The Role of Castes in Formal Specification of MAS

Hong Zhu

School of Computing and Mathematical Sciences, Oxford Brookes University
Gipsy Lane, Headington, Oxford, OX3 0NW, UK
Email: hzhu@brookes.ac.uk

Abstract. One of the most appealing features of multiagent technology is its natural way to modularise a complex system in terms of multiple, interacting and autonomous components. As a natural extension of classes, castes introduced in the formal specification language SLAB provide a language facility that provides modularity in the formal specification of multiagent systems. A caste represents a set of agents of common structural and behavioural characteristics. A caste description defines the tasks that the agents of the caste are capable of, the rules that govern their behaviour, and the environment that they live in. The inheritance relationship between castes defines the sub-group relationship between the agents so that special capabilities and behaviours can be introduced. The instance relationship between an agent and a caste declares that an agent is a member of a caste. This paper discuses how the caste facility can be employed to specify multiagent systems so that the notion of roles, organisational structures of agent societies, communication and, collaboration protocols etc. can be naturally represented.

1. Introduction

From software engineering point of view, one of the most appealing features of multiagent technology is its natural way to modularise a complex system in terms of multiple, interacting, autonomous components that have particular objectives to achieve [1]. Such modularity achievable by multiagent systems is much more powerful and natural than any kind of modularity that can be achieved by existing language facilities such as type, module and class. However, extant multiagent systems are mostly developed without a proper language facility that supports this. We believe that the lack of such a facility is one of the major factors hampering the wide-scale adoption of agent technology. In the formal specification language SLAB [2, 3], a facility called *caste* was introduced as a natural evolution of the notion of type in data type and the notion of class in object-oriented paradigm. In this paper, we further examine the uses of the caste facility in formal specification of multiagent systems.

The remainder of the paper is organised as follows. Section 2 is an introduction to the notion of caste. Section 3 outlines syntax and semantics of the caste facility in the formal specification language SLAB. Section 4 discusses the uses of the caste facility in the formal specification of MAS by some examples. Section 5 is the conclusion of the paper.

2. The Notion of Caste

This section first reviews our model of multiagent systems. Based on this model, we introduce the notion of castes and discuss its similarity and differences with the notion of classes in object-oriented paradigm.

2.1. A Model of Multiagent Systems

In our model, agents are defined as encapsulations of data, operations and behaviours that situate in their designated environments. Here, data represents an agent's state. Operations are the actions that an agent can take. Behaviour is a collection of sequences of state changes and operations performed by the agent in the context of its environment. By encapsulation, we mean that an agent's state can only be changed by the agent itself, and an agent has its own rules that govern its behaviour. Each agent must also have its designated environment. Constructively, agents are active computational entities with a structure containing the following elements.

- *Agent name*, which is the identity of the agent.
- *Environment*, which consists of a set of agents that interact with the agent.
- *Visible state*, which consists of a set of variables. The values of these variables are visible, but cannot be changed by other agents in the environment.
- *Visible actions*, which are the atomic actions that the agent can take. Each action has a name and may have parameters. When the agent take such an action, it becomes an event visible by the agents in the environment.
- *Internal state*: which consists of a set of variables and defines the structure of the internal state, such as the desires, beliefs and intentions of the agents in the BDI model [4, 5]. The values of visible and invisible states can be the first order or higher order, structural or scalar, symbolic or numeric, etc.
- *Internal actions*: which are the internal actions that the agent can take. Other agents in the environment cannot tell if the agent takes such an action. Actions can have parameters, which may also have first order or higher order, structural or scalar, and symbolic or numeric values.
- *Behaviour rules*: which determine the behaviour of the agent. They should cover the following aspects.
  - *The applicability condition*. The agent takes an action only when the environment and its own state are at the right condition, which is called the applicability condition of the action.
  - *The effects of the action*. An action may have effects on visible and/or internal parts of its own state. It is worthy noting that an agent cannot effect the state of any other agent or object.
  - *None-deterministic and stochastic behaviour*. An agent may have none-deterministic and/or stochastic behaviour. If a number of actions are applicable, choices between the actions may be none-deterministic and/or may have a certain probabilistic distribution.

Agents constructively defined above have a number of features. Firstly, they are autonomous. As Jennings [6] pointed out, autonomous means that agents 'have control

both over their internal state and over their own behaviour'. In [7, 8], agents' autonomous behaviour was characterised by two features that an agent can say 'go' (the ability to initiate action without external invocation) and say 'no' (the ability to refuse or modify an external request). Our definition requires that agents have their own rules of behaviour. Hence, they are autonomous.

Secondly, agents defined above are communicative and social. Communication plays a crucial role in multiagent systems. Agents must communicate with each other to collaborate, to negotiate and to compete with each other as well. By dividing an agent's states and actions into visible and invisible parts, we have given agents the capability of communication with each other. We human beings communicate with each other by taking actions. We speak, shout, sing, laugh, cry, and write to communicate. We even make gesture or other body languages to communicate. All these means of communication are 'visible' actions. We also utilise visible states to communicate. For example, the colour of traffic lights indicates whether you should cross the road. We show a smiling face to indicate we are happy and a sad face to indicate unhappy. These means of communication are based on visible states. However, taking a visible action or assigning values to visible state variables is only half of the communication process. The agent at the receiver side must observe the visible actions and/or read the values of the visible state in order to catch the signal sent out by the sender. The details of the protocols and meanings of such actions and state values are of premier importance in the development of multiagent systems. However, they are a matter of design decision and should left for software engineers to design and specify rather than pre-defined by the agent model or the language.

Thirdly, our model emphasizes that agents are situated in their designated environments. The power of agent-based systems can best be demonstrated in a dynamic environment [9, 10]. Characteristics of agents can also be defined in terms of their relationship with the environment. For example, agents are considered as 'clearly identifiable problem solving entities with well-defined boundaries and interfaces'. They are 'situated (embedded) in a particular environment - they receive inputs related to the state of their environment through sensors and they act on the environment through effectors' [1]. Our model requires a explicitly and clearly specification of the boundary and interface between an agent and its environment as well as the effects of environment on the agent's behaviour. Usually, such an environment consists of a set of objects and agents, which include equipment, devices, human beings, software and hardware systems, etc. As argued in [2, 3] and briefly summarised below, all these can be considered as agents as defined above. Therefore, the environment of an agent consists of a set of agents.

Fourthly, the definition implies that objects are special cases of agents in a degenerate form, while agents may be not objects. We consider objects as entities that have no control over their behaviours because an object has to execute a method whenever it receives a message that calls the method. The computational model of object-orientation defines the behaviour of all objects by the default rule of 'if receive a message, then execute the corresponding method'. Therefore, objects are agents with such a simple and uniform behaviour rule. With respect to the relationships with the environment, there are two key differences between objects and agents. First, agents are active in the sense they take initiative actions to effect the environment. In contrast, objects are passive, they are driven by the messages sent by the entities in

the environment. Second, an agent selectively observes a part of the environment that it is interested in, while an object is open to the environment in the sense that an object executes a method no matter who sends the message. These highlight the difference in the degrees of encapsulation achieved by objects and agents. Generally speaking, encapsulation means to draw a boundary between the entity and its environment and to control the accesses across the boundary. In object-oriented languages, the boundary enhances the access to object's state via method calls so that the integrity of an object's state can be ensured. However, such a boundary is weak because all entities in the environment can send a message to the object and hence call the method. The object cannot refuse to execute the method. In contrast, agents are able to selectively respond to the actions and state changes of certain entities in the environment. In our definition, each agent can explicitly specify its own subset of entities in the environment that can influence its behaviour.

Finally, our model is independent of any particular model or theory of agents. We believe that specific agent models can be naturally defined in our model. It is also independent of any particular agent communication language or protocol. A formal definition of the model can be found in [3].

2.2 The Notion of Caste

Existing language facilities provided by object-oriented languages cannot solve all the problems that software engineers face in developing agents [11]. New language facilities must be introduced to support agent-oriented software development. We believe that agent-orientation should be and can be a natural evolution of object-orientation so that the so-called agent-oriented paradigm can be build on the bases of object-oriented paradigm. In particular, the notion of caste is a natural evolution of the key notion of class in object-oriented paradigm. However, there are a number of significant differences between classes and castes. The following discusses such similarities and differences.

2.2.1 Structure

In object-oriented languages, a class is considered as the set of objects of common structure and function. Objects are instances of classes. Similarly, we define a set of agents of same structural and behavioural characteristics as a caste, where the term caste is used to distinguish from classes in object-oriented languages. Agents are therefore instances of castes. The agents in a caste share a common subset of state structures and a common subset of visible and internal actions, and some common behaviour characteristics. Similar to class, a caste is a structural template of agents. Therefore, a caste should have the same structural elements as agent.

For example, consider basketball players as agents. The environment of a basketball player in a game consists of the ball, the referee and other players in the game. The state of a player consists of a number of parameters, such as its position in the field, the speed and direction of movement, whether holding the ball, etc. These are the visible state of the agent. Of course, a player should also have invisible internal states, such as its plan and intention of actions, its energy level and skills, etc. The skills of a basketball player often represented by a number of statistics, such as

field goal percentage, three-point field goal percentage, etc. Another important state of a basketball player is the team that he/she plays for. In the real world, this state is made explicitly visible by requiring the players of a side to ware clothes of the same distinctive colour. A basketball player should also be able to take a number of basic actions, such as to move, to catch a ball, to pass the ball, to shoot, to dribble, to hold ball, etc. These are the visible actions a player is capable of. A good player should also follow certain basic strategies about when to pass ball and to whom, when to shoot, and how to take a good position in order to catch a rebound, how to steal, etc. These are some of its behaviour characteristics. Such structural and behavioural characteristics are common to all basketball player agents. In a system that simulate basketball games, we can define a caste with these characteristics and declare ten agents as instances of the caste. The following illustrates the caste structure by the example of basketball players.

```
Caste Players
  Environment:
    Ball, All:Players, Referee: Referees;
  Visible State
    Team: string;
    Position: Integer X Integer;
    Direction: Real;   Speed: Real;
    Holding_ball: boolean;
    ... ;
  Visible actions
    Move(direction: real, speed: real);
    Jump(direction, speed, upward: real);
    Pass(direction, speed, upward: real);
    Shoot(direction, speed, upward: real);
    ... ;
  Internal State
    FieldGoalPeercentage: real;
    ThreePointFieldGoalPercentage: real;
     ...;
  Internal action
    ...;
  Behaviour
    If HoldingBall
      & No player within the distance of 4 feet
      & Distance to the goal < 15 feet
    Then shoot(d, s, u).
    ...
End Players
```

Obviously, a caste differs from a class in their structures. A caste contains two essential parts that are not included in a class, the description of environment and the description of behaviours. The dynamic semantics of castes is also different from class. Firstly, for a caste, the visibility of a state variable does not imply that other objects or agents can modify the value of the variable. Secondly, an action in a caste is visible does not imply that it can be invoked by all objects and agents in the system. Instead, it is only an event that can be observed by other entities. Only the agent can decide whether and when to take an action. For the basketball player example, only the agent (i.e. the player) can decide when to shoot and how to shoot. It would not be a basketball game if a player shoot whenever someone (including players of the opposite team) asked it to shoot. Finally, communications between the agents are in the direction opposite to message passing between objects and follow the so-called Hollywood principle: 'You don't call me. I call you'. For the basketball example, a player looks for a teammate to pass the ball, rather than waits for a team-mate's request of the ball.

The example of basketball players shows that agents who play the same role and have the same structural and behavioural characteristics can be specified by a caste. If different roles require taking different actions or have different behaviour, separate castes should be defined for the roles. For example, referees of basketball games

should be specified by a caste different from the players.

2.2.2 Inheritance

In a way similar to classes, inheritance relationships can be defined between castes. A sub-caste inherits the structure and behaviour characteristics from its super-castes. It may also have some additional state variables and actions, observe more in the environment and obey some additional behaviour rules if any. Some of the parameters of the super-castes may be instantiated in a sub-caste. The inheritance relationship between castes is slightly different from the inheritance relationship between classes. A sub-caste not only inherits the state and action descriptions of its super-castes, but also the environment and behaviour descriptions. However, a sub-caste cannot redefine the state variables, actions, environment or behaviour rules that it inherits.

For example, the role of basketball players can be further divided into five positions: the inside post players, the left forward players, the right forward players, the left guard players and the right guard players. The behaviour of a basketball player is determined by his position. An inside post player will be responsible for rebounds and attacking from the inside. Therefore, an inside post player will take a position close to the goal. We, therefore, define inside post players as a sub-caste of basketball players with additional behaviour rules. Similarly, we can define the castes of left forwards, right forwards, left guards and right guards as sub-castes of the Players.

```
Caste InsidePosts is Sub-Caste of Players
  Behaviour  (* rules for inside post players *)
    If one of the teammates controls the ball
    Then take the position close to the goal;
    ...
End InsidePosts
Caste LeftForwards is Sub-Caste of Players
  Behaviour (* rules for left forward players *)
  ...
End LeftForwards
Caste RightForwards is Sub-Caste of Players
  Behaviour (*rules for right forward players*)
  ...
End RightForwards
Caste LeftGuards is Sub-Caste of Players
  Behaviour (* rules for left guard players *)
  ...
End LeftGuards
Caste RightGuards is Sub-Caste of Players
  Behaviour (* rules for right guard players *)
  ...
End RightGuards
```

The InsidePosts caste defined above is logically equivalent to the following caste. However, caste InsidePosts2 defined below is not a sub-caste of Players while InsidePosts is.

```
Caste InsidePosts2
  Environment:
    Ball, All:Players, Referee: Referees;
  Visible State:
    Team: string;
    Position: Integer X Integer;
    Direction: Real; Speed: Real;
    Holding_ball: boolean;
    ... ;
  Visible actions:
    Move(direction: real, speed: real);
    Jump(direction, speed, upward: real);
    Pass(direction, speed, upward: real);
    Shoot(direction, speed, upward: real);
    ... ;
  Internal State:
    FieldGoalPeercentage: real;
    ThreePointFieldGoalPercentage: real;
     ...;
  Internal action:
    ...;
  Behaviour
```

```
  If HoldingBall
    & No player within the distance of 4 feet
    & Distance to the goal < 15 feet
  Then shoot(d, s, u).
  ...
```

```
  (* rules for inside post players *)
  If one of the teammates controls the ball
  Then take the position close to the goal;
  ...
End InsidePosts
```

2.2.3 Instance

The relationship between agent and caste is also an instance relationship. When agent is declared as an instance of a caste, it automatically has the structural and behavioural features defined by the caste. The features of an individual agent can be obtained by initialisation of the parameters of the caste, such as the initial state of the agent. For example, Micheal Jordon was a basketball player for the team Bulls. The following declares such an agent as an instance of the caste Players. In addition to those structural and behavioural common features to all agents of a caste, an agent can also have additional properties of its own. For example, a basketball player may have his own style, which can be considered as additional behaviour characteristics. For instances, Jordon was good at shooting three point goals. Because of the uniqueness of his style, it is more natural to specify such a behaviour rule as a part of the agent's specification rather than to introduce a new caste. Therefore, we would have the following specification for the agent MJordan.

```
Agent MJordan: Players
  Visible State
    Team = 'Bulls';
  Internal State
    FieldGoalPeercentage= 50;
    ThreePointFieldGoalPercentage= 62;
  Behaviour
  If HoldingBall
    & No player within the distance of 6 feet
    & Distance to the goal < 30 feet
  Then shoot(d, s, u).
End MJordan
```

This agent declaration is logically equivalent to the following declaration. However, the agent Mjordan2 does not belong to the caste Players.

```
Agent Mjordan2
  Environment:
    Ball, All:Players, Referee: Referees;
  Visible State
    Team: string = 'Bulls';
    Position: Integer X Integer;
    Direction: Real;   Speed: Real;
    Holding_ball: boolean;
    ... ;
  Visible actions
    Move(direction: real, speed: real);
    Jump(direction, speed, upward: real);
    Pass(direction, speed, upward: real);
    Shoot(direction, speed, upward: real);
    ... ;
  Internal State
    FieldGoalPeercentage: real = 50;
    ThreePointFieldGoalPercentage: real =62;
    ...;
  Internal action
    ...;
  Behaviour
    If HoldingBall
      & No player within the distance of 4 feet
      & Distance to the goal < 15 feet
    Then shoot(d, s, u).
    If HoldingBall
      & No player within the distance of 6 feet
      & Distance to the goal < 30 feet
    Then shoot(d, s, u).
    ...
End Player
```

3. The SLAB Language

This section briefly reviews the SLAB language. We demonstrate how the caste facility is combined with other language facilities to enhance their expressiveness.

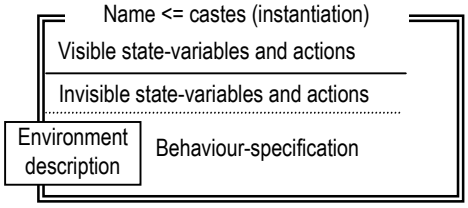
3.1 Agents and Castes

The specification of a multiagent system in SLAB consists of a set of specifications of agents and castes. There is a most general caste, called AGENT, that all castes in SLAB are sub-caste of AGENT.

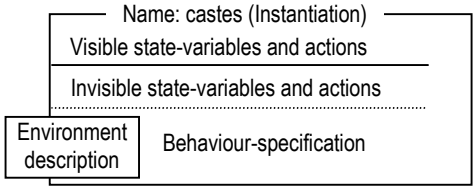
System ::= **{**Agent-description **|** caste-description**}**$^{+}$

The main body of a caste and agent specification in SLAB contains a structure description of its state and actions, a behaviour description and an environment description. The heads of caste and agent specifications give the name of caste or agent and their inherited castes. In a caste description, the clause '*Caste New_Caste* <= *Caste*$_1$*, ..., Caste*$_n$' specifies that *New_Caste* is a sub-caste of *Caste*$_1$*, ..., Caste*$_n$. Similarly, in an agent description, the clause '*Agent New_agent* <= *Caste*$_1$*, ..., Caste*$_n$' specifies that the *New-agent* is an instance of the castes *Caste*$_1$*, ..., Caste*$_n$. When no inherited caste is given, it is by default a sub-caste of the pre-defined caste AGENT. Every agent must be an instance of a caste. When caste name(s) are given in an agent specification, the agent is an instance of the castes; otherwise, the caste is by default AGENT. All the parameters in the specification of the caste must be instantiated in the specification of the agent. The following gives the syntax of castes and agents in EBNF. It can also be equivalently represented in graphic forms similar to the schema in Z [12].

caste-description **::=**
Caste name **[** <= **{** caste-name */* , **}** **;** **]**
[instantiation ; **]**
[environment-description ; **]**
[structure-description ; **]**
[behavior-description ; **]**
end name



agent-description **::=**
agent name **[** : **{** caste-name */* , **}** **]**
[instantiation **;** **]**
[environment-description ; **]**
[structure-description ; **]**
[behavior-description **]**
end name



The SLAB language requires an explicitly specification of the environment of an agent as a subset of the agents in the system that may influence its behaviour. The syntax for the description of environments is given below.

Environment-description **::=**
ENVIRONMENT **{** (agent-name **|** *All*: caste-name **|** variable : caste-name) */* , **}**$^{+}$

where an agent name indicates a specific agent in the system. 'All' means that all the agents of the caste have influence on its behaviour. As a template of agents, a caste

may have parameters. The variables specified in the form of "identifier: class-name" in the environment description are parameters. Such an identifier can be used as an agent name in the behaviour description of the caste. When instantiated, it indicates a specific agent in the caste. The instantiation clause gives the details about how the parameters are instantiated.

Instantiation **::= {** variable := agent-name */* , **}**$^+$

In SLAB, the state space of an agent is described by a set of variables with keyword VAR. The set of actions is described by a set of identifiers with keyword ACTION. An action can have a number of parameters. An asterisk before the identifier indicates invisible variables and actions.

structure-description **::= [** *Var* **{[** * **]** identifier: type */* ; **}**$^+$ **] [***Action* **{ [***]** action */* ; **}**$^+$**]**

action **::=** identifier **|** identifier (**{ [** parameter**:]** type */* , **}**$^+$)

In a caste and agent specification, the additional state variables and actions should not overlap with the state variables, action identifiers and parameter variables defined in the super-castes. Moreover, the castes that it inherits should have no common variables, no common action identifiers, and no common parameters. In other words, no re-definition of state variables and actions are allowed.

3.2 Behaviour Rules

In SLAB, the behaviour of an agent is specified by a set of rules.

Behaviour-rule **::= [** <rule-name> **]** pattern | **[** prob**]** –> event, **[**Scenario**] [***where* pre-cond**]** ;

In a rule, the pattern describes the agent's previous behaviour. The scenario describes the situation in the environment. The where clause is the pre-condition of the action to be taken by the agent. The event is the action to be taken when the scenario occurs and the pre-condition is satisfied. The agent may have a non-deterministic behaviour. The prob is an expression that defines the probability for the agent to take the specified action. When the prob is omitted, it means that the probability is greater than 0 and less than 1.

A scenario is a set of situations that might occur in the operation of a system. Here, in a multiagent system, we consider a scenario as a set of typical combinations of the behaviours of related agents in the system. SLAB's basic form of scenario description is pattern. Each pattern describes the behaviour of an agent in the environment by a sequence of observable state changes and observable actions. A pattern is written in the form of $[p_1, p_2, ..., p_n]$, where $n \geq 0$, and p_i are events. Patterns can be combined together by logic connectives and quantifiers to describe global situations of the whole system. The syntax of patterns and scenarios is given below. Their meanings are given in Table 1.

pattern **::=** [**{** event **[** || constraint **]** */* , **}**]

event ::= **[** time-stamp: **]** **[** action **]** **[** ! state-assertion **]**

action **::=** atomic-pattern **[** ^ arithmetic-expression **]**

atomic-pattern **::=** $ **| ~ |** action-variable **|** action-identifier **[** (**{** arithmetic-expression **}**) **]**

time-stamp ::= arithmetic-expression

Scenario **::=** Agent-Name : pattern **|** arithmetic-relation

| ∃ [arithmetic-exp] Agent-Var ∈ Caste-Name: Pattern **|** ∀ Agent-Var ∈ Caste-Name: Pattern

| Scenario & Scenario **|** Scenario ∨ Scenario **|** ~ Scenario

where a constraint is a first order predicate. An arithmetic relation can contain an expression in the form of μAgent-var$\in$ Caste.Pattern, whose value is the number of agents in the caste that whose behaviour matches the pattern.

Table 1. Semantics of scenario descriptions

Pattern/Scenario	Meaning
$	The *wild card*, it matches with all actions
~	The *silence* event
Action variable	It matches an action
P^k	A sequence of *k* events that match pattern *P*
Action (a_1, ...a_k)	An *action* that takes place with parameters match (a_1, a_2, ...a_k)
! *Predicate*	The state of the agent satisfies the *predicate*
[p_1,..., p_n]	The previous sequence of events match the patterns p_1, ..., p_n
A: P	The situation when agent *A*'s behaviour matches pattern *P*
$\forall X \in C : P$	The situation when the behaviours of all agents in caste *C* match pattern *P*
$\exists_{[m]} X \in C : P$	The situation when there are at least m agents in caste *C* whose behaviour matches pattern *P* where the default value of the optional expression *m* is 1
$\mu X \in C: P$	The number of agents in caste *C* whose behaviour matches pattern *P*
S_1 & S_2	The situation when both scenario S_1 and scenario S_2 are true
$S_1 \vee S_2$	The situation when either scenario S_1 or scenario S_2 or both are true
$\neg S$	The situation when scenario *S* is not true

The following are some examples of scenarios.

(1) $\exists$ p$\in$ Players: [shoot(x, y, z)].
It describes the situation that there is a player who is shooting.

(2) μ p$\in$ Players: [!position(x, y) || Is-Inside(x,y)] = 3
It describes the situation that there are 3 players inside the goal area.

(3) MJ: [!position(x,y)] & $\forall$p$\in$ Players: [!position(x', y') || Distance(<x,y>, <x', y'>) > 3 & p$\neq$MJ]
It is the situation when all players are at a distance more than 3 feet from MJ.
Obviously, without the caste facility, it is not possible to describe such scenarios.

4. Uses of Castes in Formal Specification

In [2, 3], we have shown how to use SLAB to specify personal assistants such as Mae's Maxims [13], reactive agents like ants, and speech act. This section further illustrates the uses of the caste facility in the specification of communication protocol and organisations of agent societies.

4.1 Organisation of Agent Society

Multiagent systems often divide agents into a number of groups and assign each group a specific role. Such a structure of multiagent system can be naturally specified by using the castes and the inheritance and instance relationships.

For example, in section 2, we have seen how castes are used to specify the roles and the organisational structure of a basketball game simulation system. Fig. 1 below shows the inheritance and instance structure of the example.

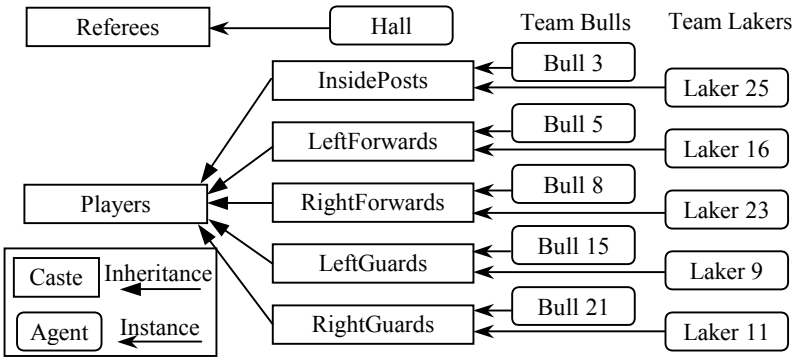


Fig. 1. Castes / agents structure of the basketball example

4.2 A Simple Communication Protocol

A typical example of common behaviour rules that all agents in a multiagent system follow is a communication protocol that defines how agents communicate with each other. Such rules can be specified in a caste and all other castes are then specified as its sub-caste.

For example, the following castes specify a synchronised communication process between agents. If an agent want to send a message to another agent, it signals to the receiver, waits for the receiver to signal back, and then passes the message to the receiver. When an agent saw another agent's signal, it signals back and then receives the message. Here, we have two roles: the senders and the receivers. Each role is specified by one caste.

A sender in the Senders caste has a visible state variable Signal, which indicates whether the sender want to send a message. The process of sending a message is defined by 3 rules. By the <Start sending> rule, the scenario to apply the rule requires that the receiver agent must be in the state of !SignalBack=off, where !pred means that the state of the agent must satisfy the predicate. An agent starts sending a message if its Intention is 'send' and its State is idle, i.e. the assertion !(Intention = 'send' & State=Idle) is true. The result of taking this action is that the state of the agent satisfies the predicate !(Signal = ON & State = Sending). In other words, it will set variable Signal to be ON and State to be Sending. Once this has been done, the agent can take a second action as specified by the <Send message> rule if the receiver's SignalBack turns into

ON. Similarly, the <Finish sending> rule defines the state change for the sender after sending a message.

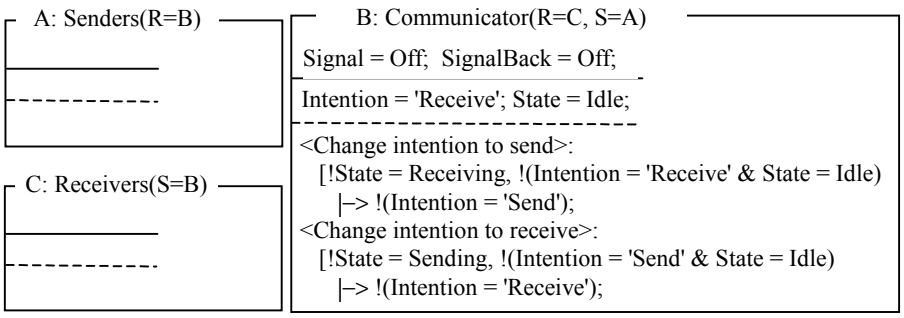
```
Senders
Var      Signal: {ON, OFF};
Action   Send(AgentName, Message);
-------------------------------------
Var    State: {Sending, Receiving, Idle} = Idle;
       Intention: String;
       Message: String;
-------------------------------------
R: Receivers
<Start sending>:
  [ !(Intention = 'send' & State = Idle)] |–> ! (Signal = ON & State = Sending),
                          if R: [!SignalBack = OFF];
<Send message>:
  [ !(Signal = ON & State = Sending)] |–> Send(R, Message),
                          if R:[SignalBack = ON];
<Finish sending>:
  [ Send(R, Message) ! (Signal = ON & State = Sending)]
      |–> Signal = OFF & State = Idle;
```

```
Recievers
Var      SignalBack: {ON, OFF};
Action   Receive(AgentName,Message);
-------------------------------------
Var    State: {Sending, Receiving, Idle} = Idle;
       Intention: String;
       Message: String;
-------------------------------------
S: Senders
<Start Receiving>:
  [!(Intention = 'Receive' & State = Idle)]
    |–> ! (SignalBack = ON & State = Receiving), if S: [Signal = ON];
<Receive message>:
  [!(State=Receiving & SignalBack = ON)]
    |–> Receive(S, x)!(Message=X), if S:[Send(Myself, X) ! Signal = ON];
<Finish receiving>:
  [ Receive(S, X) ! (State = Receiving & SignalBack = ON)]
    |–>  !(Signal = OFF & State = Idle);
```

The Receivers caste also has three rules, <Start receiving>, <Receiving message> and <Finish receiving>. They define the process of state change for the receiver agent. An agent can be a sender and receiver at the same time. Hence, we define a caste Communicators that inherits both castes of Senders and Receivers as follows.

```
Communicators <= Senders, Receivers
-------------------------
-----------------------
```

Agents that follow the same communication protocol can be declared as instances of the castes or their sub-castes. For example, the following specifies a system that consists of 3 agents, A, B and C. Here, agent A sends messages to agent B, and B passes the message to agent C. Notice that, the castes Senders, Receivers and Communicators do not specify when an agent will have the intention to send or receive a message. Therefore, additional behaviour rules are added to the specification of agent B so that it repeats the cycle of receiving a message from A, then passing it to agent C.



This example shows that the caste and inheritance facilities provide a powerful vehicle to describe the normality of a society of agents. Multiple inheritances enable an agent to belong to more than one society and play more than one role in the system at the same time.

5. Conclusion

The SLAB language integrates a number of novel language facilities that intended to support the development of agent-based systems. Among these facilities, the notion of caste plays a crucial role. A caste represents a set of agents that have same capability of performing certain tasks and have same behaviour characteristics. Such common capability and behaviour can be the ability of speaking the same language, using the same ontology, following the same communication and collaboration protocols, and so on. Therefore, caste is a notion that generalises the notion of types in data type and the notion of classes in object-oriented paradigm. This notion is orthogonal to a number of notions proposed in agent-oriented methodology research, such as the notions of role, team, organisations, but it can be naturally used to implement these notions. A caste can be the set of agents playing the same role in the system. However, agents of the same caste can also play different roles especially when agents form teams dynamically and determines its role at run time. Using the caste facility, a number of other facilities can be defined. For example, the environment of an agent can be described as the agents of certain castes. A global scenario of a multiagent system can be described as the patterns of the behaviours of the agents of a certain caste. The example systems and features of agent-based systems specified in SLAB show that these facilities are powerful and generally applicable for agents in various models and theories.

Our model of agents is closely related to the work by Lesperance, *et al* [14], which also focused on the actions of agents. However, there are two significant differences. First, they consider objects and agents are different types of entities. Consequently, they allow an agent to change the state of objects in the environment, while we only allow an agent to modify its own state. Second, the most important difference is, of course, there is no notion of caste or any similar facility in their system. The notion of agent groups has been used in a number of researches on the multi-modal logic of rationale agents, such as in Wooldridge's work [5], etc. However, such notion of groups of agents is significantly different from the notion of caste, because there is neither inheritance relationships between the groups, nor instance relationship between an agent and a group. The only relationship is the membership relationship. Any subset of agents can form a group regardless of their structure and behaviour characteristics. Many agent development systems are based on object-oriented programming. Hence, there is a natural form of castes as classes in OO paradigm, which is often called agent class. However, as argued in section 2, although agents can be regarded as evolved from objects and castes as evolved from classes, there are significant differences between agents and objects and between castes and classes. Therefore, the notion of caste deserves a new name.

The use of scenarios and use cases in requirements analysis and specification has been an important part of object-oriented analysis; see e.g. [15]. However, because an object must respond in a uniform way to all messages that call a method, there is a huge gap between scenarios and requirements models. As an extension to object-oriented methodology, a number of researchers have advanced proposals that employ scenarios in agent-oriented analysis and design [16, 17, 18]. In the design of SLAB, we recognised that scenarios can be more directly used to describe agent behaviour. The gap between scenarios and requirements models no longer exists in agent-based systems because the agent can controls its behaviour. Its responses can be different from scenario to scenario rather than have to be uniform to all messages that call a method. When the notion of scenario is combined with the caste facility, we obtained a much more powerful facility for the description of scenarios than any existing one.

There are a number of problems related to the caste facility that need further investigation. For example, in SLAB an agent's membership of a caste is statically determined by agent description. Static membership has a number of advantages, especially its simplicity and easy to prove the properties of agents. A question is whether we need a dynamic membership facility in order to specify and implement dynamic team formation. An alternative approach to the problem of team formation is to define aggregate structures of agents and castes. Another design decision about the caste facility that we faced in the design of SLAB was whether we should allow re-definitions of behaviour rules in the specification of sub-castes.

Although the caste facility was first introduced as a specification facility, we believe that it can be easily adopted in an agent-oriented programming language for the implementation of multiagent systems. How to implement the facility is an important issue in the design and implementation of agent-oriented programming languages. It also deserves further research.

References

1. Jennings, N. R., On agent-based software engineering, *Artificial Intelligence*, Vol. 117, 2000, pp277~296.
2. Zhu, H. Formal Specification of Agent Behaviour through Environment Scenarios, *Proc. of NASA First Workshop on Formal Aspects of Agent-Based Systems*, LNCS, Springer. (In press) Also available as Technical Report, School of Computing and Mathematical Sciences, Oxford Brookes University, 2000.
3. Zhu, H., SLAB: A Formal Specification Language for Agent-Based Systems, Technical Report, School of Computing and Mathematical Sciences, Oxford Brookes University, Feb. 2001.
4. Rao, A. S., Georgreff, M. P., Modeling Rational Agents within A BDI-Architecture, in *Proc. of the International Conference on Principles of Knowledge Representation and Reasoning*, 1991, pp473~484.
5. Wooldrighe, M., *Reasoning About Rational Agents*, The MIT Press, 2000.
6. Jennings, N. R., Agent-Oriented Software Engineering, in *Multi-Agent System Engineering, Proceedings of 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Valencia, Spain, June/July 1999, Garijo, F. J., Boman, M. (eds.), LNAI Vol. 1647, Springer, Berlin, Heidelberg, New York, 1999, pp1~7.
7. Bauer, B., Muller, J. P., and Odell, J., Agent UML: a formalism for specifying multiagent software systems, in *Agent-Oriented Software Engineering*, Ciancarini, P. and Wooldridge, M. (Eds.), LNCS, Vol. 1957, Springer, 2001, pp91~103.
8. Odell, J., Van Dyke Parunak, H., and Bauer, B., Representing Agent interaction protocols in UML, in *Agent-Oriented Software Engineering*, Ciancarini, P. and Wooldridge, M. (Eds.), LNCS, Vol. 1957, Springer, 2001, pp121~140.
9. Jennings, N. R., Wooldridge, M. J. (eds.), *Agent Technology: Foundations, Applications, And Markets.* Springer, Berlin, Heidelberg, New York, 1998.
10. Huhns, M., Singh, M. P. (eds.), *Readings in Agents*, Morgan Kaufmann, San Francisco, 1997.
11. Lange, D. B. and Oshima, M., Mobile agents with Java: the Aglet API, *World Wide Web Journal*, 1998.
12. Spivey, J. M., *The Z Notation: A Reference Manual*, (2nd edition), Prentice Hall, 1992.
13. Maes, P., Agents That Reduce Work And Information Overload, *Communications of the ACM*, Vol. 37, No.7, 1994, pp31~40.
14. Lesperance, Y., levesque, H. J., Lin, F., Marcu, D., Reiter, R. and Scherl, R., Foundations of logical approach to agent programming, in *Intelligent Agents II*, Eds. Wooldridge, M., Muller, J., and Tambe, M., LNAI, Vol. 1037, Springer-Verlag, 1996, pp331~346.
15. Jacobson, I., et al., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
16. Iglesias, C. A., Garijo, M. Gonzalez, J. C., A Survey of Agent-Oriented Methodologies, in *Intelligent Agents V,* Muller, J. P., Singh, M. P., Rao, A., (eds.), LNAI Vol. 1555. Springer, 1999, pp317~330.
17. Iglesias, C. A., Garijo, M., Gonzalez, J. C., Velasco, J. R., Analysis And Design of Multiagent Systems Using MAS-Common KADS, in *Intelligent Agents IV,* Singh, M. P., Rao, A., Wooldridge, M. J. (eds.), LNAI Vol. 1356, Springer, 1998, pp313~327.
18. Moulin, B. Brassard, M., A Scenario-Based Design Method And Environment for Developing Multi-Agent Systems, in *Proc. of First Australian Workshop on DAI*, Lukose, D., Zhang, C. (eds.), LNAI Vol. 1087, Springer Verlag, 1996, pp216~231.