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.

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 confictions 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 confictions 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]...
agents named \( P_1 \) through \( P_5 \). The program contains a caste named \( \text{Philosopher} \) and its five agents named \( P_1 \) through \( P_5 \), respectively. Caste \( \text{Philosopher} \) declares six actions \( \text{think} \), \( \text{eat} \), \( \text{takeLeft} \), \( \text{putRight} \) and \( \text{putLeft} \), and two behavior rules \( \text{think0} \) and \( \text{think} \). The rules instruct the agent to take action \( \text{think} \) after \( \text{start} \) or \( \text{eat} \), respectively. When action \( \text{eat} \) is taken, it triggers the action \( \text{takeLeft} \) to get the left dinner-set, and then the action \( \text{takeRight} \) to get the right one. After 5 seconds of eating, it takes action \( \text{putLeft} \) and \( \text{putRight} \) to return the dinner-sets.

Each of agent \( P_1, \ldots, P_5 \) joins caste \( \text{Philosopher} \), and defines an additional behavior rule \( \text{eat} \) so that it takes action \( \text{eat} \) after action \( \text{think} \) 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](image)

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


