http://dx.doi.org/10.3745/KIPSTD.2012.19D.1.081

휘처-아키텍처 대응을 통한 UML 기반 FORM 아키텍처의 가변성 모델링 및 관리

이 관 우†

요 약

FORM 아키텍처 모델은 소프트웨어 프로덕트 라인 내의 제품 개발에 재사용될 수 있는 아키텍처로서 FORM 방법론의 핵심적인 역할을 한다. 하지만 기존의 FORM 아키텍처 모델을 실무에 적용할 때 다음과 같은 문제점들이 있다. 첫째, UML(Unified Modeling Language)과 같이 표준화된 모델이 아니므로, 이 모델을 작성하기 위해서는 고유한 모델링 도구가 필요하다. 둘째, FORM 아키텍처 모델은 휘처모델과의 대응 관계를 통해 가변성을 관리만 할 뿐, 아키텍처의 가변성을 명시적으로 나타내지 않았다. 본 논문에서는 이러한 FORM아키텍처 모델의 문제점을 해결하기 위해서, 먼저 FORM 아키텍처 모델을 UML 모델로 표현할 수 있는 방법을 개발하였다. 이는 FORM 아키텍처 모델링에 다양한 UML 모델링 도구를 이용할 수 있는 장점이 있다. 또한, 휘처모델과의 대응관계를 통해서 FORM 아키텍처 모델의 가변성을 효과적으로 관리할 뿐만 아니라 표현 할 수 있는 방법을 개발하였다.

키워드 : 프로덕트 라인 공학, 프로덕트 라인 아키텍처, 가변성 모델링, 휘처 대응

Managing and Modeling Variability of UML Based FORM Architectures Through Feature-Architecture Mapping

Kwanwoo Lee†

ABSTRACT

FORM(Feature-Oriented Reuse Method) is one of representative product line engineering methods. The essence of FORM is the FORM architecture models, which can be reused in the development of multiple products of a software product line. The FORM architecture models, however, have the following problems when applied in practice. First, they are not standardized models like UML(Unified Modeling Language) and therefore they can be constructed only through a specific modeling tool. Second, they do not represent architectural variability explicitly. Instead their variability is only managed through a mapping from a feature model. To address these two problems, we developed at first a method for representing the FORM architecture models using UML, which enables the FORM architecture models to be constructed through various available UML modeling tools. Also, we developed an effective method for representing as well as managing the variability of the FORM architecture models through a mapping from a feature model.

Keywords : Product Line Engineering, Product Line Architecture, Variability Modeling, Feature Mapping

1. 서 론

소프트웨어 프로덕트 라인은 유사한 기능을 지닌 소프트웨어 제품들의 집합을 의미하며, 소프트웨어 프로덕트 라인 공학(Software Product Line Engineering)[2]은 소프트웨어 프로덕트 라인 범위 내의 소프트웨어 제품 개발에 재사용될 수 있는 공통의 핵심자산 (아키텍처 혹은 구현 모듈)을 개발한 후에, 이를 체계적이고 계획적으로 재사용하여 개별 제품을 생산시키는 소프트웨어 재사용 패러다임이다.

FORM(Feature-Oriented Reuse Method) 방법론[9]은 대표적인 소프트웨어 프로덕트 라인 공학 방법론 중의 하나로서, 프로덕트 라인 범위 내의 소프트웨어 제품들 간의 공통성과 가변성을 분석하여 표현한 휘처모델 (Feature Model) [4, 8]을 이용하여 공통의 핵심자산을 개발하고 핵심자산의 가변성을 체계적으로 관리함으로써 다양한 제품 개발의 생산성 및 유지보수성을 높이는 방법으로 활용되어 왔다. 지금까지 FORM은 EBBS(Electronic Bulletin Board System)

※ 본 논문은 한성대학교 교내연구비에 의해 지원되었음.
† 정 회 원 : 한성대학교 정보시스템공학과 부교수
논문접수 : 2011년 12월 5일
수 정 일 : 1차 2012년 1월 30일
심사완료 : 2012년 2월 10일

[9], PBX (Private Branch Exchange) [10], 엘리베이터 제어 시스템 [13] 등 다양한 시스템의 프로덕트 라인 공학에 적용되어 왔고, 여러 프로덕트 라인 공학 방법 [4, 7, 16]에서 확장되어 오고 있다.

FORM 아키텍처 모델은 소프트웨어 프로덕트 라인 내의 제품 개발에 재사용될 수 있는 아키텍처로서, FORM 방법론의 핵심적인 역할을 한다. 하지만 기존의 FORM아키텍처 모델을 실무에 적용할 때 다음과 같은 문제점들이 있다. 첫째, FORM 아키텍처 모델은 UML(Unified Modeling Language)[19]과 같이 표준화된 모델이 아니므로, 이 모델을 작성하기 위해서는 고유한 모델링 도구가 필요하다. 둘째, FORM 아키텍처 모델은 제품들 간의 공통적인 부분과 가변적인 부분을 명시적으로 나타내지 않고, 공통성과 가변성 정보를 표현한 휘처모델과의 대응 관계를 통해 가변성을 관리하므로, 아키텍처 수준에서의 가변성을 직관적으로 관측하기 어려운 점이 있다.

본 논문에서는 이러한 FORM 아키텍처 모델의 단점을 해결하기 위해서 FORM 아키텍처 모델을 UML 모델로 표현할 수 있는 방법과, 휘처모델과 FORM 아키텍처 모델의 대응 관계를 통해서 FORM 아키텍처 모델의 가변성을 효과적으로 표현하고 관리할 수 있는 방법을 제안한다.

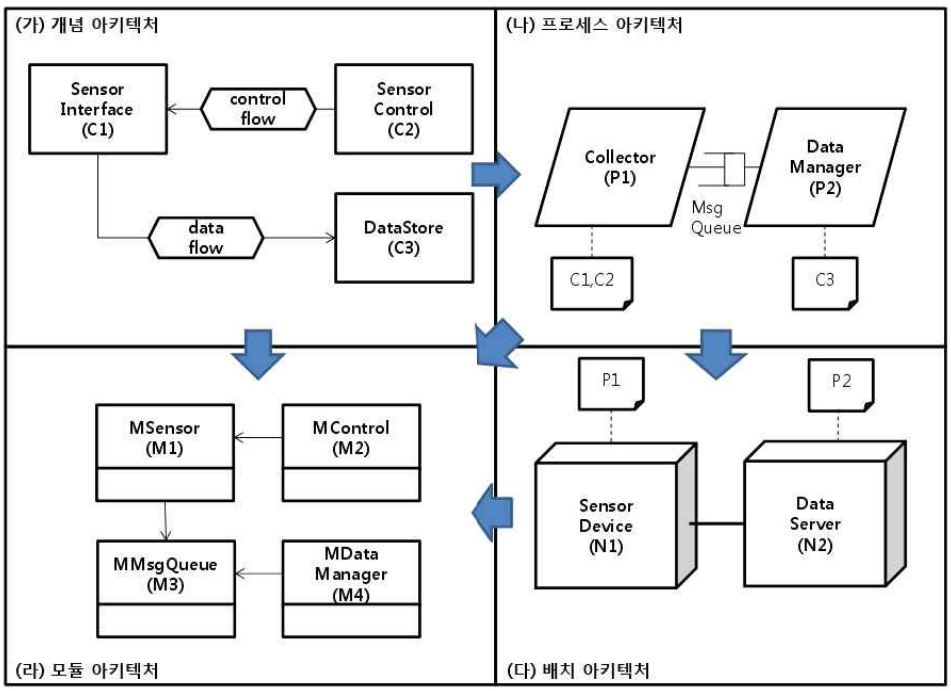
본 논문의 구성은 다음과 같다. 2장에서는 기반 연구로써 FORM 아키텍처 모델에 대해서 간략히 설명한다. 3장에서는 UML을 이용하여 FORM 아키텍처의 가변성을 모델링하고 관리하는 방법을 제안한다. 4장에서는 제안된 방법을 적용한 사례 연구에 대해서 기술한다. 5장에서는 기존 연구와 본 연구와의 차이에 대해서 비교분석을 하고, 6장에서는 본 연구에 대한 토의와 향후 연구 과제에 대해 언급하면서 결론을 내린다.

2. FORM 아키텍처 모델

FORM 아키텍처 모델 [9, 11]은 개념, 프로세스, 배치, 모듈 관점의 아키텍처로 구성되어 있다. 본 절에서는 FORM 아키텍처 모델에 대해서 예를 통하여 간략하게 설명한다.

개념 아키텍처 (Conceptual Architecture)는 특정한 구현 및 하드웨어에 독립적인 기능 관점을 나타내며, 주로 시스템의 논리적인 기능 단위들을 찾고, 이들 간의 논리적인 상호작용을 나타낸다. (그림 1)의 (가)는 개념 아키텍처의 간단한 예로서 *SensorInterface* (*C1*), *SensorControl* (*C2*), *DataStore* (*C3*)의 개념 컴포넌트로 구성된다. *SensorInterface* 컴포넌트는 입력센서 다바이스의 확장성 및 변경성을 지원하기 위하여 구체적인 센서 디바이스의 정보를 감추는 역할을 수행하며, *SensorControl* 컴포넌트는 *SensorInterface* 컴포넌트의 동작을 제어하는 역할을 수행하므로, *SensorControl* 컴포넌트와 *SensorInterface* 컴포넌트 사이에 *control flow* 커넥터로 연결되어 있다. *SensorInterface* 컴포넌트는 감지된 센서 값을 *DataStore* 컴포넌트로 *data flow* 커넥터를 이용하여 전송하게 된다.

프로세스 아키텍처 (Process Architecture)는 시스템의 동시성 구조를 나타내는 것으로, 시스템에서 동시에 수행될 수 있는 단위인 프로세스 컴포넌트 (Process Component)를 찾고, 이들 간의 상호작용을 프로세스 통신 커넥터 (Process Communication Connector)를 통해서 나타낸 것이다. (그림 1)의 (나)에서 보는 바와 같이, *Collector*와 *DataManager*는 동시에 수행될 수 있는 단위를 나타내고, *Collector*에서 *DataManager*와의 상호작용은 메시지큐 방식의 *MsgQueue* 커넥터를 통해서 이루어짐을 나타낸다.



(그림 1) FORM 아키텍처 모델

프로세스 아키텍처와 개념 아키텍처는 서로 별개의 산출물이라기 보다는 서로 대응관계를 가지고 있다. 즉, 프로세스 아키텍처의 프로세스 컴포넌트는 개념 아키텍처의 개념 컴포넌트와 대응되고, 프로세스 통신 커넥터는 개념 커넥터와 대응이 된다. (그림 1)의 (나)에서는 이러한 대응관계를 노트 표시로 표현하였다. 예를 들면, *Collector* (*P1*) 프로세스 컴포넌트는 *SensorInterface* (*C1*)과 *SensorControl* (*C2*)와 대응되고, *DataManager* (*P2*) 프로세스 컴포넌트는 *DataStore* (*C3*)와 대응됨을 표시하였다.

배치 아키텍처는 프로세스 컴포넌트가 물리적인 하드웨어 노드에 어떻게 배치되는 가를 나타내는 것으로, 노드 컴포넌트, 링크 커넥터, 그리고 이들 간의 연결관계로 정의된다. (그림 1)의 (다)에서 보는 바와 같이, *SensorDevice* (*N1*)과 *DataServer* (*N2*)는 프로세스 컴포넌트가 배치될 하드웨어 노드를 나타내고, 이들 간의 물리적인 링크 커넥터가 연결되어 있다. (그림 1)의 (다)에서 보는 바와 같이, *Collector* (*P1*) 프로세스 컴포넌트는 *SensorDevice* (*N1*) 노드 컴포넌트에, *DataManager* (*P2*) 컴포넌트는 *DataServer* (*N2*) 컴포넌트에 대응되었다.

마지막으로 모듈 아키텍처는 개념, 프로세스, 배치 아키텍처 모델에서 내려진 설계 결정이 어떻게 구현 단위인 모듈로 대응 될 수 있는 지를 나타낸다. (그림 1)의 (라)에서 보는 바와 같이, 네 개의 모듈 *MSensor, MControl, MMsgQueue, MDataManager*는 개념, 프로세스, 배치 아키텍처의 설계결정을 구현단위인 모듈로 나타내고, 이들 간의 호출 관계를 표시한 것이다.

이와 같은 FORM 아키텍처 모델은 표준화된 모델이 아니므로, FORM 아키텍처 모델을 작성하기 위해서는 ASADAL[18]과 같은 특화된 모델링 도구가 필요하다. 따라서 본 연구에서는 FORM 아키텍처를 산업계의 표준으로 자리잡고 있는 UML (Unified Modeling Language)를 이용하여 표현하는 방법을 제시함으로써, 다양한 UML 모델링 도구를 활용하여 FORM 아키텍처 모델링을 지원 하는 것을 목표로 하고 있다.

3. UML 기반 FORM 아키텍처의 가변성 모델링과 관리

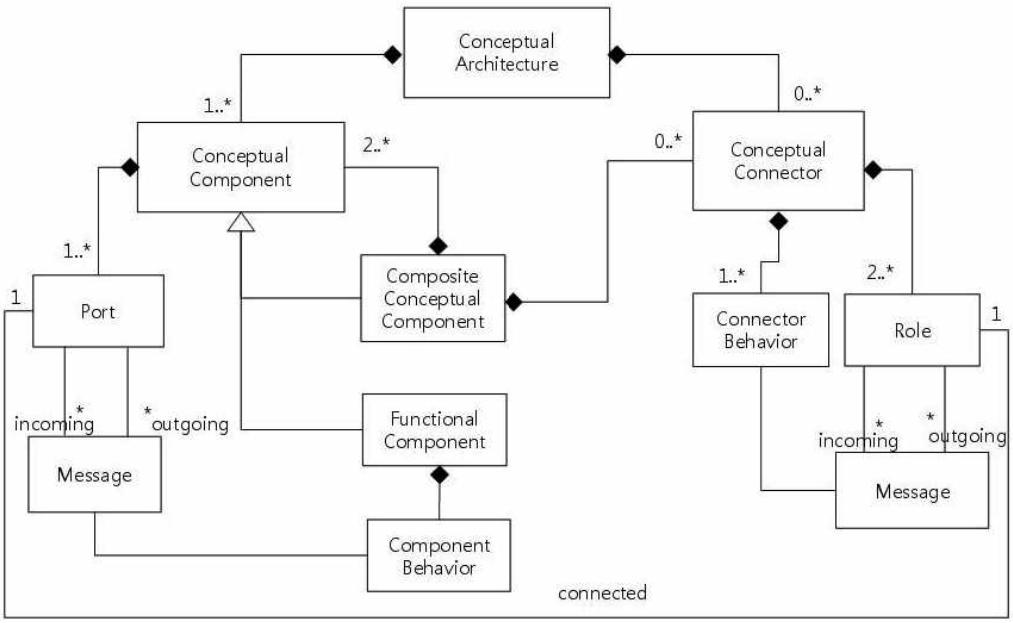
본 장에서는 먼저 FORM 아키텍처 모델을 UML 모델을 이용하여 표현하는 방법을 설명하고, UML 기반 FORM 아키텍처 모델의 가변성을 표현하고 관리하는 기법을 차례대로 설명한다.

3.1 UML 기반 FORM 아키텍처 모델링

FORM 아키텍처 모델 중에 배치 아키텍처와 모듈 아키텍처 모델은 UML 배치 다이어그램과 UML 클래스 다이어그램에 바로 대응시킬 수 있으므로, 본 논문에서는 FORM의 개념 아키텍처와 프로세스 아키텍처 모델을 위주로 UML 모델과의 대응을 설명한다.

3.1.1 개념 아키텍처 모델링

2장에서 간단히 설명했듯이, FORM 개념 아키텍처는 시스템 혹은 시스템들의 논리적인 기능 단위를 개념 컴포넌트(Conceptual Component)로, 이들 간의 논리적인 상호작용을 개념 커넥터 (Conceptual Connector)로 정의하고, 개념 컴포넌트와 개념 커넥터 간의 연결(Connection) 관계를 통해 아키텍처를 구성한다.



(그림 2) FORM 개념 아키텍처 메타 모델

(그림 2)에서 보는 바와 같이, 하나의 개념 컴포넌트는 여러 개념 컴포넌트와 개념 커넥터로 분해될 수 있는데, 다른 개념 컴포넌트를 포함하는 개념 컴포넌트를 복합 개념 컴포넌트 (Composite Conceptual Component)라 하고, 컴포넌트 분해 계층의 말단 개념 컴포넌트를 특별히 기능 컴포넌트 (Functional Component)라 명명한다. 이 기능 컴포넌트는 컴포넌트 행위 (Component Behavior)를 갖게 된다. 또한 개념 컴포넌트는 다른 개념 컴포넌트와의 메시지 상호작용 지점을 나타내는 하나 이상의 포트 (Port)를 가지고 있다. 각 포트는 나가는 메시지(Outgoing Message) 혹은 들어오는 메시지(Incoming Message)와 관련된다.

개념 커넥터는 커넥터 행위 (Connector Behavior)와 롤 (Role)을 구성 요소로 가지고 있는데, 커넥터 행위는 개념 컴포넌트의 컴포넌트 행위에 대응되고, 롤은 개념 컴포넌트의 포트에 대응된다. 따라서, 각 개념 커넥터의 롤은 개념 컴포넌트의 포트와 마찬가지로, 나가는 메시지와 들어오는 메시지로 정의된다.

개념 아키텍처는 관련된 개념 컴포넌트의 포트와 개념 커넥터의 롤이 연결을 이루면서 구성된다. 즉, 개념 컴포넌트의 포트로부터 나가는 메시지는 연결된 개념 커넥터의 롤로 전달되고, 개념 커넥터로부터 나가는 메시지는 연결된 개념 컴포넌트의 포트로 전달된다.

UML 2.x에서는 (그림 2)에서 정의된 개념 아키텍처를 모델링 하기 위해 유용한 모델링 요소를 제공하고 있다. <표 1>에서 보는 바와 같이, 복합 개념 컴포넌트는 UML 서브시스템으로 나타내고, 기능 컴포넌트는 UML 컴포넌트로 모델링 한다. 개념 커넥터는 UML에서 명시적으로 지원하지 않으므로, UML 컴포넌트로 모델링하고 UML 스테레오타입을 이용하여 개념 컴포넌트와 구분 짓는다. 개념 컴포넌트는 <<conceptual>> 혹은 <<functional>> 스테레오타입을 사용하여 복합 개념 컴포넌트와 기능 컴포넌트를 구분하고, 개념 커넥터는 개념 컴포넌트와 구분하기 위해서 <<c-connector>>로 표현한다. 또한, 포트를 통해 메시지가 들어온다는 것은 해당 포트가 제공하는 인터페이스를 다른 외부 컴포넌트가 사용한다는 의미이고, 포트를 통해 메시지가 나간다는 것은 다른 외부 컴포넌트에서 제공하는 인터페이스를 해당 포트를 통해서 요구한다는 의미이므로, 포트를 통해 들어오는 메시지는 UML의 제공 인터페이스(Provided Interface)로 대응시키고, 나가는 메시지는 UML의 요구 인터페이스(Required Interface)로 대응시킨다.

3.1.2 프로세스 아키텍처 모델링

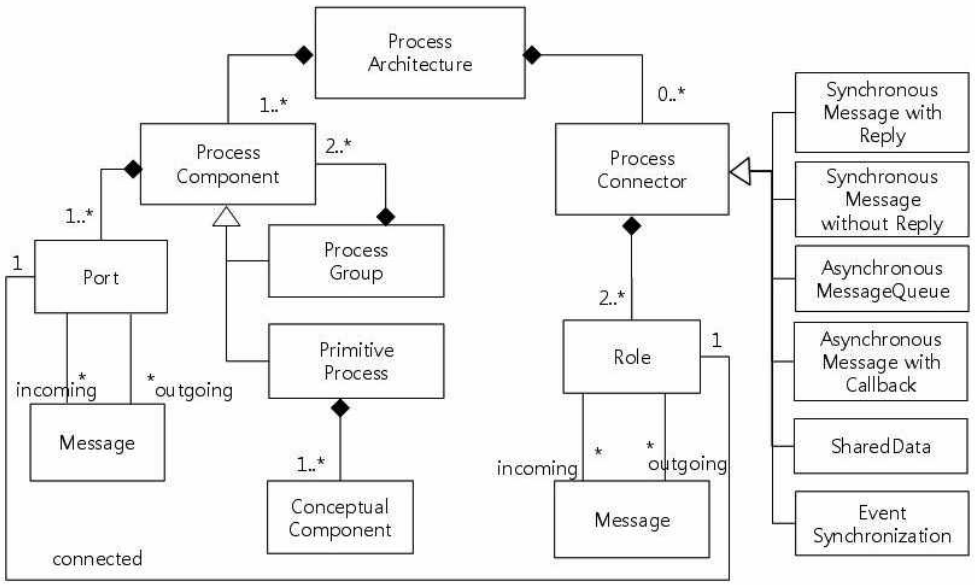
프로세스 아키텍처는 시스템 혹은 시스템들의 동시성 구조를 나타내는 것으로, 프로세스 컴포넌트 (Process Component), 프로세스 통신 커넥터 (Process Communication Connector), 그리고 이들 간의 연결관계를 통해 아키텍처를 구성한다.

(그림 3)에서 보는 바와 같이, 프로세스 컴포넌트는 프로세스 그룹 (Process Group)과 기본 프로세스 (Primitive Process)로 구분된다. 프로세스 그룹은 프로세스 컴포넌트들의 단순한 묶음을 나타내며, 기본 프로세스는 동시에 실행될 수 있는 단위를 나타낸다. 개념 아키텍처의 개념 컴포넌트들은 논리적인 기능 단위이므로 반드시 동시성 단위와 일치하지는 않는다. 따라서 하나의 기본 프로세스 컴포넌트에는 하나 이상의 개념 컴포넌트들이 대응될 수 있으며, 대응된 개념 컴포넌트들은 하나의 실행 쓰레드에 의해서 순차적으로 실행된다.

프로세스 통신 커넥터는 프로세스 컴포넌트 간의 메시지 상호작용을 나타내는 것으로, 메시지 통신 (Message Communication), 공유 데이터 (Shared Data), 이벤트 동기화 (Event Synchronization)의 세 가지로 분류된다. 메시지 통신은 다시 동기 메시지 통신과 비동기 메시지 통신으로 분류되고, 각각은 응답이 있는 통신과 무응답 통신으로 세

〈표 1〉 개념 아키텍처 요소의 UML 모델링 요소와의 대응

FORM 구성요소	UML 구성요소	UML 표기법
Conceptual Component	UMLSubsystem UMLComponent	«conceptual» Name «functional» Name
Port (from Conceptual Component)	UMLPort	PortName
Incoming Message (from Port)	UMLInterface	Provided Interface
Outgoing Message (from Port)	UMLInterface	Required Interface
Conceptual Connectors	UMLComponent	«c-connector» Name
Role (from Conceptual connector)	UMLPort	roleName «c-connector» Name roleName



(그림 3) FORM 프로세스 아키텍처 메타 모델

분화 될 수 있다. 프로세스 통신 커넥터는 개념 커넥터가 프로세스 아키텍처에서 상세화 (Refinement)된 것으로 볼 수 있다. 즉 개념 커넥터가 개념 컴포넌트간의 논리적인 상호작용을 모델링 한 것이라면, 이러한 논리적인 상호작용이 프로세스 컴포넌트 간에 일어날 때, 어떠한 메커니즘을 통해서 일어나는 가를 구체화한 것이다.

프로세스 아키텍처에서의 프로세스 컴포넌트와 프로세스 통신 커넥터도 모두 UML 컴포넌트를 통해서 표현이 가능하다. 다만 개념 컴포넌트, 개념 커넥터와 구분하기 위해서 프로세스 컴포넌트는 스테레오타입 <<process>> 로, 프로세스 통신 커넥터는 스테레오타입 <<pc-connector>> 로 구분한다. 한편 프로세스 컴포넌트에는 하나 이상의 개념 컴포넌트가 대응되므로, UML의 확장 메커니즘이 태그값 (Tagged Value)를 이용하여 프로세스 컴포넌트에 할당된 개념 컴포넌트를 모델링 한다.

3.2 FORM 아키텍처 모델의 가변성 표현 및 관리 기법

본 논문에서는 FORM 아키텍처 모델의 가변성을 휘처모델과의 대응 관계를 통해서 표현하고 관리한다. 본 절에서는 먼저 휘처모델과 FORM 아키텍처 모델간의 대응에 대해서 먼저 설명하고, 휘처의 가변성 정보에 따라 FORM 아키텍처 모델 요소의 가변성을 결정하는 방법을 차례대로 기술한다.

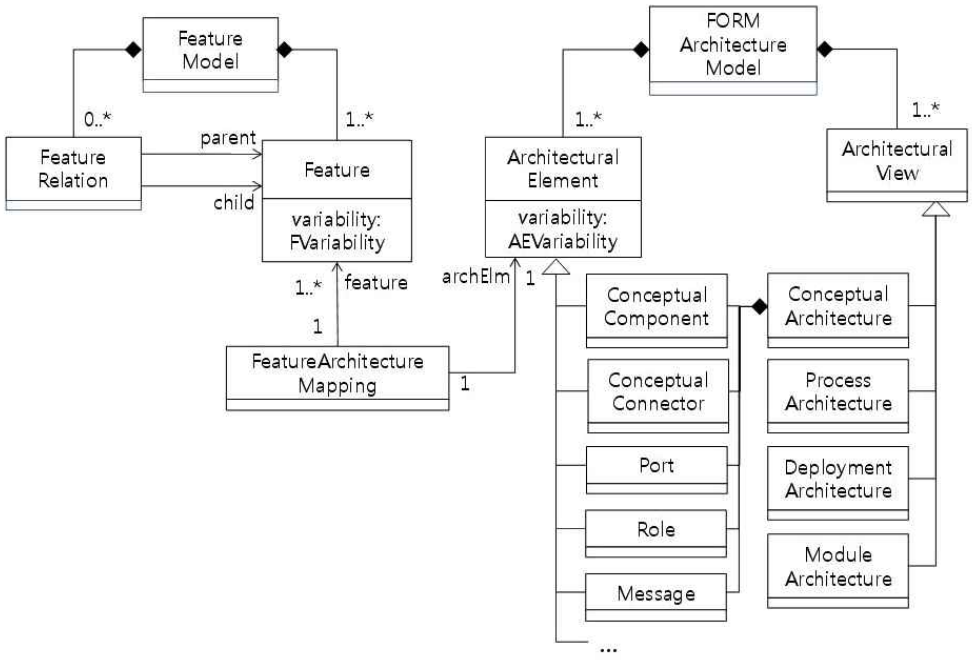
3.2.1 휘처모델과 FORM 아키텍처 모델간의 대응

휘처모델은 프로덕트 라인의 공통성과 가변성을 휘처 단위로 표현한 모델로서 (그림 4)의 왼쪽부분은 휘처모델의 메타모델 일부분을 나타낸 것이다. 휘처모델은 휘처 (Feature)와 휘처 관계 (Feature Relation)으로 구성되며, 휘처는 속성 *variability*를 가지고 있으며, 이 속성의 타입 *FVariability*는 *mandatory, optional, alternative*의 열거형 (Enumeration)으로 정의된다. 여기서 *mandatory* 속성 값을 갖는 휘처는 그의 부모 관계에 있는 휘처가 특정 제품을 위해 선택될 때 항상 선택되어야 하는 필수적인 요소임을 나타내고, *optional*은 해당 휘처의 부모 관계에 있는 휘처가 제품에 포함되어 있을 때, 해당 휘처가 선택될 수도 있고 선택되지 않을 수 도 있는 가변성을, *alternative*는 해당 휘처와 동일한 부모 관계에 있는 휘처들 간에 택일적으로 선택되는 가변성을 나타낸다. 휘처모델에 대한 자세한 내용은 [4, 8]을 참조한다.

FORM 아키텍처모델은 (그림 4)의 오른쪽과 같이, 아키텍처 뷰 (Architectural View)로서 개념 아키텍처, 프로세스 아키텍처, 배치 아키텍처, 모듈 아키텍처를 가지고 있으며, 개념 아키텍처는 개념 컴포넌트, 개념 커넥터, 포트, 롤, 메시지 등을 아키텍처 요소 (Architectural Element)로서 포함하고 있다. 각 아키텍처 요소는 속성 *variability*를 가지고 있으며, 이 속성의 타입 *AEVariability*는 *M, VO, VA, VOP, VAP, VP*의 열거형으로 정의된다. 이들 값에 대한 의미는 다음 절에서 자세히 설명한다.

FORM 아키텍처모델은 휘처모델과 대응관계를 가지는데, 이러한 대응관계는 (그림 4)에서 보는 바와 같이 휘처 아키텍처 대응 (Feature Architecture Mapping)을 통해서 설정된다. 휘처 아키텍처 대응은 휘처모델과 FORM 아키텍처 모델과는 별도로 작성되는 모델링 요소로서, FORM 아키텍처 모델을 구성하는 각각의 아키텍처 요소에 대해서 다수의 관련된 휘처들이 대응되는 관계로 정의된다. 한가지 주지할 사항은 모든 아키텍처 요소가 반드시 하나 이상의 휘처와 대응 관계를 가져야 한다.

휘처와 아키텍처 요소와의 대응 의미는 해당 아키텍처 요소가 해당 휘처를 실현하는 관계에 있음을 의미한다. 따라



(그림 4) 휘처모델과 FORM 아키텍처 모델의 대응

서, 해당 휘처가 프로덕트 라인 범위 내의 특정 제품을 위해서 선택되지 않는 다면, 이 휘처를 실현하는 아키텍처 요소도 해당 제품을 위해서는 포함되지 말아야 한다.

3.2.2 아키텍처 모델의 가변성 표현

아키텍처 모델의 가변성은 크게 가변점과 가변성 종류의 두 가지 관점에서 정의된다.

가변점 (Variation Point)은 아키텍처 요소 중에서 변화 요소가 있는 지점을 의미하고, 아키텍처 요소 (컴포넌트나 커넥터 등) 자체가 가변점일 수도 있고, 아키텍처 요소 내부에 가변점이 정의될 수도 있다. 가변점이 아키텍처 요소 자체에 설정된 경우에는 해당 아키텍처 요소가 그대로 제품에 따라 재사용될 수 도 있고, 재사용되지 않을 수도 있는 단위이므로 이는 해당 아키텍처 요소가 블랙박스 재사용 [5]됨을 의미한다. 하지만, 가변점이 아키텍처 요소 내부에도 정의된다면, 해당 아키텍처 요소는 그대로 재사용되기보다는 수정을 통해서 재사용될 수 있음을 나타내므로, 이는 화이트박스 재사용[5]에 해당된다. 따라서 본 논문에서는 아키텍처 요소 자체에만 가변점이 있는 경우를 블랙박스 가변점이라 정의하고, 내부에도 가변점이 있는 아키텍처 요소를 화이트박스 가변점이라 정의한다.

이와 같이 아키텍처 요소 자체뿐만 아니라 내부의 가변성도 명시적으로 표현함으로써, 계층적으로 표현되는 아키텍처 모델의 가변성 정보가 좀더 풍부하게 표현되는 장점이 있다. 예를 들면, 상위 수준의 아키텍처 모델은 추상 컴포넌트들로 구성되는데, 이 추상 컴포넌트 자체가 가변성을 가질 수도 있고, 추상 컴포넌트를 구체화시키는 하위 단계에서 가변성이 나타날 수가 있다. 이와 같이 하위 단계에서 가변성이 나타나는 추상컴포넌트를 화이트박스 가변점으로 정의함으로써, 아키텍처 모델의 각 단계별로 가변성 정보가 보다 구체적으로 제시되는 효과가 있다.

한편, 각 가변점이 가지는 가변성의 종류도 선택적(optional) 혹은 택일적(alternative) 중에 하나로 표현된다. 따라서 블랙박스 가변점이 선택적 혹은 택일적 가변성을 가질 수도 있고, 화이트박스 가변점이 선택적 혹은 택일적 가변성을 가질 수도 있다.

위와 같은 아키텍처 요소의 가변성은 UML 스테레오타입을 이용하여 표현된다. 선택적-블랙박스 가변점은 <<VO>> (혹은 <<●>>), 택일적-블랙박스 가변점은 <<VA>> (혹은 <<▲>>), 선택적-화이트박스 가변점은 <<VOP>> (혹은 <<○>>), 택일적-화이트박스 가변점은 <<VAP>> (혹은 <<△>>)으로 표현되고, 아키텍처 요소 자체에는 가변성은 없지만 내부에 임의의 가변 요소가 존재할 경우에는 <<VP>> (혹은 <<□>>)으로 표현된다. 단, 아키텍처 요소가 독립적인 가변성이 없다면 <<M>>으로 표현되며, 이러한 아키텍처 요소는 해당 아키텍처 요소가 프로덕트 라인의 모든 제품에 항상 포함되어야 하는 경우나 해당 아키텍처 요소의 존재 유무가 이를 포함하는 상위 아키텍처 요소가 선택되는 경우에 반드시 같이 존재해야 하는 경우에 해당된다. 예를 들면, 선택적 가변성을 가지는 컴포넌트 c1에 하나의 포트 p1이 정의되어 있고, p1은 c1이 선택될 때 항상 같이 존재해야 한다면, c1의 가변성은 <<VO>>이지만 포트 p1의 가변성은 <<M>>으로 표현된다.

본 논문에서는 이와 같은 아키텍처 요소의 가변성이 휘처모델의 휘처와의 대응 관계를 통해서 자동으로 결정되는 방

법을 사용한다. 이어지는 절에서는 대응되는 휘처의 가변성에 따라서 아키텍처 요소의 가변성이 결정되는 규칙에 대해서 설명한다.

3.2.3 아키텍처 모델의 가변성 결정 규칙

아키텍처 모델요소의 가변성 결정 규칙을 정의하기에 앞서서, 규칙 정의에서 사용되는 유용한 술어 함수 (Predicate Function)는 다음과 같이 정의된다.

- Mandatory(*f*), Optional(*f*), Alternative(*f*)

각 함수에 대해서, 주어진 휘처 *f*의 속성 variability 값이 각각 *mandatory, optional, alternative*이면 참을 반환, 그렇지 않으면 거짓을 반환한다. (단, *f*.variability는 휘처 *f*의 속성 variability를 의미한다.)

Mandatory(*f*) ≡ *f*.variability = mandatory

Optional(*f*) ≡ *f*.variability = optional

Alternative(*f*) ≡ *f*.variability = alternative

- OnlyMFeatures(*F*): 주어진 휘처 집합 *F*가 필수적 휘처만 포함하면 참을 반환, 그렇지 않으면 거짓을 반환한다.

OnlyMandatory(*F*) ≡ ($\forall f \in F \cdot$ Mandatory(*f*))

- VariableFeatures(*F*): 주어진 휘처 집합 *F*가 선택적 휘처 혹은 택일적 휘처를 포함하면 참을 반환, 그렇지 않으면 거짓을 반환한다.

VariableFeatures(*F*) ≡ ($\exists f \in F \cdot$ Optional(*f*) ∨ Alternative(*f*))

- AlternativeFGroup(*F1,F2*): 주어진 휘처 집합 *F1*과 *F2*가 각각 택일적 관계에 있는 하나의 택일 휘처 만을 포함하고 나머지 휘처들은 모두 동일하다면 참을 반환, 그렇지 않으면 거짓을 반환하는 함수이다.

AlternativeFGroup(*F1,F2*) ≡ ($\exists f \in F1-F2 \cdot$ Alternative(*f*)) ∧ ($F1-F2-\{f\} = \varnothing$) ∧ ($\exists x \in F2-F1 \cdot$ Alternative(*x*)) ∧ ($F2-F1-\{x\} = \varnothing$)

예를 들면, 주어진 휘처 집합 *F1*={*f1, f2*}, *F2*={*f2, f3*}에 대해서 *f1*과 *f3*는 택일적 휘처이고 *f2*는 필수적 휘처이면, 이 술어 함수는 참을 반환한다.

- MandatoryElms(*E*): 주어진 아키텍처 요소 집합 *E*가 오로지 필수적 요소만을 포함하면 참을 반환, 그렇지 않으면 거짓을 반환하는 함수.

MandatoryElms(*e*) ≡ ($\forall e \in E \cdot e$.variability = M)

- VariableElms(*E*): 주어진 아키텍처 요소 집합 *E*가 선택적 가변요소 혹은 택일적 가변요소를 포함하면 참을 반환, 그렇지 않으면 거짓을 반환하는 함수.

VariableElms(*E*) ≡ ($\exists e1 \in E \cdot$ (*e*.variability = VO) ∨ (*e*.variability = VOP) ∨ (*e*.variability = VA) ∨ (*e*.variability = VAP)

본 논문에서는 FORM 아키텍처 모델 중에서 개념 아키텍처 모델과 프로세스 아키텍처 모델의 가변성 결정 규칙에 초점을 두어 기술한다. 이유는 배치 아키텍처 모델과 모듈 아키텍처 모델의 가변성 결정 규칙도 개념 혹은 프로세스 아키텍처 모델의 경우와 매우 유사하기 때문이다.

개념 아키텍처 모델의 구성 요소들은 (그림 2)에서 보는 바와 같이 복합 개념 컴포넌트는 개념 컴포넌트를 포함하고, 개념 컴포넌트의 일종인 기능 컴포넌트는 포트와 함께 컴포넌트 행위를 포함한다. 또한 포트는 하나 이상의 메시지와 연관되어 있다. 따라서 메시지부터 가변성 결정규칙을 설명하고, 점차적으로 상위 단계의 포함 관계에 있는 아키텍처 모델 구성요소의 가변성 결정 규칙을 설명한다.

가변성 결정 규칙을 설명하기에 앞서, 술어함수 Features(e)는 아키텍처 요소 e에 대응된 휘처들의 집합으로, Port(m)은 주어진 메시지 m과 관련된 포트로 정의되고, MESSAGE는 개념 아키텍처 내의 모든 메시지 집합을 의미한다. 또한 가변성 결정 규칙 c⇒a는 조건식 c가 만족되었을 때, 결과식 a가 결정된다는 의미를 나타낸다. 마지막으로 모든 아키텍처 요소의 초기 가변성 값은 M (필수적 요소)으로 초기화 되어 있다. 따라서 다음에 설명될 가변성 결정 규칙에 적용되지 않는 아키텍처 요소의 가변성은 필수적 요소임을 나타낸다.

규칙 1 메시지의 가변성 결정 규칙

주어진 메시지 m에 대해서,

1. OnlyMFeatures(Features(m)) ∨ (Features(m) ⊆ Features(Port(m)) ⇒ m.variability = M
2. ∃m1 ∈ MESSAGE · (AlternativeFGroup(Features(m), Features(m1)) ∧ (Port(m) = Port(m1))) ⇒ m.variability = VA
3. VariableFeatures(Features(m) - Features(Port(m))) ∧ ¬∃m1 ∈ MESSAGE · (AlternativeFGroup(Features(m), Features(m1)) ∧ (Port(m) = Port(m1))) ⇒ m.variability = VO

첫 번째 규칙은 메시지 m에 할당된 휘처들이 모두 필수적 휘처이거나, m에 대응된 휘처 집합(Features(m))이 m과 관련된 포트에 대응된 휘처 집합(Features(Port(m)))의 부분집합이면, m의 가변성은 필수적 요소를 나타내는 M으로 설정됨을 의미한다. 예를 들면, 포트 p를 통해서 전달되는 메시지 m이 있고, p와 m에 모두 선택적 휘처 f1이 대응되었다고 한다면, m의 가변성은 선택적 가변성을 가지기 보다는 필수적 가변성을 가진다. 그 이유는 3.2.2에서 언급했듯이, 아키텍처 요소의 필수적 가변성은 해당 구성요소를 포함하고 있는 구성요소가 선택되었을 때 반드시 같이 존재해야 하는 가변성을 의미하기 때문이다.

두 번째 규칙은 메시지 m에 대응된 휘처 집합과 임의의 다른 메시지 m1에 대응된 휘처 집합이 택일적 관계에 있고 m과 m1이 동일한 포트와 관련되어 있다면, 메시지 m의 가변성이 택일적-블랙박스 가변점인 VA로 설정됨을 의미한다. 예를 들면, 휘처 f1과 f2는 택일적 휘처이고, 포트 p를 통해서 전달되는 메시지 m1과 m2가 있을 때, f1은 m1에, f2는 m2에 대응된다고 가정하면, m1과 m2는 택일적-블랙박스 가변성인 VA로 결정된다.

세 번째 규칙은 메시지 m에 대응된 가변휘처들(Features(m)) 중에서 m과 관련된 포트에 대응된 가변휘처들 (Features(Port(m)))을 제외한 휘처 집합이 가변 휘처를 포함하고 있고, 메시지 m과 택일적 관계에 있는 다른 메시지가 동일 포트 내에 없다면, m의 가변성은 선택적-블랙박스 가변점인 VO로 설정됨을 의미한다. 예를 들면, 휘처 f1과 f2는 선택적 휘처로서, f2는 f1의 자식 관계에 있는 휘처이고, 포트 p를 통해서 전달되는 메시지 m1과 m2가 있고, p와 m1에는 f1이, m2에는 f2가 대응된다고 가정하자. 이때 메시지 m1의 가변성은 첫 번째 규칙에 의해서 필수적 가변성 M이 되지만, 메시지 m2의 가변성은 세 번째 규칙에 의해서 선택적-블랙박스 가변성인 VO로 결정된다.

포트의 가변성 결정 규칙을 설명하기에 앞서, Msgs(p)는 포트 p와 관련된 메시지들의 집합으로 정의되고, Container(e)는 아키텍처 요소 e를 포함하는 관계에 있는 아키텍처 요소를 의미한다. 또한, PORT는 개념 아키텍처 내의 모든 포트들의 집합을 나타낸다.

규칙 2 포트의 가변성 결정 규칙

주어진 포트 p에 대해서,

1. $\exists p1 \in PORT \cdot AlternativeFGroup(Features(p), Features(p1)) \wedge (Container(p) = Container(p1)) \wedge MandatoryElms(Msgs(p)) \Rightarrow p.variability = VA$
2. $VariableFeatures(Features(p) - Features(Container(p)) \wedge \neg\exists p1 \in PORT \cdot AlternativeFGroup(Features(p), Features(p1)) \wedge (Container(p) = Container(p1)) \wedge MandatoryElms(Msgs(p)) \Rightarrow p.variability = VO$
3. $\exists p1 \in PORT \cdot AlternativeFGroup(Features(p), Features(p1)) \wedge (Container(p) = Container(p1)) \wedge VariableElms(Msgs(p)) \Rightarrow p.variability = VAP$
4. $VariableFeatures(Features(p) - Features(Container(p)) \wedge \neg\exists p1 \in PORT \cdot AlternativeFGroup(Features(p), Features(p1)) \wedge (Container(p) = Container(p1)) \wedge VariableElms(Msgs(p)) \Rightarrow p.variability = VOP$
5. $MandtoryFeatures(Features(p)) \vee (Features(p) \subseteq Features(Container(p)) \wedge VariableElms(Msgs(p)) \Rightarrow p.variability = VP$

첫 번째 규칙은 포트 p에 대응된 휘처 집합과 포트 p와 같은 컴포넌트 내에 있는 임의의 다른 포트 p1에 대응된 휘처 집합이 택일적 관계를 맺고, 포트 p와 관련된 메시지들이 모두 필수적인 요소라면, 이 포트의 가변성은 택일적-블랙박스 가변성을 나타내는 VA로 결정된다는 것을 나타낸다. 예를 들면, 개념 컴포넌트 c에는 두 개의 포트 p1과 p2가 정의되어 있다고 가정할 때, c에는 필수 휘처 f1이, p1과 p2에는 각각 택일 휘처 f2와 f3가 대응되었고, p1과 p2와 관련된 메시지들이 필수적 요소라면, p1과 p2의 가변성은 택일적-블랙박스 가변점으로 모델링 된다.

두 번째 규칙은 포트 p에 대응된 휘처들(Features(p)) 중에서 p을 포함하는 컴포넌트에 대응된 휘처들 (Features(Container(p)))을 제외한 휘처 집합이 가변 휘처를 포함하고 있고, 포트 p와 택일적 관계에 있는 다른 포트가 동일 컴포넌트 내에 없고, 포트 p와 관련된 메시지들이 모두 필수적인 요소라면, p의 가변성은 선택적-블랙박스 가변점인 VO로 설정됨을 의미한다. 예를 들면, 어떠한 개념 컴포넌트 c에 하나의 포트 p가 정의되어 있고, p를 통해 전달되는 메시지 m이 있다고 가정할 때, c에는 필수 휘처 f1이, p에는 선택 휘처 f2가 대응되고 p와 관련된 메시지들이 필수적 요소라면, p의 가변성은 선택적-블랙박스 가변점으로 결정된다. 만약 포트 p에 선택 휘처 f2외에 택일 휘처 f3가 대응되었고, f3와 택일 관계에 있는 다른 택일 휘처가 포트 p가 속한 동일한 컴포넌트 내의 다른 포트에 대응되지 않았다면, 이 경우에도 p의 가변성은 선택적-블랙박스 가변점으로 결정된다.

세 번째와 네 번째 규칙은 각각 첫 번째와 두 번째 규칙과 매우 유사하고, 유일한 차이점은 해당 포트가 내부적인 가변성을 가짐을 나타내게 되어 각각 택일적-화이트박스, 선택적-화이트박스 가변성을 나타내는 VAP, VOP로 결정됨을 나타낸다. 예를 들면, 개념 컴포넌트 c에는 두 개의 포트 p1과 p2가 정의되어 있다고 가정할 때, c에는 필수 휘처 f1이, p1과 p2에는 각각 택일 휘처 f2와 f3가 대응되었고p1과 p2와 관련된 메시지들이 일부 가변적 요소를 가진다면 이들 포트는 택일적-화이트박스 가변점으로 결정된다.

마지막으로, 다섯 번째 규칙은 포트 p에 대응된 휘처 들이 모두 필수적 휘처 만을 포함하거나 포트p에 대응된 휘처 들이 p를 포함하는 컴포넌트에 대응된 휘처 집합의 부분집합이어서, p는 이를 포함한 컴포넌트가 존재하면 항상 필수적으로 존재해야 하지만, 포트 p와 관련된 메시지들이 가변점을 포함하게 된다면 이 포트의 가변성은 VP로 결정됨을 의미한다.

포트의 가변성 결정 규칙과 유사하게, 개념 컴포넌트의 가변성은 다음 규칙3에 의해서 결정된다.

규칙 3 개념 컴포넌트의 가변성 결정 규칙

주어진 개념 컴포넌트의 c에 대해서,

1. $\exists c1 \in COMPONENT \cdot AlternativeFGroup(Features(c), Features(c1)) \wedge (Container(c) = Container(c1)) \wedge MandatoryElms(SubElms(c)) \Rightarrow c.variability = VA$
2. $VariableFeatures(Features(c) - Features(Container(c)) \wedge \neg\exists c1 \in COMPONENT \cdot AlternativeFGroup(Features(c), Features(c1)) \wedge (Container(c) = Container(c1)) \wedge MandatoryElms(SubElms(c)) \Rightarrow c.variability = VO$
3. $\exists c1 \in COMPONENT \cdot AlternativeFGroup(Features(c), Features(c1)) \wedge (Container(c) = Container(c1)) \wedge VariableElms(SubElms(c)) \Rightarrow c.variability = VAP$
4. $VariableFeatures(Features(c) - Features(Container(c)) \wedge \neg\exists c1 \in COMPONENT \cdot AlternativeFGroup(Features(c), Features(c1)) \wedge (Container(c) = Container(c1)) \wedge VariableElms(SubElms(c)) \Rightarrow c.variability = VOP$
5. $MandtoryFeatures(Features(c)) \vee (Features(c) \subseteq Features(Container(c)) \wedge VariableElms(SubElms(c)) \Rightarrow c.variability = VP$

$$CComp(p) \equiv \begin{cases} AC(p) & if\ Type(p) = PrimtiveProcess \\ \cup_{\forall c \in SubComp(p)} AC(c) & if\ Type(p) = ProcessGroup \end{cases}$$

규칙 3에서 사용된 술어함수SubElms(c)는 개념 컴포넌트 c에 포함된 포트, 컴포넌트 행위 혹은 서브 개념 컴포넌트들의 집합을 의미한다. 규칙 3의 규칙들은 규칙 2의 것과 매우 유사하므로 자세한 설명은 생략한다. 다만 차이점으로는 개념 컴포넌트 c에 포함된 컴포넌트 행위, 포트, 혹은 서브 개념컴포넌트 들이 가변성을 가지느냐에 따라서, 화이트박스 가변점 인지 블랙박스 가변점 인지가 결정된다.

프로세스 아키텍처 모델의 가변성을 결정하기 위해서 사용되는 함수 CComp(p)는 프로세스 컴포넌트 p 가 기본 프로세스라면, p에 할당된 개념 컴포넌트들을 반환하고, 프로세스 그룹이라면 프로세스 그룹에 포함된 모든 프로세스 그룹 컴포넌트에 할당된 개념 컴포넌트들을 반환한다. 여기서 함수 AC(p)는 프로세스 컴포넌트 p에 할당된 개념 컴포넌트들의 집합을, SubComp(p)는 프로세스 컴포넌트 p에 포함된 서브 프로세스 컴포넌트들의 집합을 반환한다.

규칙 4 프로세스 컴포넌트의 화이트박스 가변성 결정 규칙
프로세스 컴포넌트 p에 대해서,

1. $\exists p1 \in PCOMPONENT \cdot AlternativeFGroup(Features(p), Features(p1)) \wedge (Container(p) = Container(p1)) \wedge MandatoryElms(CComp(p)) \Rightarrow p.variability = VA$
2. $VariableFeatures(Features(p) - Features(Container(p)) \wedge \neg\exists c1 \in PCOMPONENT \cdot AlternativeFGroup(Features(p), Features(p1)) \wedge (Container(p) = Container(p1)) \wedge MandatoryElms(CComp(p)) \Rightarrow p.variability = VO$

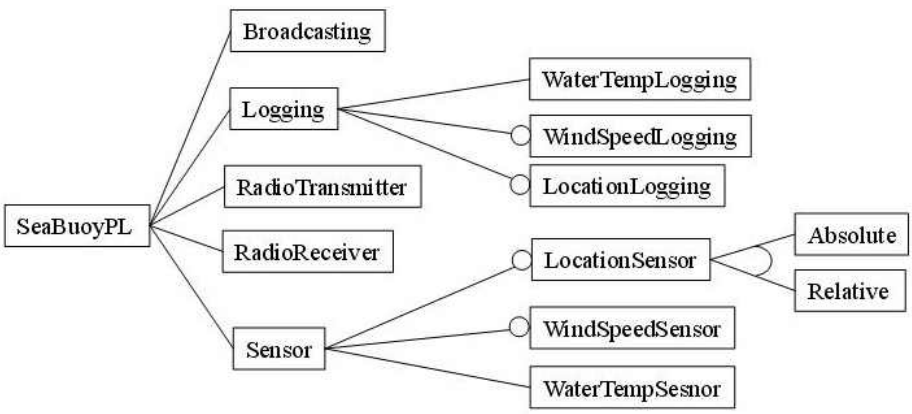
규칙 4는 프로세스 컴포넌트의 블랙박스 가변성을 결정하는 규칙만을 나열하였다. 화이트박스 가변성을 결정하는 방식은 규칙 2와 규칙 3에서와 유사하게, 규칙 4의 첫 번째와 두 번째와 규칙에서 MandatoryElms(CComp(p)) 대신에 VariableElms(CComp(p))을 대치하면 되므로 생략한다. 규칙 4의 첫 번째 규칙은 프로세스 컴포넌트 p에 대응된 휘처 집합과 택일적 관계에 있는 휘처 집합이 p와 동일한 포함 계층 수준의 다른 프로세스에 대응된 것이 있고, 포트 p와 관련된 메시지들이 모두 필수적인 요소라면 p의 가변성을 택일적-블랙박스로 결정함을 나타내고, 두 번째 규칙은 프로세스 컴포넌트 p에 대응된 휘처들(Features(p))이 가변 휘처를 포함하고 있고, p와 택일적 관계에 있는 동일 수준의 다른 프로세스 컴포넌트가 없고, p와 관련된 개념 컴포넌트들이 모두 필수적인 요소라면 p의 가변성은 선택적-블랙박스로 결정됨을 의미한다.

4. 사례 연구

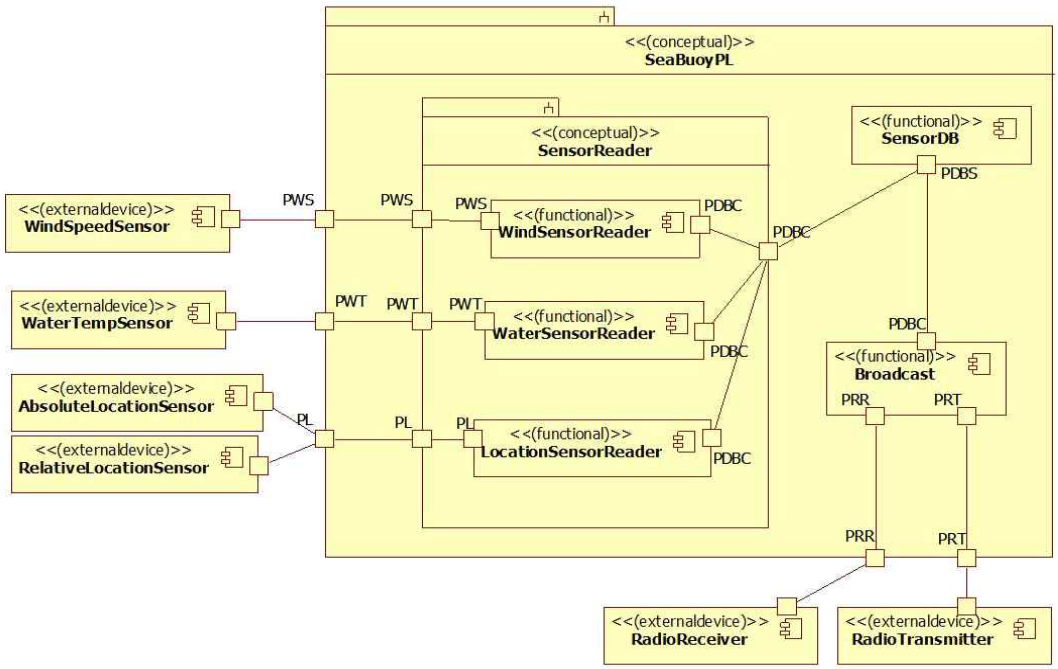
본 장에서는 제안된 방법을 예시하고 실무에 적용 가능성을 테스트하기 위해서, 바다 부표 시스템 (Sea Buoy System)에 대한 사례연구를 설명한다.

4.1 휘처 모델링

휘처 모델링은 프로덕트 라인의 공통성과 가변성을 휘처 단위로 구조화한 휘처모델을 산출하는 활동이다. (그림 5)는 바다 부표 시스템의 휘처모델이다. 바다 부표 시스템은 기능(Capability) 휘처로 Logging과 Broadcasting 이 있고, WindSpeedLogging과 LocationLogging은 Logging의 선택적 서브 휘처로 모델링 되었다. 이는 제품에 따라 선택 가능한 가변성을 나타낸다. 이와 유사하게 WindSpeedSensor와 LocationSensor도 선택적 휘처로서 운영환경 (Operating Environment)의 선택적 가변성을 표현하였다. 한편, LocationSensor의 서브 휘처로 Absolute와 Relative는 택일적 휘처로서, 둘 중에 하나는 반드시 하나의 제품을 위해서 선택되어야 하는 가변성을 나타낸다.



(그림 5) 휘처 모델 예



(그림 6) 개념 아키텍처 모델 예

4.2 아키텍처 모델링

앞서 언급하였듯이, 본 연구에서는 FORM 아키텍처 모델을 UML 모델로 대응시켜서 모델링 한다. 따라서 본 연구에서는 FORM 아키텍처 모델링을 위해 기존에 존재하는UML 모델링 도구인 StarUML [20]을 이용하였다.

(그림 6)은 StarUML을 이용하여 모델링 된 개념 아키텍처이다. SeaBuoyPL은 최상위 개념 컴포넌트로 UML 서브시스템 요소로 모델링 되었고, 이것이 개념 컴포넌트임을 나타내기 위하여 스테레오타입으로 <<(conceptual)>>이라 명명하였다. SeaBuoyPL은 SensorReader, SensorDB, Broadcast의 개념 컴포넌트로 분해되고, SensorReader는 다시 WindSensorReader, WaterSensorReader, LocationSensor Reader로 분해된다. 컴포넌트 분해 계층의 최말단 컴포넌트는 기능 컴포넌트이므로, WindSensorReader나 LocationSensorReader 등은 <<(functional)>> 스테레오타입으로 표현되었다. 이들 개념 컴포넌트는 다른 개념 컴포넌트와 상호작용하기 위해서 컴포넌트 경계에 붙어있는 포트를 통해서 연결된다.

4.3 휘처-아키텍처 대응 및 아키텍처 가변성 결정

휘처와 아키텍처 모델요소를 대응시키기 위해서는 UML 모델로부터 아키텍처 모델 요소를 추출하고, 각 아키텍처 모델 요소 별로 해당되는 휘처를 대응시키는 과정으로 진행된다. <표 2>의 첫 번째와 세 번째 열은 추출된 아키텍처 모델요소에 휘처를 대응시킨 결과를 보여준다. 예를 들면 SeaBuoyPL 개념 컴포넌트는 개념 아키텍처의 최상위 컴포넌트이고, 이 개념 컴포넌트에는 SeaBuoyPL 휘처가 대응되었다. 한편, SeaBuoyPL 개념 컴포넌트의 서브 컴포넌트로 SensorReader가 있고, 이 컴포넌트에는 Logging 휘처가, 다시 SensorReader 개념 컴포넌트의 서브 컴포넌트로 LocationSensorReader가 있고, 이 개념 컴포넌트에는 LocationLogging 휘처가 대응되었다. 또한 LocationSensorReader 컴포넌트는 두 개의 포트 PL과 PDBC를 포함하고 있고, 이들 포트도 LocationLogging 휘처와 대응관계를 가지고 있다. 마지막으로 포트 PL은 이를 통해 외부로 나가는 메시지를 나타내는 두 개의 요구 인터페이스 인 IALSensor와 IRLSensor와 연관되어 있고, 이들은 공통으로 대응된 LocationLogging 휘처와 함께 Absolute와 Relative 휘처가 각각 대응되어 있다.

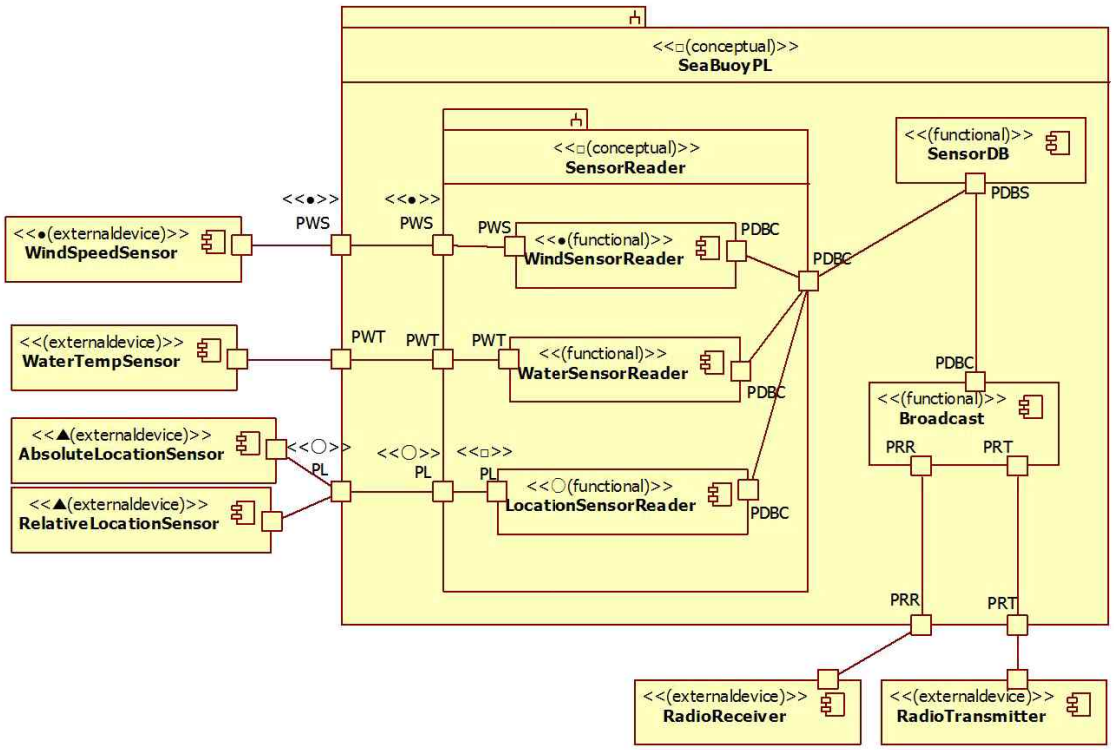
이와 같은 휘처 아키텍처 대응과 휘처의 가변성 정보를 바탕으로 3.2.3 절의 규칙을 적용한 결과는 <표 2>의 두 번째 열과 같다. 먼저, 두 개의 요구 인터페이스 IALSensor와 IRLSensor는 각각 택일적 휘처가 하나씩 대응되었고, 이들은 동일한 포트 PL과 관련이 되어 있으므로, 규칙 1의 두 번째 규칙에 의해 VA로 가변성이 결정되었다.

한편, 포트 PL은 선택적 휘처인 LocationLogging과 대응되어 있으나 이 포트를 포함한 컴포넌트도 동일한 휘처인 LocationLogging과 대응되어 있으므로, 이 포트 자체의 가변성은 필수적 요소이나, 이 포트 내부의 관련된 메시지들이 VA 가변성을 가지므로 규칙 2의 다섯 번째 규칙에 의해서 VP로 결정되었다.

개념 컴포넌트 LocationSensorReader도 포트 PL과 유사하게 선택적 휘처인 LocationLogging이 대응되어 있어 있지만, 이 컴포넌트를 포함하는 개념 컴포넌트 SensorReader에 대응되는 휘처가 필수적 휘처인 Logging이므로 이 컴포넌트 자체는 선택적 가변성을 가지게 된다. 하지만, 이 컴포넌트 내부에 가변성을 지니는 포트 PL을 포함하므로, 규칙 3의 네 번째 규칙에 의해서 VOP로 결정되었다.

〈표 2〉 휘처-아키텍처 대응과 가변성 결정

Architecture Elements					variability	Features	variability
(ConceptualComponent)SeaBuoyPL					VP	SeaBuoyPL	mandatory
1	(Port)PWS				VO	WindSpeedLogging	optional
		(RInterface)IWSSensor			M	WindSpeedLogging	optional
	(Port)PWT				M	WaterTempLogging	mandatory
		(RInterface)IWTSensor			M	WaterTempLogging	mandatory
	(Port)PL				VOP	LocationLogging	optional
		(RInterface)IALSensor			VA	LocationLogging	optional
						Absolute	alternative
		(RInterface)IRLSensor			VA	LocationLogging	optional
						Relative	alternative
	(Port)PRR				M	SeaBuoyPL	mandatory
		(RInterface)IRequest			M	SeaBuoyPL	mandatory
	(Port)PRT				M	SeaBuoyPL	mandatory
		(RInterface)ITransmitter			M	SeaBuoyPL	mandatory
	(ConceptualComponent)SensorReader				VP	Logging	mandatory
		(Port)PWS			VO	WindSpeedLogging	optional
			(RInterface)IWSSensor		M	WindSpeedLogging	optional
		(Port)PWT			M	WaterTempLogging	mandatory
			(RInterface)IWTSensor		M	WaterTempLogging	mandatory
		(Port)PL			VOP	LocationLogging	optional
			(RInterface)IALSensor		VA	LocationLogging	optional
						Absolute	alternative
			(RInterface)IRLSensor		VA	LocationLogging	optional
						Relative	alternative
		(Port)PDBC			M	Logging	mandatory
			(RInterface)IDB		M	Logging	mandatory
		(FunctionalComponent)WindSensorReader			VO	WindSpeedLogging	optional
			(Port)PWS		M	WindSpeedLogging	optional
				(RInterface)IWSSensor	M	WindSpeedLogging	optional
			(Port)PDBC		M	WindSpeedLogging	optional
				(RInterface)IDB	M	WindSpeedLogging	optional
		(FunctionalComponent)WaterSensorReader			M	WaterTempLogging	mandatory
			(Port)PWT		M	WaterTempLogging	mandatory
				(RInterface)IWTSensor	M	WaterTempLogging	mandatory
			(Port)PDBC		M	WaterTempLogging	mandatory
				(RInterface)IDB	M	WaterTempLogging	mandatory
		(FunctionalComponent)LocaionSensorReader			VOP	LocationLogging	optional
			(Port)PL		VP	LocationLogging	optional
				(RInterface)IALSensor	VA	LocationLogging	optional
						Absolute	alternative
				(RInterface)IRLSensor	VA	LocationLogging	optional
						Relative	alternative
			(Port)PDBC		M	LocationLogging	optional
				(RInterface)IDB	M	LocationLogging	optional
	(ConceptualComponent)SensorDB				M	Logging	mandatory
		(Port)PDBS			M	Logging	mandatory
			(PInterface)IDB		M	Logging	mandatory
	(ConceptualComponent)Broadcast				M	Broadcasting	mandatory
		(Port)PDBC			M	Broadcasting	mandatory
			(RInterface)IDB		M	Broadcasting	mandatory
		(Port)PRR			M	Broadcasting	mandatory
			(RInterface)IRequest		M	Broadcasting	mandatory
		(Port)PRT			M	Broadcasting	mandatory
			(RInterface)ITransmitter		M	Broadcasting	mandatory



(그림 7) 가변성 적용 후 아키텍처 모델

개념 컴포넌트 SensorReader는 필수적인 휘처인 Logging과 관련되어 있으므로 그 자체는 필수적인 요소이나, WindSensorReader와 LocationSensorReader를 내부의 가변적인 아키텍처 요소로서 가지고 있으므로, 규칙 3의 다섯 번째 규칙에 의해서 VP 로 결정되었다. 마지막으로 개념 컴포넌트SeaBuoyPL도 SensorReader와 동일한 방식으로 VP로 결정되었다.

(그림 7)은 <표 2>의 가변성 결정을 바탕으로, 개념 아키텍처의 각 아키텍처 요소에 가변성 정보를 스테레오 타입으로 나타낸 것이다. (그림 7)에서 보는 바와 같이 각 아키텍처 요소가 선택적 가변성을 가지는지 택일적 가변성을 가지는 지와 함께 내부에 가변요소가 있는지 혹은 없는지를 나타내는 다양한 가변성 정보를 간단한 형태의 스테레오타입으로 표현함으로써, 프로덕트 라인 아키텍처 모델의 가변성 정보를 효율적으로 나타낼 수 있다.

본 연구에서는 아키텍처 모델의 가변성 정보가 휘처모델과의 대응 관계 설정으로부터 자동으로 추출되어 결정되므로, 휘처모델의 가변성이 변경되었을 때 아키텍처 모델의 가변성이 자동으로 변경되어, 휘처모델의 가변성과 아키텍처 모델의 가변성을 일관적으로 유지시킬 수 있는 장점이 있다.

5. 관련 연구

프로덕트 라인 아키텍처의 가변성을 모델링하고 관리하는 일은 프로덕트 라인 공학의 필수적인 활동이다. 프로덕트 라인 아키텍처의 가변성을 표현하고 관리하는 종래의 방법은 크게 두 가지로 구분된다. 첫째는 프로덕트 라인 아키텍처 모델 자체에 가변성을 표현하고 관리하는 방법[1, 6, 14, 17] 이고, 둘째는 프로덕트 라인 아키텍처 모델 자체에는 가변성 정보를 직접 표현하지 않고, 가변성 정보는 별도의 독립 가변성 모델 (휘처모델[5] 혹은 OVM [3])로 표현하되, 이 가변성 모델과 프로덕트 라인 아키텍처 모델에 명시적인 대응 관계를 통해서 프로덕트 라인 아키텍처의 가변성을 관리하는 방법 [3, 5, 12]이다.

첫 번째 방법의 예로는 UML의 메타모델을 가변성 관점에서 확장하여 프로덕트 라인 아키텍처의 가변성을 표현하는 방법 [14]이 있고, UML의 확장 메커니즘인 스테레오 타입을 이용하여 가변성을 표현하는 방법 [1, 6, 17]이 있다. KobrA [1] 는 프로덕트 라인 공학과 컴포넌트 기반 소프트웨어 개발을 통합한 방법으로서, 컴포넌트의 가변성을 표현하기 위해서 <<variant>> 스테레오 타입을 이용하였고, Ziadi의 연구 [17] 에서는 UML 클래스 다이어그램과 UML 시퀀스 다이어그램을 프로덕트 라인 모델로 정의하기 위해서, UML 모델링 요소의 가변성을 선택적 가변성과 가변점-가변치의 두 가지 방법으로 표현하는 방법을 제안하였다. 선택적 가변성을 지니는 UML 모델링 요소는 <<optional>>이란 스테레오타입으로 정의하고, 가변점은 <<Variation>>으로 가변치는 <<variant>>로 모델링 한다. Gomaa [6]도 이와 유사하게 스테레오타입을 이용하여 프로덕트 라인 모델 요소의 가변성 표현하였지만, Ziadi의 연구와는 달리 아키텍처 단계의 모델을 이용하였다.

이러한 방법들의 문제점은 프로덕트 라인 아키텍처 모델에 표현된 가변성과 프로덕트 라인 분석 모델 (예, 휘처 모델)에 표현된 가변성 간에 일관성이 별도로 관리되어야 한다는 것이다. 본 논문에서도 이들 연구와 마찬가지로 스테레오타입과 태그값을 이용하여 아키텍처 모델의 가변성을 표현하였지만, 계층적으로 표현되는 아키텍처 모델의 가변성을 보다 효과적으로 표현하기 위하여 블랙박스 가변성과 화이트박스 가변성의 개념을 도입한 점이 하나의 주목할 만한 특징이라고 볼 수 있다.

두 번째 방법은 프로덕트 라인 아키텍처 모델 자체에는 가변성이 표현되지 않으므로, 기존의 단일 시스템 개발에서 사용되어 왔던 모델 (예, UML 모델)을 그대로 사용할 수 있을 뿐만 아니라, 별도의 독립 가변성 모델 (휘처 모델 혹은 OVM)에 표현된 가변성 정보를 바탕으로 일관적으로 프로덕트 라인 아키텍처의 가변성을 관리할 수 있는 장점은 있지만, 프로덕트 라인 아키텍처 모델에는 가변성이 명시적으로 표현되지 않는 문제점이 있다.

Czarnecki [3] 는 UML 2.0의 액티비티 모델과 클래스 모델의 일부 요소에 휘처모델의 휘처를 *존재 조건(Presence Condition)* 으로 첨부시키고, 휘처모델의 휘처 선택에 따라서 선택된 휘처가 존재조건으로 첨부된 모델 요소는 포함시키고 그렇지 않는 모델 요소는 제거시킴으로써, 액티비티 모델과 클래스 모델의 가변성을 관리하는 방법을 제안하였다. 하지만, 이 방법은 액티비티 모델이나 클래스 모델로 표현된 프로덕트 라인 모델의 가변성이 명시적으로 표현되지는 못한다는 단점이 있다.

Lee와 Muthig [12] 연구는 휘처모델을 이용하여 문제공간의 가변성을 휘처 혹은 바인딩 단위 (Binding Unit)단위 표현하고, 프로덕트 라인 컴포넌트 모델을 통해서 해결공간의 가변성을 컴포넌트 단위로 모델링 한 후에, 의사결정 테이블(Decision Table)을 통해서 문제공간의 가변성 단위 (즉, 휘처 혹은 바인딩 단위)를 해결공간의 가변성 단위 (즉, 컴포넌트)와 연결함으로써, 휘처 관점에서 소프트웨어 개발 생명주기 전반을 관리할 수 있는 방법을 제안하였다.

이들의 연구는 Czarnecki [3] 연구와는 달리 프로덕트 라인 모델의 가변성을 명시적으로 표현할 뿐만 아니라, 휘처 모델과 프로덕트 라인 모델의 대응이 별도의 독립적인 의사결정 테이블을 통해서 이루어진다는 점에서 본 논문과 공통점이 있다. 하지만 이들의 연구에서는 프로덕트 라인 컴포넌트 모델에 표현된 가변성이 모델 작성자에 의해서 결정된 것이며, 이것은 의사결정 테이블에 정의된 정보와 일관성을 유지해야 한다. 이와 같이 프로덕트 라인 모델의 가변성이 휘처모델과 별개로 모델링된 경우에는 휘처모델의 가변성과 프로덕트 라인 모델과의 가변성이 일관적으로 정의되었는지를 검사하는 것이 필수적이지만, 본 연구에서는 프로덕트 라인 모델의 가변성이 휘처 모델과의 대응 정보로부터 자동으로 결정되므로, 별도의 일관성 검사없이 프로덕트 라인 모델의 가변성이 표현되고 관리될 수 있는 장점이 있다.

본 논문에서는 앞의 두 가지 방법의 장점을 결합하여 휘처와 아키텍처 모델요소의 대응관계와 휘처의 가변성 정보를 바탕으로 아키텍처 모델요소의 가변성을 결정하고 표현하는 방법을 제안하였다. 특히, 본 연구에서는 산업계의 표준으로 자리잡고 있는 UML을 기반으로 프로덕트 라인 모델의 가변성을 표현하는 방법을 채택함으로써, 기존의 UML 기반 상용 모델링 도구를 활용하여 프로덕트 라인 아키텍처 모델을 작성할 수 있는 장점이 있다.

6. 결론 및 향후 연구

본 논문에서는 FORM 아키텍처 모델을 표준화된 UML 모델링 요소로 대응시키고 FORM 아키텍처 모델의 가변성을 UML의 확장 메커니즘을 이용하여 표현하는 방법을 제안하였다. 특히, FORM 아키텍처 모델의 가변성이 대응되는 휘처모델로부터 결정되는 기법을 개발하였다.

본 논문의 주된 공헌은 다음과 같다.

- 복잡한 계층적 아키텍처 모델의 가변성을 효과적으로 표현하기 위해 블랙박스 가변성과 화이트박스 가변성의 개념을 제안하고, 이를 UML의 스테레오타입을 이용하여 간결하게 나타낼 수 있는 모델링 기법을 제안하였다. 이는 간단하면서도 다양한 가변성 정보를 아키텍처 모델에 반영시키게 함으로써 작업의 효율을 도모 할 수 있게 하였다.
- 휘처모델과의 대응관계를 통해서 아키텍처 모델의 가변성이 결정되는 기법을 개발함으로써, 휘처모델과 아키텍처 모델 사이의 가변성이 일관적으로 관리될 수 있는 장점이 있다.

제안된 방법에서의 한 가지 가정은 휘처와 아키텍처 모델요소와의 대응이 올바르게 설정되었다는 가정을 내포하고 있다. 만약 이러한 가정이 맞지 않다면, 휘처-아키텍처 대응으로부터 유도되는 아키텍처 모델의 가변성이 잘못 결정될 수 도 있다. 따라서 향후에는 휘처와 아키텍처 모델요소와의 대응에 문제가 없는 지를 검사하는 기법을 개발할 예정이다. 또한 도구적인 측면에서는 현재까지 개발된 도구는 UML모델에 첨부된 태그값의 정보를 시각적으로 표시하지 못하는 단점이 있다. 따라서 태그값으로 표시되는 가변성 정보를 효과적으로 표현해주기 위해서는 UML모델 외에 다른 시각화 기법이 필요하다.

참 고 문 헌

[1] C. Atkinson, J. Bayer, C. Bunse, O. Laitenberger, R. Laqua, E. Kamsties, D. Muthig, B. Paech, J. Wüst, and J. Zettel, *Component-based Product Line Engineering with UML*, Component Series, Addison-Wesley, 2001.

[2] P. Clements, L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.

[3] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in Proceedings of GPCE, 2005.

[4] K. Czarnecki, U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.
[5] E. Gamma,R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[6] H. Gomaa, Designing Software Product Lines with UML, Addison-Wesley, 2004.
[7] M. L. Griss, J. Favaro, M. d'Allesandro, "Integrating Feature Modleing with the RSEB", In Proceedings of the 5th International Conference on Software Reuse, pp.76-85, 1998.
[8] K. C. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", *Technical Report CMU/SEI-90-TR-21*, Software Engineering Institute, Carnegie Mellon University, 1990.
[9] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, M., "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", *Annals of Software Engineering*, 5, pp.143-168, 1998.
[10] K. C. Kang, S. Kim, J. Lee, K. Lee, "Feature-Oriented Engineering of PBX Software for Adaptability and Reuseability", Software: Practice and Experience, Vol.29, Issue10, pp.875-896, 1999.
[11] K. C. Kang, J. Lee, P. Donohoe, "Feature-Oriented Product Line Engineering", IEEE Software, Vol.9, No.4, pp.58-65, 2002.
[12] J. Lee and D. Muthig, "Feature-Oriented Variability Management in Product Line Engineering Implementing Feature-Oriented Variability Modeling Throughout the Life Cycle", Communication of the ACM, Vol.29, No.12. pp.55-59, December, 2006.
[13] K. Lee, K. C. Kang, E. Koh, W. Chae, "Domain Oriented Engineering of Elevator Control Software-A Product Line Practice", In Proceedings of the First Software Product Line Conference, pp.3-22, 2000.
[14] D. Muthig and C. Atkinson, "Model-Driven Product Line Architectures", G. Chastek (Ed.): SPLC2 2002, LNCS 2379, pp.110-129, 2002.
[15] K. Pohl, G. Böckle, F. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer, 2005.
[16] P. Sochos, I. Philippow, M. Riebisch, "Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture", LNCS 3263, pp.138-152, 2004.
[17] T. Ziadi, L. H´elou¨et, and J.-M. J´ez´equel, "Towards a UML Profile for Software Product Lines," in Workshop on Product Familiy Engineering (PFE), pp.129 - 139, 2003.
[18] ASADAL Case Tool, http://selab.postech.ac.kr
[19] OMG Unified Modeling LanguageTM (OMG UML), Superstructure, http://www.omg.org/spec/UML/2.3/Superstructure
[20] StarUML, http://staruml.sourceforge.net/ko/

이 관 우

e-mail : kwlee@hansung.ac.kr
1994년 포항공과대학교 전자계산학과 (학사)
1996년 포항공과대학교 컴퓨터공학과 (공학석사)
2003년 포항공과대학교 컴퓨터공학과 (공학박사)
2003년～현 재 한성대학교 정보시스템공학과 부교수
관심분야 : 소프트웨어 제품계열 (Software Product Line), 관점지향 프로그래밍(Aspect-Oriented Programming), 소프트웨어 아키텍처