2014년 4월 전자공학회 논문지 제 51권 제 4 호
Journal of The Institute of Electronics and Information Engineers Vol. 51, NO. 4, April 2014 http://dx.doi.org/10.5573/ieie.2014.51.4.090

논문 2014-51-4-11

상황인지 시스템의 예외 처리 자동화를 위한 소프트웨어 프레임워크

(A Software Framework for Automatic Exception Handling of Context-aware Systems)

윤 태 섭*, 조 은 선**

(Tae-Seob Yoon and Eun-Sun Cho©)

요 약

상황인지 시스템은 상황 데이터의 변화에 따라 적절한 서비스를 해주는 시스템이다. 하지만 일상에 적용되지 못하는 이유로 예외 발생 가능성을 들 수 있다. 데스크탑 컴퓨팅 프로그램에서는 C++/C#/JAVA등 프로그래밍 언어에서 예외 처리를 지원하지만 상황인지 시스템에서는 다양한 장비들이 연결되어 있어 개발자의 부담이 커지고 처리의 시점이나 위치를 결정하기 어렵기 때문이다. 따라서 본 논문에서는 프로그래머가 예외를 다루는 데에 있어서 다수의 장비를 지속적으로 탐지할 수 있도록 장비의 의미구조와 AspectJ를 이용한 상황인지 시스템의 예외 탐지 및 처리 방법을 제안한다.

Abstract

Context-aware systems provide proper services for the user according to current contexts. However, it is not actively deployed in our daily lives as expected, due to many concerns including occurrences of exceptions. C++/C#/JAVA provide exception handling facilities for desktop computing programs, but context-aware system developers might feel far more difficult to make use of such general facilities, because it is hard to decide the timing and position of exception handling with various devices engaged. In this paper, we propose an exception detection and handling mechanism using device semantics and AspectJ so that it can constantly detect a large number of devices to allow the programmers to detect and handle exceptions with less burden.

Keywords: Context-aware systems, Exception handling, AspectJ, Semantics

Ⅰ. 서 론

상황인지 시스템은 사용자의 명령 없이도 주변 상황에 따라 반응적으로 동작하는 시스템이다. 따라서 다양한 장비(device)가 네트워크로 연결되어 있으며, ECA (Event-Condition-Action) 규칙의 집합으로 프로그램이 구성되는 경우가 많다. 또한 사용자의 개입이 없는 자동화된 컴퓨팅을 통해 사용자 각각에게 상황에 맞는 개인화된 서비스를 제공할 수 있다는 장점이 있다.

이러한 상황인지 시스템은 기술적인 성숙도는 상대적으로 충분하다고 평가되고 있으나 일반인들의 실생활에 보편적으로 사용되기는 아직 이른 분위기이다. 그 대표적인 이유 중 하나로는 예외(exception) 발생에 대한 우려를 들 수 있다. 즉, 기존 데스크탑 컴퓨터에 비

* 학생회원, ** 정회원, 충남대학교 컴퓨터공학과
(Dept. of Computer Sci. & Eng. Chungnam National University)
© Corresponding Author(E-mail: eschough@cnu.ac.kr)
※ 이 논문은 2013년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임 (No. 2010-0013386)
접수일자: 2014년2월7일, 수정일자: 2014년3월19일
수정완료: 2014년4월3일

해 장비와 네트워크 및 복잡한 컴퓨팅 구조를 가지는 상황인지 시스템은 계획된 대로 동작하지 않는 예외적인 상황이 발생될 개연성이 상대적으로 크다. 예외 상황의 결과로는, 만족스럽지 못한 동작이나 거슬리는 동작정도로 그칠 수도 있으나, 때로는 위험한 상황(예를 들어 문을 열고 오븐을 동작시킴으로 인한 과열이나 로봇 청소기의 오동작으로 인한 사용자의 위해)을 초래하는 경우를 배제할 수 없다[1].

전통적으로 장비와 관련된 예외 상황은 프로그래밍 언어에서 제공되는 예외 처리 기능을 사용하여 관리된다. C++, Java, C# 등으로 프로그래밍하는 경우에는 파일 입출력이나 네트워크 접속에서 예외 처리 기능이 자주 사용되는데, 프로그래머가 해당 연산 수행 후 정상적으로 수행되었는지 여부를 명시적 또는 암묵적으로 점검하게 된다. 만일 예외가 발생된 것으로 판명되면, 핸들러(handler)를 정의하고 그것에 따라 처리하게 된다[2].

그러나 이러한 방법을 그대로 상황인지 시스템에 적용하기에는 적당하지 못하다. 첫째, 프로그래머는 유기적으로 연결된 다수의 장비를 모두 동시에 고려하여 예외 상황들과 이에 따른 핸들러들을 작성해야 한다는 부담이 생긴다. 예를 들어 파일을 오픈한 후 문제가 있었는지 플래그나 리턴값을 검사해서 조치를 취하는 전통적인 방법과 달리, 상황인지 시스템에서는 현재 상황이 예외적인지를 알아보기 위해 여러 장비를 동시에 점검해야하고 만일 예외 상황이라면 여러 장비의 상태를 동시에 조정하는 핸들러 코드를 작성할 필요가 있다. 이러한 부담의 문제는, 이미 많은 프로그래머가 일반적인 예외 처리에 대해 소극적이라는 측면을 감안할 때[3], 실제 장비의 작동을 다루는 상황인지 시스템에서는 물리적으로 위험한 상황을 방치하게 되는 우려스러운 결과를 초래할 수 있다.

둘째, 기존의 예외처리 모델에서는 예외 탐지 위치와 예외 처리 위치가 비교적 명확하였지만, 상황인지 시스템에서는 그렇지 못하다. 예를 들어 파일 오픈 등의 명령 수행 후에 점검하는 절차적 패턴이 예외 탐지의 전통적인 방법이었다면, 상황인지 시스템에서는 예외 탐지를 위해 전체 장비의 상태에 대해 계속적으로 주시하다가 적절한 위치에서 핸들러를 수행시켜야한다. 따라서 기존의 Java/C++/C#과 같은 예외 처리 방식과 다른 예외 처리 관련 시스템이 필요하다.

따라서 기존의 상황인지 시스템에서는 이러한 방법보다는 결함 허용(fault tolerant) 기술로 관리하려는 노력이 있어 왔다[4]. 즉 장비의 오류나 네트워크의 단절 등 시스템 전체적으로 결함(fault)이라는 오류상황을 정의하고 해당 상황을 지속적으로 탐지하여 만일 발생이 확인되면 재시도를 하거나 오류 장비를 고립시키는 등의 조치를 취하게 된다. 이 방법은 프로그래머의 개입을 최소화하므로 편리하고 안전한 방법일 수 있다.

그러나 이 방법은 응용 프로그램에 대한 지식이 전혀 없는 상태에서 전체 시스템의 일반적인 오류상황만을 처리하게 되므로, 응용 프로그램의 의미를 고려한 특화된 예외상황(semantic exception[5])에 대해서는 여전히 프로그래머의 예외 처리가 필요로 하게 된다. 그리고 대부분 오류를 개별 장비의 작동여부에 국한하게 되므로, 점점 다변화 하고 기능이 풍부한 장비에 대한 의미적인 예외상황을 위해서는 결함 허용 기법만으로는 한계가 있다.

본 논문에서는 이러한 기존 기법들의 단점을 극복하기 위해 상황인지 예외처리 시스템을 제안한다. 이 방법은 프로그래머가 예외를 다루는 데에 있어서 다수의 장비를 지속적으로 탐지할 수 있게 하여 상황인지 시스템에서의 예외 상황에 대해 보다 편리하게 관리할 수 있게 한다. 모든 응용 프로그램이 공통적으로 지니게 되는 성질은 미리 정의하여 프로그래머의 부담을 덮과 동시에, 장비의 의미구조(semantics)를 고려하여 평면적인 예외 처리에서 벗어나 다양한 장비와의 연결을 허용하는 것을 목표로 하고 있다.

이 때 예외는 정상적이지 않은 상황으로 정의되므로 정상상황에 대한 모델을 정의하는 것이 중요하다. 기존의 상황인지 시스템에서 모델링 방법은 예외를 고려하였으나 시간적 요소를 고려하지 않았거나[5], 시간적인 요소를 고려하였지만 예외를 고려하지 않았기 때문에[6~7], 본 논문에서는 기존의 것들을 참조, 확장하여 새로운 모델을 사용하였다.

본 논문의 구성은 다음과 같다. 먼저 Ⅱ장에서는 상황인지시스템의 모델에 대해 정의하고, Ⅲ장에서는 장비의 상태 행위에 따른 상태전이 모델에 대해 다룬다. Ⅳ장에서는 이를 바탕으로 한 시스템 전체에 대한 의미구조를 소개하고, 예외 상황이 발견되고 처리하는 과정에 대한 모델 및 실제 구현을 제시한다. Ⅴ장에서는 관련 연구 및 토의를 다루고, Ⅵ장에서는 결론을 맺는다.

Ⅱ. 상황 데이터 모델

1. 상황 데이터 모델

일반적으로 상황 데이터 모델은 스마트 환경에서 사용되는 데이터로서. 데이터 소유자, 자주 변화하는 데이터 또는 정적인 데이터와 같은 변함의 정도[8], 컬렉션이나 스칼라 값 등의 구조[8]등을 고려하여 나타내게 된다. 그리고 본 논문에서는 상황 정보가 상호 관계와 관련 데이터 값 뿐 만이 아니라 응용 프로그램의 사용자(User), 공간(Space)과 장비(Device)로 이루어져 있다고 본다. 따라서 이러한 것을 만족하는 시멘틱웹 모델링 도구인 RDF(Resource Description Format)을 활용하여 상황데이터 모델을 정의하는 방법을 취한다. 그러기 위해서 기존 연구[6]에 정의되어 있는 RDF 트리플로서의 상황 데이터 모델을 기반으로 한다. <*subject property value*> 형태인 RDF 트리플에서 *subject*는 *User, Device, Space*나 다른 중간 노드로 이루어지며, *property*는 이들 *subject*가 가지는 각 상황 데이터를 나타낸다. *property*는 변화 정도가 상이하여, 정적인 것과 동적인 것으로 나누어볼 수 있다. 예를 들면 장비의 정적인 *property*로 생산자 이름, 장비 고유의 최대 속도[9] 등의 기본 특성을 들 수 있다.

2. ECA(Event-Condition-Action) 규칙

본 논문에서는 일반적인 상황인지 응용 프로그램에서 사용되는 ECA(Event-Condition-Action) 규칙을 사용한다. 각 이벤트는 RDF 트리플을 통해 접근할 수 있는 정보를 갖고 있다. 이 때 *subject*는 이벤트 자체에 해당하며 속성과 *object*는 정보의 제목과 정보를 나타내는 값 또는 개체에 각각 해당한다[7]. 각 상황 정보는 이벤트 등의 루트 객체로부터 RDF 트리플로 연결되는 링크를 반복적으로 거쳐 도달할 수 있다.

액션은 프로그램의 단위로도 볼 수 있다. 더 이상 쪼개지지 않는 최소의 액션을 단위(unit) 액션이라고 부르며 여기서는 일반적인 프로그램의 명령어 대신 장비에 대한 명령이라고 가정한다. ECA 규칙을 구성하는 액션은 일련의 이러한 단위 액션을 수행하도록 하는 내용을 가지게 된다.

Ⅲ. 장비의 상태 전이 모델

본 논문에서는 개발자의 편의를 위하여, 미리 기술된 정상적인 모델을 벗어났는지를 자동으로 감지하여 이를 예외로 다루는 방법을 사용하고 있다. 따라서 보다 전체적으로 시스템을 기술할 필요가 있으므로, 상황 데이터와 ECA 규칙만으로 구성되는 일반적인 상황모델 대신, 각 장비 모델을 통합한 모델을 사용하게 된다[7]. 상황인지 시스템에서 장비와 관련된 행위와 속성들은 시스템 전체의 동작을 결정하는 데에 핵심적인 요소이기 때문이다.

장비에는 센서(Sensor)와 실행기(Actuator)의 두 종류가 있다. 본 논문에서 센서는 각각 하나의 동적 속성 값을 센싱한다고 가정한다. 기존의 연구들은 센서가 데이터를 수집하는 기초적이고 필수적인 요소라는 측면 때문에, 실행기보다는 센서에만 초점을 맞추는 경우가 많았다. 그러나 본 연구에서는 예외를 발생시키는 주체가 센서보다는 실행기이므로 의미구조를 상황 모델에 결합시키는 방향에 보다 초점을 맞춘다.

실행기는 모든 장비가 갖고 있는 정적 속성과 함께 실행기의 상태 값을 읽는 다중 동적 속성을 함께 갖고 있다. 각 실행기는 외부 명령에 대응하는 고유의 운용 의미구조(operational semantics)를 갖는다고 가정한다. 여기서는 장비의 제조사가 이러한 의미구조를 출시 당시 제공한다고 가정하며, 비슷한 것을 가정한 예로는 집과 환경 네트워크의 간단한 장비 연결을 위한 UPnP(Universal Plug and Play protocol)에서 사용된 상태전이가 대표적이다[10]. 이러한 장비 의미구조는 상태 전이 형태 외에도, 구조적 의미구조(structural operational semantics) 등을 사용하여 기술될 수 있다. 두 경우 모두 각 수행 단계마다 상황데이터를 변화시킨 결과로 장비의 구동 과정을 표현하게 된다. 응용 프로그램에서는 이것을 기반으로 하여 명령과 다른 결과를 낼 때 예외를 정의하게 된다.

주어진 장비의 의미를 상태 전이로 나타낸다면 장비 전이 시스템 $dt \in DT = 2^{SD} \times 2^{CD} \times Tr_D \times S_D \times S_D$는 S_D가 장비의 가능한 상태들의 집합일 때, 각 원소가 <*device_states*, *input_commands*, *transitions*, *start_state*, *current_state*>와 같은 것을 의미한다. *device_states*나 *input_commands* 등에 대해서는 각 장비의 생산자가 장비에 대해 모든 가능한 정상적인 기능에 관한 내용을

표 1. 장비의 명령어
Table 1. Command set of devices.

Device	Command set
Light	{turn_on, turn_off}
Motor	{set $v \mid 0 \le v \le 180$}
Music Player	{play, pause, .. stop}

완벽하게 제공한다고 가정한다. *input_commands*의 한 원소는 장비의 한 입력 명령어로서 상황인지 시스템 전체로 볼 때는 ECA 규칙의 단위 액션에 해당된다.

$tr \in Tr_D = S_D \times C_D \times S_D$은 장비의 한 상태에서 다른 상태로의 전이를 나타낸다. 시작 상태와 현재 상태는 일반적인 상태 전이 시스템에서와 동일한 역할을 한다. 각 *dt*에 대해 함수 *Status*(*dt*)는 튜플 *dt*의 유일한 동적 *property*인 현재 상태를 반환한다. 몇 가지 장비의 명령어의 예는 표 1과 같다*. 예를 들어, 꺼져 있는 자동 조명에 `setTarget`이라는 명령이 전달되어 조명이 켜질 때, 현재 상태는 장비 전이를 따라 turn_off 상태에서 turn_on 상태로 변경된다.

장비는 여러 가지 상태 값을 가질 수 있는데 여기서는 RDF triple에서 장비를 *subject*로 하는 *property*로 기술된다. 장비의 속성이 가질 수 있는 값의 집합은 함수 *domain*을 써서, *domain*($light_{01}$, switch)는 {on, off}, *domain*($motor_{01}$, torque)는 {0, ..., 180}, *domain*($music\_player_{01}$, status)는 {STOP, PLAY, PAUSE} 등으로 나타낼 수 있다.(단, $light_{01}$, $motor_{01}$, $music_player_{01}$는 개별 장비들의 식별자) 함수 *command*: *device* → $2^{command}$는 해당 장비가 수행 가능한 명령어 집합을 반환한다.

S_{device}는 장비의 상태전이를 표현하는 구조적 의미 구조(structural operational semantics[11])를 나타낸다. 본 논문에서 센서는 상태 1개로 구성된 단순한 의미구조를 가진다고 가정하고, 실행기는 입력 명령에 의해서만 동작하는 '수동 동작(passive action)[12]'만이 가능한 장비로 간주하여 모델을 단순화시킨다. 함수 *env* : *device* -> *dynamicProp*(*device*) → *domain*(*device*, *property*)는 장비와 property 이름으로 현재의 값을 얻는 함수이다. *e*가 함수일 때 '*e*[*x*/*p*]'는, 주어진 *e*와 나머지는 동일하나 *p*에 대해서는 기존의 값 대신 *x*로 바꾼 함수를 일컫는다. 각 장비들이 가지는 의미구조는 다음과 같다.

$S_{light01}$[turn_on] (*env* $light_{01}$) = (*env* $light_{01}$)[on/*switch*]
$S_{light01}$[turn_off] (*env* $light_{01}$) = (*env* $light_{01}$)[off/*switch*]
$S_{motor01}$[set v] (*env* $motor_{01}$) = (*env* $motor_{01}$)[*v*/*torque*]
$S_{music\_player01}$[play] (*env* $music\_player_{01}$)
=(*env* $music\_player_{01}$)[PLAYING/*status*]
$S_{music\_player01}$[pause] (*env* $music\_player_{01}$)
=(*env* $music\_player_{01}$)[STOPPED/*status*]
$S_{music\_player01}$[stop] (*env* $music\_player_{01}$)
=(*env* $music\_player_{01}$)[STOPPED/*status*]

Ⅳ. 시스템의 의미론과 예외 상황

1. 시스템의 상태 변화

전통적인 데스크탑 컴퓨팅 시스템과는 달리, 장비를 동반한 상황인지 시스템은 감지와 작동 시간의 지연을 유발하는 경우도 있고 예외상황이 시간적인 특성을 위배하여 생기는 경우가 많다[6]. 따라서 본 모델도 상황인지 시스템의 시간적 특성을 고려한 temporal 모델을 기반으로 다음과 같이 정의한다.

정의 1 ▷: *Actions* × *time* × (*ContextTriples* × (*Actuator* → *DT*)) → *Actions* × *time* × (*ContextTriples* × (*Actuator* → *DT*)) 는 전체 시스템의 ECA 규칙에서 액션의 운용에 대한 의미 구조이며 상황 정보와 시간정보를 사용하여 다음과 같이 정의된다.

$\langle a,\ t,\ \langle rs,\ dt \rangle \rangle \triangleright \langle \varnothing,\ t+\Delta t, \langle rs',\ dt' \rangle \rangle$

i) *a*가 실행기 *ac*의 입력 명령어일 때, 즉 단위 액션일 때, Δt는 *a*가 완료되기까지 소요된 시간이며, *dt*(*ac*)는 〈*d, i, tr, ss, cs*〉이고 *dt'*(*ac*)는 〈*d, i, tr, ss, cs'*〉으로서 장비의 현재 상태만 변화된다. 현재 상황데이터를 나타내는 *rs'*은 *a*의 영향으로 인해 *rs*로부터 변한 상황 데이터이므로 의미구조 S_{device}를 적용한 결과가 나타내는, 변화된 triple 집합이다.

ii) *a*가 $a_1;...a_n$일 때 Δt는 $a_{1,\ ...,\ }a_n$이 모두 완료되기까지 소요된 시간이며, $\langle a_1,\ t,\ \langle rs,\ dt \rangle \rangle \triangleright \langle \varnothing,\ t+\Delta t, \langle rs_1,\ dt_1 \rangle \rangle$, $\langle a_2,\ t,\ \langle rs_1,\ dt_1 \rangle \rangle \triangleright \langle \varnothing,\ t+\Delta t, \langle rs_2,\ dt_2 \rangle \rangle$ ..., $\triangleright \langle a_1,\ t,\ \langle rs_{n-1},\ dt_{n-1} \rangle \rangle \triangleright \langle \varnothing,\ t+\Delta t, \langle rs_n,\ dt_n \rangle$

* UPnP에서 Light는 Binary Lights 타입, SwitchPower 서비스템플릿에 해당한다. 여기서는 UPnP의 setTarget <인자> 대신 직접 turn_on, turn_off을 사용하였다. Music Player는 UPnP의 AVTransport 서비스에 해당하며 여기서는 UPnP의 상태변수 TransportState와 액션 play, pause, stop만 고려한다[10]

〉이 성립하는 경우를 말한다. (비동기적인 액션은 고려하지 않는다.)

$\triangleright^*$ 는 $\triangleright$를 연속적으로 적용한 것이다.

2. 예외(Exceptions) 상황

앞서 언급했듯이 본 논문에서는 개발자의 편의를 위하여, 미리 기술된 정상적인 모델을 벗어났는지를 자동으로 감지하여 이를 예외로 다룬다. 상황인지 시스템에서의 예외는 실행기의 작동 오류 외에도 프로그램이나 사용자 잘못 등 다른 이유로도 유발될 수 있으나, 여기서는 실행기에서 발생된 예외 상황에 초점을 맞춘다.

함수 *Sen*: *ac* → *value* 은 실행기 *ac*를 인자로 받아 실행기의 현재 상태를 반환한다. 여기서는 실행기가 의도한 대로 동작했는지 확인해 줄 수 있는 별도의 센서가 존재한다고 가정하는데, *Sen*은 그 값을 나타낸다. 이러한 가정은 결함허용 시스템에서 널리 쓰이는 중복되는 센서 집합을 도입하는 개념[4]과 유사하다.

장비전이시스템 *dt*의 현재 상태 *CurrentState*(*dt*) 는 직전 상황에서 액션을 수행한 결과에 대한 예측 값이 되며 이것이 실제 값인 *Sen*의 결과와 일치하지 않을 때는 예외로 볼 수 있다. 즉, 현재 상황 데이터를 나타내는 triple 집합이 *rs*이고 장비 *ac*에 대한 현재의 장비전이시스템이 *dt*일 때, *CurrentState*(*dt*), *Sen*(*ac*)가 다를 때를 예외로 감지한다.

3. Atomic 액션

ECA 규칙은 현재 발생된 이벤트가 매치되고 주어진 조건이 만족되면 해당 액션이 수행되는 방식으로 실행된다. 이 때, 현재 발생한 이벤트 집합을 *EventFrom*으로 나타낸다. *EventFrom*은 상황 변화를 대변하는 이벤트의 집합을 표현하기 위해 현재 상황 값과 이전 상황 값의 쌍을 인자로 취하는 함수 형태이다.

발생된 이벤트와 매치되는 ECA 규칙들 중 조건을 만족하는 규칙들에 대해 액션들만 추출한 것은 함수 *ActionsOfECA*로 표현된다. 인자는 이벤트와 상황데이터를 넘기는데, 상황데이터는 조건을 만족하는지 검사하기 위해 사용된다.

함수 *ActuatorStatusExceptionHandler: Actuator* × *DeviceStatus* × *ContextValue* → *Action* 는 적절한 예외 핸들러를 결과로 주는 중요한 함수이다. *CurrentState*(*dt*)와 *rs.get*<*Sen*(*ac*) *value*>가 다른 값이면 함수 *ActuatorStatusExceptionHandler*는 처리 액션을 반환한다. 따라서 현재 관여하고 있는 실행기와 실행기의 현재 상태 및 그 기대값을 인자로 주게 되며, 결과는 실행단위인 액션이 된다. 예외가 발생하지 않는 경우에는 공집합을 반환한다.

시스템의 전체적인 구동 과정은 주어진 시간 t의 상황데이터인 *Context*(t)의 변화로 나타내어질 수 있다.

정의 2 *Context* : $t \rightarrow$ *ContextTriples* × (*Actuator*→*DT*) 는 다음관계식에 의해 귀납적으로 정의된다.

i) 초기 상태 데이터 *Context*(t_0) 는 〈*Initial_RDF_Triples*, { 〈*ac*, *dt*〉 | *ac* ∈ *Actuator*, *dt* ∈ *DT*, *Status*(*dt*) = *fourth*(*dt*)} 〉 이다. 여기서 *dt*의 네 번째 원소는 *start_state*이다.

ii) *event_set* ≔ *EventFrom*(*Context*(t-$\Delta t'$), *Context*(t)) = ∅인 경우는 일치하는 이벤트가 없으므로 *Context*(t)=*Context*(t-$\Delta t'$)이다.

iii) *event_set* ≔ *EventFrom*(*Context*(t-$\Delta t'$), *Context*(t)) ≠ ∅인 경우, 즉 일치하는 이벤트가 있는 경우, <*a, t, Context*(t)> $\triangleright^*$ 〈∅, $t+\Delta t$, *cx*〉 일 때, *Context*($t+\Delta t$)는 *cx* 이다. 단, 여기서 *a*는 {$e \in$ *event_set* | *ActionsOfECA*(*e*, *Context*(t)) } 중 우선순위가 가장 높은 액션이고, Δt는 *a*의 수행 시간이다.

iv) iii)과 같은 경우 *handler* ≔ *ActuatorStatus ExceptionHandler*(*ac*, *CurrentState*(*dt*), *rs.get*<*Sen*(*ac*) *value*>)에 대해, *handler* ≠ ∅ 일 경우, 예외상황이 발생되었다는 의미이므로, 우선순위가 가장 높은 핸들러 $h \in$ *handler*에 대해 그 수행시간이 $\Delta t'$일 때, <h, $t+\Delta t$, *Context*($t+\Delta t$) > $\triangleright^*$ <∅, $t+\Delta t+\Delta t'$, *cx*'> 일 때, *Context*($t+\Delta t+\Delta t'$) = *cx*' 이다.

위 iv)는 h가 null인 경우 핸들러를 수행할 필요가 없으며 h가 null이 아닌 경우에는 처리 액션을 수행하여 상황 데이터가 변하게 되는 것을 의미한다. 후자의 경우 h에 의해 실행기의 현재 상태가 시간 $t+\Delta t$부터 바뀔 수 있다.

예를 들어, 시간 t-Δt에서 아무도 없는 방 $room_{01}$에 존재하는 에어컨 $aircon_{01}$에 대한 정보는 다음과 같이 구해진다.

Prop($aircon_{01}$) = {status, temperature}
domain($aircon_{01}$, status) =
{STOP, COOLING, VENTILATION}

$command(aircon_{01})$ = {chill, vent, stop}

$aircon_{01}$의 장비 상태 전이 시스템이 초기값을 가지고 있다고 할 때, 유일한 상태 변수인 status 중심으로 표현하면 다음과 같다.

<{STOP, COOLING, VENTILATION},
{chill, vent, stop},
{<STOP, chill, COOLING>,
<STOP, vent, VENTILATION>,
<COOLING, vent, VENTILATION>,
<COOLING, stop, STOP>,
<VENTILATION, chill, COOLING>,
<VENTILATION, stop, STOP>},
STOP, **STOP**>

시간이 t-Δt에서 t로 흐른 시점에서 $room_{01}$에 사용자가 들어와 chill 명령을 실행했다고 가정할 때, 장비 전이 시스템의 현재 상태가 자동화된 규칙에 의해 아래와 같이 변한다. Δt는 chill 명령이 전달 및 해석되어 장비가 활성화되는데 소요되는 시간이다. 현재 상태인 STOP을 제외한 다른 정보들은 수행 중 변화되지 않는다.

<{STOP, COOLING, VENTILATION},
{chill, vent, stop},
{<STOP, chill, COOLING>,
<STOP, vent, VENTILATION>,
<COOLING, vent, VENTILATION>,
<COOLING, stop, STOP>,
<VENTILATION, chill, COOLING>,
<VENTILATION, stop, STOP>},
STOP, **COOLING**>

만일 시간 t+Δt에 현재 상태를 읽었을 때, $Sen(ac)$이 COOLING이라면 정상적이므로 *ActuatorStatusException Handler*의 결과인 *handler*는 공집합이다. 그러나 현재상태가 COOLING이 아니고 *handler*가 공집합이 아니라면, chill을 재시도하거나 다른 에어컨을 찾아 chill하는 등의 예외 처리가 수행된다. 처리 액션이 수행되는 동안에는 예외가 발생하지 않는다고 가정한다.

4. 구현

그림 1은 music player, air conditioner 각각에 대한 의미구조 파일들이다. 상태 및 입력명령 등의 내용을 가지며 장비 제조사에서 텍스트 형태로 제공하는 것을 가정하여 구성하였다.

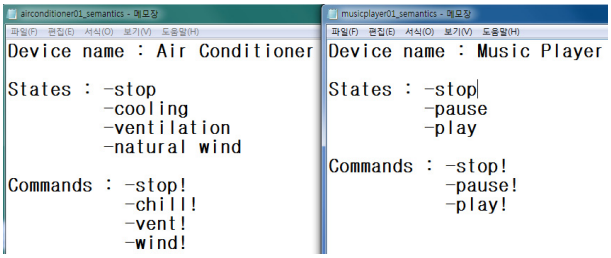

그림 1. 장비의 의미구조 파일들
Fig. 1. Files on Device Semantics.

예외 탐지와 처리에 관한 모듈은 JVM 1.7.0과 AspectJ 2.2.0[14]로 구현되어 현재 프로토타입이 sourceforge에 공개되어 있다[15]. 프로토타입은 위 그림 1의 장비 의미구조를 파싱한 후 텍스트 기반의 장비가 명령어에 대응하는 상태로 변하지 않을 경우 예외로 간주한다. 그 후 정의된 aspect가 예외를 탐지하여 의도했던 상태로 다시 설정하는 구조이다. AspectJ가 지원하는 aspect oriented programming 환경은, 개발자가 원하는 부분에서 직교적(orthogonal)으로 별도의 로직을 실행시킬 수 있는 도구로서, 상황인지 시스템에서도 개발자가 필요한 부분에 자유롭게 예외 탐지를 삽입할 수 있는 장점이 있다. 아래는 Aspect를 정의하는 것으로 pointcut을 메소드 수행 후로 설정한 후 exception을 확인하여 handler를 수행하고 있는 것을 보여준다. 현재 이러한 구현의 프로토타입은 미국 Florida 대의 상황인지 시스템 미들웨어인 ATLAS[13]에서 테스트되었다.

```
package com.pervasa.demo.re.impl.data;

public aspect DeviceAspect {
  pointcut da() : call(
      AtomicAction CheckedAtomicAction
              .exceptionCase(AtomicAction));
      after() returning(Action handler): da()
{
          handler.execute();
}
```

V. 관련 연구 및 논의

앞서 언급했듯이 상황인지 시스템에서 예외 상황에 대한 대책으로 주로 도입되어 왔던 방법은 결함허용 시스템이다[4]. 그러나 의미론적 예외 처리가 불가능하다는 단점으로 인해 개발자의 개입에 대한 필요성이 대두되어 최근 몇 가지 노력들이 있어 왔다[16~17]. 이러한 연구들은 파일 네트워크 이외의 장비에 대한 예외 처리를

구조적으로 할 수 있도록 Java를 확장하거나[16], 의미론적 예외 상황을 기술할 수 있는 기능을 지원하는 방법을 제안하고 있다[8~9, 17].

그러나 이들 대부분은 본 논문과 달리 개발자의 추가적인 오버헤드에 대해 고려하지 않고 있기 때문에, 개발자가 예외 상황에 대한 프로그래밍을 게을리 할 수 있는 가능성이 농후하다[3]. 본 논문의 방법은 장비 명세 및 표준 등에 입각하여 자동적인 예외 처리를 고안함으로써 기존 방법에 비해 개발자 부담을 경감하여, 궁극적으로는 상황인지 시스템의 안전성을 개선할 것으로 기대하고 있다. 최근 연구 중 예외상황에 대한 표현을 단순화시키려는 접근도 있었으나[9], 자주 사용되는 예외 상황에 대해 미리 정의하여 제공하는 것으로서, 본 논문처럼 장비의 명세를 충분히 이용하여 예외 표현 부담을 줄이는 것과는 거리가 있다.

VI. 결 론

본 논문에서는 상황인지 시스템에서 다수의 장비와 연결된 예외상황 처리를 효과적으로 지원해주는 모듈에 대해 소개하였다. 장비 제조사에서 텍스트 형태로 제공하는 장비의 의미구조가 있다고 가정하였다. 장비가 이 의미구조 명세에 일치하게 동작하는지 단위 액션마다 확인하여 일치하지 않을 시 예외가 발생한 것으로 간주하며 AspectJ를 이용하여 명령을 재시도하도록 하였다. 이는 예외 탐지 및 처리를 자동화하여 개발자의 부담을 경감시켜 줄 것으로 예상된다.

향후, UPnP/DLNA를 따르는 장비의 규격에 맞춘 예외 처리 방안과, 예외가 timed automata로 표현된 경우에 대한 처리의 자동화 방안에 대해 연구할 계획이다.

REFERENCES

[1] Yang, Hen-I., and A. Helal. “Safety enhancing mechanisms for pervasive computing systems in intelligent environments.” *Pervasive Computing and Communications*, IEEE, pp. 525-530, 2008.

[2] Oracle, Java Platform Standard Ed. 7, http://docs.oracle.com/javase/7/docs/api/

[3] Shah, “Why do developers neglect exception handling?.” *Exception handling*. ACM, pp. 62-68, 2008.

[4] Mohamed, et al. “A fault Detection and Diagnosis Framework for Ambient Intelligent Systems.” *Ubiquitous Intelligence & Computing.* IEEE, pp. 394-401, 2012.

[5] Rocha, et al. “Towards a formal model to reason about context-aware exception handling.” *Exception Handling*, IEEE, pp. 27-33, 2012.

[6] E. S. Cho, et al. “An Integrated Formal Model for Context-Aware Systems.” *Computer Software and Applications Conference Workshops,* IEEE, pp. 163-168, Kyoto, Japan, Jun. 2013.

[7] E. S. Cho and Y. M. Min, “A Formal Framework for Context-Aware System Modeling”, Journal of the IEEK, Vol. 46, Issue 2, pp. 114-123, Mar. 2009.

[8] E. S, Cho and Sumi Helal. “A situation-based exception detection mechanism for safety in pervasive systems.” *Applications and the Internet (SAINT),* IEEE, pp. 196-201, Munich, Germany, Jul. 2011.

[9] E. S. Cho, J. H. Choi, S. Helal, “Dynamic Parameter Filling for Semantic Exceptions in Context-Aware Systems”, *Ubiquitous Intelligence and Computing,* IEEE, pp. 293-300, Vietri sul Mare, Italy, Dec. 2013.

[10] Larry Buerk, et al. AVTransport:1 Service Template Version 1.01, UPnP Forum, 2002, http://upnp.org/specs/av/UPnP-av-AVTransport-v1-Service.pdf

[11] H. R. Nielson, F. Nielson, Semantics with applications, John Wiley & Sons, pp. 19-50, 1999.

[12] Lee, S. Helal, “From Activity Recognition to Situation Recognition,” *Health and Wellbeing in the Community, and Care at Home*. Springer Berlin Heidelberg, pp. 245-251, 2013.

[13] King, et al. “Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces.” *Local Computer Networks,* IEEE, pp. 630-638, 2006.

[14] The Eclipse Foundation, AspectJ Documentation, http://eclipse.org/aspectj/docs.php

[15] Dice, The exception detection, http://sourceforge.net/projects/theexceptiondetection/

[16] Damasceno, et al. “Context-aware exception handling in mobile agent systems: the MoCA case,” *Software engineering for large-scale multi-agent systems*. ACM, pp. 37-44, 2006.

[17] Kulkarni, et al. “A framework for programming

robust context-aware applications." *Software Engineering*, IEEE Trans. on 36.2, pp. 184-197, 2010.

저 자 소 개

윤 태 섭(학생회원)
2012년 충남대학교 컴퓨터공학과 졸업.
2014년 현재 충남대학교 컴퓨터 공학과 석사과정.
<주관심분야 : 상황 인지 시스템, 예외 처리>

조 은 선(정회원)
전자공학회 논문지
제49권 CI편 제2호 (2012년 3월) 참조