Abstraction of agent cooperation in agent oriented programming language

Nguyen Tuan Duc and Ikuo Takeuchi

The University of Tokyo, Japan
duc@nue.ci.i.u-tokyo.ac.jp, nue@nue.org

Abstract. Collective operation is a concept of parallel programming in which many processes participate in an operation. Since collective operations are suitable for modeling the coordination of many processes, they can be used to model cooperating agents in a multiagent system. In this paper, we propose an agent oriented programming language that exploits collective operations to abstract the cooperating process of agents. We also present a method for implementing collective operations while maintaining the autonomous computational model of agent. Our experiment shows that our language and cooperation model have many advantages in developing multiagent systems....

1 Introduction

In multiagent system (MAS), agents need to exchange useful information with each other in order to reach an agreement or collaborate for achieving a goal. The process of communicating and exchanging knowledge is known as cooperation. Cooperation is a crucial requirement in MAS because without cooperation the system is simply a set of separated agents and has no ability of collaborating to reach the goal.

On the other hand, in agent oriented programming (AOP), a new programming paradigm proposed by Y. Shoham [1], agent is modeled as an autonomous, reactive and pro-active entity. Because of this autonomous computational model, the integration of autonomous agent and cooperating agent is not simple. Agents need to be autonomous, however, they also need to collaborate in order to achieve the goals.

In this paper, we propose an agent oriented programming language that supports the cooperation of agents. The language uses the concept of collective operation in parallel distributed programming to abstract the cooperating process of agents. Moreover, it maintains the autonomy of agent and provides constructs for describing agent's *mental state* (i.e., belief, desire, intention [3]). We have implemented a framework called Yaccai (Yet Another Concurrent Cooperating Agent Infrastructure) to support the execution of multiagent systems written in our language. The communication model underlying our language's execution environment ensures the autonomy of each agent while providing full

support for message passing. Our experiment shows that by using collective operations, global knowledge, an important element in multiagent systems, can be easily derived.

The rest of this paper is organized as follows. Section 2 compares our system with existing AOP languages and cooperation models. Section 3 presents our language design and language constructs for abstraction of agent cooperation using collective operations. Section 4 describes the execution model that supports the implementation of collective operations. We discuss about the application of collective operations in Section 5. Section 6 shows empirical results for evaluating the system. Finally, Section 7 discusses about future work and concludes.

2 Related work

Research on AOP language has focused on how an autonomous agent can be described in the language, that is, how to express the mental state (the intra-agent aspects) of an agent efficiently and easily using constructs provided by the language [1][4][2]. However, existing agent oriented programming languages do not concentrate on the communication model of agents and do not pay enough attention to abstraction of agent cooperation despite the fact that cooperation is a very important issue in MAS.

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.

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. 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.

Our system combines the advantages of agent oriented programming languages and the coordination models mentioned above. The system ensures the autonomy of agents and provides constructs for description of cooperation, communication between agents.

3 Language design and abstraction of agent cooperation

3.1 Constructs for modeling mental state and reasoning cycle

Our language is agent oriented because it supports the description of mental state of agents and automatically generates reasoning cycle (the cycle of sense - reasoning - act). The language provides constructs to define classes and agent

classes like in normal object oriented languages. Each agent has its own independent integrated belief-base to avoid the overhead of synchronizing common belief-base and ensure the autonomy of the agent. Belief-base query/update operations are integrated in the language as language constructs that are similar to LINQ [9].

```
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

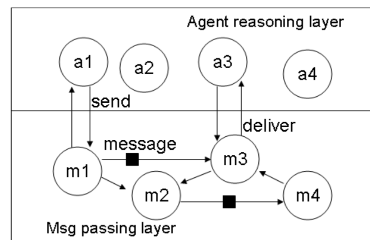

Fig. 2. Communication model

Reasoning cycle of agent is automatically realized when agent program defines a special plan with name "act". Fig. 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 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 main program (the `SimpleMAS` class) uses `create` statement to create agents at desired host. Complete grammar of the language is available at the Yaccai's homepage[1].

3.2 Abstraction of agent cooperation

We present a new approach to model the cooperating process of agents, that is, using collective operations to abstract cooperation. Collective operations are operations that involve in many processes and data of these processes of execution. For instance, broadcasting a message to all agents in a group is a collective

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

operation because all agents are affected by the operation. MPI[6], a famous parallel distributed programming interface, defines many collective operations such as barrier, broadcast, gather, reduce, ... to support synchronization and cooperation of processes.

We use the concept of "communicator" in MPI (that is similar to "blob" in ECM [8]) to represent agent group, agent can freely participate into a group by invoking the following method:

```
comm = Environment.GetCommunicator( "comm_name", listenerPlan );
```

This method will get the communicator named "comm_name" and set "listenerPlan" as the message processing plan for this communicator. Every message comes to this agent from the communicator will be passed to `listenerPlan` to be processed.
An agent can leave from a communicator by invoking the "Leave" method on the communicator:

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

Messages come from this communicator will not be passed to "listenerPlan" anymore.
Agents form group of agents when they participate in the same communicator (by invoking the method `GetCommunicator` with the same communicator name). The concept of "communicator" in our language is the same as in MPI but

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);

collective operations' syntax and semantics are different. Collective operations in 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. For example, an agent may invoke the following method to broadcast message to all agents in the same communicator:

```
comm.Broadcast( message );
```

Table 1 shows the list of collective operations that we support.
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.

4 Communication model and execution environment

The execution of collective operation is not simple because it involves in all agents while the operation is invoked by just one agent. To cope with this problem, we use a new execution and communication model for agents as shown in Fig. 2. 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 (the system provides primitives for message passing and collective operations). The former contains exactly one thread while the later may contain several threads of execution (each agent is mapped to a process, possibly in a remote host). 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 message processing plan is responsible for reactive reasoning while the main plan is 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.

The separation of agent's reasoning code and message passing code supports the implementation of collective operations with different semantics from semantics in normal parallel distributed programming: collective operations can be invoked by just one agent, not all agents in parallel because the message passing layer does the job of passing messages independently from agent's reasoning code.

5 Application of collective operations

Global knowledge is knowledge that involves 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 way to achieve the same goal is using the gather operation to gather idle state of all 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".

6 Evaluation

In this section, we provide some experiment results to evaluate our language and cooperation model.

We built a multiagent system for simulation of Vacuum Cleaner Robot problem. Each Vacuum Cleaner is represented as an agent whose capabilities are move and clean. The simulation server constructs a virtual space which is a grid of 20x30 cells; each cell contains zero or more units of 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., number of units of dirt in the cell) from the server. The agent can only clean a unit of dirt in each cycle by sending a "clean" command. The Vacuum Cleaner agent team is written in our language (complete source code for the agent can be viewed at the Yaccai's homepage [2]). A game lasts for 1000 cycles; at each cycle, the server reports the score of the game by the following formula:

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

In the first experiment, we created a multiagent system which contains 3 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 3 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 units of dirt

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

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. When an agent successfully cleaned a position, it also broadcasts the information to another agents. 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 3 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 nearest idle agent. Then it sends the information about the cell to that agent only (not broadcast to all agents).

Table 2. Average score of 5 times of simulation

Map	NoComm	Bcast	Reduce
map_dense	32.7(±0)	24.28(±0)	32.2 (±0.3)
map_sparse	75.22(±0)	90.91(±0.5)	86.6 (±0.1)

Table 2 shows the score for the non-communication agents, broadcast agents and reduce agents in our experiment with two scenarios: `map_dense` contains large amount of dirt that broadly distributed across many cells in the virtual space, `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 6 machines connected by Gigabit ethernet to guarantee that each agent is executed on a separate machine. The score is the average score of 5 times of simulation reported at the end of each simulation, the numbers after symbol ± inside the parenthesis are standard deviations (standard deviation is very small because we do not use any random parameter).

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. They 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 only one agent is affected by the message when a dirt position is found. When there are many places have 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 complex cooperation protocols may be easily described using collective operations. 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 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. 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 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. Shoham Y.: Agent oriented programming. Artificial Intelligent **60(1)** (1993) 51–92
2. Hindriks K.V., et al.: Architecture for Agent Programming Languages. In Proc. of the 14th European Conference on Artificial Intelligence (ECAI 2000)
3. Rao A., Georgeff M.: Modeling rational agents within a BDI architecture. In Proc. of the Intl. Conf. on Knowledge Representation and Reasoning KR-91 (1991)
4. Rao A.: AgentSpeak(L): BDI Agents speak out in a logical computable language. In Proc. of the 7th European Workshop on Modeling Autonomous Agents in a Multi-Agent World (1996)
5. Jennings, N. R.: Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. Artificial Intelligence **75(2)** (1995) 195–240
6. The Message Passing Interface. http://www-unix.mcs.anl.gov/mpi/
7. Pynadath D., et al.: Toward team-oriented programming. In Proc. of the 6th International Workshop on Agent Theories, Architectures, and Languages (ATAL 1999)
8. Schumacher M.: Objective coordination in multi-agent system engineering. Lecture Notes in Computer Science **2039** (2001)
9. The LINQ project. http://msdn.microsoft.com/en-us/netframework/aa904594.aspx