논문 2009-46CI-2-15

상황인식 시스템 모델링을 위한 정형화 프레임워크

(A Formal Framework for Context-Aware System Modeling)

조 은 선*, 민 영 목**

(Eun-Sun Cho and Young-Mok Min)

요 약

상황인식 시스템은 이용자의 주변 상황을 인지해서 적절한 대응을 수행하는 시스템이다. 컴퓨터 외에도 각종 센서나 실행기들이 시스템에 연결되어 동작하게 되므로 전통적인 시스템과 상이하며 응용 프로그램 개발의 복잡성이 증가하는 것이 특징이다. 본 논문에서는 이러한 상황인식 시스템들이 공통적으로 가지고 있는 특징을 기반으로 하여 정형화된 프레임워크를 제시한다. 이것은 복잡한 전체 시스템을 조망할 수 있는 도구가 되며, 이를 반영한 시스템 개발 환경은 편의를 제공할 뿐 아니라, 유용한 분석의 기반이 되는 등의 장점들을 가진다. 제시된 프레임워크에서는 반응적 (reactive) 작업 수행을 위해 이벤트에 대한 규칙을 통해 동작하도록 정의하고, 많은 상황인식 응용 프로그램에서 인식하는 상태 전이 개념을 도입하였다. 또한, 각 장비의 상태 변화가 전체 작업 수행에 연결되어 모델링 될 수 있도록 하였다.

Abstract

Context-aware systems are reactive computing systems, aware of external context. Considering various sensors and actuators connected, application programming on top of such systems is known to be much more complex than in contentional ones. This paper suggests a formal framework for context-aware systems, by extracting their common properties. That makes a useful birds-eye view for the behaviors of a whole complex system, as a base for a convenient developing environment and systematic analysis. In this framework, reactive-ness is handled by event-condition-action rules and global state-transitions, which are essential in a lot of context-aware applications. In addition, behaviors of each elementary device are modelled with its own state-transitions, and tightly bound to the entire task.

Keywords: Context-aware systems, ECA rules, State transitions, Semantics

I. 서 론

상황인식 (context-aware) 시스템에서 응용 프로그램을 개발하는 것은 일반 응용 프로그램 작성과는 차이가 있다. 무엇보다도 개발자가 인지하는 전체 상황은 일반적인 데이터 외에 센서, 사용자, 서비스 등의 다양한 동적인 구성요소를 포함하여 복잡성이 증가한다. 또한, 이들을 구성하는 방식이 현존하는 각 상황인식 미들웨어마다 다르므로[1~4], 새로운 시스템에 대한 개발을 위한 진입 소요 시간이 길어지고, 하나의 시스템에서 개발된 응용을 다른 시스템으로 이식하는 과정의 복잡성이 증가한다는 점도 있게 된다.

본 논문에서는 상황 인식 시스템의 작업 구조에 보편적으로 적용될 수 있는 모델을 정의하고, 정형화 (formalization) 된 프레임워크를 제시한다. 이러한 상황인식 시스템들이 공통적으로 가지고 있는 특징을 기반으로 하여 정형화된 프레임워크를 제시한다. 이것은 복잡한 전체 시스템을 조망할 수 있는 도구가 되며, 이를 반영한 시스템 개발 환경은 편의를 제공할 뿐 아니라, 유용한 분석의 기반이 되는 등의 장점들을 가진다.

먼저 본 논문에서는 이를 위해 상황인식 시스템의 특징을 구성하는 요소들을 다음과 같이 정리한다.

* 정회원, ** 학생회원, 충남대학교 컴퓨터전공
(Dept. of Computer Sci. & Eng. Chungnam National University)

※ 이 논문은 2006년도 충남대학교 학술연구비의 지원에 의하여 연구되었음.
접수일자: 2008년11월18일, 수정완료일: 2009년2월26일

- 개발 단계에서 설계되는 전체시스템의 작업 시나리오[5~6]
- 단위 데이터와 이들 간의 관계[5, 7]
- 상황의 변화에 따라 발생하는 이벤트 및 이에 반응적인 컴퓨팅[5, 7]
- 컴퓨팅을 수행하는 개개의 물리적 환경[7]
- 시스템 전체의 현 상태[7]

기존에 시도되었던 상황인식 시스템의 정형화된 프레임워크는 이러한 사항들을 표현하기에 충분하지 못하다. Univ. of Illinois[1, 8~9]에서 제시한 1차 논리식 기반의 데이터 상황 정의와 반응형 프로그래밍을 위한 이벤트 시스템의 정의는 전체 시스템의 작업 시나리오나 동작 등에 대한 체계적인 표현이 부족하다. Univ. of Florida[10]에서도 연산 의미구조를 사용하여 장비와 상황의 변화를 정의하였으나, 처리모듈에 대한 고려 없이 상태의 변환만을 다루었기 때문에 반응적 프로그래밍이나 상황의 변화에 따른 상태 전이에 대한 부분이 미흡하다. Ambient calculus에 기반한 정형화도 가능하나[11~12], 코드의 활발한 이동이 중심이 되는 로직으로서, 평범한 상황인식 시스템을 묘사하기에 적합하지 않다.

또한, 전체 상황인식 시스템에 대한 모델은 아니지만 개별 장비의 상태를 묘사하는 모델링 방법들이 제시되어 왔다[13~15]. 이들을 이용하여 전체 시스템을 표현하기 위해 단위 모델을 다단계로 결합하는 방식[16~17]을 취해볼 수도 있으나, 상황정보 표현을 비롯하여 전체 시스템의 수행이나 상태를 묘사하기 위한 추상화 정도가 낮아, 앞서 언급된 상황인식 시스템의 특징들을 표현하기에는 필요이상의 복잡성을 가지게 될 가능성이 크다.

본 논문에서는 상황인식 시스템의 설계단계에서 정의되는 전체 작업 시나리오와, 실제 개별 장비들이 수행하는 상태 전이도들이 체계적으로 융화되어 정의될 수 있는 프레임워크를 제시한다. '상황'은 일반 '데이터 상황' 뿐 아니라, 시스템, 장비들과 연결되어 있는 동적 구성요소들의 상태인 '구동 구조 상황'을 포함하도록 정의된다. 또한 상황의 변화나 센서 입력 값 등에 반응하는 컴퓨팅은 ECA 규칙[18]으로 표현하고, 상황의 변화에 따른 반응을 위해서는 태스크(task)라고 불리는 전역적 유한상태기계를 도입함으로써, 개발 단계에서 정의되는 시나리오를 모델링하였다.

그리고 이러한 태스크의 상황정보 처리 과정은 구조적 연산 의미구조 (structural operational semantics)[19]로 정형화 하여 기술하였다. 여기서는 개별 장비의 상태 전이와 시스템 내의 데이터의 변화, 이벤트, 메모리 등이 포함된 반응적 수행에 대한 총체적인 의미구조를 구성하고 있다. 또한 제안된 프레임워크에 따라 구성된 데모 구현에 대한 예를 소개한다.

II. 본 론

1. 데이터 상황(data context)

데이터상황은 현재 시스템 내에 있는 서로 유기적으로 연결되어 있는 각종 정보 중, 시스템의 동작과 관련되지 않은 일반적인 정보를 의미한다. 이것은 변경이 잦지 않아 데이터베이스나 파일시스템에 저장되는 것이 적합한 정적인 정보와, 메모리나 센서로부터 취득한 동적인 정보를 모두 포함하게 된다. 이들 데이터는 서로 연관되어 있어 객체지향적인 표현, 세미 구조 데이터 모델 (semi-structured data) 패스 표현식[20], 또는 시멘틱 웹[21~22]과 같은 관계 중심적 표현방식으로 나타내는 것이 자연스럽다. 우리는 이것을 *Context_data*라고 명명한다.

본 논문에서는 W3C의 시멘틱웹 표준 중 RDF (Resource Description Format)[21]를 토대로 한 구조를 사용하여 데이터 상황을 나타낸다. 따라서 RDF의 <*subject property object*>의 트리플의 집합으로 모델링하는 것이 상황을 나타내는 정보의 주된 구조가 된다. *subject*와 *property*는 각각 ID와 문자열로 표현되고, *object*는 ID이거나 구체적인 값(여기서는 문자열 또는 수)으로 단순화시켜 가정한다. 각각의 집합을 *Subject, Property, Object*라 하면 다음과 같이 정의할 수 있다.

$$Subject \in ID,\ Property \in String \tag{1}$$

$$Object \subseteq ID \cup String \cup Num \tag{2}$$

$$Context\_data = 2^{Subject \times Property \times Object} \tag{3}$$

각 데이터에 대한 접근은 get, set 연산으로 가능하다고 가정한다. get <*subject property*>는 주어진 *subject*와 *property*에 대한 *object* 값을 돌려주며, set <*subject property new_value*>는 새로운 값 (*new_value*)로 해당 *object* 부분을 변경하는 것을 의미한다. 만일 기존에 주어진 *subject, property*로 시작하는 트리플이 없는 경우

에는 새로 추가되는 결과를 가진다. RDF데이터 관련 질의나 패스(path) 형식 등의 다변화된 정보 추출 등의 사용은 get을 반복적으로 사용함으로써 쉽게 확장될 수 있으나 본 논문에서는 생략한다. 데이터에 대한 세분화된 의미에 관한 처리[23]나 시멘틱웹의 추론 기법[21~22] 역시 적용은 가능하나, 주제에서 벗어나므로 포함하지 않는다.

2. 구동 구조 상황(behavioral context)

일반적인 상황 정보와 달리, 구동 구조 상황은 기기들의 작동 현황을 표현하는 상황이다. 이러한 구동 구조 상황에는 장비들의 수행 구조와 응용 프로그램인 태스크의 수행 구조가 포함된다.

가. 장비 수행 구조(actuator status)

컴퓨터, 로봇, 센서 등을 비롯한 각종 장비들의 내부 상태를 나타낸다. 장비가 가지는 각종 속성값 자체는 앞서 설명된 데이터 상황 (data context)에 포함된다. 그러나 어떤 작업을 수행해왔었고, 향후 어떤 작업 수행이 가능한지에 대한 정보는 표현되지 못한다. 이러한 각 장비들의 수행 구조를 표현하는 상황을 여기서는 *Actuator_status* 라고 부른다.

이를 위해 각종 장비들은 각각 저마다의 상태 전이도와 자신의 현재 상태로 추상화된다. 장비 상태의 전이는 외부에서 주어진 명령 등으로 수행된다. 이것은 임베디드 시스템에서 일반적으로 사용되는 장비 모델링 방법과 유사한 개념이다[13]. 따라서 장비가 가지는 상태와 받아들일 수 있는 명령들, 명령에 의해 상태를 전이하는 규칙들, 시작 상태, 최종 상태들로 정의된다. 본 논문에서는 장비 ID마다 내장된 상태전이도를 대응시키는 함수 *FS* (a Factory of State transition machines)가 정의되어 있다.

$$FS : \mathrm{ID} \rightarrow State\_Transition\_Machines \quad (4)$$

For $id \in \mathrm{ID}$,

$$State\_Transition\_Machines\ (id) = \langle machine\_states,\ inputs\_commands,\ transitions,\ start\_state,\ accepting\_states \rangle \quad (5)$$

transitions 은 주어진 $c \in input\_commands$ 와 $s_1 \in machine_states$에 대해 전이 결과인 $s_2 \in machine_states$를 결과로 하는 함수이다. *accepting_states*는 전원이 꺼진 상태*를 *start_state*는 전원이 켜진 직후 상태로 간주하며, *transitions*에는 최소한 *accepting_states*로 부터 *start_state*로의, 전원을 켜는 명령에 반응하는 전이를 반드시 포함하고 있다. 편의상 이들 각 원소들은 함수 이름으로도 함께 사용하여, 주어진 장비 *id*에 대해 각각의 결과들이 *machine_states(id)*, *inputs_commands(id)*, *transitions(id)*, *start_state(id)*, *accepting_states(id)*로 얻어진다고 가정한다.

각 장비 ID는 현재의 *device_state*를 가지는데, 주어진 일련의 명령 수행의 결과로 얻어진 현재 상태를 의미한다. 따라서 주어진 장비 $id \in \mathrm{ID}$의 *actuator_status (id)*는 $\langle FS(id),\ device_state(id) \rangle$로 나타내어진다.

이러한 장비 구동 구조는 개발 시 사용되는 서비스들에 독립적이다. 즉 상황인식 응용 프로그램 개발자가 각종 장비에 대해 수행을 요청할 때는 장비가 인식하는 명령들을 서비스라는 형태로 추상화하여 전달하게 된다. 이를 통해 장비들이 상황인식 시스템에 동적으로 연결될 수 있는 것을 고려할 수 있으며, 특정 서비스를 지원하는 장비를 탐색할 수 있는 기술 (service discovery)[24~25]의 적용 가능성을 가진다는 장점이 있다. 본 논문에서는 이와 같은 서비스들의 집합을 *Services*라고 정의하고, 서비스에 대한 장비 추출 함수 *discovered_devices*를 다음과 같이 정의한다.

$s \in Services$ 에 대해

(i) initially, discovered_devices $s = \varnothing$

(ii) discovered_devices s

$= discovered\_devices\ s \cup \{id\}$

$if\ id \in \mathrm{ID}$ and $s \in inputs\_commands(id)$ (6)

나. 태스크 구조 상황

상황인식 시스템의 전체 구동을 위한 응용 프로그램은 앞서 언급된 바와 같이 태스크(task)로 표현될 수 있다. 따라서 태스크는 전체 프로그램을 의미하며 그 추상화된 문법의 예시는 그림 1과 같다. 각 심볼 정의에 대한 설명은 추후 각 부분의 해당 주제에 대한 절에서 분산되어 언급된다.

* 전력이 차단된 상태가 아니라 최소한의 전력이 유지되어 연결은 되어 있는 상태로 sleep 상태를 의미한다. 전력이 차단되면 시스템 구성에 포함되지 않음을 의미한다.[24]

(1) ECA 규칙

이벤트(event)는 시스템의 상황이나 상태 변경을 유발하는 것이다. 단일 이벤트와 함께 논리적인 연결, 시간 명시등을 통한 복합이벤트가 존재하게 되나[26], 여기서는 모든 이벤트들이 *Event*라는 집합의 원소로 표현된다고 가정한다. 특정 타입의 이벤트가 인식되었는지에 대해 확인하기 위하여 그림 1과 같이 이벤트 타입을 두어 인지하도록 한다. 이벤트가 전달하는 인자는 이벤트 타입을 *subject*로 하고 $arg_1..arg_n$을 *property*로, 데이터 값을 *object*로 하는 RDF 트리플이 데이터 상황에 포함되는 것으로 본다.

이벤트의 발생은 특정 조건(condition)이 만족되는 경우에 한하여 의미가 부여되기도 한다. 조건은 프로그램의 일반 조건식과 같이 묘사할 수도 있으나, 여기서는 상황정보를 고려한 조건식이라는 점에서 데이터 상황과 일반 값의 비교로서 한정 한다. 그림 1의 '**get**'부분은 데이터 상황을 추출하는 연산이며 <comparator>는 비교연산자, <value_expr>는 일반적인 문자열, 수 등의 값을 가지는 표현식이다.

액션 (action) 이란 시스템이 하는 일의 단위로서, 한번 시작되면 외부 입력의 개입을 받지 않고, 끝마칠 때까지 해당 프로그램에 위임된다. 예를 들어 사용자와의 대화가 전혀 없는 고전적인 프로그램의 경우, 프로그램

```
<task> := task id { <state_list> }
<state_list> := <state_def> <state_list> | ε
<state_def> := [initial] state { <entry_exit_spec>
<eca_rules> <transitions> }
<entry_spec> := entry { <action> }
<exit_spec> := exit { <action> }
<eca_rules> := on ( <event> ) ( <condition> )
               { <action> }
<event> := <event_type>
<condition> := get ( <subject> , <property> )
               <comparator> <value_expr>
<action> := <command_list>
<comman_list> := <command>
               | <command> ; <comman_list>
<command> := { <command_list> }
           | <lhs> := <rhs>
           | set ( <subject> , <property>
                 , <value_expr>)
           | <if_statement> | <while_statement>
           | <function_calls>
           | ...
           | <service_command>
           | idle | skip | exit
<rhs> := <expression> <bop> <expression>
           | <uop> <expression>
           | get ( subject , property )
```

그림 1. 태스크의 구문
Fig. 1. Syntax of Tasks.

을 시작시키는 이벤트 하나와 액션 하나로 구성된다고 볼 수 있다. 그러나 사용자의 개입이나 외부 입력, 상황 변화 등이 프로그램 제어 흐름을 좌우하는 최근의 대부분의 새로운 도메인의 응용들은 다수의 이벤트에 의해 다수의 액션이 수행된다고 볼 수 있다.

ECA 규칙은 이벤트(event)와 조건 (condition)과 액션을 묶은 것이다. 이러한 ECA 규칙의 집합은 *ECA*라고 명명하며, 다음과 같이 정의된다.

$$ECA = Events \times Conditions \times Actions \quad (7)$$

단, *Events, Conditions, Actions* 는 각각 이벤트의 집합, 조건의 집합, 액션의 집합이다.

(2) 상태 전이와 태스크

앞서 설명한 장비 상태 외에도, 전체 시스템의 상황을 추상화한 '상태(state)'가 존재한다. 상태 식별자를 통해 상태에 대한 정보를 접근할 수 있도록 하고, 해당 상태에서 받아들이고 수행하기로 한 ECA 규칙들의 집합을 가지고 있다. 또한 매 상태마다 해당 상태를 시작하는 액션과 벗어날 때의 액션을 정의한다.

$$States = ID \times 2^{ECA} \times Actions \times Actions \quad (8)$$

상황인지 시스템에서 전체 프로그램 단위를 표현하는 태스크 (task) 는 작업 시나리오를 표현하기 위해 다음과 같이 4가지의 튜플로 구성된다.

$$t = \langle task\_states, transitions, initial\_state, accept\_states \rangle \in Task \quad (9)$$

$task_states \in 2^{States}$는 해당 태스크가 가질 수 있는 가능한 유한한 상태들의 집합이다. 이들 상태는 특정 조건에 맞으면 다른 상태로 전이할 수 있는데, $transitions \in 2^{states \times states \times ECA}$는 이것을 나타낸다. 각각의 전이는 출발 상태와 목적 상태를 가지며, 전이를 일으키는 이벤트, 조건 및 전이 동안 수행해야 하는 액션을 ECA 규칙으로 표현하게 된다. $initial_state \in States$와 $accept_states \in 2^{States}$는 각각 해당 태스크를 위한 하나의 시작 상태와 최종 상태들의 집합을 나타낸다. 편의상 이들 각 원소들은 함수 이름으로도 사용하여, 주어진 태스크 t에 대해 각각의 결과들을 $task\_states(t)$, $transitions(t)$, $initial\_state(t)$, $accept\_states(t)$ 등으로 표기한다.

3. 상황 정보 처리를 위한 의미구조 (Semantics)

가. 표현식 의미구조

표현식은 상황정보나 메모리 구조, 상태등을 변화시키지 않고, 결과값을 가지는 언어적 표현을 의미한다. 여기서는 결과값을 수와 논리값으로 정의하고, 표현식의 의미를 나타내는 함수 $A[\bullet]$를 A: $Exp \rightarrow Context_data \rightarrow (Num \cup Bool)$로 나타낸다.

대부분의 표현식들의 의미는 일반적인 프로그래밍 언어에서와 동일하다. 논리식과 수식에 대해 동일한 의미구조 함수 A를 사용하므로 만일 계산이 산술적으로 불가능한 경우가 생길 때는 프로그램 오류로 간주한다. 지면 관계상 덧셈과 동일성 검사 규칙만 예를 들면 다음과 같다.

$$A[e_1 + e_2]\ c = A[e_1]c + A[e_2]\ c \quad (10)$$

$$A[e_1 == e_2]\ c = A[e_1]c == A[e_2]\ c \quad (11)$$

'**get**' 구문은 데이터 상황에 존재하는 정보값을 추출한다. 따라서 가능한 값의 집합이 결과가 된다.

$$A[\mathbf{get}\ s\ p]\ c = \{\ x \mid <s\ p\ x> \in c\} \quad (12)$$

나. 실행 규칙의 특성

먼저 가독성을 위해 하나의 ECA 규칙 r에 대해 $fst(r)$, $snd(r)$, $trd(r)$에 대해 각각 $event(r)$, $condition(r)$, $action(r)$로 표기한다. $State$ 의 원소 s에 대해서도 $snd(s)$ $trd(s)$ $fth(s)$를 각각 $ecas\_of\_state(s)$, $entry\_action\_of(s)$, $exit_action_of(s)$라고 명명한다. 또한 주어진 태스크 t의 $transitions(t)$의 원소 tr에 대해 $fst(tr)$, $snd(tr)$, $trd(tr)$를 각각 $from\_state(tr)$, $to\_state(tr)$, $eca\_of\_tr(tr)$ 로 표기한다.

그리고 이들의 사용을 위해 다양한 함수를 제공한다. 첫째, $next\_cas$ 은 주어진 상태에서 주어진 이벤트에 반응할 가능성이 있는 규칙들을 추출한다.

$$next\_cas\colon States \rightarrow Events \rightarrow 2^{Conditions \times Actions \times States} \quad (13)$$

현재 상태 $s \in States$와 이벤트 $e \in Events$에 대해 $next\_cas\ s\ e$ 는 반응하는 규칙들이 존재 할 때, 해당 조건과 액션, 그리고 다음 상태를 결과에 포함시키게 된다. 만일 추출된 규칙이 다른 상태로의 전이에 속하는 것이 아니라 일반 ECA 규칙이라면 다음 상태는 현재 상태 s로 유지한다. 반응할 규칙이 하나도 존재하지 않는다면 $\varnothing$ 이다.

$$\begin{aligned}
&(i)\ initially,\ next_cas\ s\ e = \varnothing \\
&(ii)\ next_cas\ s\ e = next_cas\ s\ e \\
&\qquad \cup\ (condition(eca_of_tr(t)), \\
&\qquad action(eca_of_tr(t)),\ to_state(t)) \\
&\quad \text{if } tr \in transitions(t) \text{ s.t. } from_state(tr) == s \\
&\quad \text{and } event(eca_of_tr(tr)) == e \\
&(iii)\ next_cas\ s\ e = next_cas\ s\ e \\
&\qquad \cup\ (condition(r),\ action(r),\ s) \\
&\quad \text{if } r \in ecas_of_state(s) \text{ s.t. } event(r) == e \quad (14)
\end{aligned}$$

또한 가독성을 위해 $next_cas\ s\ e$의 한 원소 x가 (c, a, st)일 때, 각 자리의 원소 중 c는 $condition\_of$ (x), a는 $action\_of$ (x), st는 $next_state_of(x)$로 표현하여 사용한다.

다. 의미 구조의 틀

의미 구조를 표현하는 기본적인 틀은 다음과 같은 전이이다. 주어진 환경에 대해 현재의 이벤트($event$)와 명령($command\_list$)이 상황($context\_data$, $device\_state$), 메모리($memory$), 현재 상태($current\_state$), 다음 상태($next\_state$)를 어떻게 바꾸는지를 정의한다.

$$\begin{aligned}
&task,\ FS \vdash <event,\ command_list,\ context_data,\\
&\ device_state,\ memory,\ current_state,\ next_state\\
&\triangleright <event',\ command_list',\ context_data',\\
&device_state',\ memory',\ current_state',\ next_state'>
\end{aligned} \quad (15)$$

여기서는 태스크 구조 상황과 장비 수행 구조의 $FS(id)$ 를 정적인 것으로 보고 의미구조의 수행 환경으로 정의하여 '$\vdash$'의 앞에 표현한다. $context_data$와 $device_state$는 각각 데이터 상황과 장비 수행 구조상의 $current_state$를 나타낸다. '$\triangleright$'의 양쪽에 나타나는 튜플은 각각 단계 (step)라고 명명한다. 본 절의 나머지 부분은, 이러한 형태로 정의된 의미구조 규칙에 대한 소개이다. 수행할 태스크는 t로 명명한다.

라. command 수행 규칙들

액션, 즉 <command_list>에 존재하는 각 단위 명령들은 조건문, 반복문 등 일반 프로그래밍 언어 구조를

포함한다. 여기서는 이들 중에서는 지정 연산(assignment)에 관한 규칙만을 소개한다. 현재 데이터 상황(*cs*)를 인자로 하여, 지정될 값을 계산하는 표현식의 의미 *A*[*exp*]를 구한 후, 이를 가지고 메모리 *m*의 변수 *v*의 값을 변경한다.

$$t, FS \vdash \langle e, \mathbf{v:=}exp, c, d, m, cs, ns\rangle \triangleright \langle e, \mathbf{skip}, c, d, m[v \mapsto A[exp]\ cs], cs, ns\rangle \quad (16)$$

skip은 아무런 변화나 처리도 수행되지 않는 상황이고, 새로운 이벤트도 반응하지 않는다. 시스템 전체에 수행 대상 명령이 **skip** 뿐일 때는 계속 수행이 불가능하므로 종료로 간주한다.

$$t, FS \vdash \langle e, \mathbf{skip}, c, d, m, cs, ns\rangle \triangleright \langle e, \mathbf{skip}, c, d, m, cs, ns\rangle \quad (17)$$

두 개의 명령이 연속적으로 있는 경우에는 앞의 것을 먼저 수행한 결과 상황을 가지고 뒤의 것을 수행한다. 단, 특수한 경우로, 앞의 명령이 **skip**일 경우, 뒤의 명령만을 수행한 것과 동일하다.

$$\frac{t, FS \vdash \langle e_1, a_1, c, d, m, cs, ns\rangle \triangleright \langle e_2, a_1', c', d', m', cs', ns'\rangle}{t, FS \vdash \langle e_1, a_1;a_2, c, d, m, cs, ns\rangle \triangleright \langle e_2, a_1';a_2, c', d', m', cs', ns'\rangle} \quad (18)$$

$$\frac{t, FS \vdash \langle e_1, a_1, c, d, m, cs, ns\rangle \triangleright \langle e_2, \mathbf{skip}, c', d', m', cs', ns'\rangle}{t, FS \vdash \langle e_1, a_1;a_2, c, d, m, cs, ns\rangle \triangleright \langle e_2, a_2, c', d', m', cs', ns'\rangle} \quad (19)$$

데이터 상황은 **set** 명령에 의해 변환될 수 있다. 기존에 관련 정보가 있다면 제거되고 새 정보가 지정되는 방식을 취한다.

$$task, FS \vdash \langle e, \mathbf{set}\ (\mathrm{s\ p\ o}), c, d, m, cs, ns\rangle \triangleright \langle e, \mathbf{skip}, c', d, m, cs, ns\rangle$$

단, $c' = c - \{\langle s\ p\ x\rangle \mid for\ all\ x\ \} \cup \{\langle s\ p\ o\rangle\}$ (19)

서비스를 장비에게 요청하여 수행하는 것은 그림 3.1 문법의 <service_command> 항목에 해당한다.

$$t, FS \vdash \langle e, service\_command, c, d, m, cs, ns\rangle \triangleright \langle e, \mathbf{skip}, c, d\ [id \mapsto new\_dstate], m', cs', ns'\rangle$$

단, $id \in discovered_devices\ service_command$이고,

$$new\_dstate = transitions(id)\ d(id)\ service\_command \quad (20)$$

*id*는 해당 장비의 식별자를 나타내며, 결정적으로 정의될 수도 있고, 제일 먼저 발견되는 장비, 또는 우선순위가 높은 장비가 선정될 수도 있다. *d(id)*는 현재의 장비 상태이고 *new_state*는 현재의 장비상태로부터 *service_command*에 의해 전이되어 나온 상태이다. 장비의 작업 수행이 현재의 데이터 상황 *c*와 독립적이나, 추후 영향을 미칠 수 있도록 정의해야한다. 또한 현재는 *service_command*의 인자는 고려하지 않는다.

마. 상태 전이

여기서는 상태 전이를 반영하는 작업 수행 절차를 규칙화하고 있다. 먼저 액션 선정을 위한 규칙으로부터 시작하는데, **idle**은 특수한 명령으로서 액션과 액션을 수행하는 중간에 공회전을 위한 명령으로 사용된다. 새로운 이벤트를 받아 차기 액션을 선정하는 단계를 표현하기 위해 사용한다.

현재 발생한 이벤트가 *e*인 상태에서 **idle** 명령을 만나면, *e*에 반응할 수 있는 액션을 찾아 수행한다. 이 때 상황이나 상태는 거의 변화하지 않고, 다만 전이가 일어나는 경우 *next_state*가 새로 정의될 수 있다. *e*가 특수한 이벤트인 **null**이 아닐 때 다음 규칙들이 적용된다.

$$(i)\ t, FS \vdash \langle e, \mathbf{idle}, c, d, m, cs, cs\rangle \triangleright \langle e, action\_of(x), c, d, m, cs, next\_state\_of\ (x)$$

if x ∈ *next_cas cs e* and $A[condition\_of(x)]\ c = \mathbf{true}$

$$(ii)\ t, FS \vdash \langle e, \mathbf{idle}, c, d, m, cs, cs\rangle \triangleright \langle \mathbf{null}, \mathbf{idle}, c, d, m, cs, ns\rangle, otherwise \quad (22)$$

null 이벤트는 아무런 이벤트가 발생되지 않은 상태로 환원함을 나타내며 *next_cas* 의 인자로 적용되는 경우 결과는 언제나 ∅이다. 위 규칙은 만일 이벤트가 **null** 이 아니라면 매치되는 ECA 규칙을 *next_cas*에 의해 찾고, 그 규칙의 조건부가 데이터 상황에 의거하여 **true** 일 때, 액션부를 수행함을 나타낸다.

모든 액션은 마지막에 **exit**을 수행한다고 간주하고, 시작 액션 등에서 특별한 내용이 없는 경우에도 **exit** 명령은 가지고 있다고 간주한다. 이 명령은 곧장 **idle** 명령을 수행하면서 다음 이벤트를 기다리거나, 다음 상태로 전이하게 한다. 단, 해당 상태를 벗어날 때 수행되는 액션은 편의상 **exit**이 없다고 가정한다.

$$(i)\ t, FS \vdash \langle e, \mathbf{exit}, c, d, m, cs, ns\rangle \triangleright \langle \mathbf{null}, \mathbf{idle}, c, d, m, cs, ns\rangle$$

if $cs == ns$

(ii) $t, FS \vdash \langle e, \textbf{exit}, c, d, m, cs, ns\rangle$
$\triangleright \langle \textbf{null}, exit\_action\_of(cs); entry\_action\_of(ns), c, d, m, ns, ns\rangle$, if $cs \neq ns$ (23)

공회전 동안 이벤트를 받으면 다음과 같이 단계가 바뀌고 (22)번 규칙을 다시 적용할 수 있게 된다.

$t, FS \vdash \langle \textbf{null}, \textbf{idle}, c, d, m, cs, cs\rangle$
$\triangleright \langle e, \textbf{idle}, c, d, m, cs, cs\rangle$
if e = *new_event* (24)

*new_event*는 발생된 이벤트 중 가장 먼저 처리해야 할 대상이다. 아무 이벤트가 발생하지 않았다면 **null**로 간주한다. 이는 이벤트를 일반적인 프로그래밍 언어의 의미구조에서 동적으로 메모리를 할당받을 때 새로운 주소값을 정의하는 과정과 유사하게 표현한 것이다. (예: 함수 $new : Loc \rightarrow Loc$ 와 토큰 *next*[19])

전체 태스크 t 의 시작을 위해서는 다음과 같은 식을 사용한다. **init_event**는 특수한 이벤트로서 태스크의 시작을 위해 발생된다고 가정한다. $x \in entry_action_of(init_state(t))$이다. t의 *init_state*와 하나의 상태의 *entry*는 각 한 개이므로, 이와 같은 x는 결정적으로 정해진다. *device_state*의 초기값은 각 장비 id를 시작 상태로 대응시켜주는 *start_state*과 동일하다. *initial_context_data*는 *context_data*의 초기 상황을 나타낸다.

$t, FS \vdash \langle \textbf{init\_event}, action\_of(x), initial\_context\_data, start\_state, memory, initial\_state(t), initial\_state(t)\rangle$ (25)

4. 적용 예

이와 같은 프레임워크는 새로운 상황인지 프로그래밍 언어나 개발자 라이브러리를 만들 때에 사용될 수 있으며 미들웨어를 구상할 때도 도움을 받을 수 있다. 현재 이 프레임워크를 전적으로 채용한 상황인지 프로그래밍 언어로서 PLUE[5]가 개발 되어 있다. PLUE 개발자는 응용 시나리오대로 태스크 형태로 프로그램을 기술하면 궁극적으로는 각 장비의 상태를 전이시키게 되어 원하는 결과를 얻게 된다. 그림 2는 조명, RFID, PDA, 레고 로봇(Lego NXT)으로 이루어진 PLUE의 데모 구성을 보여준다. 특정 공간에 사람이 들어오면 조명이 켜지고 로봇이 반기며, 사람이 나가면 로봇은 인사를 하고 조명이 꺼지면서 청소를 시작한다. 만일 외부에서 청소를 멈추라는 신호를 서버에 보내면 로봇에 전달하여 중단하게 한다. 그림 3은 이와 같은 구성 내의 장비 상태전이도를 나타내며, 그림 4는 태스크를 나타낸다.

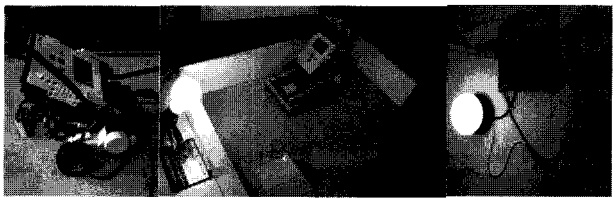

그림 2. 데모 구현을 위한 장비들의 구성 예
Fig. 2. Example organization of devices for the demo implementation.

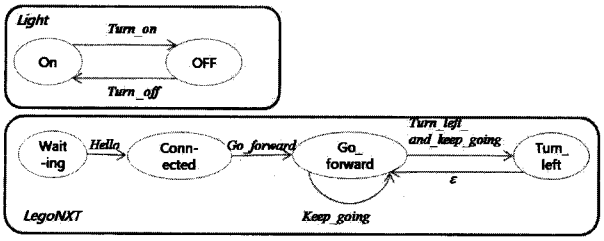

그림 3. 개별 장비 상태 전이도 예
Fig. 3. Example of state transition diagrams of individual devices.

이와 같은 예에서 태스크가 시작된 후 프레임워크에 따른 수행 절차는 다음과 같다. 처음에는 Entry 액션을 수행하며 셋업을 시킨 후 새로운 이벤트가 들어올 때까지 idle 하며 대기한다.

$t, FS \vdash \langle \textbf{init\_event}, \textbf{connect\_Robot; exit}, c,$ {LegoNXT↦Waiting, Light↦OFF...}, $\lambda v.0$, Welcoming, Welcoming>
$\triangleright \langle \textbf{init\_event}, \textbf{exit}, c,$ {LegoNXT↦Connected, Light↦OFF...}, $\lambda v.0$, Welcoming, Welcoming> *by* (18),(20)
$\triangleright \langle \textbf{null}, \textbf{idle}, c,$ {LegoNXT↦Connected, Light↦OFF...}, $\lambda v.0$, Welcoming, Welcoming> *by* (23)
$\triangleright \langle \textbf{null}, \textbf{idle}, c,$ {LegoNXT↦Connected, Light↦OFF...}, $\lambda v.0$, Welcoming, Welcoming> *by* (24)
... (반복)

사용자가 공간에 진입하면 RFID 로 읽게 되고 **UserEnter** 이벤트가 발생된다. 따라서 매치되는 규칙을 찾아 수행하게 된다. **exit** 명령을 만나면 다시 idle을 시작한다.

▷ <**UserEnter**, **idle**, *c*, {LegoNXT↦Connected, Light↦OFF...}, λυ.0, Welcoming, Welcoming> *by* (24)

▷ <**UserEnter**, **turn_on the_light; send Robot the_user;exit**, *c*, {LegoNXT↦Connected, Light↦OFF...}, λυ.0, Welcoming, Welcoming> *by* (22)

▷ <**UserEnter**, **send Robot the_user;exit**, *c*, {LegoNXT↦Connected, Light↦ON...}, λυ.0, Welcoming, Welcoming> *by* (18),(20)

▷ <**UserEnter**, **exit**, *c*, {LegoNXT↦Go_forward, Light↦ON...}, λυ.0, Welcoming, Welcoming> *by* (18),(20)

▷ <**null**, **idle**, *c*, {LegoNXT↦Go_forward, Light ↦ON...}, λυ.0, Welcoming, Welcoming> *by* (23)

▷ <**null**, **idle**, *c*, {LegoNXT↦Go_forward, Light ↦ON...}, λυ.0, Welcoming, Welcoming> *by* (24)

..... *(반복)*

사용자가 공간에서 나가면 **UserExit** 이벤트가 발생하며 매치되는 전이코드가 수행된다.

▷ <**UserExit**, **idle**, *c*, {LegoNXT↦Go_forward, Light↦ON...}, λυ.0, Welcoming, Welcoming> *by* (24)

▷ <**UserExit**, **make_bow Robot; exit**, *c*, {LegoNXT↦Go_forward, Light↦ON...}, λυ.0, Welcoming, Waiting> *by* (22)

▷ <**UserExit**, **exit**, *c*, {LegoNXT↦Turn_left, Light↦ON...}, λυ.0, Welcoming, Waiting> *by* (18),(20)

전이 코드 수행 후에는 현재 상태와 다음 상태가 다른 것으로 상태 전이가 발생함을 알게된다. 따라서 Exit 액션과 다음 상태의 Entry 코드를 수행 한다. 현재 상태는 다음 상태로 바뀐다.

▷ <**null, turn_off the_light; start_cleaning Robot;exit**, *c*, {LegoNXT↦Turn_left,Light↦ON...}, λυ.0, Waiting, Waiting> *by* (23)

▷ <**null, start_cleaning Robot;exit**, *c*, {LegoNXT↦Turn_left, Light↦OFF...}, λυ.0, Waiting, Waiting> *by* (18),(20)

▷ <**null,exit**, *c*, {LegoNXT↦Cleaning, Light↦OFF...}, λυ.0, Waiting, Waiting> *by* (18),(20)

▷ <**null,exit**, *c*, {LegoNXT↦Cleaning, Light↦OFF...}, λυ.0, Waiting, Waiting> *by* (18),(20)

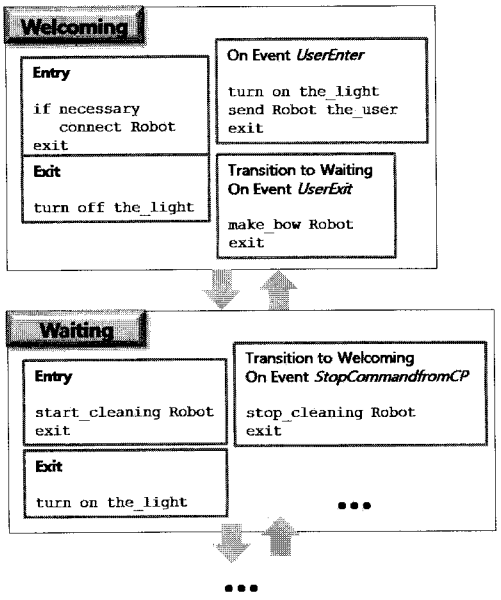

그림 4. 태스크 예 (부분)
Fig. 4. Example of a fragment of a Task.

이후 Entry에서 요청한 청소 수행을 로봇이 끝마칠 수도 있고, 그 전에 외부의 제어 포인트(여기서는 PDA)의 제어 명령 전달에 의해 시스템에 새로운 이벤트 **StopCommandfromCP**가 발생되어 청소가 중단 될 수도 있다. 중단 되는 경우에는 같은 방법으로 상태 전이가 일어나 **Welcoming** 상태로 들어간다.

Ⅳ. 결 론

상황인식 시스템은 각종 센서나 실행기들이 시스템에 연결되어 동작하게 되므로 복잡성이 증가하는 것이 특징이다. 정형화된 프레임워크는 이러한 복잡한 시스템 전체를 조망할 수 있는 유용한 도구가 될 수 있다. 그러나 기존의 상황인지 이벤트 시스템의 정형적 정의나 의미구조는 전체 시스템의 작업 시나리오나 동작에 대한 체계적인 표현이 미흡하다는 단점이 있다. 또한 기존의 개별 장비를 위한 모델링 방법들을 적용하는 경우 추상화 정도가 낮아 표현의 복잡성이 불필요하게 증대되어 부적절하다.

본 논문에서 제시한 정형화된 프레임워크는 상황인식 시스템의 설계 단계에서 정의되는 전체 작업 시나리오를 추상화하고, 이와 함께 실제 개별 장비들이 수행하는 상태 전이도들이 체계적으로 융화되어 정의될 수 있는 프레임워크를 제시한다. 이를 위해, ECA 규칙, 시나리오 개념 등의 추상화 단위를 두었으며, 이들이 각

개별 장비의 상태의 전이과 함께 전체 시스템 모델에 유기적으로 연결되도록 하고 있다. 이에 따라 서론에서 언급한 상황인지 시스템 프레임워크의 주요 요소들(전체시스템의 작업 시나리오, 단위 데이터와 그 들 간의 관계, 반응적인 컴퓨팅, 물리적 디바이스, 전체 시스템의 상태)을 보다 명확히 반영하고 있다.

이러한 정형화된 프레임워크는 또한 전체 시스템의 동작을 예측, 분석하는 것에도 응용될 수 있다. 현재 이 프레임워크를 바탕으로 자원 및 장비 사용 충돌 등에 대한 분석에 대한 연구가 진행 중이다.

참 고 문 헌

[1] C. Shankar, A. Ranganathan and R. H. Campbell, "An ECA-P Policy-based Framework for Managing Ubiquitous Computing Environments", *in Proc. of Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services*, pp33-44, San Diego, California, USA, Jul. 2005.

[2] R. Grimm et. al, "System support for pervasive applications", *ACM Transactions on Computer Systems*, Vol. 22, no. 4, pp421-486, Nov. 2004.

[3] H. Chen et. al, "Intelligent agents meet semantic web in a smart meeting room", *in Proc. of the Third International Joint Conference on Autonomous Agents & Multi Agent Systems (AAMAS 2004)*, pp 854-861, New York, NY, USA, Jul. 2004.

[4] A. Messer, et. al, "A Classification of Pervasive System Software", *in Proc. of Common Models and Patterns for Pervasive Computing Workshop at the 5th International Conference on Pervasive Computing, position paper*, Toronto, Canada, May 2007.

[5] E.-S. Cho, et. al, "Scenario-Based Programming for Ubiquitous Applications", *Lecture Notes in Computer Science, Vol. 4239*, pp286-299, Oct. 2006.

[6] 김병철 외, "유비쿼터스 컴퓨팅 서비스 개발을 위한 시나리오 기반 계층적 접근법", 한국 HCI 학술대회 논문집, 피닉스파크, 대한민국, 2006년 2월

[7] M. Blackstock, R. Lea, and C. Krasic, "Adapting Ubicomp Systems to a Common Model", *in Proc. of Common Models and Patterns for Pervasive Computing Workshop, at the 5th International Conference on Pervasive Computing, position paper*, Toronto, Canada, May 2007.

[8] A. Ranganathan and R. H. Campbell, "An infrastructure for context-awareness based on first order logic", *Personal and Ubiquitous Computing*, Vol. 7, no. 6, pp353-364, Oct. 2003.

[9] C. S. Shankar and R. Campbell, "A Policy-based Management Framework for Pervasive Systems using Axiomatized Rule-Actions", *in Proc. of the Fourth IEEE International Symposium on Network Computing and Applications*, pp255-258, Washington, DC, USA, Jul, 2005.

[10] E. Jansen et. al, "A Programming Model for Pervasive Spaces," Submitted to the 3rd *International Conference on Service Oriented Computing*, Amsterdam, Netherlands, Dec., 2005.

[11] L. Cardelli and A. D. Gordon, "Mobile Ambients", *Lecture Notes in Computer Science*, Vol. 1378, pp 140--155, Apr. 1998.

[12] P. Braione and G. P. Picco, "On Calculi for Context-Aware Coordination", *Lecture Notes in Computer Science*, Vol. 2949, pp. 38-54, Feb. 2004.

[13] D. Harel, "Statecharts: A Visiual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, pp. 231-274, Jun. 1987.

[14] N. H. Cohen and K. T. Kalleberg, "EventScript: an event-processing language based on regular expressions with actions", *in Proc. of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pp111-120, Tucson, USA, Jun. 2008.

[15] P. Caspi et. al,"LUSTRE: a declarative language for real-time programming", *in Proc. of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp178-188, Munich, Germany, Jan. 1987.

[16] E. Cheong et. al, "TinyGALS: a programming model for event-driven embedded systems", *in Proc. of the 2003 ACM symposium on Applied computing*, pp698-704, Melbourne, Florida, USA, Mar. 2003.

[17] A. Ledeczi et. al, "Modeling Methodology for Integrated Simulation of Embedded Systems", *ACM Transactions on Modeling and Computer Simulations*, Vol. 13, no. 1, pp82-103, Jan. 2003.

[18] Corporate Act-Net Consortium, "The Active Database Management System Manifesto: A Rulebase of ADBMS Features", *ACM SIGMOD Record*, Vol. 25, No.3, pp 40-49, Sept. 1995.

[19] H. R. Nielson and F. Nielson, *Semantics with Applications*, Wiley, ISBN 0-471-92980-8, 1992.

[20] J. Mchugh and S. Abiteboul, "Lore: A Database Management System for Semistructured Data", *SIGMOD Record*, Vol 26, No 3.. pp54-66, Sept. 1997.
[21] Frank Manola, Eric Miller, eds., Resource Description Framework Primer, W3C Recommendation, http://www.w3.org/TR/rdf-primer/, Feb. 2004.
[22] D. L. McGuinness and F. v. Harmelen, OWL Web Ontology Language Overview, W3C Recommendation, http://www.w3.org/TR/owl-features/, Feb. 2004
[23] 이창열, "*유비쿼터스 정보 모델링 및 표현 언어 개발*", *대한전자공학회 논문지 CI*, Vol. 44 no.1, pp. 19~25, Jan. 2007.
[24] UPnP Device Architecture 1.0, UPnP Forum, Apr. 2008.
[25] JiniTM Technology Architectural Overview, Technical white paper, Sun microsystems, Inc. 1999, http://www.sun.com/software/jini/whitepapers/architecture.html
[26] I. Cervesato, M. Franceschet and A. Montanari, "A Guided Tour Through Some Extensions Of The Event Calculus", Computational Intelligence, Vol. 16 no. 2, pp307-347, May. 2000.

저 자 소 개

조 은 선(정회원)
1991년 서울대학교 계산통계학과 졸업(계산학전공)
1993년 서울대학교 전산과학과 석사
1998년 서울대학교 전산과학과 박사.
1999년~2000년 한국과학기술원 연구원
2000년~2001년 아주대학교 정보통신전문대학원 조교수 대우
2002년~2006년 충북대학교 조교수
2006년~현재 충남대학교 전기정보통신학부 컴퓨터전공 조교수.
<주관심분야: 상황인지 시스템, 상황데이터 모델링 및 언어 등>

민 영 목(학생회원)
2007년 충남대 공과대학 컴퓨터전공 졸업.
2008년 현재 충남대학교 공과대학 컴퓨터공학과 석사과정.
<주관심분야 : 이벤트 프로그래밍, 상황데이터 관리 언어 등>