컴포넌트 상호작용 명세기법

이 창 훈†

요 약

컴포넌트 기반 플랫폼이 갖는 주 한계점은 컴포넌트 기반 시스템이 갖는 컴포넌트간 상호작용과 관련된 행위에 대한 기술과 추론에 대한 적당한 수단을 제대로 제공하지 못한다는 것이다. 현 컴포넌트 기반 플래폼이 CORBA의 IDL과 같은 컴포넌트에 대한 시그네처 수준에서의 기술을 위한 방법을 제공하기는 하지만, 이는 컴포넌트들간 상호작용과 같은 행위 기술을 저 수준에서 제공하는 것에 불과하다. 컴포넌트 기반 시스템에서 중요한 과제 중 하나는 바로 컴포넌트들간 상호작용과 같은 행위를 기술하는 것과 이를 바탕으로 행위에 대한 검증이 필요하다. 본 논문에서는 프로세스 대수를 이용하여 컴포넌트 인터페이스의 명세 정의와 추상화된 소프트웨어 아키텍처를 어떻게 만족시킬 것인가를 보장받기위한 role의 결합방법, 생성된 아키텍처에 대한 전개규칙, 시각화방법을 연구한다. 또한 사례연구를 통해 본 논문에서 제안한 명세기법 및 정의의 타당함을 보였다.

The Specification Techniques of Component Interactions

Chang Hoon Lee††

ABSTRACT

A major limitation of available component-based platforms is that they do not provide suitable means for describing and reasoning on the concurrent behaviour of interacting component-based system. Indeed while these platforms provide convenient ways to describe the typed signatures of components, e.g. like CORBA's IDL, they offer a quite low-level support to describe the concurrent behaviour of component. The ability to describe and verify the concurrent behaviour of interacting components is key aspect in the development of large component-based software system. This study propose a component interface specification using process algebra and configuration's role which allows one to prove correctness of software architecture generated at design level as well as to define compatibility relations by our evolution rule and π-graph. Also, we shown on an appropriateness of a specification techniques and definitions proposed in this paper by case-study.

키워드 : 컴포넌트(Component), 인터페이스(Interface), 아키텍처(Architecture)

1. 서 론

컴포넌트는 소프트웨어 개발을 조립의 개념으로 발전시켰으며 외부에서 개발된 컴포넌트를 조립하여 쉽게 소프트웨어를 개발할 수 있는 구조를 제공한다. 조립에 의한 소프트웨어의 개발은 도메인의 업무에 맞는 다른 컴포넌트로 교체하거나 컴포넌트를 추가하여 도메인의 요구사항을 빠르고 쉽게 충족시킬 수 있다[1].

CBD(Component Based Development)의 정의는 한마디로 정의하기가 어렵지만, 일반적으로 재사용 가능한 소프트웨어 컴포넌트를 생성 및 조립생산, 선택, 평가 및 통합의 과정을 통하여 더 큰 컴포넌트를 생성하거나 완성된 어플리케이션 소프트웨어를 구축하는 개발기법이다. 즉, 이미 만들어진 컴포넌트를 기본 아키텍처와 설계도에 따라 조립하는 방식의 새로운 개발 형태로서, 컴포넌트의 오퍼레이션 및 상호 오퍼레이션을 정의하는 명세의 개발, 객체나 컴포넌트들로부터 컴포넌트의 구축, 컴포넌트들을 이용한 어플리케이션의 조립 등 세 가지 활동을 필요로 한다. 컴포넌트는 어플리케이션을 개발하기 위한 기본단위로 컴포넌트 이용자들에게 컴포넌트 인터페이스(Component Interface)와 컴포넌트 사양(Specification)을 제공한다. 또한 컴포넌트 기반 소프트웨어 시스템을 개발함에 있어 컴포넌트에 대한 상호교류적(Interaction) 행위의 기술(Description)과 검증은 매우 중요한 요소이다. 따라서 이러한 소프트웨어 개발환경에서는 주어진 컴포넌트가 특정 어플리케이션의 다른 컴포넌트와 대체성(replacement) 검사와 같은 컴포넌트들간의 호환성 문제가 컴포넌트의 재사용성을 높이기 위한 기법으로 평가되고 있다[9]. 일반적으로 컴포넌트들간의 호환성 검사는 시

† 종신회원 : 한경대학교 컴퓨터공학과 교수
논문접수 : 2003년 12월 22일, 심사완료 : 2004년 4월 29일

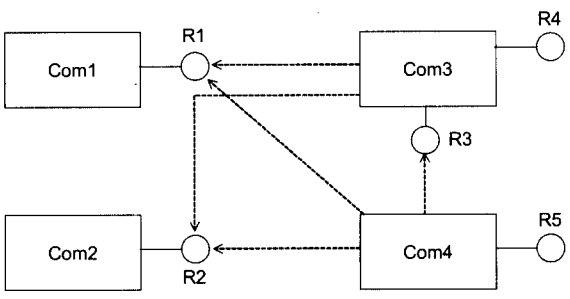
그네처(Signature), 프로토콜(Protocol), 그리고 의미(Semantic) 수준에서 이루어지고 있다[3]. 특히 시그네처 수준의 컴포넌트 명세는 IDL(Interface Description Language)을 통해, 그리고 의미 수준의 명세는 오퍼레이션의 동작에 따른 각종 제약사항을 표현함으로써 주로 Pre/Post나 UML의 OCL 등으로 명세된다. 그러나 컴포넌트의 이러한 두 가지 명세 기법은 CBD의 원래 목적인 서로 다른 컴포넌트들간의 조립을 통한 어플리케이션의 구축이라는 측면을 충분히 만족시키지 못한다. 일반적으로 주어진 컴포넌트들로부터 원하는 어플리케이션을 구축하기 위해서는 우선 설계된 소프트웨어 아키텍처를 기반으로 컴포넌트 저장소에서 해당 컴포넌트를 선택하고 이들을 이용하여 실제 구축할 아키텍처를 재구성하는 일련의 작업이 요구된다.

① 어플리케이션의 아키텍처 결정
② 이를 이용한 후보 컴포넌트 선별
③ 실제 어플리케이션 구축

가령 설계단계에서 (그림 1)과 같은 컴포넌트들과 아키텍처가 생성된 경우, 실제 이에 해당되는 어플리케이션을 구축하기 위해서는 컴포넌트 저장소로부터 해당 컴포넌트를 선별하는 작업이 요구된다.

$Com_1 = \{R_1\}$, $Com_2 = \{R_2\}$, $Com_3 = \{R_3, R_4, R'_1, R'_2\}$,
$Com_4 = \{R_5, R'_1, R'_2, R'_3\}$

여기서 R_i는 컴포넌트가 제공하는 인터페이스이고, R'_i는 컴포넌트가 필요로 하는 인터페이스를 의미한다.



(그림 1) 어플리케이션의 아키텍처

컴포넌트 저장소로부터 다음과 같은 후보 컴포넌트가 선택되었다고 가정하자.

$C_1 = \{R_1\}$
$C_2 = \{R_2\}$
$C_3 = \{R_1, R_3, R'_2\}$
$C_4 = \{R_4\}$
$C_5 = \{R_3, R_5, R'_1\}$
$C_6 = \{R_2, R_4, R'_6\}$

이 6개의 후보 컴포넌트들로부터 얻을 수 있는 재구성 가능한 아키텍처는 이론적으로는 2^6개가 존재한다. 그러나 이들 중 몇몇은 가령, C_1, C_2, C_3, C_4와의 조합과 같이 의도한 기능을 제공 못하는 조합도 존재한다. 또한 C_2, C_3, C_4, C_5와 같은 조합은 비록 R_3의 기능이 C_3와 C_5에서 중복되지만 의도한 기능을 제공한다.

따라서 본 논문에서는 이러한 일련의 작업이 진행되기 위해 컴포넌트간 상호작용 명세기법을 통해, 설계단계에서 결정된 아키텍처에 대한 올바른 검증과 이를 토대로 컴포넌트 저장소로부터 후보 컴포넌트 선별 시 설계 컴포넌트와의 대체성을 보장받을 수 있는 방법을 제안한다.

본 논문의 구성은 다음과 같다. 2장은 컴포넌트 인터페이스를 기존 방법에서는 어떻게 명세하였는지를 알아보고 3장은 프로세스 대수에 기본 개념과 이를 이용한 컴포넌트 인터페이스 명세의 정의, 대체성 검증 방법, 그리고 구성된 아키텍처에 대한 검증방법을 알아본다. 마지막으로 4장에서는 결론을 제시한다.

2. 관련 연구

[컴포넌트와 인터페이스]

인터페이스는 컴포넌트의 구현과 컴포넌트의 연결을 분리한다. 컴포넌트의 구현은 인터페이스에만 의존한다. 즉, 인터페이스는 컴포넌트에 의해 실제화(Realization)할 기능에 관한 설명이다. 컴포넌트는 자신과 상호작용할 다른 컴포넌트들의 구현에 의존하지 않고 자신에게 정의된 인터페이스들만을 참조하여 이들을 준수하도록 구현된다. 물론, 컴포넌트들간의 연결관계도 인터페이스에만 의존하여 정의된다. 컴포넌트는 시스템에 독립적으로 특정 기능에 대한 잘 정의된 여러 인터페이스를 제공하는 모듈이지만, 실제로 시스템을 구축할 때에는 해당 컴포넌트가 시스템에서 어떻게 구성되고 사용되는지에 대한 정보는 시스템마다 다르게 된다. 이러한 이유로 컴포넌트 시스템에서의 인터페이스 명세는 시스템에서 사용되는 컴포넌트의 역할이 무엇인지, 또 어떻게 컴포넌트와 상호작용이 이루어지는지에 대한 명세를 정의해야 한다[10]. 이는 인터페이스의 명세를 단순히 오퍼레이션들의 시그네처 명세로는 불충분하다는 의미이다. 컴포넌트 명세의 확장으로 계약(Contract)을 사용하기도 한다[2, 3]. [3]에서는 컴포넌트 명세에 포함될 성질들을 컴포넌트들 사이의 협상 가능 정도에 따라 4가지 레벨로 분류하였으며 레벨이 높을수록 컴포넌트의 변경 및 조립정보의 조절(협상)이 좋아진다. 각 레벨별 종류와 의미는 다음과 같다.

• 문법 혹은 시그네처 계약

컴포넌트가 실행가능한 오퍼레이션, 해당 입출력 매개변수, 수행 중 발생할 수 있는 예외 등을 기술하는 레벨로서 IDL이 주로 이 범주에 속하며 매개변수의 타입 검사(Type Check)을 통해 컴포넌트간 호환성을 보장받는다. DCOM이나 CORBA와 같은 대부분의 컴포넌트 모델은 이 레벨의 계약을 포함하고 있다. 그러나 이 레벨에서 제공하는 정보는 효율적인 소프트웨어의 재사용을 보장받기에는 충분치 못하다. 독립적으로 개발된 각각의 컴포넌트들에서 명세된 오퍼레이션 및 매개변수들이 동일한 이름을 사용할 것이라는 것을 가정할 수 없기 때문이다.

```
interface ITextModel{
    int length( ) ;
    char charAt(in int pos) ;
    void deleteCharAt(in int pos) ;
    ...
    void register(in ITextObserver o) ;
} ;
```

(그림 2) 시그네처 계약에 의한 인터페이스 명세 예

• 행위 계약

행위적 계약에서 명세되는 주된 내용은 한 컴포넌트가 다른 컴포넌트에 대해 기대하는 행위를 해당 컴포넌트가 충분히 그 행위를 제공하는가를 표현한다. 이와 관련해 많은 방법들이 제안되었고, 각각의 방법들은 주어진 특징에 따라 전/사후(Pre/Post) 조건, 시제논리(Temporal Logic)[11], 페트리넷(Perti Net)등이 이 범주의 명세를 위해 사용된다.

```
interface ITextModel{
    int length( ) ;
    //@ pre : true
    //@ post : result == len(text)
    char charAt(in int pos) ;
    //@ pre : 0 <= pos && pos < len(text)
    //@ post : result == text[pos]
    void deleteCharAt(in int pos) ;
    //@ pre : 0 <= pos && pos < len(text)
    //@ post : \forall i [0..pos-1] : test'[i] == text[i] &&
    //@          \forall i [pos..len(text)-1] : test'[i] == text[i+1] &&
                 len(text') == len(text)-1
    ...
} ;
```

(그림 3) 행위 계약에 의한 인터페이스 명세 예

전/사후 조건에 의한 명세의 단점은 컴포넌트로부터 야기되는 외부호출(External Call)에 대한 정보는 표현하지 못하고 단지 어떻게 컴포넌트의 상태가 변경되는가에 대한 정보만을 표현하고 있다. 이를 극복하고자 [5]에서는 전/사후 조건을 기반으로 불변조건(Invariants), Assertion등을 이용한 grey-box 컴포넌트 명세를 제안했다.

• 동기화 계약

이 범주에 속하는 명세는 Allen과 Garlan그리고 Nierstraz의 연구에 기초하여 처음으로 Yellin과 Strom[6]에 의해 유한상태머신(Finite State Machines)을 이용하여 구체화 되었으며 오퍼레이션이 호출되거나 호출되어질 순서에 대한 계약이다. 즉, 두 컴포넌트들 간의 상호 교류적 행위가 일치하는가를 검증할 수 있는 토대를 기반으로 컴포넌트의 명세가 이루어진다.

```
protocol ITextModel{
    ITextModel(ref, observers) =
        ref?length(rep).(^v)rep(v).ITextModel(ref, observes)
    + ref?charAt(p, rep).(^c) rep!(c).ITextModel(ref, observes)
    + ref?deleteCharAt(p, rep).
        ( [observes != <>] Notify(ref, observes, p, rep)
        + [observes  = <>] rep!( ).ITextModel(ref, observes))
    ...
    Notify(ref, observes, p, rep) =
        [observes = <b>++others] b!deleteNotif(p).b?( ).Notify(ref,
        others, p, rep)
    + [observes = <>] rep!( ).ITextModel(ref, observes)
} ;
```

(그림 4) 동기화 계약에 의한 컴포넌트의 명세 예

• 서비스 품질 계약

최대 응답 지연시간, 평균 응답시간, 응답 결과의 질, 데이터 스트림의 처리양등 주요 네트워크 서비스 관련 컴포넌트 및 서버 컴포넌트의 서비스 품질을 명세하는 계약이다.

3. 컴포넌트 인터페이스

3.1 행위식

앞서 언급한 바와 같이 컴포넌트는 시스템에 독립적으로 특정 기능에 대한 잘 정의된 여러 인터페이스를 구현한 모듈이다. 또한 컴포넌트 시스템에서의 인터페이스 명세는 시스템에서 사용되는 컴포넌트의 역할이 무엇인지, 또 어떻게 컴포넌트와 상호작용이 이루어지는지에 대한 명세를 정의해야 한다. 이를 위해 본 논문에서는 Milner에 의해 제안된 프로세스 대수(Process Algebra)의 일종인 π-calculus[7]를 사용한다.

컴포넌트의 상호교류적 행위를 행위식(behaviour expression)으로 명세하며 행위식은 다음과 같은 구문을 갖는다.

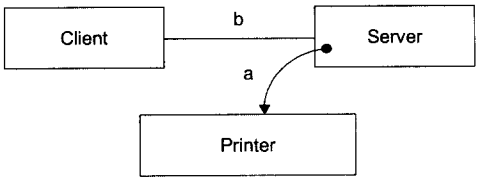
E:: = **0** | a.E | (x)E | [x=y]E | E|E | E+E

a: = τ | x?(d) | x!<d>

여기서 프로세스 **0**은 무활동의(Inaction) 특수한 프로세스를 의미하며 τ는 컴포넌트가 외부환경과는 무관하게 독립적으로 무언가를 수행하는 경우 주로 사용되는 내부액션(Internal Action)이다.

입출력과 관련된 액션은 각각 x?(d)와 x!<d>로 표현되며 이때 x를 통신채널명 혹은 링크(Link)하고 하며 d는 x를 통해 전달되는 링크 혹은 자료들의 튜플(Tuple)을 의미한다. 각각의 액션들은 병렬(Parallel : |)과 비결정적선택(Non-Deterministic Choice : +) 연산자에 의해 합성이 가능하다. 예를 들어, 행위식 x!<a>.S | x?(c).c!<d>.P 은 S | a!<d>.P로 전개(Evolve)된다.

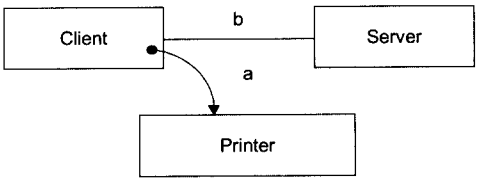
다음과 같은 클라이언트-서버형태의 프린터 시스템이 주어졌을 때, 클라이언트가 프린터에 5라는 값을 출력하려고 한다.



(그림 5) C/S 프린터 시스템 I

만약 서버만이 프린터에 접근이 가능하다고 가정하면 이들 각각의 행위식은 다음과 같이 표현된다.

$$C = b!<5>.C'$$
$$S = b?(x).a!<x>.S'$$
$$P = a?(y).P'$$



(그림 6) C/S 프린터 시스템 II

그러나 클라이언트가 서버의 중재를 거치지 않고 그 제어권을 받아 직접 출력하는 경우라면 그 행위식은 다음과 같이 표현된다.

$$C = b?(x).x!<5>.C'$$
$$S = b!<a>.S'$$
$$P = a?<y>.P'$$

두 경우 모두 어떤값이 출력되기 위해서는 시스템을 구성하는 컴포넌트들간의 상호작용에 의해 이루어지며 이는 각각의 컴포넌트를 합성한 형태인 다음과 같은 행위식으로 표현된다 :

$$PRINTER\_SYSTEM \triangleq C \mid S \mid P = b?(x).x!<5>.C' \mid b!<a>.S' \mid a?<y>P'$$

이 예는 또한 프로세스 대수가 시스템의 동적인 측면과 시간에 따라 진화되는 측면을 표현함에 적합함을 보여주는 예 이기도 하다.

3.2 컴포넌트 인터페이스

컴포넌트 인터페이스는 role들에 의해 표현된다[4]. role은 일종의 다른 컴포넌트와의 교류에 대한 추상적 기술이므로 컴포넌트의 인터페이스는 이들 role의 집합으로 정의될 수 있다.

role에 대한 명세형식과 의미는 다음과 같다.

```
role roleName(listOfopenName)={
        behaviour behaviour expression
}
```

[정의 1] 프로세스 P가 다음을 만족할 경우 이를 컴포넌트, C_p,의 role이라고 한다.

$$fn(P) \subseteq fn(C_p) \wedge P \approx C_p / (fn(C_p) - fn(P))$$

$fn(P)$은 일종의 다른 컴포넌트와의 상호작용을 위한 채널을 의미한다.

간단한 클라이언트/서버 시스템과 관련된 컴포넌트의 명세는 다음과 같다. 물론, 이 경우에는 컴포넌트의 role이 각각 하나씩 갖는 예이다.

```
role Client(c) = {
        c!<request>.(c?(reply).0 + τ.c!<error>.0)
}
role Server(s) = {
        s?(request).(s!<reply>.0 + s?(error).0)
}
```

이 경우 fn(Client)은 c이고 fn(Server)은 s이다.

한편, 컴포넌트를 구성하고 있는 각각의 role들이 다음식을 만족할 때 주어진 컴포넌트의 행위를 올바르게 명세했다고 말한다($C_p \cong \{P_1, \cdots, P_n\}$).

① $fn(P_i) \cap fn(P_j) = \varnothing \quad \forall i, j = 1 \cdots n, i \neq j$

② $C_p \equiv \mathbf{0} \Leftrightarrow \forall i \; P_i \equiv \mathbf{0}$

③ 만약 $\exists \alpha. \; C_p \Rightarrow \alpha \Rightarrow C'_p$, 이면 $\exists i, P'_i. \; P_i \Rightarrow \alpha \Rightarrow P'_i \wedge C_p \cong \{P_1, \cdots, P'_i, \cdots, P_n\}$

여기서 P_i는 컴포넌트를 구성하는 각각의 role들의 집합을 의미한다.

또한 이식은 컴포넌트의 인터페이스를 role(P_i)들의 집합으로 충분히 표현됨을 보여준다.

가령, 하나의 파일을 블록단위로 읽고 이를 다른이름의 파일로 복사하는 컴포넌트가 다음과 같다고 한다면,

Reader = read?(block).([x != EOF] fwrite!(block).Reader
+ [x = EOF] fclose!( ).**0**) + τ.break!( ).**0**

이 컴포넌트의 행위를 만족하는 role은 다음과 같이 정의될 수 있다.

role R1(read, break) = {
read?(block).**0** + τ.break!( ).**0**
}
role R2(fwrite, fclose) = {
τ.fwrite!<block>.**0** + τ.fclose!( ).**0**
}

[정의 2] role간의 결합(*Cfg*)

Cfg = ($\cup_i$ *fn*(R_i))(Π_i R_i)로 정의한다.

R_i : role의 집합을 의미한다.

그리고 초기 *Cfg*는 임의의 R_i라 가정하면, *Cfg*은 다음과 같이 *con*( )연산자를 사용하여 표현할 수 있다.

Cfg' = *con*(R', σ, *Cfg*)로 정의되며 이때 *con*(R', σ, *Cfg*) = R'σ | *Cfg*σ이고, σ은 substitution을, R'는 새로이 결합되는 role을 의미한다.

클라이언트-서버 프린터의 예에서

C(b) = b!<5>.C'
S(a, f) = f?(x).a!<x>.S'
P(a) = a?(y).P'

초기 *Cfg*를 C(b)라고 한다면, *Cfg'*는 C(b)와 S(a, f)와의 결합인 C(b) | S(a, f)가 된다. 그러나 클라이언트 C와 서버와의 상호작용을 위해 일치된 이름이 존재하지 않으므로 사실상 결합이 불가능하다. 따라서 σ함수를 {z/b, z/f}로 하면 *Cfg'* = C(z) | S(a, z)이다. 즉, *Cfg'* = z!<5>.C' | z?(x).a!<x>.S'

[정의 3] 컴포넌트의 대체성

하나의 컴포넌트(P)가 다른 컴포넌트(R)와 대체가능하기 위해서는 다음과 같은 조건을 만족하여야 한다.

① 만약 P $\tau\rightarrow$ P'가 존재한다면 P'와 R과 대체가능 하여야 한다.
② 만약 R $\alpha\rightarrow$ R'($\alpha\neq\tau$)라면 $\exists$P'. P $\alpha\Rightarrow$ P'가 존재하고 P'와 R'가 대체가능 하여야 한다.
③ 만약 P $\alpha\rightarrow$ P'($\alpha\neq\tau$)이고 R $\alpha\Rightarrow$ R'면 P'와 R'가 대체 가능하여야 한다.

가령, R_1 = a?(x).**0** + b?(y).**0**, R_2 = τ.a!<u>.**0** + τ.b!<w>.**0**의 경우 이 둘 간의 상호교류는 원활하게 이루어질 수 있으므로 대체성을 갖는다. 그러나 R_1대신 R_3 = τ.a?(x).**0** + τ.b?(y).**0**을 적용시키면 R_2와 R_3간의 원활한 상호교류를 보장을 받지 못하므로 대체성을 갖지 못한다. 또한 대체성 문제와 관련하여 고려되어야할 점은 주어진 아키텍처가 각 컴포넌트들간의 정보교류를 원활히 지원하는가이다. 만약 정보의 교류가 제대로 이루어지지 않는다면 주어진 아키텍처의 구성이 잘못된 경우이다. 이를 검증하기 위한 방법으로 다음과 같은 전개규칙을 정의한다.

[정의 4] 전개 규칙(Evolution Rule) :

E1. 만약 P $\tau\rightarrow$ P', (P | $Cfg_{/P}$) $\rightarrow$ (P' | $Cfg_{/P}$)
E2. 만약 $\exists$P', Q'. P x?(z)$\Rightarrow$ P' $\wedge$ Q x!<z>$\Rightarrow$ Q', (P | Q | $Cfg_{/PQ}$) $\tau\rightarrow$ (P' | Q' | $Cfg_{/PQ}$)
여기서 $Cfg_{/P}$는 *Cfg* - P를 의미한다.

한 예로, 교차로 시스템(Crossing System)을 구성하는 각각의 컴포넌트가 다음과 같을 경우, [정의 2]를 적용하여 시스템의 아키텍처를 얻는다.

Crossing(light, gate) = light!<green>.light?(x).**0** + gate!<up>.gate?(y).**0**
Road(g) = car.g?(dir).cpass.g!<down>.**0**
Rail(l) = train.l?(signal).tpass.l!<red>.**0**
Crossing System = ch_1!<green>.ch_1?(x).**0** + ch_2!<up>.ch_2?(y).**0** | car.ch_2?(dir).cpass.ch_2!<down>.**0** | train.ch_1?(signal).tpass.ch_1!<red>.**0**

그리고 이 아키텍처에 [정의 4]를 적용하면,

ch_1!<green>.ch_1?(x).**0** + ch_2!<up>.ch_2?(y).**0** | car.ch_2?(dir).cpass.ch_2!<down>.**0** | train.ch_1?(signal).tpass.ch_1!<red>.**0** $\tau\rightarrow$ ch_2?(y).**0** | cpass.ch_2!<down>.**0** | train.ch_1?(signal).tpass.ch_1!<red>.**0** $\tau\rightarrow$ **0** | train.ch_1?(signal).tpass.ch_1!<red>.**0** 또는

ch_1!<green>.ch_1?(x).**0** + ch_2!<up>.ch_2?(y).**0** | car.ch_2?(dir).cpass.ch_2!<down>.**0** | train.ch_1?(signal).tpass.ch_1!<red>.**0** $\tau\rightarrow$ ch_1?(x).**0** | car.ch_2?(dir).cpass.ch_2!<down>.**0** | tpass.ch_1!<red>.**0** $\tau\rightarrow$ **0** | car.ch_2?(dir).cpass.ch_2!<down>.**0**

이로써 교차로 시스템을 구성하고 있는 각각의 컴포넌트들이 교차로의 상태에따라 한번은 자동차가 그리고 또 한번은 기차가 충돌없이 통과됨을 볼 수 있다.

[정의 5] 닫힌 아키텍처

다음과 같은 조건을 만족하는 *Cfg'*가 존재하면 개방된 아키텍처라고 정의한다.

$$\exists Cfg'.\ Cfg \Rightarrow Cfg',\ Cfg' \tau\not\rightarrow \wedge\ Cfg' \neq \mathbf{0}$$

반대로 *Cfg*가 위 조건을 만족하지 않는 경우를 닫힌(Closed) 아키텍처라 부른다.

[정의 5]는 각 컴포넌트들간의 상호 교류가 실패없이 원활하게 교환됨을 보여주는 것으로 이는 주어진 컴포넌트들로 구성된 아키텍처가 제대로 구성되었음을 의미한다.

3.3 전개의 시각화

컴포넌트들과 이들의 관련성을 정의하는 아키텍처의 전개를 통해 주어진 아키텍처가 닫힌 아키텍처임을 보장하는 것이 필요함을 보았다. 본 절에서는 아키텍처의 전개과정을 시각화하여 설계자로 하여금 이해도를 배가시키는 도구로 π-그래프를 제시한다.

[정의 6] π-그래프

G = (V, E) 다음을 만족하는 방향성 그래프이다.

V(G) : name들의 집합

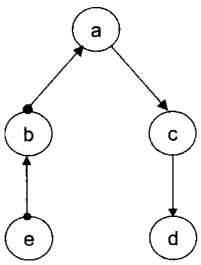
E(G) : 다음과 같은 두 종류의 간선(Edge)들의 집합.

① output edge : object → subject(channel) ●——▶

② input : subject(channel) → object ——▶

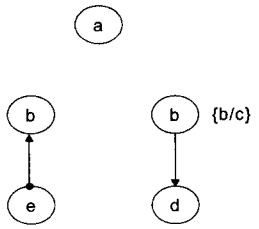
이때 입력과 출력 간선을 동시에 갖는 노드를 싱크(Sync)노드라 부르며 이 노드를 통해 두 프로세스간의 상호교류가 이루어진다. 즉, 출력간선에 위치한 노드의 name이 싱크노드를 통해 입력간선에 위치한 노드의 name과 치환이 이루어진다.

예를 들어, (a?(c).c?(d) | a!<b> | b!<e>)를 그래프로 표현하면 다음과 같다.



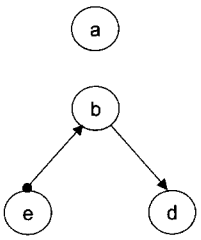
(그림 7) π-그래프의 예

(그림 7)에서 싱크노드는 a이다. 따라서 출력간선과 연관된 b가 싱크노드 a를 통해 입력간선과 연관된 노드 c와 치환이 이루어진다.



(그림 8) 전개 후 π-그래프

싱크노드 a를 통해 전개가 이루어진 후 (그림 8)처럼 동일한 이름의 노드가 존재하면, 이 두 노드를 하나의 노드로 통합한다.



(그림 9) 통합된 후 π-그래프

π-그래프 전개의 시각화 과정을 요약하면 다음과 같다.

① τ전위(Transition)가 가능한 경우(싱크노드가 존재하는 경우)

싱크노드 a를 통해 출력노드 b를 입력노드 x와 치환하고 관련된 간선(edge)을 삭제한다.

② 이 결과로 고립된 노드들이 존재하면 해당 노드를 삭제한다.

③ 동일한 이름의 노드들이 존재하는 경우 이를 하나의 노드로 통합한다.

4. 사례 연구

사례연구의 예로서 우리에게 잘 알려진 주유 시스템[8]을 적용한다. 주유 시스템을 예로 선정한 이유는 많은 연구자들이 소프트웨어 아키텍처나 컴포넌트관련 부분에서 자주 인용되왔기 때문이다.

다음은 주유 시스템을 구성하고 있는 각 컴포넌트를 본 논문에서 제안한 명세방법으로 명세한 것이다.

Customer(s) = s!<pay>.s?(u).u?(fuel).**0**

Pump(c) = c?(w, amnt).w!<fuel>.**0**

Cashier(d, p) = d?(amnt).(d!<p>.**0** | p!<p, amnt>.**0**)

주어진 각각의 컴포넌트로 구성된 시스템의 구조를 얻기 위해 정의 2를 적용하여 이들 컴포넌트들로 이루어진 주유 시스템의 아키텍처를 다음과 같이 얻을 수 있다.

Cfg_1 = Customer(s) = s!<pay>.s?(u).u?(fuel).**0**

Cfg_2 = *con*(Pump, σ, Cfg_1) = s!<pay>.s?(u).u?(fuel).**0** | c?(w, amnt). w!<fuel>.**0**

Cfg_3 = *con*(Cashier, σ, Cfg_2) = *con*(Cashier, {ch_1/c, ch_1/p, ch_2/d, ch_2/s}, Cfg_2) = ch_2!<pay>.ch_2?(u).u?(fuel).**0** | ch_1?(w, amnt).w!<fuel>.**0** | ch_2?(amnt).(ch_2!<ch_1>.**0** | ch_1!<ch_1, amnt>.**0**)

이러한 과정이 필요한 이유는 각각의 컴포넌트들만으로 설계단계에서 얻어진 혹은 그 이전단계에서 추상화된 시스템의 구조를 충분히 만족시킬 수 있는가를 보여주기 위한 것이다. 그러나 이것은 시스템의 정적인 측면을 보장하는 것으로 실제로 해당 컴포넌트들간의 상호작용이 제대로 진행될 것이라는 보장을 받지는 못한다. 따라서 정의 4의 전개 규칙을 적용하여 각각의 컴포넌트들이 제대로 상호작용을 원활히 하는가를 보장받아야 한다.

전개 규칙을 주유 시스템에 적용하면 다음과 같다.

$(ch_2!\langle pay\rangle.ch_2?(u).u?(fuel).\mathbf{0} \mid ch_1?(w, amnt).w!\langle fuel\rangle.\mathbf{0} \mid ch_2?(amnt).(ch_2!\langle ch_1\rangle.\mathbf{0} \mid ch_1!\langle ch_1, amnt\rangle.\mathbf{0})) \tau \rightarrow$

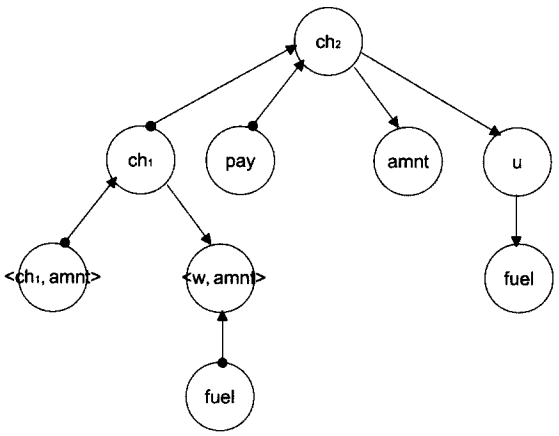
$(ch_2?(u).u?(fuel).\mathbf{0} \mid ch_1?(w, amnt).w!\langle fuel\rangle.\mathbf{0} \mid (ch_2!\langle ch_1\rangle.\mathbf{0} \mid ch_1!\langle ch_1, amnt\rangle.\mathbf{0})) \tau \rightarrow$

$(ch_1?(fuel).\mathbf{0} \mid ch_1?(w, amnt).w!\langle fuel\rangle.\mathbf{0} \mid (\mathbf{0} \mid ch_1!\langle ch_1, amnt\rangle.\mathbf{0})) \tau \rightarrow$

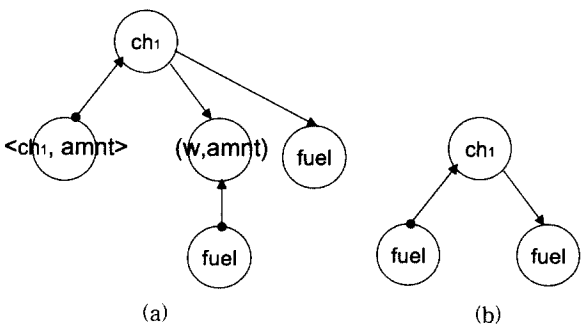
$(ch_1?(fuel).\mathbf{0} \mid ch_1!\langle fuel\rangle.\mathbf{0} \mid (\mathbf{0} \mid \mathbf{0})) \tau \rightarrow \mathbf{0}$

따라서 주어진 컴포넌트들간의 상호작용은 충돌없이 진행됨을 보여준다.

한편, 주유 시스템을 π-그래프로 표현하면 다음과 같다.



(그림 10) 주유시스템의 π-그래프



(그림 11) 전개 후 주유 시스템

π-그래프의 시각화 과정을 적용하면 (그림 11)(b)에서 볼 수 있듯이 최종적으로 채널 ch_1을 통해 주유기와 고객과의 상호교류가 진행되고 있음을 알 수 있다.

5. 결 론

본 논문에서는 컴포넌트 기반 시스템 구축 시 일련의 작업들, 컴포넌트의 생성, 선택, 평가 및 통합과정 중 컴포넌트의 선택, 통합과 관련하여 요구되는 기본적인 연구로써 컴포넌트간 상호작용 명세 기법을 제안하였다. 컴포넌트 기반 개발방법이 갖는 가장 큰 잇점 중 하나가 서로 다른 컴포넌트들간의 조립을 통한 어플리케이션의 구축이라는 측면에서, 각각의 컴포넌트가 갖는 행위의 정형적 명세는 매우 중요한 것이며 또한, 컴포넌트들간 조립의 형태인 아키텍처에 대한 검증역시 설계단계에서 꼭 필요한 요소이다. 이를 위해 본 논문에서는 컴포넌트의 상호작용의 측면에서 인터페이스를 명세하고 정형적 의미를 통해 컴포넌트간 대체성의 보장과 구성된 아키텍처상에서 각각의 컴포넌트들이 원활하게 상호교류적 정보를 유지하는가를 보장받을 수 있도록 하였다. 또한 아키텍처의 전개를 시각화하여 설계자로 하여금 이해를 배가할 수 있도록 π-그래프를 제안하였다. 향후 연구과제로는 π-그래프의 보다 정형적인 정의와 프로토콜의 수행순서 추가, 그리고 자동화된 프로그램의 지원이 요구된다.

참 고 문 헌

[1] Kang K : Issue in Component-Based Software Engineering, 1999 International Workshop on Component-Based Software Engineering.

[2] B. Meyer, Applying Design by Contract, IEEE Computer, pp.40-52, October, 1992.

[3] A. Beugnard, J. Jezequel, N. Plouzeau and D. Watkins. Making components contract aware, IEEE Computer, Vol. 13, No.7, July, 1999.

[4] C. Canal, E. Pimentel and J. M. Troy. Specification and refinement of dynamic software architectures. In Software Architecture, pp.107-126. Kluwer Academic Publishers, 1999.

[5] M. Buchi, W. Weck, The greybox approach : When black-box specifications hide too much. Technical Report 297, Turku Center for Computer Science, Aug., 1999.

[6] D. M. Yellin, R. E. Strom, Protocol specifications and components adaptors. ACM Trans., Vol.19, No.2, pp.292-333, Mar., 1997.

[7] R. Milner, The polyadic π-calculus : A tutorial. In Logic and

Algebra of Specification, pp.203-246, Springer-Verlag, 1993.
[8] D. Helmbold and D. Luckham, Debugging Ada Tasking Programs, IEEE Software, Vol.2, No.2, pp.47-57, 1985.
[9] Mary Shaw, Architectural issues in Software Reuse, Proc IEEE Symposium on Software Reusability, April, 1995.
[10] Christine Mingins, Yu Liu, From UML to Design by Contract, JOOP, April, 2001.
[11] E. Clarke, J. Wing, et al., Formal methods : State of the art and future directions. ACM Computing Surveys, Vol.28, No.4, pp.626-643, Dec., 1996.

이 창 훈

e-mail : be4u@hnu.hankyong.ac.kr
1987년 광운대학교 전자계산학과 이학사
1989년 중앙대학교 전자계산학과(이학석사)
1998년 중앙대학교 컴퓨터공학과(공학박사)
1999~2002년 중앙대학교 정보통신연구소 연구전담교수
2002년~현재 한경대학교 컴퓨터공학과 조교수
관심분야 : 소프트웨어공학, 형식명세기법, 컴포넌트 기반 방법론, 품질 및 프로세스개선 등