Towards An Agent Oriented Programming Language with Caste and Scenario Mechanisms *

Ji Wang
National Laboratory for Parallel and Distributed Processing
Changsha, 410073, China
jiwang@mail.edu.cn

Rui Shen
National Laboratory for Parallel and Distributed Processing
Changsha, 410073, China
shenrui98@yahoo.com

Hong Zhu
Department of Computing
Oxford Brookes University
Oxford, OX33 1HX, UK
hzhu@brookes.ac.uk

ABSTRACT

The paper presents an agent-oriented programming language SLABSp. It provides caste and scenario mechanisms in a coherent way to support the caste-centric methodology of agent-oriented software development. It uses caste as a modular facility to organize agents into castes and to represent their structure and behavior characteristics. SLABSp also uses scenarios to define agents' behaviors in the context of environment situations. In the paper, the implementation of the language is briefly described. An example of the program is given to illustrate its programming style.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*

General Terms

Languages

Keywords

Caste, Scenario, Agent-Oriented Programming

1. INTRODUCTION

Recent years has seen a rapid growth of research in agent-oriented (AO) software development methodologies. However, programming languages based on AO methodologies have not been explored as desired from the perspective of software engineering. It is desirable to find/invent suitable language facilities to support efficient and direct implementations of the concepts and characteristics of agents presented in AO analysis and specification. Recently, in [4, 5, 6], caste is proposed as the classifier of agent to define a collection of agents that have the same behavior and structural characteristics, and scenario is proposed to define agent behaviors in the context of environment situations. This paper reports a programming language SLABSp and its implementation that supports these language facilities directly and coherently.

*Supported by the National NSF of China under grant No. 60233020 and 90104007, the National High Technology R&D 863 Programme of China under grant No. 2002AA116070, and Program for New Century Excellent Talents in University.

AAMAS'05, July 25-29, 2005, Utrecht, Netherlands.

2. CASTE AND SCENARIO

SLABSp is a Java-extended programming language designed to support the caste-centric approach to AO software development methodology [5]. Its key concepts and language facilities are castes and scenarios.

SLABSp regards a multi-agent system (MAS) as a set of agents, which are encapsulations of states, actions and behavior rules that govern its behaviors. SLABSp organizes agents in a MAS into castes, which is the classifier of agents and a modular programming unit that defines templates of the structure and behavior characteristics of agents. Just as classes in OO languages are abstractions of sets of objects, castes are abstractions of sets of agents that have the same features of state spaces, actions, behaviors and environments. However, in contrast to the static bindings of objects to classes in OO paradigm, an agent can be bounded to a caste dynamically, i.e. it may join to or quit from a caste at runtime. Each agent can join multiple castes. When an agent joins a caste, it will obtain all elements of the caste, including the state variables, actions, and behavior rules. Naming conflictions must be carefully avoided in programming when an agent may join multiple castes. Currently, elements defined in different castes but of the same name and type are considered as the same element. A more complicated mechanism for the detection and resolution of name conflictions is still under investigation. The concept of caste has been presented in [5] and examined in [4, 6] to justify its features as a step beyond object orientation. The structure of agent and caste in SLABSp is shown in Figure 1.

SLABSp uses scenarios to describe agent behaviors in the context of environment situations. Using scenarios, agents can perceive other agents' behaviors in its environment to decide its action rather than driven by message communications. Here, the environment of an agent is the set of agents in the system that can affect its behavior.

In general, the notion of scenario as presented in [1, 3, 5]

Philosopher.p

```
caste Philosopher {
   action think(){
      do{
         try { Thread.sleep(5000);
         } catch (Exception e){}
      }
   }
   action eat(){
      do{
         @takeLeft();
         @takeRight();
         try { Thread.sleep(5000);
         } catch (Exception e){}
         @putLeft();
         @putRight();
      }
   }
   action takeRight(){ do{} }
   action takeLeft(){   do{} }

   action putRight(){ do{}  }
   action putLeft(){   do{}  }

   behavior think0(){
      do { @think(); }
   } when {[ @start() ] } while {}

   behavior think(){
      do { @think(); }
   } when {[ @eat() ] } while {}
}
```

P1.p

```
agent P1 join Philosopher {
   behavior eat(){
      do { @eat(); }
   } when {
      [ @think() ]
   } while {
      not (P5: [@takeRight()])
        and not (P2: [@takeLeft()])
   }
}
```

P2.p

```
agent P2 join Philosopher {
   behavior eat(){
      do { @eat(); }
   } when {
      [ @think() ]
   } while {
      not (P1: [@takeRight()])
        and not (P3: [@takeLeft()])
   }
}
```

P3.p

```
agent P3 join Philosopher {
   behavior eat(){
      do { @eat(); }
   } when {
      [ @think() ]
   } while {
      not (P2: [@takeRight()])
        and not (P4: [@takeLeft()])
   }
}
```

P4.p

```
agent P4 join Philosopher {
   behavior eat(){
      do { @eat(); }
   } when {
      [ @think() ]
   } while {
      not (P3: [@takeRight()])
        and not (P5: [@takeLeft()])
   }
}
```

P5.p

```
agent P5 join Philosopher {
   behavior eat(){
      do { @eat(); }
   } when {
      [ @think() ]
   } while {
      not (P4: [@takeRight()])
        and not (P1: [@takeLeft()])
   }
}
```

Figure 1: Dinning philosophers in SLABSp

is a set of typical situations in the operation of a system in the form of a sequence of activities. Its most fundamental characteristic is to put events in the context of the history of behavior. In SLABS and SLABSp, it is extended to describe the situations in the executions of MAS as combinations of the behaviors of related agents. The basic form of scenario description in SLABSp is a pattern of an agent's behavior, which is a sequence of observable state changes and actions taken by the agent. SLABSp can also describe the situations that a specific agent behaves in a certain pattern, a number of or all agents of a caste behave in a certain pattern, and logic combinations of such situations and relational expressions that contain such descriptions. Once an agent's state is changed or an observable action is taken, the pattern hence the scenario will be evaluated to decide whether an action should be taken.

A runtime environment to support the execution of MAS has been implemented as extension of Java runtime environment. In particular, an automaton, the Pattern Process Machine [2], is designed and implemented to process patterns and scenarios. A compiler has been written to translate SLABSp programs into Java and to execute in the runtime environment. The approach to implementing the AO language facilities is to embed AO mechanisms in Java.

3. EXAMPLE

The SLABSp and its implementation has been tested on a number of examples. Figure 1 shows the program in SLABSp for the classic dinning philosophers problem. The program contains a caste named Philosopher and its five agents named P1, ..., P5, respectively. Caste Philosopher declares six actions think, eat, takeRight, takeLeft, putRight and putLeft, and two behavior rules think0 and think. The rules instruct the agent to take action think after start or eat, respectively. When action eat is taken, it triggers the action takeLeft to get the left dinner-set, and then the action takeRight to get the right one. After 5 seconds of eating, it takes action putLeft and putRight to return the dinner-sets. Each of agent P1, ..., P5 joins caste Philosopher, and defines an additional behavior rule eat so that it takes action eat after action think has been taken when both of its neighbors have not taken the dinner-sets between them. Figure 2 is a screen snapshot of the execution of the program.

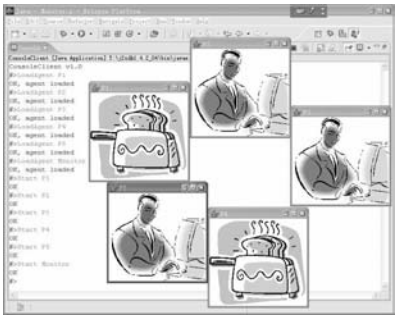

Figure 2: Screen snapshot of dinning philosophers

It is worth noting that there is no explicit message-based communications between agents in the dinning philosophers program. Our experiences in SLABSp programming show that caste and scenario can provide a powerful means of abstraction in AO software development.

4. CONCLUSION

The design and implementation of SLABSp demonstrated that caste and scenario are feasible as programming language facilities. Our experiences and experiments with the language clearly showed that they can provide power abstractions for AO programming. In particular, the caste facility enables the modularity in the concept of agents to be realized directly and in full strength. An obvious advantage of using scenarios to define agents' behaviors is that it can significantly reduce the unnecessary explicit message-based communications among agents. This also enables AO programming at a very high level of abstraction.

5. REFERENCES

[1] B. Moulin and M. Brassard. A scenario-based design method and environment for developing multi-agent systems. In *Proceedings of First Australian Workshop on DAI*, volume 1087 of *LNAI*, pages 216–232, 1996.

[2] R. Shen, J. Wang, and H. Zhu. Scenario mechanism in agent-oriented programming. In *Proceedings of APSEC'04*, pages 464–471, Busan, Korea, 2004.

[3] H. Zhu. Scenario analysis in an automated requirements analysis tool. *Journal of Requirements Engineering*, 5(1):2–22, 2000.

[4] H. Zhu. The role of caste in formal specification of MAS. In *Proceedings of PRIMA'2001*, volume 2132 of *LNCS*, pages 1–15, 2001.

[5] H. Zhu. SLABS: A formal specification language for agent-based systems. *International Journal of SEKE*, 11(5):529–558, 2001.

[6] H. Zhu and D. Lightfoot. Caste: A step beyond object orientation, in modular programming languages. In *Proceedings of JMLC'2003*, volume 2789 of *LNCS*, pages 59–62, 2003.