SLABS: A Formal Specification Language for Agent-Based Systems

Hong Zhu
School of Computing and Mathematical Sciences, Oxford Brookes University
Gipsy Lane, Headington, Oxford, OX3 0NW, UK
Email: hzhu@brookes.ac.uk, Fax: ++44 1865 483666

Abstract

Being autonomous, proactive and adaptive, an agent-based system may demonstrate emergent behaviours, which are neither designed by the developers nor expected by the users of the system. Whether or not such emergent behaviours are advantageous, methods for the specification of agent behaviours must be developed to enable software engineers to analyse agent-based systems before they are implemented. This paper presents a formal specification language SLABS for agent-based systems. It is a model-based specification language defined based on the notion of agents as encapsulations of data, operations and behaviours. The behaviour of an agent is defined by a set of rules that describe the action/reaction of the agent in certain environment scenarios. The style and expressiveness of the language is demonstrated by examples like ants, personal assistant and speech-act style of agent communications.

Keywords: Agent-based systems, formal specification language, software engineering, scenario description

1. Introduction

Agent technology has long been predicted to be the next mainstream computing paradigm, see, for example, [1, 2, 3]. It is perceived to be a viable solution for large-scale industrial and commercial applications. However, researches on agent-based systems have been mainly an AI endeavour so far. The majority of extant agent applications are developed in an ad hoc fashion without proper analysis and specification of system's requirements, and without systematic verification and validation of the properties of the implemented system. For a long time, software engineers and computer scientists alike have learned from many incidents that the behaviours of a system should be understood and documented before the system is put in operation, even before a serious implementation effort is made. One of such incidents that is related to autonomous software agents in particular is the crash of Air France's Airbus 320 at an air show in June 1988 [4, 5]. Airbus 320 was the first fly-by-wire passenger aircraft in the world. In other words, it was controlled by an autonomous agent. 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. More than a dozen of years has passed and autonomous agents have gained much wider applications see e.g. [6], but open questions remain

about how to specify autonomous agents' behaviour and how to verify and prove their properties so that such tragedy can be prevented. Being autonomous, proactive and adaptive, an agent-based system can be very complicated, and sometimes may demonstrate emergent behaviours, which are neither designed by the developers nor expected by the users of the system. 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].

The past few years have seen increasing research interests in agent-oriented software development methodology. Existing work can be classified into three main groups. The first is towards the theoretical foundations for specifying and modelling agent-based systems. Work in this direction includes the development of logic as the foundation for the specification and proof of agents' properties, which is dominated by researches on temporal and multi-modal logic of agents. The work on modal logic of knowledge can be dated to Hintikka's work published in 1962 [8] and Kripke's work in 1963 [9]. Recent work on the logic of knowledge includes Fagin, Halpern, Moses, Vardi, and their colleagues' study of knowledge in the context of distributed systems [10, 11, 12, 13]. In the AI community, much work has been focused on modelling agents' rational behaviour by introducing modalities for belief, desire and intention. Among the most well-known are Rao and Georgeff's modal logic framework of BDI agents [14], Singh's study of the modal operators [15], Chainbi, Jmaiel and Abdelmajid's language L_{ω} based on linear time logic [16], and Wooldridge's work on use of such logic in the specification of and reasoning about rational agents [17]. Game theory has also found its position in the formalisation of agent models, e.g. [18]. A great number of formal models of agents have been proposed and investigated in the literature, see e.g. [19, 20]. Most of them are based on an internal mental state model of agents, such as the BDI model, but with subtle differences, yet some are based on a model of the external social behaviours of collaborative agents, e.g. [21]. It is still far away from reaching a unified model of agents. As pointed out by Michael Fisher [22], 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. This makes reasoning about agent specification difficult.

The second group of researches is on the development process and development methods for engineering agent-based systems. A number of proposals have been advanced in the literature, which include Kinny, Georgeff and Rao's Agent modelling techniques for systems of BDI agents [23], Moulin et al.'s MASB [24, 25], and Wooldridge, Jennings and Kinny's methodology for agent-oriented analysis and design [26], and many others, see [27] for a survey. These works mostly focused on diagrammatic notations that support the analysis and design of multi-agent systems in software engineering processes. Some of the notations extend object-oriented methods and notations such as UML. Some introduce new models for agents and corresponding new diagrammatic notations as well. How such diagrammatic notations are related to the logic and formal models of agents remains as 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. Brazier, Dunin-Keplicz, Jennings, and Treur *et al*.'s work on DESIRE [7] is perhaps the most well known in this direction. Another example of this type is Conrad, Saake and Turker's

specification language ETL [28]. The use of existing formal specification languages, such as Z, has also been explored to specify agent architecture [29] and concepts related to agents [30]. Despite the large number of publications on agents in the literature, there is little research on language facilities that support the development of large-scale complicated multi-agent systems. It is impractical to use a logic notation directly in the specification and reasoning about large-scale multi-agent systems, because such a specification will be a lengthy and complicated logic formula that consists of mathematical notations and symbols. The modularity achieved in agent-based systems by decomposing the functions and tasks of a system into a number of agents are completely lost in such a logic formula. In particular, there are few 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 [31]. What is most important is the lack of facilities that can clearly state how agents' behaviours are related to the environment. This paper searches for such language facilities that support the specification of agent-based systems in the context of software engineering.

The main contribution of the paper is a set of language facilities for formal specification of agent-based systems. Of course, it would not be possible to define the formal semantics of a model-based specification language without a model. In order to avoid the drawbacks of using a specific model of agents as discussed above, a simple but widely applicable model of agents and agent-based systems was proposed in [32]. This model can be considered as weak agency according to [20]. It is further developed and formally defined in this paper. The set of language facilities proposed in the paper includes a modular structure suitable for the formal specification of multi-agent systems, a scenario description mechanism for defining agents behaviour in the context of environment situations, and a notion of caste as a collection of agents that have same behaviour and structural characteristics. These facilities are integrated together into a formal specification language called SLABS, which stands for *S*pecification *L*anguage for *A*gent-*B*ased *S*ystems. This language is also independent of any particular agent theory, and independent of any particular agent communication languages or protocols. Its semantics are formally defined using the model.

The paper is organised as follows. Section 2 reviews the informal model of agent-based systems proposed in [32] and further discusses the rationale underlying the model. Section 3 presents a formal model of multi-agent systems. Section 4 describes the syntax and semantics of the SLABS language. Section 5 illustrates the style and expressiveness of SLABS specifications. It gives three examples of formal specifications of different types of agent-based systems, which include ants, a personal assistant and speech-act communication in agent society. Section 6 is the conclusion of the paper, which discusses future work.

2. Multi-agent systems

SLABS is a model-based formal specification language for agent-based systems. This section gives the preliminary notions of multi-agent systems underlying the formal model defined in the next section.

2.1. The notion of agents

Agency is the most important notion in agent-oriented and agent-based computing, though what agenthood exactly means is a matter of controversy. People tend to characterise agents as computation systems that have certain properties, see e.g. [33, 34]. Among many such properties are the following.

(a) *Autonomous*: the capability of performing autonomous actions.

(b) *Pro-activity*: the capability of exhibiting opportunistic and goal-directed behaviour and taking initiative where appropriate.

(c) *Responsiveness*: the capability of perceiving the environment and responding in a timely fashion to the changes that occur in it.

(d) *Social*: the capability of interacting with other artificial agents and humans when appropriate in order to complete their own problem solving and to help others with their activities.

These properties have been used to explain the differences between objects and agents. For example, when an object is told that its trousers are on fire (when it receives such a message), it will jump into a river to put the fire out (if it is the method defined for such a message) no matter whether the message is true or false. In contrast, when an agent is told that his trousers are on fire, he would first check if the message were true to determine if he should change his belief if the agent does not believe his trousers are on fire before receiving the message. Consequently, he may do nothing if the message is false. By doing so, the agent demonstrates an autonomous behaviour. A typical AI point of view of the differences between object-oriented programming and agent-oriented programming can be found in Shoham's work on agent-oriented programming, where agents are considered as a specialisation of objects. Table 1 below is from Shoham [35].

Table 1. AI view of OOP and AOP

	OOP	**AOP**
Basic unit	Object	Agent
Parameters defining the state of basic unit	Unconstrained	Beliefs, commitments, capabilities, choices, ...
Process of computation	Message passing and response methods	Message passing and response methods
Types of message	Unconstrained	Inform, request, offer, promise, decline, ...
Constrains on methods	None	Honesty, consistency, ...

The same view of the relationship between agents and objects can be found in a number of publications, for example, Figure 1 is from [30]. According to this view, one would expect that object-oriented programming languages of full strength should be perfectly suitable for programming agents and agent-based systems. However, this contradicts the observations that there are serious difficulties in programming agent-based systems in Java [36].

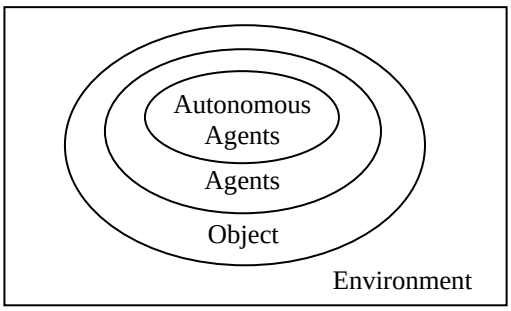

Figure 1. The entity hierarchy as in [30]

In [32], it was argued that agents are a generalisation of objects. In fact, when use computation terminology to define the AI terms such as beliefs, capabilities, and so on, the following table can be obtained from the above.

Table 2. Computational view of OOP and AOP

	OOP	**AOP**
Basic unit	Object	Agent
Parameters defining the state of basic unit	First order data	First order data, second order entity (such as functions, logic clauses, etc.)
Process of computation	Message passing and response methods	Perceiving the environment and reacting (such as event capture, event-driven)
Types of message	Method call	Unconstrained
Constrains on methods	none	none

As shown in Table 2, the parameters of an object's state are not unconstrained, but rather strictly restricted to be first order data, which in object-oriented programming languages are also objects. However, the parameters of an agent's state can be beliefs, intentions, plans and goals, etc., which, mathematically speaking, are higher order entities. Of course, an agent can also have first order data as parameters of its states. An agent's process of computation can be much more complicated than an object's process. It is not just receiving messages and then calling the corresponding methods. Instead, it can also perceive the changes in the environment (even the situation of no changes in the environment for a given period of time) and take actions according to its internal state. Message passing is just one method by which the changes in the environment can attract an agent's attention. Nevertheless, an agent can discard an incoming message without any response. Messages with illocutionary forces are only one of many methods that agents communicate with each other, although it is an important method. The importance of agent communication via such messages is that the content of the message can be in higher order. For example, a message can ask an agent to do something, where 'do something' is an action. So in agent-oriented programming, the types of messages are not restricted in comparison with objects. Finally, it is impractical to set a constraint on an agent's methods to be honest and consistent. Of course, it will be nice to set such constraints so that there would be no problem with the security of mobile agents. In summary, agents are extensions of objects, rather than specialisations. Consequently, agent-oriented languages must be developed for both specifying and programming agent-based systems.

In [32], a constructive definition of agents was proposed. Agents are defined as encapsulations of data, operations and behaviour. The data of an agent represents the internal state of the agent, which can be divided into two parts: a visible part and an invisible part. The visible part of the state is visible from outside of the agent, such as the facial expressions in Mae's personal assistants [45]. The invisible part of the state is internal to the agent, such as the desires, beliefs, and intentions of the agents in a BDI model. The operations are the conceptually atomic actions that an agent is capable of. Such an action can be visible from the outside, or invisible as an internal event. Behaviour is a collection of sequences of state changes and operations that can be performed by the agent in the context of its environment. As Jennings [31] pointed out, autonomous means that agents 'have control both over their internal state and over their own behaviour'. The encapsulation of data,

operation and behaviour means that each agent has its own rules that govern its behaviour. Such rules have to be explicitly specified for each agent rather than defined by the language using a set of default rules. In this paper, objects are considered as having no control over their behaviour because an object has to execute a method whenever it receives a message that calls the method. In other words, the language defines the behaviour of all objects by the default rule of 'if receive a message, then execute the corresponding method'. However, objects' behaviour can be considered as a degenerate case of agents, if such a simple and uniform pattern of behaviour can be explicitly specified as a rule of behaviour. There is no reason why the possibility that an agent adopts such behaviour should be ruled out.

The above discussion can be summarised by the following pseudo-equation that characterises the notion of agent.

$$\text{Agent} = \langle \text{Data, Operations, Behaviour} \rangle_{\text{Environment}} \quad (1)$$

2.2. The notion of Castes

In object-oriented languages, a class is considered as the set of objects of common structure and function. Objects are instances of classes. Similarly, the notion of caste is defined here as a set of agents with the same structural and behavioural characteristics, where the term caste is used to distinguish from classes in object-oriented languages. Agents are therefore instances of castes. If an agent is specified as an instance of a caste, it has the structure and behaviour characteristics of the caste. However, in addition to those inherited structure and behaviour, an agent can also have additional behaviour and structure descriptions of its own. Agents of the same caste may play the same role in an agent-based system, especially in a society of agents. An example of behaviour characteristics is that an agent follows a specific communication protocol to communicate with other agents. Therefore, such a communication protocol can be specified by defining a caste with the protocol as behaviour characteristic.

In a similar way to classes, inheritance relationships can be defined between castes. A caste is defined as a sub-caste of existing castes by indicating the super-castes. A sub-caste inherits the structure and behaviour descriptions from its super-castes. It may also have some additional actions and obey some additional behaviour rules if they are specified in the sub-caste declaration. Some of the parameters of the super-castes may also be instantiated in a sub-caste. As it will be shown in section 5, the caste and inheritance facilities provide a powerful vehicle to describe the normality of a society of agents. Multiple inheritances are allowed to enable an agent to belong to more than one society and play more than one role in the system at the same time. The notion of castes can also be expressed in the form of a pseudo-equation as follows.

$$\text{Castes} = \{ \text{agents} \mid \text{structure characteristics} \ \& \ \text{behaviour characteristics} \} \quad (2)$$

2.3. Environments and multi-agent systems

It is widely recognised that the power of agent-based systems can be best demonstrated in a dynamic environment [6, 37] because an agent can adapt its behaviour into the environment to achieve its designed purpose. 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' [31, 33]. Obviously, there are two key differences between objects and agents with respect to their relationships with the environment. Firstly, agents are active in the sense they observe their environment and they are always prepared to take

actions to effect the environment. In contrast, objects are passive, they are driven by the messages sent by the objects in the environment. Because of this, agents are sometimes considered as active objects. Secondly, agents selectively observe a part of the environment that they are interested in, while objects are open to all objects in the environment. In fact, an object executes a method no matter who sends the message. It cannot even identify the sender of the message. These highlight the differences in the degrees of encapsulation in objects and agents. Encapsulation means to draw a boundary between the entity and its environment and protect the entity by controlling the accesses across the boundary. In object-oriented languages, the boundary enhances the access to the object's state via method calls so that the integrity of the objects' state can be ensured. However, such a boundary is weak because all entities in the environment of an object can send a message to the object and hence call the method. The object cannot even tell where the message came from. In other words, an object's boundary is open to the environment. In contrast, an agent should be able to selectively respond to the actions and changes of certain entities in the environment rather than to everything. In agent-oriented systems, the encapsulation of behaviour means that each agent has its own subset of entities in the environment that can influence its behaviour. Such a subset of entities must be specified explicitly, rather than defined by the language uniformly. Again, it is easy to see that objects can be considered as special cases of agents in degenerate form when the subset of influential entities contains all entities in the environment.

Therefore, the specification of an agent-based system must also specify how the environment affects the behaviour of the agent. To do so, one must first answer the question what is the environment of an agent. A simple answer to this question is that in a multi-agent system, the environment of an agent consists of a number of agents and a number of objects. However, as discussed above, an object is a degenerate form of agent. The behaviour of an object is simply to respond to every message sent to the object by executing the corresponding method. Based on this understanding of the relationship, our second design decision is to specify a multi-agent system as a set of agents, nothing but agents. In other words, agent is the only type of computation unit in an agent-oriented system. This can be represented in the form of pseudo-equations as follows.

$$\text{Multi-agent system MAS} = \{\text{Agent}_n\}_{n \in I} \tag{3}$$

$$\text{Environment(Agent, MAS)} \subseteq \text{MAS} - \{\text{Agent}\} \tag{4}$$

2.4. Communications between agents

Communication plays a crucial role in multi-agent systems. Agents must communicate with each other to collaborate, negotiate, and to compete with each other as well. The discussion so far has not explicitly addressed the communication issues of multi-agent systems. However, dividing an agent's states and actions into visible and invisible parts has already given agents the capability of communicating with each other. Human beings communicate with each other by taking actions. People speak, shout, sing, laugh, cry, and write to communicate, even make gesture or other body languages movements. All these means of communication are 'visible' actions. People also utilise visible states to communicate. For example, the colour of traffic lights indicates whether a pedestrian should cross the road. One may show a smiling face to indicate he/she is happy and a sad face to indicate that he/she is 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 of the communication 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 model proposed above already assigned agents the ability of communication, since agents as senders can take visible actions and change their visible states and

agents as receivers can observe the visible actions and the visible states. The details of the protocols and meanings of such actions and state values should be left for software engineers to specify rather than pre-defined by the model or the language. The SLABS language based on this model is capable of specifying agent communication languages and protocols. An example of SLABS specification of a simple communication protocol can be found in [38]. Section 5.3 also gives an example of SLABS specification that is related to the communication and collaboration between agents.

The above discussion can be summarised by the following pseudo-equations.

Communication from agent A to B = A.Action + B.Observation (5)

3. A formal model of multi-agent systems

This section defines a formal model of multi-agent systems based on the informal model presented in the previous section. It is used as the semantic domain to define the semantics of the SLABS language in section 4.3.

3.1. States and Actions

A *multi-agent system* consists of a finite set of *agents* $\{A_1, A_2, ...A_n\}$. These agents belong to a hierarchy of castes $C_1, C_2, ...C_m$. A binary relation $\prec$, called the inheritance relation, is defined on the castes. The inheritance relation is required to be a partial ordering on castes. Let $A \in C$ denote that agent A belongs to caste C. It is also required that for all agents A and castes C and C',

$$A \in C \wedge C \prec C' \Rightarrow A \in C'. \quad (6)$$

Each agent A has its own *state space*, which is a non-empty set S_A. Each state consists of two disjoint parts, the externally visible part and the internal part. The external part is visible to all agents in the system, while the internal part is not visible to any other agents in the system. Therefore, $S_A = S_A^V \times S_A^I$, where S_A^V and S_A^I are the externally visible part and the internal part of the state space, respectively. An agent is capable of taking an action with various parameters at any particular time when it decides to do so. The set of actions is a finite non-empty set, denoted by Σ_A. An action can also be either externally visible or internal (hence externally invisible). It is assumed that an agent cannot take two actions at the same time. Thus, $\Sigma_A = \Sigma_A^V \cup \Sigma_A^I$, where $\Sigma_A^V \cap \Sigma_A^I = \varnothing$ and Σ_A^V is the subset of externally visible actions and Σ_A^I is the subset of internal actions.

3.2. Run and Time

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. A *run r* of a multi-agent system is a mapping from time T to the set $\prod_{i=1}^{n} S_{A_i} \times \Sigma_{A_i}$. 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. $\exists t_0 \in T. \forall t \in T. (t_0 \le t)$; (7)

2) Unbounded in the future, i.e. $\forall r \in \mathrm{R}. \exists t \in T. (t > r)$; (8)

3) Uniformity, i.e. $\forall\, t_1, t_2, t_3 \in T.\ (t_2 > t_1 \Rightarrow t_3 + t_2 - t_1 \in T)$. (9)

□

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 $\rho \geq 0$.

Lemma 1.

For all subsets T of real numbers that satisfy properties of Eq. (7)~(9), 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 $T=\{r \mid r \in R \text{ and } r \geq t_0\}$. In the former case, the time index set T is called *discrete*, and in the latter case, T is called continuous.

Proof.

Let $\rho(T) = inf\{t-s \mid s, t \in T \wedge t>s\}$. By using properties of Eq. (7)~(9), it is easy to prove that when $\rho > 0$, $T=\{t_n \mid t_n = t_0 + n\rho, n=0, 1, 2, \ldots\}$. When $\rho=0$, $T=\{r \mid r \in R \text{ and } r \geq t_0\}$. □

The real number $\rho(T)$ defined above in the proof is called the *resolution* of the time index set T.

On the other hand, it is easy to see that any discrete time index set of the form $T=\{t_n \mid t_n = t_0 + n\rho, n=0, 1, 2, \ldots\}$ satisfies the properties of Eq. (7)~(9). Any subset $T=\{r \mid r \in R \text{ and } r \geq t_0\}$ of real numbers also satisfies the propertiesof Eq. (7)~(9). Therefore, the model defined below applies to both discrete time index and continuous time index. Without loss of generality, subsequently, it is assumed that $t_0 = 0$.

For any given run r of the system, a mapping h from T to $S_A \times \Sigma_A$ is called 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 part of $r(t)$ in $S_A \times \Sigma_A$. 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.

3.3. Assumptions

In the construction of the model, it is assumed that a multi-agent system has the following properties.

- *Instantaneous actions*

It is assumed that actions are instantaneous, i.e. they take no time to complete. This means, actions taken at different times are considered as different events even if they are the same action. Notice that, an event that takes a period of time to complete can be modelled by two actions: one for the start of the event and one for the finish of the event.

- *Silent moments*

It is assumed that an agent can take no action at a time moment t. In such a case, the agent is silent at time t. For the sake of convenience, silence is treated as a special action and denoted by the symbol τ. Therefore, it is assumed that for all agents A, $\tau \in \Sigma_A^V$.

- *Separatebility*

It is also assumed 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 $r_A^C(t) \neq \tau \Rightarrow \forall x \in T.(t < x \leq t + \varepsilon \Rightarrow r_A^C(x) = \tau)$, 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. The agent A is *sleeping* before time moment $t_{init,A}$. A special symbol $\perp \notin S_A$ is used to indicate such a state of an agent. Of course, it is required 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 $r_A^S(t) \neq \perp \wedge \forall t' \in T.(t' < t \Rightarrow r_A^S(t') = \perp)$.

3.4. Agent's view of the environment

The global state of the system at any particular time moment belongs to the set $\prod_{i=1}^{n} S_{A_i} \times \Sigma_{A_i}$. Each agent can view other agents' externally visible states and actions, but it may only selectively observe some of the agents' external states and actions. These agents constitute the environment of the agent. The environment of an agent *A* is denoted by $Env_A \subseteq \{A_1, A_2, ..., A_n\}$. Therefore, an agent *A* can only view a part of the state of the system, which belongs to the set $\prod_{X \in Evn_A} S_X^V \times \Sigma_X^V$. A given agent *A*'s view of the system is defined as a mapping $View_A$ from global state $\prod_{i=1}^{n} S_{A_i} \times \Sigma_{A_i}$ to $\prod_{X \in Evn_A} S_X^V \times \Sigma_X^V$ as follows:

$$View_A\left(\left\langle \left\langle s_1, s_1', c_1 \right\rangle, \left\langle s_2, s_2', c_2 \right\rangle, ..., \left\langle s_n, s_n', c_n \right\rangle \right\rangle\right) = \left\langle \left\langle s_{i_1}, c_1' \right\rangle, \left\langle s_{i_2}, c_2' \right\rangle, ..., \left\langle s_{i_k}, c'_{i_k} \right\rangle \right\rangle \quad (10)$$

where $Env_A = \left\{A_{i_1}, A_{i_2}, ..., A_{i_k}\right\}$, $i_u = v$ 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 $c_v \in \Sigma_{A_v}^I$. An agent's behaviour is influenced by its view of the system's state. 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.

$$\forall x, y \in \prod_{i=1}^{n} S_{A_i} \times \Sigma_{A_i} .(x \approx_A y \Leftrightarrow View_A(x) = View_A(y)). \quad (11)$$

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

3.5. Execution history

Although an agent may not be able to distinguish two global states, the history of the run 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 $\left\{x \leq t \middle| x \in T\right\}$ 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 $\left\{x \leq t \middle| x \in T\right\}$ of time moments to its views of the system's states in the run *r*. It can be defined as follows.

$$View_A\left(r \downarrow t\right)(u) = View_A(r(u)), \text{ for all } u \in T \text{ and } u \leq t. \quad (12)$$

Similarly, $View_A(r)$ is defined to denote an agent *A*'s view of a run *r*, and $View_A(r_B)$ to denote agent *A*'s view of agent *B*'s behaviour in a run *r*. Notice that, the symbol $View_A$ is now overloaded. The equivalence relation defined on the state space can be extended to histories and runs as follows.

$$r_1 \approx_A r_2 \Leftrightarrow View_A(r_1) = View_A(r_2) \quad (13)$$

$$(r_1 \downarrow t) \approx_A (r_2 \downarrow t) \Leftrightarrow View_A(r_1 \downarrow t) = View_A(r_2 \downarrow t) \quad (14)$$

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 time moments when the actions are taken place, i.e. $r_A^C(t_i) = c_i$ for all *i* =1, 2, ..., *n*, ... At a time moment $t \in T$, c_n is called agent *A's* current action, and c_{n+1} the

next action, if $t_n \le t < t_{n+1}$. The expression *Current*($r_A \downarrow t$)=<t_n, s_n, c_n> denotes that agent *A*'s current action is c_n which was taken at time t_n and its state was s_n. Similarly, the expression *Next*($r_A \downarrow t$)=<t_{n+1}, s_{n+1}, c_{n+1}> denotes that the next action taken by agent *A* in a run *r* at time moment *t* is c_{n+1} at the time moment t_{n+1} with state s_{n+1}. The expression *Events*($r_A \downarrow t$)=<<t_1, s_1, c_1>, ..., <t_n, s_n, c_n>> denotes the sequence of events taken by agent *A* in the run *r* up to time moment *t*.

4. The SLABS language

This section defines the syntax and semantics of the SLABS language. The meta-language to define the syntax is EBNF, which is given in Table 3. In a syntax definition, the 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 3. 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. For example, A B A B A.
Positive repetition	**{ }⁺**	**{** A **}** means that A may appear at least once.
Parenthesis	**( )**	They are used to change preference.

4.1. Agents and Castes

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

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

There is a most general caste, called AGENT, such that all castes in SLABS are sub-castes of AGENT. The main body of a caste specification in SLABS contains a description of the structure of its states and actions, a description of its behaviour, and a description of its environment. The syntax of caste descriptions is given below in EBNF. It can also be equivalently represented in a graphic form similar to the schema in Z [39].

caste-description **::=**

Caste name **[** <= **{** caste-name **[** *(* instantiation *)* **]/** , **}⁺** ; **]**

[environment-description ; **]**

[structure-description ; **]**

[behavior-description ; **]**

end name

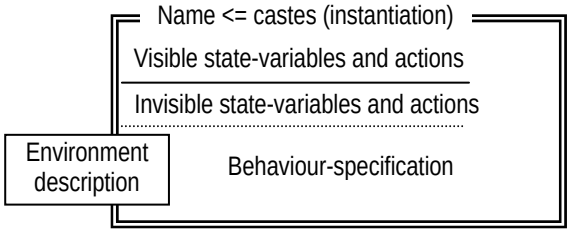

Figure 2. Graphic form of caste specification

The clause 'Caste *New_Caste* <= $Caste_1$, $Caste_2$, ..., $Caste_n$' specifies that the defined caste *New_Caste* is a sub-caste of $Caste_1$, $Caste_2$, ..., $Caste_n$. That is, the defined caste inherits the structure, behaviour and environment descriptions of existing castes $Caste_1$, $Caste_2$, ..., $Caste_n$. When no inherited caste is given in a caste specification, it is by default a sub-caste of the predefined caste AGENT.

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 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 an agent in the caste. The instantiation clause gives the details about how the parameters are instantiated.

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

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 **/** ;**}**$^+$**]**

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

In a caste specification (and agent specification as well), the additional state variables and actions should have no overlap with the state variables, action identifiers and parameter variables defined in the super-castes. Moreover, the castes $Caste_1$, $Caste_2$, ..., $Caste_n$ that it inherits should have no common variables, no common action identifiers, and no common parameters. However, they can overlap with agent names in the environment descriptions.

In SLABS, 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. If no caste name is given in an agent specification, the caste of the agent is by default AGENT. If an agent is an instance of a caste, it must have all the structural, behaviour and environment descriptions given in the caste's specification. 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. The following gives the syntax of agent specifications in SLABS. It can also be represented in a graphic form equivalently.

agent-description **::=**

Agent name **[** : **{** caste-name **[** *(* instantiation *)* **]/** , **}⁺ ;]**

[environment-description ; **]**

[structure-description ; **]**

[behavior-description **]**

end name

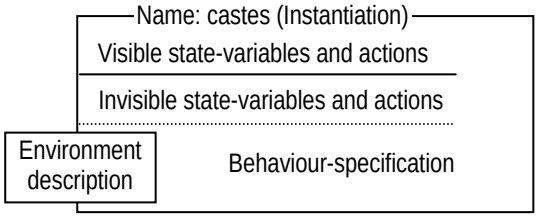

Figure 3. Graphic form of agent specification

If an agent is specified as an instance of a caste, all the parameters in the specification of the caste must be instantiated in the specification of the agent.

4.2. Behaviour

As discussed in section 2, an agent's autonomy is its capability of controlling its internal state and action. An agent changes its state and takes an action as a response to the situation in its environment rather than simply as a response to a request of its services. Various models of agents such as the BDI model have been proposed and investigated to represent and reason about agent's autonomous behaviour, e.g. [14, 17]. The structure description facility that SLABS provides is intended to specify such a structural model of agents. However, a structural model alone is insufficient to specify agent's autonomous behaviour. A facility is required to specify explicitly how the structural model (such as the belief, desire and intention model) is related to actions and how observations of the environment are related to the changes at internal states. Among many possible forms of such a facility such as procedural specifications and temporal logic formulas, the author of this paper believes that the most effective form is a set of transition rules.

(a) Rules

Based on the characteristics of agent's behaviour, it is recognised that a rule should contain the following parts:

- *Rule-name*: which enables us to indicate which rule is used in the reasoning of the system's behaviour;
- *Scenario*: which specifies the situation when a rule is applicable;
- *Transition*: which specifies the action or state change to take place when the rule is applied;
- *Probability distribution*: the probability that the rule is applied when the scenario occurs;
- *Pre-condition*: the condition for the action to take place.

The syntax of a rule is given below.

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 specification of non-deterministic behaviour 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 1.

(b) Scenarios

The notion of scenario has been used in a number of areas in computing with different meanings. For example, in UML, scenarios are described as the sequences of messages passing between the system and the objects that represent the users. In the application of scenarios in testing software requirements [40], a scenario is described as an activity list that represents a task of human computer interaction. Generally speaking, a scenario is a set of situations that might occur in the operation of a system [41]. No matter how scenarios are described, their most fundamental characteristic is to put events in the context of the history of behaviour. Here, in a multi-agent system, a scenario is a set of typical combinations of the behaviours of related agents in the system.

The use of scenarios and use cases in requirements analysis and specification has been an important part of object-oriented analysis; see for example, [42]. 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. The object-oriented paradigm is lack of a method to analyse the consistency between use cases (or scenarios) and requirements models and a method to synthesise requirements models from use cases or scenarios, although such methods exist for structured analysis [41]. As extensions to OO methodology, the use of scenarios in agent oriented analysis and design has been proposed by a number of researchers, for example [27, 43, 44].

In the design of SLABS, it was 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 itself 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.

In SLABS, a basic form of scenario description is a set of patterns. 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 $n \geq 0$. Table 4 gives the meanings of the patterns.

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

where a constraint is a first order predicate.

Table 4. Meanings of the patterns

Pattern	Meaning
$	The *wild card*, it matches with all actions
~	The *silence* event
Action variable	It matches an action

$P^{\wedge}k$	A sequence of k events that match pattern P
Action (a_1, a_2, ...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

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

| arithmetic-relation

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

| ∀ Agent-Var ∈ Caste-Name: Pattern

| Scenario & Scenario

| Scenario ∨ Scenario

| ~ Scenario

where an arithmetic relation can contain an expression in the form of μAgent-var∈Caste.Pattern, which is an expression whose value is the number of agents in the caste whose behaviour matches the pattern.

arithmetic-relation **::=** expression relational-op expression

expression **::=** atomic-expression **|** (expression) **|** expression numerical-op expression

atomic-expression **::=** numerical-constants **|** numerical-variable **|** μ Agent-var∈Caste.Pattern

relational-op **::=** < **|** > **|** ≤ **|** ≥ **|** = **|** ≠

numerical-op **::=** + **|** - **|** / **|** *

The semantics of scenario descriptions are given in Table 5.

Table 5. Semantics of scenario descriptions

Scenario	Meaning
A: P	The situation when agent A's behaviour matches pattern P
∀X∈C: P	The situation when the behaviours of all agents in caste C match pattern P
∃ $_{[m]}$X∈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
μ X∈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
¬ S	The situation when scenario S is not true

The following are some examples of scenarios.

(1) ∃ 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).

4.3. Formal semantics

Given an agent specification, it is easy to see how the structural description corresponds to the visible state space, invisible state space, visible actions and invisible actions in the formal model given in section 3. It is also easy to see how the environment of an agent defined in the formal model can be uniquely determined by any given specification in SLABS. These correspondence relationships constitute the static semantics of the SLABS language. For the sake of space, the formal definition of these correspondences between the syntax and the formal model is omitted. The following defines the dynamic semantics of the language.

(a) Pattern matching

Let *p* be a pattern. The expression $B: r_A \downarrow t \models p$ denotes 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.

An agent *A*'s behaviour in a run *r* matches a pattern *p* at time moment *t* from an agent *B*'s point of view, written $B: r_A \downarrow t \models p$, 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* and all 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 $Current(View_B(r_A\downarrow t)) = \alpha(x)$;
- $B: r_A\downarrow t \models_\alpha [\ t_x: C(e_1, e_2, ...,e_n)\ !\ pred(s)\ ||\ Constraint\ ]$, if $Current(View_B(r_A\downarrow t)) = <t_c, S, C(\alpha(e_1), \alpha(e_2),...,\alpha(e_n))>$, *S* satisfies the predicate $\alpha(pred(s))$, $\alpha(t_x)=t_c$, and $\alpha(Constraint)$ is true.
- $B: r_A\downarrow t \models_\alpha [p\text{^}k\ ]$, 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*, $B: r_A\downarrow t_i \models_\alpha [\ p_i\ ]$. □

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*.

(b) Scenario satisfaction

Let *Sc* be a scenario. The expression $A: r \downarrow t \models Sc$ denotes 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.

- $A:r\downarrow t \models B:p \Leftrightarrow A:r_B \downarrow t \models p$; (15)
- $A:r\downarrow t \models Sc_1 \wedge Sc_2 \Leftrightarrow A:r\downarrow t \models Sc_1$ and $A:r\downarrow t \models Sc_2$; (16)
- $A:r\downarrow t \models \neg Sc \Leftrightarrow A:r\downarrow t \models Sc$ is not true; (17)
- $A:r\downarrow t \models \forall x\in G.(x:Sc) \Leftrightarrow A:r_x \downarrow t \models Sc,$ for all agents x in G ; (18)
- $A:r\downarrow t \models \exists x\in G.(x:Sc) \Leftrightarrow A:r_x \downarrow t \models Sc,$ for some agent x in G . (19)

□

(c) Rules

Let R be the set of runs in a formal model of an agent-based system. To define the semantics of rules, the following first defines a probabilistic space.

For each scenario Sc, agent A, and constraint Cn, which is a predicate $Cn(r, t) \rightarrow \{tt, ff\}$ on the run r up to time moment t, define $R_{\hookleftarrow}(Sc, A, Cn)$ as a subset of histories $H=\{r\downarrow t \mid r\in R, t\in T\}$ such that

$$R_{\hookleftarrow}(Sc, A, Cn) = \{\, r\downarrow t \mid r\in R,\ t\in T,\ A:r\downarrow t \models Sc,\ Cn(r, t)\,\} \quad (20)$$

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 complement, 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 ordered pair $\langle R, Pr\rangle$ is called the probabilistic model of the agent-based system.

Definition 4.

Let R_A= 'p |(exp)→e if Sc where Cn' be a rule for agent A, where Sc is a scenario and Cn is a constraint. It is said that in a probabilistic agent-based system $\langle R, Pr\rangle$ the agent A's behaviours satisfy the rule R_A and write $\langle R, Pr\rangle$: $A \models R_A$, if

$$Pr(R_{\hookleftarrow}(A{:}p\#e, A, True) \mid R_{\hookleftarrow}(Sc \wedge (A{:}p), A, Cn)\ ) = exp, \quad (21)$$

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

5. Examples

This section gives three examples to illustrate SLABS' style and expressiveness. These examples demonstrate that various types of agent-based systems can be formal specified in the SLABS language and the formal model of agent-based systems. The readers are referred to, for example, [29, 30] for formal specification of agent-based systems in other existing specification languages.

5.1. Ants

The multi-agent system of ants is a typical example of a reactive agent system that may demonstrate emergent behaviours. In this system, food is scattered in the field, and ants try to find food and to move the food back to their home. When an ant walks across the field, its hormone spreads on the path in the field. The density of ant's hormone decreases as time ticks away. The food in the field decreases when taken by the ants.

Field

VAR Hormone: N×N→N; Food: N×N→N;

All: Ants

[$] |→ Hormone'(n,m) = Hormone(n,m)–1 , if ¬∃x∈ Ants:[!position=(n,m)] ;
[$] |→ Hormone'(n,m) = Hormone(n,m)+100, if ∃x∈ Ants:[!position=(n,m)]
[$] |→ Food'(n,m)=Food(n,m)–k, if ∃[k]x∈ Ants:[Take_Food ! position=(n,m)]

Figure 4. Specification of the field in the ants system

An ant can move around in the field searching for food. In such a search mode, the ant's movement is rather random. Once it has found some food, it takes a bite and carries it back home by tracing the hormone it left behind its path. Once back home with food, the ant comes back with other ants to the location to get more food. In the specification of ants, each ant can be in one of three mental states, which are *search*, *way_back* and *way_out*. Here, way_back is the state when an ant has found some food in the field and is carrying some food back home. An ant is in the state of way_out when it goes to the field to get food after an ant comes home with food. The following gives a specification of the *Ants*.

Ants

VAR Position: N×N;
ACTION Move(Direction); Take_Food; Save_Food;

VAR State: {search, way_back, way_out};

Field; all: Ants

[Move(u) ! State=search] |(0.5)→ Move(u)!(position' = New(position, u)),
if field: [!Food(position)<ε];
[Move(u) ! State=search] |(0.5)→ Move(v)!(position' = New(position, v)),
if field:[!Food(position)<ε]; where u≠v
[$] |→ Take_Food !State = Way_back, if field: [!Food(position)≥ε];
[!State = Way_back & position ≠ home] |→ Move(u),
where u=arg $\text{Min}'_{u\in \text{Direction}}$ field: Hormone(New(position, u));
[!State = Way_back & position = home] |→ Save_Food ! State = Way_out;
[$] |(0.8)→ ! State = Way_out, if ∃x∈ Ants:[Save_Food]
[!State = Way_out] |→ Move(u), where u=arg $\text{Max}_{u\in \text{Direction}}$ field:Hormone(New(position, u));

Figure 5. Specification of the Ants caste

In the above specification, *Home*, *Direction* and the function *New*(*position*, *u*) are defined as follows using a notation borrowed from *Z* for specifying global concepts in a formal specification.

Home ≅ (0, 0)
Direction ≅ {*east*, *west*, *north*, *south*}
New: *N*×*N*× *Direction* → *N*×*N*

New((*n*, *m*), *east*) = (*n*+1, *m*);
New((*n*, *m*), *west*) = (*n*–1, *m*), if *n* > *0*;
New((*n*, *m*), *north*) = (*n*, *m*+1);
New((*n*, *m*), *south*) = (*n*, *m*–1), if *m* > *0*.

Figure 6. Specification of global concepts in the ants systems

5.2. The Maxims system

Maes' Maxims system [45] is a personal assistant agent for handling emails. The system consists of Eudora and the user as the environment of the Maxims agent.

Eudora contains a number of mail folders and can perform a number of operations on emails. The operations include reading a mail in the inbox, deleting a mail from the mail box, archiving a mail in a folder, sending a mail to a number of addresses, and forwarding a mail to a number of addresses. For the sake of simplicity, it is assumed that there is only one mailbox named as inbox in the following specification of the software. The behaviour of Eudora is a typical object's behaviour. That is, whoever sends a command to Eudora, it will perform the corresponding operation. This behaviour is explicitly specified in the following SLABS specification through two facilities. Firstly, in the specification of its environment, it is made clear that all agents in the environment have influences on its behaviour. Secondly, in the specification of the behaviour rules, it is clearly specified that the only condition for Eudora to take an action is that some agent sends a command to ask it to do so.

```
Eudora
VAR     Inbox: list(Mail) ;  Folders : string → list(Mail);
ACTION  Read(Mail);          Delete(Mail);        Archive(Mail, Folder);
        Send(Mail, list(address));                Forward(Mail, list(address));

All: Agent

<Read>     [ !mail∈ Inbox] |→ Read(mail), if ∃u∈ Agent:[Command(Eudora, Read(mail))];
<Delete>   [ !mail∈ Inbox] |→ Delete(mail) ! mail∉ Inbox,
                                 if ∃u∈ Agent:[Command(Eudora, Delete(mail))];
<Archive>  [ !mail∈ Inbox] |→ Archive(mail, folder) !mail∉ Inbox & mail∈ folder,
                                 if ∃u∈ Agent: [Command(Eudora, Archive(mail, folder))];
<Send>     [ !mail∈ Inbox] |→ Send(mail, list(address)),
                                 if ∃u∈ Agent:[Command(Eudora, Send(mail, list(address)))];
<Forward>  [ !mail∈ Inbox] |→ Forward(mail, list(address)),
                                 if ∃u∈ Agent:[Command(Eudora, Forward(mail, list(address)))];
```

Figure 7. Specification of Eudora in Mae's Maxim system

A user's behaviour is non-deterministic. The specification given below only shows the possible actions a user may take. There are two types of such actions. One is to command an agent to take an action; the other is to grant a permission of taking a suggested action. Notice that, the rules that specify the user's behaviour have an unknown probabilistic distribution.

```
User
ACTION   Command(Agent, Action)   (* Command the agent to take an action *)
         Grant(Agent, Action)     (* Grant the agent to take a suggested action *)

Eudora, Maxim

<Grant suggestion>      [$] |→ Grant(Maxim, action); if Maxim: [Suggest(user, action)]
<Not grant suggestion>  [$] |→ Command(Eudora, another-action);
                                  if Maxim: [Suggest(user, action)]
                                  where (another-action ≠ action)
<Act-as-Predicted>      [$] |→ action, if Maxim: [Predict(user, action)]
<Act-not-Predicted>     [$] |→ action, if Maxim: [Predict(user, another-action)]
                                  where (another-action ≠ action)
<Set do-it level>       [$] |→ Command(Maxim, set-do-it(r));  where (0<r<1)
<Set tell-me level>     [$] |→ Command(Maxim, set-tell-me(r)); where (0<r<1)
```

Figure 8. Specification of the User in Mae's maxim system

Maxims observes the user's actions and the state of Eudora. When a mail is delivered to Eudora's Inbox, Maxim finds out the best match in the set of emails that user has handled and the action that the user has taken in the situation. It, then, makes a suggestion or a prediction of user's actions. It also communicates with the user through facial expressions. Once the user grants a suggestion, Maxims commands Eudora to perform the action. These are specified by a set of rules. The <Command> rule states that maxims can command Eudora to take an operation on behalf of the user if its confidence level is greater than or equal to the do-it threshold. The <Suggest> rule states that it makes a suggestion if the confidence level is higher than tell-me threshold but lower than do-it. The <Predict> rule states that it predicts the user's action if the confidence level is lower than tell-me. There are also rules in the specification of Maxims that specify its reaction to the user's responses to the agent's suggestions and predictions. For the sake of space, the definition of the function *Best-match*: (*Mail* × List(*Mail* × *Action*)) → (*Action* × *Confidence-level*) is omitted.

Maxims

VAR Facial-expression: {Working, Suggestion, Unsure, Gratified, Pleased, Surprised, Confused}
Action Command(Agent, Action); Suggest(Agent, Action); Predict(Agent, Action);

VAR Tell-me-level, do-it-level : Real

User, Eudora

<Suggest> [$] |→ Suggest(user, action) ! Facial-expression = Suggestion,
if Eudora: [!mail∈ Inbox] & user: [X[n]^k],
where (action, confidence) = Best-match(mail, {X[n]^k| n=1,..,k})
& tell-me-level ≤ confidence < do-it-level
<Predict> [$] |→ Predict(user, action) ! Facial-expression = Unsure,
if Eudora: [!mail∈ Inbox] & user: [X[n]^k],
where (action, confidence) = Best-match(mail, {X[n]^k| n=1,..,k})
& confidence < tell-me-level
<Work> [$] |→ action ! Facial-expression = Working, if Eudora: [!mail∈ Inbox] & user: [X[n]^k],
where (action, confidence)= Best-match(mail, {X[n]^k| n=1,..,k})
& do-it-level-level ≤ confidence
<Set tell-me-level> [$] |→ ! tell-me-level = r, if user: [Command(Self, set-tell-me(r))]
<Set do-it-level> [$] |→ ! do-it-level = r, if user: [Command(Self, set-do-it(r))]
<Gratified> [Suggest(user, action)] |→ action ! Facial-expression = Gratified,
if user:[Grant(self, action)];
<Surprised> [Suggest(user, action)]|→ ! Facial-expression = Surprised, if user:[another-action],
where another-action ≠ action
<Pleased> [Predict(user, action)] |→ ! Facial-expression = Pleased, if user:[action];
<Confused> [Predict(user, action)] |→ ! Facial-expression = Confused, if user:[another-action],
where another-action ≠ action
<Command> [Suggest(user, action] |→ action ! Facial-expression = Working,
if user:[Grant(self, action)]

Figure 9. Specification of the behaviour of Mae's Maxim system

Maxims' autonomous behaviour is reflected in the specification in SLABS. Firstly, it selectively observes the environment. It observes the state of Eudora to determine if there is a mail in its Inbox. It also observes the action taken by the user to learn from the user's behaviour. Secondly, as discussed above, its behaviour is not simply determined by the event, but also the history of the user's behaviour. It can even take actions without the user's command. Of course, an agent may also have a part of behaviour that simply obeys the user's command. The maxims agent obeys the user's commands on setting tell-me and do-it thresholds. The rules <Set do-it-level> and <Set tell-me-level> specify such behaviour.

5.3. Speech-act and collaborative behaviour

In a multi-agent system, agents communicate to each other and collaborate with each other. To illustrate

SLABS' capability of specification of such behaviour, the following example describes the differences between illocutionary forces in communications as their effects on agent behaviour. As in [46, 47], illocutionary forces are classified into 7 types.

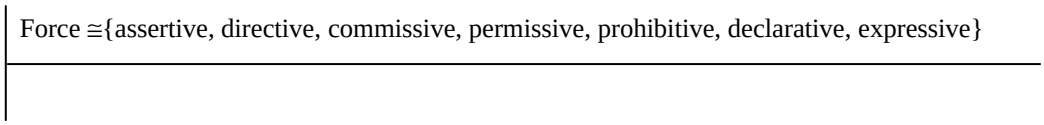

Figure 10. Illocutionary forces

A caste of agents, called social-agents, is defined, which can communicate with each other. Informally, the action Communicate(X, Y, Z) is to send a message Y to agent X with illocutionary force Z, where the message Y is an action.

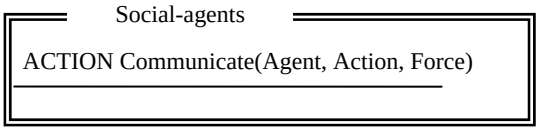

Figure 11. Specification of social-agents

The meaning of the communication depends on the illocutionary force, which is understood by all the agents in the society. An approach to the formal specification of the meaning of an illocutionary force is to define a set of rules for how agents should interpret such communications. However, in a human society, people play different roles. The same sentence may have different effects depending on who says it to whom. For example, a commander can give an order to a soldier and expect the soldier to perform the action as ordered. However, if the same message is communicated in the opposite direction from the soldier to the commander, it would not be socially acceptable and one would not expect any action to be taken. Therefore, instead of giving a set of behaviour rules for all agents to interpret the meanings of illocutionary forces in the same way, this example adds one more twist to show how to specify the situation in which different agents can interpret them differently according to their roles in the system. The situation specified in the example is a work place, where the agents are divided into two groups: the workers who perform various tasks and the managers who assign tasks to workers. Here, the details about how a manager makes management decisions are left open.

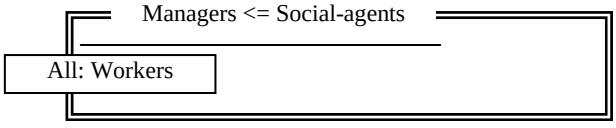

Figure 12. Specification of Manager agents as sub-caste of Social-agents

There are two basic requirements of a worker agent. One is to follow the orders of its manager; and the other is to report to its manager when it finishes a task. The <Obey-Order> rule in the caste Workers specifies that a worker agent must take the order from its manager agent Boss (which is a parameter of the caste) and perform the action as the boss ordered. The <Report> rule specifies that after a worker agent finishes a job, it must report to its boss. Every worker agent must satisfy these rules, but a manager agent does not need to. Here, the situation is simplified so that managers themselves are not organised in a hierarchical structure.

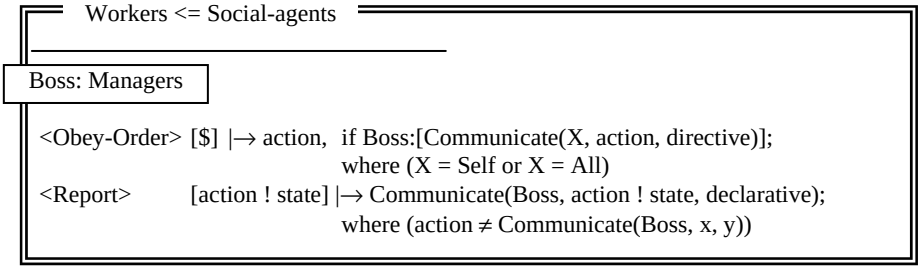

Figure 13. Specification of Workers' behaviour

The personality of an agent can also be described by rules governing its behaviour. For example, a naughty apprentice would take any chance to play (i.e. take actions that are neither ordered nor approved by the boss, even prohibited by the boss, when the boss is busy in communication with other workers and does not give him an order for a while). However, as a worker, an apprentice will still obey the orders of the boss. Such behaviour is specified in the caste Apprentices.

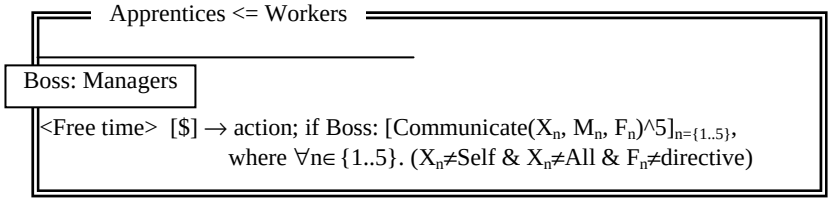

Figure 14. Specification of Apprentices' behaviour

In contrast, an experienced worker will not only follow the rules but also do more than what is ordered.

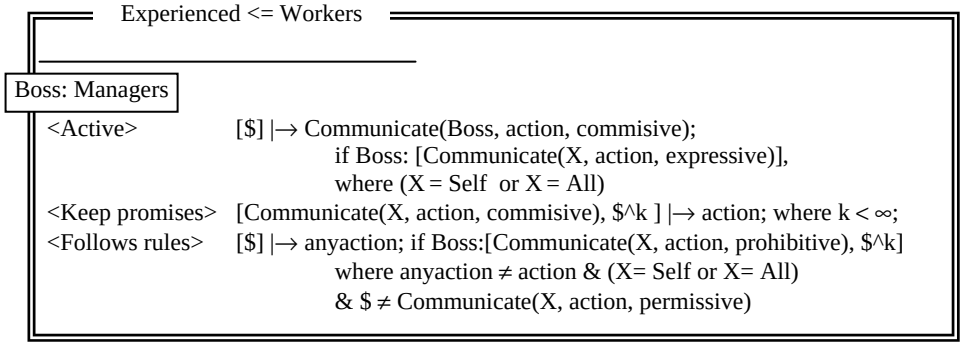

Figure 15. Specification of Experienced Workers' behaviour

The <Active> rule specifies that if the boss asked everybody or the worker in particular if there is any volunteer to take an action, the worker would do that. The <Keep promises> rule specifies that if the agent promised to do something, he will eventually take the action. The <Follows rules> rule specifies that the agent will never take an action that the boss prohibited him or everybody to do, unless the boss subsequently permitted him to do so.

6. Conclusion

This paper presented a language SLABS for the formal specification of multi-agent systems. It integrates a number of novel language facilities that intended to support the development of agent-based systems, especially for the specification of such systems. The example systems and features of agent-based systems specified in

SLABS show that the SLABS language and the agent model underlying the language have the following features.

(1) The language facilities that SLABS provided are powerful to specify agent-based systems in various agent models and theories. Examples are given in this paper for the specification of reactive systems, personal assistants, and agent societies.

(2) The language is capable of specifying many different aspects of agent-based systems, which include communication protocol, collaborative and autonomous behaviour, probabilistic and non-deterministic behaviour, etc.

(3) Specifications of agent-based systems are organised in a modular structure, called castes, that naturally represents the structure and behaviour characteristics of agents in encapsulated units. Each unit represents one type of agents, whose the dependence on the environment is explicitly specified. This not only makes the specification of agents more readable, reusable, and maintainable, but also enables the implementation of a caste to be realised relatively independently based on the information contained in the caste specification.

(4) The model of multi-agent system underlying SLABS is constructive and has a computational interpretation. It clarifies the relationship between object orientation and agent orientation. It is also general enough to cover the agent models and theories in the literature that we know so far.

A direction for further research is a logic and formal proof system to support the verification and refinement of specifications in SLABS. The author believes that SLABS' modular structure should be supportive to the maintenance of formal verification and refinement processes. Another direction for future work is to relate the language facilities to the methodologies for agent-oriented software development.

Acknowledgement

The author is most grateful to his colleagues at Oxford Brookes University, especially Prof. David Duce, Dr. Nick Wilson, Mr. Ken Brownsey, Ms. Sue Greenwood for their comments on earlier drafts of the paper. The author would also like to thank Mr. John Nealon, Mr. Qingning Huo, Mr. Yanglon Zhang, and Dr. Lu Zhang, Dr. Jiangrong Chen, Prof. Jiafu Xu, et al., for the discussions on many related subjects in a number of occasions.

References

1. P. C. Janca, Pragmatic application of information agents, BIS Strategic Report, 1995.
2. P. Sargent, Back to school for a brand new ABC. The Guardian, 12 March, 1992, page 28.
3. Ovum Report, Intelligent Agents: The New Revolution in Software, 1994.
4. P. Webster, M. Smith, and P. Murtagh, Four Killed as Airbus Crashes. The Guardian, 27 June 1988, page 1.
5. ACM, The Risks Digest: Forum on Risks to the Public in Computers and Related Systems 7, Issues 10~12, 27~30 June, 1988, http://catless.ncl.ac.uk/Risks/(7.10.html | 7.11.html | 7.12.html).
6. N. R. Jennings and M. J. Wooldridge (Eds.), Agent Technology: Foundations, Applications And Markets, Springer, Berlin, 1998.
7. F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur, DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework. Int. Journal of Cooperative Information Systems 1, 1997, 67-94.
8. J. Hintikka, Knowledge and Belief, Cornell University Press, 1962.

9. S. Kripke, Semantical analysis of modal logic. Zeitschrift fur Mathematische Logik und Grudlagen der Mathematik 9, 1963.
10. J. Y. Halpern and Y. Moses, Knowledge and common knowledge in a distributed environment. Journal of ACM 37, 1990, 549-587.
11. J. Y. Halpern and L. D. Zuck, A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols. Journal of ACM 39, 1992, 449-478.
12. R. Fagin and J. Y. Halpern, Reasoning about Knowledge and probability. Journal of ACM 41, 1994, 340-367.
13. R. Fagin, J. Y. Halpern, Y. O. Moses and M. Y. Vardi, Reasoning about Knowledge, The MIT Press, Cambridge, Massachusetts, 1995.
14. A. S. Rao and M. P. Georgreff, Modeling Rational Agents within A BDI-Architecture in Proc. of the International Conference on Principles of Knowledge Representation and Reasoning, 473-484, 1991.
15. M. P. Singh, Semantical considerations on some primitives for agent specificatio in Intelligent Agents: Agent Theories, Architectures, and Languages (M. Wooldridge, J. Muller and M. Tambe, Eds), LNAI 1037, 49-64, Springer, 1996.
16. W. Chainbi, M. Jmaiel and B. H. Abdelmajid, Conception, Behavioural Semantics and Formal Specification of Multi-Agent Systems in Multi-Agent Systems: Theories, Languages, And Applications, 4th Australian Workshop on Distributed Artificial Intelligence Selected Papers, Bristane, QLD, Australia, July 1998 (C. Zhang and D. Lukose, eds), LNAI 1544, 16-28, Springer, Berlin, 1998.
17. M. Wooldridge, Reasoning About Rational Agents, The MIT Press, 2000.
18. S. Ambroszkiewicz and J. Komar, A model of BDI-agent in game-theoretic framework in [19], 8-19, 1999.
19. J-J. Myer and P-Y. Schobbens (Eds.), Formal Models of Agents - ESPRIT Project ModelAge Final Workshop Selected Papers, LNAI 1760, Springer, Berlin, 1999.
20. M. J. Wooldridge and N. R. Jennings, Agent theories, architectures, and languages: a survey in Intelligent Agents: Theories, Architectures, and Languages, LNAI 890, 1-32, Springer-Verlag, 1995.
21. S. Ossowski and A. Garcia-Serrano, Social structure in artificial agent societies: implications for autonomous problem-solving agents in Intelligent Agents V: Agent Theories, Architectures, and Languages, (J. P.Muller, M. P. Singh and A. S. Rao, Eds.), LNCS 1555, 133-148, Springer, 1999.
22. M.Fisher, If Z is the answer, what could the question possibly be? On developing formal methods for agent-based systems in Intelligent Agents III: Agent Theories, Architectures, and Languages (J. Muller, M. Wooldridge and N. Jennings, Eds.), LNAI 1193, 65-66, Springer, 1997.
23. D. Kinny, M. Georgeff and A. Rao, A methodology and modelling technology for systems of BDI agents in Agents Breaking Away: Proc. of MAAMAW'96, LNAI 1038, Spriger-Verlag, 1996.
24. B. Moulin and L. Cloutier, Collaborative work based on multiagent architectures: a methodological perspective in Soft Computing; Fuzzy Logic, Neural Networks and Distributed Artificial Intelligence, (F. Aminzadeh and M. Jamshidi, Eds.) , 261-296, Prentice-Hall, 1994.
25. B.Moulin and M. Brassard, A scenario-based design method and an environment for the development of multiagent systems in First Australian Workshop on Distributed Artificial Intelligence (D. Lukose and C. Zhang, Eds.), LNAI 1087, 216-231, Springer-Verlag, 1996.
26. M. Wooldridge, N. Jennings and D. Kinny, A methodology for agent-oriented analysis and design in Proc. of ACM Third International Conference on Autonomous Agents, 69-76, Seattle, WA, USA, 1999.
27. C. A. Iglesias, M. Garijo and J. C. Gonzalez, A Survey of Agent-Oriented Methodologies in Intelligent Agents V: Agent Theories, Architectures, and Languages (J. P. Muller, M. P. Singh and A. Rao, Eds.), LNAI 1555, 317-330, Springer, Berlin, 1999.
28. S. Conrad, G. Saake and C. Turker, Towards an Agent-Oriented Framework for Specification of Information Systems in [19], 57-73, 1999.

29. M. D'Inverno, D. Kinny, M. Luck and M. Wooldridge, A formal specification of dMARS in Intelligent Agents IV: Agent Theories, Architectures, and Languages (M. P. Singh, A. Rao and M. Wooldridge, Eds.), LNAI 1365, 155-176, Springer, 1998.
30. M. Luck and M. d'Inverno, A formal framework for agency and auotnomy in Proc. of First International Conference on Multi-agent Systems, 254-260, AAAI Press / MIT Press, 1995.
31. N. R. Jennings, Agent-Oriented Software Engineering in Multi-Agent System Engineering: Proceedings of 9'th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Valencia, Spain, June/July 1999 (F. J. Garijo and M. Boman, Eds.), LNAI 1647, 1-7, Springer, Berlin, 1999.
32. H. Zhu, Formal Specification of Agent Behaviour through Environment Scenarios in Proc. of NASA First Workshop on Formal Aspects of Agent-Based Systems.
33. N. R. Jennings, On agent-based software engineering. Artificial Intelligence 117, 2000, 277-296.
34. D. B. Lange, Mobile Objects and mobile agents: The future of distributed computing? In Proceedings of The European Conference on Object-Oriented Programming, 1998.
35. Y. Shoham, Agent-oriented programming. Artificial Intelligence 60, 1993, 51-92.
36. D. B. Lange and M. Oshima, Mobile agents with Java: the Aglet API. World Wide Web Journal, 1998.
37. M. Huhns and M. P. Singh (Eds.), Readings in Agents, Morgan Kaufmann, San Francisco, 1997.
38. H. Zhu, The Role of Caste in Formal Specification of MAS in Proc. of PRIMA'2001, 2001.
39. J. M. Spivey, The Z Notation: A Reference Manual, 2nd edition, Prentice Hall, 1992.
40. H. Zhu, L. Jin, and D. Diaper, Application of Task Analysis to the Validation of Software Requirements in Proc. SEKE'99, 239-245, Kaiserslautern, Germany, 1999.
41. H. Zhu and L. Jin, Scenario Analysis in An Automated Requirements Analysis Tool. Journal of Requirements Engineering 5, July 2000, 2~22.
42. I. Jacobson, *et al.*, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992.
43. C. A. Iglesias, M. Garijo, J. C. Gonzalez and J. R. Velasco, Analysis And Design of Multiagent Systems Using MAS-Common KADS in Intelligent Agents IV: Agent Theories, Architectures, and Languages (M. P. Singh, A. Rao and M. J. Wooldridge, Eds.), LNAI 1356, 313~327, Springer, Berlin, 1998.
44. B. Moulin and M. Brassard, A Scenario-Based Design Method And Environment for Developing Multi-Agent Systems in Proc. of First Australian Workshop on DAI (D. Lukose and C. Zhang, eds.), LNAI 1087, 216-231, Springer Verlag, Berlin, 1996.
45. P. Maes, Agents That Reduce Work And Information Overload. Communications of the ACM 37(7), 1994, 31-40.
46. M. P. Singh, A Semantics for Speech Acts. Annals of Mathematics and Artificial Intelligence 8(II), 1993, 47-71.
47. M. P. Singh, Agent Communication Languages: Rethinking the Principles. IEEE Computer, Dec. 1998, 40-47.