1). 호출된 Required Interface의 가변성

실행 정보는 Required Interface에 대한 구현 객체에 저장된다(1.1). 그 후에 사용자는 Provide Interface의 한 오퍼레이션을 호출한다(2). 그러면 오퍼레이션은 컨트롤 객체를 실행한다(2.1). rent 메소드가 호출된 지점에서 가변성이 발생하므로 retrieveMember 메소드를 호출하기 전에 Filter 객체의 callBefore를 호출한다(2.1.1). Filter 객체의 callBefore 메소드는 가변성에 대한 정형 명세를 읽어서 retrieveMember 메소드를 호출하기 전에 Required Interface에서 어떤 가변성이 셋팅되었는지를 조사한다(2.1.1.1). 그리고 실행해야 할 가변성이 무엇인지를 조사한다(2.1.1.2). 그림에서는 retrieve Member 메소드를 호출하기 전에 실행해야 할 가변성은 없다. 그 후에 rent 객체는 Filter 객체의 isCall 메소드를 호출한다. isCall 메소드는 retrieveMember 메소드를 호출해야 하는지 판단한다. 예를 들어 가변성에 대한 명세가 Delete(retrieveMember)라면 isCall 메소드는 False를 리턴할 것이다. 그림에서 isCall 메소드의 호출 결과는 True이므로 retrieveMember 메소드를 호출한다(2.1.3.1). retrieveMember 메소드를 호출한 후에는 Filter 객체의 callAfter 메소드를 호출한다(2.1.4). callAfter 메소드의 내부에서는 Filter의 getVariability를 호출하여 실행해야 할 가변성이 무엇인지 조사한다(2.1.4.1). *getVariability* 메소드는 가변성에 대한 정형 명세를 조사하여 실행해야할 가변성이 있으면 리턴한다. 가변성이 신용 평가 정보를 리턴하는 것이므로 Filter는 Member의 getCredit 메소드를 호출한다. callAfter 메소드를 호출한 후에는 가변성이 체크되어 있지 않으므로 컨트롤 객체는 Filter 객체에게 메시지를 보내지 않는다.

컨트롤 객체가 제어하지 못하는 워크플로우 가변성에 대해서는 워크플로우 가변성을 실행해야 하는 객체를 직접 상속받아야 한다. Required Interface에 대한 구현 객체를 사용하여 상속 받은 객체가 생성되도록 한다. 즉 Required Interface에 대한 구현 객체는 Required Interface의 호출에 대한 정보를 가지고 있을 뿐 아니라 워크플로우를 실행하는 객체를 생성하는 Factory 클래스의 역할도 한다.

4.4 단계 4. Required Interface 설계

Required Interface의 구현 객체는 interface 호출에 대한 parameter 정보를 저장하고 워크플로우 가변성 명세에 따라 동시에 실행될 수 없는 워크플로우 가변성의 조합에 대해 에러를 리턴한다. 에러 리턴 코드는 표 9의 Required Interface에 대한 명세를 사용한다. 워크플로우 가변성은 Required Interface의 오퍼레이션 하나를

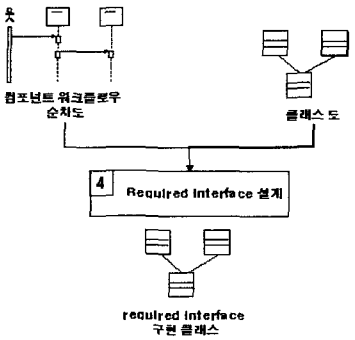


그림 13 단계 4의 입, 출력 산출물

호출하여 설정하는 방법이 있고, 한 오퍼레이션의 매개변수의 값을 정하여 설정하는 방법이 있다. 오퍼레이션 하나에 워크플로우 가변성 하나를 매핑하는 방법은 컴포넌트가 실행 중 일 때 워크플로우의 가변성을 설정할 때 적합하다. 본 논문에서는 컴포넌트가 실행 중 일 때의 가변성 설정은 고려하지 않는다. Required Interface에 오퍼레이션 하나를 사용하고 매개변수를 통해 가변성을 설정하는 방법은 컴포넌트 실행 전에 가변성을 설정할 때 적합하다. 본 논문의 비디오 예제에서는 두번째 방법을 사용한다. 비디오 예제의 Required Interface 구현 클래스는 배열을 사용하여 가변성의 값을 저장한다. 다음 Java 코드는 비디오 예제의 Required Interface 구현을 보여준다.

```
interface RequiredInterface{

Boolean setWorkflowVariabilities(Boolean var1, Boolean
var2, Boolean var3, Boolean var4, Boolean var5, Boolean
var6, Boolean var7, Boolean var8, Boolean var9, Boolean
var10);
}
class RequiredInterface_Impl implements
RequiredInterface{

    Boolean Variabilities[] = new Boolean[10];

    Boolean setWorkflowVariabilities(Boolean var1,
Boolean var2, Boolean var3,Boolean var4, Boolean var5,
Boolean var6, Boolean var7, Boolean var8, Boolean var9,
Boolean var10){

        if( check(var1, var2, var3, var4, var5, var6,
var7, var8, var9, var10) == false){
            return false;
        }

        Variabilities[0] = var1;
        Variabilities[1] = var2;
        Variabilities[2] = var3;

        Variabilities[3] = var4;
        Variabilities[4] = var5;
        Variabilities[5] = var6;
        Variabilities[6] = var7;
        Variabilities[7] = var8;
        Variabilities[8] = var9;
```

```
        Variabilities[9] = var10;
    }

    Boolean check(Boolean var1, Boolean var2, Boolean
var3, Boolean var4, Boolean var5,
    Boolean var6, Boolean var7, Boolean var8, Boolean
var9, Boolean var10){

        if( (var3 == true && var4 == true) ||
            (var3 == true && var7 == true)
                || (var3 == true && var10 == true)
|| (var8 == true && var10 == true) )
                return false
        else
                return true;
    }
}
```

위의 코드를 보면 check 함수는 Required Interface의 정형 명세에서 매핑될 뿐 아니라, Required Interface 구현 코드가 정형 명세에서 자동으로 생성될 수 있음을 알 수 있다.

4.5 단계 5. 설계된 워크플로우의 정제

컴포넌트 워크플로우의 모델은 순차도, 컴포넌트 명세서, 가변성 명세서, Required Interface 명세서, 클래스도, Filter에 대한 클래스 도로 이루어진다. 워크플로우 모델에 대한 산출물이 완성되면 각 산출물 사이의 일관성을 검증해야 한다. 각 산출물 들 사이에 다음 사항을 검사한다.

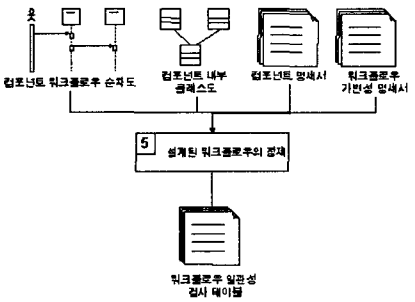


그림 14 단계 5의 입, 출력 산출물

- 컴포넌트 워크플로우 순차도와 컴포넌트 명세서
  워크플로우 순차도가 컴포넌트 명세서에 모두 반영되어 있는지 검사한다.
- 가변성 명세서와 컴포넌트 워크플로우 순차도
  가변성 명세서를 사용했을 때 컴포넌트 워크플로우에 모순이 발생하지 않는지 검사한다. 이 경우는 컨트롤 클래스를 포함한 컴포넌트 워크플로우가 모순이 없는지 검사해야 한다.
- Filter에 대한 클래스 도와 가변성 명세서
  Filter에 대한 클래스 도가 가변성 명세서를 만족하는지 검사한다. 이 경우는 Filter에 대한 클래스와 가변성 명세서를 사용하여 원하는 가변성이 실행 가능한지를 검사해야 한다.

검사가 완료되면 일관성이 없는 정형 명세와 다이아그램에 대한 수정이 이루어진다.

6. 검증 및 평가

본 논문에서 제시된 결과를 컴포넌트 정형 명세인 Componentware, Piccola, Acme와 비교하면 표 11과 같다.

표 12 컴포넌트 워크플로우 명세 비교

	본 논문의 정형 명세	Compo-nentware	Piccola	Acme
워크플로우 명세 방법	메시지 수열과 연산자를 사용하여 명세	인터페이스의 입, 출력만 메시지 수열로 명세	컴포넌트 간에 주고 받는데 이 타 를 form을 통해 명세	컴포넌트 간에 주고 받는 메시지를 connect를 통해 명세
워크플로우 가변성에 대한 명세	연 산 자 를 사 용 하 여 명세	없음	Form을 사용하여 가변성의 값 설정을 명세하는 것이 가능	없음
워크플로우 검증 기법	가 변 성 에 대한 연산자를 사용하여 검증	없음	없음	없음

본 논문은 컴포넌트 내부의 워크플로우 명세에 중점을 두고 있지만 Componentware는 컴포넌트를 이용한 시스템 구성에 Piccola는 컴포넌트를 이용한 시스템 구성시의 확장성과 유연성에 Acme는 컴포넌트 아키텍쳐에 중점을 두고 있다. 따라서 본 논문은 다른 정형 명세를 보완하는 역할을 한다. 본 논문의 명세는 단순한 집합과 연산자를 사용했기 때문에 다른 명세와 통합이 가능하다.

워크플로우 설계 기법을 다른 컴포넌트 방법론과 비교하면 다음과 같다.

본 논문의 설계 기법은 워크플로우 가변성 추출에서, 설계, 검증, 구현까지를 다루고 있으며 다른 컴포넌트 방법론에서 미흡한 워크플로우 분석, 설계, 구현을 보완할 수 있다.

본 논문이 제시한 명세의 장점은 다음과 같다. 본 논문에서 제시된 컴포넌트 워크플로우 설계기법은 컴포넌트 워크플로우 설계시의 복잡성을 감소시킨다. 워크플로우 가변성은 독립적이고 불일치가 발생하지 않는 워크플로우 가변성의 조합으로 변환되며 모순이 발생하는

표 13 컴포넌트 워크플로우 설계 기법 비교

	본 논문의 설계 기법	Business Component Factory	Software Reuse	Paul Allen 방법론	RUP
워크플로우에 대한 설계	순차도와 정형 명세를 사용	순차도를 사용하여 컴포넌트 간의 워크플로우 설계	순차도를 사용하여 컴포넌트 간의 워크플로우 설계	순차도와 활동도를 사용하여 컴포넌트 간의 워크플로우 설계	순차도를 사용하여 컴포넌트간의워크플로우와컴포넌트내부의워크플로우 설계
워크플로우 가변성에 대한 설계	정형 명세를 사용하여 설계	없음	가변성 식별	없음	없음
워크플로우에 대한 설계 패턴	워크플로우와 워크플로우 가변성에 대한 설계 패턴제시			워크플로우에 대한 설계 패턴 제시	워크플로우에 대한 설계 패턴 제시

경우는 Required Interface의 명세에서 에러 검사 코드로 변환된다. 본 논문은 정형 명세를 사용하기 때문에 설계 단계에서 에러가 발생할 경우를 정확하게 찾을 수 있다. 또한 본 논문의 정형 명세는 UML의 클래스도, 순차도와 함께 작성되기 때문에 작성하기가 쉽다. 또한 정형 명세는 Required Interface에 대한 코드와 컨트롤 클래스에 대한 코드에 포함되기 때문에 코드 작성에 대한 노력을 감소시킨다. 본 논문의 워크플로우 정형명세와 검증 기법은 워크플로우 작성을 위한 CASE 툴의 개발로 더욱 생산성을 높일 수 있다.

본 논문에서 제시된 Filter를 사용한 워크플로우 설계 방법은 워크플로우 가변성이 발생하는 워크플로우의 한 지점에서만 사용되며 본 논문에서 제시된 Filter를 상속하여 컴포넌트가 실행되는 플랫폼에 맞게 최적화할 수 있다. 예를 들어 EJB 플랫폼에서는 가변성에 대한 정형 명세를 ejb-ja.xml 파일에 넣고 Filter에서 읽어오면 된다. COM의 경우는 정형명세를 Registry에 등록하고 Filter에서는 Registry를 읽는 API를 사용한다.

또한 본 논문에서 제시된 컴포넌트 워크플로우 설계 기법은 특정한 컴포넌트 방법론을 가정하지 않았기 때문에 어떤 컴포넌트 개발 방법론과도 통합이 가능하다. 예를 들어 UML Component의 Requirements Definition 단계에서 각 패밀리의 워크플로우 요구사항을 수집하고 Component Specification 단계에서 본 논문의 모든 단계를 수행한다. Component Specification에서는 Pre, Post 조건을 OCL로 표현하기 때문에 본 논문과 긴밀하게 통합할 수 있다.

6. 맺음말

지금까지 컴포넌트 워크플로우에 대한 설계 기법을 제시했다. 컴포넌트 워크플로우와 워크플로우 가변성의 복잡성 때문에 워크플로우에 대한 설계는 정형 명세를 통해 이루어진다. 본 논문에서는 컴포넌트에 대한 정형적인 모델을 제시한 후에, 워크플로우 가변성을 독립적인 단위로 그룹핑하였다. 가변성을 독립적인 단위로 그룹핑한 후, 워크플로우 가변성에 모순이 발생하는 경우는 Required Interface에 대한 명세로 변환된다. 워크플로우 가변성은 기본 워크플로우를 변환하는 방식으로 명세되며 이렇게 명세된 워크플로우 가변성은 Filter 클래스에서 워크플로우 가변성을 실행할 때 사용된다. 워크플로우에 대한 설계는 컨트롤 클래스와 Filter 클래스, 워크플로우 가변성을 저장한 Repository를 사용하여 작성된다. 워크플로우 가변성은 워크플로우가 컨트롤 클래스에 의해 통제될 경우는 Filter 클래스를 사용하여 설계되고, 컨트롤 클래스가 없을 경우는 메시지를 전달하는 클래스를 각각 상속하여 설계된다.

본 논문은 현재까지 컴포넌트 방법론에서 개념적으로만 제시된 워크플로우와 워크플로우 가변성에 대하여 구체적인 설계 기법을 제시하였다. 따라서 본 논문은 컴포넌트를 개발하고자 하는 개발자들과 구체적이고 실무적인 컴포넌트 방법론을 제시하고 하는 사람들에게 도움이 될 것으로 기대된다.

참 고 문 헌

[1] Heineman, G. T., Council, W. T., *Component-based Software Engineering*, Addison Wesley, 2001.

[2] Hopkins, J., "Component Primer," Communication of the ACM Vol. 43, No.10 , October 2000.

[3] Kang, K., "Issues in Component-Based Software Engineering," International Workshop on Component-Based Software Engineering 1999.

[4] Jacobson, I.., Griss, M., Jonsson, P., Sofwtware Reuse Architecture Process and Organization for Business Success, ACM Press, 1997.

[5] Lee, S. D., Yang, Y. J., Cho, E. S., Kim, S. D., "COMO : A UML-Based Component Development Methodology," Asia-Pacific Software Engineering Conference(APSEC99), Takamachu, Japan, PP. 54 - 61, Dec. 7-10, 1999.

[6] Herzum, P., Sims, O., Business Component Factory

A Comprehensive Overview of Component-Based Development for the Enterprise, Wiley & Sons, 2000.
[7] D'Souza, D. F., and Wills, A. G., Objects, *Components, and Frameworks with UML,* Addison Wesley, 1999.
[8] Sterling Software, "The COOL:Gen Component Standard3.0," Sept.1999.
[9] Allen, P., Frost, S., Component-Based Development for Enterprise Systems, Cambridge, 1998.
[10] Atkinson, C., Component-Based Product Line Engineering with UML, Addison Wesley, 2002.
[11] Allen, P., Realizing e-Business with Components, Addison-Wesley, 2000.
[12] Jacobson, I., Booch, G., Rumbaugh, J., *The Unified Software Development Process*, Addison Wesley, 1999.
[13] Bergner, K., Rausch, A., Sihling, M., Vilbig, A., Broy, M., "A Formal Model for Componentware," *Foundations of Component-based Systems*, pp. 189-210, Cambridge, 2000.
[14] Achermann, F., Lumpe. M., Schneider, J., Nierstrasz, O., "Piccola - a Small Composition Language," *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, Howard Bowman and John Derrick (Eds.), pp. 403-426, Cambridge University Press, 2001.
[15] Garlan. G., Monroe, R. T., Wile, D., "Acme: Architectural Description of Component-Based Systems," *Foundations of Component-based Systems*, pp. 47-67, Cambridge, 2000.
[16] UML Specification v1.4, OMG,Inc., September, 2001.
[17] Finkbeiner, B., Kruger, I., "Using Message Sequence Charts for Component-based Formal Verification," OOPSLA, October 2001.
[18] Warmer, J. B., Kleppe, A. G., The *Object Constraint Language:Precise Modeling with UML*, Addison-Wesley, 1999.
[19] Smith, G., *The Object-Z Specification Language*, Kluwer Academic Publishers, 2000.
[20] Mahony, B., Jin, S.D., "Timed Communicating Object Z," IEEE Transaction on Software Engineering, Vol.26, N0.2, February 2000.
[21] Smith, G., *The Object Specification Language*, Kluwer Academic Publisher, 1999.
[22] Leavens, G. T., Sitraman, M., *Foundation of Component-based Systems*, Cambridge, 2000.
[23] Spivey, J.M., *The Z Notation A Reference Manual*, URL : http://spivey.oriel.ox.ac.uk/~mike/ zrm/, 1992.

부 록

부록에서는 +,-,↔에 대한 23,24 페이지의 식에 대한 증명을 보여준다.

식 (1)에 대한 증명은 다음과 같다.

한 수열의 뒤에 다른 수열을 연결하는 연산자를 ^라고 하면 seq1에 seq2가 포함되어 있다는 것은 다음과 같이 표시 할 수 있다.

$seq_1 = seq_1' \hat{} seq_2 \hat{} seq_1''\hat{}seq_2\cdots$

seq_1/seq_2+seq_3는 ^을 사용하여 다음과 같이 표시된다.

$seq_1/seq_2+seq_3 = seq_1'\hat{}seq_2\hat{}seq_3\hat{}seq_1''\hat{}seq_2\hat{}seq_3..$

윗 식에 seq_4가 포함되어 있다면 다음과 같이 표시된다.

$seq_1'\backslash\hat{}seq_4\hat{}seq_1'\backslash\backslash..seq_2'\hat{}seq_4\hat{}seq_2''..seq_3'\hat{}seq_4\hat{}seq_3''..\hat{}seq_1''\backslash\hat{}seq_4..$

$(seq_1/seq_2+seq_3)/seq_4+seq_5$는 다음과 같이 표시된다.

$seq_1'\backslash\hat{}seq_4\hat{}seq_5\hat{}seq_1'\backslash\backslash..seq_2'\hat{}seq_4\hat{}seq_5\hat{}seq_2''..seq_3'\hat{}seq_4\hat{}seq_5\hat{}seq_3''..\hat{}seq_1''\backslash\hat{}seq_4\hat{}seq_5..$

윗식을 보면 seq_3/seq_4+seq_5가 항상 seq_2에 연결되어 있다. 따라서 윗식은 다음과 같이 변경된다.

$(seq_1'\backslash\hat{}seq_4\hat{}seq_5\hat{}seq_1'\backslash\backslash..seq_2'\hat{}seq_4\hat{}seq_5\hat{}seq_2''..seq_1''\backslash\hat{}seq_4\hat{}seq_5..)/(seq_2/seq_4+seq_5)+(seq_3/seq_4+seq_5)$

왼쪽 식은 seq_1/seq_4+seq_5와 같다.

따라서

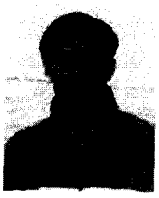
$(seq_1/seq_2+seq_3)/seq_4+seq_5$
$=(seq_1/seq_4+seq_5)/(seq_2/seq_4+seq_5)$
$+(seq_3/seq_4+seq_5)$

나머지 (2)-(8)식도 같은 방법으로 증명할 수 있다.

이 종 국
1991년 고려대 물리학과 이학사. 1993년 고려대 물리학과 석사. 1995년 ~ 1999년 10월 대우정보 시스템 연구원. 2000년 ~ 현재 숭실대학교 컴퓨터학과 박사과정 재학중. 관심분야는 컴포넌트 개발 방법론, 컴포넌트 정형명세, 소프트웨어 아키텍처, Web Service

조 은 숙

1993년 동의대학교 전산통계학과 이학사. 1996년 숭실대학교 컴퓨터학과 공학석사. 2000년 숭실대학교 컴퓨터학과 공학박사. 1999년 ~ 2000년 ㈜객체정보기술 과장. 2000년 9월 ~ 현재 동덕여대 정보학부 데이터정보학과 강의전임교수. 2002년 1월 ~ 현재 한국전자통신 연구원 초빙연구원. 관심분야는 컴포넌트기반 소프트웨어 개발 방법론, 소프트웨어 아키텍처, 컴포넌트 정형명세, 컴포넌트 메트릭



김 수 동

1984년 Northeast Missouri State University 전산학 학사. 1988년 The University of Iowa 전산학 석사. 1991년 The University of Iowa 전산학 박사. 1991년 ~ 1993년 한국통신 연구개발단 선임연구원. 1993년 ~ 1994년 The University of Iowa 교환교수. 1994년 ~ 1995년 현대전자 소프트웨어연구소 책임연구원. 1995년 9월 ~ 현재 숭실대학교 컴퓨터학부 부교수. 관심분야는 컴포넌트 개발 방법론, 객체지향 개발 방법론, 분산 시스템, 컴포넌트 정형명세, 소프트웨어 시험