World Academy of Science, Engineering and Technology
International Journal of Information and Communication Engineering
Vol:1, No:1, 2007

Rule-Based Message Passing for Collaborative Application in Distributed Environments

Wataru Yamazaki, Hironori Hiraishi, and Fumio Mizoguchi

***Abstract*— In this paper, we describe a rule-based message passing method to support developing collaborative applications, in which multiple users share resources in distributed environments. Message communications of applications in collaborative environments tend to be very complex because of the necessity to manage context situations such as sharing events, access controlling of users, and network places. In this paper, we propose a message communications method based on unification of artificial intelligence and logic programming for defining rules of such context information in a procedural object-oriented programming language. We also present an implementation of the method as java classes.**

***Keywords*— agent programming, logic programming, multi-media application, collaborative application**.

I. Introduction And Motivation

With the recent speeding up of the computer network, rapid development of high-speed devices, and commoditization of our personal computers, our daily-use computer software needs to perform more advanced and complex processing. For example, many ubiquitous or GRID-enabled applications need support for dynamic context-aware situations. In such environments, the software must manage many IP addresses of multiple users in a dynamic network, QOS of applications, access control of users, and so on.

To achieve such management, distributed computers must communicate with each other so as to meet these system constraints. Collaborative software for multiple uses in distributed places is one of the most extreme examples of software that needs such complex communications. For developing such software, developers must design numerous communication protocols and have to implement these protocols correctly. However, developers currently must use extremely low-level APIs for implementing network communications. Most current general methods for message communication in a computer network employ Socket communication. In this method, however, we must manage a byte queue even for simple event dispatching. New communication methods like Remote Procedure Call (RPC), XML-RPC, and Object Request Broker (ORB) can hide low-level byte sequences, but developers still have to manage troublesome procedural processing such as string processing to manage context-aware programming, and these processes are difficult for the software developer. To define context-aware programming, we can use pattern-matching methods. The most general and widely spread pattern-matching method is "regular expression," which is used in most common programming languages. However, regular expression simply checks if the character sequence of a target string matches that of pattern strings, so developers still have to mange procedural processing that is not strongly related to the application logic, even for getting single argument from the coming event. When we use pattern matching for XML document, we can use the Document Object Model (DOM) [1] API or database management query languages such as XQuery [2]. To use these methods, however, developers still need to manage the same procedural processing.

Unification, which is the one of most attractive aspects of logic programming languages, is used infrequently compared to regular expression, but it is a simple and powerful pattern-matching method. In unification, patterns can be seen as rules defined in declaratory statements. Indeed, prolog programming language, which supports unification for execution in clauses, can easily define state transition rules [3]. Unification is a powerful method, but at this moment, few systems support unification for procedural programming languages.

In this paper, we design and implement a system that can support unification-based rule defining and rule execution for message communication in a procedural object-oriented programming language. We also show an example of a video-conference application that was implemented by the proposed method. By using the proposed systems, the developer will be able to declaratively define rules of complex message communication and suitable application behavior.

The remainder of this paper is structured as follows. In Section 2, we briefly present the target application for our proposed method. In Section 3, we describe the design approach and give an overview of our proposed rule-based method. In section 4, we then demonstrate the implementation of the java library that supports unification-based rules for message communications. Section 5 describes related works and presents our conclusion.

Authors are with the Information Media Center, Tokyo University of Science, 2780-1 Yamazaki, Noda, Chiba, Japan. (e-mail:yamazaki@imc.tus.ac.jp).

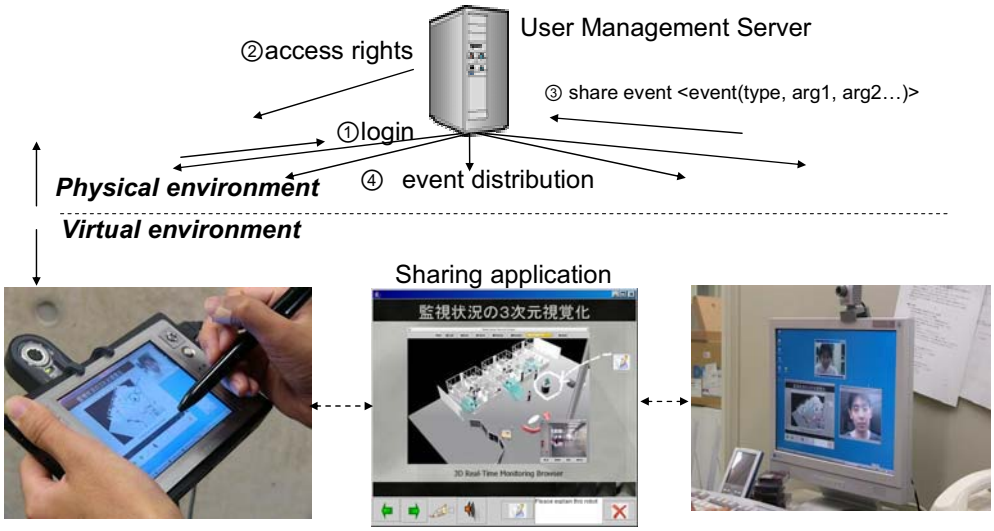

Fig. 1 Target Application

II. Target Application

We now introduce the target application that was implemented using the proposed library. The application is a collaborative software environment with an integrated video-conference application, and resource sharing application such as presentation application, web browser and text editors. Figure 1 gives an overview of the application. When one authorized user sends a control command to the system, the command is shared by all (or enrolled) users. In the user's view (virtual environment), the GUI and its operations are all allowed operations. Illegal operations and terminal-dependent operations are not shown to the user automatically. To realize such a transparent virtual environment, the physical (system) environment should manage a large volume of message communications of many kinds.

In Fig. 1, a presentation application is shared by multiple users. Some events, such as controlling slides and adding annotation generated by a user are shared by appropriate users. The sharing applications share states of the application, not the graphics of the application. (Of course, graphics sharing is also realized by the sharing state of the application.) Therefore, not all events need to be shared by users. Here, shared events are saved by an event log. By accessing this event log, for example, users who did not attend a meeting in real-time can also learn about the progress of the meeting. Automatic proceeding generation is another possible application using the event logs.

For the above application, the main message communications illustrated in Fig. 1 are as follows. 1. A user logs in to the system. 2. A server gives the user access rights. 3. A user sends shared a event to the system. 4. The system multicasts the event to suitable users. Each message contains at least the sender name, the message address, command, and arguments of the command. The message content may thus differ considerably.

As we mentioned in the previous section, developers should design and implement complex protocols properly, but this is difficult in conventional procedural languages. The primary reason for this problem is that designing a protocol is a deductive process, but the implementing program language does not support direct deductive programming. Starting in the next section, we will demonstrate how to adopt the deductive defining to procedural programming languages by focusing on the message communications.

III. Approach

Generally, pattern matching for a communication system consists of the following three parts. 1. The inner state of the system when the system received the message. (Inner state) 2. The kind of message the system received. (Message type) 3. The kind of executions to be performed when the system received the message. (Execution and rewrite the state) Figure 2 illustrates this situation using state transitions.

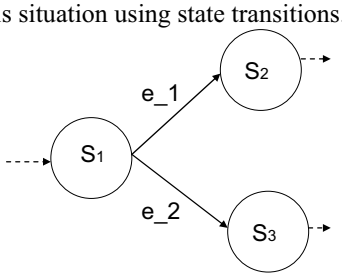

Fig. 2 Message communications by state transition

Figure 2 indicates that when the system's state is S1 and the system get the message e_1, then the state will change to S2, and that when the system's state is S1 and the system receives the message s_2, the system changes its inner state to S3. Here, some executions generally occur before the state transition. These state transitions can be easily defined by commit-choice parallel logic programming languages such as GHC [4].

run(S1) :- mes(e1) | exec_1…exec_n, run(S2).
(Rule1)

run(S1) :- mes(e2) | exec_1…snd(mes), run(S3).
(Rule2)

In GHC, the part to ":-" is called the head, the part from ":-" to "|" is called the guard, and the remainder is called the body. As shown in rule 1 and rule 2, the events are located in the guard part. The rules are checked in parallel, and the body part is executed only when head matching (run predicates with argument S1 and S2 in the example rules) and guard matching are successful. The guard part can consists of multiple predicates. In that case, the body will be executed only when every predicate is successful. Therefore, predicates in the guard portion cannot have any side effect because when one predicate in the guard part fails, the side effect cannot be canceled by the system. For example, in the guard part, the state of message may be checked, but the message cannot be removed until all predicates in the body succeed. Also, once the guard part successfully executes all its predicates, all predicates in the body part must succeed. In GHC, a state transition can be defined by using recursive calling. As rule1 and rule 2 demonstrate, message communication can be defined and implemented deductively by state transition and logical unification.

IV. Rule-Based Message Passing

A. Design

Figure 3 presents an overview of the method of adapting a GHC-like rule base to an object-oriented procedural programming language.

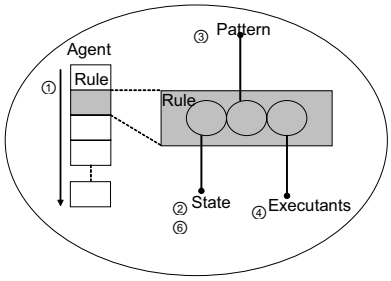

Fig. 3 Architecture of Agent

In our system, the component that contains the list of rules and that manages the pattern matching is called the Agent. Each rule consists of three parts mentioned in the previous section, namely, inner state (corresponding to the head in GHC), pattern (corresponding to the guard part in GHC), and executing objects (corresponding to the body part). To realize these components in object-oriented programming, we need the following objects and functions as a minimum.

- **Agent object** contains the list of rules and checks if each rule is executable.
- **Rule object** is the rule checked by the agent and consists of three parts: the inner states, the message pattern, and the executing target.
- **Term object** is an abstract object for the logical term and is used for defining the pattern to try unification of the pattern and the coming message term.
- **State-checking interface** is the interface for inner state matching and is registered to the agent so the agent can call this method.
- **Message pattern** is defined as logical term and is registered to the rule for the pattern matching.
- **Body interface (shown as executants in Fig. 3)** is the interface corresponding to the body part in GHC, and these objects are executed after the above unifications. Sending message and state rewrites are examples of implementing this interface.

Assuming the above objects and functions, we can define and execute the rules as follows. The step numbers in the following description correspond to the numbers in Fig. 3.

[step 0] Initialize the applying rule index "i" to 0 (i:=0) and go to step1.
[step 1] The agent selects the "i" th rule (rules[i]), and goes to step 2.
[step 2] The agent checks the inner state by calling the registered method. If the check succeeds, go to step 3. If not, and if i is r (r is the size of rule list), i:=0 and go to step 1, if i is not r, i:= i+1 and go to step 1.
[step 3] If the incoming message (top of the queue) and registered term (pattern) are unified, then go to step 4. If not and if I is r, i:=0, and go to step 5. If i is not r, i:=i+1, and go to step 1.
[step 4] The agent executes the method of the registered body interface. Message sending and state rewrite are performed in this step.
[step 5] The agent removes the coming message (top of the message queue).

Step 5 is not shown in Fig. 3, because the step is subordinate, but step 5 is important. If the agent is in step 5, it means that none of the rules in the agent matched this message at that moment. Therefore, if the agent leaves the message in the queue, the agent will never be able to change state.

B. Form of defining rules

```
01  public class TestAgent extends Agent{
….
02     public TestAgent(String name){
03        super(name);
04        StateListener st1 = new StateListener(){
             public boolean stateChanged(){return foo();};};
05        Executant el1 = new Executant(){
             public void execRule(Message m){bar(m);};};
06        Rule r = new Rule(st1,"event(Name, No,slide(X)))",el1);
07        addRule(r);
08     }
09     public boolean foo(){
```

```
10      return controllable;
11    }
12    public void  bar(Message m){
13      Message smes = new Message("ok(Name,No2)");
14      smes.substitute("Name",getID());
15      smes.substitute("No2",m.get("No"))
16      m.getAgent().sendMessage(smes);
17    }
17  }
```

Fig. 4 Form of defining rules

Figure 4 illustrates the usage of the java implementation of the rule-based pattern-matching library. In Java programming language, event handling and its interface execution can be seen as the same kind of deductive style. We therefore adapt this method to our rule-based unification pattern matching. As Fig. 4 shows, developers extend the Agent class to define the rules (line 1). In this example, if the agent gets message "event(Name, No, slide(X))" and the inner state of the agent is controllable, then the agent will send the ACK message "ok(Myid, No)" to the sender. This program is part of a simplified version of the protocol used in the application we mentioned in the previous section. The original sender sent a command for changing the slide to page "X" to the receivers; the receivers change the slide if possible and send the ACK message to the sender.

Here, variables (starts with capital) have following definitions. Name is the sender ID, No is the event ID, and X is the page number of the controlling slide. In the ACK message "ok(Myid, No)," the variables are for the ID of the receiving agent and for the event id contained in the original message. After defining the rules, developers register the rule with the agent (line 07). For the rule-matching agent, call the registered methods. The rule definition itself is described in line 06. In line 06, message pattern is defined as "event(Name, No, slide(X))", and if the matching succeeds, registered methods in line 04 (public void foo()) are called. If the method returned true, the registered method in line 05(public void bar(Message m)) will be called.

Here, "Message" object (line 13) manages the relation between the received message term and the registered pattern. For example, if we want to get the term corresponding to "No" in the registered pattern "event(Name,No,slide(X))" from the received term, we use the method of Message object as follows. (This example is shown in line 15 in Fig. 4.)

Term t = m.get("X") ;

The usage of message sending is shown in lines 13 to 16 in Fig. 4. To send a message, the developer creates an instance of "Message" object (line13). Here the developer can specify variables in the message by using capitals. (In this example, we use "Name" and "No2.") We can use the method of Message class to substitute a practical message in the variables as follows (in Fig. 4 line 14 and line 15).

substitute(#variablename, #constant)

To define many message patterns in one agent, the rules can share the registering method to simplify the program. In this way, we can define rules and pattern matching in a Java program more simply than conventional approaches.

V. Related Work and Conclusion

The interactive work space [6] is a collaborative environment to integrate many applications and physical devices, and its event model can manage state transition. In this research, however, the developer must write the state and actions in its own language. Furthermore, it does not support powerful pattern matching. In contrast, our proposed approach is in the original java syntax and supports unification. Workspace emphasizes a simple syntax and easy management, but we implement the function as a library in the original language, so the approaches are different.

Jinni [7] is a black-board type integration model using unification. The jinni program syntax is based on prolog, so this approach doesn't integrate the rule-based approach and procedural programming either.

SOBA [8] is a framework for developing P2P applications, and some shared event management is similar to our method. However, SOBA does not support rule-based definition or powerful pattern matching like unification.

The concept of DJ [9] (Declarative Java) may be similar to our proposed method. DJ introduces constraint programming for the Java GUI in java's original syntax. DJ is focused on the GUI program while our approach focuses on message passing.

In this paper, we have introduced rule-based message passing using the unification method, and implemented a java library the general programmer can easily use with the original java procedural programming syntax.

References

[1] DOM: Document Object Model http://www.w3.org/DOM/
[2] XQuery: http://www.w3.org/XML/Query
[3] Ivan Bratko, Prolog Programming for Artificial Intelligence, Second Edition. Addison-Wesley 1990.
[4] E.Shapiro, The Family of Concurrent Logic Programming Languages, ACM Computing Serveys,Vol.21, 1989.
[5] K. Ueda and T. Chikayama, Desing of the kernel language for the parallel inference machine, The Computer Journal, 1990.
[6] Brad Johanson, Armando Fox, Terry Winograd, The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. IEEE Pervasive Computing Magazine 1(2), April-June 2002
[7] Paul Tarau. Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. Proceedings of PAAM'99, 1999
[8] SOBA project http://www.soba-projet.org
[9] Neng-Fa Zhou: Building Java Applets by Using DJ - A Java-based Constraint Language. COMPSAC 1999: