Caste-Centric Agent-Oriented Programming *

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

The paper presents a caste-centric approach to agent-oriented programming by introducing SLABSp language. The fundamental concepts of caste-centric methodology, caste and scenario, as well as environment descriptions, are available as language facilities in SLABSp in a coherent way. In SLABSp programming, agents are organized into castes to represent their structure and behavior characteristics, and their behaviors are defined by scenarios and rules in the context of their environment. The relations between agents and castes are bound at runtime, and the perceptions and interactions between agents are supported with scenarios and behavior rules. Two selected SLABSp programs are demonstrated to illustrate the programming style.

1. Introduction

Recently, 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 supports to agent-oriented methodologies have not been explored as desired from the perspective of software engineering. Currently, many agent-based systems are developed in programming languages of other paradigms, such as in object-oriented, logic or functional programming languages. The high level concepts of the methodologies can not be transformed to programs naturally and smoothly, and therefore the main features of agents and multi-agent systems are not effectively or efficiently supported by such programming languages to fully realize their advantages.

* Supported by the National Key Basic Research and Development Program (973) of China under Grant No. 2005CB321802, the National NSF of China under grant No. 60233020, the National High Technology R&D 863 Programme of China under grant No. 2005AA113130, and Program for New Century Excellent Talents in University.

The aim of the research reported in this paper is to investigate the new paradigm of programming, caste-centric agent-oriented programming, by designing and implementing new language facilities that directly support caste-centric methodology for agent-oriented software development [11, 12, 13, 15], where the concepts of caste and scenario play the crucial roles. In this methodology, agents are the active and persistent computational entities that encapsulate data, operations and behavior protocols and are situated in their designated environments. As the classifier of agents in the meta-model, a caste defines a collection of agents that have the same behavior and structural characteristics. Scenario is the facility that describes the situation of the agent's environment so that agent's behaviors can be defined in the context of environment situations. The idea is to make the models and specification of agent-based systems be transformed to the programs in a direct way.

To support the caste-centric methodology, the programming language SLABSp is proposed, as a companion of SLABS [12] and CAMLE [6], which are designed for modular and composable specification and modeling of multi-agent systems. The key elements in the modeling and specification are directly available at the level of programming. In [7, 9, 10], we examined the feasibility of caste and scenario mechanisms in programming on a non-distributed platform. Compared to these work, SLABSp is extended to incorporate the declaration of the environment of agents explicitly, and a 'where' clause in behavior rules, which is intended to specify the perception of the environment reference to the agents of different castes. In addition, the runtime platform has been transferred to a distributed environment instead of a single JVM in the previous implementation. Two interesting examples, autonomous sorting and vote backers, are used to illustrate the programming style of SLABSp. One may notice that the programming style of agent orientation in SLABSp has been significantly different to the traditional object-orientation. The code of the examples is much clear and easier to understand and maintain

Proceedings of the Fifth International Conference on Quality Software (QSIC'05)
1550-6002/05 $20.00 © 2005 IEEE

than object-oriented code.

The remainder of the paper is organized as follows. Section 2 presents the fundamental concepts of cast-centric agent-oriented programming, with the focus on the novel language facilities, caste, scenario and environment. Section 3 describes the overview of SLABSp, including its language constructs and agent structures. The examples are demonstrated in section 4. Section 5 compares the related work, and concludes the paper with discussions of the future work.

2. Fundamental concepts

Caste, scenario and environment are the fundamental concepts in caste-centric agent-oriented programming, and have been supported as language facilities of SLABSp. Derived from the caste-centric meta-model of agent-oriented systems [12], they can represent software systems which can closely reflect the structures of the real world systems.

2.1. Caste

In caste-centric agent-oriented programming, a multi-agent system is regarded as a set of the agents, which are defined as encapsulations of states, actions and behavior rules. Each agent has its own rules that govern its behaviors. In the system, agents are organized into castes. Caste 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 object-oriented methodology 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 or quit a caste at runtime. In addition, a caste may extend the other castes. Caste `A` extends caste `B` means that all agents in caste `A` have all structural, behavioral and environmental features specified by caste `B`. The concept 'caste' has been presented in [12] and has been examined in [11, 15] 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 its states, actions, and behavior rules. Currently, when multiple castes are joined, the name/behavior conflictions of these elements should be avoided.

The notion of roles in the other agent-oriented methodologies can be described by castes directly. For instances, agents that play the same role can be defined by a caste. Caste is not only a well-define abstract notion, but also a language facility available in programming.

2.2. Scenario

Scenario is used 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 direct message communications.

In general, the notion of scenario as presented in [4, 12] 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 caste-centric agent-oriented programming, it is extended to describe the situations in the executions of a multi-agent system as combinations of the behaviors of related agents. The basic form of scenario description is a pattern of an agent's behavior, which is a sequence of observable state changes and actions taken by the agent. One 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 observable action is taken, the pattern hence the scenario will be evaluated, and together with the evaluation of the pre-condition which is used to percept the environment reference to several agents of difference castes, to decide whether an action should be taken.

2.3. Environment

The environment of an agent is the set of agents in the system that may affect its behaviors. Since the related agents may change during time, environment references are used to follow these changes. The use of scenarios in conjunction with agents' visible actions and environment descriptions enables communications and collaborations among agents to be described at a high level of abstraction and in the same style of conditional expressions in structured programming, e.g. in the '`where`' clause in behavior rules, which is the pre-condition of the action to be taken by the agent.

3. SLABSp overview

SLABSp is a programming language designed to support the caste-centric approach to agent-oriented programming by extending the object-oriented programming language Java.

Based on the work of [7, 9, 10], SLABSp has been extended to support explicit environment descriptions, pre-condition in '`where`' clause, and distribution of agents among networked nodes. The SLABSp compiler translates its programs into Java, which can run within the distributed multi-agent runtime environment.

3.1. Language constructs

The main language constructs of SLABSp are shown in Figure 1 in EBNF. As an extension to Java grammar, many Java language constructs are involved, such as *Java-Import*, *Conditional-Expression*, *Statement* and etc., with some alteration of identifiers to reference agent's state and action preceded by '#' and '∼' respectively, and the additional 'join' and 'quit' statements on the purpose of dynamic caste joining and quitting.

```
Caste ::= { Java-Import }
      'caste' Name [ ':' { Name / ',' }+ ] '{'
            { Environment }
            { State | Action | Rule }
      '}'
Agent ::= { Java-Import }
      'agent' Name [ ':' { Name / ',' }+ ] '{'
            { Environment }
            { State | Action | Rule }
      '}'
Environment ::= Name Id ';'

State ::= [ 'internal' ] Type '#'Id '(' Formal-Parameters ')' '{'
            { Java-Definition }
            'get' ':' Statement
            [ 'set' ':' Statement ]
      '}'
Action ::= [ 'internal' ] '~'Id '(' Formal-Parameters ')' '{'
            { Statement }
      '}'
Rule ::= 'rule' Id '(' Formal-Parameters ')'
      [ 'when' '(' Scenario ')' ]
      [ 'where' '(' Conditional-Expression ')' ]
      'do' '{' { Statement } '}'

Scenario ::= ( Name | 'self' ) Pattern                    (a)
      |  '<' [Number] ':' [Number] '>' Name Pattern       (b)
      |  Count-Conditional-Expression                     (c)
      |  Scenario ',' Scenario                            (d)
      |  Scenario '|' Scenario                            (e)
      |  '!' Scenario                                     (f)
      |  '(' Scenario ')'                                 (g)
Count-Expression ::= '*' Name Pattern
Pattern ::=
      '\' { ( Action-Pattern | State-Assertion )  / ',' } '\'
Action-Pattern ::=
      ('~' | '*' | '~'Id '(' Parameters ')' ) ['^' Number]
State-Assertion ::= Conditional-Expression
```

Figure 1. EBNF syntax definitions.

Caste A caste in SLABSp is declared by using keyword 'caste', followed by its name, and (optional) names of its super castes. The main body of a caste definition contains declarations of the environment references, state elements, action elements, and behavior rules.

Agent Similar to the declaration of a caste, an agent in SLABSp is declared by using keyword 'agent', followed by its name, and (optional) names of the castes it joins initially. The main body of an agent definition contains declarations of the environment references, state elements, action elements, and behavior rules.

Environment The environment of an agent/caste is declared by an identifier and its type name, which can be either the name of a caste or the name of an agent. The environment reference can be accessed and modified in behavior rules of the agent.

State To be distinguished from normal identifiers of Java, identifiers of state elements are preceded by '#'. If the optional keyword 'internal' is present, the state element can only be accessed by the owner agent, otherwise it can be observed by other agents. Each state has a result type and several (or none) formal parameters, and together with the *Java-Definition*s in the body of state declaration, it can express complex and multi-dimensional values, such as arrays and matrices. The statements following keyword 'get' and 'set' leave the customizations of the read and write operations to programmers. If there is no 'set' statement present, the state is declared readonly.

Action Identifiers of action elements are preceded by '∼' for the same reason of explained above. If the optional keyword 'internal' is present, the action element can only be visible to the owner agent, otherwise it can be visible to other agents. The formal parameters are used on the purpose of parameterized action invocation, and when invoked, the statements inside the body of the declaration will be executed.

Rule A behavior rule is declared by the keyword 'rule' with an identifier to name it. Formal parameters of a rule element declare variables that can be bound and accessed by scenarios, expressions and statements within the rule. The 'when' keyword is followed by scenario declaration, and the 'where' keyword is followed by pre-condition. The statements declared in the body of 'do' keyword will be executed when the scenario happens and the pre-condition is satisfied.

Scenario There are several different scenario definitions as shown in Figure 1. An expression in the form of (a) describes the situation that a specific agent behaves in a certain pattern, where the agent is referred to by its name or keyword 'self'. Expressions in the form of (b) describe the situation that the number of agents of a caste that behave in a certain pattern is within a specified interval, where the interval's boundaries are optional. The default value of the left boundary (i.e. the lower number) is 'zero'. When the right boundary is absent, it means 'all', i.e. the size of all the caste. The *Count-Conditional-Expression* in the form of (c) is an extension of Java *Conditional-Expression* with *Count-Expression* to evaluate the number of agents in a caste that behave in the pattern. Expressions in the form of (d), (e) and (f) are the logic 'and', 'or' and 'not' combination of scenarios in the above forms, respectively. Expressions in the form of (g) are used to change the preference of the logic combinations.

Pattern A pattern is used to specify the sequence of observable states related conditional expressions (*State-Assertion*s) and observable actions' invocations (*Action-Patterns*). There are two special action pattern, '$\sim$' and '$*$', which respectively stand for the silent action and the wildcard matching all actions. When the action identifier is provided, variables declared in the behavior rule's formal parameters can be bound when they are followed by the '?' mark, and then they can be accessed like normal variables. The continuous repeat of an action pattern can be simply expressed by suffixing a '^' mark with the number of repeat times.

3.2. Agent structures

The structure of an agent is illustrated in Figure 2, which contains the environment references, internal states and actions, observable states and actions, and behavior rules.

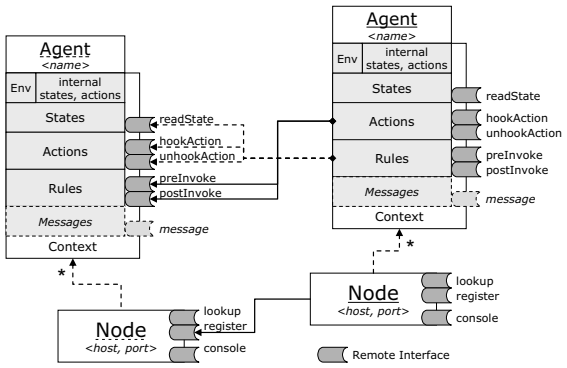

Figure 2. Agent structures in the distributed multi-agent runtime environment.

In SLABSp, the multi-agent runtime environment leverages Java RMI to support distributed execution. The remote interfaces shown in Figure 2 denotes the remote methods for RMI purpose, which are used by the underlying object model of the multi-agent system.

A node serves as the container of physically aggregated agents, and provides remote methods such as lookup (to lookup an agent's remote handle by its name), register (to register an agent or another node) and console (to administrate the node via a console client).

The remote methods of an agent are used to support the scenario processing. Other agents' behavior rules can use method readState to read the values of observable states of this agent, and register/unregister to watch observable action invocations by hookAction/unhookAction. When an observable action is taken, preInvoke/postInvoke of registered watchers will be called, hence related behavior rules will be informed.

For each caste in SLABSp, there is a corresponding special agent with several additional remote methods (getCaste, agentJoined, and agentQuitted) and utility states (#population, #populate, and etc.) to support caste joining/quitting and scenarios involving castes. This special agent provides its caste object via getCaste, and tracks the membership of its agent by agentJoined and agentQuitted. The state #population presents the total number of its agents, and state #populate returns one of its agent randomly.

4. Programming in SLABSp

In this section, two examples are selected to illustrate the programming style supported by SLABSp. They reflect the features of programming for internet computing.

4.1. Autonomous sorting

The formal specification of autonomous sorting has been investigated in [14]. The SLABSp program of this example is shown in Figure 3.

```
caste Linker {
   Linker higher;
   Linker lower;

   int #value() { int v; get: return v; set: v = value;}

   rule IL(Linker A)
      when ( Mediator\~introduce(self, ?A)\ )
      where ( (A#value() < #value())
         && ( lower == null || lower#value() < A#value()) )
      do { lower = A; }

   rule IH(Linker A)
      when ( Mediator\~introduce(self, ?A)\ )
      where ( (A#value() > #value())
         && ( higher == null || higher#value() > A#value()) )
      do { higher = A; }

}

agent Mediator {
   ~introduce(Linker A, Linker B){
      System.out.println("I introduce " + A + " to " + B);
   }

   rule I() when ( self\~\ ) do {
      Linker A = Linker#populate();
      Linker B = Linker#populate();
      ~introduce(A, B);
   }
}
```

Figure 3. Linker example.

Caste Linker has two environment references (higher and lower) which are both of type Linker, a state #value wrapping an integer value, and two behavior rules IL and IH. Since #value is not defined as an internal state, it can be observed by other agents.

Agent Mediator has an action ~introduce which introduces two agents of caste Linker, and a behavior rule I which ran-

domly populate two agents from caste Linker and invoke action ∼introduce with them periodically.

Behavior rule IL indicates that, when Mediator takes action ∼introduce with the owner agent as the first parameter, the variable A declared in its formal parameters will be bound to the second parameter, and when the state #value of A is lower than that of the owner agent, and if the environment reference lower is null or the state #value of lower is even lower than that of A, the environment reference lower is set to A, i.e. it introduces an agent with a nearer lower #value to the owner agent. Symmetrically, behavior rule IH introduces an agent with a nearer higher #value.

After instantiating agent Mediator and numerous agents of caste Linker with different #value, these agents will eventually form an ordered list linked by higher and lower.

This example shows that SLABSp's programming style enables natural description of agent's perception of the environment and then taking appropriate actions rather than driven by messages. What is more important, is that the desired result of the program (say sorting) is achieved by the emergent behavior of a set of agents which behave individually according to its designated environment with partial information. The feature is essential in programming for Web applications.

4.2. Vote backers

With the enhancement of SLABSp, the vote backers example mentioned in [9] can now be more efficiently programmed. As shown in Figure 4, the backers can support one of the two candidates: Tommy and Jerry. Caste Backer declares an action ∼support with a parameter specifying whom to support. Caste TommyBacker groups the agents who support Tommy, while caste JerryBacker groups those who support Jerry. They both extend caste Backer and declare a behavior rule support to periodically invoke action ∼support with Tommy and Jerry as the parameter respectively.

Caste PityBacker extends caste Backer, and declares two additional behavior rules (supportJerry and supportTommy). Behavior rule supportJerry makes the agent quit caste TommyBacker and join caste JerryBacker when Jerry's backers is less then Tommy's, while behavior rule supportTommy makes the agent quit caste JerryBacker and join caste TommyBacker when Tommy's backers is less then Jerry's. Caste RitzyBacker adopts a symmetrical strategy compared to caste PityBacker.

When there are only agents of caste PityBacker in the system, because they support the weaker candidate, the final support ratio is fifty-fifty. But when there are only agents of caste RitzyBacker, all the agents will support the lucky one at last.

This example gives a sample of simulation of social activity and further demonstrates how agents perceive the behaviors of other agents to adjust their own actions accordingly, and the ability to join and quit castes dynamically. Such behaviors would be harder to implement in object-oriented languages directly especially when agents can dynamically join and quit the system.

```
caste Backer {
      ~support(String whom){
            System.out.println("I support " + whom);
      }
}

caste TommyBacker : Backer{
      rule support() when ( self\~\ )
      do { ~support("Tommy"); }
}
caste JerryBacker : Backer{
      rule support() when ( self\~\ )
      do { ~support("Jerry"); }
}

caste PityBacker : Backer{
      rule supportJerry()
      when ( self\~support("Tommy")\
            ,*Backer\~support("Tommy")\ > *Backer\~support("Jerry")\)
      do { quit TommyBacker; join JerryBacker; }

      rule supportTommy()
      when ( self\~support("Jerry")\
            ,*Backer\~support("Jerry")\ > *Backer\~support("Tommy")\)
      do { quit JerryBacker; join TommyBacker; }
}
caste RitzyBacker : Backer{
      rule supportJerry()
      when ( self\~support("Tommy")\
            ,*Backer\~support("Tommy")\ < *Backer\~support("Jerry")\)
      do { quit TommyBacker; join JerryBacker; }

      rule supportTommy()
      when ( self\~support("Jerry")\
            ,*Backer\~support("Jerry")\ < *Backer\~support("Tommy")\)
      do { quit JerryBacker; join TommyBacker; }
}
```

Figure 4. Backer example.

As shown in the above two examples, agents in a multi-agent system need to communicate with others, thus cooperation and coordination among agents are possible. In SLABSp, agents can cooperate and coordinate by using scenarios in behavior rules, which provide a communication mechanism at a higher level of abstraction and more suitable for agent-oriented style of programming. However, direct message-based communication between agents can also be supported in SLABSp for the sake of expressiveness and flexibility.

5. Discussions

5.1. Related work

Agent-oriented programming languages and systems have been investigated for more than one decade since AGENT-0 [8]. Many of them are from the perspective of artificial intelligence, typically based on BDI model with reasoning support, e.g. 3APL [2], AgentSpeak(L) [5], JACK [1], JAM [3]. An agent in SLABSp achieves its goals by reacting to its environment via behavior rules. The conceptual level of the language design is different from that of

the languages based on BDI model. It provides a new language framework for programming multi-agent systems. 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.

There are some researches of agent-oriented programming from the perspective of software engineering, which can be categorized into language based approach and tool based approach. SLABSp takes the language based approach. One of the representatives of the programming language approach is JACK, which shares the component based idea with SLABSp on the implementation of agent-oriented programming language. The JACK Agent Language (JAL) is a programming language that extends Java with agent-oriented concepts, such as Agents, Capabilities, Events and Plans etc. SLABSp takes a different, the caste-centric approach to agent-orientation. Firstly, JAL uses Java interfaces to categorize agents, while SLABSp uses castes as the classifier of agents. Since the interfaces can not be changed during runtime, the agents is categorized statically, thus the whole system need to be rebooted if there is any change in the categorizing of agents. In SLABSp, agents can join or quit castes dynamically, so that the online evolution of the system can be exploited. Secondly, JAL uses events to fire actions of related agents, while SLABSp use scenarios in behavior rules to trigger actions of an agent. Since an agent's behavior depends on the events emitted by other agents, its autonomy is restricted by other agents. In SLABSp, an agent can decide what to do totally by its perception of other agents with behavior rules, thus the coupling of agents is much looser and the system can be more easily maintained.

5.2. Conclusions and future work

The paper presents a programming language SLABSp to support the caste-centric approach to agent-oriented programming. The fundamental concepts of the methodology, caste and scenario, as well as environment descriptions, are available as language facilities in SLABSp in a coherent way. In SLABSp programming, agents are organized into castes to represent their structure and behavior characteristics, and their behaviors are defined by scenarios and rules in the context of their environment. The relations between agents and castes are bound at runtime, and the perceptions and interactions between agents are supported with scenarios and behavior rules. With scenarios and behavior rules, a 'guard'-alike description can be employed to encapsulate and reduce the complex direct communications among agents. SLABSp has been implemented over a distributed platform with Java RMI. Autonomous sorting and vote backers are used to illustrate the high level programming abstraction supported.

Further investigations of the features and benefits of caste-centric agent-oriented programming are in progress to realize a complete framework for a new programming paradigm.

References

[1] *JACK Intelligent Agents: Agent Manual, version 5.0*. http://www.agent-software.com, 2005.

[2] K. Hindrikis, F. de Boer, W. van der Hoek, and J. J. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[3] M. J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of The Third International Conference on Autonomous Agents*, pages 236–243, 1999.

[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] A. Rao. AgentSpeak(L): BDI agent speak out in logical computable language. In *Proceedings of 7th European Workshop on MAAMAW*, pages 42–55, 1996.

[6] 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.

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

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

[9] J. Wang, R. Shen, and H. Zhu. Agent oriented programming based on SLABS. In *Proceedings of COMPSAC'05*, Edinburgh, Scotland, July 25-28 2005.

[10] J. Wang, R. Shen, and H. Zhu. Towards an agent oriented programming language with caste and scenario mechanisms. In *Proceedings of AAMAS'05*, Utrecht, Netherland, July 25-29 2005.

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

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

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

[14] H. Zhu. Towards formal reasoning about emergent behaviors in MAS. In *Proceedings of SEKE'05*, Taipei, Taiwan, July 14-16 2005.

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