Scenario-Based Programming for Ubiquitous Applications

Eun-Sun Cho[1,*], Kang-Woo Lee[2], Min-Young Kim[2], and Hyun Kim[2]

[1] Dept. of Computer Science & Engineering, Chungnam National University
220 Gung-dong Yusong-gu, Daejeon, Korea, 305-764
eschough@cnu.ac.kr
[2] Electronic and Telecommunications Research Institute,
161 Kajong-dong Yusong-gu, Daejon, Korea, 305-350
{kwlee, hkim}@etri.re.kr, tristan88@hanmail.net

Abstract. Ubiquitous applications usually involve highly interactive context data management. Traditional general-purpose programming languages are not sufficient for use in this domain, as they do not have the capability to manage such data effectively. We have developed a scenario-based programming language that we call 'PLUE (Programming Language for Ubiquitous Environment)', which is a Java-based prototyping language for ubiquitous application development. PLUE supports ECA (event-condition-action) rules and finite state automata-based (FSA-based) interactive responses to dynamic situations. In addition, PLUE programmers are able to manage heterogeneous data with a uniform view of path expressions. We have implemented PLUE on top of CAMUS (Context-Aware Middleware for Ubiquitous Robotic Companion System), a framework for context-aware applications that was originally developed for network-based robots.

1 Introduction

Ubiquitous computing systems must handle context data from various sources, including mobile devices and sensors. These systems usually have services that are widely distributed over networks and devices. However, they must also be highly interactive with their surrounding environments.

Since they run on top of such systems, ubiquitous applications usually consist of a number of complicated commands that handle various kinds of context data. Well-designed event handling is necessary to cope with changing situations [1, 2]. Transforming scenarios that are conceived by designers into programs in C or Java takes far too long using currently available tools.

This paper describes our new language, PLUE (A Programming Language for Ubiquitous Environments), a prototyping language for ubiquitous applications. PLUE was developed in accordance with the following preliminary attributes:

* This work was supported in part by MIC & IITA through IT Leading R&D Support Project.

H.Y. Youn, M. Kim, and H. Morikawa (Eds.): UCS 2006, LNCS 4239, pp. 286 – 299, 2006.

- Handy structures for describing situation flows: a designer must always visualize the scenario of a ubiquitous service before he can proceed with the development of applications. As an example of such a scenario, consider a speaker giving a presentation at a conference. The room lights would automatically dim at the beginning and a slide show would start once the speaker has begun his talk. This will happen repeatedly for each speaker, but only until the last speaker has finished. Therefore, the whole task forms a sequence of actions with patterns and can be represented as a kind of finite state automaton. PLUE supports the description of the flow of such scenarios through its state transition facilities.
- Transparent context management: since context data usually forms a tree in a ubiquitous environment, PLUE provides path expressions for context access. The path traversal is transparent using the usual dot notation of member access in object-oriented languages. It may involve automatic context data matching provided by any third-party services. To manage context data from diverse sources uniformly, we have defined a minimal data model called the 'UDM (Universal Data Model)'. PLUE offers path expressions for UDM data access.
- Prompt responsiveness: prompt reaction to their surrounding environments is important for ubiquitous applications. PLUE allows programmers to generate ECA rules (event-condition-action) in order to encode the necessary actions in a straightforward manner.
- User-friendly language facilities: we assume that most programmers do not want to waste time learning a new language. To achieve user friendliness, PLUE is based on Java, a popular language. A PLUE program has very little dependence on its underlying system because the transparent distributed object invocation hides the complex system architecture. With simple event-handling features, interactions between the dynamic situations can be described easily.

Our current version of PLUE is implemented on top of CAMUS (Context-Aware Middleware for URC System) [3], a framework for context-awareness applications. It was originally developed for a network-based robot infra-system called 'URC (Ubiquitous Robotic Companion)' that requires software infrastructure to enhance the intelligence and context-awareness of network-based robots. CAMUS has been successfully deployed in 'Ubiquitous Dream Hall' [4] as a middleware to demonstrate future ubiquitous cities.

Fig. 1. Ubiquitous Dream Hall: the living room of u-Home (left) with a home-care robot (middle), a girl kicking imaginary balls at u-Street (right)

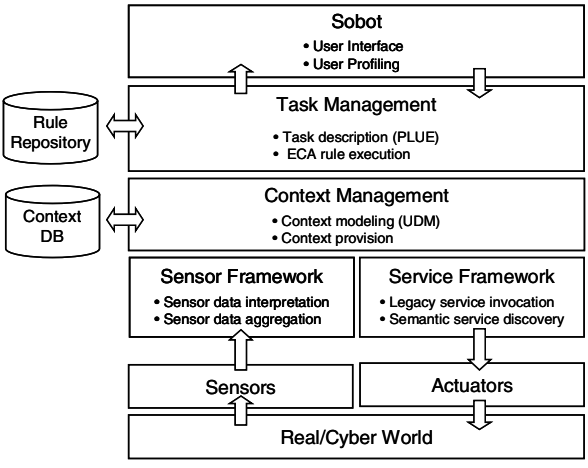

Fig. 2. CAMUS Architecture

Fig. 2 shows an abstract architecture of CAMUS. CAMUS receives the information from sensors installed in the real world. The information is delivered in the form of events that are in turn fed into the 'Sensor Framework'. The Sensor Framework gathers the events and interprets/aggregates them to generate high-level context information that is then sent to the 'Context Manager' in UDM form. Any change of context information triggers the Context Manager to transfer corresponding events to the 'Task Manager'. When an event is raised, the Task Manager searches for task rules that are interested in the event and invokes the rule action if it satisfies the condition. While performing an action, the task invokes services that make changes on the real/cyber world.

In the next section, we present a brief survey of related work. Section 3 shows how to manage context data in PLUE programs. Section 4 introduces the task description mechanism in PLUE, with ECA rules and the state transition mechanism. Section 5 shows the underlying architecture of PLUE. Finally, section 6 includes a discussion of our work and our conclusions.

2 Related Works

Compared to other topics in ubiquitous systems, the design of new tools and languages that can be used to help develop programs have not been actively pursued by the research community. In this section, we will skip RMI (remote method invocation)[5], XPath [6], and ECA (event condition action) rules [7]. While they are definitely related to our research, they are adequately described elsewhere.

In RCSM [8], and Salsa [9], script languages based on ECA rules are used to describe the behavior of agents. The behavior of an agent consists of a set of ECA rules that permit prompt responses to context changes. Such languages are useful for non-centralized agent systems, where task execution must necessarily consist of both communication between agents and the execution of the agents. However, this

approach is too platform dependent, since script languages are too weak to describe sizable programs in other than an autonomous agent platform. This is especially true in centralized controlled systems that require the description of entire tasks.

COP (Context Oriented Programming) [10] from the University of Queensland is based on ambient calculus [11]. It is widely known for its theoretically appealing context management mechanism using dynamic scoping variables for context data. A COP program executes self-adjustable behaviors as the current context is changing, while also focusing on context matching and dynamic function binding. Despite the excellent manner in which it can manipulate context and context-dependent code fragments, COP does not concentrate on the task description. It also lacks the capability to handle scenarios and events.

One world [12], one of the famous pervasive computing systems developed at the University of Washington, provides a programming model with nested 'environments' each of which contains 'components' and 'tuples'. Applications are composed from components that exchange events. The flow of control for the application consists of several instances of event handling. Remote event passing, operation migration, object sharing, and querying are also supported, all of which can be designed using Java API's without transparency. This technique was founded on the premise that since ubiquitous computing has so many context changes, it would be dangerous for them to be hidden from programmers. Given this insight, programmers would then be able to handle them correctly, with their own scenarios. However, this is not necessarily true since it is likely that such detailed implementation and maintenance information would distract programmers from the task at hand. This well-known proposition has been proven true several times in the past in the field of software engineering. In addition, the language provided by one world is not suitable for prototyping.

'Olympus [13]', a high-level programming model suggested by University of Illinois at Urbana Champaign, allows a developer to specify entities and operations in ubiquitous computing environments at abstract level. Despites its entity abstraction facilities and excellent service discovery scheme, Olympus lacks support for task description based on the flow of scenarios.

3 Context Data Access

3.1 Universal Data Model

Since context awareness is one of the key characteristics of ubiquitous computing, several studies examined the management of context information in ubiquitous applications. To date, however, no satisfactory context data model for heterogeneous data management has been found. Ontology represented in logic would be our choice, but is hardly considered practical so far.

We have defined a new simple context data model for context information that we have called UDM (Universal Data Model). It is similar to OEM [14], a semi-structured database model, in that context information is denoted as an edge-labeled graph. Although we developed UDM for PLUE, it can be used for other ubiquitous programming environments. Its main concepts are described below.

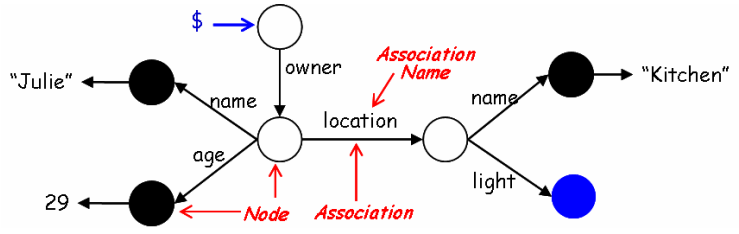

Fig. 3. Context data for the SmartRoom Application

- **Nodes:** Each node in UDM represents an entity, such as a person, place, task, service, and so on.
- **Associations:** An association is a labeled directed edge. It presents the relationship between nodes. Some associations change with time. In Fig. 3, the location of a person varies whenever he/she moves to a different place.

Fig.3 is an example of a simple UDM model that shows that "29-year-old Julie is in a kitchen that has a light fixture". As you can see, UDM provides a very easy way to describe the context.

PLUE provides facilities to access context data represented in UDM. PLUE programs can handle any data from various sources in a uniform format. 'UDM bindings' provide a transparent view of the context data from heterogeneous data sources. When a new data source is added, a corresponding UDM binding should also be supported to map from the native data into a UDM view and vice versa.

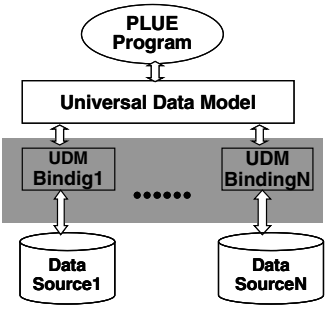

Fig. 4. Context data integration using UDM bindings

Fig. 4 shows the data flow among data sources, UDM bindings, and PLUE programs. Our current version of PLUE provides built-in bindings for XML documents, Unix/Linux/Windows file systems, JavaBeans, and annotated plain Java objects.

3.2 Path Expressions

PLUE supports path expressions to retrieve and modify the context data modeled in UDM. A path expression is a sequence of association names starting from a node. By evaluating a path expression, we obtain a set of nodes that are reached through the

sequence of associations beginning at the starting node. In PLUE, the starting node is denoted by the '$' symbol. It points to the root context node for a PLUE application. For instance, in Fig. 3, the path expression "$.owner.location.name" points to the valued node whose value is "Kitchen". The meaning of the expression is "the name of the place where the owner is currently located."

Using path expressions, a PLUE application is able to reach all the necessary context information, such as the name of the person who executes this application, the address of the owner's location, and other application-specific data.

PLUE supports a set of sophisticated path expressions to query the interesting data. These expressions are highly expressive, but they are still simple and easy to describe. We categorize them as shown in the following examples:

Basic path expressions
A UDM node may have more than one departing edge with the same tag. A path expression will return multiple values in that case.

- $.owner.location: is the single value for the location
- $.owner.location.residents: is a set of multiple values when multiple residents are present in a location.
- $.owner.location.'temporal residents during a month'.id: is the id of the multiple temporal residents that are present in a location. (a quoted string in an association name is useful for a long sentence with spaces)

Selective association traversal
PLUE assigns orders sequentially for departure edges with the same tag that leave from a single UDM node. Specifically, the ordinals or the ranges of ordinal numbers can confine a multi-valued path expression. The following examples assume a multi-valued association 'children' from the location of the owner.

- $.owner.location.children[2]: Of the multiple associations tagged with '**children**', the second one is selected.
- $.owner.location.children[2-4]: Of the multiple associations tagged with '**children**', the second, third, and fourth are selected.
- $.owner.location.children[4-]: Of the multiple associations tagged with '**children**', the ones whose ordinal numbers are larger than four are selected.
- $.owner.location.children[1,4-5,9]: Of the multiple associations tagged with '**children**', the ones whose ordinal numbers are 1, 4, 5, and 9 are selected.

Conditional path traversal
Selection on multiple associations with a shared name is done based on matching values, as is done in database queries. When a comparison is made between multiple values and a scalar or between two sets of multiple values, the condition will be true for at least one case.

- $.owner.location.residents[.name=='Tom']: retrieves people who are at the same place as the owner and whose names are 'Tom'.
- $.owner.location.residents[.name=='Tom' && .age=='10]: retrieves 10-year-old Tom from among the residents who are at the same place as the owner.

Wildcards
Traversing an anonymous edge is denoted by a wildcard ('*') in a path expression. '**' denotes any number of anonymous edges. This enables edge selection even when the programmer does not know the exact path. '%' is used for a wild card character in an association name.

- $.owner.location.*.id: the nodes that are reachable from $.owner.*some_anonymous_link*.id.
- $.owner.**.id: the nodes of the set of 'id' links that are reachable from $.owner
- $.owner.location.pa%.id: the nodes of the set of 'id' links that are reachable from $.owner.location and links whose names are prefixed with "pa".

Built-in functions
Our current version of PLUE supports two built-in functions that transform multiple values to a single value.

- exists($.owner.location.parent): true if the cardinality of $.owner.location.parent is not 0.
- count($.owner.location.parent): the cardinality of $.owner.location.parent

4 Task Description

4.1 State Transitions

Ubiquitous computing applications, like other user-centric services, are usually designed based on preconceived corresponding scenarios. For instance, a smart conference room would be built for a scenario where conferences are managed automatically. The flow of an example scenario is as shown in Fig. 5.

In PLUE programs, all of the work that must be done for a scenario is called a '*task*'. For programmers, a PLUE task is much like a Java class with methods, instance variables, and inner classes. Since a task forms a sequence of actions with patterns that are a kind of finite state automaton, a task definition is augmented with a flow description using '*states*' and '*transitions*'.

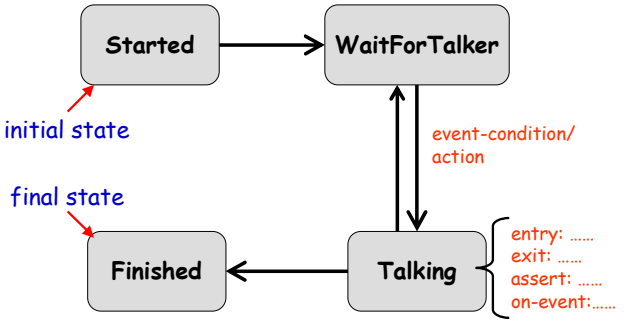

Fig. 5. The flow of a smart conference room scenario

A '*state*' has a duration for which some property in the context must remain constant. A PLUE state consists of four parts – *the entry*, *the exit*, *asserts*, and *on-event-rules*. The *entry* part initializes the state. The variables used in the state are initialized and resources are prepared. *Assert* phrases are conditions that are true for the duration of the current state of the task.

On-event-rule phrases describe the actions to be taken when specific events occur. The *exit* part describes the work that must be done before the task transits to other states. The keywords 'initial' and 'final' are attached to the initial and final states, respectively. Currently, we have implemented only one state for each. The entry and exit phrases can be omitted if no action needs to be specified while entering/exiting the state. A state can have multiple asserts and on-event rules.

Fig. 6 depicts a code fragment of a PLUE program, showing that the state 'Talking' describes a situation where a speaker is giving a talk in a conference. The entry part of this state prepares the presentation foils for him on the screen in the conference room. If any transition occurs, then the exit phrase sets the current speaker to the next speaker. The assert phrase ensures that the platform light is turned off while the speaker is talking.

```
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();}
}
```

Fig. 6. An example of a state definition: state 'Talking'

If the speaker says "*next*", then the next slide is shown. Note that a task writer can assign event variables to their target event names. These variables can then be used in on-event rules. In the example, the event variable "e" is bound to an event named "VoiceReceived". PLUE supports seven built-in event types: TagEntered, TagLeft, UserEntered, UserLeft, SpeechReceived, PropertyChanged, and TimeExpired. However, task writers could add any new event types that would be necessary to develop their tasks. The 'condition' clause expresses the condition for which a rule is invoked. Path expressions, the event variable, and the usual Java comparators can appear as well. The braced body denotes the action part of the on-event rule.

A '*transition*' describes the actions taken during the transition from the current state into the next state when the event satisfying the corresponding conditions occurs. The transition consists of a from-state, a to-state, the 'on-event/condition', and the action. The 'on-event /condition' includes the transition condition, as well as what must be executed during the transition. This is similar to the on-event rules for a state.

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

Fig. 7. An example of a transition definition: from 'WaitForTalker' to 'Talking'

Fig. 7 describes the transition from the WaitForTalker state to the Talking state. If the chairman says "*Start presentation*", then the data for the first speaker is prepared and a TTS (Text-To-Speech) system says "*Start*" to those present while the transition is made.

A PLUE task is modularized into 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 in the transition. Since massive events and fluctuating situations in ubiquitous environments can be directly modeled in PLUE, it is easier for the programmers to write and maintain application programs for ubiquitous computing.

Fig. 8 depicts a part of a task with two states and a transition. 'Rule-based programming' for proactive services in context-aware applications is achieved by using on-event rules and asserts in states and transitions in PLUE. On-event rules and transitions are only invoked when the required events are received. The actions of assert phrases are invoked when the required conditions occur. Path expressions are used extensively to express such ECA rules and asserts.

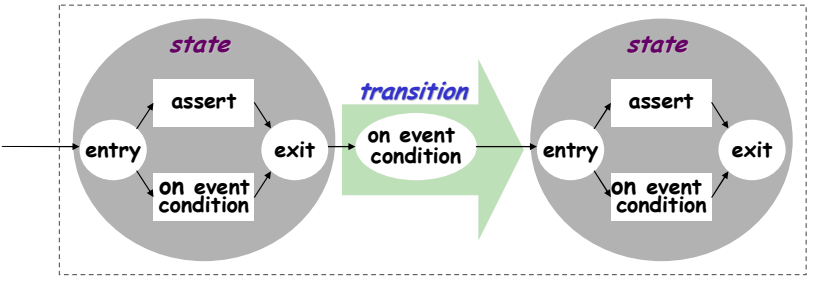

Fig. 8. Transition of states while a task is running

4.2 Context Management

Context information gained from path expressions is manipulated extensively in the entry/exit phrases and rule descriptions that are contained in a task definition. Such information is usually calculated with other values and modified by assignment in a C-like program. In addition, it is more difficult to handle a path expression that returns a multi-value like a set, a list, and a bag. PLUE supports tools that make it easier to process path expressions.

- <*path expression*> <*op*> <*scalar value*>
- When <*path expression*> results in value nodes, PLUE allows basic operations to be conducted on the path expression. Our current version of PLUE supports +,-,*, and / with integer values, and + for string concatenation. If <*path expression*> has multiple values, each value will undergo an operation <*op*> with a <*scalar value*> that will then be merged into the resulting set. These expressions, simple path expressions, and values are collectively called 'complex path expressions'.
- <*path expression*> = <*complex path expression*>
- When <*path expression*> results in a value node, it will update the value. If <*path expression*> has multiple values, the right side should have multi-values. All of the departing edges of the resulting nodes of <*path expression*> will be replaced with the new ones.
- <*path expression*> <= <*complex expression*>
- The result of <*complex expression*> will be added as departing edges to the resulting nodes of <*path expression*>.
- <*path expression*> = nil
- PLUE supports a special keyword nil to clear the departing edges of the resulting nodes of <*path expression*>.
- foreach (<*var*> = <*path expression*> : [<*condition on the var*>]) <*action on the var*>
- The foreach statement can be used as an iterator for multiple edges. <*var*> is the usual variable name that is prefixed with a '$' for discrimination with other Java variables. In the above example, since the location 'e.location' has more than one light, the variable $light points to each element of the set of lights. If the light that is referenced by the variable $light is off, ('light.power==false'), then action will be taken on it.
- foreach <*var*> in (select <*path expression*> from <*path expression*> where <*path expression*> <*op*> <*complex path expression*>)
- Like object database query languages [15], a 'select-from-where' clause is provided in this special foreach-statement. Since a select-from-where statement can also be expressed with the usual path expressions, programmers can choose not to use this version of a foreach-statement.

This language extension is preprocessed into a plain Java program before compilation. Every new feature is similar to a plain Java expression and would therefore be familiar to Java programmers. The following example shows an on-event rule in PLUE that ensures that at least one light is on whenever a person enters the kitchen. A variable for a path expression ('$light') and a Java variable ('flag') both appear in the foreach statement.

```
on event UserEntered(e)
condition ( e.platform.name == 'Kitchen') {
   boolean flag = false;
   foreach ($light=e.platform.light:
              $light == false && flag == false) {
      $light = true;
      flag = true;
   }
}
```

5 Implementation

A PLUE task program developed with a context data model is basically put into a PLUE Preprocessor and translated into Java code and an XML file. Rules are translated into a code fragment that generates rule objects and registers them with the Rule Processor. The path expressions are converted into a composition of appropriate API calls to the underlying Path Expression Processor. As a result, any Java compiler can compile the generated Java code into Java byte code.

The loader module creates a task object from the Java byte code and the XML file and then it delivers it to the State Transition Machine dedicated to the task. Fig. 9 depicts the processing flow of a PLUE task.

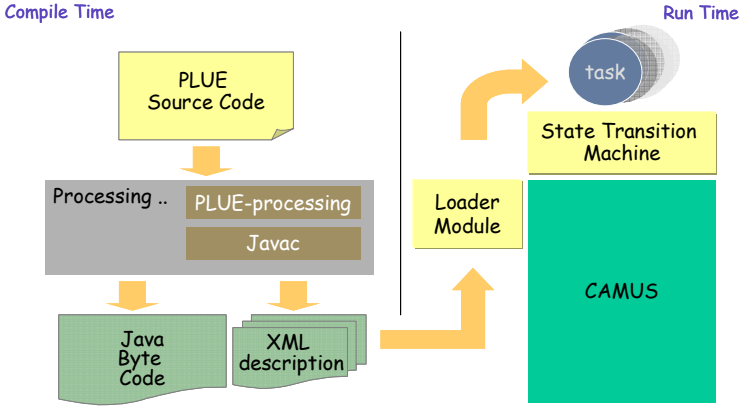

Fig. 9. PLUE architecture

```
<?xml version="1.0" encoding="EUC-KR"?>
<task name= "ConfAssistant" >
      …
<state name = "Talking"  >
   <entry name= "stTalking$entry" />
   <exit name= "stTalking$exit" />
   <assert name = "stTalking$assertECA1" />
   <rule name = "rule0" >
     <event name = "VoiceReceived" />
     <condition >
       <![CDATA[ e.speech=='next' ]]>
     </condition >
     <action name = "stTalking$rule0Action" />
   </rule>
</state>
<transition from = "Talking" to = "WaitForTalker" >
   <rule name = "rule3" >
     <event name = "VoiceReceived" />
     <condition >
       <![CDATA[ e.speech=='end' ]]>
      …
<action name ="trTalkingWaitForTalker$rule3Action"/>
</rule>
</transition>
```

Fig. 10. Translated XML document for the state ‘Talking’ (in part)

The PLUE Preprocessor, based on JavaCC 4.0 beta 1 [16] for parsing, inputs a PLUE program and outputs new Java byte code and an XML file. Each tag in the XML file corresponds to a PLUE language feature, such as '<task>', '<state>', '<entry>', '<exit>', and '<assert>'. The 'name' attribute of '<task>' or '<state>' represents the name of the task (or the state). The name attribute of '<entry>', '<exit>', '<assert>', or '<action>' is for the method name in the Java byte code that describes the corresponding behaviour. Fig. 10 is an example of the XML file that is generated for the state 'Talking' that was introduced above.

At run time, the Loader Module registers the states and transitions of the task with the State Transition Machine for the task. The State Transition Machine executes the entry phrases of the initial state of the task and runs the state transition machine until it encounters the exit phrase of the final state. When an outside event occurs, the Rule Processor selects the related on-event rules or transitions in the task by checking the on-clause and the condition-clause of the rules. While the action part of the matched rule is being executed, it interacts with the Path Expression Processor and the remote services, as shown in Fig. 11.

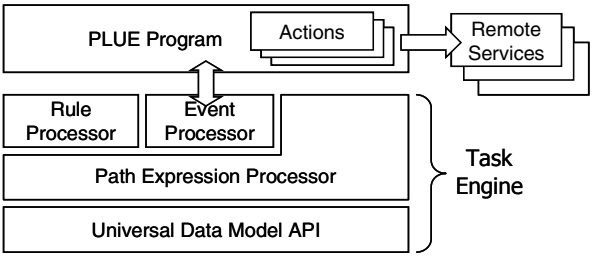

Fig. 11. Interactions with CAMUS

This approach with XML file generation achieves a kind of separation of concerns between PLUE Preprocessor and State Transition Machine, but imposes runtime overhead. As an alternative, we are currently developing Java API for registering and loading tasks directly from the translated PLUE program, which enables getting rid of the intermediate XML description.

6 Discussion and Conclusions

Traditional programming languages are not sufficient for ubiquitous application programming because they cannot manage the various types of data and dynamic changes that occur in real-world scenarios. However, relatively less interest is given to programming paradigms for ubiquitous environments.

This paper describes an object-oriented language named 'PLUE (a Programming Language for Ubiquitous Environments)' that will help programmers write ubiquitous applications. PLUE supports ECA (event-condition-action) rules and finite state automata-based (FSA-based) interactive responses to dynamic situations. It also allows the manipulation of UDM (Universal Data Model) data in the form of conventional path

expressions. We expect that PLUE will facilitate the rapid prototyping of ubiquitous applications and will help to accelerate the deployment of ubiquitous computing.

Some commonly used concepts for data processing are reflected in the context management of PLUE. The idea for the powerful, but succinct, path expressions in PLUE comes from the query languages for XML [6]. Select-from-where clauses in PLUE are similar to query languages for OODBMS [13].

Although PLUE provides powerful expressions to develop context-aware applications, the preprocessing based extension to a common language may give rise to complications in using an IDE (Integrated Development Environment) tool, such as Eclipse or JBuilder. To overcome this limitation, we have also provided an annotation-based rule description. Based on Java 1.5 program annotations and the APT tool (annotation processing tool) [17], task rules are translated to work as normal Java code.

PLUE is now deployed on top of CAMUS (a Context-Awareness Middleware for URC Systems), the ubiquitous middleware system running in the Ubiquitous Dream Hall [2]. However, any PLUE applications could be ported on other ubiquitous middleware systems in a straightforward manner.

We are currently extending the expressiveness of the path expressions to handle ontology data derived from an external OWL resource and exploring methods of extending the state transition mechanism for more complex ubiquitous applications.

References

1. Wang Z, Garlan D, Task-Driven Computing, Technical Report, CMU-CS-00-154, School of Computer Science, Carnegie Mellon University, May 2000
2. Banavar G, Beck J, Gluzberg E, Munson J, Sussman JB, Zukowski D. Challenges: an application model for pervasive computing. In *Mobile Computing and Networking*, pages 266-274, 2000
3. H. Kim*, Y.-J. Cho*, S.-R. Oh, CAMUS: A Middleware Supporting Context-aware Services for Network-based Robots, In Proc. of IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO '05)
4. Ubiquitous Dream Hall, http://www.ubiquitousdream.or.kr/, 2005
5. Java Remote Method Invocation (Java RMI), http://java.sun.com/products/jdk/rmi/
6. W3C XML Query (XQuery), http://www.w3.org/XML/Query, 2005
7. Lopez de Ipina D, An ECA Rule-Matching Service for Simpler Development of Reactive Applications, *IEEE. Distributed Systems*, Vol. 2, 2001
8. Yau SS, Karim F, Wang Y, Wang B, Gupta SKS. Reconfigurable Context-Sensitive Middleware for Pervasive Computing, *IEEE Pervasive Computing*, Vol. 1, Issue 3, July 2002
9. Rodríguez M, Favela J, Preciado A, Vizcaíno A. An Agent Middleware for Supporting Ambient Intelligence for Healthcare, In Proc. of ECAI 2004 Second Workshop on Agents Applied in Health Care, Aug 2004
10. Rakotonirainy A. Context-Oriented Programming for Pervasive Systems, Technical Report, University of Queensland, Sep 2002
11. Cardelli L, Gordon AD. Mobile ambients. *Theoretical Computer Science*, 240(1): 177-- 213, 2000

12. Grimm R, Davis J, Lemar E, MacBeth A, Swanson S, Anderson T, Bershad B, Borriello G, Gribble S, Wetherall D. System support for pervasive applications. *ACM Transactions on Computer Systems*, 22(4):421-486, Nov 2004
13. Ranganathan A, Chetan S, Al-Muhtadi J, Campbell RH, Mickunas MD. Olympus: A High-Level Programming Model for Pervasive Computing Environments, In Proc. of International Conference on Pervasive Computing and Communications (PerCom 2005), Kauai Island, Hawaii, March 8-12, 2005
14. Papakonstantinou Y, Garcia-Molina H, Widom J. Object exchange across heterogeneous information sources. In Proceedings of IEEE International Conference on Data Engineering (ICDE), pages 251--260, Taipei, Taiwan, Mar 1995
15. Cattell RGG, Barry DK, Berler M, Eastman J, Jordan D, Russell C, Schadow O, Stanienda T, Velez F, The Object Data Standard: *ODMG 3.0*, ISBN 1-55860-647-4, Academic Press, 2000
16. JavaCC Home, https://javacc.dev.java.net/, 2004
17. Annotation Processing Tool (apt), http://java.sun.com/j2se/1.5.0/docs/guide/apt/, 2004