Yaccai: A multiagent system development framework

Nguyen Tuan Duc † and Ikuo Takeuchi†

This paper presents a new agent oriented programming language which utilizes the idea of collective operations in parallel programming to abstract agents cooperation while maintaining the autonomy of agent programs. The communication model underlying our language execution system separates program code for agent behavior from code for message passing and makes the description of autonomous agent easier. Moreover, collective operations simplify the process of deriving global knowledge, a crucial problem in multiagent systems. Because of these characteristics, our system is effective for developing multi-agent systems.

1. Introduction

Research in agent oriented programming language (AOP)[1] has focused on the description of mental state of autonomous agents which contains the expression of belief, desire and intention in the language. Beside autonomy, cooperation is another important aspect of agent and a lot of studies on cooperation model have been done[2][3]. However, there are few studies on integrating cooperation in AOP. In this paper, we present a method for simplifying the description of cooperating agents by collective operations in AOP. Moreover, we propose a new communication model that provides full support for message passing while maintaining the autonomous computational model of agents. Therefore, our language and communication model are appropriate for description of multiagent systems. We have developed the interpreter and execution environment for the proposed language in a framework called Yaccai (Yet another concurrent cooperating agents infrastructure☆).

2. Overview of the language

Our language is agent oriented because it supports the description of mental state of agents and automatically generates reasoning cycle (the cycle of sense - reason - act). The language provides constructs to define classes and agent classes like in normal object oriented languages. It borrows the concept of "communicator" and "collective operations" from MPI[4] to model agent group and abstract agent cooperation.

We provide a built-in belief-base in which agent program may use to store agent's beliefs and language constructs for querying and managing the belief-base based on ideas from LINQ (language integrated query). Belief-base operations are briefly described in **Table 1**.

Table 1 Belief-base operations

operation	example
add fact	belief fact new {x = 1, y = 2};
query	belief query {x, y};
projection	belief query {x, y} as p{x};
conditional query	belief query {x, y} as p where p.x == 1;
remove	belief remove {x, y};
conditional remove	belief remove {x, y} as p where p.x == 1;
join	(belief query {x, y} as p) join (belief query {y, z} as q) on p.y == q.y;

Figure 1 shows an example of a multiagent system definition in our language. Once the agent is created, the constructor will be called and then the plan "act" will automatically be executed. Each agent is mapped to a process (including process in remote host). The main plan ("act") and the plan for reacting to incoming messages ("MsgListener") are executed in different threads. Each agent in the example simply broadcasts a "Hello" message and its identifier (rank) to others. When an agent received "Hello" message, it stores the host address of the sender into belief-base.

3. Abstraction of agent cooperation

We use collective operations (Broadcast, Reduce, Gather etc.) to abstract the cooperation process of agents. The concept of "communi-

† Graduate School of Information Science and Technology, The University of Tokyo

☆ Yaccai can be downloaded from http://www.nue.ci.i.u-tokyo.ac.jp/%7Educ/yaccai/

```
1  agentclass HelloAgent {
2    public m_comm;
3    public function HelloAgent() {
4       m_comm = Environment.GetCommunicator(
5                        "World", MsgListener );
6    }
7    public plan MsgListener( msg ) {
8       id = -1;
9       match msg.Value with {
10         "Hello from", @{id}, "at", @{addr} -> {
11           belief fact new {rank=id, host=addr};
12         }
13      }
14   }
15   public plan act() {
16      myRank = Environment.GetRank();
17      m_comm.Bcast( "Hello from " + myRank +
18         " at " + Environment.Hostname() );
19   }
20 }
21
22 class SimpleMAS {
23   public static function Main() {
24      a1 = create HelloAgent() at "localhost";
25      a2 = create HelloAgent() at "somehost.com";
26   }
27 }
```

Fig. 1 A simple multiagent system definition

cator" is the same as in MPI[4], but collective operations' syntax and semantics are different. **Table 2** shows the list of collective operations that we support.

Table 2 Collective operations

Operation	Example
Barrier	comm.Barrier("barrier_name");
Broadcast	comm.Bcast(msg);
Reduce	comm.Reduce(operator, expression);
Gather	comm.Gather(expression);
Scatter	comm.Scatter(array of msg);

Collective operations in normal parallel distributed programming library such as MPI[4] must be invoked in parallel by all processes that are participating in the operations. This requirement ensures the efficiency for the execution of collective operations but it causes difficulty in maintaining the autonomy of each process. In our system, we allow collective operations to be invoked by just one agent and other agents will automatically participate in the operations, as in the plan "act" in Fig. 1.

Collective operations allow agents to effectively cooperate with other agents in the same communicator. They make the process of deriving global information easier. For example, an agent can get the sum of ID of all agents in the system by invoking a reduce operation: "comm.Reduce(SUM, belief query ID);". The expression "belief query ID" is evaluated at each agent and the results are summed up by the operator SUM.

4. Communication and execution model

The execution of collective operation is not simple because it involves in all agents while the operation is invoked by just one agent. To support this new syntax of collective operation, we divided each agent into two layers: the agent reasoning layer and the message passing layer as shown in **Fig. 2**. The reasoning layer contains user's code for the agent program while the message passing layer contains code of the execution environment. The message processing plan and main plan are in the agent reasoning layer. This separation correctly fits the

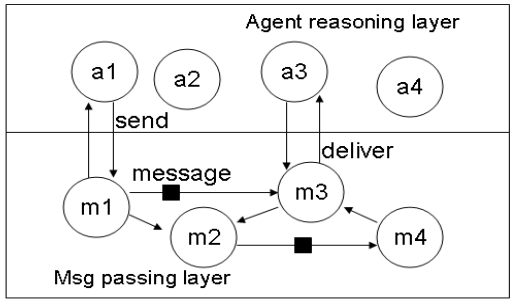

Fig. 2 Communication model

requirements of autonomous agent description and the new collective operation syntax.

5. Conclusion and future work

We have presented a new agent oriented programming language in which agent cooperation is highly abstracted by using collective operations. The communication model underlying our system ensures the autonomy of agents while providing full support for message passing and coordination. We are going to implement several real-world multiagent system benchmarks to investigate the effectiveness of the language and the communication model.

References

1) Shoham, Y.: *"Agent Oriented Programming"* in: Journal of Artificial Intelligence, 60 (1) (1993) 51-92.
2) Pynadath, D., et al.: *Toward team-oriented programming*. Proceedings of ATAL'99, published as Springer Verlag LNAI "Intelligent Agents VI".
3) Michael Schumacher: *Objective coordination in multi-agent system engineering*. LNAI 2039.
4) *The Message Passing Interface*. http://www-unix.mcs.anl.gov/mpi/.