유비쿼터스 환경 지원을 위한 서비스 Rule 자동 생성기: HCI 의 최소화

유성훈[1], 허길[1], 김진혁[2], 조위덕[1]
유비쿼터스 시스템 연구센터[1], 아주대학교정보통신대학원[2]
{sy05804[1],semylab[1], gene[2],chowd[1]}@ajou.ac.kr

Automatic Rule Generation for Supporting Ubiquitous Environment: Minimization of HCI

Soung Hun You[1], Gil Heo[1], Jin Hyuk Kim[2] , We Duke Cho[1]
Center of Excellence for Ubiquitous System[1], Graduate School of Information and Communication, Ajou university[2]

요 약

컴퓨팅 디바이스의 존재를 의식 하지 않고 원하는 서비스를 제공 받을 수 있는 유비쿼터스 환경하에서는 최소한의 HCI 또는 배제된 HCI 가 요구된다. 이러한 요구를 충족시키기 위해 효율적인 서비스를 제공하는 시스템들은 추론을 통해 사용자의 의도 파악 및 그에 따른 서비스를 제공 할 수 있으나 그것에 대한 정확한 판단은 실질적으로 달성하기 어렵다. 또 다른 접근 방법으로는 Event-Condition-Action (ECA) Rule 형태 기반으로써 명확한 Event Trigger 와 Event 발생시의 상황 조건을 기반으로 이미 기술된 서비스를 제공하는 것이다. ECA 에 의한 서비스의 제공은 확률 기반의 추론을 통한 서비스 제공보다 더욱 명확한 서비스 제공의 판단이 가능하나 복잡한 환경에서 방대한 양의 발생 가능한 모든 Rule 에 대한 기술은 많은 노력이 필요하거나 심지어는 그것이 불가능하다는 단점을 갖고 있다. 이에 본 논문은 이러한 문제를 해결하고자 효과적인 서비스 제공을 위한 ECA Rule 자동 생성 기법을 소개하고자 한다. 본 논고에서 제안하는 시스템은 사용자의 행동과 상황을 추적 및 저장하여 그 정보를 바탕으로 XML 형태의 ECA Rule 을 자동 생성하여 그를 바탕으로 동일한 조건 및 상황 발생시 이미 기술된 서비스를 제공한다. 이러한 과정은 ECA Rule 기반의 서비스 제공 운용에 있어 가장 취약점인 ECA Rule 작성에 대한 사용자의 노력을 Rule 의 양에 상관없이 손쉽게 해결 할 뿐만 아니라 각 사용자 별 Rule 을 생성함으로써 유비쿼터스 환경하에서의 개인화된 서비스를 효율적으로 제공할 것이다.

Keyword : Ubiquitous, HCI, ECA, XML, Rule based

1. 개요

유비쿼터스 환경하에서 Rule 기반 서비스 제공은 사용자의 명시적인 행동과 상황 및 환경 조건을 기반으로 이미 기술 된 Action 이라는 서비스를 제공할 수 있도록 한다. 이러한 행동과 상황 조건에 의한 Action 의 기술은 이미 Rule 형태의 한 축을 담당하며 그것을 통상 ECA (Event-Condition-Action)로 명칭하고 있다. 이러한 ECA 의 기술 (Description)은 여러 형태로 나타나고 있으나 Event 의 발생시 그리고 발생 당시의 조건을 고려하여 특정 형태의 Action 을 취하는 지에 대한 궁극적인 흐름에서는 일치하고 있다. 간단한 예를 들어 표 1 과 같은 경우 거실의 불을 켜기 위한 여러 Rule 중 하나의 Rule 이 될 수 있을 것이다.

Event: 사용자 A가 거실에 들어온다
Condition 1: 현재 시간은 밤 8시이다
Condition 2: 거실의 불이 켜져 있지 않다
Action: 거실의 불을 켠다

표 1. ECA 형태의 Rule

이러한 ECA Rule 에 대한 연구는 여러 분야에서 현재 활발히 이루어지고 있다 [1] [2] [3]. 그렇다면 만약 서비스를 제공하는 시스템이 철저히 Rule 기반으로 운용된다고 가정하여 본다면 서비스 제공을 위한 Rule 은 모든 가능한 경우를 기술하여야 할 것이다. 특정 도메인에서의 서비스 제공은 관련 Rule 을 어느 정도 예측 가능할 것이며 그의 수 또한 제한될 수 있으므로 사용자의 인위적인 작성을 통하여 시스템이 운용할 수 있는 Rule 을 작성 할 수 있다. 그러나 광범위한 영역에서 사용자와의 상호 작용을 최소화 하여 효율적인 서비스를 제공하여야 하는 유비쿼터스 환경하에서는 이러한 강제적인 Rule 정보의 작성은 수많은 경우와 그 방대한 양으로 강제적 또는 인위적 작성은 불가능 할 것이며 결국 시스템이 소유한 Rule 정보에서 제외된 서비스는 결코 사용자에게 적용 될 수 없다. 예를 들어 현재 시각이 9 시 일 경우 위 제시된 Rule 은 결코 적용 될 수 없다. 이러한 문제점을 수정하기 위하여 Condition 1 을 “ 현재 거실의 조명이 켜져 있지 않다” 라는 좀더 광범위한 범위로 확대된 조건의 Rule 로 이 문제를 해결할 수 있으나 역시 방대한 양의 Rule 을 인위적으로 작성하여야 한다는 문제점을 해결하고 있지는 않다. 결론적으로 이 문제를 해결하기 위해서는 효율적인 Rule 의 작성도 중요하지만 그러한 Rule 의 자동 작성이 근본적인 해결책을 제시 할 것이며 사용자로 하여금 이 기술은 수많은 Rule 기술의 자동화로 인하여 그것의 작성으로부터 사용자를 자유스럽게 할 것이다. 이에 본 논고에서는 유비쿼터스 환경하의 원활한 Rule 기반 서비스의 실현을 위한 자동 Rule 생성을 위한 개념과 관련된 운용 시스템을 소개하여 위에서 언급한 문제점을 해결하고자 한다. 이에 본 논고는 Rule 생성을 위한 시스템의 구조와 그 각각 Component 들의 역할과 최종적으로 생성된 XML 타입의 Rule 에 대한 설명을 2 절에서 제시하고 있다. 간단한 시뮬레이션을 통한 구현은 3 절에 소개되었다.

2. Rule 자동 생성기의 구조

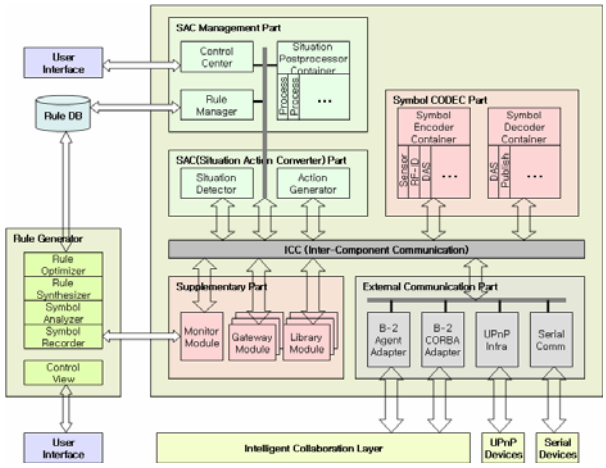

그림 1. Rule 자동 생성기의 운용

그림 1 은 Rule 자동 생성기가 상황 판단과 서비스 선택 및 제공을 위해 본 연구소에서 개발한 Situation Manager (SM) [4] 의 한 컴포넌트로써 연동되고 있는 모습을 보여주고 있다. Rule 기반의 서비스 제공을 위한 SM 에서의 Rule 자동 생성기는 SM 의 기능 중 서비스 제공을 위한 핵심 모듈을 담당하며 이러한 Rule 기반 서비스 시스템에 있어 중요한 역할을 담당한다. 이 그림에서 Rule 자동 생성기는 연동 개체인 SM 의 외부에 존재하며 SM 으로부터 획득한 정보를 가공하여 최종적으로 XML 형태의 Rule 로 생성 시킨다. 생성된 Rule 은 SM 의 Rule Manager 가 서비스를 제공하기 위해 이용하게 된다. 이와 같은 Rule 자동 생성기는 사용자가 Rule 생성을 지시하는 UI 와 그로부터 Event 를 전달 받아 SM 으로부터 정보를 획득하여 Rule 생성기에 전달하는 Monitor Module, Motitor Module 로부터 정보를 획득하여 Rule 생성기의 Local Repository 에 저장하는 Symbol Recorder, 그리고 획득한 정보를 1 차로 정제하여 Draft 한 Rule 로 생성하는 Symbol Analyzer, 기존의 완성된 Rule 과 새로 생성된 Draft Rule 과의 비교를 통하여 Drop, Add, Fusion 의 판단을 내리는 Symbol Synthesizer, 그리고 최종적으로

생성된 새로운 Rule 과 기존의 Rule 들과의 비교를 통하여 최종적으로 새로운 Rule 의 추가 여부를 결정 짓는 Rule Optimizer 로 구성 된다. 또한 최종적으로 구성된 DB 형태의 룰을 XML 타입으로 변환 시키는 DB2XML 의 컨포넌트들로 구성이 되어 있다. 본 절에서는 이러한 각 컨포넌트들의 기본 구성 요소들을 소개하고자 한다.

2.1 Monitor Module (MM)

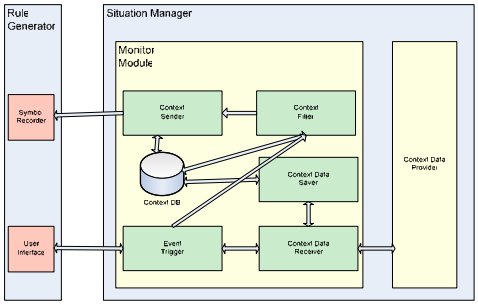

그림 2. Monitor Module 의 구성 요소

그림 2 는 Rule 생성기로부터 Rule 작성의 여부를 전달 받아 상황 정보를 제공 개체로부터 획득하여 그것을 1 차 가공 후 Rule 생성기에 전달하는 역할을 하는 MM 의 구조를 보여주고 있다. 최초 ***Event Trigger*** 는 사용자의 Rule 생성에 대한 여부를 획득한 후 그것이 Start 이라면 ***Context Data Receiver*** 에게 Context Provider 로부터 정보 획득을 시작하라는 명령을 내린다. 만약 그것이 Stop 이라면 ***Context Data Receiver*** 로 하여금 더 이상의 정보 획득을 중지 시키고 ***Context Filter*** 로 하여금 현재까지 저장된 Rule 들의 정제를 시작하도록 지시한다. ***Context Receiver*** 는 ***Context Provider*** 와의 Access Point 역할을 하여 정보를 획득하여 ***Context Data Saver*** 에 전달하면 ***Context Data Saver*** 는 이것을 Local Repository 에 저장한다. ***Context Filtering*** 은 ***Event Trigger*** 로부터 Stop 의 명령을 받으면 현재까지 저장된 정보들 중 불 필요하거나 중복된 정보를 제거하고 임무 완료 시 ***Context Sender*** 에게 그 사항을 알려 1 차로 정제된 정보를 Rule 생성기에 전달하게 된다. 전달이 끝나면 ***Context Sender*** 는 현재 Local Repository 에 저장된 정보들을 삭제한다.

2.2 Symbol Recorder

Symbol Recorder 는 MM 의 Context Sender 로부터 정보 Data 를 전송 받아 단순히 Rule 생성기의 Local Context DB (LCDB)내 Draft_Context_tbl 에 저장 및 기록하는 역할을 수행한다. 기록의 임무 수행이 끝나면 Symbol Analyzer 에게 알려 기록된 Content 들에 대한 분석을 시작하도록 한다.

2.3 Symbol Analyzer

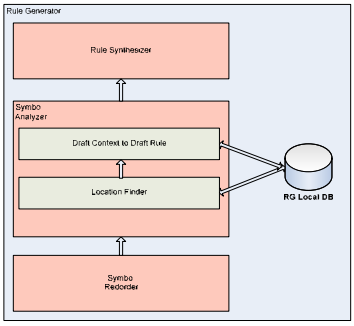

그림 3. Symbol Analyzer 의 구성 요소

Symbol Analyzer 는 저장된 정보에 대한 분석을 수행하고 가장 기본 적인 1 차의 Rule 형태로 변환 및 저장한다. 본 논고에서 제안하는 Rule 생성기는 각각의 Rule 을 위치 단위로 구분하고 있다, 즉 사용자가 방에서 거실로 그리고 화장실로 이동하면서 어떠한 행동을 취하였을 경우 이 상황 및 행동 정보는 시스템에 의해 모두 생성기로 전송이 되어진다. 이러할 경우 Rule 생성기는 이러한 행동 및 상황을 방, 거실, 화장실등의 3 가지 형태로 구분하며 각 지역에만 해당하는 3 개의 Rule 로 생성하게 된다. 이렇게 지역별 Rule 의 구분을 규정 짓는 것은 차후 Rule 의 검색과정이나 적용 과정에서 더욱 유리 할 것이라 판단되기 때문이다. 이러한 Symbol Synthesizer 는 Context 의 정보를 Draft_Context_tbl 로부터 획득한다. 획득한 정보는 ***Location Finder*** 에 의해 현재의 정보가 어느 위치에서 획득됐는지를 확인하고 그에 따란 위치 별로 사용자의 행동과 행동 발생시의 상황을 나타내는 정보들을 구분하여 Action-List_tbl 과 Situation-List_tbl 로 재 저장한다. 이것은 ECA Rule 의 기본

형태를 최초로 따름으로써 Draft Rule 이라 할 수 있으며 ***DraftContexttoDraftRule*** 의 컨포넌트가 담당한다. 이와 같이 새로이 전송되어진 정보를 일차적인 Rule 로 변환 시키는 과정은 Symbol Analyzer 가 담당한다.

2.3 Symbol Synthesizer

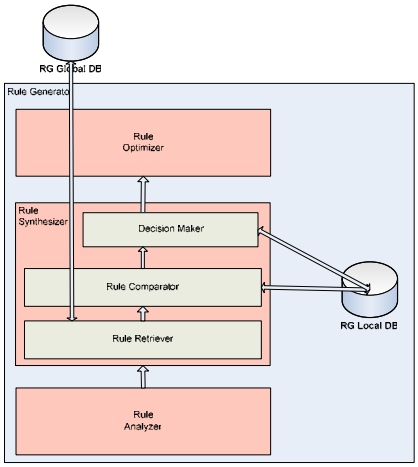

그림 4. Rule Synthesizer 의 구성 요소

Symbol Synthesizer 는 현재 LCDB 에 존재하는 1 차 Rule 과 이미 정제되어 최종적으로 Rule 생성기의 Global DB 에 저장되어 있는 Rule 들과의 비교를 통해 그것을 Drop, Add, 또는 기존의 Rule 과의 Fusion 에 대한 판단을 담당한다. 이러한 기능을 수행하기 위하여 ***Retriever*** 는 현재 새롭게 수립한 Rule 의 발생 위치 (예, 방, 거실, 화장실)를 이용하여 같은 위치에서의 관련 Rule 들을 Global DB 로부터 획득한다. 이렇게 획득한 기존의 완성된 Rule 은 Local DB 로부터 획득한 Draft Rule 과의 비교를 위해 ***Comparator*** 로 전송된다. 이렇게 전송된 정제된 Rule 과 새롭게 작성된 Rule 은 그것들의 유사 정도를 결정하여 그 값을 ***Decision Maker*** 에게 전달한다. ***Decision Maker*** 는 그 유사 정도에 의해 새로운 Rule 으로써의 Add, 또는 기존의 Rule 과의 중복 또는 적정 Threshold 이상으로 유사하여 Drop 할 경우, 또는 기존의 Rule 과 유사하나 Drop 을 할 수 없을 경우 기존의 Rule 과의 Fusion 결정한다. Drop 의 경우 Rule 은 생성 되지 않고 Add 일 경우는 Global DB 에 바로 저장이 된다. 그러나 Fusion 일 경우는 Rule Optimizer 의 과정을 거치게 된다.

2.4 Rule Optimizer

Symbol Synthesizer 의 결정으로 Fusion 이 된 Rule 은 기존의 Rule 과 중복되거나 그 유사한 Rule 이 존재 하고 있을 수도 있다. 이러한 경우는 Optimizer 의 비교를 통하여 새로운 Rule 로써의 등록이 취소 될 수 있다. 하지만 취소되지 않고 새로운 하나의 Rule 로써 생성된 것은 Global DB 에 새로운 Rule 로써 등록을 마치게 된다. 최후 등록을 마치게 되면 Global DB 는 단순한 RDB 형태의 Rule 이 존재하게 된다. 이것을 시스템 또는 실제적 Rule 서비스를 제공해야 하는 컨포넌트가 효율적으로 이용 할 수 있도록 XML 타입의 형식으로 변환이 된다. 이 역할은 Optimizer 내의 ***Symbol Converter*** 가 담당한다. Symbol Converter 는 현재 DB 상에 저장된 Data 의 정보를 본 연구소에서 개발 정의 한 상황과 액션의 기술을 정의하는 XML 기반의 ***Situation Description Markup Language (SDML)***와 사용자의 행동 양식을 기술하는 ***Action Description Markup Language (ADML)***로의 변환을 담당한다. 최종적으로 서비스를 이용하는 개체는 작성된 SDML 및 ADML 을 parsing 함으로써 현재의 상황을 판단하고 그에 따른 액션의 서비스를 제공하게 된다.

2.5 SDML 과 ADML

SDML

SDML 은 사용자가 Rule 생성을 할 당시의 상황을 기술하며 그에 따른 State Diagram 을 자동 생성한다. 표 2 는 SDML 의 DTD 를 보여주고 있다.

```
<?xml version="1.0" encoding="euc-kr"?>

<!ELEMENT rule (header, fa)>
    <!ELEMENT header (name, comment?, workingtime?)>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT comment (#PCDATA)>
        <!ELEMENT workingtime (wtime*) >
            <!ELEMENT wtime (wstart, wstop)>
                <!ELEMENT wstart (#PCDATA)>
                <!ELEMENT wstop (#PCDATA)>
    <!ELEMENT fa (init, transition)>
        <!ELEMENT init (#PCDATA)>
        <!ELEMENT transition (state+)>
            <!ELEMENT state (num, group*, final?, stimeout?)>
                <!ELEMENT num (#PCDATA)>
                <!ELEMENT group (delay?, timeout?, goto,
```

```
                    input+)>
                <!ELEMENT delay (#PCDATA)>
                <!ELEMENT timeout (#PCDATA)>
                <!ELEMENT goto (#PCDATA)>
                <!ELEMENT input (plugin, id, type?, value,
                    value2?)>
                    <!ELEMENT plugin (#PCDATA)>
                    <!ELEMENT id (#PCDATA)>
                    <!ELEMENT type (#PCDATA)>
                    <!ELEMENT value (#PCDATA)>
                    <!ELEMENT value2 (#PCDATA)>
            <!ELEMENT final (fid, fvalue)>
                <!ELEMENT fid (#PCDATA)>
                <!ELEMENT fvalue (#PCDATA)>
            <!ELEMENT stimeout (tid, tvalue)>
                <!ELEMENT tid (#PCDATA)>
                <!ELEMENT tvalue (#PCDATA)>
```

표 2. SDML 의 DTD

ADML

ADML 은 사용자가 Rule 생성을 할 당시의 액션을 기술하며 그에 따른 State Diagram 을 자동 생성한다. 표 3 은 ADML 의 DTD 를 보여주고 있다.

```
<?xml version="1.0" encoding="euc-kr"?>

<!ELEMENT rule (header, fa)>
    <!ELEMENT header (name, comment?, workingtime?)>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT comment (#PCDATA)>
        <!ELEMENT workingtime (wtime*) >
            <!ELEMENT wtime (wstart, wstop)>
                <!ELEMENT wstart (#PCDATA)>
                <!ELEMENT wstop (#PCDATA)>
    <!ELEMENT fa (init, transition)>
        <!ELEMENT init (#PCDATA)>
        <!ELEMENT transition (state+)>
            <!ELEMENT state (num, group*, action*, final?,
                stimeout?)>
                <!ELEMENT num (#PCDATA)>
                <!ELEMENT group (delay?, timeout?, goto,
                    input+)>
                    <!ELEMENT delay (#PCDATA)>
                    <!ELEMENT timeout (#PCDATA)>
                    <!ELEMENT goto (#PCDATA)>
                    <!ELEMENT input (id, type?, value,
                        value2?)>
                        <!ELEMENT id (#PCDATA)>
                        <!ELEMENT type (#PCDATA)>
                        <!ELEMENT value (#PCDATA)>
                        <!ELEMENT value2 (#PCDATA)>
                <!ELEMENT action (signal*)>
                    <!ELEMENT signal (sdelay?, splugin, sid,
                        svalue)>
                        <!ELEMENT sdelay (#PCDATA)>
                        <!ELEMENT splugin (#PCDATA)>
                        <!ELEMENT sid (#PCDATA)>
                        <!ELEMENT svalue (#PCDATA)>
                <!ELEMENT final EMPTY>
                <!ELEMENT stimeout EMPTY>
```

표 3. ADML 의 DTD

표 2, 3 의 각 element 에 대한 설명은 본 연구소에서 작성한 uSDML/uADML Language Specification 에 기술되어 있다.

3. 구현

Rule 의 자동생성 구현 확인을 위해 간단한 Simulation 을 실행하였다. 사용자는 User Interface 의 Start button 을 통하여 Rule 생성의 시작을 지시하였고 SM 은 그 이벤트를 받아 Sample Data 로써 light 의 상태를 임의로 지정 Monitor Module 을 통하여 Rule 생성기에 제공하였다. Rule 생성기는 위에 언급한 일련의 과정을 거쳐 SDML 과 ADML 의 Rule 을 생성하였다. Rule 은 Start 와 Stop 버튼 실행 시 하나의 Rule 을 생성하며 하나의 룰은 SDML (예 SDML_2) 및 ADML (예 ADML_2)을 생성한다. 서비스 제공을 위해서는 위 두 ML 을 동시에 Parsing 함으로써 그것을 가능하게 할 수 있다. 표 4 는 Simulation 을 통하여 생성된 Rule 의 일부를 보여주고 있다.

```
<?xml version="1.0" encoding="UTF-8"?>
<rule>
    <header>
        <name>SDML_2</name>
    </header>
    <fa>
        <init>0</init>
        <transition>
            <state>
                <num>0</num>
                <group>
                    <delay>0</delay>
                    <timeout>0</timeout>
                    <goto>1</goto>
                    <input>
                        <plugin>DAS</plugin>
                        <id>light_0</id>
                        <value>33</value>
                    </input>
                </group>
            </state>
            <state>
                <num>1</num>
                <group>
                    <delay>0</delay>
                    <timeout>1</timeout>
                    <goto>2</goto>
                    <input>
                        <plugin>DAS</plugin>
                        <id>light_1</id>
                        <value>33</value>
                    </input>
                </group>
            </state>
            <state>
                <num>2</num>
                <group>
                    <delay>0</delay>
                    <timeout>2</timeout>
                    <goto>3</goto>
                    <input>
                        <plugin>DAS</plugin>
                        <id>light_2</id>
                        <value>33</value>
                    </input>
                </group>
            </state>
```

표 4. SDML_2

```
<?xml version="1.0" encoding="UTF-8"?>
<rule>
    <header>
        <name>ADML_2</name>
    </header>
    <fa>
        <init>0</init>
        <transition>
            <state>
                <num>0</num>
                <group>
                    <delay>0</delay>
                    <timeout>0</timeout>
                    <goto>1</goto>
                    <input>
                        <id>SDML_2</id>
                        <value>1</value>
                    </input>
                </group>
                <action>
                    <signal>
                        <splugin>DAS</splugin>
                        <sid>light_0</sid>
                        <svalue>33</svalue>
                    </signal>
                </action>
            </state>
            <state>
                <num>1</num>
                <group>
                    <delay>0</delay>
                    <timeout>1</timeout>
                    <goto>2</goto>
                    <input>
                        <id>SDML_2</id>
                        <value>2</value>
                    </input>
                </group>
                <action>
                    <signal>
                        <splugin>DAS</splugin>
                        <sid>light_1</sid>
                        <svalue>33</svalue>
                    </signal>
                </action>
            </state>
            <state>
                <num>2</num>
                <group>
                    <delay>0</delay>
                    <timeout>2</timeout>
                    <goto>3</goto>
                    <input>
                        <id>SDML_2</id>
                        <value>3</value>
                    </input>
                </group>
                <action>
                    <signal>
                        <splugin>DAS</splugin>
                        <sid>light_2</sid>
                        <svalue>33</svalue>
                    </signal>
                </action>
            </state>
```

표 5 ADML_2

표 4 에 의하면 Rule 2 는 ADML_2 의 State 0 로부터 시작한다. Rule 적용과 동시에 ADML 은 액션을 취하는

```
<action>
    <signal>
        <splugin>DAS</splugin>
        <sid>light_0</sid>
        <svalue>33</svalue>
    </signal>
</action>
```

부분을 통해서 DAS 의 light_0 의 값을 33 으로 조정한다. 이때 현재의 상황을 제어하는 SDML_2 는 light_0 의 값이 33 임을 확인하고 현재의 상황을 상황의 State 1 로 변이 시킨다. 이때 적용된 부분은

```
<goto>1</goto>
<input>
    <plugin>DAS</plugin>
    <id>light_0</id>
    <value>33</value>
</input>
```

에 해단된다. 마찬가지로 액션 제어의 ADML_2 의

```
<goto>1</goto>
<input>
    <id>SDML_2</id>
    <value>1</value>
</input>
```

부분에 의해 현재 상태 State 0 에서 상황의 변화가 SDML_2 의 State 1 으로의 변이를 확인하고 액션을 위한 State 1 으로 이동을 하여

```
<action>
<signal>
    <splugin>DAS</splugin>
    <sid>light_1</sid>
    <svalue>33</svalue>
</signal>
</action>
```

의 액션 제어에 의해 light_1 의 불빛을 33 으로 제어하도록 실행을 시킨다.

이와 같은 일련의 과정을 거친다면 사용자는 의도한 서비스를 위한 Rule 을 자동 생성 할 수 있으며 시스템은 적절한 인터페이스를 통하여 저장된

서비스를 제공할 수 있다.

4. 결론

본 논고는 Rule 기반 서비스의 취약점인 Rule 의 생성 문제를 해결하고자 하였다. 유비쿼터스 환경 하에서 Rule 기반 서비스는 의도 추론 기반의 서비스와 함께 서비스 제공 기법의 한 축을 담당할 것이며 두 가지의 기법이 혼용 된다면 더욱 효과적인 서비스를 제공 할 수 있을 것이다. 현재 본 논고에서 제안 한 구조는 기본적인 Rule 자동 생성기로써 아직 각 컨포넌트 별로 더욱 정교한 역할을 할 수 있도록 개선 발전되어야 할 것이다. 차후 개발로써는 현재 사용자의 Rule 생성을 위한 작동 절차에 의해서 Rule 의 생성이 달성되나 더욱 지능화된 시스템으로 발전 된다면 이러한 절차조차 시스템이 담당하여 사용자와의 상호작용은 더욱 간소화 될 것이다.

5. 참고 문헌

[1] Diego Lopez de Ipina, Eleftheria Katsin, “An ECA Rule-Matching Service for Simpler Development of Reactive Applications”, IEEE Distributed Systems Online 2001

[2] J. Bailey, G. Papamarkos, A. Pouloyassilis, P. Wood, “An Event Condition Action Language for XML”, Web Dynamics 2004

[3] T. Heimrich, G. Specht, “Enhancing ECA Rules for Distributed Active Database Systems”, NODe 2002

[4] H, Park, H, Heo, W, Cho, “Situation-Aware based Digital Appliance Control System for Smart Home”, UbiCNS 2005