아키텍처 기반 소프트웨어 개발에서 소프트웨어 아키텍처 변형을 지원하기 위한 방법

(An Approach to Support Software Architecture Transformation in Architecture-Based Software Development)

최 희 석 † 염 근 혁 ††
(Heeseok Choi) (Keunhyuk Yeom)

요 약 소프트웨어 아키텍처는 복잡 다양한 소프트웨어 시스템을 개발하는 데 있어서 개발될 소프트웨어의 품질 달성에 중요한 영향을 미치는 핵심 설계로서 인식되고 있다. 따라서 아키텍처 기반의 소프트웨어 개발에서 고품질 소프트웨어 개발을 위하여 소프트웨어 아키텍처에 대한 변형이 필수적으로 요구된다. 그러나 아키텍처 변형 시 적용 가능한 설계 대안들의 다양성과 그것들이 아키텍처에 미치는 결과에 대한 예측의 어려움으로 인하여, 소프트웨어 아키텍처 변형을 적용하는 것이 쉽지 않다. 그러므로 다양한 설계 대안들이 아키텍처에 미치는 결과를 이해하고 분석하는 것을 통하여 소프트웨어 아키텍처 변형을 지원하기 위한 방법이 필요하다.

본 논문에서는 소프트웨어 아키텍처 변형을 체계적으로 지원하기 위한 방법을 제안한다. 제안하는 방법에서는 소프트웨어 아키텍처에 포함된 아키텍처 설계 결정들과 그것들에 대한 제약조건들을 바탕으로 결정 제약조건 그래프를 정의한다. 결정 제약조건 그래프를 이용하여 소프트웨어 아키텍처의 주요 설계 결정들 간의 의존 관계를 명시적으로 나타냄으로써, 소프트웨어 아키텍처 변형 과정에서 특정 설계 결정에 대한 설계 변형이 아키텍처에 미치는 영향을 체계적으로 분석 가능하게 한다. 본 논문에서 제시하는 소프트웨어 아키텍처 변형 방법은 아키텍처 변형에 대한 전반적인 이해를 용이하게 함과 동시에, 궁극적으로는 고품질 소프트웨어 개발을 위한 새로운 버전의 아키텍처 재생성을 돕는다.

키워드 : 소프트웨어 아키텍처, 소프트웨어 아키텍처 변형, 결정 제약조건 그래프

Abstract Software architecture is increasingly being viewed as a key design in developing complex software systems, which largely affects the achievement of quality attributes. During the architecture-based software development, therefore, architectural transformation is needed to achieve quality attributes. Due to the variety of design alternatives and the poor predictability of the effects of the transformation, however, it is not easy to apply architectural transformation. Therefore, the method is needed to support architectural transformation through systematically analyzing the effects of applying various design alternatives to the architecture.

This paper proposes an approach to support software architecture transformation. Based on architectural design decisions and the constraints on them included in the architecture, our approach defines a decision constraint graph representing the dependencies among architectural design decisions. Through using the decision constraint graph, architectural transformation can be systematically performed by understanding the effects of applying a transformation. While this work supports more understanding of applying architectural transformation, it also helps reconstruct a software architecture to improve the quality of the software.

Key words : Software architecture, Software architecture transformation, Decision constraint graph

· 본 연구는 한국과학재단 목적기초연구(R01-2003-000-10197-0)지원으로 수행되었음
† 비 회 원 : 한국전자통신연구원 임베디드S/W연구단 펀재형컴퓨팅미들웨어연구팀 연구원
choihs@etri.re.kr
†† 종신회원 : 부산대학교 컴퓨터공학과 교수
yeom@pusan.ac.kr
논문접수 : 2003년 9월 29일
심사완료 : 2004년 11월 10일

1. 서 론

오늘날 복잡 다양한 소프트웨어 시스템을 개발하는 데 있어서 상위 수준의 추상화를 나타내는 소프트웨어 아키텍처(Software Architecture)는 소프트웨어 개발에 참여하는 사람들간의 원활한 의사 소통과 시스템 설계

결정에 대한 합리적 판단을 가능하게 한다[1,2]. 더구나 대규모 소프트웨어 시스템에 요구되는 품질의 달성이 주로 그 시스템의 소프트웨어 아키텍처에 의해 결정된다[3]는 사실은 소프트웨어 아키텍처가 고품질 소프트웨어 개발의 중요한 척도가 됨을 의미한다. 따라서 소프트웨어를 개발하기 전에 소프트웨어 아키텍처를 설계하고 평가하는 활동과 더불어 소프트웨어 아키텍처 변형(Software Architecture Transformation)이 아키텍처 기반 소프트웨어 개발을 위한 활동으로서 중요하게 인식되고 있다. 소프트웨어 아키텍처 변형은 동일한 기능성을 가진 서로 다른 품질의 소프트웨어 아키텍처를 재정의하는 것을 가능하게 된다[4]. 즉, 소프트웨어 아키텍처를 토대로 개발될 소프트웨어의 품질 요구사항이 충족되지 못하거나 아키텍처를 향상시키고자 할 경우에 소프트웨어 아키텍처에 대한 변형이 요구된다. 그러나 소프트웨어 아키텍처 변형은 적용 가능한 설계 대안들의 다양성과 그것들이 아키텍처에 미치는 결과에 대한 예측의 어려움으로 인하여, 소프트웨어 아키텍처 변형을 적용하는 것이 쉽지 않다. 그러므로 다양한 설계 대안들이 아키텍처에 미치는 결과를 이해하고 분석하는 것을 통하여 소프트웨어 아키텍처 변형을 지원하기 위한 방법이 필요하다.

소프트웨어 아키텍처 변형에 관한 현재까지의 연구는 아키텍처 이해, 아키텍처 분석, 그리고 아키텍처 변경 등 서로 다른 목적을 가지고 이루어져 왔다[4-8]. 이들 연구들을 살펴보면 주로 역공학적 측면에서 소프트웨어를 이해하거나 유지보수하기 위한 활동으로서 소프트웨어 아키텍처의 표현과 변형을 다루고 있다. 그 밖의 연구에서는 소프트웨어 아키텍처 변형 시 적용 가능한 일반적인 규칙들을 정의하거나 다양한 아키텍처 설계 대안들이 소프트웨어 품질에 미치는 영향과 아키텍처 변형의 자동화를 위한 고려 사항들을 다루고 있다. 그러나 소프트웨어 아키텍처를 변형하는 과정에서 아키텍처에 대한 이해와 품질 특성과의 연관성에 대한 고려가 여전히 아키텍처 설계자의 경험과 직관에 의존하고 있다. 또한 소프트웨어 아키텍처를 변형하는 데 있어서 적용 가능한 설계 대안들의 다양성과 그것들이 아키텍처에 미치는 결과에 대한 분석 및 이해가 부족하다.

본 논문에서는 소프트웨어 아키텍처 변형을 체계적으로 지원하기 위한 방법을 제안한다. 소프트웨어 아키텍처는 아키텍처 설계 시 결정되고, 소프트웨어의 품질 달성에 중요한 영향을 미치는 설계 결정들을 포함하고 있다. 그러므로 제안하는 방법에서는 소프트웨어 아키텍처에 포함된 아키텍처 설계 결정들과 그것들에 대한 제약조건들을 바탕으로 결정 제약조건 그래프를 정의한다. 결정 제약조건 그래프를 이용하여 소프트웨어 아키텍처의 주요 설계 결정들 간의 의존 관계를 명시적으로 나타냄으로써, 소프트웨어 아키텍처 변형 과정에서 특정 설계 결정에 대한 변형이 아키텍처에 미치는 영향을 체계적으로 분석 가능하게 한다. 본 논문에서 제시하는 소프트웨어 아키텍처 변형 방법은 아키텍처 변형에 대한 전반적인 이해를 용이하게 함과 동시에, 궁극적으로는 고품질 소프트웨어 개발을 위한 새로운 버전의 아키텍처 재생성을 돕는다.

2. 관련 연구

본 논문은 소프트웨어 아키텍처에 대한 변형 방법을 제안한다. 이를 위하여 소프트웨어 아키텍처를 먼저 이해하고, 소프트웨어 아키텍처 변형을 위한 기존 연구들을 살펴본다.

2.1 소프트웨어 아키텍처

소프트웨어 아키텍처는 소프트웨어 개발에 참여하는 사람들간의 원활한 의사 소통과 시스템 설계 결정에 대한 합리적 판단을 가능하게 하는 상위 수준의 시스템 추상화이다[2]. 그리고 소프트웨어 아키텍처는 시스템 구조 및 그 시스템을 구성하는 컴포넌트들의 동작 등 개발될 소프트웨어의 품질 달성에 중요한 영향을 미치는 아키텍처 설계 결정들을 포함한다[2,9]. 아키텍처에 포함된 설계 결정들을 잘 표현하고 아키텍처에 대한 이해를 높이기 위한 대표적인 아키텍처 모델로서 Perry & Wolf의 모델[10], Shaw & Garlan의 모델[11], 그리고 Kruchten의 "4+1" 뷰 모델[9] 등이 소개되어 있다. 예를 들어, 본 연구에서 적용한 4+1 뷰 모델의 소프트웨어 아키텍쳐는 표 1에서 나타낸 바와 같이 추상화 요

표 1 4+1 뷰 모델의 아키텍처 정보

뷰	아키텍처 수준의 정보
Use case view	시스템의 주요 목적 주요 목적 수행과 관련된 일련의 행동 흐름
Logical view	설계에 대한 개념적 모델 -정적 구조와 동적인 상호작용
Development view	개발 환경에서의 소프트웨어 정적 구조 -구현 모듈과 그것들간의 상호관계 -컴포넌트의 그룹화 및 분리, 가시적 인터페이스 정의
Process view	동시성 및 동기화 -자원의 사용, 병렬 수행, 비동기적 이벤트의 처리 -작업 그룹과 복제단위로서의 프로세스 분할 작업 단위간의 상호작용 메커니즘 메시지 흐름 및 프로세스의 부하
Physical view	소프트웨어 구성 요소의 하드웨어로의 배치 관계 Logical, Process, Development 뷰에서 결정된 요소들의 처리 노드로의 매핑 관계

소, 논리적 구성, 컴포넌트의 동시성과 동기화, 컴포넌트 간 상호작용 및 연결 메커니즘, 병렬화, 물리적 배치, 아키텍처 패턴 및 아키텍처 스타일 등과 관련된 아키텍처 정보를 포함하게 된다. 그러므로 폭넓게 적용되고 있는 아키텍처 모델에서 정의한 아키텍처 정보들을 바탕으로 다양한 아키텍처 설계 결정들을 이해하는 것이 가능하다.

2.2 소프트웨어 아키텍처 변형

소프트웨어 아키텍처 변형에 관한 연구는 여러 문헌에서 발견된다. 현재까지의 연구는 아키텍처 이해, 아키텍처 분석, 그리고 아키텍처 변경 등 서로 다른 목적을 가지고 이루어져 왔다[4-8]. 이와 관련하여 소프트웨어 아키텍처 변형에 관한 대표적인 연구들을 살펴보면 다음과 같다.

Bosch[4]는 하나 이상의 소프트웨어 품질 특성을 향상시키기 위한 활동으로서 소프트웨어 아키텍처 변형을 이해하고 있다. 또한 소프트웨어 아키텍처 향상을 위한 일반적인 규칙을 정리하여 제시하였다. 즉, 설계 변경을 위하여 아키텍처 스타일을 부여하는 것, 설계 패턴을 적용하는 것, 비기능적 요구사항을 기능적 요구사항으로 변환하는 것, 요구사항을 여러 컴포넌트에 분산시키는 것 등의 규칙을 제시하였다. 그러나 아키텍처 스타일을 어떻게 적용해야 하는가, 비기능적 요구사항을 변환하는 과정에서 어떠한 일을 수행해야 하는가, 그리고 설계 변경이 미치는 영향에 대한 결정 등 자세한 사항에 대한 것들을 아키텍처 설계자의 몫으로 남겨두고 있다. 다음으로 Ambriola와 Kmiecik[5]는 소프트웨어 아키텍처 변형을 아키텍처 설계의 핵심 활동으로 이해하고, 아키텍처 변형을 아키텍처 스타일, 아키텍처 기술 언어(ADLs), 비기능적 요구사항과 밀접하게 연관시키고 있다. 또한 아키텍처 변형을 적용하는 데 있어서 어려운 점들과 자동화가 필요한 사항들을 제시하고 있다. 즉, 올바른 설계 변형을 위해 설계 구성 요소 및 구성 요소들 간의 제약조건 등에 대한 기억과 아키텍처 설계 변형을 적용했을 때 미치는 영향에 대한 이해 등을 도울 수 있는 자동화가 요구됨을 지적하고 있다. 그리고 Krikhaar[6]는 아키텍처 변형을 코드 수준에서 수행되는 오퍼레이션으로 정의하였다. 가령, CreateUnit, DeleteUnit, CombineUnit 등과 같이 단위 요소의 생성, 삭제, 결합 등을 주로 다루었다. 그러나 설계 변형에 대한 영향 분석은 소프트웨어 아키텍처 설계자의 경험에 의존하고 있다. 또한 Carriere, Woods, 그리고 Kazman [7]은 아키텍처 요소를 정적, 동적 피처(Features)로 기술하고, 이러한 피처의 변경으로 아키텍처 변형을 정의하고 있다. 그들은 시스템을 역공학 하기 위한 프로세스를 제시하였고, 아키텍처의 재생성과 변형 과정을 통하여 코드의 변형을 꾀하고 있다. 마지막으로 Fahmy와 Holt[8]는 아키텍처 이해를 위한 목적으로 아키텍처를 그래프로 표현, 아키텍처의 변형을 그래프의 재정의로 나타내고 있다.

살펴본 바와 같이 지금까지의 소프트웨어 아키텍처 변형에 관한 연구는 주로 역공학적 측면에서 소프트웨어를 이해하거나 유지보수하기 위한 활동으로서 소프트웨어 아키텍처의 표현과 변형을 다루고 있다. 이외에도 소프트웨어 아키텍처 변형 시 적용 가능한 일반적인 규칙들을 정의하거나 다양한 아키텍처 설계 대안들이 소프트웨어 품질에 미치는 영향과 아키텍처 변형의 자동화를 위한 고려 사항들을 제시하고 있다. 그러나 소프트웨어 아키텍처를 변형하는 과정에 있어서는 여전히 아키텍처에 대한 이해와 품질 특성과의 연관성에 대한 고려가 아키텍처 설계자의 경험과 직관에 의존하고 있다. 또한 소프트웨어 아키텍처를 변형하는 데 있어서 적용 가능한 설계 대안들의 다양성과 그것들이 아키텍처에 미치는 결과에 대한 분석 및 이해가 부족하다. 그러므로 다양한 설계 대안들이 아키텍처에 미치는 영향을 체계적으로 분석하는 것을 통하여 소프트웨어 아키텍처 변형을 지원하기 위한 방법이 필요하다.

3. 소프트웨어 아키텍처 변형 방법

소프트웨어 아키텍처 변형은 그림 1에서 나타낸 바와 같이 변형 요구영역 식별, 변형 전략 수립, 아키텍처 변형, 그리고 아키텍처 검증 등 네 가지 단계로 수행된다. 먼저 변형 요구영역 식별 단계에서는 소프트웨어 아키텍처에서 설계 변형이 요구되는 영역 즉, 개발될 소프트웨어의 품질 요구 충족을 저해하거나 아키텍처 불일치를 나타내는 영역 등을 결정한다. 이를 위하여 제시된 아키텍처에 대한 평가 뿐만 아니라, 그림 1에서 언급한 결정 제약조건 그래프(Decision Constraint Graph)를 통하여 설계 결정들간의 불일치를 분석하고 이해한다. 다음으로 변형 전략 수립 단계에서는 설계 변형이 요구되는 영역들에 대하여 적용 가능한 일련의 설계 변형 전략을 구하는 단계이다. 이를 위하여 변형 요구영역에 대한 설계 대안들의 다양성과 특정 영역의 설계 변형이 다른 설계 결정에 미치는 영향 등을 체계적으로 분석한다. 이 단계에서는 그림 1에서 나타낸 결정 제약조건 그래프를 이용하여 특정 영역의 설계 변형이 아키텍처에 미치는 영향을 체계적으로 분석한다. 이를 통하여 품질 향상을 기대할 수 있고, 동시에 아키텍처 불일치를 유발하지 않는 일련의 설계 대안들로 구성된 설계 변형 전략을 결정한다. 아키텍처 변형 단계에서는 결정된 설계 변형 전략에 따라 아키텍처를 단계적으로 변형한다. 마지막으로 아키텍처 검증 단계에서는 변형된 아키텍처가

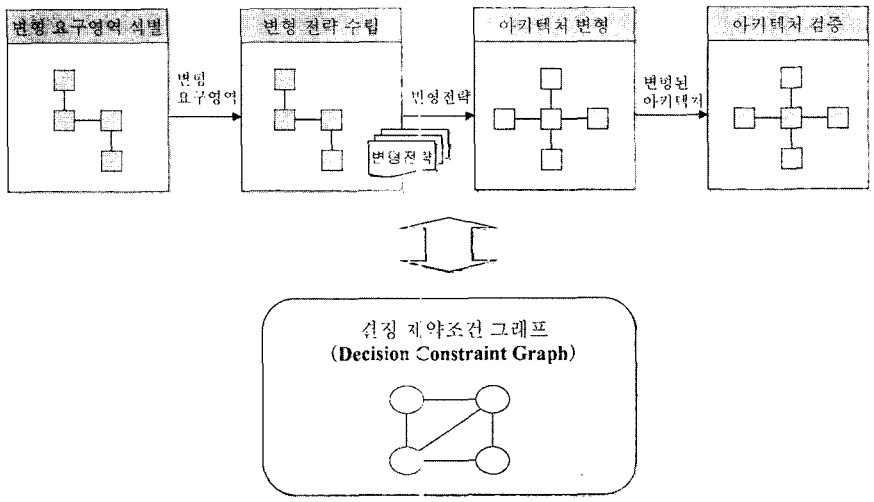

그림 1 제안한 소프트웨어 아키텍처 변형 방법

아키텍처 불일치를 나타내는지 여부와 목표했던 품질의 개선을 기대할 수 있는 있는지를 확인 또는 검증한다.

본 연구에서 제안하는 소프트웨어 아키텍처 변형 방법을 보다 구체적으로 설명하기 위하여, 제안한 아키텍처 변형 방법에 대하여 단계별 세부 활동들을 설명한다.

단계 1. 변형 요구영역 식별

변형 요구영역 식별 단계는 소프트웨어 아키텍처에서 설계 변형이 요구되는 영역 즉, 개발될 소프트웨어의 품질 요구 충족을 저해하거나 아키텍처 불일치를 나타내는 영역 등을 결정하는 단계이다. 소프트웨어 아키텍처는 소프트웨어 시스템에 요구되는 품질의 달성에 중요한 영향을 미치는 설계 결정들로 구성된다. 그리고 이러한 설계 결정들은 동일한 기능성에 대하여 서로 다른 정적, 동적 특성을 나타내고, 소프트웨어 시스템에 요구되는 품질 특성들과의 관련성에 있어서도 다양성을 드러낸다. 뿐만 아니라 소프트웨어 아키텍처에 포함된 아키텍처 설계 결정들은 그것들의 관련 품질 특성들 사이의 상충된 목적에 따라 소프트웨어의 품질 요구를 저해하거나, 혹은 아키텍처 설계 결정들 간의 호환성 문제로 인한 아키텍처 불일치를 나타내기도 한다. 그러므로 설계 변형이 요구되는 설계 영역을 결정하기 위해서는 제시된 대상 아키텍처에 포함된 설계 결정들과 관련 품질 특성들에 대한 이해가 필요하다. 또한 설계 결정들 간의 상호 제약조건 및 관련성에 대한 이해가 필요하다. 이 단계에서의 세부 활동은 다음과 같이 수행된다.

1) 아키텍처 설계 결정들을 구한다.

소프트웨어 아키텍처 설계 결정은 아키텍처 설계 중에 고려될 수 있는 중요한 설계 문제들에 대하여 적용 가능한 설계 대안들에 대한 결정을 의미한다. 이러한 아키텍처 설계 결정은 소프트웨어 시스템에 요구되는 품질의 달성에 중요한 영향을 미친다. 그러므로 아키텍처 설계 결정은 아키텍처를 설계할 때 뿐만 아니라, 시스템 개발 초기 단계에 개발될 소프트웨어의 품질을 조절하고, 설계 결정들간의 아키텍처 불일치 등을 해결하기 위한 설계 변형 시에도 충분히 고려되어야 한다. 아키텍처 설계 결정들을 구하는 데 있어서, 하나의 설계 결정은 특정 설계 문제 영역을 나타내는 설계 결정 변수와 적용 가능한 설계 대안을 나타내는 설계 결정 값으로 나누어 해석할 수 있다[12,13]. 이러한 구분을 통하여 아키텍처 설계 결정의 도출을 돕는다. 설계 결정 변수는 앞서 표 1에서 나타낸 바와 같이 특정 아키텍처 모델에서 표현 가능한 아키텍처 정보로부터 결정될 수 있다. 즉, 4+1 뷰 모델로 표현되는 소프트웨어 아키텍처는 아키텍처 설계 시 추상화 요소, 논리적 구성, 컴포넌트의 동시성과 동기화, 컴포넌트간 상호작용 및 연결 메커니즘, 병렬화, 물리적 배치, 아키텍처 패턴 및 아키텍처 스타일 등을 결정해야 되는 설계 문제들을 포함하게 된다. 이러한 설계 문제들에 대한 설계 결정 값 즉, 적용 가능한 설계 대안들은 대상 아키텍처 또는 후보 아키텍처, 그리고 아키텍처 설계자의 경험과 지식 등으로부터 추출될 수 있다[12,13].

2) 결정 제약조건 그래프를 생성한다.

본 연구에서는 소프트웨어 아키텍처 변형을 체계적으로 지원하기 위하여, 아키텍처에 포함된 아키텍처 설계 결정들과 그것들에 대한 제약조건들을 바탕으로 결정 제약조건 그래프를 생성한다. 아키텍처 설계 결정들은 하나의 설계 결정이 다른 설계 결정의 다양성을 제약하거나 설계 결정들 간의 아키텍처 불일치를 일으킬 수

있는 의존 관계를 가진다[2,4,14]. 가령, 시스템을 구성하는 두 컴포넌트에 대한 역할 결정으로부터 두 컴포넌트 사이에 아키텍처 불일치가 발생할 수 있다. 이것은 두 컴포넌트 상호간의 역할에 대한 제약조건에 의해 두 컴포넌트가 서로 밀접하게 관련될 수 있음을 나타낸다[2]. 이와 같이 설계 결정들 간의 의존 관계는 각 아키텍처 설계 영역 즉 설계 결정 변수에 대하여 적용 가능한 설계 대안들을 제약하는 제약조건들에 의해 결정될 수 있다. 이러한 제약조건은 좀더 구체적으로는 단일 설계 결정을 제약하는 단일 제약조건(Unary Constraints)과 설계 결정들 간의 상호 관계를 제약하는 상호 제약조건(Binary Constraints)으로 나누어진다[15]. 그러므로 결정 제약조건 그래프는 그림 2에서와 같은 방법으로 생성할 수 있다. 그림 2의 결정 제약조건 그래프에서 노드(Nodes)는 설계 결정이 이루어지는 설계 영역 즉 설계 결정 변수를 나타내고, 간선(Edges)은 설계 결정들 간의 상호 제약관계를 나타낸다. 그림 2에서처럼 소프트웨어 아키텍처는 추상화 요소, 논리적 구성, 컴포넌트의 동시성과 동기화, 컴포넌트간 상호작용 및 연결 메커니즘, 병렬화, 물리적 배치, 아키텍처 패턴 및 아키텍처 스타일 등 다양한 설계 문제들을 포함하고 있고, 이러한 설계 문제들이 결정 제약조건 그래프에서 각각 노드로 할당되어 표현된다. 또한 소프트웨어 아키텍처 상에서 서로 밀접하게 관련되어 있는 아키텍처 설계 문제들이 결정 제약조건 그래프에서 간선으로 표현된다. 이와 같은 방법으로 소프트웨어 아키텍처에 포함된 아키텍처 설계 결정 및 그것들 간의 의존 관계를 나타내는 결정 제약조건 그래프가 정의된다.

3) 변형 요구영역을 결정한다.

소프트웨어 아키텍처에 포함된 아키텍처 설계 결정들은 그것들의 관련 품질 특성들 사이의 상충된 목적에 의해 소프트웨어의 품질 요구를 저해하거나, 혹은 아키텍처 설계 결정들 간의 호환성 문제로 인한 아키텍처 불일치를 나타내기도 한다. 그러므로 설계 변형이 요구되는 설계 영역을 결정하는 데 있어서, 소프트웨어 아키텍처 평가는 아키텍처에 포함된 설계 결정들이 가지는 품질 특성에 대한 이해를 제공한다[12,13]. 그러므로 아키텍처 평가를 통하여 개발될 소프트웨어의 품질 특성을 저해하는 설계 영역을 결정할 수 있다. 또한 생성된 결정 제약조건 그래프에 대한 이해를 통하여 아키텍처 불일치 문제를 일으키는 설계 영역을 결정할 수 있다.

단계 2. 변형 전략 수립

변형 전략 수립 단계는 설계 변형의 대상이 되는 하나 또는 그 이상의 아키텍처 설계 영역에 대하여 일련의 변형 활동들로 정의되는 설계 변형 전략을 결정하는 단계이다. 아키텍처 설계 변형은 설계 변형의 대상이 되는 영역에 대하여 적용 가능한 설계 대안들을 구하여 적용함으로써 이루어진다. 그러나 아키텍처 설계 대안들은 그것이 아키텍처에 미치는 영향에 있어서 서로 다른 특성을 가진다. 다시 말해서, 설계 변형은 특정 영역 혹은 이웃 설계 영역, 심지어 아키텍처 전반에 영향을 미칠 수 있다. 그러므로 아키텍처 설계 변형은 하나 또는 그 이상의 설계 영역에 대한 일련의 변형 활동에 의해 이루어질 수 있다. 그러므로 설계 변형이 요구되는 영역에 대하여 가능한 설계 대안들을 합리적으로 판단하고, 특정 영역의 설계 변형이 다른 설계 영역에 미치는 영향을 체계적으로 분석하는 것이 필요하다. 이 단계에서의 세부 활동은 다음과 같이 수행된다.

1) 아키텍처 설계 대안들을 구한다.

아키텍처 설계 대안들은 아키텍처 설계 시에 접하게 되는 설계 문제들에 대한 설계 방안으로서 다양한 설계 대안들이 여러 문헌에서 발견된다[16-18]. 또한 적용 가능한 설계 대안들은 대상 아키텍처에 대한 후보 아키텍처 또는 아키텍처 설계자의 경험으로부터 추출될 수 있다.

2) 아키텍처 설계 대안들을 분석한다.

소프트웨어 아키텍처 변형은 일반적으로 하나 이상의 설계 영역에 대한 변형을 필요로 하고, 이를 통하여 동일한 기능성을 가진 서로 다른 품질의 소프트웨어 아키

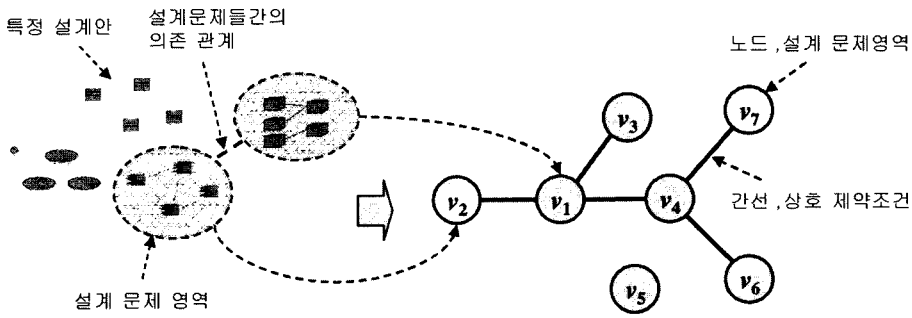

그림 2 결정 제약조건 그래프

텍처를 재정의하게 된다. 그러나 소프트웨어 아키텍처 변형 시 적용 가능한 설계 대안들의 다양성과 그것들이 아키텍처에 미치는 결과에 대한 예측의 어려움으로 인하여, 소프트웨어 아키텍처 변형을 적용하는 것이 쉽지 않다. 따라서 생성된 결정 제약조건 그래프를 이용하여 각 설계 영역에 대하여 적용 가능한 설계 대안들을 선택하는 데 있어서 개별 설계 영역에 대한 호환성(Consistency)을 유지할 수 있도록 설계 대안들을 결정한다. 뿐만 아니라 결정 제약조건 그래프를 이용하여 소프트웨어 아키텍처에 포함된 설계 결정들 간의 의존 관계를 체계적으로 다룸으로써, 설계 결정들 상호간의 호환성 유지를 바탕으로 특정 영역의 설계 변형이 아키텍처에 미치는 영향을 가시적으로 분석한다. 아키텍처 설계 대안들을 분석하고 결정하는 데 있어서, 결정 제약조건 그래프는 다음 두 가지 특성에 의해 분석된다.

① 노드 호환성

단일 설계 결정을 제약하는 단일 제약조건을 바탕으로 개별 설계 영역에 대한 설계 호환성을 의미한다. 따라서 특정 설계 영역에 대해 적용 가능한 설계 대안들은 최소한 그 설계 영역의 노드 호환성을 충족시켜 줄 수 있어야 한다.

② 간선 호환성

설계 결정들 간의 상호 관계에 제약하는 상호 제약조건을 바탕으로 설계 영역들 상호 간의 설계 호환성을 의미한다. 따라서 특정 설계 영역에 대해 적용 가능한 설계 대안들은 그 설계 영역의 이웃 설계 영역들에 대해서 간선 호환성을 충족시켜 줄 수 있어야 한다.

3) 설계 변형 전략을 결정한다.

아키텍처 설계 변형 전략은 설계 변형의 대상이 되는 하나 또는 그 이상의 아키텍처 설계 영역에 대한 설계 대안들의 집합으로서 정의된다. 따라서 설계 변형 요구영역들에 대하여 획득된 설계 대안들 중에서, 결정 제약조건 그래프의 노드 호환성과 간선 호환성을 모두 만족하는 일련의 설계 대안들의 집합으로서 설계 변형 전략을 결정한다.

단계 3. 아키텍처 변형

아키텍처 변형 단계에서는 아키텍처 설계 변형을 위해 필요한 일련의 변형 활동을 포함하는 아키텍처 설계 변형 전략을 소프트웨어 아키텍처에 실질적으로 적용하는 단계이다. 아키텍처 설계 변형 전략은 이미 하나 또는 그 이상의 아키텍처 설계 영역에 대한 설계 대안들의 집합으로서 정의되었다. 따라서 변형이 요구되는 설계 영역들에 대해 주어진 설계 대안들을 단계적으로 적용한다. 이 단계의 결과로서 동일한 기능성을 가진 서로 다른 품질의 소프트웨어 아키텍처가 재정의된다.

단계 4. 아키텍처 검증

아키텍처 검증 단계에서는 식별된 설계변형 요구영역을 변형 전략에 따라 아키텍처를 변경한 후, 변형된 아키텍처가 설계 결정들 간의 결정 제약조건들을 모두 만족하는지 혹은 목표했던 품질의 개선을 달성했는지를 확인 또는 검증하는 단계이다. 이를 위해서는 이미 제시한 바와 같이 소프트웨어 아키텍처에 포함된 설계 결정들을 중심으로 아키텍처 불일치를 평가하여야 한다. 또한 아키텍처 설계 결정들을 중심으로 품질 요구사항과 소프트웨어 아키텍처 간의 연관성을 분석, 평가함으로써 품질 만족도를 평가한다. 이 단계에서의 세부 활동은 다음과 같이 수행된다.

1) 결정 제약조건을 재평가한다.

아키텍처 기반 소프트웨어 개발에 이용될 소프트웨어 아키텍처는 어떠한 아키텍처 불일치를 포함하지 않아야 한다. 그러므로 변형된 아키텍처에 대하여 그것의 아키텍처 설계 결정들을 바탕으로 재정의될 수 있는 결정 제약조건 그래프를 이용하여, 변형된 아키텍처가 설계 결정들 간의 결정 제약조건들을 모두 만족하는지를 확인한다.

2) 품질 만족도를 재평가한다.

아키텍처 기반 소프트웨어 개발에 있어서 소프트웨어 아키텍처는 개발될 소프트웨어의 품질 달성에 중요한 영향을 미친다. 그러므로 고품질 소프트웨어 개발을 위해서, 변형된 소프트웨어 아키텍처를 그것의 품질 요구사항에 대하여 평가한다. 소프트웨어 아키텍처 평가는 [12,13]에서 제시한 아키텍처 평가 방법을 따른다.

지금까지 살펴본 바와 같이 본 연구는 소프트웨어 아키텍처에 포함된 아키텍처 설계 결정들과 그것들에 대한 제약조건들을 바탕으로 결정 제약조건 그래프를 정의하고, 이를 바탕으로 소프트웨어 아키텍처 변형을 위한 방법을 제시하였다. 결정 제약조건 그래프는 아키텍처에 포함된 설계 결정들과 그것들 간의 제약조건들을 명시적으로 표현하여 다룸으로써, 설계 결정들 간의 의존 관계를 쉽게 이해할 수 있도록 한다. 또한 설계 결정에 대한 호환성 유지를 바탕으로 특정 설계 결정 변형이 이웃 설계 결정 및 아키텍처 전반에 미치는 영향을 파악할 수 있도록 한다. 이는 소프트웨어 아키텍처 변형에 있어서 아키텍처에 대한 이해와 품질 특성과의 연관성에 대한 고려가 아키텍처 설계자의 경험과 직관에 의존했던 점을 보완해준다. 또한 소프트웨어 아키텍처를 변형하는 데 있어서 적용 가능한 설계 대안들의 다양성과 그것들이 아키텍처에 미치는 결과에 대한 분석 및 이해를 가능하게 한다. 그러므로 본 논문에서 제시하는 소프트웨어 아키텍처 변형 방법은 아키텍처 변형에 대한 전반적인 이해를 용이하게 함과 동시에, 궁극적으로는 고품질 소프트웨어 개발을 위한 새로운 버전의 아키

텍처 재생성을 돕는다.

4. 사례 연구

본 연구에서 제안한 소프트웨어 아키텍처 변형 방법의 적용성을 확인하고 제안한 방법을 쉽게 이해하기 위하여, 실시간 운영 시스템의 일종인 House Alarm System[4,19]에 적용하였다. House Alarm System은 많은 센서(Sensors)와 경보장치(Alarms)로 구성되는데, 센서는 특정 지역에서의 움직임을 감지하여 중앙 시스템에 전달하고, 경보장치는 침입자를 쫓기 위하여 소리 또는 빛을 발생시키는 역할을 한다. 이러한 House Alarm System은 실시간 시스템에서 중요하게 다루어지는 동시성 및 동기화, 상호작용 메커니즘 등과 관련된 요구사항들을 포함하고 있으며, 관련 품질 특성으로 결함 방지(Fault tolerance), 성능(Performance), 분산성(Distribution) 등을 포함한다. 그러므로 House Alarm System의 소프트웨어 아키텍처는 추상화 요소, 논리적 구성, 컴포넌트의 동시성과 동기화, 컴포넌트간 상호작용 및 연결 메커니즘, 병렬화, 물리적 배치, 아키텍처 패턴 및 아키텍처 스타일 등 다양한 아키텍처 설계 결정들을 포함하고 있다.

4.1 House Alarm System의 소프트웨어 아키텍처 및 설계 결정

House Alarm System의 소프트웨어 아키텍처를 나타내는 데 있어서 본 논문은 제안한 방법을 이해하는 데 도움이 되는 아키텍처 요소만을 4+1 뷰 모델을 이용하여 기술하였다. 먼저 그림 3에서는 House Alarm System의 주요 목적을 Use case view를 이용하여 나타내었다. House Alarm System의 주요 목적에 해당하는 기능으로는 *'Detect Movements'*와 *'Generate Alarms'*가 있다. 그리고 외부 시스템으로서 다양한 종류의 *'Sensors'*와 *'Alarms'*가 연결되어 있다.

다음으로 그림 4와 그림 5는 House Alarm System의 논리적 구성 요소 및 그것들 간의 상호 연결 관계와

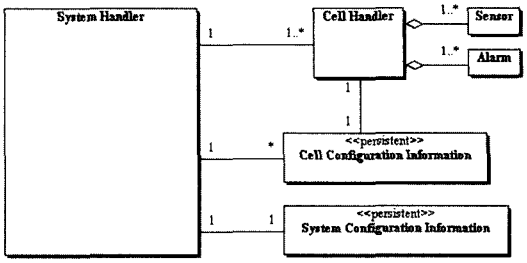

그림 4 Logical view: 논리적 구성 요소 및 연결 관계

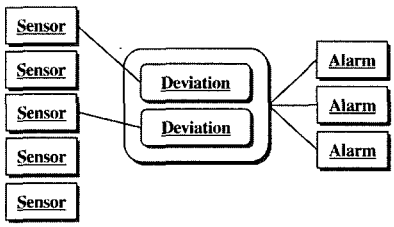

그림 5 Logical view: 논리적 구조

논리적 구조 등을 시스템에 대한 논리적 관점(Logical view)에서 나타낸 것이다. 그림 4에서 보는 바와 같이 시스템의 주요 구성 요소로는 입출력 요소에 해당하는 'Sensor'와 'Alarm', 중앙 처리 요소에 해당하는 'System Handler'와 'Cell Handler'가 있다. 그리고 입출력 요소와 중앙 처리 요소는 'Supervisor and subordinators' 관계[19]로써 연결되어 있다. 그리고 그림 5는 이들 요소로 구성되는 시스템의 논리적 구조가 'Shared memory' 스타일[2]의 구조를 가짐을 나타낸다.

이어서 그림 6, 7은 House Alarm System의 동시성과 동기화와 관련된 동적 상호작용을 나타내기 위하여 4+1 뷰 모델의 Process view를 이용하여 나타낸 것이다. 먼저 그림 6은 독립적으로 수행 가능한 작업 단위와 작업간 상호작용을 나타낸 것이다. 그림 6에서 보는 바와 같이, *'Cell Handler'*와의 상호 작용은 비동기적으로 수행되며 *'Supervisor'*와의 상호 작용은 동기적으로 수

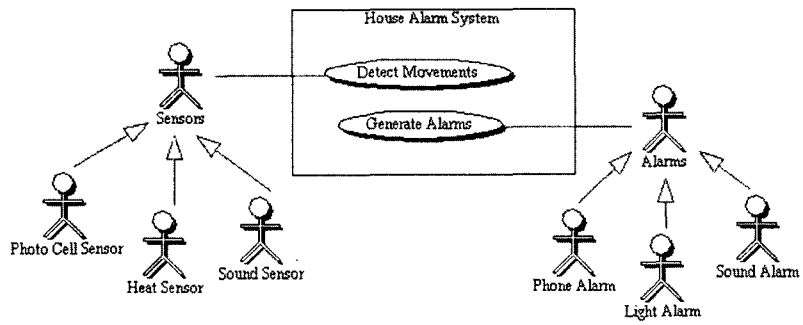

그림 3 Use case view

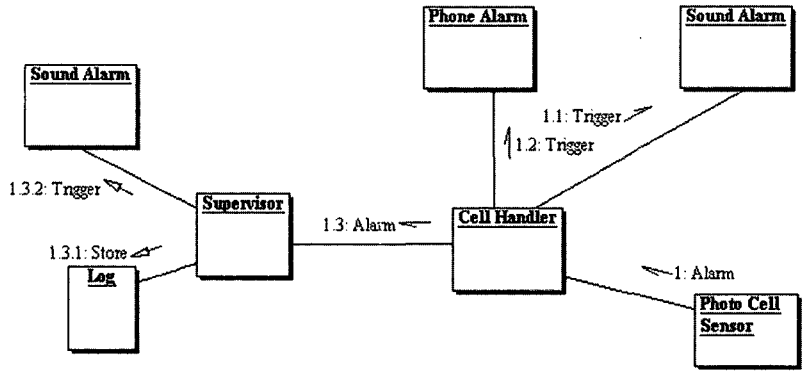

그림 6 Process view: 작업 분할 및 작업간 상호작용

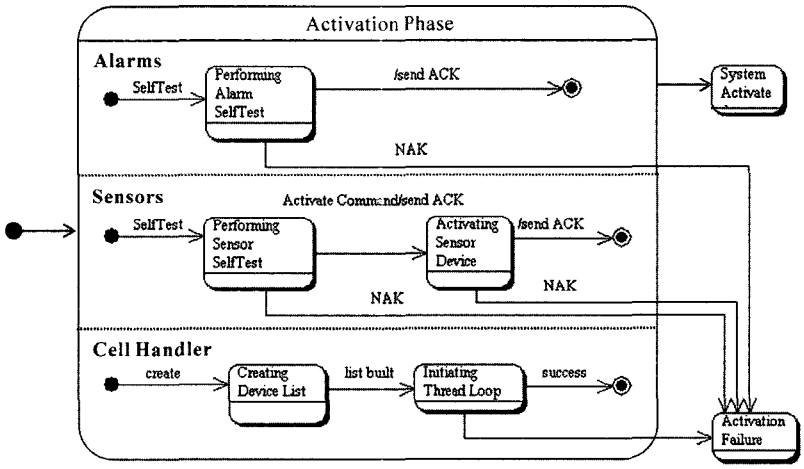

그림 7 Process view: 시스템 활성화

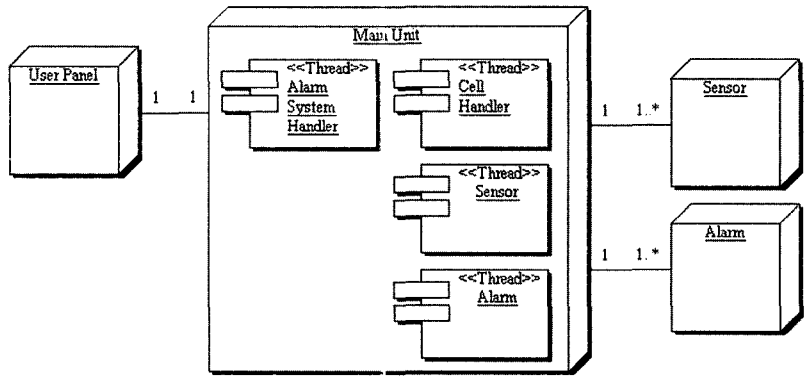

그림 8 Physical view

행된다. 그리고 그림 7은 시스템의 활성화 과정에서 동시 발생의 상태들과 그것들을 이용한 활성화를 나타낸다.

그림 8은 House Alarm System의 주요 컴포넌트들이 물리적 요소에 어떻게 배치되었는가를 나타내기 위하여 물리적 관점에서 시스템을 나타낸 것이다. 물리적 구성 요소는 크게 *'User Panel'*, *'Main Unit'*, *'Sensor'*, 그리고 *'Alarm'*으로 구성되어 있으며, Process view에서 식별된 작업(Tasks)들은 모두 *'Main Unit'*에

배치되었다.

이와 같이 소프트웨어 아키텍처는 표 1에서 정의된 아키텍처 모델에 따라서 소프트웨어 개발 시 중요하게 여겨지는 아키텍처 정보들을 포함하여 나타낸다. 본 예제의 그림 3에서 그림 8까지의 아키텍처 표현 등을 기본으로 하여 추출될 수 있는 아키텍처 설계 결정들을 표 2에서 요약 정리하였다. 표 2에서 나타낸 바와 같이, House Alarm System의 소프트웨어 아키텍처에 포함된 설계 결정들을 설계 결정 변수, 설계 결정 값 즉 현재의 설계안, 각 설계 결정에 대하여 가능한 설계 대안들, 그리고 그러한 설계 결정들이 추출될 수 있는 관련 뷰로 정리하였다. 예를 들면, 결정 변수 'Abstraction'과 'Interconnection'은 그림 4의 아키텍처 표현으로부터 추출되었고, 'Organization'은 그림 5의 아키텍처 표현으로부터 결정되었다.

4.2 결정 제약조건 그래프 정의

결정 제약조건 그래프는 소프트웨어 아키텍처에 포함된 아키텍처 설계 결정들과 그것들에 대한 제약조건들

표 2 아키텍처 설계 결정

결정 변수	결정 값 (설계 방안)	설계 대안들	관련 뷰 (UML 다이어그램)
Abstraction (논리적 구성요소)	Input/output units and central units	-Periodic units and a scheduler	Logical view (Class diagram)
Organization (논리적 구조)	Shared memory	-Blackboard with a scheduler	Logical view (Class diagram)
Interconnection (연결 메커니즘)	Supervisor and subordinators	-Subject and observers	Logical view (Collaboration diagram)
Tasks Partition (작업 분할)	Unit of active classes identified in the logical view	-Periodic components and a scheduler	Process view (Collaboration diagram)
Parallel Execution (병렬 실행)	Execution of sensors and alarms	Nothing	Process view (Activity diagram)
Events Handling (이벤트 처리)	Supervising/ broadcasting	-Operation calls -Mailboxes/message queues -Shared memory -Rendezvous	Process view (Collaboration diagram)
System Activation (시스템 활성화)	Concurrent execution of Concurrent substates	Nothing	Process view (Activity diagram)
Synchronization (동기화 메커니즘)	Monitoring (preemptive)	-Periodic execution (time-slicing)	Process view (Sequence diagram)
Tasks-Interaction (작업간 상호작용)	Event-based communication	-Call-and-return	Process view (Sequence diagram)
Flow-Control (흐름 제어)	Synchronous run of parallel activities	Nothing	Process view (Activity diagram)
Tasks-Deployment (물리적 작업 배치)	Main unit, user panel, and sensor/alarm devices	Nothing	Physical view (Deployment diagram)

표 3 단일 제약조건

결정 변수	단일 제약조건 (Unary constraints)
Abstraction	-Sensors and alarms should be identified as active classes
Organization	-Each alarm depends on a collection of sensors
Interconnection	-Sensors and alarms are connected to a Cell Handler, an active class that handles a specific cell -The Cell Handler is connected to the System Handler, which is an active class that handles the user communication
Tasks Partition	-Each task has separate threads of control that can be individually scheduled on separate processing nodes
Parallel-Execution	-Reading of inputs and potentially generating corresponding outputs take place concurrently
Events-Handling	-Events occurring in system should be processed synchronously and serially
System-Activation	-The activation of the entire system is performed by the successful activation of the tasks
Synchronization	-Concurrent tasks should be synchronized when accessing shared data
Tasks-Interaction	-Communication between the tasks should be possible
Flow Control	-The synchronization of parallel activities should be achieved
Tasks-Deployment	-The components should all been allocated to the main unit

표 4 상호 제약조건

결정 변수		상호 제약조건 (Binary constraints)
결정 변수	결정 변수	
Abstraction	Organization	The system is logically organized with a set of abstractions
Abstraction	Interconnection	A set of abstractions are the elements interconnected
Abstraction	Tasks Partition	The independent tasks are equal to the active classes identified in the logical view
Abstraction	System Activation	Concurrent substates in system activation are determined according to a set of abstractions
Tasks Partition	Parallel Execution	Concurrent tasks run in parallel
Tasks Partition	Synchronization	Tasks are synchronized in concurrently accessing shared data
Tasks Partition	Tasks Interaction	The independent tasks communicate with each other through specific interaction mechanism
Tasks Partition	Tasks Deployment	The independent tasks are physically deployed and distributed among the actual computers
Tasks Interaction	Events Handling	The interaction between tasks is performed through handling the events
Parallel Execution	Synchronization	The synchronization between concurrent tasks should be achieved when accessing shared data
Parallel Execution	Flow Control	Parallel activities should be handled along with the synchronized of those activities

로 구성된다. 본 연구의 방법을 이해하기 위하여 House Alarm System 예제에서 소프트웨어 아키텍처에 포함된 아키텍처 설계 결정들을 표 2에서 요약 정리하였다. 그리고 표 2에서 정리된 아키텍처 설계 결정들에 대한 단일 제약조건과 상호 제약조건들에 대해서는 표 3과 표 4에서 각각 정리하여 나타내었다.

그러므로 House Alarm System의 소프트웨어 아키텍처에 대한 예제에서 결정 제약조건 그래프는 그림 3에서 그림 8까지의 UML 다이어그램으로 표현된 아키텍처 설계 정보를 요약 정리한 표 2, 3, 4의 내용을 바탕으로 그림 9와 같이 나타낼 수 있다. 즉, 11 가지 종류의 설계 결정들이 그것들의 설계 결정 변수에 의해 각각 그래프의 노드로 표현되었고, 설계 결정들 간의 상호 제약조건에 의한 의존 관계가 그래프의 간선으로 표현되었다. 가령, 설계 결정 변수 'Abstraction'은 다른 설계 결정 변수들인 'Organization', 'Interconnection', 'Tasks-Partition', 그리고 'System-Activation' 등과 표 4에서 정리된 상호 제약조건에 의해 의존 관계를 가지게 된다.

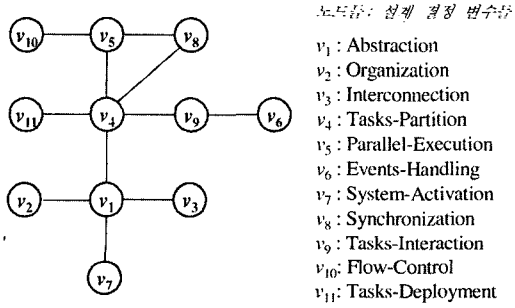

그림 9 House Alarm System 아키텍처에 대한 결정 제약조건 그래프

4.3 결정 제약조건 그래프를 이용한 아키텍처 설계 변형

제시된 대상 아키텍처에서 설계 변형이 가능한 영역으로 설계 영역 *v8* 즉, 공유 자원에 대하여 동시 수행 가능한 요소들 간의 동기화 메커니즘에 대한 설계 결정을 변형하는 경우를 예로 든다. 설계 영역 *v8*에 대한 설계 결정은 공유 자원에 대한 동시 접근을 제어하고 동기화하는 데 있어서 모니터링 방법을 적용하고 있지만, 레이스 컨디션(Race Condition) 등을 유발할 수 있어 에러 발생 가능성을 내재하고 있다[4]. 이에 대해 다른 설계 대안으로서 스케줄러에 의한 주기적 실행 방법을 적용하는 방안을 생각해볼 수 있다[4]. 그러나 설계 변형을 적용하기에 앞서, 해당 설계 영역의 설계 변형이 아키텍처에 미치는 결과에 대한 이해가 필요하다. 이를 위하여 그림 9에서 나타낸 House Alarm System 아키텍처에 대한 결정 제약조건 그래프를 바탕으로 설계 영역 *v8*에 대한 설계 변형이 이웃해 있는 설계 결정 뿐만 아니라 아키텍처 전반에 미치는 영향을 분석하였다. 그림 10은 결정 제약조건 그래프에 의한 변형 영향 분석 결과를 나타낸 것이다. 그림 10에서 보는 바와 같이 설계 영역 *v8*의 설계 변형에 대하여 ①에서 ⑨로 표시된 방향으로 이웃해 있는 설계 영역들에 대한 영향을 노드 호환성과 간선 호환성을 고려하여 분석하였다. 결과적으로 그림 10에서 설계 영역 *v8*의 설계 변형은 설계 영역 *v5*에 대해서는 표 3과 표 4에서 나타낸 바와 같이 *v5*에 대한 단일 제약조건(입출력 작업의 병렬 수행)과 상호

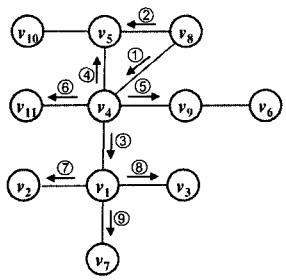

그림 10 결정 제약조건 그래프에 의한 변형 영향 분석

제약조건(공유 자원 동시 접근에 대한 동기화)을 위배하지 않으므로 *v5* 노드에는 영향을 주지 않는다. 그러나 그림 10에서 회색의 노드로 표시한 설계 영역들 *v1, v2, v4*의 경우, *v4*의 단일 제약조건(독립적으로 수행 가능한 작업), *v4*와 *v1*간 상호 제약조건(추상화 요소와 독립적으로 수행 가능한 작업간의 관계), 그리고 *v1*과 *v2*간 상호 제약조건(추상화 요소와 논리적 구조간의 관계) 등을 위배하므로 추가적인 설계 변형이 필요하였다. 이와 같은 방법으로 특정 설계 영역의 설계 변형이 설계 대안에 따라 아키텍처에 어떠한 영향을 미치는가를 이해할 수 있다.

이와 같이 결정된 설계 변형 영역들 *v1, v2, v4, v8*로 구성된 아키텍처 설계 변형 전략에 따라 아키텍처 설계 변형을 실질적으로 수행한다. 다음 그림 11은 설계 영역 v8에 대한 설계 변형에 의해 그림 5에서 나타낸 *v2*의 설계가 어떠한 방법으로 설계 변형이 이루어졌는가를 나타낸 것이다. 즉, 동기화 메커니즘에 대한 설계 변형이 House Alarm System에 대한 논리적 설계 구조에서 새로운 설계 요소 *'Scheduler'*가 포함된 주기적 실행 구조로 변형될 수 있음을 나타낸 것이다. 따라서 공유 자원에 대한 동기화 부분에 있어서의 에러 가능성을 줄일 수 있지만, 한편으로는 논리적 구조에 대한 설계 변형에 의해 시스템 전체의 성능 측면에서는 악영향을 줄 수 있음을 판단할 수 있다[4].

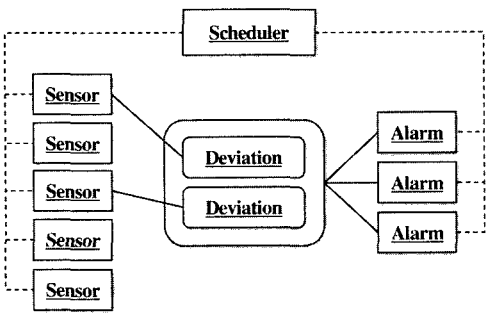

그림 11 논리적 구조에 대한 설계 대안

5. 결론 및 향후 연구 과제

소프트웨어 아키텍처는 복잡 다양한 소프트웨어 시스템을 개발하는 데 있어서 개발될 소프트웨어의 품질 달성에 중요한 영향을 미치는 핵심 설계로서 인식되고 있다. 따라서 소프트웨어 아키텍처를 토대로 개발될 소프트웨어의 품질 요구사항이 충족되지 못하거나 아키텍처를 향상시키고자 할 경우에 소프트웨어 아키텍처에 대한 변형이 필수적으로 요구된다. 그러나 소프트웨어 아키텍처 변형은 적용 가능한 설계 대안들의 다양성과 그것들이 아키텍처에 미치는 결과에 대한 예측의 어려움으로 인하여, 소프트웨어 아키텍처 변형을 적용하는 것이 쉽지 않다. 본 논문에서는 소프트웨어 아키텍처 변형을 체계적으로 지원하기 위해서 우선 소프트웨어 아키텍처 변형을 위한 전체적인 프로세스를 정의하였다. 또한 소프트웨어 아키텍처에 포함된 아키텍처 설계 결정들과 그것들에 대한 제약조건들을 바탕으로 결정 제약조건 그래프를 정의하였다. 결정 제약조건 그래프를 이용하여 소프트웨어 아키텍처의 주요 설계 결정들 간의 의존 관계를 명시적으로 나타냄으로써, 소프트웨어 아키텍처 변형 과정에서 특정 설계 결정에 대한 변형이 아키텍처에 미치는 영향을 체계적으로 분석 가능하게 하였다. 그러므로 소프트웨어 아키텍처 변형을 적용하는 데 있어서 소프트웨어 아키텍처에 대한 이해를 높였고, 아키텍처 변형에 대한 가이드를 제공하였다. 그리고 제안한 방법을 실시간 운영 시스템에 속하는 House Alarm System의 소프트웨어 아키텍처를 변형하는 데 적용해 봄으로써, 제안한 방법이 아키텍처 변형을 효과적으로 지원함을 확인하였다.

향후에는 다양한 도메인의 소프트웨어 아키텍처에 대한 사례 연구를 통하여 제안한 방법의 정당성과 효과를 분석, 평가한다. 또한 설계 결정변수들에 대한 결정값, 단일 제약조건, 상호 제약조건들에 대한 표현을 보다 엄밀하게 정의하고 이를 토대로 변형을 위한 설계 대안들을 선택할 수 있는 기준 제시가 필요하다. 마지막으로 아키텍처 기반 소프트웨어 개발을 지원하기 위하여, 아키텍처 평가 및 변형을 체계적으로 통합하는 프레임웍크를 개발한다.

참 고 문 헌

[1] Clements, P. and Northrop, L., "Software Architecture: An Executive Overview," CMU/SEI-96-TR-003, Carnegie Mellon University, February 1996.

[2] Bass, L., Clements, P., and Kazman, R., *Software Architecture in Practice*, Addison-Wesley, 1998.

[3] Kazman, R. et al., "The Architecture Tradeoff

Analysis Method," *Proceedings of the 4th IEEE International Conference on Engineering of Complex Computer Systems*, August 1998, pp.68-78.
[4] Bosch, J., *Design and Use of Software Architecture*, Addison-Wesley, 2000.
[5] Ambriola, V. and Kmiecik, A., "Architectural Transformations," *Proceedings of the 14th IEEE International Conference on Software Engineering and Knowledge Engineering*, July 2002, pp.275-278.
[6] Krikhaar, R., et al., "A Two-phase Process for Software Architecture Improvement," *Proceedings of the International Conference on Software Maintenance*, August 1999, pp.371-380.
[7] Carriere, S.J., Woods, S., and Kazman, R., "Software Architectural Transformation," *Proceedings of the 6th Working Conference on Reverse Engineering*, October 1999, pp.13-23.
[8] Fahmy, H. and Holt, R.C., "Using Graph Rewriting to Specify Software Architectural Transformations," *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, September 2000, pp.187-196.
[9] Kruchten, P., "The 4+1 View Model of Software Architecture," IEEE Software, Vol. 12, No. 6, November 1995, pp.42-50.
[10] Perry, D.E. and Wolf, A.L., "Foundations for The Study of Software Architecture," ACM SIGSOFT Software Engineering Notes, Vol. 17, No. 4, October 1992, pp.40-52.
[11] Garlan, D. and Shaw, M., "An Introduction to Software Architecture," Advances in Software Engineering and Knowledge Engineering, Series on Software Engineering and Knowledge Engineering, Vol. 2, World Scientific Publishing Co., 1993, pp.1-39.
[12] Choi, H. and Yeom, K., "An Approach to Effective Software Architecture Evaluation in Architecture-Based Software Development," *Journal of the Korea Information Science Society: Software and Application*, Vol. 29, No. 5, June 2002, pp.295-310.
[13] Choi, H. and Yeom, K., "An Approach to Software Architecture Evaluation with the 4+1 View Model of Architecture," *Proceedings of the 9th Asia Pacific Software Engineering Conference*, December 2002, pp.286-293.
[14] Ran, A. and Kuusela, J., "Design Decision Trees," *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996, pp.172-175.
[15] Gustafsson, J., et al., "Architecture-Centric Software Evolution by Software Metrics and Design Patterns," *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, March 2002, pp.58-70.
[16] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns*, Addison-Wesley, 1995.
[17] Buschmann, F., et al., *Pattern-Oriented Software Architecture, A System of Patterns*, John Wiley & Sons, 2000.
[18] Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F., *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, John Wiley & Sons, 2000.
[19] Eriksson, H.E. and Penker, M., *UML Toolkit*, John Wiley & Sons, 1998.

최 희 석

1998년 2월 부산대학교 컴퓨터공학과(공학사). 2000년 2월 부산대학교 컴퓨터공학과(공학석사). 2000년 3월~2003년 8월 부산대학교 컴퓨터공학과 박사과정(2002년 2월 박사수료). 2003년 9월~2004년 4월 한국정보컨설팅 선임컨설턴트. 2004년 5월~현재 한국전자통신연구원 연구원. 관심분야는 소프트웨어 아키텍처, 도메인 공학, 임베디드 소프트웨어, 유비쿼터스 컴퓨팅 미들웨어 등임

염 근 혁

1985년 2월 서울대학교 계산통계학과(학사). 1992년 8월 Univ. of Florida 컴퓨터공학과(석사). 1995년 8월 Univ. of Florida 컴퓨터공학과(박사). 1985년 1월~1988년 2월 금성반도체 컴퓨터연구실 연구원. 1988년 3월~1990년 6월 금성사 정보기기연구소 주임연구원. 1995년 9월~1996년 8월 삼성SDS 정보기술연구소 책임연구원. 1996년 8월~현재 부산대학교 컴퓨터공학과 부교수. 부산대학교 컴퓨터 및 정보통신 연구소 연구원. 관심분야는 소프트웨어 재사용, 프로덕트라인 공학, 소프트웨어 아키텍처, 컴포넌트 기반 소프트웨어 개발, 객체지향 소프트웨어 개발방법론, 요구 검증 등임