재사용을 위한 요소간 결합의 명세 및 호환성 체크

이 창 훈† · 이 경 환†

요 약

객체지향 방법론에서 대부분의 객체들은 독자적인 독립체이지만 시스템 전체적인 측면에서는 서로가 자기에게 주어진 역할을 완수하기 위해서 다른 객체와 상호 협력관계를 가져야만 한다. 이는 객체들의 정적인 측면도 중요하지만 다른 객체와의 interaction을 통한 상호 협력관계의 명세가 중요함을 의미한다. 일반적으로 객체들간의 상호작용을 선이나 박스 등과 같은 비정형적인 형태로 표현함으로써, 추론과 정확성 검증 그리고 호환성 체크와 같은 것이 어렵다는 한계를 갖고 있다. 재사용 전문가들은 설계의 재사용이 보다 일반적이고 광범위하게 적용되기 때문에 코드의 재사용보다 중요하다고 한다. 또한 프레임워크는 일반적으로 코드수준의 재사용과 설계단계의 재사용을 지원하는 재사용 기법으로 알려져 있는데, 여기에서도 각 객체들간의 결합성이 매우 중요한 개념으로 취급되고 있다. 따라서 본 논문에서는 LOTOS를 이용하여 요소들 간의 결합에 관한 표기를 정형화하고 아울러 호환성 여부를 체크 할수 있도록 함으로써 향후 프레임워크로 확장이 가능할 수 있는 토대를 마련하고자 한다.

Specification and Compatiblity Check of the Component compositions for the Reuse

Chang-Hoon Lee† · Kyung-Whan Lee†

ABSTRACT

In the object oriented methods, most of the objects are independent from one another. However to get their job done from the system's point of view, they must have some kind of connection established among them. This means that the cooperation among the objects through the interaction is just as important as the static side of the objects. Usually, checking for correctness, compatibility and reasoning of the objects is limited due to the fact that the interactions between the objects are expressed in the form of a line or a box. The reuse experts often claim that the design reuse is more important than code reuse, mostly because it can be applied in more contexts and so is more common. The composition of the objects is also considered as a very important definition in the area of framework which is generally known as a technique to support reuse at both the coding and the designing level. Therefore on this thesis, the composition of such objects has been studied to provide a formal means of evaluating the component's compatibility and better possibility for further improvement in the area of framework, by formalizing the component compositions using the LOTOS.

1. 서 론

빠른 하드웨어의 개발주기와 이에 따른 사용자의 소프트웨어 요구를 만족시키기 위해 소프트웨어를 하드웨어의 칩처럼 부품화하여 재사용하려는 개념이 나오면서 객체지향 방법론이 재사용의 중요한 해결책으로 대두되었다[1]. 즉, 객체 지향 방법론들의 목적은 소프트웨어의 부품들을 되도록 별도의 수정을 가하지 않으면서도

†정 회 원 : 중앙대학교 컴퓨터 공학과 소프트웨어 공학 연구실
논문접수 : 1997년 12월 3일, 심사완료 : 1998년 3월 9일

효과적으로 재사용할 수 있도록 하는 것이다. 이러한 목적을 달성하기 위해 지금까지의 재사용 방법이라고 하는 것은 객체 클래스를 라이브러리에 저장하여 이를 개발자가 불러다 사용하는 형태가 주로 이용되었다. 따라서 개발자는 자신이 원하는 부품에 대한 이해뿐만 아니라, 만약 제 사용하려는 부품이 없을 경우에는 유사한 부품을 가져다가 요구에 부합되도록 수정을 해야 하는 단계가 필수적으로 필요했다. 이러한 재사용의 방법은 재사용의 단위를 코드 수준에서의 재사용에 초점을 두었기 때문이며, 협력군으로서의 보다 넓은 단위의 재사용을 고려하지 않았기 때문에 발생될 수 밖에 없는 문제들이다[2]. 한편, 재사용 전문가들은 설계의 재사용이 보다 일반적이고 광범위하게 적용되기 때문에 코드의 재사용보다 중요하다고 한다[3]. 이는 개발자가 새로운 시스템을 개발하기 위해 필요한 코드를 재사용하는 방법과, 시스템을 개발하기 위해 필요한 지식을 책을 통해 습득하여 처음부터 시스템을 개발하는 방법과 유사하다고 할 수 있다. 시스템을 구성하고 있는 각각의 객체는 "단독으로 고립된 객체는 존재하지 않는다"[4]는 문장에서도 알수 있듯이 궁극적으로 모든 객체는 결합이라는 방법을 통해 재사용된다는 것이다. 이러한 결합은 어떤 목적을 공동으로 달성하기 위해 발생되며, 이때 작업에 참여하고 있는 객체들의 집합을 협력그룹[5] 이라고 한다. 객체지향 시스템에서 재사용성은 객체 하나 하나에 있는 것이 아니라 공동의 일을 달성하기 위해 서로 상호작용하는 협력그룹의 재사용성에 있다는 것이다[6]. 이러한 시도로써 현재 "프레임워크(Framework)"이라고 하는 재사용 기법에 대한 연구가 활발히 진행되고 있다. 프레임워크는 그 특성상 코드의 부분적 재사용과 설계 수준의 재사용을 지원하는데, 이 역시 쉽게 결합할 수 있는 요소의 제공이 중요하다[3]. 한편 설계의 재사용이 갖고 있는 주요 문제중 하나는 설계정보에 대한 표현과 표준화된 설계표기나 재사용하기 위한 설계분류 방안이 미비하다는 것이다[3]. 따라서 본 논문에서는 요소들에 대한 정형적 표현과 협력그룹의 정형적 표현을 제공함으로써 각 요소들간의 호환성 여부를 체크할 수 있는 토대를 마련하고자 한다.

본 논문의 구성은 다음과 같다: 제 2장에서는 기반연구로써 재사용의 기본개념과 본 연구에서 형식언어로 채택한 LOTOS의 기본연산자에 대하여 기술한다. 제 3장에서는 요소들간의 결합의 의미와 이들을 정형화하여 호환성 여부를 체크하는 방법에 대해 기술한다. 끝으로 제 4장에서는 결론 및 향후 연구과제를 기술한다.

2. 기반연구

2.1 재사용의 단계

소프트웨어 개발 단계에서 재사용은 어느 단계에서나 적용될 수 있다. 그러나 가장 쉽게 적용되는 단계는 설계 단계와 코딩 단계이다. 객체지향 프로그래밍에서 코드의 재사용은 새로운 소프트웨어를 개발할 때, 라이브러리에서 관리되고 있는 클래스의 상속 메카니즘을 통해 재사용하는 것을 의미한다.[7] 여러 클래스들에 의해 공유되는 코드를 공통의 상위 클래스에 두고, 새로운 클래스를 생성할 때 상위 클래스로부터 메소드와 속성을 상속받아 응용 프로그램에서 필요한 클래스를 만들어 낸다. 여기서 상위 클래스로부터 코드를 상속받는 클래스, 즉 하위 클래스는 상속받은 메소드나 속성을 재정의하고 새로운 메소드나 속성을 추가함으로서 사용자 또는 개발자가 원하는 최종의 클래스를 얻을 수 있다. 이러한 클래스의 상속은 사용자가 새로운 클래스를 정의할 때, 사용자가 원하는 클래스와 가장 유사한 클래스를 찾아 이를 상위 클래스로 선택하고 약간의 다른 점만을 하위 클래스에 추가 또는 재정의함으로써 사용자가 최종 클래스를 얻을 수 있는 스타일을 제공한다.

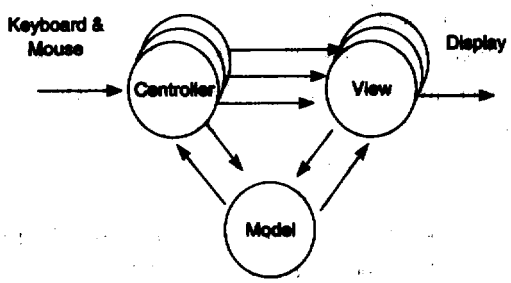
그러나 코딩 단계의 재사용은 프로그램의 원시 코드로써 그 적용 도메인이 한정적이고 코드 자체는 너무나 상세하여 재사용될 확률을 떨어뜨리게 되며, 자세한 구현 사항 정보에 의존하여야 한다는 단점을 갖고 있다. 또한 원시 코드만을 재사용 대상으로 정할 때 소프트웨어의 재사용성과 생산성 향상에 한계를 갖는다.

2.2 재사용의 방법

일반적으로 재사용의 형태는 블랙박스 재사용(blackbox reuse)과, 화이트박스 재사용(whitebox reuse)로 나눌 수 있다[8]. 블랙박스 재사용은 일체의 수정없이 라이브러리에 저장되어 있는 클래스를 사용하는 것을 의미하며, 화이트박스 재사용은 원하는 기능을 가진 클래스를 얻기 위하여 라이브러리 클래스를 상속받아 원하는 형태의 클래스로 만들어 사용하는 것을 의미한다. 개발자의 관점에서 보았을 경우 블랙박스 재사용이 가장 쉬운 방법이다. 그러나 라이브러리 클래스가 한정적이며, 요구되는 라이브러리를 구축하기 위해서는 그 크기가 매우 커지고, 개발자가 원하는 클래스를 찾는 일은 더욱 어려워질 것이다. 반면, 화이트박스 재사

용의 경우는 일반화된 클래스를 가져와 용도에 맞도록 수정하여 사용한다. 그러나 이 경우는 개발자가 클래스의 구현 부분까지도 완전히 이해하여야 한다. 이러한 재사용의 형태는 분석과 설계에 대한 재사용을 기대하기 힘들다. 일반적으로 코드 수준의 재사용보다는 이를 보다 추상화시킨 형태의 분석과 설계 수준에서의 재사용이 전체적인 개발 생산성을 높여준다. 그래서 최근에는 분석과 설계 수준의 재사용과 재사용의 잠재력을 높이기 위해서 프레임워크에 대한 연구가 진행되고 있다. 프레임워크는 객체들과 그들간의 상호작용, 그리고 큰 문제를 어떻게 작은 문제로 나누는 지를 기술함으로써 분석 단계를 재사용할 수 있다. 또한, 프레임워크의 설계 단계에서는 추상화 알고리즘과 인터페이스를 정의하는 방법을 포함하여 설계 단계에서의 재사용을 가능하게 한다[9]. 프레임워크가 이러한 기능을 발휘할 수 있기 위해서는 프레임워크에서 지원하는 요소들간의 결합이 용이 해야 하고, 만약 결합이 가능하다면 이들간의 호환성 여부를 보장해 주어야 한다.

처음으로 널리 사용되었던 프레임워크는 70년대 말에 개발되었던 Model/View/Controller(MVC)라고 불리우는 Smalltalk 80 사용자 인터페이스 프레임워크이다[9]. MVC에서는 사용자 인터페이스를 3종류: 모델, 뷰, 콘트롤러의 구성요소로 나누며 모델은 응용 객체로서 사용자 인터페이스와 독립적인 형태로 존재하며 뷰는 모델의 상태를 일관성있게 유지 관리하는 기능을 갖는다. 한편 콘트롤러는 사용자 이벤트를 받아 이를 모델과 뷰상에서 정의된 오퍼레이션으로 변환하는 기능을 갖는다.



(그림 1) MVC모델에서의 상호작용
(Fig. 1) Interaction of MVC Model

2.3 LOTOS

LOTOS는 1983년 TOS(Temporal Ordering Specification)에서 처음으로 그 이름이 명명되었으며 1988년 ISO로 OSI시스템을 명세할 목적으로 제안되었다. LOTOS는 순차적, 분산 및 concurrent 시스템을 명세하는데 유용한 것으로 알려져 있으며, 크게 자료구조를 표현하는 ADT부분과 시스템의 행위를 기술하는 process부분으로 구성된다. ADT부분은 대수적 명세기법 언어인 ACT ONE[11]을 기반으로 하고 있으며 process부분은 CSP, CCS를 기반으로 하고 있다. LOTOS에서 프로세스를 정의함에 있어 주 요소는 행위식(behaviour expression)이다. 본 연구에서는 각각의 요소(component)와 결합자를 LOTOS의 행위식으로 표현 하였다.

다음 표는 LOTOS에서 사용되는 주요 연산자들에 대한 목록이다:

여기서 a는 임의의 액션을, Bi는 행위식을 그리고 gi는 게이트를, P는 술어논리를 의미한다.

〈표 1〉 LOTOS의 기본 연산자
〈Table 1〉 Basic operator

	명 칭	행위식	의미
기본 행위식	Inaction	stop	deadlock이나 비활성을 의미
	성공적 종결	exit	성공적 종결
	프로세스 인스턴스	ProcName[g_1, . . , g_n]	
기본 연산자	Action Prefix	a; B	액션 a수행후 행위식 B를 수행
	선택	B_1 [] B_2	B_1 또는 B_2를 수행
	Enabling	B_1 >> B_2	B_1 수행후 B_2를 수행
	Disabling	B_1 D B_2	B_1 수행중 B_2에 의해 Interrrupt됨
합 성	병렬합성	B_1 |[g_1, . . . , g_n]| B_2	g_1, . . . , g_n에 의해 동기화가 됨
	Interleaving	B_1 ||| B_2	
	Full synchronization	B_1 || B_2	
기 타	Hiding	hide g_1, . . . , g_n in B	g_1, . . . g_n를 행위식 B안에 감춤(정보의 은닉)
	Guarded Behaviour	[P] → B	P가 참이면 B를 수행
	Local Definition	let x: s = E in B	

3. 구성요소들간의 결합성

구성요소들간의 결합여부는 새로운 시스템을 개발하

고자 할 경우 재사용의 여부를 결정해주는 주요 개념이다. 이를 위해, 호환성 여부를 보장할 수 있는 형태의 표기법과 검증이 필요하다.

3.1 개요

요소(component)란 잘 정의된 채널을[12] 통해 다른 요소들과의 상호작용이 가능한 것을 말하며 넓은 의미에서 객체까지도 포함하는 개념이다. 즉, 어떤 전체 시스템을 구성하고 있는 소프트웨어 부품으로 하나 이상의 객체, 또는 하나 이상의 협력그룹을 말한다. 하나의 단위를 이루고 있는 요소들은 어떤 주어진 "계약(contract)"[5]에 따라 서로 협동하는 부품으로 이루어져 있으며 그 자체로 어떤 목적의 기능을 수행하도록 되어 있다. 여기서 계약이라고 하는 것은 요소들이 새로운 목적을 이루기 위해 결합할 수 있는가에 대한 제약조건이라고 할 수 있다. 즉, 주어진 요소들이 결합에 대한 계약을 수행할 수만 있다면, 이들은 자신에게 주어진 역할(role)를 수행함으로써 계약을 통해 공동의 목적을 수행할 수 있는것이다.

즉, 본 연구에서는 이러한 결합과 관련하여 역할과 계약에 정의를 형식 명세 언어인 LOTOS로 표현하는 방법과 이를 이용하여 결합성의 여부를 체크할 수 있도록 하였다.

3.2 계약의 의미

Helm[5]에 의하면 계약이란 행위적 합성(혹은 결합)을 명확하게 명세하기 위한 것으로 정의한다. 행위적 결합이란 어떤 공동의 작업을 함께 수행하기 위해 요소들간의 협력적 결합과 이들이 지켜야 되는 불변의 조건(invariant)을 의미한다. 즉, 계약이란 협력그룹들이 지켜야하는 불변의 조건과 이들이 행위적으로 결합을 하였을 경우, 무엇을 할 것인가에 대한 명세로 구성된다. 다음은 [5]에서 예를들고 있는 MVC와 관련된 계약에 대한 사례이다.

```
contract SubjectView
  Subject supports{
      value : Value
      SetValue(val: Value) ↦ △value{value = val};
      Notify()
      GetValue(): Value ↦ return value
      Notify() ↦ ⟨|| v: v ∈ Views: v → Update()⟩
      AttachView(v: View) ↦ {v ∈ Views}
      DetachView(v: View) ↦ {v ∉ Views}
      }
 Views: Set(View) where each View supports{
      Update() ↦ Draw()
      Draw()  ↦  Subject  →  GetValue(){View  reflects
          Subject.value}
          SetSubject(s: Subject) ↦ { Subject = s }
          }
    invariant
          Subject.SetValue(val) ↦ ⟨∀v: v ∈ Views : v reflects
          Subject.value⟩
    instantiation
          ⟨|| v: v ∈ Views: ⟨Subject→AttachView(v) || v↦
          SetSubject(Subject)⟩⟩
end contract
```

(그림 2) SubjectView 계약
(Fig. 2) contract

즉, Subject와 뷰가 행위적으로 결합을 하여 공동의 작업을 수행하기 위해서는 위에서 정의된 계약을 준수해야 가능하다는 것을 의미한다. 예를들어, Draw() ↦ Subject → GetValue()는 뷰가 Draw()메소드를 수행하기 위해서는 반드시 Subject에게 GetValue()메소드를 보내야함을 의미한다. 물론, Draw()는 Update()로부터 야기되고 Update()는 Subject에서 각 뷰에게 보내진 메소드이다. 여기서 중요한 사실은 계약에 참여하는 구성요소들이 주어진 계약을 준수할 수 있는가 하는 것이다. 만약 가능하다면 준수 여부를 어떻게 사전에 알 수 있는가 하는 것이다. (그림2)에서 볼 수 있듯이 이 방법이 갖는 다소의 비 정형성으로 인해 사전에 호환성 체크가 어렵다. 이를 위한 기반 개념으로 본 연구에서는 다음과 같이 각 요소들에 대한 관찰가능한 행위를 정의하였다. 다시 말해 계약으로부터 기대되는 행위가 주어졌을 때 그 계약에 참여하고자 하는 각각의 요소는 최소한 그 기대행위를 제공해야 되기 때문이다.

관찰가능한 행위

임의의 행위식 P가 주어졌을 때, 이로부터 얻을 수 있는 관찰가능한 행위는 P로부터 유도되는 모든 가능한 경로들의 집합으로 정의한다.

간단한 예로 다음과 같이 스택을 LOTOS로 표현했을 때 얻을 수 있는 관찰 가능한 행위는 {⟨⟩, ⟨push⟩, ⟨push, pop⟩, ⟨push, push⟩, ⟨push, push, pop⟩, . . .}이지만, ⟨pop, push⟩과 같은 행위는 존재하지 않는다.

```
Stack[push, pop]:=
    push ?x: Integer; (pop !x: Stack[push,
    pop]
```

```
    []
    i: Stack(push, pop))
```

만약 P —e1→ P1 —e2→ P2 —e3→ . . . —ei→Pi = P'를 P ⟹t⟹ P'로 표현 한다면 위의 정의는 "관찰 가능한 행위(P) = {t | ∃P' • P ⟹t⟹ P')"로 표현된다.

결합자와 역할(Role)

결합자는 Helm의 예 에서와 같이, 각각의 요소들을 결합하기 위해서는 이들을 중재해 주는 매체가 필요한데 그러한 기능을 제공 해주는 것을 말한다. 하드웨어에서 각각의 칩을 요소로 본다면 칩과 칩들을 연결해 주는 기판(board)이 곧 결합자가 되는 것이다. 따라서 결합자는 각각의 요소들이 수행하게 되는 역할을 제공해야 할 뿐만 아니라 협력그룹으로서의 요소들간의 상호작용에 대한 논리적 결합을 제공해야 한다. 이러한 목적을 위해 결합자를 다음과 같이 결합자에서 제공될 역할과 이들 역할들이 각 요소에 의해 수행될 때 협력그룹이 형성되도록 해주는 glue들의 집합으로 정의한다:

{ (Ri; G) | ∀i • Ri ∧ G ∈ (R1 ||| R2 ||| ... ||| Rn)}, 여기서 Ri는 배역을 의미
따라서 결합자의 전체적인 의미는 CT = (R1 ||| R2 ||| ... ||| Rn) || G 이다.

이는 각각의 역할들은 Helm이 제시한 계약과 마찬가지로 glue의 전체적 통제하에 있어야 함을 의미한다. ■
다음은 MVC모델을 결합자와 glue의 개념을 이용하여 형식 언어로 표현한 것이다
:

```
결합자 MVC_Model[set_value, invoke, return, result] :noexit:=
role   Model[set_value, notify, invoke, return] :noexit:=
          set_value ? value: Nat_Sort
          ; notify; Model[set_value, notify, invoke, return]
          []
          invoke
          ; return ! value; Model[set_value, notify, invoke, return]
role  View[requset, result] :noexit:=
hide update, draw in
          request
          ;result ? val: Nat_Sort
          ; update; draw; View[request, result]
glue:= M1[set_value, notify] >> M2[req, return]
|[reg]| View[req, result]
end 결합자
```

(그림 3) MVC모델을 위한 결합자
(Fig. 3) Connector

위 예에서 M1[set_value, notify]:= set_value ? value; notify; M1[set_value, notify]이고 M2[req, return]:= req; return ! value; M2[req, return]이다.

3.3 요소와 역할의 호환성

결합자에서 제공되는 역할은 반드시 어떤 요소들에 의해 수행되어야 한다. 이렇게 주어진 역할을 수행하기 위해서는 이들간의 호환성 문제를 보장받아야 한다. 이는 앞서 설명한 관찰가능한 행위는 적어도 역할에서 정의된 기대행위를 만족해야 함을 의미하지만, 기대행위를 만족한다고 해서 호환성이 있다고는 단정할 수 없다. 왜냐하면, 시스템은 현상태에 따라 외부로부터의 이벤트를 무시하는 경우가 발생하기 때문이다. 이렇게 무시되는 행위도 시스템에서는 중요한 인자로 작용되기 때문에 이를 포함할 수 있는 개념이 필요하다. 즉, 호환성 여부를 체크하기 위해서는 시스템의 현상태에 따라 무시되는 이벤트가 무엇이고, 그 경우 시스템의 상태는 어디에 도달해 있는가에 대한 정보도 고려되어야 한다는 것이다.
다음과 같이 아주 단순한 형태의 자판기가 있다고 가정해 보자:

```
VMachine[ ... ]: exit :=
      insert_coin [ in_coin ]
      >>
      accept Total: Coin_Sort in
      [ Total eq 100 ] ↦ ( get_coffee; exit []
                           get_milk; exit []
                           get_cocoa; exit)
      []
      [ Total eq 200 ] ↦ ( get_juice; exit []
                           get_ice_coffee; exit []
                           get_cola; exit)
                           ...
insert_coin[ in_coin ]: exit( Coin_Sort) :=
      in_coin ? coin: Coin_Sort;
                           ...
      exit( any Coin_Sort )
```

(그림 4) 자판기의 예
(Fig. 4) Example

위에 경우에서 볼 수 있듯이 우리가 얻을 수 있는 관찰 가능한 행위는 〈insert_coin, get_coffee〉, 〈insert_coin, get_milk〉, 〈insert_coin, get_juice〉등이 있다. 이렇게 관찰가능한 행위만을 가지고 요소(component)를 아래와 같이 정의한다면, 투입된 액수(시스템의 현 상태)에 관계없이 모든 버튼을 사용자가 사용할 수 있게 되므로 잘못된 결과를 초래하게 된다. 따라서 이 경우에 우리는 호환성이 있다고는 말 할 수 없다.

component Only_Obser_VM[...]: **exit** :=
    in_coin;
    (get_coffee [] get_milk [] get_cocoa []
     get_juice [] get_ice_coffee [] get_cola);
    exit

즉, 관찰가능한 행위(Only_Obser_VM) = 관찰가능한 행위(VMachine)이지만 Only_Obser_VM이 VMachine의 역할을 대신할 수는 없다.

따라서 호환성에 대한 검증을 위해서는 다음과 같이 행위의 거부(refusals)라는 것을 정의할 필요가 있다:

행위거부(B) = { A ⊆ ***alpha***(*E*) | ∃B′ • B = 〈〉⇒ B′ ∧ A ∩ ***initials***(B′) = ∅ }

여기서 B는 행위식을 의미하며 ***alpha***(*E*)는 외부환경으로부터 받아들일 수 있는 이벤트의 집합을 의미하고, ***initials***(P)는 프로세스 P가 현 상태에서 받아들일 수 있는 행위를 의미한다. 즉, ***initials***(P) = { a ∈ ***alpha***(*E*) | ∃B′ • B =〈a〉⇒ B′ } 이다. ■

정의에서 알 수 있듯이, 기본적으로 ∅는 모든 거부행위의 원소이다.

위 예에서 ***alpha***(VMachine) = ***alpha***(Only_Obser_VM) = { in_coin,get_coffee, get_milk, get_cocoa, get_juice, get_ice_coffee, get_cola } 이고, ***initials***(VMachine) = alpha(Only_Obser_VM) = { in_coin } 이다. 따라서 초기 상태에서 거부되는 행위(refusals)는 다음과 같이 계산한다:

행위거부(VMachine) = 행위거부(Only_Obser_VM) = { ***alpha***(*E*) \ { in_coin } }

그러나 in_coin을 수행한 후의 거부되는 행위는 같지 않다:

행위거부(VMachine/〈in_coin〉) = { { in_coin, get_coffee, get_milk, get_cocoa }, { in_coin, get_juice, get_ice_coffee, get_cola } } 그리고

행위거부(Only_Obser_VM/〈in_coin〉) = {{in_coin}}

이를 종합하여 말하면, 요소와 배역의 호환성 문제는 배역에서 관찰가능한 모든 행위를 요소에서도 관찰가능해야 하고, 아울러 도달할 수 있는 모든 상태에서의 거부행위도 포함되어야 한다. 도달할 수 있는 모든 상태에 대한 거부행위들의 집합을 '총체적 거부(*total refusals*)'라고 부른다. 즉, '총체적 거부'라고 하는 것은 관찰 가능한 모든 행위와 이를 통해 도달한 현상태에서의 거부되는 행위를 조합한 결과를 의미한다.

총체적 거부(***total_refs***)

총체적 거부(P) = { (t, A) | t ∈ 관찰가능한 행위(P) ∧ A ∈ 행위거부(P/t) }

P/t: 관찰가능한 행위(*t*)를 수행한후의 상태를 나타냄. ■

위에서 정의된 사실을 이용하여 요소/배역의 호환성 검증을 다음과 같이 정의한다:

호환성(compatibility): f_{assoc} (*n*) ***comp*** *n*

f_{assoc} (*n*) ***comp*** *n* *if and only if:*

1. 관찰가능한 행위(f_{assoc} (*n*)) ⊇ 관찰가능한 행위(*n*)
2. ∀t ∈ 관찰가능한 행위(*n*) • ∀A ∈ P(E) • *if* 〈*t*, *A*〉 ∈ 총체적 거부(f_{assoc} (*n*)) *then* 〈*t*, *A*〉 ∈ 총체적 거부(*n*)

여기서, f_{assoc} : $R \to C$ | ∃ r_1, r_2 ∈ *R* • if r_1 = r_2, then $f_{assoc}(r_1) = f_{assoc}(r_2)$ ∧ dom $f_{assoc} \subset R$이고 *R*은 역할들의 집합, *C*는 요소들의 집합을 의미한다. ■

예를 들어 다음과 같은 요소가 정의되었다고 한다면, 아래 기술을 통해서 (그림3)에서 정의된 결합자와 Component C와의 호환성을 체크할 수 있다.

component C:
    Display[request, result, init]:=
        requset; result; Display[request,
                result, init]
        []
        i; result; exit
        []
        init; Display[request, result, init]

end

f_{same} (View) = Display가 되기 위해서는 위에서 정의한 두 가지 사실을 만족해야 한다. 이때 주의해야 할 점은 각각의 프로세스가 정의된 외부환경이 서로 다를 수 있다는 것이다. 이를 동일하게 하기 위해 각 프로세스의 ***alpha***(P_1)를 ***alpha***(P_1) ∪ (***alpha***(P_2) \ ***alpha***(P_1))로 대치한다.

alpha(View) = ***alpha***(Display) = { request, result, init }

관찰가능한 행위(View)
= { ⟨request⟩, ⟨request, result⟩ }

관찰가능한 행위(Display)
= { ⟨request⟩, ⟨result⟩, ⟨init⟩, ⟨request, result⟩ }

총체적 거부(View)
= { (⟨⟩,∅), (⟨⟩, { result, init }), (⟨request⟩, ∅), (⟨request⟩, { request, init }) (⟨request, result⟩, ∅), (⟨request, result⟩, { result, init }) }

총체적 거부(Display)
= { (⟨⟩, ∅), (⟨request⟩, ∅), (⟨requset⟩, { request, init }), (⟨request, result⟩, ∅), (⟨request, result⟩, { result, init }) }

따라서 관찰가능한 행위(View) ⊆ 관찰가능한 행위(Display)이고 관찰가능한 행위(View)는 $P_{\emptyset}$, $P_{\langle request \rangle}$, $P_{\langle request,\ result \rangle}$로 분할(partition)할 수 있으므로 관찰가능한 행위(View) = { ⟨⟩, ⟨requset⟩, ⟨request, result⟩ }이다.

$\forall t \in$ "관찰가능한 행위(View)"에 대해, 만약 ⟨t, A⟩ ∈ 총체적 거부(Display)이면 ⟨t, A⟩ ∈ 총체적 거부(View)이다.

이는 Dispaly가 View의 역할을 할 수 있음을 보여주고 있다.

3.4 부분적 호환성

소프트웨어의 재사용성을 높이기 위해서는 앞서 언급한 바와 같이 역할과 요소간의 호환성 여부는 매우 중요하다. 그러나 일반적인 환경에서 주어진 역할을 요소가 완벽하게 수행할 수 없다 할지라도 특정 환경에서는 그 역할을 대신할 수 있는 경우가 있다. 요소가 특정 환경하에서 역할을 대신할 수 있는 경우를 '부분적 호환'이라고 부르기로 한다. 여기서 일반적인 환경이란 결합자를 고려하지 않고 역할과 요소만으로 호환성 여부를 체크하는 경우를 말하며, 특정환경이란 결합자에서 제공되는 환경을 고려하여 호환성 여부를 체크하는 것을 말한다. 예를들어 역할에서는 read를 하거나 write를 하도록 역할이 정의되고 요소에서는 오직 write만을 할 수 있다고 가정하면 일반적인 환경하에서는 당연히 요소가 그 역할을 대신할 수 없다. 그러나 결합자에서 오직 write만을 필요로 한다면 요소는 그 역할을 대신할 수도 있을 것이다. 이러한 부분적 호환성을 허용함으로써, 보다 유통성 있는 재사용을 제공할 수 있다.

다음은 제한된 환경에서 호환성 체크를 위해, 주어진 역할에 결합자의 환경을 반영하기 위한 정의이다.

제한된 배역(restricted role)
$\exists i \cdot H$ = ***alpha***(R_i) ∩ ***alpha***(CT)
rstR := **hide** ***alpha***(R_i) \ H **in** Ri ■

예로, 다음과 같은 역할과 요소가 정의 되었다고 가정하자:

role R_i := write: R_i [] read: R_i [] close: **exit**
role R_j := open: initial: ...
component P := write: P [] **exit**
glue CT := write: **exit** [] open: initial: CT

이를 "제한된 배역"에 적용하면:
H = ***alpha***(R_i) ∩ ***alpha***(CT) = { write } 이고,
rstR := **hide** ***alpha***(R_i) \ H **in** R_i를 이용하면 새롭게 제한된 **role** R_i는
rstR := write: R_i [] i: R_i [] i: **exit** 이다.

따라서 요소 P는 제한된 환경하에서는 역할 ***rstR***의 기능을 대신할 수 있으며 이렇게 제한된 호환성을 통하여 부분적인 재사용성을 보장받을 수 있다.

4. 결론 및 향후 연구과제

객체지향 시스템에서 요소들간의 결합문제는 설계단계에서의 소프트웨어 재사용성을 증가시키는 중요한 부분이다. 요소들간의 결합에 관한 연구는 R. Helm의 계약(contract)[5], Oscar Nierstrasz의 스크립트[13], Kurki-Suonio, R.의 조인트 액션 시스템(joint action system)등 다양하게 이루어지고 있다. 그러나 지금까지의 연구에서는 요소들간의 결합을 다소 비정형적인 형태로 표현함으로써 차후 이들에 대한 확인, 검증 작업이 어렵다는 단점을 안고 있다. 따라서 본 논문에서는 이러한 연구의 한 부분으로 요소들간의 결합을 정형화함으로써 차후 결합에 대한 추론이나 검증, 그리고 재사용성의 체크를 설계단계에서 이루어질 수 있도록 기반을 제공하도록 하였다. 기존의 정형화 명세언어는 ADT를 정확하게 명세할 수 있는 대수적 명세언어가 주로 사용되었다. 그러나 대수적 명세언어는 정적인 측면을 표현하는데는 유용하지만, 요소들의 결합과 같이 interaction을 표현하는데는 그 한계가 있다. 동적인 측면을 표현하기 위해 Propositional Temporal Logic, CSP, CCS등과 같은 여러 가지 명세언어가 있지만 이들 또한 나름대로 단점을 가지고 있다. 본 연구에서는 정적인 측면과 시간적 측면을 지원하는 프로세스 대수(process algebras)기반의 LOTOS를 명세언어로 선택하였다. 이는 또한 ISO표준으로 향후 많은 사람들이 표준화된 방법으로 사용할 수 있다는 장점을 갖고 있기 때문이다.

향후 연구과제로는 객체지향적 기법을 이용한 컴포넌트 기반의 개발방법이 요구된다. 즉, 주어진 문제를 객체중심의 분석에서 탈피하여 객체들간의 협력그룹을 중심으로 분석하여 객체 각각이 보다 다양한 역할을 수행할 수 있도록하는 것이다. 한편, 프레임워크는 코드 수준의 재사용과 설계단계의 재사용을 지원하는 재사용 기법으로 알려져 있다. 프레임워크에서 이러한 장점을 제공하기 위해서는 이들이 지원하는 요소들간의 결합이 얼마만큼 쉽게 이루어 질 수 있느냐가 매우 중요하기 때문에 프레임워크 개발자가 설계한 요소들간의 결합여부를 분석 및 설계단계에서 결정할 수 있도록 본 연구를 확장 하여야 할 것이다.

참 고 문 헌

[1] 이경환, "소프트웨어 재사용을 위한 객체모델링 기법," 교학사, 1993.

[2] Massiomo D'Alessandro, "The Generic Reusable Component : An Approach to Reuse Hierarchical OO Design," Proc. of 2nd International Workshop on [illegible], 1991.

[3] Don Roberts and Ralph E. Johnson, "Evolving Frameworks : A Pattern Language for Developing Frameworks," Addison-Wesley, Reading, Massachusetts, 1997.

[4] K. Beck and W. Cunningham, " A Laboratory for Teaching Object-Oriented Thinking," Proceedings of OOPSLA 1989.

[5] R. Helm, I. Holland, and D. Gangopadhyay, "Contracts : Specifying Behavioural Composi-tions in Object- Orinted Systems," Proceedings of OOPSLA/ ECOOP 1990.

[6] Mary Shaw, "Architectural issues in Software Reuse," Proc IEEE Symposium on Software Reusability, April 1995

[7] Gerhard Fischer, "Cognitive view of reuse and redesign," IEEE Software, pp.60-72, Vol.4, Number 4, July 1987.

[8] Mark Lattanzi, "Who Reuses What? C++ Libraries and Programmer's Traits," http : //www.cs.vt.edu/reports/94/tr-94-18.ps.Z

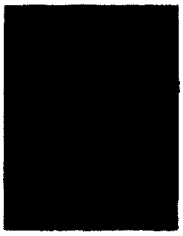
[9] Ralph E. Johnson, Vincent F. Russo, "Reusing Object-Oriented Designs," Univ. of Illinois TR UIUCDCS 91-1696, 1991.

[10] Wilf R. LaLonde and John R. Pugh, "Inside Smalltalk," Vol. 2, Prentice Hall, 1991.

[11] B. Ehrig and B. Mahr, Fundamentals of Algebraic Specifications, Springer-Verlag, 1985.

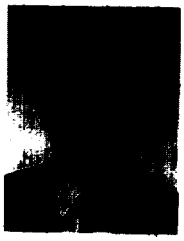
[12] David Garlan and Mary Shaw, "An Introduction to Software Architecture," CMU Technical Report, January 1994.

[13] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey, "Objects + Scripts = Applications," Proceedings, Esprit 1991 Conference, pp. 534-552, 1991.



이 창 훈

1987년 광운대학교 전자계산학과 이학사
1989년 중앙대학교 전자계산학과 이학 석사
1994년~현재 중앙대학교 컴퓨터 공학과 박사 수료
관심분야 : 소프트웨어 공학, 재사용, 형식 명세 기법, 객체지향 방법론, 프레임워크, 멀티미디어, 컴포넌트 개발방법



이 경 환

1980년 중앙대학교 대학원 응용수학 전공(이학박사)
1982년~1983년 미국 Auburn대학 객원교수
1986년 서독 Bonn대학 객원교수
1971년~현재 중앙대학교 컴퓨터 공학과 교수
관심분야 : 소프트웨어 공학, 객체모델링, 재사용 기법, 품질보증, 소프트웨어 표준화