A Formal Specification Language for Agent-Oriented Software Engineering

Hong Zhu
Department of Computing, School of Technology, Oxford Brookes University
Wheatley Campus, Oxford OX33 1HX, UK. Tel.: ++44 1865 484580, Email: hzhu@brookes.ac.uk

ABSTRACT

One of the most appealing features of agent technology is its natural way to modularise complex systems in terms of multiple interacting autonomous components. This feature is supported by the language facility castes in the formal specification language SLABS, which is designed for modular and composable specification of multi-agent systems. The paper reports the syntax and semantics of the language, and illustrates its style of formal specification by a distributed synchronisation algorithm.

Keywords

Formal specification language, Multiagent Systems, Scenario, Software Agents, Castes

1. INTRODUCTION

In recent years, agent technology has been applied to more and more critical application areas such as telecommunications [1], space adventure [2], power grid control, military [3], etc. to provide viable solutions to the problems that was unable to be solved satisfactorily by other existing techniques. It is also clearly demonstrated that agent technology is particularly suitable to solve problems related to web-based applications such as e-commerce and web search engines [4]. However, developing agent-based systems is extremely difficult because the dynamic behaviours of agent-based systems are difficult to specify, analyse, verify and validate. Being autonomous, proactive and adaptive, an agent-based system may demonstrate emergent behaviours that are hard to predict, difficult to design, and expensive to test. An early incident of software failure attributed to autonomous agents in particular is the crash of Air France's Airbus 320 at an air show in June 1988 [5]. Airbus 320 was the first fly-by-wire passenger aircraft in the world. In other words, an autonomous agent controls the aircraft. The incident was caused by a conflict between the human pilot's instruction and the autonomous control by the software. While the pilot intended to fly over the airport in the air show, the fly-by-wire control software seems to have instructed the aircraft to land, which was believed to be the cause of the accident [6].

The new features of agent-based systems demand new methods for the specification of agent behaviours and for the verification and validation of their properties to enable software engineers to develop reliable and trustworthy agent-based systems. It has been recognised that the lack of rigour is one of the major factors hampering the wide-scale adoption of agent technology [7]. On the other hand, the modularity inherent in multi-agent systems can offer a new approach to decomposing complicated formal specifications into composable modular components.

The past few years have seen increasing research interests in agent-oriented software development methodology. Existing work falls into three main classes. The first is towards the theoretical foundations for understanding agent-based systems. Much work has been focused on modelling and reasoning agents' rational behaviour by introducing modalities for belief, desire and intention, e.g. [8, 9, 10, 11]. Game theory has also found its position in the formalisation of agent models, e.g. [12]. A great number of formal models of agents have been proposed and investigated in the literature; see e.g. [13, 14]. Most of them are based on an internal mental state model of agents, yet some are based on a model of the external social behaviours of collaborative agents, e.g. [15]. Although these formal models of agents significantly improved our understanding of agent-based systems, they do not immediately facilitate the development of agent-based systems. As pointed out in [16], a specification method based on a specific model of agents may result in the existence of certain agent theory and systems that do not match the concept in the specification formalism. Moreover, temporal logics, particularly when combined with modalities for belief, desire, etc., can be very complex. The second group of researches is on the development process and development methods for engineering agent-based systems, see e.g. [17, 18, 19, 20, 21]. These works mostly focused on diagrammatic notations that support the analysis and design of multi-agent systems. Some of the notations extend object-oriented methods and notations such as UML. Some introduce new models and new diagrammatic notations. How such diagrammatic notations are related to the logic and formal models of agents remains an open problem. The third group consists of the researches on the language facilities and features that support the formal specification and verification of agent-based systems in a software engineering context, although there is little such work reported in the literature [22]. The use of existing formal specification languages, such as Z, has also been explored to specify agent architecture [23] and concepts [24, 25]. Despite the large number of publications on agents in the literature, we are lack of researches on language facilities that support the development of large-scale complicated multi-agent systems. In particular, we are lack of language facilities to explicitly specify the environment of agents and agent-based systems although it is widely recognised that an important characteristic of agents is that they are entities situated (embedded) in a particular environment [26]. We are lack of facilities that can clearly state how agents' behaviours are related to the environment. We are lack of language facilities that enable us to maximise the power of the way that multi-agent systems modularise complex systems into cooperative autonomous components. Addressing these problems, in the past 3 years we have been searching for language facilities to support the analysis, specification, design and implementation of agent-based systems in the context of software engineering. Some of the results have been incorporated in the design of a formal specification language called SLABS [27, 28].

In this paper, we report the main features of the language SLABS and demonstrate its uses in the development of multi-agent systems. The remainder of the paper is organised as follows. Section 2 defines the syntax and semantics of the language. Section 3 illustrates SLABS' style of specification by an example. Section 4 concludes that paper with discussions on related work

and future work.

2. THE LANGUAGE SLABS

The meta-language used to define the syntax is EBNF; see Table 1. In a syntax definition, meta-symbols are in bold font such as **::=**. Terminals are in *italic* font such as *Var*. Non-terminals are in normal font such as Agent-Description.

Table 1. The meta-symbols in EBNF

Name	Symbol	Means
Definition	**::=**	A **::=** B means that A is defined as B.
Concatenation		AB means that A is followed by B.
Optional	**[ ]**	**[** A **]** means that A is optional.
Choice	**|**	A **|** B means either A or B.
Repetition	**{ }**	**{** A **}** means that A may appear any times including zero times or more times.
Repetition with separator	**{ / }**	**{** A **/** B **}** means a sequence of A separated by B, where the number of A's can be zero or more.
Positive repetition	**{ }+**	**{** A **}** means that A may appear at least once.
Parenthesis	**( )**	They are used to change preference.

2.1 Agents and Castes

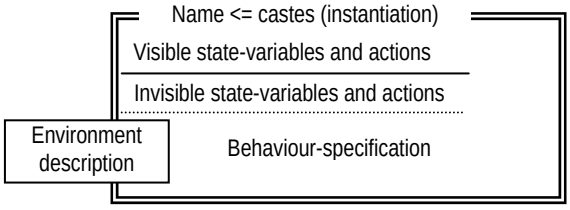
The specification of a multi-agent system in SLABS consists of a set of specifications of agents and castes.

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

Caste is one of the novel concepts in SLABS, which is a natural evolution of the concepts of classes in object-orientation. Castes can play a significant role in the requirements analysis and specification as well as design and implementation of multi-agent systems [29]. There is a most general caste, called AGENT, such that all castes are its sub-castes. The main body of a caste description contains a description of the structure of its states and actions, a description of its behaviour, and a description of its environment. The following gives the syntax of castes description in EBNF.

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

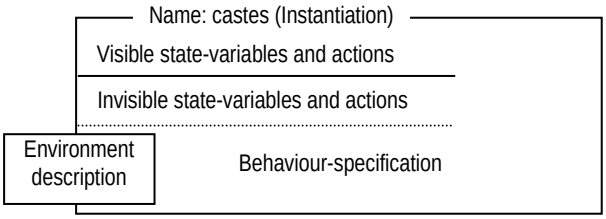
It can also be equivalently represented as follows in a graphic form similar to schemas in Z [30].



The clause 'Caste C <= C_1, C_2, ..., C_n' specifies that caste C inherits the structure, behaviour and environment descriptions of existing castes C_1, C_2, ..., C_n. When no inherited caste is given in a caste specification, it is by default a sub-caste of the predefined caste AGENT. Thus, a binary inheritance relation $\prec$ is defined on the castes C_1 and C_2, if C_1 is specified as a sub-caste of C_2. The inheritance relation is required to be a partial ordering on castes. Notice that, the inheritance relation is static in the sense that it does not change at run-time.

The relationship between agents and castes is similar to what is between objects and classes. What is different is that an agent can join into a caste or quit from a caste at run-time. If an agent is an instance of a caste, it has all the structural, behaviour and environment descriptions given in the caste's specification. The following gives the syntax of agent descriptions in SLABS and its equivalent graphic form.

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



When caste names are given in an agent description, the agent is an instance of the castes when it is created. If no caste name is given in an agent specification, the caste of the agent is by default AGENT. All the parameters in the specification of the caste must be instantiated. Moreover, it may have additional structural, behaviour and environment descriptions to extend its state space, to enhance its ability to take actions and to widen its view of the environment.

We define that a *multi-agent system* consists of a finite set of *agents* $\{A_1, A_2, ..., A_n\}$. These agents belong to a partially ordered set of castes $C_1, C_2, ..., C_m$. Let $A \in_t C$ denote that agent A belongs to caste C at time t. We require that for all agents A and castes C and C', for all times t, $A \in_t C \wedge C \prec C' \Rightarrow A \in_t C'$.

2.2 Environment

The SLABS language enables software engineers to explicitly specify the environment of an agent as a subset of the agents in the system that may affect its behaviour. This is another fundamental difference between agents (castes) and objects (classes). 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. It indicates an agent in the caste when instantiated. The instantiation clause gives the details about how the parameters are instantiated.

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

The environment of an agent is a subset of the agents in the system $Env_{A,t} \subseteq \{A_1, A_2, ..., A_n\}$, which may change at run-time. Let *EC* be an environment description expression. Then, the meaning

$[\![EC]\!]_t$ of the environment expression *EC* at time *t* can be defined as follows.

$$[\![agent]\!]_t = \{agent\}; \quad (1)$$

$$[\![All : Caste]\!]_t = \{X \mid X \in_t Caste\}; \quad (2)$$

$$[\![x : Caste]\!]_t = \{A_k\}, \quad (3)$$

where "$x := A_k$" is the instantiation of variable *x* when the agent joining the caste.

2.3 State and Action Spaces

In SLABS, 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-declaration **/** ; **}+]**
action-declaration **::=** identifier **|** identifier (**{ [** parameter**:]** type **/** , **}+**)

2.4 Inheritance and Dynamic Caste

As informally stated above, a caste can inherit from multiple super-castes. We now formally define the semantics of inheritance and agent's dynamic joining a caste or retreat from a caste. Let *C* be a caste specified as follows.

Caste C <= $C_1(x_{11}:=exp_{11}, \ldots, x_{1n_1} := exp_{1n_1})$,
$\ldots, C_k(x_{k1}:=exp_{k1}, \ldots, x_{kn_1} := exp_{kn_1})$
ENVIRONMENT $EC_1, \ldots, EC_w$;
VAR $*v_1:T_1, \ldots, *v_m:T_m; u_1:S_1, \ldots, u_l:S_l;$
ACTION $*A_1(p_1), \ldots, *A_s(p_s); B_1(q_1), \ldots, B_t(q_t);$
RULES $R_1, R_2, \ldots, R_h$
End *C*;

Let $S^V(C)$ and $S^I(C)$ denote the visible and invisible state spaces of a caste *C*, respectively. We define $S^V(C)$ and $S^I(C)$ as follows.

$$S^V(C) = \{C.v_1:T_1, \ldots, C.v_m:T_m\} \cup \bigcup_{x=1}^{k} S^V(C_x). \quad (4)$$

$$S^I(C) = \{C.u_1:S_1, \ldots, C.u_j:S_j\} \cup \bigcup_{x=1}^{k} S^I(C_x). \quad (5)$$

Let $\Sigma^V(C)$ and $\Sigma^I(C)$ denote the visible and invisible action spaces of caste *C*, respectively. Similar to state spaces, we define $\Sigma^V(C)$ and $\Sigma^I(C)$ as follows.

$$\Sigma^V(C) = \{C.A_1(p_1), \ldots, C.A_s(p_s)\} \cup \bigcup_{x=1}^{k} \Sigma^V(C_x). \quad (6)$$

$$\Sigma^I(C) = \{C.B_1(q_1), \ldots, C.B_t(q_t)\} \cup \bigcup_{x=1}^{k} \Sigma^I(C_x). \quad (7)$$

Let *RULE*(C) denote the set of behaviour rules of caste *C*. We have that

$$RULE(C) = \{R_1, \ldots, R_h\} \cup \bigcup_{x=1}^{k} RULE(C_x). \quad (8)$$

Let *ENV*(*C*) denote the environment for caste *C*, we have that

$$ENV(C) = \{EC_1, \ldots, EC_w\} \cup \bigcup_{x=1}^{k} ENV(C_x). \quad (9)$$

Let agent *A* be a member of castes $C_1, C_2, \ldots, C_n$ at run-time *t*. Let $S_{A,t}$ denote the state space of agent *A* at time *t*. Each state consists of two disjoint parts, the externally visible part and the internal part. The external part is visible for all agents in the system, while the internal part is not visible for any other agents in the system. Therefore, $S_{A,t} = S^V_{A,t} \times S^I_{A,t}$, where $S^V_{A,t}$ and $S^I_{A,t}$ are the externally visible part and internal part of the state space, and defined as follows, respectively.

$$S^V_{A,t} = \bigcup_{i=1}^{n} S^V(C_i), \text{ and } S^I_{A,t} = \bigcup_{i=1}^{n} S^I(C_i). \quad (10)$$

Let $\Sigma_{A,t}$ denote the set of actions that an agent can take at time *t*. An action can also be either externally visible or internal (hence externally invisible). Assume that an agent cannot take two actions at the same time, thus $\Sigma_{A,t} = \Sigma^V_{A,t} \cup \Sigma^I_{A,t}$, where $\Sigma^V_{A,t} \cap \Sigma^I_{A,t} = \varnothing$, $\Sigma^V_{A,t}$ and $\Sigma^I_{A,t}$ are the sets of externally visible actions and internal actions, respectively. They are defined as follows.

$$\Sigma^V_{A,t} = \bigcup_{i=1}^{n} \Sigma^V(C_i), \text{ and } \Sigma^I_{A,t} = \bigcup_{i=1}^{n} \Sigma^I(C_i). \quad (11)$$

The environment of agent *A* at run-time *t*, denoted by $Env_{A,t}$ is defined as follows.

$$Env_{A,t} = \bigcup_{i=1}^{n} [\![EC \mid EC \in ENV(C_i)]\!]_t. \quad (12)$$

The set of rules that agent *A* must satisfy at time *t* is the set $RULE_{A,t}$, which is defined as follows.

$$RULE_{A,t} = \bigcup_{i=1}^{n} RULE(C_i). \quad (13)$$

Notice that, the value of the set $ENV_{A,t}$ depend on time *t*. for two reasons. First, the set of agents in a caste *C* may change when the agent joins into or retreats from a caste. Second, the agent A may join a caste or retreat from a caste so that the set $\{C_1, \ldots, C_n\}$ changes from time to time. For the sake of simplicity, when there is no risk of confusion, we will omit the subscript *t* in the sequel.

For example, consider the following specification.

```
Caste Person;
    ENV All: Persons;
    VAR *Surname: String, *Name: String, *Birthday: DATE,
            Nationality: String;
    ACTION *Speak(String);
    RULES     <r1>: [] |-> Speak("Hello World"),
            <r2>: [] |-> Speak("My name is " , Name, Surname)
End Person;
Caste OBU-Students <= Persons;
    ENV    Tutor: OBU-Staff;
    VAR    *Number: Integer;
            Field: {Computing, Business}
    ACTION *ChangeField(Field);
    RULES     <s1>: [] |-> ChangeField(NewField),
                        where NewField ≠Field;
            <s2>: [ChangeField(NewField)] |-> !Field'=NewField,
                if Tutor: [ApproveChangeField(Slef, NewField);
End OBU-Students;
Caste OBU-Staffs <= Persons;
    ENV Tutee1, Tutee2: OBU-Students;
    VAR *Number: Integer;
            *Dept: {Computing, Math, Business}
```

ACTION *ApproveChangeField(OBU-Students, Field);
RULES <st1> : [] |-> ApproveChangeField(St, NewField),
If ∃St∈OBU-Student: [ChangeField(NewField)],
Where St=Tutee1 or St=Tutee2
End OBU-Staffs;

The visible state variables of caste OBU-Students contains the following elements {Person.Surname: String, Person.Name: String, Person.Nationality: String, OBU-Students.Number: Integer}. The visible actions of the caste OBU-Staffs contains the following elements: {Person.Speak(String), OBU-Staffs. ApproveChangeField(OBU-Students, Field)}.

An agent John initially crated as a member of Persons can join the caste OBU-Staffs a tutor in the department of business and obtain two additional state variables OBU-Staffs.Number and Dept, and one additional action OBU-Staff.ApproveChangeFields. It is also assigned with two tutees, say Harry and Nigel, as a part of it environment. A member of the OBU-Staffs caste, for example John, can join the OBU-Students. By doing so, the agent obtains two new state variables OBU-Students. Number and OBU-Students.Field, one action OBU-Students. ChangeField, one environment agent Tutor, which must be instantiated when agent John join the caste, say Chris, as its tutor, and also two new behaviour rules <s1> and <s2>. When John quits from the OBU-Students caste, his state space reduces by deleting the variables OBU-Students.Number and OBU-Students.Field. This does not affect his other parts of state space and action space. Similarly, John can also quit from the OBU-Staff caste.

2.5 Behaviour

Agents behave in real-time concurrently and autonomously. To capture the real-time features, an agent's behaviour is modelled by a set of sequences of events indexed by the time when the events happen.

2.5.1 Runs and Time

A *run r* of a multi-agent system is a mapping from time *t* to the set $\prod_{i=1}^{n} S_{A_i,t} \times \Sigma_{A_i,t}$. The behaviour of a multi-agent system is defined to be a set *R* of possible runs. Instead of defining a fixed set of time moments, the set of time moments are characterised by a collection of properties.

Definition 1.

Let *T* be a non-empty subset of real numbers. *T* is said to be a *time index set*, or simply the *time*, if

1) Bounded in the past, i.e. $\exists t_0 \in T.\forall t \in T.(t_0 \le t)$; (14)

2) Unbounded in the future, i.e. $\forall r \in \mathrm{R}.\exists t \in T.(t > r)$; (15)

3) Uniformity, i.e. $\forall\ t_1, t_2, t_3 \in T.\ (t_2 > t_1 \Rightarrow t_3 + t_2 - t_1 \in T)$. (16)

□

The following lemma states that a time index set *T* can be characterised by two real numbers: the *start time* t_0 and the *time resolution* ρ, where $\rho \ge 0$. Readers are referred to [28] for the proof of the lemma.

Lemma 1.

For all subsets *T* of real numbers that satisfy properties (14), (15) and (16), we have that either $T=\{t_n \mid t_n = t_0 + n\rho, n=0, 1, 2, \ldots\}$ for some positive real number ρ, or $T=\{r \mid r \in R \text{ and } r \ge t_0\}$. In the former case, we say that the time index set *T* is *discrete*, and in the latter case, we say that *T* is continuous and ρ is called the *resolution* of the time index set *T*. □

For any given run *r* of the system, we say that a mapping *h* from *T* to $S_{A,t} \times \Sigma_{A,t}$ is the run of agent *A* in the context of *r*, if $\forall t \in T.h(t) = r_A(t)$, where $r_A(t)$ is the restriction of $r(t)$ on $S_{A,t} \times \Sigma_{A,t}$. Let r_A denote the run of agent *A* in the context of *r*, and $R_A = \{r_A \mid r \in R\}$ denote the behaviour of agent *A* in the system.

2.5.2 Assumptions

We assume that a multi-agent system has the following properties.

Instantaneous actions. We assume that actions are instantaneous, i.e. they take no time to complete.

Silent moments. We assume that an agent may take no action at a time moment *t*. In such a case, we say that the agent is silent at time *t*. For the sake of convenience, we treat silence as a special action and use the symbol τ to denote silence. Therefore, we assume that for all agents *A*, $\tau \in \Sigma_A^V$.

Separability. We assume that the actions taken by an agent are separable, i.e. for all runs *r*, and all agents *A*, there exists a real number $\varepsilon_{r,A} > 0$ such that $r_A^C(t) \ne \tau$ implies that for all $x \in T$, $t < x \le t + \varepsilon \Rightarrow r_A^C(x) = \tau$, where $r_A^C(t)$ denotes the action taken by agent *A* at time moment *t* in the run *r*. Consequently, an agent can take at most a countable number of non-silent actions in its lifetime.

Initial time and sleeping state. An agent can join the system at a time, say $t_{init,A}$, later than the system's start time. We say that the agent A is *sleeping* before time moment $t_{init,A}$. We use a special symbol $\perp \notin S_A$ to indicate such a state of an agent. Of course, we require that if an agent is sleeping, it will take no action but silence, i.e. $\forall t \in T.\left(r_A^S(t) = \perp \Rightarrow r_A^C(t) = \tau\right)$, where $r_A^S(t)$ denotes agent *A*'s state at time moment *t* in the run *r*. The initial time $t_{init,A}$ of an agent *A* in a run *r* can be formally defined as the time moment $t \in T$ that $r_A^S(t) \ne \perp \wedge \forall t' \in T.(t' < t \Rightarrow r_A^S(t) = \perp)$.

2.5.3 Agent's View of the Environment

The global state S_g of the system at any particular time moment *t* belongs to the set $\prod_{i=1}^{n} S_{A_i,t} \times \Sigma_{A_i,t}$. However, each agent *A* can view the externally visible states and actions of the agents in $Env_{A,t}$. In other words, an agent *A* can only view the part of S_g in the space $\prod_{X \in Evn_{A,t}} S_{X,t}^V \times \Sigma_{X,t}^V$. Agent *A*'s view of the system state at a time moment *t* is defined as a mapping $View_{A,t}$ from global state $S_g \in \prod_{i=1}^{n} S_{A_i,t} \times \Sigma_{A_i,t}$ to a value in $\prod_{X \in Evn_{A,t}} S_{X,t}^V \times \Sigma_{X,t}^V$ as follows.

$$View_A\left(\left\langle\left\langle s_1, s_1', c_1\right\rangle, \ldots, \left\langle s_n, s_n', c_n\right\rangle\right\rangle\right) = \left\langle\left\langle s_{i_1}, c_1'\right\rangle, \ldots, \left\langle s_{i_k}, c'_{i_k}\right\rangle\right\rangle \quad (17)$$

where $Env_{A,t} = \left\{A_{i_1}, A_{i_2}, \ldots, A_{i_k}\right\}$, $i_u = v$ implies that $s_{i_u} = s_v$, $c'_{i_u} = c_v$ if $c_v \in \Sigma_{A_v}^V$, and $c'_{i_u} = \tau$ if $c_v \in \Sigma_{A_v}^I$. Because an agent's view is only a part of the system's global state, two different global states become equivalent from its view. The following formally

defines the relation.

$$\forall x, y \in \prod_{i=1}^{n} S_{A_i} \times \Sigma_{A_i} .(x \approx_A y \Leftrightarrow View_A(x)=View_A(y)). \quad (18)$$

It is easy to see that the binary relation $\approx_A$ is an equivalence relation.

2.5.4 Execution History

Although an agent may not be able to distinguish two global states, the histories of the runs leading to states may be different. An intelligent agent may decide to take different actions according to the history rather than only depending on the visible global state. Let t be any given time moment. The *history* of a run r up to t, written as $r{\downarrow}t$, is a mapping that is the restriction of r to the subset $\{x \le t \mid x \in T\}$ of T. The history of a run up to t in the view of an agent A, denoted by $View_A(r{\downarrow}t)$, is the mapping from the subset $\{x \le t \mid x \in T\}$ of time moments to its views of the system's states in the run r. It can be defined as follows.

$$View_A(r \downarrow t) = \lambda u.View_{A,u}(r(u)) \text{ , for all } u \in T \text{ and } u \le t . \quad (19)$$

Similarly, we define $View_A(r)$ to be an agent A's view of a run r, and $View_A(r_B)$ to be agent A's view of agent B's behaviour in a run r. The equivalence relation defined on the state space can be extended to histories and runs as follows.

$$r_1 \approx_A r_2 \Leftrightarrow View_A(r_1) = View_A(r_2) \quad (20)$$

$$(r_1 \downarrow t_1) \approx_A (r_2 \downarrow t_2) \Leftrightarrow View_A(r_1 \downarrow t_1) = View_A(r_2 \downarrow t_2) \quad (21)$$

Before we finish this section, we introduce some further notation. Let A be any given agent in a multi-agent system. Let c_1, ..., c_n , ... $\in \Sigma_A - \{\tau\}$ be the sequence of non-silent actions taken by agent A in a run r and t_1, t_2, ..., t_n, ...$\in T$ are the times of the actions, i.e. $r_A^C(t_i) = c_i$ for all i =1, 2, ..., n, ... At a time moment $t \in T$, we say that c_n is agent A's current action, and c_{n+1} the *next* action, if $t_n \le t < t_{n+1}$. We write

$Current(r_A{\downarrow}t)=<t_n, s_n, c_n>$,
$Next(r_A{\downarrow}t)=<t_{n+1}, s_{n+1}, c_{n+1}>$, and
$Events(r_A{\downarrow}t)=<<t_1, s_1, c_1>, ..., <t_n, s_n, c_n>>$.

2.6 Specification of Behaviour

2.6.1 Patterns of Behaviours

A pattern describes the behaviour of an agent 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 $n \ge 0$. Table 2 gives the meanings of the patterns.

pattern **::=** [**{** event *I* , **}**] **[** || constraint **]**
event ::= **[** time-stamp: **]** **[** action **]** **[** ! state-assertion **]**
action **::=** atomic-pattern **[** ^ arithmetic-expression **]**
atomic-pattern **::=** $ **|** ~ **|** action-variable
| action-identifier **[** (**{** arithmetic-expression *I* , **}**) **]**
time-stamp ::= arithmetic-expression

where a constraint is a first order predicate.

Table 2. Meanings of the patterns

Pattern	Meaning
$	The *wild card*, which matches with all actions
~	The *silence* event
X	*Action variable, which* matches an action
P^*k*	A sequence of *k* events that match pattern *P*
! *Predicate*	The state of the agent satisfies the predicate
Act (a_1, ...a_k)	An action *Act* that takes place with parameters match (a_1, ...a_k)
[p_1,..., p_n]	The previous sequence of events match the patterns p_1, ..., p_n

Formally, Let p be a pattern. We write $B: r_A \downarrow t \models p$ to denote that from agent B's viewpoint the behaviour of an agent A in a run r matches the pattern p at time moment t. The relationship |= can be defined inductively as follows.

Definition 2.

We write $B: r_A \downarrow t \models p$, if there is an assignment α such that $B: r_A \downarrow t \models_\alpha p$, which is inductively defined as follows.

- $B{:}r_A{\downarrow}t \models_\alpha$ [$], for all agents A, B, runs r and time moments t;
- $B{:}r_A{\downarrow}t \models_\alpha [\tau]$, if $View_B(r_A{\downarrow}t)(t) = \tau$,
- $B{:}r_A{\downarrow}t \models_\alpha [x]$, if $Current(View_B(r_A{\downarrow}t)) = \alpha(x)$;
- $B{:}r_A{\downarrow}t \models_\alpha [t_x{:}\ C(e_1,...,e_n)\ !\ pred(s)]$, if $Current(View_B(r_A{\downarrow}t)) = <t_c, S, C(\alpha(e_1),...,\alpha(e_n))>$, S satisfies the predicate $\alpha(pred(s))$, $\alpha(t_x)=t_c$.
- $B{:}r_A{\downarrow}t \models_\alpha [p\hat{}k]$, if $Events(View_B(r_A{\downarrow}t))=<..., <t_1, s_1, c_1>, <t_2, s_2, c_2>, ..., <t_v, s_v, c_v>>$, where $v = \alpha(k)$, and for all i=1,2,..., v, $B{:}r_A{\downarrow}t_i \models_\alpha [p]$;
- $B{:}r_A{\downarrow}t \models_\alpha [p_1, p_2, ..., p_v]$, if $Events(View_B(r_A{\downarrow}t))=<..., <t_1, s_1, c_1>, <t_2, s_2, c_2>, ..., <t_v, s_v, c_v>>$, and for all i=1,2,..., v, $B{:}r_A{\downarrow}t_i \models_\alpha [p_i]$.
- $B{:}r_A{\downarrow}t \models_\alpha (p \parallel Constraint)$, if $B{:}r_A{\downarrow}t \models_\alpha p$ and $\alpha(Constraint)$ is true. □

Informally, $B{:}r_A{\downarrow}t \models_\alpha p$ means that agent A's behaviour in a run r matches a pattern p at time moment t from an agent B's point of view under assignment α. An assignment α for a set X of variables is a mapping that assigns values to variables in X.

2.6.2 Scenarios of Environment

The use of scenarios in agent oriented analysis and design has been proposed by a number of researchers, for example [21, 31,19]. We define scenario as a set of typical combinations of the behaviours of related agents in the system. In addition to the pattern of individual agents' behaviour, SLABS also provides facilities to describe global situations of the whole system. The syntax of scenarios is given below.

Scenario **::=** Agent-Name : pattern **|** atomic-predicate
| ∃ [arithmetic-exp] Agent-Var ∈ Caste-Name: Pattern
| ∀ Agent-Var ∈ Caste-Name: Pattern
| Scenario & Scenario **|** Scenario ∨ Scenario **|** ~ Scenario

An atomic predicate in a scenario can be an expression in one of the following forms:

- *Agent*∈*Caste*, the agent is in the caste at that time;
- *AgentA*=*AgentB* (or *AgentA*≠*AgentB*), the identifiers indicates the same (or different) agent;
- A set relation expression, which may contain expressions in the form of {*X*∈*Caste* | *X* : *Pattern*}, which is the set of agents

whose behaviour matches the pattern; or

- An arithmetic relation, which may contain an expression in the form of $\mu X \in Caste.Pattern$, which is the number of agents in the caste whose behaviour matches the pattern.

The semantics of scenario descriptions are given in Table 3.

Table 3. Semantics of scenario descriptions

Scenario	Meaning
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 exists at least m agents in caste C whose behaviour matches pattern P where the default value of the optional expression m is 1
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 two examples of scenarios.

(1) $\exists$ p∈Parties: t_{2000}: [nominate-president(Bush)] || t_{2000}=(March/2000).

It describes the situation that at least one agent in the caste called Parties took the action nominate-president(Bush) at the time of March 2000.

(2) (μ x∈ Voter: [vote(Bush)] > μ x∈ Voter: [vote(Gore)])

It describes the situation that there are more agents in the caste Voter who took the action of vote(Bush) than those in the caste who took the action of vote(Gore).

Let *Sc* be a scenario. We write $A: r \downarrow t \models Sc$ to denote that from agent *A*'s point of view, the scenario *Sc* occurs at time moment *t* in a run *r*.

Definition 3.

From an agent *A*'s point of view, a scenario *Sc* occurs at time moment *t* in a run *r*, iff $A: r \downarrow t \models Sc$, which is inductively defined as follows.

- $A: r \downarrow t \models B: p \Leftrightarrow A: r_B \downarrow t \models p$; (22)
- $A: r \downarrow t \models Sc_1 \wedge Sc_2 \Leftrightarrow A: r \downarrow t \models Sc_1$ and $A: r \downarrow t \models Sc_2$; (23)
- $A: r \downarrow t \models \neg Sc \Leftrightarrow A: r \downarrow t \models Sc$ is not true ; (24)
- $A: r \downarrow t \models \forall x \in G.(x: Sc) \Leftrightarrow A: r_x \downarrow t \models Sc$, for all $x \in G$; (25)
- $A: r \downarrow t \models \exists x \in G.(x: Sc) \Leftrightarrow A: r_x \downarrow t \models Sc$, for some $x \in G$ (26)

□

2.6.3 Rules

In SLABS, an agent's behaviour is defined by a set of transition rules to describe its responses to environment scenarios.

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

In a behaviour rule, the pattern on the left-hand-side of the –> symbol describes the pattern of the agent's previous behaviour. The scenario describes the situation in the environment, which specifies the behaviours of the agents in its environment. The where-clause is the pre-condition of the action to be taken by the agent. The event on the right-hand-side of –> symbol is the action to be taken when the scenario happens and if the pre-condition is satisfied. The agent may have a non-deterministic behaviour. The expression prob in a behaviour rule is an expression that defines the probability for the agent to take the specified action on the scenario. SLABS also allows specifications of non-deterministic behaviours without giving the probability distribution. In such cases, the probability expression is omitted. It means that the probability is greater than 0 and less than or equal to 1.

Let *R* be the set of runs in a formal model of agent-based system. To define the semantics of rules, we first define a probabilistic space as follows.

For each scenario *Sc*, agent *A*, and constraint $Cn(r, t) \rightarrow \{tt, ff\}$, we define $R_{\hookleftarrow}(Sc, A, Cn)$ as a subset of histories $H=\{r\downarrow t \mid r \in R, t \in T\}$ such that

$$R_{\hookleftarrow}(Sc, A, Cn) = \{ r\downarrow t \mid r \in R, t \in T, A: r \downarrow t \models Sc , Cn(r, t) \} \quad (27)$$

Let H^* be the set that contains *H* and all the subsets in the form of $R_{\hookleftarrow}(Sc, A, Cn)$ and closed under set complementary, finite intersections and countable unions. Therefore, H^* constitutes a σ–field. A probabilistic space can then be constructed over the σ–field H^* by associating a probabilistic distribution over H^*. Let $Pr(R_{\hookleftarrow}(Sc, A, Cn)$ be the probability that the scenario *Sc* with constraint *Cn* occurs from agent *A*'s point of view. Notice that agent *A*'s behaviour matches a pattern *p* can be expressed equivalently as a scenario $(A:p)_A$. The order pair <*R*, *Pr*> is called the probabilistic model of the agent-based system.

Definition 4.

Let $R_A = 'p \mid (exp) \rightarrow e$ if *Sc* where *Cn'* be a rule for agent *A*, where *Sc* is a scenario and *Cn* is a constraint. We say that in a probabilistic agent-based system <*R*, *Pr*> the agent *A*'s behaviours satisfy the rule R_A and write $<R, Pr>: A \models R_A$, if

$$Pr(R_{\hookleftarrow}(A:p\#e, A, True) \mid R_{\hookleftarrow}(Sc \wedge (A:p), A, Cn)\ ) = exp, \quad (28)$$

where $p\#e = [p_1, p_2, ..., p_n, e]$, if $p = [p_1, p_2, ..., p_n]$. □

3. EXAMPLE

A number of examples of intelligent agents have been specified in SLABS in our previous papers, which include the Mae's personal assistant Maxim [27], Ants [32], a simple communication protocol, speech-act [28] and the evolutionary multi-agent ecosystem Amalthaea [35], etc. In this paper, we use an example of a distributed algorithm to demonstrate how SLABS supports the modularity and composability of formal specifications.

The algorithm is for synchronisation of the accesses to critical regions in distributed systems. The following is the original informal specification of the algorithm given in the textbook [33] (Page 267).

> *When a process wants to enter a critical region, it builds message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available, can be used instead of individual messages.*
>
> *When a process receives a request message from another process, the action it takes depends on its state with respect to*

the critical region named in the message. Three cases have to be distinguished:

1. *If the receiver is not in the critical region and does not want to enter it, it send back an OK message to the sender;*
2. *If the receiver is already in the critical region, it does not reply. Instead, it queues the request.*
3. *If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and send nothing.*

After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends OK message to all processes on its queue and deletes them all from the queue.

It is assumed that processes are executed in a distributed system concurrently. The communications between the processes are reliable. They all use the synchronisation algorithm to access shared resources as critical regions. Regarding each process as an agent, this algorithm can be easily translated into SLABS' formal specification as follows.

```
CRegionUsers
VAR Region: String
ACTION
      Request(RName, AName: String, TimeStamp:Integer);
      PermissionOK(RName, AName: String, timestamp:Int)
VAR State: {Want, InRegion, Waiting, Finishing, Free};
      Queue: List of (RName, AName: String, TStamp:Int);
      RequestTime: Int
All: CRegionUsers
[!State=Want] |–> t: Request(Region, MyName, t)
      ! State'=Waiting & RequestTime'= t;
[!State=Free] |–> PermissionOK(Region, AN, TStamp);
      if ∃A: CRegionUsers.[Request(Region, AN, TStamp)]
[!State= InRegion]
    |–> Queue'=Queue#( Region, AN, TStamp);
      if ∃A: CRegionUsers.[Request(Region, AN, TStamp)]
[!State=Waiting & RequestTime≥TStamp]
    |–> PermissionOK(Region, AN, Tstamp);
      if ∃A: CRegionUsers.[Request(Region, AN, TStamp)]
[!State=Waiting & RequestTime<TStamp]
    |–> Queue'=Queue#( Region, AN, Tstamp);
      if ∃A: CRegionUsers.[Request(Region, AN, TStamp)]
[!State=Waiting] |–> !State'=InRegion;
      if ∀A: CRegionUsers.[ PermissionOK(Region,
                  MyName, RequestTime), $^k]
[!State=Finishing]
    |–> ForAll (Region, AN, TStamp) in Queue Do
        PermissionOK(Region, AN, TStamp) End
        !State'= Free & Queue = <>
```

The specification of the algorithm is generic. It does not over specify the specific feature of the critical region. The issues like when and how an agent wants to use the region and when to finishes the uses of the resource are left to the specification of the agent. For a specific resource, a sub-caste can be specified to make it a critical region. For example, a shared printer can be controlled by the caste specified as follows.

```
PrinterUsers<=CRegionUsers
(Region:='Printer', State:=Free)
```

Agents that use the printer can be specified as an instance of the caste *PrinterUsers* or dynamically joint the caste. Similarly, other critical regions can be specified through other sub-castes so that their uses are synchronised. The properties of the algorithms can be deducted from the generic specification, which hold for all sub-castes. Moreover, different synchronisation algorithms can be specified and used by the same agent to access different shared recourses. For example, suppose the token ring synchronisation algorithm is specified by a caste *TokenRingUsers*, and the access of a scanner is controlled by the algorithm, i.e. a caste *ScannerUsers* is specified as a sub-caste of *TokenRingUsers*. An agent A that uses both the printer and the scanner at the same time, can be declared as an instance of both castes of *PrinterUsers* and *ScannerUsers*.

This example shows that specifications can be modularised by using caste with parameters and composed through sub-castes and instantiations and used through instances, i.e. agents.

4. CONCLUSION

In this paper, we presented the formal specification language SLABS for multi-agent systems. The SLABS language integrates a number of novel language facilities that support the development of agent-based systems, especially the specification of such systems. Among these facilities, the notion of caste plays a crucial role. A caste represents a set of agents in a multi-agent system that have same capability of performing certain tasks and have same behaviour characteristics. Such common capability and behaviour can be the capability of speaking a language, using an ontology, following a communication and/or collaboration protocol, and so on. It is a notion that generalises the notion of types in data type and the notion of classes in object-oriented paradigm. This facility can be effectively used to specify or implement a number of notions proposed in agent-oriented methodologies, such as the notions of role, team, agent society, organisation, and so on. For example, 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 determine their roles at run time. Based on the caste facility, a number of other facilities in SLABS are defined. For example, the environment of an agent can be described as the agents of certain castes. A global scenario of a multi-agent 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 SLABS have shown that these facilities are powerful and useful for the formal specification of agents in various models and theories in a modular and composable way.

4.1 Related Work

The model of software agents used in this paper is closely related to the work by Lesperance *et al* [34], which also focused on the

actions of agents. However, there are two significant differences. Firstly, they consider objects and agents are different types of entities, while we consider them as the same type of encapsulated computational entities. As argued in [28], we consider objects as a degenerate form of agents that obey simple behaviour rules and open to its environment so that everything in the environment can affect its behaviour. As a consequence of regarding objects as deferent entities from agents, they allow an agent to change the state of objects in the environment while we only allow an agent to modify its own state. Secondly, the most important difference is, of course, there is no notion of caste or any similar facility in their system.

The notion of groups of agents has been used in a number of researches on the multimodality logic of rationale agents, such as in Wooldridge's work [11], etc. However, such notion of groups of agents is significantly different from the notion of caste, because there is neither inheritance relationship between the groups, nor dynamic instance relationship between an agent and a group. Their only relationship is the membership relationship.

Many agent development systems are based on object-oriented programming. Hence, there is a native and primitive form of castes as classes in OO paradigm. However, although agents can be regarded as evolved from object and caste as evolved from class, there are significant differences between agents and objects, and thus between caste and class. Therefore, the new notion deserves a new name.

In our previous papers [27~29], an agent's membership to a caste is statically determined by agent description. Static membership has a number of advantages, especially its simplicity and easiness to prove the properties of agents. Examples have shown that introducing some state variables to represent the role that an agent is playing can specify dynamic team formation [29]. In this paper, we revised the notion of caste to allow a dynamic membership facility in order to specify and implement dynamic team formation. The formal and informal definitions of the language SLABS given in this paper supersedes our previous ones. An advantage of this approach is that the dynamic formation of a team can be explicitly specified. It is natural to specify the dynamic process of evolution in a multi-agent system [35].

Another design decision that we faced in the design of SLABS was whether we should allow redefinition of behaviour rules in the specification of sub-castes. An advantage of disabling redefinition is that provable properties of a supper caste are inherited by all sub-castes. The example given in this paper shows that it enables composable modular specifications.

4.2 Further Work

There are a number of open problems that need further investigation. Although the language facilities in SLABS, such as caste and scenario, were first introduced as a specification facility, we believe that they can be easily adopted in an agent-oriented programming language for the implementation of multi-agent systems. How to implement these facilities is an important issue in the design and implementation of agent-oriented programming languages.

The design of SLABS is aimed to support as many agent-oriented methodologies as possible. How to link from such methodologies to formal specifications in SLABS deserves further investigation. We are currently working on tools and graphic notations to support the development of formal specifications in SLABS. In [35], a process and a diagrammatic notation for modelling multi-agent systems are proposed so that formal specifications in SLABS can be derived from models of multi-agent systems represented in diagrams.

ACKNOWLEDGEMENT

The author is most grateful to his colleagues at Oxford Brookes University, especially Mr. Ken Brownsey, Prof. David Duce, Ms. Sue Greenwood, Mr. John Nealon, Dr. Nick Wilson, et al., for discussions on agent technology and many related subjects. The author would also like to thank Prof. Huaglory Tianfield for many invaluable discussions.

REFERENCES

[1] Albayrak, S. Agent-Oriented Technology for Telecommunications, CACM 44(4) (April 2001), 30-33.
[2] Chien, S. *et al.* The Techsat-21 Autonomous Space Science Agent, in Proc. of AAMAS'2002 (Italy, July 2002) 570-577.
[3] Heinze, C., *et al*. Interchanging agents and humans in military simulation, AI Magazine 23(2) (Summer 2002) 37-47.
[4] Jennings, N. R., Wooldridge, M. J. (eds.). Agent Technology: Foundations, Applications, And Markets. Springer, 1998.
[5] Webster, P., Smith, M., Murtagh, P. Four Killed as Airbus Crashes. The Guardian (27 June 1988), page 1.
[6] ACM, The Risks Digest: Forum on Risks to the Public in Computers and Related Systems 7(10~12) (June 1988).
[7] Brazier, F. M. T., Dunin-Keplicz, B. M., Jennings, N. R., Treur, J. DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework, in Int. J. of Cooperative Information Systems 1(6) (1997), 67-94.
[8] 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), 473-484.
[9] Singh, M. P. Semantical considerations on some primitives for agent specification, in Intelligent Agents, Wooldridge, M., Muller, J. & Tambe, M. (eds), LNAI 1037, Springer, 1996, 49-64.
[10] Chainbi, W., Jmaiel, M., Abdelmajid, B. H., Conception, Behavioural Semantics and Formal Specification of Multi-Agent Systems, in Multi-Agent Systems, Zhang, C., Lukose, D. (eds), LNAI 1544, Springer, 1998, 16-28.
[11] Wooldrighe, M., Reasoning About Rational Agents, The MIT Press, 2000.
[12] Ambroszkiewicz, S. and Komar, J., A model of BDI-agent in game-theoretic framework, in [13], 1999, 8-19.
[13] Myer, J-J., Schobbens, P-Y. (eds.), Formal Models of Agents - ESPRIT Project ModelAge Final Workshop Selected Papers, LNAI 1760, Springer, 1999.
[14] Wooldridge, M. J. and Jennings, N. R. Agent theories, architectures, and languages: a survey, in Intelligent Agents, LNAI 890, Springer-Verlag, 1995, 1-32.
[15] Ossowski, S., and Garcia-Serrano, A. Social structure in artificial agent societies: implications for autonomous problem-solving agents, in Intelligent Agents V, Muller, J. P., Singh, M. P. and Rao, A. S. (eds.), LNCS 1555, Springer, 1999, 133-148.
[16] Fisher, M. If Z is the answer, what could the question possibly be? in Intelligent Agents III, Muller, J., Wooldridge, M., Jennings, N. (eds.). LNAI 1193, Springer, 1997, 65-66.
[17] Kinny, D., Georgeff, M., and Rao, A. A methodology and

modelling technology for systems of BDI agents, in Agents Breaking Away: Proc. of MAAMAW'96, LNAI 1038, Spriger-Verlag, 1996.
[18] Moulin, B. and Cloutier, L. Collaborative work based on multiagent architectures: a methodological perspective, in Soft Computing; Fuzzy Logic, Neural Networks and Distributed Artificial Intelligence, Aminzadeh, F. and Jamshidi, M. (eds.), Prentice-Hall, 1994, 261-296.
[19] Moulin, B., and Brassard, M. A scenario-based design method and an environment for the development of multiagent systems, in First Australian Workshop on Distributed Artificial Intelligence, Lukose, D. and Zhang C. (eds.), LNAI 1087, Springer-Verlag, 1996, 216-231.
[20] Wooldridge, M., Jennings, N. and Kinny, D. A methodology for agent-oriented analysis and design, in Proc. of ACM Third International Conference on Autonomous Agents, Seattle (WA, USA, 1999) 69-76.
[21] 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 1555. Springer, 1999, 317-330.
[22] Conrad, S., Saake, G., Turker, C. Towards an Agent-Oriented Framework for Specification of Information Systems, in [13], 1999, 57-73.
[23] D'Inverno, M., Kinny, D., Luck, M., and Wooldridge, M. A formal specification of dMARS, in Intelligent Agents IV, Singh, M. P., Rao, A. Wooldridge, M. (eds.) LNAI 1365, Springer, 1998, 155-176.
[24] Luck, M. and d'Inverno, M. A formal framework for agency and autonomy, in Proc. of First Int Conf on Multi-Agent Systems, AAAI Press / MIT Press, 1995, 254-260.
[25] D'Inverno, M. and Luck, M. Understanding Agent Systems. Springer, 2001.
[26] Jennings, N. R. Agent-Oriented Software Engineering, in Multi-Agent System Engineering, Proc 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 1647, Springer, 1999, 1-7.
[27] Zhu, H. Formal Specification of Agent Behaviour through Environment Scenarios, Formal Aspects of Agent-Based Systems, Rash, J. *et al*. (eds.), LNCS 1871, Springer, 263-277.
[28] Zhu, H. SLABS: A Formal Specification Language for Agent-Based Systems, Int. J. of Software Engineering and Knowledge Engineering 11(5) (Nov. 2001), 529-558.
[29] Zhu, H. The role of caste in formal specification of MAS, in Proc. of PRIMA'2001, LNCS 2132, Springer, 1-15.
[30] Spivey, J. M. The Z Notation: A Reference Manual (2nd edition), Prentice Hall, 1992.
[31] 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 1356, Springer, 1998, 313-327.
[32] Zhu, H. A formal specification language for MAS engineering, in Proc. of 2nd International Workshop on Agent-Oriented Software Engineering (May 29, 2001).
[33] Tanenbaum, A. S. and van Steen, M., Distributed Systems: Principles and Paradigms, Prentice Hall, 2002.
[34] 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, Wooldridge, M., Muller, J., and Tambe, M. (eds.) LNAI 1037, Springer-Verlag, 1996, 331-346.
[35] Zhu, H. Formal Specification of Evolutionary Software Agents, Proc. of ICFEM'2002 (Shanghai, China, Oct. 2002)