

A New Programming Language for Ubiquitous Applications

Min Young. Kim ,

Eun-Sun cho

Kang-Woo Lee,

Hyun Kim

School of Electrical and Computer Engineering
Chungbuk National University. Korea

{mykim, eschough}@chungbuk.ac.kr

ABSTRACT

Applications in ubiquitous environments manage various types of data and dynamic changes of situation in real world. Traditional programming languages, lack of straightforward features for such management, are not sufficient for ubiquitous application programming. This paper suggests an object-oriented language named 'PLUE (a Programming Language for Ubiquitous Environments)', which will help programmers write ubiquitous applications. PLUE supports ECA (event-condition-action) rules [1] and FSA (finite state automata)-based interactive responses to dynamic situations. It also allows manipulating UDM (Universal Data Model) data in forms of conventional path expressions.

Keywords

Ubiquitous computing, Java, ECA-rules, State transition, Path expressions

1. INTRODUCTION

As mentioned by Mark Weiser in 1988, ubiquitous computing enables people to handle all possible information 'anywhere and any time' [2]. Among the diverse research issues on ubiquitous computing in progress, the followings have been mostly concentrated.

- **Networks:** Numerous heterogeneous devices need to connect to networks anywhere and to identify themselves mutually.
- **Sensors:** By monitoring users' locations and situational changes, user's behaviours are predicted.
- **Middlewares:** Middlewares collect data delivered by sensors and offer services to users. In addition, they discover services.

In addition, they begin to devote some efforts on other interesting issues including programming paradigms for ubiquitous environments recently. Traditional programming languages are not sufficient for ubiquitous application programming due to the lack of the following features;

- **Strictly related to the environment:** A ubiquitous environment embeds complicated information, which varies dynamically and constantly. Applications must immediately respond to the changes of situations.
- **Collaboration of various services:** In ubiquitous environments, users and services themselves share various and complex services distributed in network. So the services must be accessible by applications in direct ways.

This paper suggests an object-oriented language named 'PLUE (a Programming Language for Ubiquitous Environments)', which will help programmers write ubiquitous applications. PLUE supports ECA (event-condition-action) rules and FSA

Electronics and Telecommunications Research Institute.
Korea

{ kwlee, hyunkim }@etri.re.kr

(finite state automata)-based interactive responses to dynamic situations. It also allows manipulating UDM (Universal Data Model) data in forms of conventional path expressions.

2. The Data Model for PLUE

This section covers 'UDM (Universal Data Model)', a data model which provides a uniform view of information from various sources and formats. It introduces path expressions in PLUE, a language feature to express data of UDM in a PLUE application.

2.1 UDM (Universal Data Model)

In Ubiquitous environments, various formats of data taken from heterogeneous devices are retrieved, stored and managed. Further more, the services that users request to do a job have also various formats. UDM is a kind of an interface showing such heterogeneous formed data and services in a uniform format.

Like a simple directed graph, the structure of UDM consists of nodes and associations. An association describes the relationships between two nodes. Nodes have two types -- a normal type and a valued type. While the valued nodes have actual data values, the normal nodes need not have data, but behave as placeholders to represent sequence of associations.

The Fig. 1 describes UDM data. A circle represents a node; a dark circle is for a valued node and a white circle is for a normal node. A directed arrow represents an association.

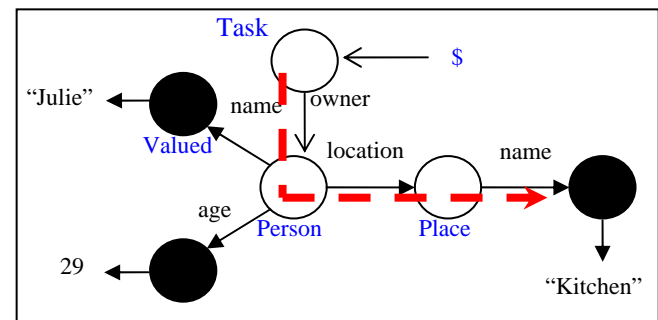


Figure 1. An example for Universal Data Model .

2.2 Path Expressions

As mentioned above, path expressions are used to describe UDM data in PLUE applications. Fig.2 shows the grammar for the path expression.

As seen above, a PLUE path expression is similar to conventional path expressions in common object oriented languages or Xpath [3], except that it consists of '\$', association names, and

conditions.

```

<PathExpression>
    ::= '$' [ '.' ] <AssociationList>
    [ '[' <Condition> ']' ]
<AssociationList> ::= <Association>
| <Association> '.' <AssociationList>
<Wildcard> ::= '*' | '**' | <id> '%'
<Association> ::= <id> | '[' <id> '['
| <Wildcard>
<Condition> ::= <Order> | <ConditionExp>
<Order> ::= <NUM> | <Range>
| <Order> ',' <NUM>
| <Order> ',' <Range>
<Range> ::= <NUM> '-' [ <NUM> ]
<ConditionExp> ::= '[' <id> '=' <Value>
<Value> ::= '[' <id> '[' | <NUM>

```

Figure 2. The PLUE Path Expression Syntax

The character '\$' represents the current task that requested a UDM data. A PLUE path expression is always started a character '\$'. As usual path expressions, a sequence of association names constitutes a path from \$ which usually ends up with a valued node. Conditions consist of order descriptions or condition expressions. Order descriptions are used to select data when a path expression describes a number of nodes with an order. Let us consider the following path expression, which describes residents that are 1st, 3rd, 4th, 5th, and 9th of the residents in the same location with the task owner.

```
$.owner.location.residents[1,3-5,9]
```

A condition expression selects nodes that satisfy the conditions. Currently, the right side of a condition expression has an association name related with the valued node, while any value can come at the left side. Two operators '==' and '!=' are used in a condition expression. The following path expression uses a condition expression for a resident that named 'Julie' and is in same location with Task owner.

```
$.owner.location.residents[.name=='Julie']
```

Path expression uses three wildcards. '*' represents an anonymous association. '**' represents a sequence of anonymous associations in any length. '%' represents an anonymized part of association name.

In Fig. 1, the red dashed arrow represents UDM data expressed a path expression '\$.owner.location.name', which returns a 'kitchen'.

3. The PLUE Grammar

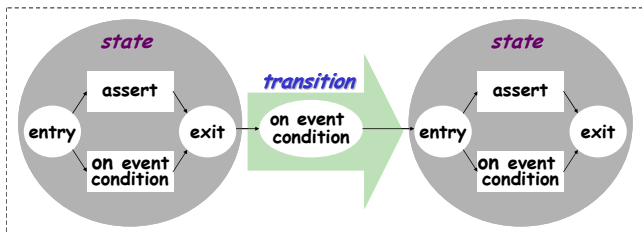


Figure 3. The Structure of PLUE

In a PLUE program, the entire job that asked to be done is called a 'task'. A PLUE task is modularized with states and their transitions. If an event occurs in the current state and it satisfies the condition of any transition, then the current state is changed to the next state along the transition. Since massive events and fluctuating situations in ubiquitous environments directly modeled in PLUE, it is easier for the programmers to write and maintain application programs for ubiquitous computing.

3.1 States

As shown in Fig. 4, a state consists of four parts -- entry, exit, assert, and on-event-condition. The entry part initializes the state. The variables used in the state are initialized and resources are prepared. Asserts phrases are conditions that must be kept for the time being when the state is the current state of the task. 'On-event-condition' phrases describe the actions to be taken when specific events occur. The exit part describes the work that must be done before transit to other states.

```

<StateDeclaration> ::=
    [ "initial" | "final" ] "state"
    <IDENTIFIER> " { "
    { <EntryDeclaration>
    | <ExitDeclaration>
    | <OnEventDeclaration>
    | <PathAssertStatement>
    } "

```

Figure 4. The State Syntax

Fig. 5 is a code fragment of PLUE program, showing that the state 'Talking' describes a situation that a speaker is giving a talk in a conference. The entry part of this state prepares the presentation slides for him on the screen of the conference room.

The assert phrase informs that the light must be turn off while the speaker is making a speech. If the speaker talks 'Next', then the next slide is shown. If any transition occurs, then the exit phrase sets current speaker to the next speaker.

```

state Talking{
  entry{
    $.platform.slideshow.slide_path
    =$.current.material;
  }
  exit{ $.current=$.current.next; }
  assert $.platform.light==false;
  on event VoiceReceived(e)
  condition(e.speech =='next'){
    $.platform.slideshow.next();
  }
}

```

Figure 5. A State in PLUE

3.2 Transitions

```

<TransitionDeclaration> ::=
  "transition" <StateName> "->" <StateName>
  "{ "
    { <OnEventDeclaration>
      | <BlockStatement>
      | <PathAssertStatement> }
  " } "

```

Figure 6. The Transition Grammar

A transition changes the current state into the next state when an event satisfying the transition condition occurs. The transition consists of from-state, to-state, the ‘on-event/condition’ and the action. The ‘on-event /condition’ includes the transition condition as well as what must be executed while the transition. The following program describes the transition from the WaitForTalker state to the Talking state. If a chairman says “Start presentation”, then the data for the first speaker is prepared and the TTS speaks “Start” to attendees while the transition is made.

```

transition WaitForTalker -> Talking{
  on event VoiceReceived(e)
  condition
  (e.speech=='startpresentation'){
    $.current = $.conference.first;
    $.room.tts.speak("start");
  }
}

```

Figure 7. A Transition in PLUE

4. PLUE processing procedure

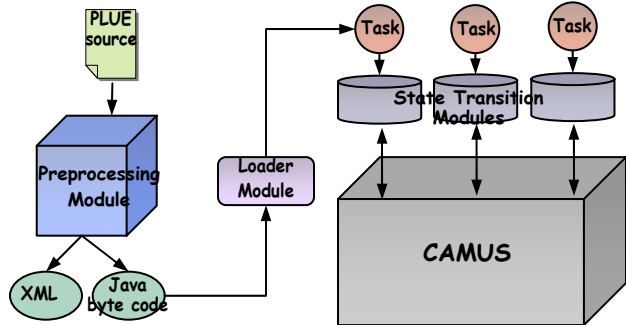


Figure 8. The Architecture

A PLUE program put into PLUE Preprocessor is translated to Java codes and an XML file, and a Java compiler compiles the generated Java codes to byte codes. Loader module creates a task object from the Java byte codes and the XML file and then delivers it to the State Transition Machine dedicated to the task.

4.1 PLUE preprocessing modules

PLUE Preprocessor, based on JavaCC 4.0 beta 1 [4] for parsing, takes a PLUE program to yield new Java byte codes and an XML file. Each tag in the XML file has corresponding to the PLUE language features including <task>, <state>, <entry>, <exit> and ‘assert’; the name attribute of <task> or <state> is the name of the task (or the state), and that of <entry>, <exit>, <assert> or <action> is the method name in the Java byte codes that describes the corresponding behaviour.

```

<?xml version="1.0" encoding="EUC-KR"?>
<task name= "ConfAssistant" >
  ...
  <state name= "Talking" >
    <entry name= "stTalking$entry" />
    <exit name= "stTalking$exit" />
    <assert name= "stTalking$assertECA1" />
    <recovery name= "stTalking$recoveryECA1" />
    <rule name= "rule0" >
      <event name= "VoiceReceived" />
      <condition >
        <![CDATA[ e.speech=='next' ]]>
      </condition >
      <action name= "stTalking$rule0Action" />
    </rule>
  </state>

  <transition from="Talking" to="WaitForTalker" >
    <rule name= "rule3" >
      <event name= "VoiceReceived" />
      <condition >
        <![CDATA[ e.speech=='end' ]]>
      </condition >
      <action name=
        "trTalkingWaitForTalker$rule3Action" />
    </rule>
  </transition>

```

Figure 9. The XML File

Path expressions are passed to Path Expression Execution Engine, and operations taking path expressions as arguments perform proper actions at runtime. Those behaviours are described in the translated Java codes with PLUE API calls. For instance, ‘assignExpression()’ is for the assignment operation with path expressions, and the usage in the translated java codes is in Fig. 10.

```

public void stWaitforTalker$assertECA1(){
  assignExpression("$.platform.light", "=", "true");
}

```

Figure 10. stWaitforTalker\$assertECA1 method

Loader module registers states and transitions of the task to State Transition Machine for the task. State Transition Machine executes the entry phrases of the initial state of the task, and run the state transition machine until it meets the exit phrase of the final state. When an event occurs, the actions of matched transitions or ECA rules will be performed.

5. Future Work and Conclusions

Traditional programming languages are not sufficient for ubiquitous application programming due to the lack of the features for management of various types of data and dynamic changes of situation in real world. However, relatively less interests are given on the programming paradigms for ubiquitous environments.

This paper suggests an object-oriented language named ‘PLUE (a Programming Language for Ubiquitous Environments)’, which will help programmers write ubiquitous applications. PLUE supports ECA (event-condition-action) rules and FSA (finite state automata)-based interactive responses to dynamic situations. It also allows manipulating UDM (Universal Data Model) data in forms of conventional path expressions. We expect that PLUE

allows rapid prototyping of ubiquitous applications and eventually helps ubiquitous dreams coming true.

Although PLUE is currently deployed on top of CAMUS (a Context-Awareness Middleware for URC Systems), the ubiquitous middleware system running on Ubiquitous Dream Hall [5], it can be ported on other ubiquitous middleware systems in a trivial way.

6. References

- [1] Diego Lopez de Ipina, "An ECA Rule-Matching Service for Simpler Development of Reactive Applications", IEEE. Distributed Systems, Vol. 2, (2001).
- [2] Mark Weiser, "Some Computer Science Problems in Ubiquitous Computing" Communications of the ACM, July 1993.
- [3] <http://www.w3.org/TR/xpath>
- [4] <https://javacc.dev.java.net/>
- [5] <http://www.ubiquitousdream.or.kr/>