エージェント指向プログラミングにおける集団操作の応用

グェン　トアン　ドゥク[†a)]　　竹内　郁雄[†b)]

Application of collective operations in agent oriented programming language

Nguyen TUAN DUC[†a)] and Ikuo TAKEUCHI[†b)]

Abstract. In multiagent systems, cooperation is the process of synchronization, sharing knowledge to achieve the overall goal. Description of cooperation is a crucial requirement for programmers to realize an intelligent multiagent system. In this paper, we propose a method for describing agents' cooperation process using collective operation, a concept in parallel programming. The method is implemented in a new agent oriented programming language called Yaccai. Using collective operations, programmer can easily create multiagent systems with complex cooperation schemes. Our experiment shows that, the application of collective operations is very effective for developing distributed multiagent systems.

Keywords. collective operation, cooperation model, agent oriented programming, language design, agent execution model

1 Introduction

A multiagent system (MAS) is a system consists of many autonomous intelligent agents which interact by sending/receiving messages of several types. Multiagent systems can be used for modeling problems which are difficult or impossible for a single monolithic program to solve. The main aspects that give multiagent systems this powerful problem solving ability are autonomy and cooperation. Each agent in a multiagent system is an autonomous intelligent entity that can react to events and adapt to the environment. On the other hand, cooperation is a very important process in multiagent systems which associates agents in a main objective of the system: achieving the overall goals. Agents cooperate with other agents by sharing knowledge and exchanging useful information to solve the problem. Especially, in software multiagent systems which are normally distributed across several machines, cooperation can be viewed as the process of synchronization and passing messages between agents in order to coordinate and sharing knowledge about the outside environment. Therefore, it is important to investigate the application of message passing model into the description of cooperation process in multiagent systems.

Since cooperation is essential for multiagent systems, many studies on agent cooperation model [5] [8] and cooperation language [3] [4] have been done. However, these languages and models either do not exploit the parallel distributed nature of MAS or do not take care of autonomy, the main characteristic of agent. Moreover, even the concept of agent is similar to actor in the Actor Model [2], a model that has influenced many distributed object oriented programming languages, there is a lack of effort for applying ideas in parallel distributed programming and distributed object oriented programming in MAS.

In this paper, we propose an approach to model agents' cooperation process using *collective operation*, a concept in the Message Passing Interface (MPI) [6], a famous parallel distributed programming interface and library. The description of many

[†] 東京大学 情報理工学系研究科
a) E-mail: duc@nue.ci.i.u-tokyo.ac.jp
b) E-mail: nue@nue.org

cooperation schemes is mapped to collective operations in a new agent oriented programming language called *Yaccai* (Yet another concurrent cooperating agent infrastructure). Using collective operations, programmers can easily create multiagent systems with complex cooperation protocols. Moreover, our model supports dynamic creation and destruction of *communicator*s (a concept refers to processes group in MPI) which makes the formation and dissolution of agent groups (agent societies) easier. Our experiment shows that, the application of collective operations is very effective for developing distributed multiagent systems.

The remainder of this paper is organized as follows. In Section 2, we introduce related work on cooperation model in multiagent systems and compare to our model. Section 3 presents method for abstracting agent cooperation by using collective operations and language constructs for representing cooperation schemes in Yaccai. The execution and communication model of agents are described in Section 4. We discuss about the application of our cooperation model, especially for deriving global knowledge of MAS in Section 5. Section 6 gives some experimental results for evaluating the cooperation model and language. Finally, Section 7 discusses about future work and concludes the paper.

2 Related work

Cooperation models such as joint-intentions model [5] or teamwork [7] allow the description of global goal for the entire multiagent system and coordination scheme is automatically derived from the team's goal. But it is difficult to ensure the autonomy of each agent because all agents have the same goal and mental state (i.e., belief, desire and intention, the BDI model [1]). Moreover, it is difficult to decentralize the system using these models because maintaining global goals and (implicitly) shared mental state leads to many problems in consistency and communication cost.

Michael Schumacher proposed a model for inter-agent coordination, called ECM [8] and a programming language to specify agent hierarchy. Agents participate in many agent societies, called "blops". Each blop is a group of agents, in which agents can easily communicate with each other and even broadcast messages when they want. A multiagent system is therefore a hierarchy of blops which contains many agents, each agent is a computational process. However, ECM and its languages do not support the description of mental state of agent, agent itself needed to be specified by another programming language. While ECM utilizes the idea of process group in parallel programming to model agent group ("blop" is similar to "communicator" in MPI), it does not exploit full power of collective operations (it only supports one collective operation, i.e., broadcasting of messages within a blop).

Our system combines the advantages of agent oriented programming (AOP) and the coordination models mentioned above. The system ensures the autonomy of agents and provides constructs for description of cooperation, communication between agents. Cooperation of agents is abstracted by using collective operations while mental state and reasoning cycle can be specified directly in the our AOP language. Agents may freely participate in *communicators* and within a communicator, agents can send messages to an individual or broadcast to all agents in the communicator. Furthermore, agent program may use "reduce", "gather", etc. operations to derive global knowledge of the entire multiagent system. With these characteristics, our language fixed the drawbacks of previous frameworks: it maintains the autonomous computational model of multiagent system while providing constructs for specifying cooperation schemes. Put in other words, by introducing "communicator" and "collective operations" in AOP, we can abstract the cooperation process of agents while autonomy of agents and efficiency of message passing are ensured by our new communication and execution model.

3 Abstracting cooperation by collective operations

Collective operations are operations that involve

in many processes and data of these processes of execution. MPI [6] defines many collective operations such as barrier, broadcast, gather, reduce, ... to support synchronization and cooperation of processes which may run in different hosts and different operating systems. For example, broadcasting message from a process (or an agent) to all processes (agents) in a group is a collective operation because all processes are affected by the operation (by receiving the message). Since collective operations give effect to many processes, it is similar to cooperation of many agents. Therefore, collective operation can be used to model the agent cooperation in MAS.

Collective operations deal with processes in a same group which called a "communicator" in MPI. A communicator is therefore a group or society of agents in our model. When an agent participates into a communicator, it can invokes collective operations within the communicator. An agent can easily join a communicator by calling a method named "GetCommunicator":

```
comm = GetCommunicator( comm_name,
                        msg_listener );
```

where `comm_name` is a string represents the name (identifier) of the communicator and `msg_listener` is a plan (procedure) to process incoming messages. The agent that invoked the above code will be a member of the communicator named `comm_name` (another agent can join this communicator by invoking the `GetCommunicator` method with the same communicator name).

After joining a communicator, an agent can initiate collective operations in that communicator. The syntax and semantics of collective operations that we support is shown in Table 1.

The "barrier" operation in Table 1 is a tool for synchronizing all agents in a communicator. The message listener (`msg_listener`) does not need to do anything for barrier operation (no message is passed to the listener when barrier operation is invoked) because the barrier is merely a tool for synchronizing agents (i.e., force agents to wait until the barrier method are invoked by all agents in the communicator). The same thing happens for "reduce" or "gather" operation: the expression is automatically evaluated at each agent and then the operator will be applied to the results or all results are collected into a result set, the message listener is not invoked. On the other hand, in "broadcast" or "scatter" operation, the message will be passed to message listener and it is the responsibility of the programmer to describe what to do when an agent received a message. By this way, an agent with reactive capability can be easily created: the action that the agent takes when an event occurred can be specified in the message listener.

An agent can leave from a communicator by invoking "Leave" method on the communicator:

```
comm.Leave();
```

Messages in the communicator will not be passed to the message listener any more after the agent leaved the communicator (even though, the agent may participate in the message passing process of the execution environment and act as a router to forward messages to another agent in the system without awareness of the programmer).

Fig. 1 shows an example of a multiagent system definition in Yaccai. Once the agent is created,

```
 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

Table 1 Collective operations

Operation	Meaning	Example
Barrier	Synchronizing all agents in the communicator	comm.Barrier(“barrier_name”);
Broadcast	Broadcasting message to all agents in `comm`.	comm.Bcast(msg);
Reduce	Evaluate `expression` at each agent and apply `operator` on the result set in round-robin manner	comm.Reduce(operator, expression);
Gather	Evaluate `expression` at each agent then gather the results to an agent	comm.Gather(expression);
Scatter	Scatter the array of messages to all agents in `comm`	comm.Scatter(array of msg);

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 the same thread but are scheduled by the scheduler of the execution environment bringing an image that they 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. The complete grammar of the language is available at the Yaccai's homepage. [1)]

Collective operations in normal parallel distributed programming library such as MPI must be invoked in parallel by all processes that are participating in the operation. This requirement ensures the efficiency for the execution of collective operations but it causes difficulty in maintaining the autonomy of each process since it requires all processes invoke the operation at the same time. In our system, we allow collective operations to be invoked by just one agent and other agents will automatically participate in the operations. Table 2 gives a list of differences between our model and MPI.

Collective operations allow agent to effectively cooperate with other agents in the same communicator. It makes 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. Section 5 shows more about application of collective operations. It is important to note that, collective operations abstract the cooperation of agents, the abstraction simplifies the description of cooperating process.

1) http://www.nue.ci.i.u-tokyo.ac.jp/%7Educ/yaccai

Table 2 Differences between Yaccai and MPI

	MPI	Yaccai
Code	All processes have same code	Can be different for each agent
Collective operations syntax	Invoked by all processes (same code)	Invoked by an agent
Can leave from a communicator	No	Yes

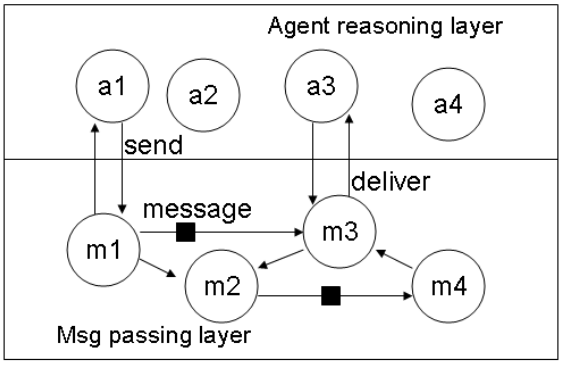

Fig. 2 Communication model of Yaccai. The circle “a_i” denotes the reasoning layer, the circle “m_i” represents the message passing layer of agent i.

4 Execution model for collective operations

The execution of collective operation is not simple because it involves in all agents while the operation is invoked by just one agent. To accomplish this goal, we propose a new communication model for multiagent systems. The model ensures the autonomy of agent while providing message passing

API for collective operations and cooperation.

Fig. 2 shows the communication model of our system. Each agent is divided into two layers: the agent reasoning layer and the message passing layer. Reasoning layer contains user's code for the agent program while message passing layer contains code of the execution environment. Therefore, users do not have to write code for message passing, the system provides primitives for message passing and collective operations. The message processing plan and main plan are in agent reasoning layer. When messages arrived at the message passing layer, it is delivered by the message dispatcher to the appropriate message processing plan. The plan is registered when the agent joined the communicator (by calling `GetCommunicator` as shown in Section 3). When the reasoning layer's code invokes *send* or *broadcast* method of communicator, the message will be passed to the message passing layer of the same agent first, and it is actually sent to destination in this layer.

The separation of agent's reasoning code and message passing code leads to some advantages. First, it ensures the autonomy of agent because agent's reasoning code can be executed in one thread and message passing code in another thread of execution. The programmer can easily specify proactive reasoning process of agent without concern about the message passing process (reactive reasoning can be specified inside the message listener of each communicator). Second, it supports the implementation of collective operations with different semantics from the semantics in normal parallel distributed programming: collective operations can be invoked by just one agent, not all agents in parallel. The message passing layer does the job of passing messages independently from agent's reasoning code. That is why collective operation may be invoked by one agent (at the agent reasoning layer) but the operation is performed in all other agents (in message passing layer and if needed, in the message processing plan). Finally, message passing optimization can be done by the system in message passing layer and user code can enjoy these optimizations without any effort.

We have implemented the language interpreter and execution environment to support the proposed communication model. Each agent is mapped to one process in the operating system which can be in a remote host. Agent can freely create new agent by calling the operator "create" and specify host name in which the new agent will be executed. Each agent contains several threads of execution. One thread executes the agent's reasoning code and zero or more threads execute the message passing layer's code (the code is provided by the execution environment). When messages come, they are stored into message queue of the agent and then the message dispatcher will fetch the messages and deliver to the appropriate communicator's message processing plan. This execution model is very suitable for modeling autonomous pro-active agent. An agent is said to be pro-active if it has its own goal and actively makes action to achieve the goal. On the other hand, agent must also deal with suddenly happened events by reacting to events it received. This type of reasoning is called reactive reasoning. The message processing plan is responsible for reactive reasoning while the main plan is the place where pro-active reasoning code could be described. By this way, programmers can easily model autonomous agent with pro-active reasoning and reactive reasoning capabilities.

5 Application of collective operations

5 1 Deriving global knowledge of multiagent systems

Global knowledge is knowledge involved in entire multiagent systems, for instance, the minimum value of a particular property of agents. Data that is distributed across many agents may be considered as global knowledge because gathering of the data involves in many agents. These kinds of knowledge can be easily obtained by using collective operations.

For example, a Vacuum Cleaner agent can know how many agents are in idle state by invoking the

following reduce operation:

```
comm.Reduce( Sum, (belief query idle)[0]  );
```

where `Sum` is an operator of 2 operands which returns the sum of these operands. The belief-base query expression is evaluated at each agent and returns a collection (contains only one element) that is 1 if the agent in idle state and 0 otherwise. The operator `Sum` is applied to the result set in a particular order (the order of the application depends on the reduce algorithm, such as tree-like or linear algorithm).

Another method to achieve the same goal is using the gather operation to gather all idle state of other agents:

```
comm.Gather( (belief query idle)[0] );
```

The gather operation returns a collection contains values representing idle state of all agents in the communicator "comm".

5 2 Synchronization

Sometimes agents need to be synchronized to perform some actions. Synchronization may be done by collective operations such as barrier or wait. For instance, an agent may wait for all other agents to call the method comm.Barrier("barrier_name") (with the same barrier name) by invoking `comm.Barrier( "barrier_name" )`. The agent will be blocked until all other agents in the same communicator also called this method.

The only drawback is that, it is the responsibility of the programmer to ensure that all agents must call the synchronization methods, otherwise the agent first called to these methods will be blocked indefinitely (while in MPI this is automatically achieved since all processes have the same code when they are calling barrier).

6 Evaluation

In this section, we provides some experiment results to evaluate our language and cooperation model. We use Vacuum Cleaner problem as a typical multiagent system benchmark to evaluate our language. The result shows that it is easy to realize a Vacuum Cleaner agents team with many complex cooperation schemes by using the framework. Moreover, the result also confirms the effectiveness of collective operation as a tool for cooperation.

We built a multiagent system for simulation of Vacuum Cleaner problem. Each Vacuum Cleaner is represented as a software agent whose capabilities are move and clean. The simulation server constructs a virtual space which is a grid of 20 x 30 cells and contains many cells that have dirt. Agent can only move up, down, left or right (one step for each cycle) in the virtual space and it receives information about the current position (e.g., contains dirt or not) from the server. The agent can only clean a unit of dirt in each cycle by sending a "clean" command. A position (a cell) in the virtual space may contain many units of dirt. The server is written in C# while the agent is written in our language (complete source code for the agent can be viewed at the Yaccai's homepage[2]). At each cycle, the server waits until all agents have issued a command, then it advances the simulation one step (an agent can only issue one command for each cycle). Communication between agents is done by using collective operations. A game lasts for 1000 cycles; at each cycle, the server reports the score of the game by the following formula:

$$Score = \frac{Total\ units\ of\ dirt\ cleaned}{Total\ units\ of\ dirt} \times 100 (1)$$

In the first experiment, we created a multiagent system which contains many simple agents: the agents do not cooperate with each other, they only send/receive commands and information from the simulation server. The agents simply scan the virtual space by moving horizontally first then go up/down one row when they could not move in horizontal direction and change the horizontal direction from left-right to right-left and vice versa.

In the second experiment, we created a multiagent system which contains agents that cooperate using broadcast operation. The agents use the same strategy in the first experiment to move around the virtual space. When an agent found a cell has dirt, it will broadcast the position of the cell and the

2) http://www.nue.ci.i.u-tokyo.ac.jp/%7Educ/yaccai

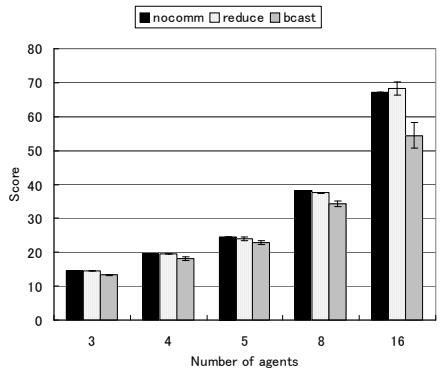

Fig. 3 Average score of 5 times of simulation on `map_dense`

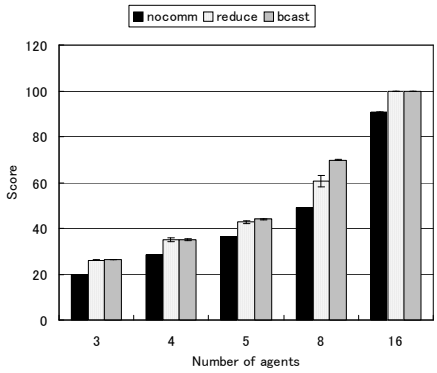

Fig. 4 Average score of 5 times of simulation on `map_sparse`

units of dirt contained there to all other agents. When received message from other agents, an agent will store the information into its belief-base and determine if it should go to the cell to clean or not. The heuristic to determine to go or not is simple: if the agent is in idle state and the number of units of dirt is bigger than 4 times of the distance between current position to dirt's position, the agent will go to dirt's position. On the way to the dirt, if the agent found another place contains dirt, it will clean the place immediately (and broadcast the position to other agents). When an agent successfully cleaned a position, it also broadcasts the information to other agents. If the agent is not in idle state, it simply stores the cell in its belief-base and when it becomes idle, it will query the belief-base for dirty cells and go to clean. An agent will remove the dirt's position from its belief-base when it received the clean message from other agents.

In the third experiment, we created a multiagent system which contains agents that use similar cooperation scheme to the agent in the second experiment except that when an agent found dirt, it uses reduce operation to know the idle agent nearest to it. Then it sends the information about the cell to that agent only (not broadcast to all agents).

We executed the simulation with 2 scenarios (maps): `map_dense` contains large amount of dirt that broadly distributed across many cells in the virtual space (the map has many cells that contains dirt, so it is dense of dirt), `map_sparse` contains large amount of dirt that comparatively concentrated on a region in the virtual space. The experiment is carried on a cluster of 20 machines (Intel Pentium 4, 2.8GHz, 2GB RAM, Linux 2.6.18) connected by Gigabit ethernet to guarantee that each agent is executed on a different host. The score is the average score of 5 times of simulation reported at the end of each simulation (i.e., at the cycle 1000).

Fig. 3 and Fig. 4 show the score for the non-communication agents, broadcast agents and reduce agents with 2 maps: `map_dense` and `map_sparse` respectively. When the number of agents increased, the score also increased (the simulations on `map_sparse` with 16 broadcast and reduce agents reached the highest possible score (100) after about 900 cycles, i.e., the agents do nothing in about 100 cycles left). Standard deviation is very small (as shown in the figures) because we do not use any random parameter. We also performed the simulation with 32 agents run on 16 hosts (each host has 2 agents), but we do not show the teams' score in the graphs because for `map_sparse` they reached the highest score too early before the last cycle.

In `map_dense`, the broadcast agents do not perform very well because they can not scan entire the space to find dirt and do too many useless moves. The broadcast agents seem to concentrate

on a place at each time (because when received broadcast message they often go to the dirt place if they are in idle state or when they become idle, they will query the belief-base for the cell and go to clean). The reduce agents have the same performance with non-communication agents because when discovered a position has dirt, an agent does not broadcast message to all agents, only one agent is affected by the message. When there are many places with dirt, the agents are busy (not in idle state) and they will not react to messages from other agents immediately so the behavior of the team is similar to non-communication team. In the `map_sparse` map, the situation is different. The broadcast team has the best performance because they do not spend a lot of time in idle moves. They can concentrate on a dirt place immediately when an agent found the dirt place. The reduce team also has relatively good performance because when agents are in idle state, they will react to messages from other agents immediately so the behavior is similar to broadcast team. Agents team without cooperation has poor performance in this situation because agents have to find dirt independently and do many useless moves to scan the virtual space.

The result confirms that collective operations are good for cooperating agents: the agents that use collective operations to cooperate obtained high score when cooperation is important to the problem. In situations where cooperation becomes very important (e.g., in the `map_sparse`), collective operation is very effective because it simplifies the description of cooperation of agents. Even in situations where cooperation is not important (e.g., in the `map_dense`), the agent team uses collective operation may also achieve good performance if it uses appropriate cooperation scheme to reduce the risk of "over-cooperating" (i.e., too concentrated on a region).

7 Conclusion and future work

We have presented a new approach for modeling cooperation process of agents using collective operations. The communication model underlying our system ensures the autonomy of agents while providing full support for message passing and coordination. The system is therefore suitable for developing multiagent system, in which agents are autonomously, pro-actively and reactively taking actions while cooperating with others to achieve the goal. We also discussed about the application of our model in deriving global knowledge and shows the experiment result that confirms the effectiveness of our cooperation model. We currently broadcast messages to all agents at the message passing layers (when broadcast operation is invoked) and ignore the messages if the agent is not in the corresponding communicator. In the future, it should be better if we manage the communicator information and only broadcast messages to agents that are in the communicator. To achieve this, we need to use a centralized agent to manage communicator's data or replicate the data at each agent. The preferable method is replication of data, but it may lead to the data consistency problem so the consistency model must be investigated. Furthermore, we are going to implement several real-world multiagent system benchmarks (such as RobocupSoccer or RobocupRescue agent team) to investigate the effectiveness of the language and the communication model.

References

[1] A. Rao and M. Georgeff, "Modeling rational agents within a BDI architecture", Proceedings of the KR91.

[2] Carl Hewitt, "A Universal Modular Actor Formalism for Artificial Intelligence", Proc. of 3rd Intl. joint Conf. on Artificial Intelligence, IJCAI 1973.

[3] Foundation For Intelligent Physical Agents, "Fipa Communicative Act Library Specification, 2001", http://www.fipa.org/specs/fipa00037/XC00037H.html

[4] Finin, T. Weber, et al., "Specification of the KQML agent-communication language", Technical Report EIT TR 92-04, Enterprise Integration Technologies, 1992.

[5] Jennings, N. R, "Controlling cooperative problem solving in industrial multi-agent systems using joint intentions", Artificial Intelligence, 75(2).

[6] The Message Passing Interface, http://www-unix.mcs.anl.gov/mpi/

[7] Pynadath, D., et al., "Toward team-oriented programming", Proc. of the ATAL'99.

[8] Michael Schumacher, "Objective coordination in multi-agent system engineering", LNAI 2039.