Agent Oriented Programming based on SLABS *

Ji Wang and Rui Shen
National Laboratory for
Parallel and Distributed Processing
Changsha, 410073, 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. An 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 oriented software methodology has been becoming widely accepted in academy and industry [7, 8], and agent based software systems have been on the service of real applications. Many existing researches have been engaged on agent oriented analysis, specification and design. However, the programming languages based on the agent oriented methodology have not been explored as desired in the past decades, especially from the perspective of software engineering. The aim of this paper is to investigate the approach to design and implementation of the new language facilities supporting agent orientation by showing an experimental programming language called SLABSp.

As the growth of agent oriented methodology, agent oriented programming is considered as the paradigm beyond object oriented programming. Recently, SLABS [10, 11, 12, 13] has been designed for modular and composable specification 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 the caste-centric methodology presented in SLABS as an approach to agent oriented software development, caste is proposed 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. Therefore, it is desired 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 [5].

In this paper, we propose a programming language SLABSp to support caste and scenario mechanisms in agent-oriented programming, as well as its implementation framework. SLABSp regards a multi-agent system as a set of agents and organizes the agents into castes. In SLABSp, an agent can be bounded to its castes dynamically (i.e. it may join to or quit from a caste at runtime), and can perceive other agents in its environment rather than communicate directly. With a component based approach, the SLABSp has been systematically implemented by compiling the programs into Java with the multi-agent runtime environment.

The remainder of the paper is organized as follows. Section 2 presents the SLABSp programming language, where castes and scenarios are introduced as the novel language facilities. Then section 3 describes the implementation of SLABSp, including the compiler and the runtime support. The examples are demonstrated in section 4, and a comparison of the related work is given in section 5. At last, section 6 concludes the paper with the contributions and future work.

2. SLABSp: Agent-Oriented Programming with Caste and Scenario

SLABSp is a Java-extended programming language designed to support the caste-centric methodology of agent oriented software development [11], whose key concepts

* 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.

are castes and scenarios. The approach here is to embed 'agent-oriented' description mechanisms in Java.

2.1. Language Framework

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 pattern of states, actions, behaviors and environments. However, in contrast with that an object is bounded to its class statically and persistently, an agent is desired to be bounded to its castes dynamically, i.e. it may join to or quit from a caste at runtime. The concept 'caste' has been presented in [11] and has been examined in [10, 13] 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 some special token '#' and '@' to reference state and action elements of the agent respectively.

Agent/Caste

```
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 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 Action-Element can 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 Behavior-Element describes that 'do' clause 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 [3, 9, 11] 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 scenario is extended to allow 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
```

SLABSp can also describe the situations that a specific agent behave 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 sequence will be evaluated by a Pattern Process Machine [5] to decide whether an action should be taken. The

atomic action ‘any’ can be matched by any actions, and the ‘id’ can be matched by action whose name is the same with ‘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

Caste is a nice language facility to enrich the expressiveness and scalability of agent programs in SLABSp. For examples, three core castes, namely core.Agent, core.Mutable and core.Social, are shown in this subsection.

Each agent in SLABSp joins caste core.Agent (in Figure 1) either explicitly or implicitly. Internal state name represents the agent’s name, and the other internal state started shows whether the agent has started running. Behavior rule fireStartup makes the agent take action start when it starts running in the platform, and action start can be observed by agents in their scenario patterns.

```
// 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.

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

SLABSp can be extended by defining new castes, e.g. to provide the direct communication mechanism, a caste

```
// 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.

named core.Social is defined as shown in Figure 3. It declares two actions (send and recv) to send and receive messages, which can ultimately use diverse Java libraries to implement direct communication between agents, such as message passing, remote procedure call, file system, email service and etc.

```
// 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 for SLABSp

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

The SLABSp library contains some standard castes defined in SLABSp language, for examples the castes in section 2.2. These castes are either for defining common states, actions and behaviors of specific kinds of agents, such as caste core.Agent shown in Figure 1, which is the caste every agent joins, or for wrapping some complex operations to provide high level facilities, such as caste core.Mutable in Figure 2, which provides actions to support dynamic caste joining and quitting.

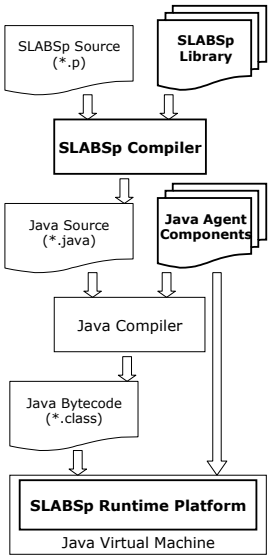


Figure 4. Overview of SLABSp.

3.1. 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 run-time, 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 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 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.

3.2. Compiling SLABSp Programs

The SLABSp compiler compiles SLABSp source code together with the SLABSp library into Java. The syntax elements of SLABSp will be compiled to subclasses of corresponding underlying classes. A SLABSp source file contains the declaration of exactly one agent or caste, and it will be compiled to a package of Java classes corresponding to the syntax elements.

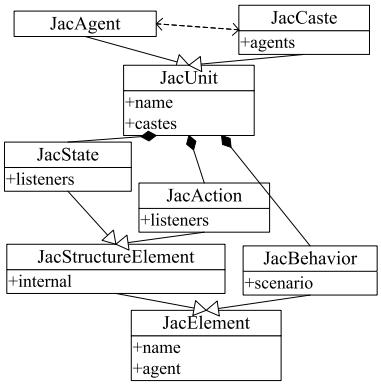


Figure 5. Main underlying classes.

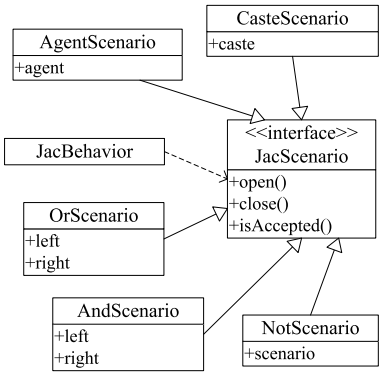


Figure 6. Underlying scenario classes.

3.3. 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.

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 demonstrate that SLABSp can make the high level concepts and powerful abstraction available in the programming level naturally.

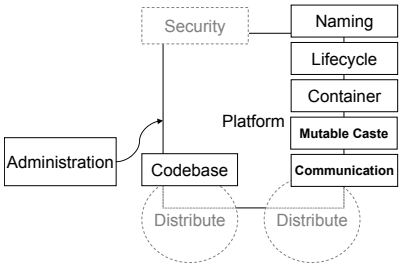


Figure 7. Runtime platform.

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

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 an 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.

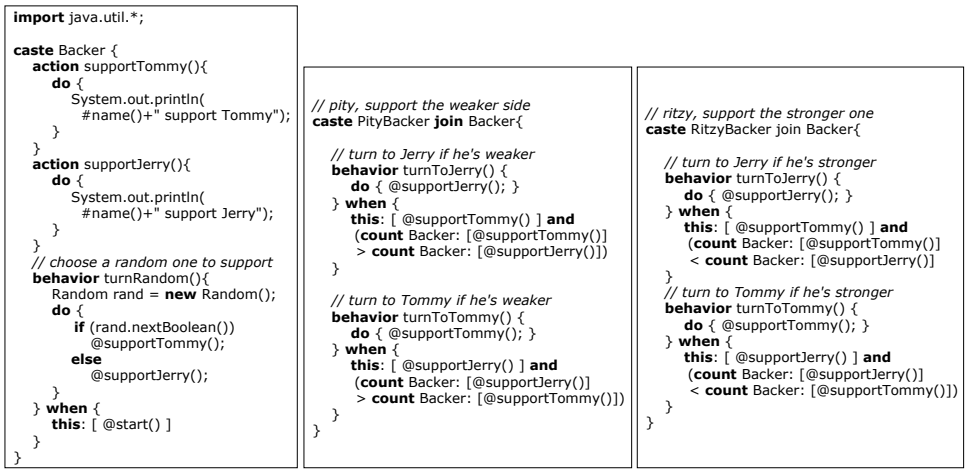
5. Related Work

Agent oriented programming languages and systems have been investigated for more than one decade since the original work presented in [6], including agent architectures and agent communication languages. From the perspective of software engineering, an alternative approach is 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. In SLABSp, it is desired to examine the extensions of object-orientation to agent-orientation steadily in a caste-centric approach, 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 'lower' than that of the languages based on BDI model. However, the idea of BDI model can still find its place in SLABSp implicitly.

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] and ZEUS Toolkit [4]. 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 specific languages in which the platform is built. Therefore, SLABSp may ease the incremental development of agent systems.

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



```
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.

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.

The future work is to support the running of SLABSp program on distributed platforms.

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] 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.

[4] H. S. Nwana, D. T. Ndumu, and L. C. Lee. ZEUS: An advanced tool-kit for engineering distributed multi-agent systems. In *Proceedings of PAAM98*, pages 377–391, 1998.

[5] R. Shen, J. Wang, and H. Zhu. Scenario mechanism in agent-oriented programming. In *Proceedings of 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 464–471, Busan, Korea, 30 Nov - 3 Dec 2004.

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

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

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

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

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

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

[12] H. Zhu. A formal specification language for agent-oriented software engineering. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS'03)*, pages 1174–1175, Melbourne, Australia, 2003.

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