아키텍처 변환 패턴을 이용한 소프트웨어 시스템 진화 프레임워크†

(A Framework for Software System Evolution using Architectural Transformation Pattern)

박태현‡ (Taehyun Park) 안 휘‡ (Hwi Ahn) 강성원¶ (Sungwon Kang)
박종빈¥ (Jongbin Park) 황상철¥ (Sangcheol Hwang)

요 약 소프트웨어 시스템 진화는 소프트웨어 시스템의 유지보수의 일종으로 계획적이고 체계적인 유지보수 프로세스이다. 소프트웨어 진화 연구는 기존 시스템의 비용편익 분석을 통해 시스템의 유지 가치에 대한 판단 근거를 제공하며, 아키텍처를 기반으로 하는 진화는 반복적인 진화 작업의 자동화를 가능케 하여 유지보수 비용 감소를 가능하게 해주는 연구이다. 본 논문에서는 아키텍처 변환 패턴을 이용한 소프트웨어 시스템 진화 프레임워크를 제안한다.

키워드 아키텍처, 소프트웨어 시스템 진화 프레임워크, 아키텍처 변환 패턴

Abstract Software System Evolution is more planned and systematic maintenance process as well as a type of maintenance. The research of software evolution provides basis of decisions for maintenance value through cost-benefit analysis of legacy system and architecture-based software evolution enables engineers to reduce maintenance cost by automation of repetitive evolution tasks. This paper proposes a framework for software system evolution using architectural transformation pattern.

Key words Architecture, Software System Evolution Framework, Architecture Transformation Pattern

1. 서 론

소프트웨어 시스템을 이용하는 유저들의 트렌드와 요구사항들은 지속적이고 빠르게 변화하며, 회사는 생존을 위해 이러한 변화에 빠르게 대응하는 것이 중요하다 [1][2][6]. 그러나 이러한 변화에 매번 새로운 소프트웨어 시스템을 개발하여 대응하는 것은 개발 예산과 시간을 고려하였을 때 불가능하다. 소프트웨어 시스템 진화(Evolution)는 소프트웨어 시스템의 유지보수의 일종으로 계획적이고 체계적인 유지보수 프로세스이다[3]. 소프트웨어 진화 연구는 기존 시스템의 비용편익 분석을 통해 시스템의 유지 가치에 대한 판단 근거를 제공하며, 아키텍처의 변경 없이도 계획된 변화의 범위 안에서 저렴한 비용으로

† 본 연구는 2012년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(2012-0007069).
‡ 학생회원 : 한국과학기술원 전산학과
〔tpark, ahnhwi〕 @kaist.ac.kr
¶ 종신회원 : 한국과학기술원 전산학과 부교수
sungwon.kang@kaist.ac.kr
¥ 비 회 원 : SK Planet Software Quality Engineering 팀
jongbin@sk.com, k16wire@gmail.com

진화가 가능한 장점을 갖는다. 그렇기 때문에, 기존의 소프트웨어 시스템을 진화시키는 기술은 이러한 변화들에 대응하기 위해 필수적이다 [1][2][4][5].

본 논문에서는 소프트웨어 시스템의 진화 과정의 자동화를 가능하게 해줄 수 있는 아키텍처 변환 패턴을 이용한 소프트웨어 시스템 진화 프레임워크를 제안한다. 아키텍처 변환 패턴을 사용함으로써 회사마다 또는 도메인에 따라 다르게 표현될 수 있는 소프트웨어 시스템 아키텍처와 각 시스템의 진화 방식을 쉽고 효율적으로 표현할 수 있다.

아키텍처 기반의 소프트웨어 기술은 정형화된 아키텍처 모델을 통해 소프트웨어 시스템을 관리, 분석 및 진화를 가능케 해주고, 반복적인 진화 작업의 자동화를 달성할 수 있게 해준다. 이를 통해 소프트웨어 시스템 진화를 체계적으로 진행할 수 있으며, 소프트웨어 시스템의 유지보수 비용 감소를 기대할 수 있다.

또한 본 논문에서는 기존의 아키텍처 기반의 소프트웨어 시스템 진화 기술과 그리고 아키텍처 변환 기술에 대해 고찰하고, 제안하는 소프트웨어 진화 프레임워크를 지원하는 툴로서 ASES(Automatic Service Evolution System)를 소개함으로써 진화 프레임워크를 설명하고 이를 프로토타입 형태로 제공한다.

본 논문의 구성은 다음과 같다. 1장의 서론에 이어 2장에서는 본 논문에서 제안하는 아키텍처 기반의 소프트웨어 시스템 진화의 관련 연구로서, CMU에서 제안한 자가 적응형 소프트웨어 프레임워크를 제안한 Rainbow(레인보우) 프레임워크, SEI에서 제안한 Simplex 아키텍처 연구를 소개한다. 3장에서는 아키텍처 기반의 소프트웨어 시스템 진화 프레임워크 제안에 필요한 소프트웨어 아키텍처 표현법, 아키텍처 표현 언어(ADL, Architecture Description Language), 그리고 아키텍처 변환 패턴 구현에 이용된 아키텍처 변환 명령어들을 제안한 Transformational Architecture Design(TAD)에 대해서 논의한다. 4장에서는 본 논문에서 제안하는 아키텍처 기반의 소프트웨어 시스템 진화 프레임워크를 설명한다. 5장에서는 4장에서 제시한 소프트웨어 시스템 진화 프레임워크를 활용한 사례 연구를 프로토타입 형태로 소개하며, 6장에서 결론 및 향후 연구를 제시한다.

2. 관련 연구

이 절에서는 본 연구에서 제안하는 아키텍처 변환 패턴을 이용한 소프트웨어 시스템 진화 프레임워크와 관련된 기존 연구들을 소개하고, 연구들이 갖는 문제점과 한계에 대해서 대해서 고찰한다.

기존의 아키텍처 기반의 소프트웨어 진화 관련 연구로는 90년대 후반에 Oreizy et al[4][5]이 소프트웨어 아키텍처 모델에 런타임상의 변화를 반영하는 프로세스를 제시하였고 이를 기반으로 K-Component Architecture Meta-model[13], Model-based development[14], Object-oriented design adaptation[15] 등 많은 연구가 진행되었다.

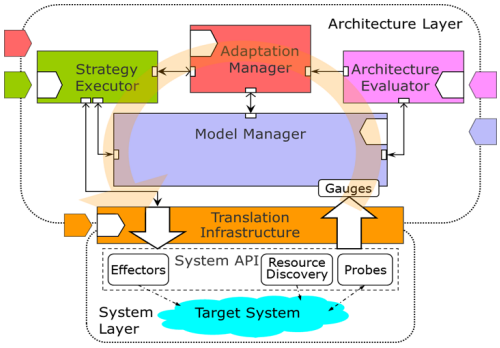
본 논문에서는 그 중에 Rainbow Framework가 제시하는 진화 프로세스 구조와 Simplex Architecture에서 아키텍처 요소들이 진화하는 자세한 프로세스에 대해서 고찰한다.

2.1 Rainbow Framework(레인보우)

Rainbow Framework[1]는 2004년 카네기 멜론 대학교의 D. Garlan과 S. Cheng이 제안한 프레임워크로 추상화된 아키텍처 모델을 사용하여 현재 동작중인 시스템의 런타임 속성을 모니터링 하고 모델을 평가하여 문제가 발생하였을 때 동작 중인 시스템에 적절한 적응을 수행할 수 있도록 한다. 현재 Rainbow, RAIDE[2], Stitch[12] 등 런타임 상에서의 소프트웨어 시스템의 아키텍처 기반의

자가적응 연구들이 카네기 멜론 대학교의 ABLE Project를 통해 활발히 진행되고 있다.

Rainbow 프레임워크의 기능은 다음과 같다. [그림 1]에서 보여주듯이, 소프트웨어 시스템 및 환경의 상태를 모니터링 해주는 모니터링 파트는 다양한 종류의 Probe로 구성되어 있다. 그리고 Gauges는 Model Manager에서 관리가 되는 아키텍처 모델의 속성에 따라 관찰을 하는 부분이다. 새로운 업데이트, 즉 변화(Change)가 발생 했을 시, Architecture Evaluator가 아키텍처 모델에서 정의된 허용치 안에서 시스템이 제대로 구동되고 있는지 확인한다. 만약에 해당 시스템이 정의된 허용치(e.g. 성능) 안에서 적절히 구동되지 못하고 있다면, Adaptation Manager가 정의된 적응 작업을 결정하도록 반응한다. 그리고 마지막으로 Executor가 선택된 자가 적응 작업들을 시스템 레벨의 Effector를 통해 런타임중인 시스템에 적용하여 변화에 대해 대응하여 시스템이 안정된 상태로 구동한다.



[그림 1] Rainbow 자가 적응 프레임워크[1]

Rainbow 프레임워크가 런타임상의 시스템에서 발생하는 변화에 대해 아키텍처 관점에서의 적응 방법을 잘 제시하고 있다. 그러나 Rainbow 프레임워크는 제시하고 있는 구조와 각 요소들의 역할에 맞춰 시스템의 아키텍처 요소가 연결되어야 한다. 그러므로 시스템의 설계 및 개발 과정에서부터 이 점을 고려하여 시스템에 적용되어 있지 않다면 프레임워크에서 제시하는 아키텍처 기반의 자가 적응 기능 발휘가 어려울 수 있다. 예를 들어, 기존에 운영되고 있는 기존 시스템의 아키텍처 대한 주의 깊은 고려 없는 Rainbow 프레임워크 적용은 다양한 문제를 발생시킬 수 있다. 또한 시스템의 변화가 감지되어 적응 전략(Adaptation Strategy)의 형태와 이 전략을 Effector를 통해 시스템에 적용할 때 어떤 형태로 적용하는지에 대한 자세한 설명이 부족하여 실제적으로 적용하기에 어려움이 있다.

2.2 Simplex Architecture

Simplex Architecture는 지속적으로 작동하는 소프트웨어 시스템에서 장애 허용(Fault tolerance) 개념과 작동 중 하드웨어와 소프트웨어 컴포넌트들의 동적 상태에서의 업그레이드(Dynamic upgrade)를 가능하게 하는 프레임워크로써, 1995년 카네기 멜론 대학교의 Software Engineering Institute(SEI)에서 개발되었다[9]. Simplex는 주로 임베디드 시스템에 적용되어 테스트 되었으며, 2000년 초반까지 연구되다가 최근에는 연구 진행이 되지 않고 있는 상태이다.

Simplex Architecture는 업그레이드를 하고자 하는 소프트웨어(또는 하드웨어) 시스템을 컴포넌트와 커넥터의 관점에서 인식한다. Simplex Architecture에서 컴포넌트는 교체 단위(Replacement Unit)를 의미하며, 이는 정형화된 업그레이드 트랜잭션(transaction)을 통해 온라인으로 추가, 삭제, 통합이 가능한 런타임 컴포넌트를 뜻한다. 커넥터는 업그레이드 트랜잭션을 의미한다.

Simplex Architecture의 구조는 계층적으로 되어 있다. System Configuration Manager(SCM)은 다수의 Processor Configuration Manager(PCM)을 포함하고 있으며, 각 PCM은 Sub-system Module들을 가지고 있다. Sub-system Module들은 교체

단위인 Application unit들을 포함하고 있는데, Application unit들은 해당 소프트웨어 시스템의 서비스를 제공하는 단위이다. 이 외에 장애 허용 기능을 수행하는 Safety unit과 이들을 관리하는 Module Management Unit이 Sub-system Module에 속해 있으며, 이들은 PCM을 통해 교체가 가능하다.

Simplex Architecture에서 커넥터의 역할을 하는 업그레이드 트랜잭션은 교체 트랜잭션들로 구성되어 있다. 예를 들어, 하나의 교체 단위를 교체하기 위한 기본적인 교체 트랜젝션은 다음과 같은 순서로 진행된다.

새로운 교체 단위가 생성되고 대기 상태에 있는다.

새로운 입력 값들이 새로운 교체 단위에 제공이 된다. 이 새로운 교체 단위가 출력하는 출력 값들은 에러 유무에 대하여 면밀히 검토된다. 출력 값들은 검토만 되고 실제로 사용되지 않는다.

Module Management Unit은 검토되는 출력 값들이 문제가 없는지 확인하고 문제가 없고 안정적인 상태에 이르렀는지 확인한다.

과거의 교체 단위가 폐기된다.

Simplex Architecture는 계층적인 구조로 이뤄진 소프트웨어 및 하드웨어의 아키텍처 컴포넌트 교체방식을 기반으로 한 진화 방법을 소개하고 있다. 하드웨어와 소프트웨어를 같은 개념에서 다룸으로써 직관적인 형태로 교체 트래젝션과 같이 소프트웨어 진화 방법을 상세히 설명하고 있다. 그러나 Simplex Architecture는 회사나 소프트웨어 도메인에 따라 다르게 존재할 수 있는 진화 방식을 표현하기 힘들다.

3. 배경 지식

이 절에서는 본 연구에서 제안하는 아키텍처 변환 패턴을 이용한 소프트웨어 시스템 진화 프레임워크를 구성 요소의 기본이 되는 소프트웨어 아키텍처 표현 방법과 아키텍처 변환 패턴인 Transformational Architecture Design(TAD)를 설명한다.

3.1 소프트웨어 아키텍처

소프트웨어 아키텍처는 진화하고자 하는 대상이 되는 소프트웨어 시스템과 진화의 목표가 되는 소프트웨어 시스템을 논리적 수준에서 표현한다. 이는 진화의 출발점과 진화의 종착점을 물리적 수준(코드, 실제 파일들)에서 논리적 수준(아키텍처)으로 추상화시켜 준다. 이를 통해 변환 패턴이 소프트웨어 시스템 진화를 위해 적용될 수 있는 기반을 마련해준다.

소프트웨어 아키텍처는 물리적으로 보이거나 만져지지 않는 특성 때문에, 왜곡 없이 정확하게 표현하려면, 다양한 뷰포인트를 통하여 표현하여야 한다.

P. Kruchten의 4+1 아키텍처 뷰포인트 프레임워크[17]를 통해 논리 뷰(Logical view), 실행 뷰(Process view), 개발 뷰(Development view), 물리 뷰(Physical view), 그리고 시나리오(Scenario view)로 분류하였다. 각 뷰들의 간단한 정의는 아래와 같다.

- 논리 뷰: 기능과 요구사항 위주로 소프트웨어 아키텍처를 표현한다.
- 실행 뷰: 소프트웨어 시스템의 실행 시 모습을 표현한다.
- 개발 뷰: 소프트웨어 시스템을 구성하는 컴포넌트들, 또는 서브시스템들의 계층 관계를 표현한다.
- 물리 뷰: 소프트웨어가 배치된 물리적 환경과의 관계를 표현한다.
- 시나리오 뷰: 위의 4개의 뷰들이 하나의 통일된 소프트웨어 시스템을 구성하도록 한다.

SEI의 L. Bass외 연구진은 소프트웨어 아키텍처를 아래와 같은 3개의 아키텍처 뷰로 표현하도록 정의하였다[7].

- 모듈 뷰(Module view): 코드 측면에서 표현한 소프트웨어 아키텍처 뷰로써, 작은 모듈들의 기능적 측면을 고려하여 표현한다.
- 컴포넌트 앤 커넥터 뷰(Component and Connector view): 소프트웨어 시스템의 실행 시 모습을 표현한다.
- 배치 뷰(Allocation view): 소프트웨어가 실행되는 물리적 환경(하드웨어)을 표현한다.

각 아키텍처 뷰들은 소프트웨어 시스템의 서비스를 제공해주는 컴포넌트와 이들을 연결하는 커넥터, 그리고 여러 컴포넌트들과 커넥터가 연결된 형상(Configuration)으로 표현될 수 있다. 예를 들어, L. Bass 등이 제시한 모듈 뷰에서는 기능 모듈이 컴포넌트가 되고, 그들 간의 계층 관계가 커넥터가 된다.

아키텍처를 표현하기 위해서는 일반적인 문서 작업 도구(마이크로소프트 워드, 파워포인트, 한글 등)를 이용하여 컴포넌트, 커넥터를 그릴 수 있다. 또한, UML과 같이 소프트웨어 개발에 일반적으로 사용되는 산출물 표현 언어를 사용하여 아키텍처를 정의할 수도 있다. 하지만 좀 더 상세한 아키텍처 정의를 위해서는 미리 엄격하게 정의된 아키텍처 표현 언어인 ADL을 쓸 수 있다. ADL에는 Acme[4], eXtensible Architecture Description Language(xADL)[8] 등이 있다. 본 연구에서는 카네기 멜론 대학교에서 개발된 Acme를 사용하였다.

Acme는 컴포넌트(Component), 커넥터(Connector), 포트(port), 역할(role)을 기본 아키텍처 구성요소로 가진다. 컴포넌트, 커넥터, 포트의 정의는 앞서 설명된 일반적인 정의와 같다. 역할은 커넥터에 속하는 아키텍처 구성요소로써, 컴포넌트의 포트와 연결되는 지점을 의미한다. Acme는 이클립스 기반의 전용 툴이 존재하며, 이를 통해 시각적인 기호와 텍스트 기반의 기호를 모두 제공한다. 또한 Acme를 사용하여 개발하기 위한 자바 기반의 라이브러리를 제공한다.

3.2 Transformational Architecture Design(TAD)

S. Kang[10]은 소프트웨어 아키텍처 설계를 위한 변환명령어를 제시한다. 기존의 소프트웨어 아키텍처 디자인에서는 먼저 아키텍처부터 설계된 후에 문법적인 유효성(syntactic validity)이 검정된다. 반면에 Transformational Architecture Design (TAD) 방법에서는 유효(valid)한 아키텍처가 설계될 수 있도록 정확하게 정의된 변환 명령어들(transportation operations)이 함께 설계된다. 또한 이 변환명령어들은 자동화된 아키텍처 디자인도 가능케 한다. 하지만 아키텍트는 문법적(Syntactic)으로 유효한 아키텍처뿐만 아니라 의도한 의미(Semantic)까지 정확히 반영하는 아키텍처 설계를 필요로 한다.

소프트웨어 아키텍처 설계가 문법적으로 유효하고 의도한 의미까지 반영되도록 하는 TAD 방법에서는 소프트웨어 아키텍처 디자인에 대한 변환 명령어들과 시스템을 구성하는 컴포넌트가 아키텍처에서 의도된 서비스 제공 여부를 체크하는 알고리즘을 함께 제안한다.

TAD 방식은 다음의 특성을 가진다[10].

- 변환 규칙(transformation rule)들은 정확히 정의된 구문적인 규칙을 가진다.
- 변환 규칙들은 아키텍트가 의도한 의미(Semantic)를 유지할 수도 있으며 문법적인 내용만을 가질 수 있다.
- TAD는 시스템의 기능성(functionality)들을 알게 해주기 때문에 아키텍처 디자인에 대해서 유용한 방법일 뿐만 아니라 어떤 품질을 변경하거나 새로운 기능을 추가하는 시스템의 진화에도 쉽게 수행 될 수 있으며 검증될 수 있다.
- 아키텍처 설계에 의도된 의미(Semantic)는 변환 규칙들이 연속성을 띈 형태인 변환(Transformation)으로 반영 또는 유지시킬 수 있다.
- TAD는 서비스 semantic perspective로부터 변환 명령어들의 의미들을 정의하고 시스템과 컴포넌트들에 대한 서비스를 수행하는 서비스 수행 알고리즘을 활용하여 아키텍트로 하여금 아키텍처 디자인이 그들의 의도에 맞는지 체크할 수 있도록 한다.

소프트웨어 진화를 정의할 수 있는 변환 패턴은 여러 변환 명령어들로 구성이 되어 있는데, 이 명령어들은 크게 아래와 같이 7가지로 분류된다 [10].

1) 생성과 제거(Creation and Destruction): 새로운, 또는 존재하는 컴포넌트, 커넥터, 또는 포트를 생성, 또는 제거한다.
2) 추가(Addition): 새로운 컴포넌트, 혹은 커넥터를 추가한다.
3) 삭제(Deletion): 존재하는 컴포넌트, 혹은 커넥터를 삭제한다.
4) 연결(Connecting): 컴포넌트의 포트와 커넥터의 포트를 연결한다.
5) 단절(Disconnecting): 컴포넌트의 포트와 커넥터의 포트를 끊는다.
6) 분해(Decomposition): 존재하는 컴포넌트를 두 개의 컴포넌트로 분해한다.
7) 결합(Merging): 존재하는 두 개의 컴포넌트를 하나의 컴포넌트로 결합한다.

각각의 분류에는 여러 상세한 명령어들이 속하고 있으며, [그림 2]은 그 중 '추가'에 해당하는 명령어의 한 예이다. 그 외의 상세한 명령어들은 [10]에 정의되어 있다.

add(Σ, Comp) :
대상이 되는 소프트웨어 아키텍처 전체(Σ)에 컴포넌트 Comp를 추가한다.

[그림 2] 컴포넌트를 추가하는 변환 명령어의 예

각 변환 명령어들은 필요한 입력 값들을 갖는다. 예를 들어, [그림 2]에 정의된 add 변환 명령어는 대상이 되는 소프트웨어 아키텍처 전체(Σ)와 추가하려는 컴포넌트 Comp를 입력 값으로 가진다. 이러한 입력 값들은 소프트웨어 아키텍처 표현 요소들 모두가 될 수 있으며, [10]에서는 아래와 같이 분류한다.

1) 시스템(System): 소프트웨어 시스템을 의미한다. 주로 Σ로 표현된다.
2) 컴포넌트: 컴포넌트는 시스템 형상(System configuration)의 가장 기본적인 단위이자, 서비스를 제공하는 기본 단위를 의미한다.
3) 커넥터: 컴포넌트들 간의 서비스를 전달해주는 아키텍처 구성 요소를 의미한다.
4) 컴포넌트와 커넥터의 연결: 시스템에서 컴포넌트는 서비스의 제공자와 사용자로 구분되며, 이들은 반드시 커넥터를 통해 연결되어야 한다.
5) 포트: 컴포넌트와 커넥터가 연결되는 지점을 의미한다.
6) 구조(Structure): 시스템 자신을 포함하여, 분해 가능한(non-atomic) 형상(Configuration)을 의미한다.

이러한 아키텍처 표현 요소들은 다양한 방법으로 표현될 수 있지만, TAD의 변환 패턴에서 정의하는 영역은 아니며 Architecture Description Language (ADL)에서 위의 요소들을 소프트웨어 아키텍처로써 표현하는 방법을 정의하고 있다 [11][16].

4. 아키텍처 기반의 소프트웨어 시스템 진화 프레임워크

4절에서는 아키텍처를 기반으로 한 소프트웨어 시스템 진화 프레임워크를 제시한다. 이를 위하여 4.1절에서는 본 프레임워크를 구성하는 소프트웨어 진화 요소들에 대하여 설명하며, 변환 패턴, 소프트웨어 아키텍처 표현 그리고 소프트웨어 진화 과정(Process)이 설명된다. 4.2절에서는 4.1절에서 설명된 소프트웨어 진화 요소들을 이용한 아키텍처 기반의 소프트웨어 시스템 진화 프레임워크가 제시된다.

4.1 프레임워크 구성요소

아키텍처 기반의 소프트웨어 시스템 진화는 진화의 대상이 되는 소프트웨어와 진화 후의 소프트웨어를 표현하는 아키텍처, 진화 과정을 표현하는 변환 패턴, 그리고 소프트웨어 아키텍처에

변환 패턴이 적용되는 소프트웨어 진화 과정으로 설명 될 수 있다. 그렇기 때문에, 변환 패턴, 소프트웨어 아키텍처, 그리고 소프트웨어 진화 과정을 소프트웨어 진화 프레임워크의 구성 요소들이라 할 수 있다. 이 절에서는 위의 각 구성 요소들을 설명한다.

4.1.1. 변환 패턴

변환 패턴은 대상이 되는 소프트웨어 아키텍처가 목표가 되는 소프트웨어 아키텍처로 진화하기 위한 변환 과정을 정형화된 변환 명령어를 사용하여 정의한 것이다. 변환 패턴은 한번 정의가 되면 Java 언어의 method처럼 재사용될 수 있다. 이 경우, 대상이 되는 소프트웨어 아키텍처가 변환 패턴의 입력 값이 되고, 목표가 되는 소프트웨어 아키텍처가 변환 패턴의 출력 값이 된다.

```
transformation_ROS (Σ,Comps,OldDB,UpdatedDB)
    createComp(ReplicaDB);
    ReplicaDB=copyComp(OldDB);
    add(Σ,ReplicaDB);
    for each component c of Comps
        connect(Σ,ReplicaDB.Read, c.Read);
        disconnect(Σ,OldDB.Read, c.Read);
        disconnect(Σ,OldDB.Write, c.Write);
    end for
    createComp(UpdatedDB);
    delete(Σ,OldDB);
    add(Σ,UpdatedDB);
    for each component c of Comps
        connect(Σ,UpdatedDB.Read, c.Read);
        connect(Σ,UpdatedDB.Write, c. Write);
        disconnect(Σ,ReplicaDB.Read, c.Read);
    end for
    if UpdatedDB is successfully committed
        then delete(Σ,ReplicaDB);
        else rollback(Σ,Comps,OldDB,UpdatedDB);
```

[그림 3] 'Transformation_ROS' 변환 패턴 정의

예를 들어, [그림 3]은 Transformation_ROS라는 변환 패턴을 정의하고 있다. 이 변환 패턴은 Σ (대상이 되는 소프트웨어 아키텍처 전체를 의미), Comps(컴포넌트 셋을 의미), OldDB(진화 대상이 되는 DB 컴포넌트), 그리고 UpdatedDB(진화의 목표가 되는 DB 컴포넌트)를 입력 값으로 가지며, 출력 값은 OldDB 컴포넌트가 UpdatedDB 컴포넌트로 진화된 소프트웨어 아키텍처가 된다. 이 변환 패턴에 정의되어 있는 변환 과정은 Read-Only Service (이하 ROS)에 해당하는 변환 과정으로써, DB 컴포넌트를 진화시키는 동안에 DB에 접근하는 컴포넌트들이 read 명령어만 할 수 있도록 임시로 변경하여 진화 목표를 달성하는 변환 과정이다.

4.1.2. 소프트웨어 진화 과정

소프트웨어 진화 과정은 변환 패턴과 소프트웨어 아키텍처를 이용하여 대상 소프트웨어 시스템을 진화시키기 위한 과정을 의미한다. 소프트웨어 진화 과정은 [그림 4]와 같은 순서를 가진다.

아래의 진화 과정을 통해, 아키텍처 수준에서 논의된 소프트웨어 진화 과정이 실제 물리적 구현 수준에 적용이 되게 된다.

입력 값:
- 소프트웨어 아키텍처: 논리적 수준에서 소프트웨어 시스템을 ADL 등의 언어를 사용하여 표현함.
- 아키텍처-구현 매핑 정보: 논리적 수준의 구성요소들과 실제 구현요소들 사이의 관계를 표현함.
- 변환 패턴: 논리적 수준에서 변환 과정을 정의함.

출력 값:
- 진화 후의 소프트웨어 아키텍처: 논리적 수준에서 변환 패턴으로 인해 진화한 소프트웨어 아키텍처를 표현함.
- 구체적 변환 패턴: 진화를 적용하는 소프트웨어 시스템이 실제로 해석할 수 있도록 구체적인 구현 정보를 담고 있는 변환 패턴을 의미함.

프로세스:
1) 논리적 수준의 변환 패턴의 모든 입력 값들과 출력 값들을 실제 구현요소들로 변환해주어 구체적 변환 패턴을 만든다.
2) 구체적 변환 패턴을 해석하여 실제 소프트웨어 시스템을 진화시킨다.
3) 논리적 수준의 변환 패턴을 소프트웨어 아키텍처에 적용하여 진화 후의 소프트웨어 아키텍처를 만든다.

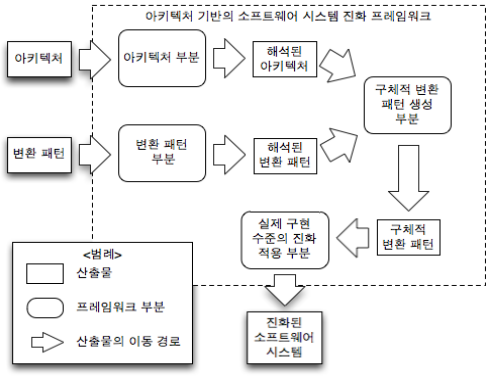
[그림 4] 소프트웨어 진화 과정

위의 진화 과정을 통해, 아키텍처 수준에서 논의된 소프트웨어 진화 과정이 실제 물리적 구현 수준에 적용이 되게 된다.

4.2 아키텍처 기반의 소프트웨어 시스템 진화 프레임워크

3.1절에서 다룬 소프트웨어 진화 요소들을 이용한 아키텍처 기반의 소프트웨어 시스템 진화 프레임워크는 크게 4부분으로 나뉠 수 있다.

- 아키텍처 부분: ADL, 혹은 그에 준하는 엄격하게 정의된 아키텍처 문서를 입력받아 내용을 해석(parse)하는 부분이다. 해석된 아키텍처를 출력하며, 해석된 아키텍처는 '구체적 변환 패턴 생성 부분'이 사용할 수 있도록 '아키텍처-구현 매핑 정보'를 반영한다.
- 변환 패턴 부분: 변환 패턴 표현 언어를 이용하여 정의된 변환 패턴을 입력받아 변환 패턴에 사용된 오퍼레이션들과 관련 입력 값들을 해석(parse)하는 부분이다. 해석된 변환 패턴을 출력한다.
- 구체적 변환 패턴 생성 부분: 아키텍처 부분에서 해석한 내용을 바탕으로 변환 패턴을 실제 구현 수준에서 적용될 수 있도록 구체적 변환 패턴으로 변경해주는 부분이다. 구체적 변환 패턴을 출력한다.
- 실제 구현 수준의 진화 적용 부분: 구체적 변환 패턴을 해석하여 실제 구현된 소프트웨어 시스템에 진화 과정을 적용하여 진화시키는 부분이다. [그림 5]는 이러한 프레임워크를 시각적으로 보여준다.



[그림 5] 아키텍처 기반의 소프트웨어 시스템 진화 프레임워크

[그림 5]에서 볼 수 있듯이, 본 프레임워크에서는 소프트웨어 아키텍처와 변환 패턴을 입력값으로 받는다. 아키텍처 부분과 변환 패턴 부분은 서로 연관 관계가 없기 때문에 평행하게 실행 될 수 있으며, 둘의 결과를 '구체적 변환 패턴 생성 부분'에서 이용하여 구체적 변환 패턴을 생성한다. 구체적 변환 패턴을 읽고 해석하는 '실제 구현 수준의 진화 적용 부분'은 실제 소프트웨어가 배치되어있는 환경(프로그래밍 언어, 운영 체제 등)에 따라 달라진다. 즉, '구체적 변환 패턴'에 정의되어 있는 오퍼레이션들을 해석하여 실제로 적용하기 위해서는 같은 오퍼레이션이라도 사용 프로그래밍 언어, 운영 체제, 구현 모습 등에 따라 달라질 수 있다.

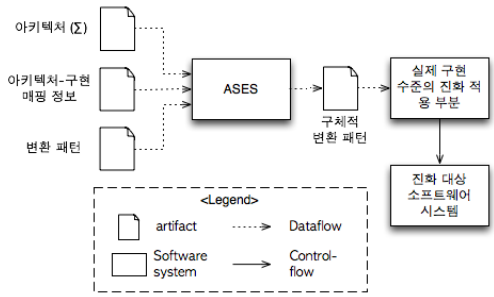
5. 아키텍처 기반의 소프트웨어 시스템 진화 프레임워크 적용 예시

5.1절에서는 4.2절에서 설명된 프레임워크를 구현한 Automatic Service Evolution System (ASES)을 배경도(context diagram), 개념도(conceptual diagram), 컴포넌트 앤 커넥터 다이어그램(Component and Connector diagram)을 통해 소개한다. 그리고 마지막으로 5.2절에서 ASES가 어떻게 적용되었는지를 설명한다.

5.1 프로토타입 구현

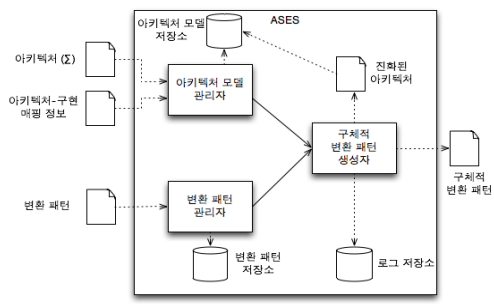
본 논문에서는 3.2절에서 설명한 아키텍처 기반의 소프트웨어 시스템 진화 프레임워크를 구현한 프로토타입을 제작하였다. 이 프로토타입은 Automatic Service Evolution System(이하 ASES)라고 부른다. ASES는 아키텍처 입력 값을 위해 카네기 멜론 대학교에서 개발된 Acme를 사용하였으며, AcmeStudio를 통해 작성된다. 또한, '아키텍처-구현 매핑 정보'를 독립적인 텍스트 기반의 파일에 저장을 하여 입력 값으로 받는다. 변환

패턴의 경우 동일한 방식으로 파일에 저장하여 입력 값으로 받는다. 출력 값은 '구체적 변환 패턴'으로 '실제 구현 수준의 진화 적용 부분'에서 해석할 수 있도록 자바 오브젝트 형태로 출력하게 된다. 마지막으로 '실제 구현 수준의 진화 적용 부분'은 자바 인터페이스 형태로 작성되어 각기 다른 구현 환경에 맞춰 개발 될 수 있도록 한다.



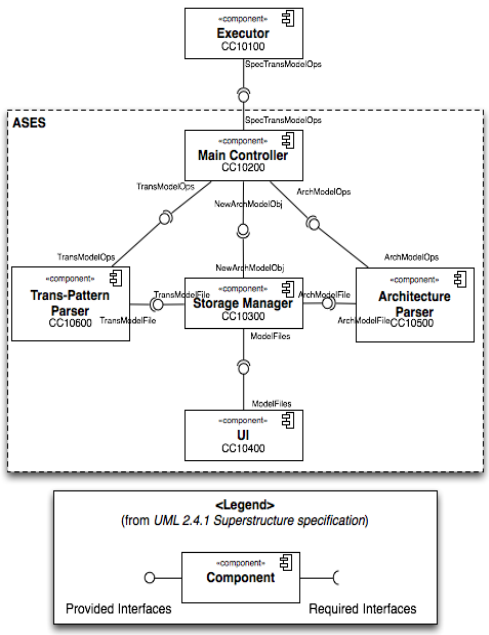
[그림 6] ASES의 배경도

[그림 6]은 ASES의 배경도를 나타낸다. 앞서 설명했듯이, 배경도에도 입력 값과 출력 값이 표현이 되어 있다. '실제 구현 수준의 진화 적용 부분'은 ASES 밖으로 빠져있으며 구현 환경에 따라 다른 모습으로 구현될 것이다. 배경도에서 볼 수 있듯이, ASES가 출력한 구체적 변환 패턴을 입력 받아 '실제 구현 수준의 진화 적용 부분'이 진화 대상 소프트웨어 시스템을 진화시킴을 알 수 있다.



[그림 7] ASES의 개념도

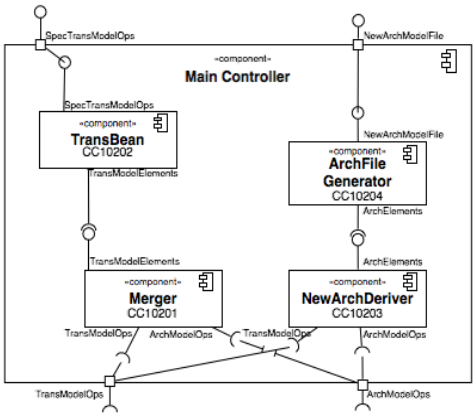
[그림 7]은 ASES의 개념도를 보여준다. 범례는 [그림 6]의 범례와 동일하다. 개념도에서 볼 수 있듯이, 아키텍처 모델 관리자는 입력 받은 아키텍처를 아키텍처 모델 저장소에 저장하고, 변환 패턴 관리자도 변환 패턴 저장소에 입력받은 변환 패턴을 저장한다. 이는 향후 재사용하거나 로그 기록을 남기기 위함이다. 구체적 변환 패턴 생성자는 해석된 아키텍처와 변환 패턴 내용을 바탕으로 구체적 변환 패턴을 생성하고, 관련 기록을 로그 저장소에 기록하며, 논리적 수준에서 진화된 아키텍처를 생성하여 아키텍처 저장소에 저장한다.



[그림 8] ASES의 컴포넌트 앤 커넥터 다이어그램

[그림 8]은 ASES의 구체적인 구현 이전에 설계한 컴포넌트 앤 커넥터 다이어그램이다. 각 컴포넌트들은 [그림 9]와 같이 상세하게 분해되어 기록되었다. [그림 7]의 개념도에서 나왔던 각종 저장소들은 Storage Manager를 통해 한 곳에서 관리되며, Architecture Parser와 Trans- Pattern Parser는 각각 아키텍처 모델 관리자, 변환 패턴 관리자의 서비스를 제공한다. Main Controller는 구체적 변환 패턴 생성자의 서비스를 제공하며, 기록을 남기고, 진화된 논리적 수준의 아키텍처를 생성

하여 Storage Manager에게 넘긴다. 출력된 구체적 변환 패턴은 Executor에게 전달되며, Executor는 이를 해석하여 실제 구현된 소프트웨어 시스템에 적용하는 역할을 한다.

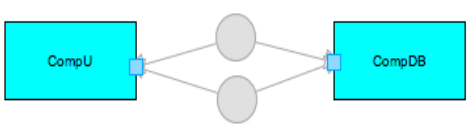


[그림 9] Main Controller (CC10200)이 분해된 모습

본 프로토타입에서는 Executor는 자바 인터페이스 형태로 작성되어 향후 서로 다른 환경에 맞춰 개발할 수 있도록 한다.

5.2 프로토타입 적용

본 프로토타입은 자바를 기반으로 작성되었으며, UI는 텍스트 기반의 콘솔 UI로 작성되었다. 본 프로토타입은 Acme로 작성된 간단한 아키텍처 파일과 '아키텍처-구현 매핑 정보' 파일, 그리고 변환 패턴 파일을 입력받아 구체적 변환 패턴을 텍스트 형태로 출력해주는 프로그램이다. 자바 인터페이스 형태로 구현된 Executor가 사용할 수 있도록 구체적 변환 패턴은 자바 오브젝트 형태로 저장된다.



[그림 10] 테스트에 사용된 Acme 기반의 아키텍처

[그림 10]의 아키텍처가 테스트에 사용이 되었다. 위의 아키텍처는 두 개의 컴포넌트와 2개의 커넥터로 구성되어 있다. 컴포넌트는 그림에서 보다시피, CompU와 CompDB이며, 커넥터는 각각 Read, Write이다.

변환 패턴은 [그림 3]의 transformation_ROS가 사용되었다.

```
*********ASES initial console-based UI**********
1. Work with a new system
2. Work with a recent system
0. exit
2
1.
2012.06.25
/Users/byron1st/Documents/workspace/nhn/data/System1.acme
/Users/byron1st/Documents/workspace/nhn/data/mapping_System1.dat
2.
2012.06.10
/Users/byron1st/Documents/ACME_workspace/Test_system/System1.acme
/Users/byron1st/Documents/ACME_workspace/Test_system/mapping_System1.dat
3.
2012.05.13
/Users/byron1st/Documents/ACME_workspace/Test_system/System_test.acme
/Users/byron1st/Documents/ACME_workspace/Test_system/mapping_System_test.dat
1
Type the path of a transformation pattern:
/Users/byron1st/Documents/workspace/nhn/data/testInput.dat
Is it correct? (/Users/byron1st/Documents/workspace/nhn/data/testInput.dat) (y/n)
y
```

[그림 11] ASES의 텍스트 기반 콘솔 UI

[그림 11]은 ASES의 텍스트 기반 콘솔 UI를 보여준다. Storage Manager를 통해 과거 사용했던 아키텍처, 변환 패턴을 불러올 수 있으며, 새로운 아키텍처, 변환 패턴으로 작업을 시작할 수 있다. [그림 11]은 [그림 2]에서 논리적 아키텍처 수준에서 표현되었던 transformation_ROS 변환 패턴이 실제 구현 수준에 적용되기 위해 구체적 변환 패턴으로 전환된 모습을 보여준다. 그림에서 볼 수 있듯이, 각 입력 값들은 모두 실제 구현 정보로 대체되었고, 구현 정보가 존재하지 않는 새로운 컴포넌트들은 <new>가 붙어 표현되었다. 이는 Executor에서 처리될 부분이다. 또한, for 구문으로 표현되어있던 부분은 아키텍처 모델을 해석한 내용을 바탕으로 판단한 횟수만큼 풀어서 표현되어 있다. 위 그림에서는 나타나있지 않지만, 'connect' 오퍼레이션은 모두 2번 반복된다. 왜냐하면, [그림 10]에서 CompU라고만 표현되어 있는 컴포넌트는 실제 구현상 2개의 컴포넌트로 구현되어 있으며, 이 점이 '아키텍처-구현 매핑 정보' 파일과 변환 패턴 파일에 담겨져 있기 때문이다. [그림 12]에서 나타난 모습은 일부이다.

```
transformation_ROS:
        /Users/byron1st/Documents/System1/
        <set>CompU
        /Users/byron1st/Documents/System1/CompDB/test1.java
        <new>UpdatedDB
<set>CompU:
        /Users/byron1st/Documents/System1/CompU/test1.java
        /Users/byron1st/Documents/System1/CompU/test2.java
createComp:
        <new>ReplicaDB
copyComp:
        /Users/byron1st/Documents/System1/CompDB/test1.java
        <new>ReplicaDB
add:
        /Users/byron1st/Documents/System1/
        <new>ReplicaDB
connect:
        /Users/byron1st/Documents/System1/
        /Users/byron1st/Documents/System1/CompU/test1.java&p&Read
        /Users/byron1st/Documents/System1/CompU/test1.java&p&Read
```

[그림 12] transformation_ROS의 구체적 변환 패턴 일부

6. 결 론

본 논문에서는 아키텍처 변화 패턴을 이용한 소프트웨어 시스템 진화 프레임워크를 제시하였다. 이를 위해, 아키텍처 기반의 진화 관련 연구인 Rainbow framework, Simplex architecture 등의 기존 연구들이 갖는 문제점들을 고찰하였다. 또한 아키텍처 변환 패턴 (Transformational Architecture Design)과 아키텍처 표현 언어인 ACME을 활용하여 아키텍처 기반의 소프트웨어 시스템 진화 프로세스를 제시하였다. 아키텍처 변화 패턴을 이용하여 직관적이고 도메인별로 다양하게 존재할 수 있는 소프트웨어 진화 패턴들을 지원할 수 있는 프레임워크를 제시함으로써 반복적인 소프트웨어 진화 작업의 자동화를 가능하게 하였다. 또한 ASES 라는 이름의 프로토타입을 설계하여 진화 과정을 실제로 적용한 예시로 구현하여 자동화가 가능한 진화의 가능성을 살펴보았다.

향후 연구로서, 소프트웨어 시스템 진화가 다양한 실제 구현 환경에 적용될 수 있도록 '실제 구현 수준의 진화 적용 부분'을 일반화하고 이를 실제 산업계에 적용할 것이다.

참 고 문 헌

[1] D. Garlan, S. Cheng, A. Huang, B. Schmerl P. Steenkiste. "Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure" In IEEE Computer, Vol. 37(10), October 2004.

[2] S. Cheng, D. Garlan and B. Schmerl. "RAIDE for Engineering Architecture-Based Self-Adaptive Systems," Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on, vol., no., pp.435,436, 16~24 May 2009

[3] I. Sommerville, "Software Engineering", 8th ed. Addison Wesley, 2006.

[4] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in International Conference on Software engineering, ICSE '98. Washington, DC, USA: IEEE, 1998, pp.177~186.

[5] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self- adaptive software," IEEE Intelligent Systems, vol. 14, pp. 54~62, May 1999.

[6] M. M. Lehman, L. A. Belady, "Program evolution: processes of software change," Academic Press Professional, Inc., San Diego, CA, 1985

[7] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice. 2nd edition, Addison-Wesley Professional, 2012.

[8] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor, "A comprehensive approach for the development of modular software architecture description languages," ACM Trans. Softw. Eng. Methodol., vol. 14, pp.199~245, April 2005.

[9] L. Sha, R. Rajkumar, M. Gagliardi, "A Software Architecture for Dependable and Evolvable Industrial Computing Systems," CMU/SEI Technical Report, Jul. 1995.

[10] S. Kang, “Transformational Architecture Design,” KAIST Technical Report (CS-TR- 2011-359), Apr. 2012.

[11] R. van Ommering., R. van der Linden, F., Kramer, J. Magee, J., “The Koala Component Model for Consumer Electronics Software,” IEEE Computer 33(3), 2000.

[12] S. Cheng, D. Garlan, “Stitch: A language for architecture-based self-adaptation,” Journal of Systems and Software, Vol. 85, Issue 12, pp 2860~2875, December 2012.

[13] J. Dowling and V. Cahill, “The k-component architecture meta-model for self-adaptive software,” in REFLECTION ’01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns. London, UK: Springer- Verlag, 2001, pp. 81~88.

[14] J. Zhang and B. H. C. Cheng, “Model- based development of dynamically adaptive software,” in ICSE ’06: Proceedings of the 28th international conference on Software engineering. New York, NY, USA: ACM, 2006,pp. 371~380.

[15] W. Cazzola, A. Ghoneim, and G. Saake, “Software evolution through dynamic adap- tation of its oo design,” in Objects, Agents and Features: Structuring Mechanisms for Contemporary Software, Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 69~84.

[16] D. Garlan, R. Monroe, and D. Wile, “Acme: An Architecture Description Interchange Language,” In Proceedings of CASCON’97, pp. 169~183, Nov. 1997.

[17] P. Kruchten. Architectural Blueprints-The “4+1” View Model of Software Architecture. IEEE Software 12, 42p~50p, 1995.

저자 소개

박 태 현

2010년 KAIST 전산학과 졸업(학사)
2012년 KAIST-CMU 소프트웨어공학 프로그램 졸업 (석사)
2012년~현재 KAIST 전산학과 박사과정
관심분야는 소프트웨어 공학, 소프트웨어 아키텍처, 소프트웨어 테스팅

안 휘

2010년 KAIST 전산학과 졸업(학사)
2012년 KAIST-CMU 소프트웨어공학 프로그램 졸업 (석사)
2012년~현재 KAIST 전산학과 박사과정
관심분야는 소프트웨어 아키텍처, 소프트웨어 제품라인 공학

황 상 철

2000년 경희대학교 산업공학과 졸업(석사)
2001년~2009년 삼성SDS 생산성혁신본부
2009년~2013년 NHN 기술혁신랩
2013년~현재 SK Planet Software Quality Engineering 팀

강 성 원

1982년 서울대학교 사회과학대학 졸업(학사)
1989년 University of Iowa 전산학 졸업(석사)
1992년 University of Iowa 전산학 졸업(박사)
1993년~2001년 한국통신(KT)선임연구원
1995년~1996년 미국 국립표준기술연구소(NIST) 객원 연구원
2001년~2005년 한국정보통신대학교 조교수
2003년~현재 Carnegie-Mellon University MSE프로그램 겸임교수
2005년~2009년 한국정보통신대학교 부교수
2009년~현재 KAIST 전산학과 부교수

박 종 빈

1994년 광운대학교 제어계측학과 졸업(학사)
2010년 University of Illinois at Urbana- Champaign 전산학 졸업(석사)
1995년~2008년 삼성 SDS
2008년~2012년 NHN 생산성혁신랩장
2012년~현재 SK Planet Software Quality Engineering 팀장