자기적응형 소프트웨어를 위한 목표 기반의 외부상황 평가 기법
(Goal-based Evaluation of Contextual Situations for Self-adaptive Software)

김 재 선 † 박 수 용 ††
(Jae Sun Kim) (Sooyong Park)

요 약 기존의 컴퓨팅 패러다임에서 개발자들은 잘 정의되고 고정된 실행 환경을 가정하고 소프트웨어를 설계하였다. 그러나 실제 실행 환경은 복잡하기 때문에 발생되는 상황들을 완벽하게 분석하는 것은 불가능하다. 그로 인해서 원하는 입력 값만을 가정하고 구현한 소프트웨어는 실행 중에 실패(failure)가 발생되기 쉽다. 이에 대한 해결책으로 자기적응형 소프트웨어(self-adaptive software)는 예상하지 못한 상황에 대해서 적응하여 실행 중의 실패가 발생되는 것을 막을 수 있다. 이를 위해 자기적응형 소프트웨어는 우선 적응의 필요성을 판별하기 위해서 실행 중에 외부 상황을 평가해야 한다. 기존의 연구들은 외부 상황의 문제를 판별하기 위한 추상화(abstraction) 기법을 제공하지 않는다. 따라서 외부 환경이 복잡해짐에 따라서 문제 자체를 판별하는 데에 한계가 발생된다. 그리고 판별 가능한 외부 상황 문제의 확장성을 지원하지 못한다. 본 연구에서는 이를 해결하기 위한 기법으로 목표(goal) 기반의 외부 상황 평가 기법을 제안한다.

키워드 : 자기적응형 소프트웨어, 외부 상황, 외부상황 문제 평가, 목표, 소프트웨어 아키텍처, 모니터링

Abstract In the traditional computing paradigm, developers design software to run in a fixed and well-defined environment. The real environment, however, is too complicated to analyze all situations perfectly. Consequently, traditional software, which is implemented only for what is wanted as input, often fails badly in real environment. As a new approach, self-adaptive software can avoid runtime failures adapting to unpredictable situations. *Self-adaptive software* must firstly evaluate the contextual situation to determine the need for adaptation. Existing researches do not support the abstraction mechanism for identifying contextual problem. Consequently, they can have troubles with identifying the contextual problem as the execution environment is getting complex. In addition, they cannot support the expandability for contextual problems, which software can evaluate. This paper suggests the goal-based evaluation method of contextual situation for coping with the limitations of existing researches.

Key words : self-adaptive software, contextual situation, contextual problem evaluation, goal, software architecture, monitoring

1. 서 론

소프트웨어의 실행 환경은 점차 복잡해지고 있다. 그러나 기존의 컴퓨팅 패러다임에서는 요구사항 분석을 통해서 실행 환경을 잘 정의하여 고정된 환경을 가정하고 소프트웨어를 설계한다. 그러나 실제 소프트웨어의 실행 중에는 예상하지 못한 상황들이 발생된다. 이러한 모든 상황들을 완벽하게 분석하고 설계하는 것은 어려운 일이다. 이로 인해 미리 가정한 상황에서 실행되도록 설계된 기존의 소프트웨어는 정상적인 실행을 유지하지 못하고 실패(failure)를 일으키게 된다[5]. 특히 임베디드 어플리케이션에서는 설치 후에 개발자가 소프트웨어를 지속적으로 관리를 할 수 없는 형태이다. 따라서 외부의 관리 없이 실행을 정상적으로 유지해야 하기 때문에 문

· 본 연구는 한국과학재단 목적기초연구 R01-2003-000-10197-0지원으로 수행되었음
† 학생회원 : 서강대학교 컴퓨터학과
faz@sogang.ac.kr
†† 정 회 원 : 서강대학교 컴퓨터학과 교수
sypark@sogang.ac.kr
논문접수 : 2005년 4월 29일
심사완료 : 2006년 2월 10일

제가 보다 심각하다.

유비쿼터스 컴퓨팅(ubiquitous computing) 환경에서의 소프트웨어는 보다 복잡한 컴퓨팅 환경에서 실행되어야 한다. 환경 내의 많은 물리적 개체들이 컴퓨팅 개체가 될 수 있기 때문에 상호 작용이 보다 복잡하다. 따라서 소프트웨어가 강건하게 실행하기 어렵다. 이러한 환경은 기존의 소프트웨어 개발자들에게 새로운 도전을 요구한다. Weiser가 말하는 '고요한 컴퓨팅(calm computing)'을 이루기 위해서는 사람의 개입 없이 소프트웨어가 스스로 적응해야 하기 때문이다[14].

위의 문제를 해결하기 위해서는 강건성(robustness)을 획기적으로 향상시킬 수 있는, 새로운 소프트웨어 패러다임이 필요하다. 이에 대한 해결 방안으로 자기적응형 소프트웨어(Self-Adaptive Software) 연구가 시작되었다. DARPA는 1997년 Broad Agency Announcement를 통해서 자기적응형 소프트웨어를 다음과 같이 정의한 바 있다[1]: *자기적응형 소프트웨어는 자신의 행위를 평가한다. 소프트웨어의 의도대로 수행하고 있지 않다고 판단되거나 더 나은 성능으로 향상시킬 수 있다고 판단될 때, 자기적응형 소프트웨어는 자신의 행위를 변경한다. (Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible)*

자기적응형 소프트웨어를 구현하기 위해서는 아래의 이슈들이 해결되어야 한다[4]:

1. 모니터링(Monitoring): 실행 중인 소프트웨어의 상태를 모니터링 한다. 모니터링의 대상은 소프트웨어의 실행 환경도 포함될 수 있다.
2. 평가(Evaluation): 소프트웨어에게 발생된 문제를 분석하고 적응의 필요성을 판단한다. 문제의 심각성 정도에 따라서 적응의 필요여부가 결정될 수 있다.
3. 문제 해결(Resolution): 발생된 문제를 해결하기 위하여 적응 전략을 찾아낸다.
4. 재구성(Reconfiguration): 적응 전략에 따라서 실행 중인 소프트웨어의 구조와 행위를 동적으로 변경한다.

위의 이슈들 중에서 평가(evaluation)는 가장 중요하고 어려운 문제이다[6]. 이 연구 분야는 초기 단계이기 때문에 관련 연구 및 적용된 사례가 적다.

자기적응형 소프트웨어의 평가를 위해서는 소프트웨어의 예측에서 벗어난 행위의 원인을 분석해야 한다. 원인을 소프트웨어의 내부상태(internal status)와 외부상황(contextual situation)에서 찾아볼 수 있다. 소프트웨어의 내부 상태에서의 예로 시스템의 작동 상의 에러가 있을 수 있다. 외부 상황은 소프트웨어의 행위에 영향을 미친다. 때문에 예측하지 못한 외부 환경의 변화로 인해 소프트웨어가 의도했던 대로 실행되지 않을 수 있다. 예로 컴퓨팅 자원 부족의 문제이나 상호 작용하는 컴퓨팅 개체의 에러로 인한 문제가 있을 수 있다.

본 연구에서 다루고자 하는 평가 대상은 외부상황이다. 사실 소프트웨어의 내부상태와 외부상황은 긴밀하게 연결 지어 고려되어야 하는 대상들이다. 발생된 문제의 원인이 내부적인 문제이거나 외부의 원인으로부터 내부 문제가 발생된 것일 수 있다.

본 연구에서 풀고자 하는 문제를 다음과 같이 정의한다.

A. 외부상황 문제를 효과적으로 평가하기 위한 기법을 제공한다.
B. 예상하지 못한 문제상황을 평가하기 위해 평가 가능한 외부상황문제 집합을 확장할 수 있어야 한다.

기존의 관련 연구들은 다음과 같은 문제점들을 지닌다.

1. 위의 문제를 해결하기 위해서는 우선 외부 상황에서 발생되는 문제를 분석하는 것이 필요하다. 이를 위해서는 외부상황 문제를 위한 추상화기법(abstraction mechanism)이 요구된다. 그러나 기존 연구에서는 이를 제공하지 않는다. 즉, 외부상황 문제의 발생 범위, 심각한 정도 등을 모델링 하기 위한 도구를 제공하지 않는다. 이로 인해 소프트웨어의 복잡한 실행 환경에서 외부상황 문제의 판별에 한계가 발생된다.
2. 외부 상황 문제들을 모델로 관리하지 않고 있기 때문에 평가 가능한 외부상황 문제 영역의 확장이 어렵다. 특히 실행 중에 발생된 예상하지 못한 상황을 평가하기 위해서는 평가 가능한 외부상황 문제 영역의 확장은 중요한 것이다.

본 논문에서 제안하는 목표 기반의 외부상황평가 기법, GECS(Goal-based Evaluation of Contextual Situation)은 다음과 같은 해결책을 제시한다.

1. 목표(goal)를 기반으로 외부상황 문제(contextual problem)를 표현하는 추상화 기법(abstraction mechanism)을 제공한다.
2. 외부상황 문제 모델들을 관리함으로써 평가 가능한 외부상황 문제 영역의 확장성을 지원한다. 이를 통해 예측하지 못한 상황에 대한 평가가 가능해진다.

본 논문에서는 우선 2장을 통해서 관련 연구를 분석한다. 관련 연구들은 외부상황의 평가 문제를 어떻게 해결하는 지 분석한다. 3장을 통해서 목표 기반의 외부상황 평가 기법, GECS(Goal-based Evaluation of Contextual Situation)을 제안한다. 4장은 사례 연구를 소개한다. 5장에서 본 연구 결과를 논의한 후, 6장에서 결론을 맺도록 하겠다.

2. 관련 연구

소프트웨어의 외부 상황을 활발하게 다루는 연구로 Context-aware computing(CAC) 연구가 있다. CAC에서 외부상황(Context)은 사용자가 어디에 있는가(where you are), 사용자의 주위에 무엇이 있는가(who you are with), 사용자의 주위에 어떤 자원들이 있는가(what resources are nearby)와 같은 관점들에 따라 정의된다. 외부상황을 인식하여 사용자와 시스템 사이의 풍부한 상호작용 기법의 제공이 목적이다[18,19]. CAC 연구는 외부상황 평가문제를 푸는 데에 도움을 줄 수 있다. 상황인지 미들웨어(Context-aware middleware)를 이용하여 외부상황의 상태를 파악할 수 있다[20]. 그러나 CAC에서는 외부상황의 평가기법을 다루지 않는다. CAC의 외부상황을 보는 관점은 자기적응형 소프트웨어의 것과는 다르기 때문이다. 자기적응형 소프트웨어는 외부상황이 소프트웨어의 기능 수행에 어떤 영향을 주는지 평가해야 한다. 즉, 기능 유지에 문제가 되는 상황이거나 소프트웨어의 성능을 향상시킬 수 있는지를 판단할 수 있어야 한다. 따라서 자기적응형 소프트웨어의 관점에서 외부상황은 CAC에서의 것과는 달리, 소프트웨어 실행에 영향을 주는 자원(메모리, 네트워크), 입력(데이타, 사용자 입력) 등의 관점에 따라 분석되어야 한다.

Shrobe는 모델기반의 진단(model-based diagnosis) 기법을 기반으로 외부상황 평가문제를 해결하는 기법을 제안하였다[9]. 기존의 모델기반의 진단(model-based diagnosis)은 물리적인 하드웨어를 다루는 연구이다[29]. 이를 소프트웨어 시스템에게 적용할 수 있도록 하였다. 우선 컴퓨팅 개체들이 소프트웨어 내부의 컴포넌트들에게 영향을 주는 관계를 그래프로 모델링 한다. 외부의 자원 컴포넌트(resource component)들의 상태(정상, 비정상)에 의해 소프트웨어의 실행이 어떻게 영향 받는지 평가한다. 베이지안 네트웍(Bayesian network)을 이용하여 자원 컴포넌트가 소프트웨어의 실행에 확률적으로 어떻게 영향을 주는지 측정한다. 외부상황 문제의 불확실성(uncertainty)을 베이지언 네트웍을 이용해서 풀었다는 장점을 지니고 있으나, 평가의 대상이 단순히 계산상의 지연(computational delay)의 문제만을 다루고 있다는 약점을 지닌다.

Garlan은 게이지(gauge)를 이용해서 아키텍처 기반의 성능 모니터링을 통한 평가 기법을 제안하였다[7]. 분산 시스템에서 시스템 간의 링크에 게이지를 설치하여 얻어낸 시스템 하위 레벨의 정보들을 수집하여 아키텍처 상의 의미로 해석한다. 이를 적응의 기반으로 이용한다. EP1에서 성능의 문제만 다룬다는 한계를 지닌다.

외부상황에서 특정 상황을 판별하기 위해 쓰일 수 있는 기본적인 도구로 제약조건(constraint)을 이용할 수 있다. 선언(assertion)은 컴포넌트에 대한 사전/사후 조건(pre/post conditions)에 대한 제약조건을 미리 형식적인 명세서로 기술한다. 소프트웨어의 올바른 동작을 보장하는 제약조건 모델을 소프트웨어 내부에 삽입하여 문제 상황 식별 기준으로 활용하는 것이다[27,28]. 관련된 연구로 Neema가 제안한 기법에서는 시스템의 비기능정 요구사항, 자원에 제약조건의 정보를 형식적인 제약조건으로 추출하여 문제 상황을 판단한다[10]. 소프트웨어의 아키텍처 상의 문제로 해석하기 위한 메커니즘을 지원하지 않기 때문에 시스템 상의 문제점을 진단하기 어렵다.

Lerner는 Containment Unit 연구를 제안한 바 있다[12,13]. Containment unit은 제약조건 명세(constraint specification)를 위한 유닛(unit)과 적응을 위한 유닛으로 구성되어 스스로 진단이 가능한 단위이다. 제약조건 명세는 컴포넌트가 실행되기에 적합한 환경을 정의한 것이다. Containment Unit 내부에 있는 평가기(evaluator)는 명세를 이용하여 실행 시간, 메모리 이용상황, 성능 검사 등을 통해 평가를 수행한다. 외부상황 문제를 아키텍처의 유닛 단위로 나누어서 고려한 연구이다. 아키텍처의 유닛단위로 가정상황과 다른 것을 판별하기 때문에 하나 이상의 유닛에 걸쳐있는 아키텍처 상의 문제를 파악하기에 어렵다.

기존 연구들에서는 외부상황 문제를 식별하기 위한 추상화 기법을 지원하지 않는다. 기존의 연구들은 주로 실행 환경의 자원에 대한 조건들을 이용해서 특정 문제 상황을 정의한다. 그러나 외부상황의 문제는 하나 이상의 외부 개체들에 의해 발생되는 것으로 소프트웨어 관점에서 묶어서 해석할 수 있어야 한다. 추상화 기법의 부재로 인해 다음의 문제들이 발생된다.

- 복잡한 환경에서의 외부상황 문제를 추상화하지 못한다. 외부상황 문제의 발생 범위와 심각한 정도 등을 모델링 하기 위한 도구를 제공하지 못한다. 이로 인해 문제의 복잡성으로 인해 문제의 판별에 한계를 지닌다.
- 외부상황 문제의 추상화를 지원하지 않기 때문에 평가 가능한 문제 영역의 확장이 어렵다. 이로 인해 실행 중에 발생된 예측하지 못한 상황의 평가가 불가능하다.

3. 목표 기반의 외부 상황 평가(Goal-based Evaluation of Contextual Situation) 기법 제안

본 연구에서는 소프트웨어가 실행 중에 추구하는 목

표(Goal)를 기반으로 외부상황(Contextual Situation) 문제를 평가(Evaluation)하는 것을 제안한다. 본 연구 기법의 기본 개념은 인간이 목표기반으로 생각하는 방식으로부터 출발한다. 인간은 목표에 따라서 행동하고, 목표를 근거로 주변에서 발생된 사건들을 평가한다. 이와 같은 원리로 소프트웨어 관점에서도 목표를 근거로 문제를 표현, 발견, 그리고 평가하는 것이 가능하다.

목표기반의 접근은 인공지능 연구진영에서의 에이전트 기술로부터 나온 것이다. 목표의 정의는 다음과 같다[31]: "에이전트는 바람직한 상황을 기술하는 목표 정보를 지닌다." 기존의 연구들, 예를 들어, 요구 공학 등이 다양한 쓰임으로 목표 개념을 채용하였다. 자기적응형 소프트웨어의 일부 측면은 에이전트와 유사하다고 볼 수 있다. '스스로' 문제를 파악하고 적응하는 성질은 에이전트 성질과 상당히 유사하다. 그렇기 때문에 본 연구는 자기적응형 소프트웨어를 위한 연구기법에 목표기반 접근을 응용하고자 한 것이다.

본 연구는 아키텍처 기반의 자기적응형 소프트웨어를 따른다[4,8]. 그림 1은 외부상황 문제에 대한 적응을 중점적으로 보여주는 자기적응형 소프트웨어의 구성 컴포넌트 간의 프로세스를 보여준다.

소프트웨어는 소프트웨어 환경(Software Context)에서 실행한다. 소프트웨어 환경은 다양한 컴퓨팅 자원들로 구성되고, 이들 구성요소들을 외부상황 팩터(Contextual Factor, C-factor)라고 한다. 외부상황모니터(Contextual Situation Monitor)는 외부상황 팩터들을 모니터링하여 이들의 현재 상태값을 가져온다. 이들 외부상황 팩터들의 값들로 구성된 외부상황 인스턴스를 외부상황문제 탐지기(*Contextual Problem Detector*)에게 넘겨준다. 외부상황문제 탐지기(*Contextual Problem Detector*)는 외부상황 인스턴스에서 문제를 발견하고 외부상황문제 평가기(*Contextual Problem Evaluator*)에 의해 적응의 필요성을 평가하게 된다. 평가 결과를 기반으로 적응전략 계획자(Adaptation Strategy Planner)는 문제에 대처하기 위한 적응전략을 선택한다. 재구성자(Reconfigurator)는 전략을 넘겨받아 실행 중인 소프트웨어의 행위를 동적으로 재구성한다. 외부상황문제 관리(*Contextual Problem Management*)는 컴포넌트가 아닌 관리 프로세스로서 분석 가능한 외부상황 문제의 종류를 확장하는 것을 지원한다.

위의 그림 1에서 색으로 채워진 컴포넌트는 본 연구 범위를 가리킨다. 각 범위에 대해서 본 연구는 다음과 같은 기법을 제안한다.

- *목표 기반의 외부상황문제 탐지기(Goal-based Contextual Problem Detector, GCPD)*
- *목표 기반의 외부상황문제 평가기(Goal-based Contextual Problem Evaluator, GCPE)*
- *목표기반의 외부상황문제 관리(Goal-based Contextual Problem Management, GCPM)*

3.1 소프트웨어, 목표, 외부 상황

본 장에서는 본 연구가 가정하는 소프트웨어, 목표, 그리고 외부 상황과 이들 간의 관계를 기술한다. 그림 2는 세 구성 요소간의 관계를 보여준다.

3.1.1 외부 상황(Contextual Situation)

소프트웨어 실행 환경 안에는 사람, 네트웍, 그리고 여러 종류의 컴퓨팅 개체 등의 여러 외부상황 팩터(Contextual factor, C-factor)들이 있다. 이들 외부상황 팩터들의 값은 소프트웨어의 실행 중 지속적으로 변한다.

특정 값을 지닌 외부상황 팩터들의 집합은 하나의 외부상황을 이룬다. 임의의 외부상황은 소프트웨어의 실행

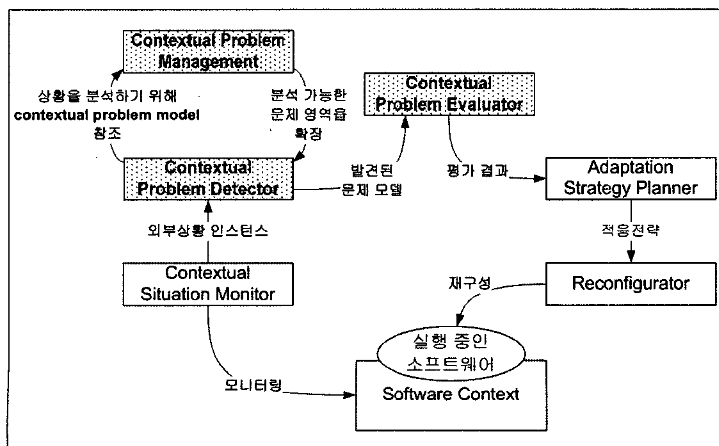


그림 1 자기적응형 소프트웨어의 구성 컴포넌트 간의 프로세스 및 본 연구의 범위

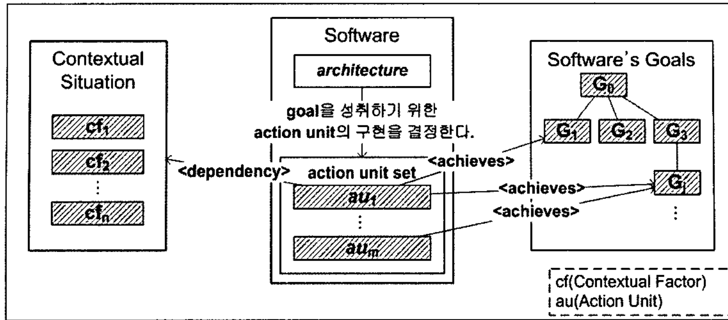


그림 2 본 연구에서 가정하는 소프트웨어, 목표, 외부상황과의 관계

에 영향을 미친다. 이는 그림 2에서 소프트웨어의 외부상황에의 의존성에 해당된다. 소프트웨어 관점에서 외부상황은 소프트웨어의 목표를 성취하기 위해서 필요한 입력 역할을 한다. 구체적으로 기술하면, 외부상황 팩터들이 소프트웨어의 행위에 영향을 미치는 타입에 따라서 요청 타입 또는 조건 타입의 팩터로 나뉜다. 요청타입의 팩터의 예로 네트웍을 사용하기 위해 준수해야 하는 네트웍 프로토콜, 서비스 요청 이벤트 등이 해당된다. 이들 팩터들은 소프트웨어 행위에게 문제 해결을 요구한다. 조건타입의 팩터는 소프트웨어가 실행에 필요한 메모리 자원, 네트웍 자원, 파일 공간 등의 조건을 제공한다. 일부 외부상황 팩터의 경우 요청타입이면서 동시에 조건타입일 수 있다. 예로 네트웍이 해당된다. 프로토콜의 구현을 요구하는 동시에 네트웍을 이용할 수 있는 조건을 제공한다.

3.1.2 소프트웨어의 행위단위(Action Unit)

본 연구에서 소프트웨어의 행위를 표현하는, 아주 작은 단위로서 '행위단위(Action Unit)'를 정의한다. 소프트웨어의 실행 중에 관찰 가능하며 실행의 시작점과 끝점을 가진다. 예를 들어, 객체지향 프로그래밍에서 클래스의 메서드(method)가 하나의 행위단위로 정의될 수 있다. 행위단위는 소프트웨어 아키텍처 스타일에 종속적이다. 아키텍처 스타일에 따라서 소프트웨어가 시스템 목표를 달성하기 위해 동작하는 방식이 달라지기 때문이다. 이러한 원리에 의해 자기적응형 소프트웨어는 아키텍처의 구성을 바꿈으로 해서 행위를 다르게 할 수 있다[8]. 행위단위의 크기는 어플리케이션 개발자에 의해서 정의되며, 크기는 작을수록 좋다. 소프트웨어가 실행하고 있는 특정 시간에 소프트웨어가 정확히 어떤 행위를 수행하고 있는지 정확하게 모니터링할 수 있기 때문이다.

기본적으로 소프트웨어가 어떤 시점에서 실행 가능한 행위의 경우 수는 한정된다. 따라서 행위단위 집합을 정의한 후, 행위단위를 단위로 현재 소프트웨어가 어떤 행위를 수행하고 있는지 파악할 수 있다. 현재 실행 중인 행위단위를 파악하는 것뿐만 아니라 다음에 실행 가능한, 하나 이상의 행위들을 예측할 수도 있다. 이는 행위단위들간에 선후 관계를 파악함으로써 가능해진다. 행위단위 간의 선후관계는 구체적으로 'call'과 'precede' 관계로 나뉜다. 행위단위 A_i가 행위단위 A_j를 호출할 경우, A_i에서 A_j로의 'call' 관계가 성립된다. 행위단위 A_i가 실행된 후에 행위단위 A_k 가 실행될 경우, A_i에서 A_k로 'precede' 관계가 성립된다. 이 때 A_i는 A_j와 A_k보다 먼저 실행된다.

3.1.3 목표(Goal)

시스템의 목표는 목표 그래프로 모델링된다. 목표 그래프는 목표기반의 요구공학 기법을 통해서 설계될 수 있다[23]. 그러나 본 연구에서의 목표의 성질은 요구공학에서의 목표의 것과는 다른 점을 가진다. 본 연구에서 목표는 소프트웨어의 실행 중에 관찰되고 평가될 수 있도록 해야 하기 때문에 런타임에 대한 성질들을 이용하여 설계되어야 한다. 그림 3은 목표 그래프의 예시를 보여준다.

본 연구에서 이용하는 목표 그래프에는 부모 목표(parent goal)와 서브 목표(sub-goal)와의 정량적 관계를 가지고 있다. 목표 그래프를 보면 목표 간의 관계 선

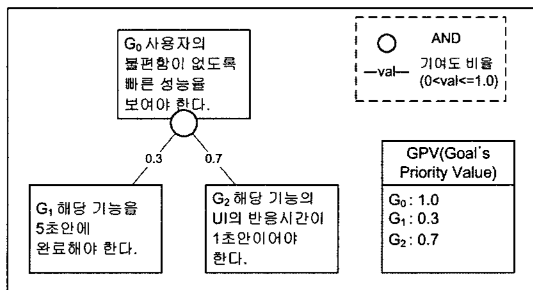


그림 3 목표 그래프의 예시 및 GPV(Goal's Priority Value)

에 숫자가 첨부되어있음을 알 수 있다. 이들 값은 부모 목표를 달성하기 위한 기여도 비율을 나타내며, 0에서부터 1까지의 값을 가질 수 있다. 예를 들어, 목표 [G_0]은 루트 목표(Root Goal)이다. [G_1]과 [G_2]는 [G_0]을 달성하기 위한 서브 목표이다. G_1은 G_0을 달성하기 위해서 30%만큼을 기여한다. 즉, [G_0]의 30%가 [G_1]에 의해서 만족된다는 것을 가리킨다.

이들 관계들을 이용하여 전체 목표 그래프에서의 특정 목표의 우선순위, 또는 비중 값(Goal's Priority Value, GPV)을 계산할 수 있다. 예를 들어, [G_0]의 GPV값은 1(100%)이다. [G_2]은 [G_0]의 GPV값에 0.7 (70%)을 곱하여 계산하여 0.7(70%)이 된다. 이는 전체 시스템에서 [G_2]이 70%만큼의 비중을 또는 우선순위를 가진다는 것을 뜻한다.

소프트웨어는 시스템 목표를 달성하기 위해서 하나 이상의 행위단위들을 수행한다. 그림 2는 소프트웨어의 행위단위들이 목표를 성취하는 관계를 보여준다. 이러한 관계는 기능과 비기능(예, 성능, 보안 등)의 측면으로 설명될 수 있다[23]. 이는 행위단위의 실행 결과로 달성여부를 판별할 수 있다[24]. 예를 들어, '10초 내로 메시지를 전송해야 한다' 라는 목표를 달성하기 위해 소프트웨어가 행위단위, '메시지 전송하기'를 실행 했을 때, 기능적 측면은 메시지를 전송한 결과가 출력된 것으로 드러나고, 비기능적 측면은 전송 시간이 10초 내로 이루어진 사실에 의해 드러난다. 역으로 실행중인 행위단위를 단위로 현재 어떤 서브 목표(sub-goal)들을 달성하고 있는 중인지 파악할 수 있다. 행위단위는 하나 이상의 목표와 관련될 수 있다. 예를 들어, '메시지 전송하기' 행위는 '메시지를 외부에 유출되지 않도록 안전하게 전송해야 한다' 라는 목표와도 연결될 수 있다. 행위단위들 중에서 목표 달성에 중요하지 않다고 판단될 경우에는 목표와 관련성이 없다고 설정한다.

앞장에서 기술한 바 있듯이 행위단위를 단위로 현재 실행 중인 소프트웨어의 행위를 파악 가능하다. 위와 같은 행위단위와 목표와의 연결 관계를 기반으로 현재 소프트웨어가 어떤 목표를 달성하기 위해서 실행하고 있는지 역시 파악 가능하다.

3.1.4 외부상황, 행위단위, 목표 사이의 영향력의 크기

그림 2에서 기술된 관계에서는 단순한 연결 관계만 정의되어 있다. 그러나 본 연구에서는 외부상황문제로부터 목표에 이르는 영향관계를 기반으로 목표기반의 평가를 하기 때문에 보다 정확한 영향 관계를 정의할 필요가 있다. 이를 위해 각 관계 상에서 발생되는 영향력의 크기를 정의하고자 한다. 아래 그림 4는 그림 2를 확장하여 영향력의 크기가 반영된 관계를 보여준다.

우선 외부상황 팩터(C-factor)들의 행위단위(action unit)로의 영향력의 크기가 존재한다. 다시 말해 행위단위의 외부상황팩터로의 의존성이라고도 할 수 있으며, DF(Dependency on C-Factor)라고 정의한다. 외부상황 팩터들이 하나의 행위단위에게 영향을 줄 경우, 각 팩터들이 동등한 크기의 영향력을 행사한다고 보기는 어렵다. 즉 해당 행위단위가 실행하는 데에 중요한 팩터는 다른 팩터들보다 큰 영향력을 가진다고 볼 수 있다. 이러한 외부상황 팩터들의 영향력 크기는 행위단위에 따라서 달라질 수 있다. 예를 들어, '메시지 전송하기'라는 행위단위 관점에서는 '네트웍' 외부상황 팩터가 가장 큰 영향력을 가진다. '메시지 입력하기'라는 행위단위 관점에서는 '메시지 크기' 외부상황 팩터가 해당될 수 있다.

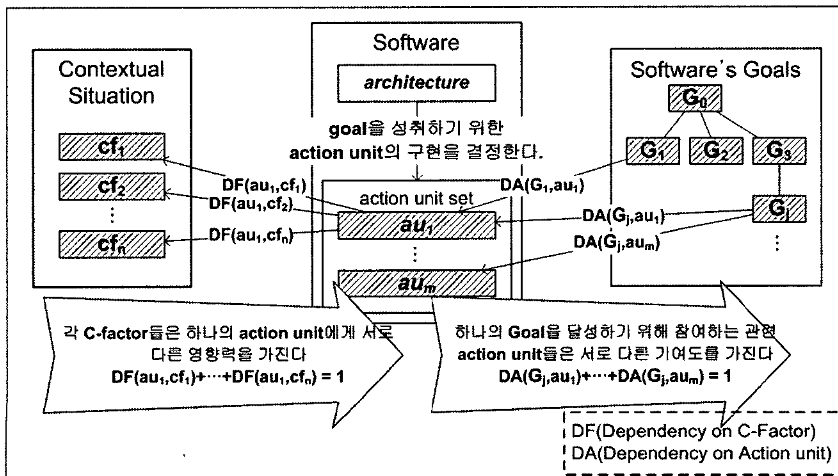


그림 4 소프트웨어, 목표, 외부상황 간의 관계상에서의 영향관계

CCG(A2, G12)		CCG(A3, G12)	
CPU:		CPU:	
If (0 <= v < 10)	PL = INF	If (0 <= v < 10)	PL = INF
If (10 <= v < 30)	PL = 100	If (10 <= v < 30)	PL = 100
If (30 <= v < 100)	PL = 0	If (30 <= v < 100)	PL = 0
MEM:		MEM:	
If (0 <= v < 0.1)	PL = INF	If (0 <= v < 0.1)	PL = INF
If (0.1 <= v < 0.2)	PL = 100	If (0.1 <= v < 0.2)	PL = 100
If (0.2 <= v < 1)	PL = 50	If (0.2 <= v < 1)	PL = 50
If (1 <= v < 10)	PL = 0	If (1 <= v < 10)	PL = 0
MSG:		MSG:	
NET:		NET:	
If (0 <= v < 5)	PL = INF	If (0 <= v < 5)	PL = INF
If (5 <= v < 10)	PL = 100	If (5 <= v < 10)	PL = 100
If (10 <= v < 50)	PL = 50	If (10 <= v < 50)	PL = 50
If (50 <= v)	PL = 0	If (50 <= v)	PL = 0

PL (Problem Level)

그림 5 CCG(Contextual Constraints for Goal)의 예

행위단위 A에게 영향을 주는 전체 외부상황팩터들 중에서 특정 외부상황 팩터 F의 행위단위 A로의 영향력의 비율을 DF(A, F)로 표기한다.

행위단위에서 목표로의 영향력의 크기, 또는 목표의 행위단위로의 의존성 크기를 DA(Dependency on Action Unit)으로 정의한다. 하나의 목표를 성취하기 위해서는 하나 이상의 행위단위들이 참여하게 된다. 이때 각 행위단위들이 동등한 크기의 기여도를 지닌다고 볼 수 없다. 목표의 달성을 위해서 가장 중요한 역할을 하는 행위단위는 다른 행위단위보다 더 큰 영향력을 지닌다. 목표 G와 관련되는 행위단위들 중에서 특정 행위단위 A의 목표 G로의 기여도의 비율을 DA(G, A)라고 표기한다.

3.2 목표 기반의 외부상황 문제(Goal-based Contextual Problem)

소프트웨어가 의도한대로 실행하기 위해서 외부상황 조건을 가정할 수 있다. 실행 중에 지속적으로 변하는 외부상황 인스턴스들이 외부상황조건을 만족하지 못했을 경우 소프트웨어에게 문제가 발생될 수 있다. 이 경우를 외부상황문제라고 정의한다.

자기적응형 소프트웨어의 Adaptation Strategy Planner (그림 1 참조)는 소프트웨어 아키텍처를 기반으로 적응 전략을 선택한다[8]. 이를 위해서 외부상황 문제는 아키텍처 문맥으로 표현되어야 한다.

본 연구에서는 외부상황 문제를 표현하기 위한 수단으로 목표를 제안한다. 목표는 아키텍처 설계의 근거를 제공하기 때문에 아키텍처 문맥에서의 문제를 표현하는데 쓰일 수 있다[24].

소프트웨어의 하나의 행위단위가 목표를 성취하기 위해서 요구되는 외부상황 조건(Contextual Constraints for Goal, CCG)을 정의한다. 이는 에이전트가 목표를 성취하기 위해서 필요한 상황조건을 정의하는 것과 같은 개념이다[32]. 예를 들어, '전송하기'라는 행위단위가 '메시지 전송을 빠르게 하기'라는 목표를 달성하는 것에 참여한다고 하자. 이 행위단위는 외부상황팩터로서 네트웍속도, 메시지길이 등과 관련된다. 이 행위단위를 위한 외부상황 조건은 '네트웍 속도는 100Mbps보다 높아야 한다'와 같이 정의할 수 있다. 만약 외부상황이 이 조건을 만족하지 않을 경우 행위단위는 해당 목표를 성취하는 데에 어려움을 가지게 될 것이다. 이 경우 외부상황 문제는 목표를 단위로 표현되기 때문에 목표 기반의 외부상황 문제(Goal-based Contextual Problem)가 발생되었다고 한다. 다음은 CCG의 예시이다.

CCG는 외부상황을 구성하는 외부상황 팩터들의 조건들로 이루어진다. 각 외부상황 팩터의 조건에 따라서 해당 행위단위가 어느 정도의 영향을 받는지를 설계한다. 영향 받는 정도, 즉 문제의 심각성 정도를 PL(Problem Level)이라고 정의한다. PL은 0%, 50%, 100%, INF을 가질 수 있다. 0%은 아무런 문제가 없음을 뜻하고, 50%은 문제가 어느 정도 있으나 실행에 큰 문제는 안됨을 뜻한다. 100%은 심각한 상태로 실행에 지장을 줄 수 있는 상태이다. INF는 치명적인 상태로 행위단위의 실행이 불가능함을 뜻한다.

목표기반의 외부상황문제를 구체적으로 정의하면 다음과 같다. 실행중인 소프트웨어의 시스템 클럭의 현재 값은 t일 때 외부상황 인스턴스는 CS(t)이라고 하자. 이때 소프트웨어는 행위단위 A_i를 실행하고 있다. 행위단위 A_i는 목표 G_j를 달성하기 위해 외부상황조건을 가정한다. 이 조건을 CCG(A_i, G_j)라고 표기한다. CS(t)가 CCG(A_i, G_j)를 만족하지 않을 경우, 목표 G_j에서 목표기반의 외부상황문제가 발생되었다고 한다. 그림 6은 목표기반의 외부상황문제가 발생되는 시나리오를 보여준다.

목표기반의 외부상황문제는 다음의 성질을 지닌다:

- 문제의 유형, 범위: 시스템의 목표 그래프에서의 목표

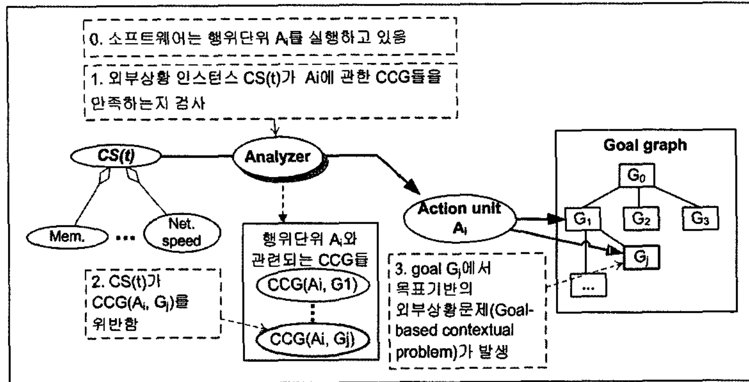


그림 6 목표기반의 외부상황 문제(Goal-based Contextual Problem) 발생 시나리오

G_j는 시스템의 최종 목표를 달성하는 데에 수행하는 역할을 나타낸다.

- 문제의 심각성 정도 값: GPV(Goal's Priority Value), CPL(Contextual Problem Level).
- 문제들간의 관계: 목표 그래프 안에서 목표들간의 관계를 따른다. 목표 간에는 AND, OR 등의 관계가 있다[23].

위에서 특히 외부상황문제의 심각성은 중요한 것이다. 심각성 정도에 따라서 적응 여부를 판단하기 때문이다. 그러나 적응 여부의 판단 기준이 되는 심각성 정도의 경계 값을 정하는 것은 모호한 작업이다. 어플리케이션 도메인이나 실행 환경에 따라서 적응이 필요한 심각성 정도 값의 수준을 설정해야 한다.

본 연구에서는 심각성의 정도를 계산할 수 있는 척도로 GPV, CPL을 제공한다. GPV(Goal's Priority Value)는 앞서 기술한 바 있듯이, 전체 시스템에서 해당 목표가 어느 정도의 비중을 차지하는 지를 가리킨다. 이를 통해서 발견된 문제가 전체 시스템 관점에서 어느 정도의 심각성을 지니는지 파악할 수 있다. 우선순위가 높은 목표에서 문제가 발생할수록 전체 시스템 목표 달성에 큰 영향을 미친다.

CPL(Contextual Problem Level)은 해당 목표에서 얼마나 문제가 발생했는지를 가리킨다. 예를 들어, CPL이 100%일 때는 해당 목표 전체를 달성하기 어렵다는 뜻이다. CPL이 50%일 경우는 해당 목표의 50% 정도의 달성되는 데에 문제가 있음을 뜻한다. CPL은 행위단위 관점에서 얼마나 중요한 외부상황팩터에서 문제가 발생되었느냐, 목표 관점에서 얼마나 중요한 행위단위에서 문제가 발생되었느냐에 따라서 정해진다. CPL을 구하는 식은 다음과 같다. CCG에서 문제를 일으킨 팩터들을 cf_k라고 하고, 관련 행위단위는 au_i, 관련 목표는 G_j라고 하자.

$$CPL = \sum_k (cf_k\text{'s}PL * DF(au_i, cf_k)) * DA(G_j, au_i)$$

예를 들어, CCG에서 문제를 일으킨 팩터를 cf_a, cf_b라고 할 때, (cf_a's PL×DF(au_i, cf_a) + cf_b's PL×DF(au_i, cf_b))×DA(G_j, au_i)와 같이 구할 수 있다.

3.3 목표 기반의 외부상황문제 탐지기(Goal-based Contextual Problem Detector)

본 연구에서 외부상황문제 탐지기(그림 1 참조)는 목표 기반의 외부상황문제 탐지기(Goal-based Contextual Problem Detector, GCPD)로 구체화된다. GCPD는 소프트웨어의 실행 중에 외부상황 인스턴스를 검사하여 목표기반의 외부상황문제를 발견해낸다. 이 때 외부상황문제 관리(Contextual Problem Management)에 의해 관리되는 CCG 집합을 이용한다. 아래의 그림 7은 GCPD의 프로세스를 보여준다.

1. SWClock은 주기적으로 외부상황문제 탐지기에게 현재의 외부상황이 문제를 가지고 있는지 검사하기를 요청한다.
2. Receiver는 CSMonitor(그림 2의 Contextual Situation Monitor)로부터 현재의 외부상황 인스턴스를 가져와서 EngineControl에게 넘겨준다.
3. EngineControl은 ActionTracer로부터 현재 실행 중인 행위단위 리스트를 가져온다. 행위 단위를 지속적으로 추적하기 위해서 소프트웨어의 행위단위들은 시작할 때와 수행을 마쳤을 때, ActionTracer에게 startAction()과 finishAction()을 요청해야 한다.
4. EngineControl은 실행 중인 행위단위 다음에 예정된 행위단위들을 얻는다. 3.1.2장에서 기술한 바 있듯이 행위단위들간의 선후 관계를 기반으로 다음에 실행 가능한 행위단위들을 얻을 수 있다. 이를 통해서 외부상황문제 탐지기는 앞으로 발생 가능한 문제를 예측할 수 있다. 본 연구에서 외부상황문제 탐지는 기본적으

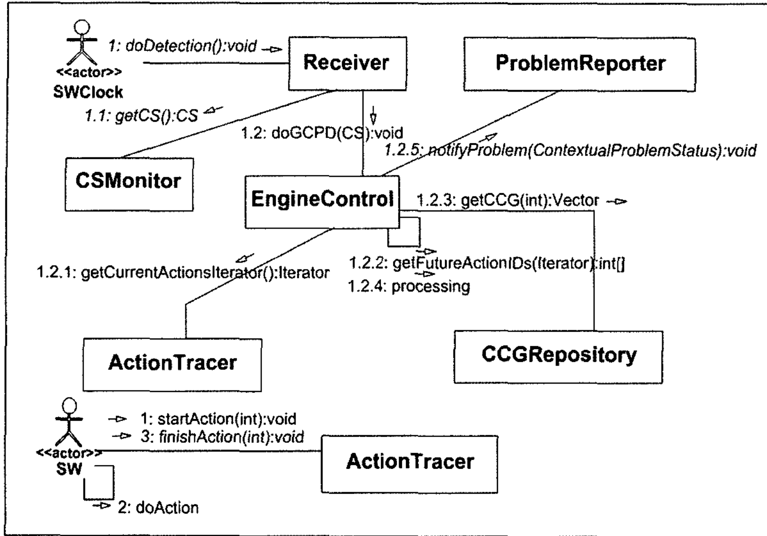


그림 7 목표 기반의 외부상황문제 탐지기(Goal-based Contextual Problem Detector)의 프로세스

로 행위단위를 기본으로 하기 때문에 가능한 것이다.

5. EngineControl은 CCGRepository로부터 선택된 행위단위들의 CCG들을 얻는다.
6. 입력으로 들어온 외부상황 인스턴스를 얻어온 CCG들을 이용하여 검사한다.
7. 분석 결과는 ProblemReporter를 통해서 GCPE에게 넘겨준다.

GCPD의 중요한 특징은 4번 절차에서도 기술한 바 있듯이 발생 가능한 문제를 예측할 수 있다는 점이다. 행위단위로 추적을 하고, 행위단위를 기준으로 외부상황 문제를 탐지하기 때문에 가능한 것이다. 소프트웨어의 실행 중 문제가 발생된 후에 탐지하는 것이 아니라 미리 탐지할 수 있기 때문에 외부상황문제에 대해서 보다 적극적인 대처를 할 수 있다는 장점을 지닌다.

3.4 목표 기반의 외부상황 문제 평가기(Goal-based Contextual Problem Evaluator)

본 연구에서 외부상황문제 평가기(그림 1 참조)는 목표기반의 외부상황문제 평가기(Goal-based Contextual Problem Evaluator, GCPE)로 구체화된다. 본 장은 GCPE에 대해 기술하고자 한다.

외부상황 문제를 평가하는 방식은 어플리케이션에 따라서 다양하게 구현되어야 한다. 어플리케이션을 구성하는 컴포넌트의 dependability, 어플리케이션의 critical한 정도 등의 성격에 따라서 다르게 풀어야 한다. 이를 해결하기 위해서 기계학습(Machine Learning)이나 에이전트(Agent) 기술을 이용해서 어플리케이션의 특화된 경험 수치들을 축적함으로 해서 평가의 성능을 향상시킬 수 있다. 이는 향후 연구에서 다룰 예정이다. 본 논문에서는 개발자의 평가 전략을 어떻게 외부상황문제 평가기에 담도록 하느냐에 초점을 두고자 한다.

GCPE는 GCPD의 결과를 기반으로 적응 여부를 판단한다. 적응 여부를 결정하기 위한 경계값으로 해당 목표의 최소 달성률(Minimum Achievement-rate)을 정의한다. 목표의 달성률이 최소 달성률보다 낮을 경우 적응이 필요하다고 판단한다. 예를 들어, 목표 g에서 외부상황 문제가 발생했다고 한다. 목표 g의 최소 달성률은 0.8이라고 하자. 만약 (1 - CPL(Contextual Problem Level))이 0.8보다 낮을 경우 적응이 필요하다고 판단하는 것이다. 최소 달성률은 목표 그래프에서 루트 목표를 포함한 주요 목표들에 설정한다. 최소 달성률을 기반으로 한 GCPE의 프로세스는 다음 표와 같다.

표 1 목표기반의 외부상황문제 평가기(Goal-based Contextual Problem Evaluator)의 프로세스

1. 문제 발생 목표가 최소 달성률이 설정되어 있을 때,
1.1 (1 - 문제 발생 목표의 CPL)가 최소달성률보다 낮을 경우, 적응이 필요하다고 판단한다.
2. 문제 발생 목표가 최소 달성률이 설정되어 있지 않을 때,
2.1 (1 - 문제 발생 목표의 CPL)을 기반으로 목표의 달성률을 계산한다.
2.2 문제 발생 목표의 달성률을 기반으로 부모 목표의 달성률을 계산한다. 부모 목표의 달성률은 각 자식 목표들의 (자식 목표의 달성률) * (자식 목표의 기여도)를 구한 후에 합산하여 계산한다.
2.3 부모 목표가 최소 달성률이 설정되어 있을 때, 1.1단계에서의 방식으로 적응의 필요성을 판단한다.
2.4 부모 목표가 최소 달성률이 설정되어 있지 않을 때, 최소 달성률을 가진 부모 목표를 찾을 때까지 2.1, 2.2 단계를 반복한다. 결국 2.3단계를 수행하게 된다.

3.5 목표 기반의 외부상황문제 관리(Goal-based Contextual Problem Management)

본 연구 기법에서 GCPD와 GCPE의 실행을 위해서는 목표 그래프와 CCG 집합이 중요한 역할을 한다. 목표 그래프와 CCG 집합을 관리함으로써 GCPD와 GCPE의 성능을 확장할 수 있다.

CCG 집합은 유한 집합이기 때문에 모든 잠재적인 문제들을 탐지하는 것은 불가능하다. 따라서 실행 중에도 지속적인 외부상황문제에 대한 관리가 필요하다.

우선 예측하지 못한 외부상황의 발생을 알아내야 한다. 소프트웨어의 내부적인 문제 상황에 대한 기록을 보고 개발자가 이를 파악한다. 시스템 클럭이 t일 때 소프트웨어 내부에 문제가 발생했고, 외부상황문제가 발견되지 않았다고 하자. 예측하지 못한 상황 P_u가 발생했음을 알 수 있다. 클럭이 t일 때 실행하던 행위단위는 A_i이다.

시스템 클럭이 t일 때 P_u가 발생하게 된 원인은 세 가지가 있다. 첫째 원인은 당시에 사용했던 CCG가 정확하지 않았기 때문이다. CCG가 완벽하지 않고 정확하지 않다면 P_u를 탐지하지 못할 것이다. 둘째 원인은 P_u를 탐지하기 위한 CCG가 없기 때문이다. 셋째 원인은 P_u를 분석하기 위한 목표가 없기 때문이다. 이 경우 목표 그래프는 P_u를 설명하지 못하므로 외부상황문제 탐지기 역시 P_u를 탐지해내지 못한다.

예측하지 못한 문제가 발생했을 경우 다음의 프로세스를 수행한다.

표 2 목표 기반의 외부상황문제 관리 프로세스

1.	해당 문제를 설명할 수 있는 목표가 기존의 목표 그래프에 있는지 검사한다.
2.	있을 경우. 해당 목표는 G_j이다. CCG(A_i, G_j)가 있는지 검사한다.
2.1	있다. CCG를 수정하여 예측하지 못한 문제를 탐지할 수 있도록 한다.
2.2	없다. CCG(A_i, G_j)를 정의하여 추가한다.
3.	없을 경우. 해당 문제를 설명하기 위한 목표를 목표 그래프에 추가한다. 유사한 부모 목표를 선택하여 서브 목표로 추가 정의한다. 새롭게 정의한 목표를 Gn라고 하자. 해당 문제 상황을 탐지하기 위해 CCG(Ai, Gn)를 정의한다.

다음 그림 8은 위의 프로세스를 보여준다.

목표 기반의 외부상황 문제는 목표 그래프를 이용하여 정의한다. 이러한 방식은 외부상황문제 관리가 쉽게 만든다. 목표 그래프는 계층 구조를 가지고 있기 때문에 새로운 목표 기반의 문제를 적당한 계층 수준에 추가하기가 쉽다. 그리고 문제 상황을 정확하게 또는 유사하게 나타내는 목표의 검색이 용이하다. 목표 그래프가 의미적인 계층 구조를 가지고 있기 때문이다.

4. 사례 연구

본 연구에서는 사례 연구로서 SMA (Self-managed Meeting Application)를 소개한다. 본 연구 기법을 적용하기 위한 간단한 어플리케이션을 개발하였다. SMA는 미팅 서비스를 지원한다. 클라이언트 프로그램은 다

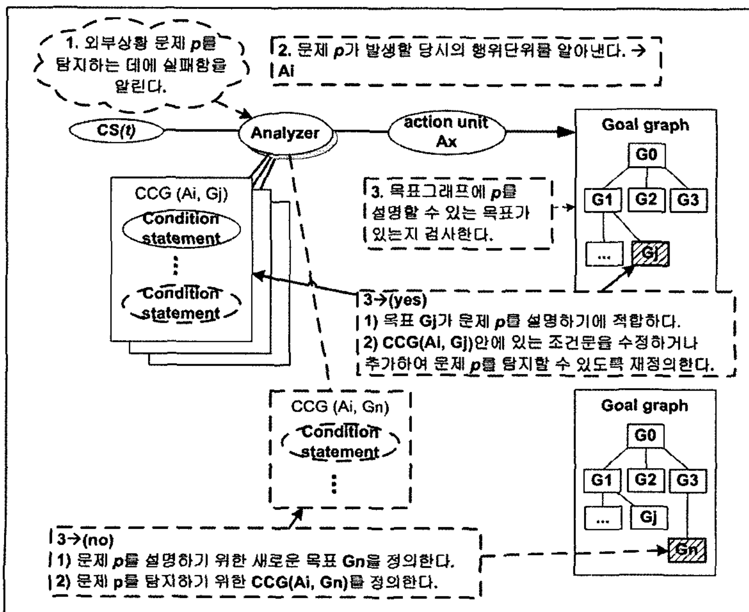


그림 8 목표기반의 외부상황 문제 관리

른 클라이언트에게 연결하여 네트워을 통해서 텍스트 기반의 커뮤니케이션을 이룬다. SMA의 주요 목적은 성공적인 미팅의 수행이다. 미팅의 수행 기간 동안 사용자들은 시스템의 실패(failure), 네트웍 문제 등으로 인해 방해 받지 않아야 한다. 논문에서는 본 연구 기법을 SMA에 적용하여 연구 목표를 기준으로 평가하는 것에 초점을 둔다. 따라서 자세한 구현은 기술하지 않도록 하겠다.

4.1 GCPD와 GCPE 구현

GCPD와 GCPE는 어플리케이션의 구현 모델들을 기반으로 개발된다. 어플리케이션의 개발 중에 발생되는 요구사항 분석 모델, 세부 설계, 구현 코드 등을 이용하여 GCPD, GCPE를 구현한다. 세부적인 개발 절차는 다음 그림 9와 같다.

4.1.1 목표 그래프 설계

어플리케이션의 요구사항 모델을 기반으로 목표 그래프를 설계한다. 목표 그래프는 목표 기반의 요구공학 기법을 이용하여 설계할 수 있다[23]. 그러나 앞서 기술한 바 있듯이 기존의 목표 그래프와는 부분적으로 다른 성질을 지닌다. 목표는 실행과 관련된 성질들로만 기술되어야 한다. 그리고 목표 간의 기여도 값을 설정해야 한다. 아래 그림 10은 SMA를 위한 목표 그래프이다.

목표 그래프는 외부상황문제를 탐지하고 평가하기 위한 기준을 제공한다. 목표 그래프 상에서 목표 간의 기여도 값은 평가 결과에 영향을 미칠 수 있다. 따라서 기여도 값은 여러 번의 테스팅 작업을 통해서 설계되어야 한다.

각 목표의 GPV는 다음 표 3과 같다.

적응 여부를 판단하기 위한 기준을 설계해야 한다. 우선 위의 목표 그래프에서 적응 여부를 결정하는 주요

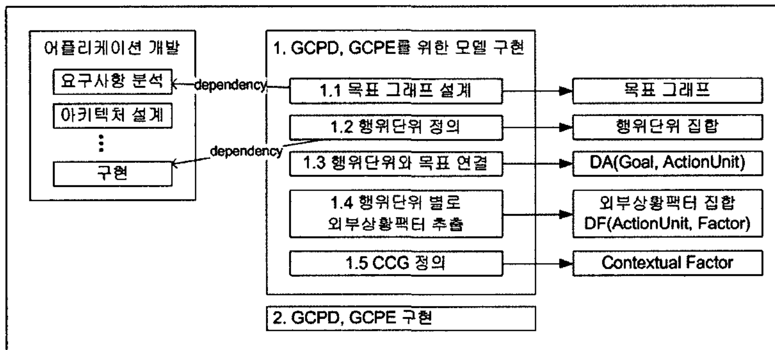


그림 9 GCPD, GCPE 구현 프로세스

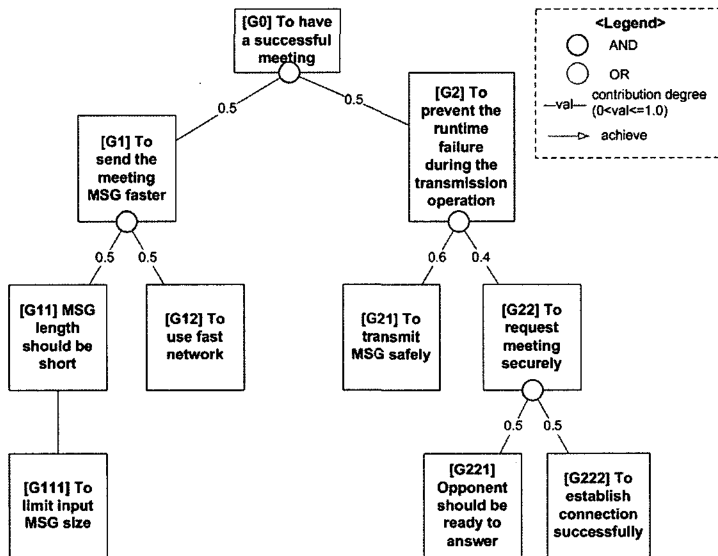


그림 10 SMA의 목표그래프

표 3 SMA의 목표 그래프의 GPV

G0: 1
G1: 0.5
G2: 0.5
G11: 0.25
G12: 0.25
G111: 0.25
G21: 0.03
G22: 0.02
G221: 0.01
G222: 0.01

목표들을 선정한다. 이들 목표들에 대해서 최소 달성률(Minimum Achievement-rate)을 정의한다. 최소 달성률의 설계는 개발자의 경험과 요구사항 모델에 의해서 정의된다. 본 사례 연구에서는 우선 개발자들에 의한 테스팅과 경험에 의한 설정으로 적용하였다. 다음 표 4는 SMA의 최소달성률를 보여준다.

4.1.2 행위 단위 정의

행위 단위를 정의해야 한다. 본 연구에서는 SMA의 구현을 기반으로 클래스의 메서드들로 정의하였다. 매서드들 중에서도 주요 기능을 수행하는 핵심 메서드들로만 추출하여 정의하였다. 다음 표 5는 SMA의 일부 행위 단위들을 보여준다.

행위 단위들 간의 'call', 'precede' 관계는 파싱할 수 있는 도구를 구현하여 찾아낼 수 있다. 그러나 본 사례

표 4 SMA의 최소 달성률

목표	최소 달성률
G0	0.99
G1	1.0
G2	0.98
G111	1.0
G21	0.99
G22	0.95
G221	1.0

표 5 SMA의 행위단위들

ID: 행위단위
…
A2: PartyConnector.connect()
A3: Receiver.run()
A4: Writer.send()
…

연구에서는 행위단위들의 수가 크지 않기 때문에 개발자가 직접 보면서 설정하였다.

4.1.3 행위단위와 목표 연결

행위단위에서 목표로의 'achieve'관계는 기존의 요구공학 기법을 이용하여 설계할 수 있다. 목표 그래프를 정제하는 과정 중에서, 목표와 관련되는 'operation'들을 추출하게 된다[23,25]. 이들 operation들을 근거로 행위단위와 연결할 수 있다. 그 결과 아래 그림 11과 같다.

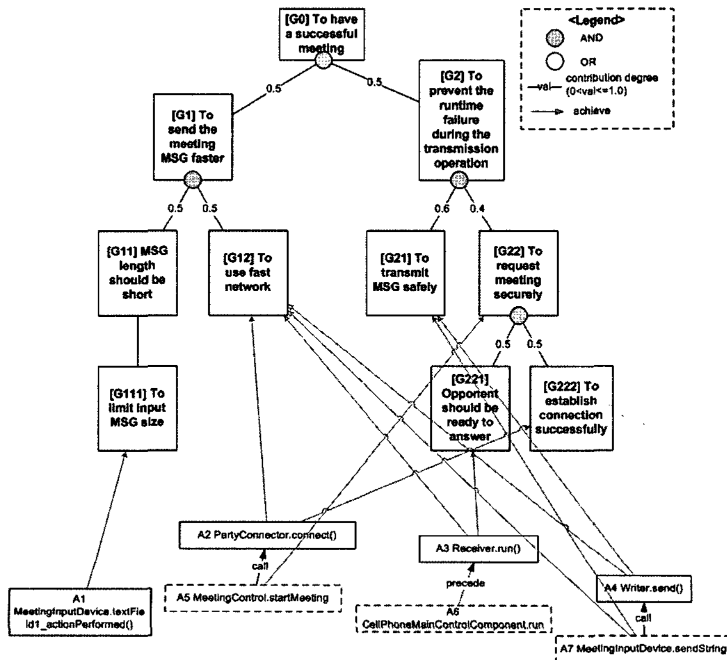


그림 11 SMA의 목표 그래프와 행위단위들간의 연결

앞서 3.1장에서 기술한 바 있듯이 하나의 목표와 연결되어 있는 행위단위들이 각각 다른 크기의 기여도 크기를 지닌다. 이를 모델링하기 위해서 DA(Dependency on Action unit)를 정의한다. 해당 목표를 위해서 관련 행위단위가 얼마나 중요한 역할을 하는지를 정의하는 것이기 때문에 개발자의 경험에 의해 결정된다. 중요하다고 생각되는 행위단위에 그만큼 더 높은 DA값을 가지도록 하게되면, 외부상황문제의 탐지 및 평가에서 해당 행위단위에 의한 문제의 비중이 다른 행위단위에 의한 문제들의 것보다 높게 나타날 것이다.

표 6 SMA의 DA들

```
...
DA(G111, A1) = 1
DA(G12, A2) = 0.35
DA(G12, A3) = 0.2
DA(G12, A4) = 0.35
DA(G12, A7) = 0.1
DA(G21, A4) = 0.8
DA(G21, A7) = 0.2
DA(G22, A5) = 1
DA(G221, A3) = 1
DA(G222, A2) = 1
...
```

4.1.4 행위단위 별로 외부상황 팩터 추출

행위단위들과 관련되는 외부상황팩터들을 추출한다. 행위단위가 실행하는 데에 필요한 팩터들을 추출하면 된다. 팩터들에는 컴퓨팅 자원, 사용자 입력 등이 포함될 수 있다. 각 행위단위들의 팩터들을 추출하여 전체 외부상황팩터들의 집합을 정의하면 {CPU(CPU's available power), MEM(available memory space), MSG (input message's length), NET(network speed)}와 같다. 아래 표는 각 행위단위별 외부상황팩터들과 전체 외부상황팩터 집합을 보여준다.

표 7 SMA의 외부상황팩터들

```
PartyConnector.connect(): CPU, MEM, NET
Receiver.run(): CPU, MEM, NET
...
C-Factot Set = {CPU, MEM, MSG, NET}
```

3.1장에서 기술한 바 있듯이 각 외부상황팩터는 행위단위에게 각각 다른 크기의 영향력을 지닌다. 이를 모델링하기 위해서 DF(Dependency on C-Factor)를 정의한다. 이것 역시 앞서 정의한 DA의 경우와 같은 방식으로 설정된다. 개발자가 어떻게 DF를 정의하느냐에 따라서 외부상황문제의 측정의 성능이 달라질 수 있다.

표 8 SMA의 DF들

```
...
DF(A3, cpu) = 0.45
DF(A3, mem) = 0.15
DF(A3, msg) = 0
DF(A3, net) = 0.4
DF(A2, cpu) = 0.3
DF(A2, mem) = 0.2
DF(A2, msg) = 0
DF(A2, net) = 0.5
...
```

4.1.5 CCG 정의

각 행위단위가 연결된 목표를 달성하기 위해서 필요한 CCG를 정의한다. 개발자들은 어플리케이션의 실행환경 요구사항을 기반으로 정의할 수 있다. 아래 그림 12는 SMA의 CCG들의 일부를 보여준다.

4.1.6 GCPD, GCPE 구현

위에서 구현한 모델들을 기반으로 그림 7의 프로세스에 따라서 GCPD를 구현할 수 있다. 그리고 GCPE는 GCPD의 결과를 입력으로 표 1의 프로세스를 구현하면

```
CCG(A1, G111)

CPU:
If (0 <= v < 3)          PL = INF
If (3 <= v < 10)         PL = 100
If (10 < =v < 100)       PL = 0

MEM:
If (0 <= v < 0.1)        PL = INF
If (0.1 <= v < 0.5)      PL = 100
If (0.5<= v< 1)          PL = 50
If (1 <= v < 10)         PL = 0

MSG:
If (v >= 100)            PL = INF
If (50 <= v < 100)       PL = 100
If (0 <= v < 50)         PL = 0

NET:
```

```
CCG(A2, G12)

CPU:
If (0 <= v < 10)         PL = INF
If (10 <= v < 30)        PL = 100
If (30 <= v < 100)       PL = 0

MEM:
If (0 <= v < 0.1)        PL = INF
If (0.1 <= v < 0.2)      PL = 100
If (0.2 <= v < 1)        PL = 50
If (1 <=  v < 10)        PL = 0

MSG:

NET:
If (0 <= v < 5)          PL = INF
If (5 <= v < 10)         PL = 100
If (10 <= v < 50)        PL = 50
If (50 <= v)             PL = 0
```

```
CCG(A3, G12)

CPU:
If (0 <= v < 10)         PL = INF
If (10 <= v < 30)        PL = 100
If (30 <= v < 100)       PL = 0

MEM:
If (0 <= v < 0.1)        PL = INF
If (0.1 <= v < 0.2)      PL = 100
If (0.2 <= v < 1)        PL = 50
If (1 <= v < 10)         PL = 0

MSG:

NET:
If (0 <= v < 5)          PL = INF
If (5 <= v < 10)         PL = 100
If (10 <= v < 50)        PL = 50
If (50 <= v)             PL = 0
```

```
CCG(A3, G221)

CPU:
If (0 <= v < 10)         PL = INF
If (10 <= v < 30)        PL = 100
If (30 <= v <= 100)      PL = 0

MEM:
If (0 <= v < 0.1)        PL = INF
If (0.1 <= v < 0.2)      PL = 100
If (0.2 <= v < 1)        PL = 50
If (1 <= v < 10)         PL = 0

MSG:

NET:
If (0 <= v < 5)          PL = INF
If (5 <= v < 10)         PL = 100
If (10 <= v)             PL = 0
```

그림 12 SMA의 CCG들 중 일부

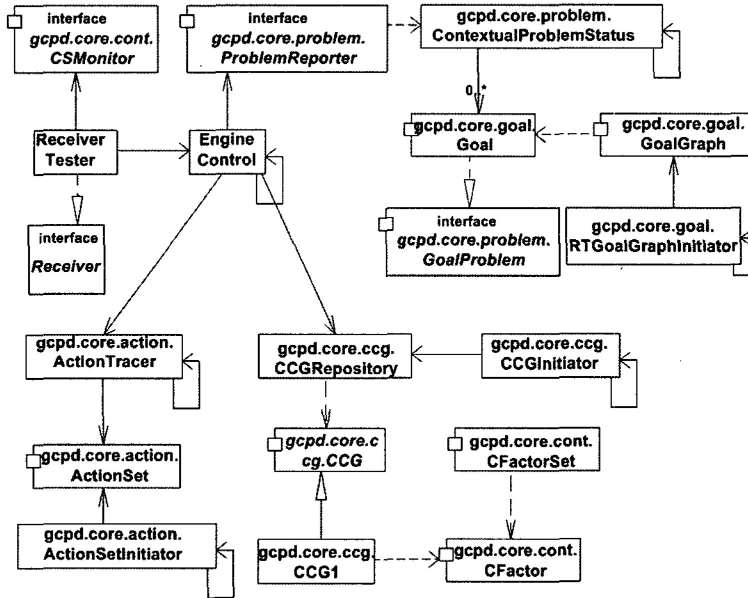


그림 13 GCPD의 구조

된다. 아래 그림 13은 GCPD의 구조를 보여준다.

성능과 관련된 다양한 이슈들이 있을 수 있다. 예를 들어, 어플리케이션 프로세스와 별도로 외부상황문제 탐지기 프로세스가 실행되어야 하는 문제 등이 있을 수 있다. 본 논문에서는 구현과 관련된 자세한 내용을 기술하지 않도록 하겠다.

4.2 외부상황 문제의 탐지 및 평가 사례

본 장에서는 GCPD와 GCPE의 동작을 보이고자 한다. 우선 본 실험을 위한 테스트를 위한 외부상황 데이타를 준비한다. 테스트 데이타는 개발자가 관찰한 값을 바탕으로 만들어진다. SMA가 실행 중에 문제가 될 수 있는 상황들이 일부 포함될 수 있다. 다음 표 9는 테스트 데이타의 일부를 보여준다.

표 9 외부상황 테스트 데이타

CSi: net(Network Speed) msg(Message Input's length) cpu(CPU's available power) mem(Memory's available space)
CS1: net=10.0 msg=77.0 cpu=88.0 mem=4.0
CS2: net=60.0 msg=90.0 cpu=50.0 mem=10.0
CS3: net=60.0 msg=300.0 cpu=25.0 mem=0.5
CS4: net=40.0 msg=400.0 cpu=15.0 mem=0.1
CS5: net=5.0 msg=10.0 cpu=90.0 mem=7.0
CS6: net=60.0 msg=77.0 cpu=88.0 mem=8.0
CS7: net=61.0 msg=23.0 cpu=48.0 mem=8.0
CS8: net=60.0 msg=115.0 cpu=28.0 mem=4.0
CS9: net=50.0 msg=90.0 cpu=30.0 mem=8.0
...
...

SMA와 동시에 GCPD, GCPE를 실행시킨다. SMA가 실행하는 동안 GCPD는 외부상황 값들을 검사하고 외부상황문제를 찾아낸다. GCPE는 GCPD의 결과를 기반으로 평가를 수행하여 적응 여부를 판단한다.

SMA의 사용자는 다른 사용자에게 연결하여 미팅을 수행하게 된다. 사용자가 SMA를 이용하면서 행위단위들이 실행된다.

4.2.1 외부상황문제 탐지 및 평가 사례

준비된 상황데이타에 따르는 시뮬레이션 실행 환경에서 SMA의 GCPD는 다음과 같은 결과를 출력한다. 아래 표 10상에 나타나는 결과는 GCPE에게 객체 모델의 형태로 입력된다.

외부상황 인스턴스가 CS5(표 9 참조)이고 SMA는 행위단위 A6(그림 11 참조)를 실행하고 있다고 하자. 행위단위 A6는 행위단위 A3에 'precede'하는 관계이기 때문에 GCPD는 외부상황 문제를 예측하기 위해서 행위단위 A3와 관련되는 CCG들을 이용해서 외부상황을 검사한다. GCPD는 CCGRepository에서 CCG(A3, G12)와 CCG(A3, G221)을 찾아낸다. 아래 그림 14는 이를 개요적으로 보여준다.

CS5는 CCG(A3, G12)(그림 12 참조) 안에 있는 NET 외부상황 팩터의 조건에 의해 CPL이 0.08이다. CS5의 NET 팩터가 5이고 CCG(A3, G12) 안의 NET 팩터의 조건문, "if (5 <= v < 10)"을 만족하기 때문에 Problem Level은 100%이다. 여기에 DF와 DA값을 곱하여 다음과 같이 구할 수 있다.

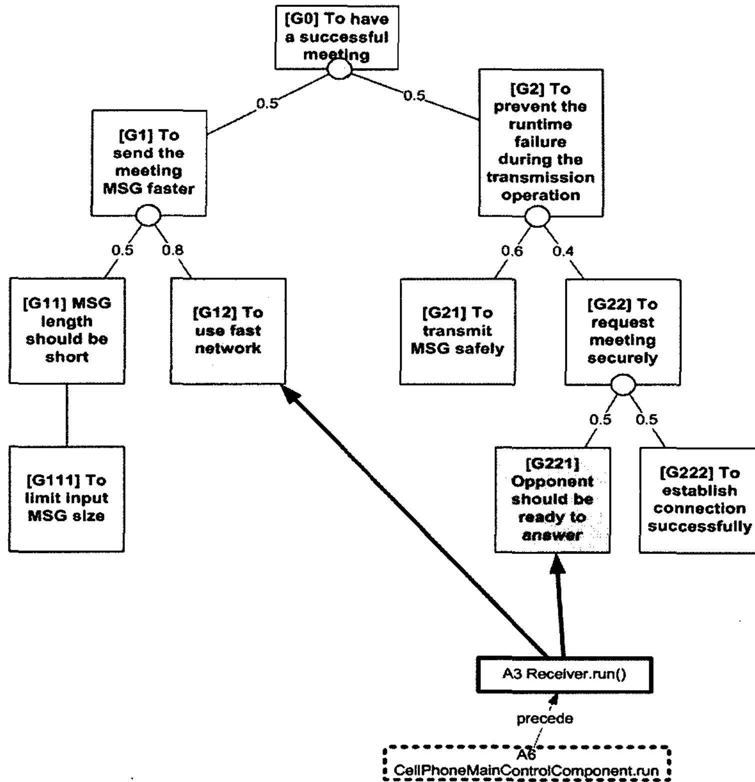


그림 14 목표 G12, G221에서의 목표기반의 외부상황문제 발생

표 10 SMA의 GCPD의 실행결과 중 일부

```
...
...
SWClock sends DETECTION_REQUEST to Receiver_8
=========================================
===
*CURRENT CS: net=60.0 msg=115.0 cpu=28.0 mem=4.0
=========================================
===
CCG3
CCG6
====================PROBLEM_REPORT============
========(start)
*Current # of problems=2
[221]Opponent is ready to answer
priority=0.1
#>>
problemLevel=33.33333333333333
####################################>>
[12]Use fast network
priority=0.4
####>>
problemLevel=33.33333333333333
####################################>>
====================PROBLEM_REPORT============
========(end)
SWClock sends DETECTION_REQUEST to Receiver_9
=========================================
===
*CURRENT CS: net=50.0 msg=90.0 cpu=30.0 mem=8.0
=========================================
===
CCG3
CCG6
====================PROBLEM_REPORT============
========(start)
*Current # of problems=0
====================PROBLEM_REPORT============
========(end)
...
...
```

$$CPL = NET'sPL * DF(A3, NET)$$
$$* DA(G12, A3) = 100\% * 0.4 * 0.2 = 0.08$$

따라서 외부상황문제로 인해 G12의 0.08만큼이 달성되는데에 어려울 수 있다고 해석할 수 있다. 그리고 발생된 문제의 전체 시스템에서의 비중은 0.25(G12의 GPV)이다.

CCG(A3, G221)에서도 NET 외부상황팩터의 조건에 의해서 문제가 발생된다. CPL을 구하면 다음과 같다.

$$CPL = NET'sPL * DF(A3, NET)$$
$$* DA(G221, A3) = 100\% * 0.4 * 1 = 0.4$$

위에서의 G12에서의 CPL에 비해서 더 높은 이유는 행위단위 A3의 목표 G221에서의 기여도 비율(DA)이 더 높기 때문이다. 본 외부상황문제의 전체 시스템에서의 비중은 0.01(G221의 GPV)이다.

본 사례에서 특정 외부상황으로 인해서 하나 이상의 목표에서 문제가 발생될 수 있음을 알 수 있다. 이 때 GPV와 CPL을 고려하고 관련 문제 목표의 최소달성률을 기반으로 문제를 평가해야 한다.

이제 위의 결과를 GCPE가 평가를 수행한다. 평가의 기준은 G12와 G221의 최소달성률이다. G12의 경우에는

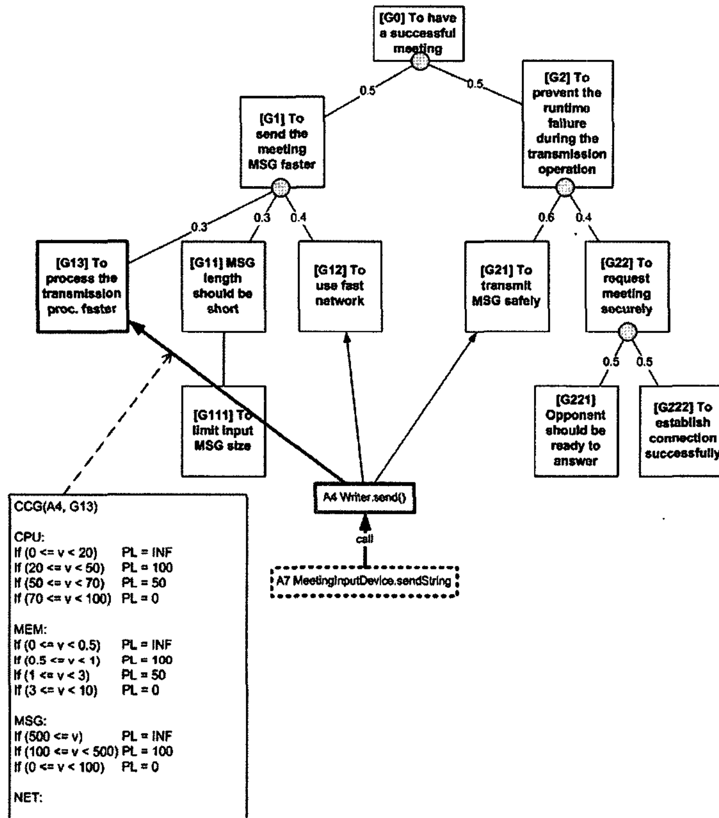


그림 15 외부상황문제를 탐지하는 능력을 확장하기 위한 목표의 추가

최소달성률이 없으나 부모목표의 것을 이용하여 구할 수 있다. G12와 G221의 최소달성률은 모두 1.0이다. G12와 G221의 달성률은 각각 0.92, 0.6이다. 따라서 최소달성률을 넘지 않았으므로 적응해야한다고 판단한다.

판단 이후에 G12와 G221의 문제를 해결하기 위한 적응전략을 선택하면 된다. 두 목표 간의 관계가 충돌관계일 경우에는 비중이 높은 목표를 선택해서 전략을 선택하는 것이 좋다. 그러나 위의 경우에는 두 목표간의 관계가 충돌이 아니기 때문에 동시에 해결할 수 있는 전략을 세우는 것이 가능하다.

4.3 외부상황 문제 관리

앞 장에서의 외부상황 평가 사례는 이미 정의되어 있는 외부상황문제일 경우에 가능하다. 그러나 소프트웨어가 실행 중에 예측하지 못한 상황이 발생될 수 있다. 이를 위해 본 연구에서는 외부상황문제를 추가하기 위한 기법을 제공한다.

외부상황 문제 관리는 예측하지 못한 문제의 발견으로부터 시작된다. 문제상황을 두가지 경우로 가정하여 기술하고자 한다. 각 경우에서의 가정사항은 다음과 같다.

- 첫번째 경우에는 실행 중인 행위단위가 A4이다.
- 두번째 경우에는 실행 중인 행위단위가 A3이다.

표 2의 관리프로세스에 따라 적용하면 다음과 같다.

표 11 외부상황문제 관리의 두가지 사례

1. 해당 문제를 설명할 수 있는 목표가 기존의 목표 그래프에 있는지 검사한다.
2. 있을 경우. 해당 목표는 G12이다. CCG(A3, G12)가 있는지 검사한다.
2.1 있다. CCG를 수정하여 예측하지 못한 문제를 탐지할 수 있도록 한다. **(두번째 경우)**
2.2 없다. CCG(A3, G12)를 정의하여 추가한다.
3. 없을 경우. 해당 문제를 설명하기 위한 목표를 목표 그래프에 추가한다. 유사한 부모 목표로 G1을 선택하여 서브목표로 추가 정의한다. 새롭게 정의한 목표를 G13이라고 하자. 해당 문제 상황을 탐지하기 위해 CCG(A4, G13)를 정의한다. **(첫번째 경우)**

첫번째 경우는 목표 그래프에 새로운 목표를 추가하여 문제를 탐지할 수 있는 관점을 넓히는 것이다. 목표를 지속적으로 추가함으로써 소프트웨어는 보다 다양한 외부상황문제들을 탐지할 수 있다.

두번 째 경우는 기존의 CCG를 수정하는 것이다. 외부상황 인스턴스가 (net=15.0, msg=10.0, cpu=90.0, mem=7.0)이고 실행하고 있는 행위단위가 A3이라고 하자. GCPD에 의해서 G12에서 문제가 발생되었다고 탐지된다. NET 외부상황 팩터의 Problem Level이 50이다. 그러나 실제 실행중인 소프트웨어에게 문제가 더 심각하여 problem level이 100이어야 한다고 판단되었다고 하자. 이 경우 기존의 CCG(A3, G12)를 수정해야 한다. CCG(A3, G12)의 수정으로 인해서 GCPD는 외부상황 문제를 보다 정확하게 탐지할 수 있다. 즉 탐지되는 Problem Level이 보다 정제되는 결과를 낳는다. 수정한 결과는 다음 그림 16과 같다.

CCG(A3, G12)	
CPU:	
If (0 <= v < 10)	PL = INF
If (10 <= v < 30)	PL = 100
If (30 <= v < 100)	PL = 0
MEM:	
If (0 <= v < 0.1)	PL = INF
If (0.1 <= v < 0.2)	PL = 100
If (0.2 <= v < 1)	PL = 50
If (1 <= v < 10)	PL = 0
MSG:	
NET:	
~~If (0 <= v < 5)~~	~~PL = INF~~
~~If (5 <= v < 10)~~	~~PL = 100~~
~~If (10 <= v < 50)~~	~~PL = 50~~
~~If (50 <= v)~~	~~PL = 0~~
If (0 <= v < 5)	PL = INF
If (5 <= v < 20)	PL = 100
If (20 <= v < 50)	PL = 50
If (50 <= v)	PL = 0

그림 16 CCG(A3, G12)의 수정

5. 사례연구 결과 평가

5.1 외부상황 문제의 추상화 기법

본 연구에서는 외부상황 문제를 모델링하기 위한 도구를 지원한다. 목표를 기반으로 외부상황 문제를 추상화하는 기법을 제안하였다. 이로 인해 얻을 수 있는 이점은 다음과 같다.

a. 외부상황 문제의 복잡성 해결: 외부상황 문제를 모델링하는 기법을 지원함으로 인해서 외부상황 문제의 복잡성을 해결할 수 있다. 소프트웨어의 외부 환경이 점차 복잡해지고 있기 때문에 외부상황 문제를 단순히 제약사항의 묶음으로 표현하는 것에는 한계가 있다. 이를 소프트웨어의 상위 레벨의 문제 단위인 목표로 연결하여 모델링함으로써 복잡한 외부상황 문제의 범위, 심각성의 정도, 문제들간의 관계 등을 표현하는 것이 가능해졌다. 이를 통해서 외부상황 문제의 복잡성을 해결할 수 있는 것이다.

b. 외부상황 문제 영역의 확장성 지원: 본 연구에서 외부상황 문제는 목표 기반으로 캡슐화하여 표현 가능하다. 따라서 목표 그래프의 확장을 통해서 평가 가능한 외부상황 문제의 영역을 확장할 수 있다. 즉 목표 그래프에서의 각 목표가 외부상황 문제를 해석하기 위한 기준을 제공하기 때문에 목표를 보다 세부적으로 서브목표로 나누거나 추가를 하는 방식으로 외부상황 문제의 영역이 확장되는 것이다.

c. 아키텍처 상의 문제 진단의 용이함: 목표 그래프와 행위단위와의 연결을 통해서 목표 그래프상의 문제를 바탕으로 아키텍처 상의 문제 진단이 용이하게 된다. 행위단위는 아키텍처의 구성(configuration)에 casually-connected된다. 따라서 시스템의 전체 목표 그래프상에서의 문제를 행위단위로 파악하여, 결국 아키텍처 상의 구성과 연결되어 진단이 가능해지는 것이다.

5.2 외부상황 문제의 목표기반 표현으로 인한 적응전략 계획의 용이성

제약 조건과 외부상황 값을 비교하여 외부상황 문제를 발견하는 것은 새로운 기술이 아니다. 본 연구에서 가장 중요한 것은 목표 모델을 기반으로 정의된 외부상황조건을 이용하여 문제를 발견한다는 것이다.

목표기반으로 외부상황문제를 표현하는 것은 자기적응형 소프트웨어에게 큰 장점을 제공한다. 자기적응형 소프트웨어의 계획자(Planner)는 발생된 외부상황문제를 해결하기 위한 아키텍처 기반의 계획을 선택한다. 아키텍처는 목표 기반으로 설계될 수 있기 때문에[24], 문제에 따라서 적절한 아키텍처 기반 전략을 선택하는 것이 용이하다. 계획자는 발생된 문제의 목표의 우선순위, 목표 그래프안에서의 역할을 고려하여 적절한 적응 계획을 선택한다.

목표기반의 외부상황 문제의 평가와 적응전략 계획간의 관계를 개요적으로 보이면 다음과 같다.

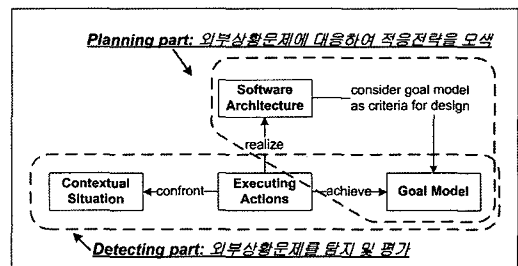


그림 17 목표기반의 외부상황문제 평가 및 적응전략의 계획

5.3 문제 발생의 예측 가능

본 연구 기법에서는 향후에 발생가능한 외부상황문제

를 탐지하는 것이 가능하다. 본 연구기법에서 소프트웨어의 실행 중에 지속적으로 행위단위를 추적하고 향후 실행 가능한 행위단위를 검색이 가능하도록 하였다. 소프트웨어가 실행중인 행위단위를 추적을 하고, 행위단위를 기준으로 외부상황문제를 탐지하기 때문에 향후 문제 발생 가능성을 탐지하는 것이 가능한 것이다. 미리 탐지하는 것은 외부상황문제에 대해서 보다 적극적인 대처를 할 수 있다는 장점을 지닌다.

완벽한 예측을 위해서는 소프트웨어의 모든 실행단위들, 이들간의 선후 관계를 모델링해야 한다. 그러나 중요한 행위단위들만 모델링함으로써 효과적으로 성능을 높일 수 있다. 행위단위들의 정의는 어플리케이션 개발자에게 맡겨진다.

5.4 외부상황 문제 탐지 및 평가 성능의 확장성

소프트웨어의 실행 중에도 지속적으로 외부상황문제의 탐지 및 평가 성능을 확장할 수 있다. 목표 그래프를 이용하여 외부상황문제를 추상하였기 때문에 문제를 평가하기 위한 CCG나 목표의 관리가 용이하기 때문이다. 목표 그래프는 계층 구조를 가지고 있기 때문에 새로운 목표 기반의 문제를 적당한 계층 수준에 추가하기가 쉽다. 그리고 문제 상황을 정확하게 또는 유사하게 나타내는 목표의 검색이 용이하다. 목표 그래프가 의미적인 계층 구조를 가지고 있기 때문이다.

외부상황 문제의 탐지 기법과 관련된 기존 연구에서는 문제의 추가 프로세스에 대해서는 중요하게 다루고 있지 않다. 그러나 실제로 소프트웨어의 실행 중에는 예측하지 못한 상황들이 발생되므로 이러한 기능은 중요한 것이다.

6. 결 론

본 연구에서는 GECS(Goal-based Evaluation of Contextual Situation)를 제안하였다. 다음과 같은 장점을 지닌다.

- 외부상황 문제의 추상화
- 외부상황 문제의 목표기반 표현으로 인한 적응전략 계획의 용이성
- 문제 발생의 예측
- 외부상황 문제의 탐지 및 평가 성능의 확장성

본 연구가 실제 어플리케이션에 적용하기 위해서는 여러 가지 문제들이 해결되어야 한다. 가장 큰 어려운 점은 DF, DA 등의 수치 값들의 설정이다. 이는 개발자들의 경험뿐만 아니라 지속적인 튜닝(tuning)이 이루어져야 하는 부분으로 기계학습(machine learning)을 이용하여 지속적으로 학습이 이루어져야만 효과적인 평가를 이룰 수 있다. 이를 위한 모델링 도구 및 기계학습을 이용한 연구를 향후 연구에서 다룰 예정이다.

본 연구에서 외부상황 문제의 탐지 및 평가는 목표 그래프를 기반으로 이루어진다. 목표 그래프 안에서는 유사한 문제들로 묶을 수 있다. 이러한 유사성은 문제 평가나 적응 전략의 선택시 효과적으로 사용할 수 있다. 향후 연구에서는 목표 그래프를 기반으로 외부상황 문제의 의미 거리(semantic distance)를 적용한 평가 기법을 연구할 예정이다.

GECS를 실제 어플리케이션에 적용하기에는 여러 가지 복잡한 절차들이 필요하다. 이를 간단화 할 수 있는 미들웨어를 구현할 예정이다. 관련 모델들의 기술 언어를 제안하고 이를 위한 파서, 컴파일러 등이 포함된다.

참 고 문 헌

[1] "Self adaptive software," December, 1997. DARPA, BAA 98-12, Proposer Information Pamphlet, www.darpa.mil/ito/Solicitations/PIP_9812.html

[2] Robert Laddaga, "Active Software," In Robert Laddaga Paul Robertson and Howard E. Shrobe, editors, Self-Adaptive Software. Springer-Verlag, 2000.

[3] Robert Laddaga and Paul Robertson, "Model Based Diagnosis in Self Adaptive Software," Proceedings of the 14th International Workshop on Principles of Diagnosis, 2003.

[4] David Garlan, and Bradley Schmerl, "Model-based Adaptation for Self-Healing Systems," ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02), November 18-19, 2002.

[5] Robert Laddaga, "Creating robust software through self-adaptation," IEEE Intelligent Systems, May/June 1999, pp. 26-29, 1999.

[6] Alex C. Meng, "On evaluating self-adaptive software," Proceedings of the first international workshop on Self-adaptive software, pp.65-74, April 2000.

[7] D. Garlan, B. Schmerl, and J. Chang, "Using gauges for architecture-based monitoring and adaptation," In Proceeding of the Working Conference on Complex and Dynamic Systems Architecture, Dec. 2001.

[8] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software," IEEE Intelligent Systems(vol. 14, no. 3), pp. 54-62. May/June 1999.

[9] Howard E. Shrobe, "Model-Based Diagnosis for Information Survivability," IWSAS 2001, pp. 142-157, 2001.

[10] Sandeep Neema, Ákos Lédeczi, "Constraint-Guided Self-adaptation," IWSAS 2001, pp. 39-51, 2001.

[11] Shang-Wen Cheng, An-Cheng Huang, David

Garlan, Bradley Schmerl, and Peter Steenkiste, "Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure," IEEE Computer Vol. 37 Num. 10, October 2004.

[12] Jamieson M. Cobleigh, Leon J. Osterweil, Alexander Wise, Barbara Staudt Lerner, "Containment units: a hierarchically composable architecture for adaptive systems," SIGSOFT FSE 2002, pp. 159-165, 2002.

[13] L. J. Osterweil, A. Wise, J. M. Cobleigh, L. A. Clarke, and B. S. Lerner, "Architecting dynamic systems using containment units," In Proceedings of the Working Conference on Complex and Dynamic Systems Architecture, Dec. 2001.

[14] Tim Kindberg , Armando Fox, "System Software for Ubiquitous Computing," IEEE Pervasive Computing, v.1 n.1, pp.70-81, January 2002.

[15] L. Bass, P. Clements and R. Kazman, "Software Architecture in Practice," Addison-Wesley, 1997.

[16] Christine Hofmeister Robert Nord Dilip Soni, "Applied Software Architecture," Addison Wesley, 2000.

[17] BERNON Carole, GLEIZES Marie-Pierre, PEYRUQUEOU Sylvain, PICARD Gauthier, "ADELFE, a Methodology for Adaptive Multi-Agent Systems Engineering," Third International Workshop "Engineering Societies in the Agents 21," (ESAW-2002), 16-17 September 2002.

[18] Dey, A., Abowd, G, "Towards a Better Understanding of Context and Context-Awareness," GVU Technical Report GIT-GVU-00-18, Graphics, Visualization and Usability Center, Georgia Institute of Technology, 1999.

[19] Bill N. Schilit, Norman Adams and Roy Want, "Context-Aware Computing Application," IEEE Workshop on Mobile Computing System and Application, December 1994.

[20] S. S. Yau and F. Karim, "A Context-Sensitive Middleware-based Approach to Dynamically Integrating Mobile Devices into Computational Infrastructures," Journal of Parallel and Distributed Computing, vol. 64(2), February 2004, pp. 301-317, 2004.

[21] Michael Jackson, "The 21 and the Machine," Proceedings of the 17th international conference on Software engineering, pp. 283-292, 1995.

[22] Jonathan Lee, Kuo-Hsun Hsum, "Modeling software architectures with goals in virtual university environment," Information & Software Technology 44(6), pp. 361-380, 2002.

[23] Axel van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," RE 2001, pp. 249-261, 2001.

[24] van Lamsweerde, A., "From System Goals to Software Architecture," SFM 2003, pp. 25-43, 2003.

[25] Hermann Kaindl, "A design process based on a model combining scenarios with goals and functions," IEEE Transactions on Systems, Man, and Cybernetics, Part A 30(5), pp. 537-551, 2000.

[26] Rolland, C., C. Souveyet, and C. Ben Achour, "Guiding Goal Modeling Using Scenarios," IEEE Trnansactions on Software Engineering, 1998. 24(12), pp. 1055-1071, 1998.

[27] D. Bartetzko, C. Fischer, M. M¨oller, and H.Wehrheim, "Jass - java with assertions," In K. Havelund and G. Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 55. Elsevier, 2001.

[28] Meyer, B., "Object-Oriented Software Construction," ISE, 2nd edition, 1997.

[29] de Kleer J. and Kurien, J., "Fundamentals of Model-based Diagnosis," In Proceedings of the Fourteenth International Workshop on Principles of Diagnosis, DX'03, June 2003. pp. 1 - 12, 2003.

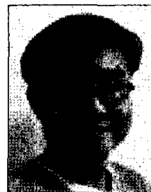
[30] M. M. Kande, "A Concern-oriented Approach to Software Architecture," Thesis 2796, 2003. EPFL, Lausanne, Switzeland. http://adhoc.dpfl.ch/EPFL/theses/2003/2796/ EPFL_TH2796.pdf.

[31] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[32] Lars Braubach, et al. "Goal Representation for BDI Agent Systems," R.H. Bordini et al. (Eds.): PROMAS 2004, LNAI 3346, pp. 44-65, 2005.

김 재 선

1999년 2월 서강대학교 전자계산학(공학사). 2002년 8월 서강대학교 전자계산학(석사). 2002년 9월~현재 서강대학교 박사과정. 관심분야는 소프트웨어 아키텍처, 유비쿼터스 컴퓨팅, 자기적응형 소프트웨어, 임베디드 소프트웨어 공학



박 수 용

1986년 2월 서강대학교 전자계산학(공학사). 1988년 5월 Florida State Univ. Computer & Information Science(석사). 1995년 5월 George Mason Univ. Information Technology(박사). 1995년 5월~1995년 8월 George Mason Univ. Research Assistant Professor. 1996년 1월~1998년 2월 TRW ISC Senior Engineer. 1998년 3월~현재 서강대학교 컴퓨터학과 부교수. 관심분야는 요구공학, 소프트웨어 아키텍처, 소프트웨어 프로덕트라인, 자기 적응형 소프트웨어