Agent Oriented Programming based on SLABS *

Ji Wang and Rui Shen
National Laboratory for
Parallel and Distributed Processing
Changsha, P. R. China
jiwang@mail.edu.cn, shenrui98@yahoo.com

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

Abstract

SLABS is a formal specification language designed for modular and composable specification of multi-agent systems. This paper reports our attempts to support SLABS at the level of programming languages. A programming language, SLABSp, is presented to support two distinguished mechanisms, namely caste and scenario, in caste-centric methodology of agent-oriented software development. Based on Java platform, the SLABSp has been implemented by compiling the programs into Java with the multi-agent runtime environment.

1. Introduction

Agent technology has been an active research area in academy and industry in the past two decades [10, 11]. A number of agent-based software systems have been developed and on the service of real applications. In recent years, many existing researches have been engaged on agent oriented software development methodologies to address the problems in the analysis, specification and design of agent-based systems. However, the programming languages based on agent oriented methodologies have not been explored as desired, especially from the perspective of software engineering. Almost all agent-based systems are developed in programming languages of other paradigms such as in object-oriented, logic or functional programming languages. The main features of agents and multi-agent systems are not effectively or efficiently supported by such programming languages to fully realize their advantages. For example, it is awkward and clumsy to implement agent's capability of collaboration with each other in complicated protocols, the perception of situations in the environment and taking actions actively and proactively, etc. by using message based communications and object method invocations in object-oriented language facilities. The aim of the research reported in this paper is to design and implement new language facilities that directly support agent orientation. It reports a programming language called SLABSp.

SLABSp is based on the formal specification language SLABS [13, 14, 15, 16], which was designed for modular and composable specifications of multi-agent systems, where agents are the active and persistent computational entities that encapsulate data, operations and behavior protocols and are situated in their designated environments. In SLABS' meta-model, caste was proposed as the classifier of agents. A caste defines a collection of agents that have the same behavior and structural characteristics. Scenario was proposed as a language facility that describes the situation of the agent's environment so that agent's behaviors can be defined in the context of environment situations. Therefore, the concepts of caste and scenario play a crucial role in the caste-centric methodology presented in SLABS as an approach to agent oriented software development. It is desirable to introduce and implement caste and scenario mechanisms in agent-oriented programming languages. An initial solution to the mechanism of scenario in programming languages has been presented in [8].

In this paper, we propose a programming language SLABSp to support caste and scenario mechanisms in agent-oriented programming, and reports its implementation. The remainder of the paper is organized as follows. Section 2 presents the SLABSp programming language with the focus on the novel language facilities, castes and scenarios. Section 3 describes the implementation of SLABSp, including the compiler and the runtime support. The examples are demonstrated in section 4. A comparison of the related work is given in section 5. Finally, section 6 concludes the paper with a summary of the main contributions of the work reported in the paper and a discussion of future work.

* 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.

Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)
0730-3157/05 $20.00 © 2005 **IEEE**

2. SLABSp programming language

SLABSp extends the Java programming language aiming at supporting the caste-centric methodology of agent oriented software development [14], whose key concepts are castes and scenarios.

2.1. Program structure

Agent/Caste SLABSp regards a multi-agent system as a set of agents. The agents are defined as encapsulations of states, actions and behavior rules, and each agent has its own rules that govern its behaviors. SLABSp organizes agents in the system into castes. Just as classes in object oriented languages to abstract a set of objects with the same pattern of data and methods, castes are designed to abstract a set of agents with the same characteristics of states, actions, behaviors and environments. However, in contrast with that an object is bound to its class statically and persistently, an agent is desired to be bound to its castes dynamically, i.e. it may join to or quit from a caste at runtime. The concept 'caste' has been presented in [14] and has been examined in [13, 16] to justify its feature as a step beyond object orientation.

Each agent can join multiple castes. When an agent joins a caste, it will copy all elements of the caste, including the states, actions, and behavior rules. Currently, when multiple castes are joined, the name/behavior conflictions of these elements should be avoided. The environment of an agent is the set of agents in the system that can affect its behavior.

The EBNF definition of SLABSp is given below. Note that 'Java-Import' is the same as Java's import declaration, and 'Java-Definition' can be any declaration clause of Java, such as class declaration and method declaration. 'Java-Code' is the sequence of Java statements, with the keywords '#' and '@' to indicate agent's state and action elements, respectively.

```
Agent ::=
   (Java-Import)*
   'agent' name ['join' Caste-Id (',' Caste-Id)*]
        '{'(Element | Java-Definition)* '}'

Caste ::=
   (Java-Import)*
   'caste' name ['join' Caste-Id (',' Caste-Id)*]
        '{' (Element | Java-Definition)* '}'

Element ::=
      State-Element | Action-Element | Behavior-Element
```

State A state element can be 'internal' to the agent or be observable for other agents. A state element must have read and write operations, i.e. 'Getf' and 'Setf' clauses, for the representation of complex, multi-dimensional values or objects.

```
State-Element ::=
   ['internal'] 'state' Type id '(' Parameter-List ')'
      '{' ( Java-Definition | Getf | Setf )*'}'

Getf ::= 'get' '{' Java-Code '}'

Setf ::= 'set' '{' Java-Code '}'
```

Action An action element can also be 'internal' to the agent or be observable for other agents. The 'do' clause will be executed when the action is invoked.

```
Action-Element ::=
   ['internal'] 'action' id '(' Parameter-List ')'  '{'
      'do' '{' Java-Code '}'
      (Java-Definition)*
   '}'
```

Behavior rules A behavior element describes a behavior rule. The 'do' clause of a behavior element will be executed if the scenario specified in 'when' clause is satisfied.

```
Behavior-Element ::=
   'behavior' id '{'
      'do' '{' Java-Code '}'
      (Java-Definition)*
   '}' 'when' '{' [ Scenario ] '}'
```

Scenarios Scenario presented in [4, 12, 14] is employed to describe a set of typical combinations of the behaviors of related agents in a multi-agent system. Its most fundamental characteristics is to put events in the context of the history of behavior. A basic form of scenario description is a pattern of an agent's behavior. In SLABSp, the description of scenarios allows the reference of the observer agent itself by 'this'.

```
Scenario ::=
      Agent-Id ':' Pattern
   |  Relation-Expression
   |  'for' (number | 'all') Caste-Id ':' Pattern
   |  Scenario 'and' Scenario
   |  Scenario 'or' Scenario
   |  'not' Scenario
   |  '(' Scenario ')'
   |  'this' ':' Pattern
```

A scenario can describe the situations that a specific agent behaves in a certain pattern, a number of or all agents of a caste behave in certain pattern, and logic combinations of such situations and relational expressions that contain such descriptions. Pattern is used to specify the sequence of observable state changes and observable actions. Once an agent's state is changed or an observable action is taken, the pattern will be evaluated by a Pattern Process Machine to decide whether an action should be taken; see [8] for details. The atomic action 'any' can be matched by any actions, and the 'id' can be matched by the action whose name is the same as 'id'.

Pattern ::= '[' Sequence-Unit (',' Sequence-Unit)* ']'

Sequence-Unit ::=
    Action-Pattern | '!' State-Assertion

Action-Pattern ::= Atomic-Action ['^' number]

Atomic-Action ::=
    'any' | id | id '(' Parameter-Value-List ')'

2.2. Core castes

The caste language facility enriches the expressiveness and scalability of agent programs. The following examples show how caste can be used to define various other language facilities in SLABSp. These castes can be regarded as the core library of all applications of multi-agent systems written in SLABSp.

The core caste Agent shown in Figure 1 defines the basic actions of agents that enable them to start executions on the runtime platform. The internal state name represents the agent's name. The internal state started shows whether the agent has started running. Behavior rule fireStartup makes the agent take action start when it starts running on the platform, which can be observed by other agents using scenarios. Each agent in SLABSp joins caste core.Agent either explicitly or implicitly.

```
// base caste every agent joins
caste core. Agent {
   // the name of the agent as a state
   internal state String name(){
      get { return getAgent().getName(); }
      set { /* name is read-only*/ }
   }
   // whether the agent has been started
   internal state boolean started(){
      boolean v = false;
      get { return v; }
      set { v = value; }
   }
   // start action
   action start(){
      do { #started() = true;}
   }
   // rule: when agent start, fire start action
   behavior fireStartup(){
      do { @start(); }
   } when {
      this: [! #started() = false]
   }
}
```

Figure 1. Base caste for all agents.

As shown in Figure 2, the core caste Mutable declares two actions (joinCaste and quitCaste), which use methods of the underlying Java classes to accomplish dynamic caste joining and quitting. Agents of core.Mutable has the ability to join and quit castes at runtime.

The core caste Social defines a caste that provides a direct communication mechanism as shown in Figure 3. It declares two actions (send and recv) to send and receive messages respectively, which can be implemented by using Java

```
// agents of this caste can dynamically
// join/quit castes
caste core.Mutable {
  // join caste action
   action joinCaste(String casteName){
      do {
         getAgent().dynamicJoin(casteName);
      }
   }
   // quit caste action
   action quitCaste(String casteName){
      do {
         getAgent().dynamicQuit(casteName);
      }
   }
}
```

Figure 2. Caste core.Mutable.

libraries. Other communication mechanisms can also be defined similarly to implement direct communication between agents, such as message passing, remote procedure call, file system, email service and etc. However, the uses of such communication mechanisms are not encouraged. As shown in section 4, the scenarios and behavior rules provide a communication mechanism at a higher level of abstraction and more suitable for agent-oriented style of programming.

```
// agents of this caste can communicate
// directly with other social agents
caste core.Social {
  // send message
   action send(Message message){
      do {
         getAgent(). send (message);
      }
   }
   // receive message
   action recv(Message message){
      do {
         getAgent().receive (message);
      }
   }
}
```

Figure 3. Caste core.Social

3. Compiler and runtime platform

The system supporting SLABSp language is based on Java. It includes the SLABSp library, the SLABSp compiler, the underlying classes (Java Agent Components), and the SLABSp runtime platform, as shown in Figure 4.

3.1. SLABSp library

The SLABSp library contains a set of standard castes defined in SLABSp language, such as the core castes discussed in section 2.2. These castes are either for defining common states, actions and behaviors of specific kinds of agents, such as the core caste Agent, which is the caste every agent joins, or for wrapping some complex operations to provide high level facilities, such as the core caste Muta-

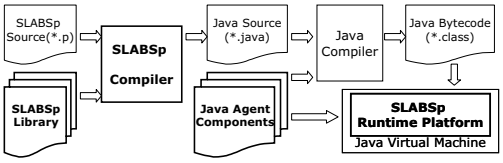


Figure 4. Overview of SLABSp.

ble, which provides actions to support dynamic caste joining and quitting.

3.2. Underlying classes

The underlying classes (Java Agent Components) are defined to serve as the semantics of SLABSp. In this model, an agent's structure can be changed at runtime, which makes dynamic caste joining and quitting possible.

In Figure 5, JacAgent represents an agent definition in SLABSp. JacCaste represents a caste definition, and it maintains a set of the agents that have joined it. JacAgent and JacCaste have the same super class JacUnit, which has a name and a set of castes to join, and maintains a composition of JacState, JacAction and JacBehavior. The listeners of JacState and JacAction can be notified when the state changes or action is invoked, driving the pattern processing in scenario mechanism. JacBehavior uses a scenario object to process the scenario declared in the 'when' clause of the behavior rule.

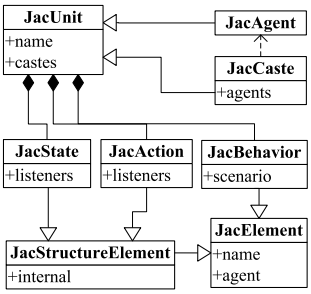


Figure 5. Main underlying classes.

Figure 6 shows the underlying scenario classes. Interface JacScenario defines the methods that all scenario classes should implement. AgentScenario processes the scenario focused on a single agent, and CasteScenario processes the scenario focused on agents of a specific caste. AndScenario, OrScenario and NotScenario process the compound scenarios.

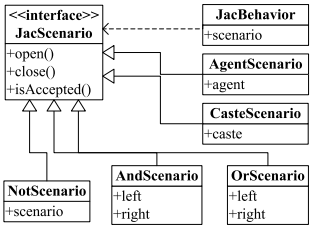


Figure 6. Underlying scenario classes.

3.3. Compiling SLABSp programs

The SLABSp compiler translates SLABSp source code together with the SLABSp library into Java. A SLABSp source file contains the declaration of exactly one agent or caste. It is compiled to a package of Java classes, which is then compiled together with Java Agent Components.

3.4. Runtime support

The runtime platform to execute SLABSp programs provides codebase management, naming service, agent lifecycle management, containers of agents and castes, dynamic caste joining and quitting support, and communication infrastructure, as shown in Figure 7.

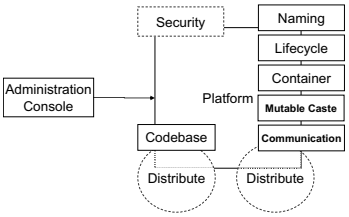


Figure 7. Runtime platform.

It manages the codebase to load necessary Java classes of the compiled agent or caste. The naming service is used to lookup the agent or caste by its qualified name, which is accomplished with the help of agent container and caste container. The agent container also manages the lifecycle of agents. When an agent dynamically joins or quits a caste, the platform should be aware of its situations, and keep everything consistent.

4. Examples

In this section, we illustrate by two examples the programming style that SLABSp supports.

4.1. Teacher or student

Agents of core.Mutable can use action joinCaste and quitCaste to join and quit a caste at runtime. In Figure 8, there is an agent named Harry of caste Person and core.Mutable. Behavior rule daytime defines when the Sun (which is an agent here) takes action rise, Harry will quit caste Student and join caste Teacher; Behavior rule night defines when the Sun takes action fall, Harry will quit caste Teacher and join caste Student.

```
// Harry is a teacher in the daytime,
// but he goes studying at night.
agent Harry join Person, core.Mutable {
   behavior daytime(){
      do {
         @quitCaste("Student");
         @joinCaste("Teacher");
      }
   } when { Sun: [ @rise() ] }

   behavior night(){
      do {
         @quitCaste("Teacher");
         @joinCaste("Student");
      }
   } when { Sun: [ @fall() ] }
}
```

Figure 8. Agent Harry in SLABSp

This simple example shows that SLABSp's programming style enable natural description of agent's perception of the environment and then take appropriate actions rather than driven by messages.

4.2. Vote backers

The backers can support one of the two candidates: Tommy and Jerry. Caste Backer declares two actions (supportTommy and supportJerry) and one behavior rule (turnRandom). Behavior rule turnRundom makes the agent randomly choose a candidate to support after it starts running, as shown in Figure 9(a).

Caste PityBacker in Figure 9(b) extends caste Backer, and declares two more behavior rules (turnToJerry and turnToTommy). Behavior rule turnToJerry makes the agent take action supportJerry when Jerry is less supported, and behavior rule turnToTommy makes the agent take action supportTommy when Tommy is less supported. Caste RitzyBacker in Figure 9(c) adopts a symmetrical strategy compared to caste PityBacker.

When there are only agents of caste Backer in the runtime platform, the support ratio is fifty-fifty. When there are only agents of caste PityBacker, because they support the weaker one, the final support ratio is also fifty-fifty. But when there are only agents of caste RitzyBacker, all the agents will support one side.

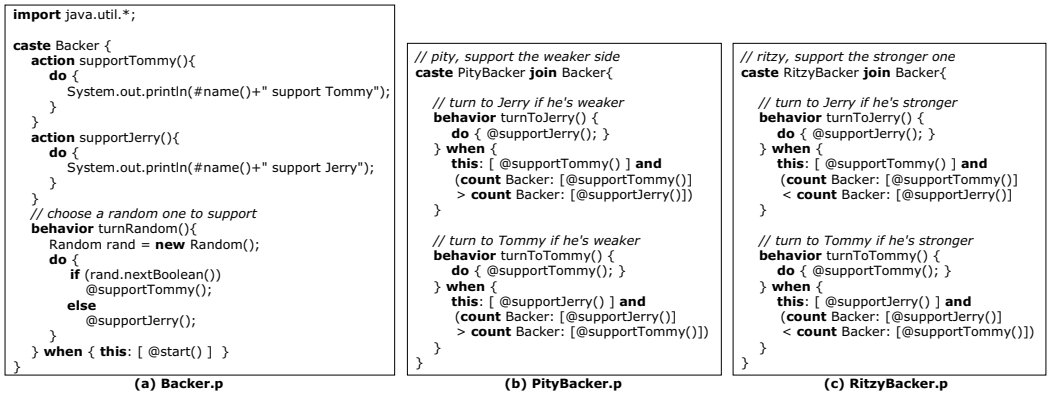
This example further demonstrates how agents perceive the behaviors of other agents to adjust their own actions accordingly. Such behaviors would be less easier to implement in object oriented languages directly especially when agents can dynamically join the system and quit from the system.

It is worth noting that in both of the above examples, direct message-based communications or method invocations are replaced by scenarios in behavior rules. The style is closer to the structured programming style using conditional branching and case-based branching, which is well understood and less complicated than synchronous or asynchronous communications. The logic of the behavior of an agent can be understood without detailed knowledge of other agents. This is enabled by the high level of abstraction of the scenario and caste language facilities.

5. Related work

Agent oriented programming languages and systems have been investigated for more than one decade since the work presented in [9], including agent architectures and agent communication languages. There exist researches to design the languages based on object-oriented programming languages such as Java. The representative one is JACK [2], which shares the component based idea with SLABSp on the implementation of agent-oriented programming language. The JACK Agent Language is a programming language that extends Java with agent-oriented concepts, such as Agents, Capabilities, Events and Plans etc. SLABSp takes a different approach, the caste-centric approach, to the extensions of object-orientation to agent-orientation. The changes are mostly at the meta-level, that is from objects to agents, from classes to castes, and from methods to scenario-based behavior rules. The principles of SLABSp are to explore the language facilities for organization of agents and capture of the behaviors of agents, which can switch object-orientation to agent-orientation in a compatible way. As a result, the conceptual level of the language design is more generic than that of the languages based on BDI model [6]. However, the idea of BDI models can still be implemented in SLABSp.

There is the tool-based approach to providing a platform including a software framework, a library of software components and tools that facilitate the development and deployment of agent based systems, such as JADE [1], DIET [3] and ZEUS Toolkit [5]. SLABSp chooses a language-based approach and can build the library of software components in castes. For example, one may write user-defined agent communication by using caste mechanism in SLABSp. While in the tool-based approach, the extensions will be carried by adding specific library in the languages in which the platform is built. Therefore, SLABSp may ease the incremental development of agent systems.



```
import java.util.*;

caste Backer {
   action supportTommy(){
      do {
         System.out.println(#name()+" support Tommy");
      }
   }
   action supportJerry(){
      do {
         System.out.println(#name()+" support Jerry");
      }
   }
   // choose a random one to support
   behavior turnRandom(){
      Random rand = new Random();
      do {
         if (rand.nextBoolean())
            @supportTommy();
         else
            @supportJerry();
      }
   } when { this: [ @start() ] }
}
```

(a) Backer.p

```
// pity, support the weaker side
caste PityBacker join Backer{

   // turn to Jerry if he's weaker
   behavior turnToJerry() {
      do { @supportJerry(); }
   } when {
      this: [ @supportTommy() ] and
       (count Backer: [@supportTommy()]
        > count Backer: [@supportJerry()])
   }

   // turn to Tommy if he's weaker
   behavior turnToTommy() {
      do { @supportTommy(); }
   } when {
      this: [ @supportJerry() ] and
       (count Backer: [@supportJerry()]
        > count Backer: [@supportTommy()])
   }
}
```

(b) PityBacker.p

```
// ritzy, support the stronger one
caste RitzyBacker join Backer{

   // turn to Jerry if he's stronger
   behavior turnToJerry() {
      do { @supportJerry(); }
   } when {
      this: [ @supportTommy() ] and
       (count Backer: [@supportTommy()]
        < count Backer: [@supportJerry()]
   }

   // turn to Tommy if he's stronger
   behavior turnToTommy() {
      do { @supportTommy(); }
   } when {
      this: [ @supportJerry() ] and
       (count Backer: [@supportJerry()]
        < count Backer: [@supportTommy()])
   }
}
```

(c) RitzyBacker.p

Figure 9. Vote backers example in SLABSp.

6. Conclusion and future work

In this paper, the programming language SLABSp is presented and implemented to demonstrate that caste and scenario are feasible as the novel facilities in agent oriented programming. The mechanism of castes is designed to organize the agents with the same pattern of states, actions, behaviors and environments. To our best knowledge, SLABSp is the first one to provide castes and to support the dynamic binding between agents and castes in programming languages. The mechanism of scenarios is designed to describe the agent's behaviors under specific environment and to support its perception to the environment. An obvious advantage is that using scenarios can reduce many unnecessary direct communications among agents in programming and achieve a powerful abstraction in programming.

We are currently working on the language support to the running of SLABSp program on distributed systems. Further linkage between the programming language and specification language SLABS and modelling language and environment CAMLE [7] is in progress to realize a relatively complete framework of caste-centric agent-oriented software development methodology.

References

[1] F. Bellifemmine, A. Poggi, G. Rimassa, and P. Turci. An object-oriented framework to realize agent systems. In *Proceedings of WOA 2000 Workshop*, pages 52–57, 2000.

[2] M. Coburn. *JACK Intelligent Agents: User Guide, version 2.0.* http://www.agent-software.com, 2001.

[3] C. Hoile, F. Wang, E. Bonsma, and P. Marrow. Core specification and experiments in DIET: A decentralised ecosystem-inspired mobile agent system. In *Proceedings of AAMAS'02*, pages 623–630, 2002.

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

[5] H. S. Nwana, D. T. Ndumu, and L. C. Lee. ZEUS: An advanced tool-kit for engineering distributed multi-agent systems. In *Proceedings of PAAM'98*, pages 377–391, 1998.

[6] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *Proceeding of the 1st International Conference on Multi-Agent Systems*, pages 312–319, San Francisco, CA, 29–30 May 1995.

[7] L. Shan and H. Zhu. CAMLE: A caste-centric agent-oriented modelling language and environment. In *Software Engineering for Multi-Agent Systems III*, volume 3390 of *LNCS*, pages 144–161. 2005.

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

[9] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[10] G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[11] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

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

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

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

[15] H. Zhu. A formal specification language for agent-oriented software engineering. In *Proceedings of AAMAS'03*, pages 1174–1175, Melbourne, Australia, 2003.

[16] H. Zhu and D. Lightfoot. Caste: A step beyond object orientation. In *Proceeding of JMLC'2003*, volume 2789 of *LNCS*, pages 59–62, 2003.