Modelling Interaction in Agent Systems

Angela Dalmonte, Mauro Gaspari
Dipartimento di Scienze dell'Informazione
University of Bologna
Piazza di Porta S. Donato 5
40126 Bologna – Italy
The correspondent author will be Mauro Gaspari
phone: +39(51)354502
fax: +39(51)354490
e-mail: gaspari@cs.unibo.it

Abstract

We present a study of the interaction properties of multiagent systems where agents communicate by means of speech act based primitives. We identify a set of basic interaction mechanisms: agent identity, asynchronous message passing, implicit receive primitive, which are closed to those of the actor model. Then, we define an actor algebra over actor terms as a basic formalism for multiagent systems. Finally, we show how it is possible to translate a speech act based language (a subset of KQML) into the given algebra. This allows to prove the equivalence of multiagent systems in terms of the equivalence of the corresponding actor terms.

keywords: Distributed AI, Multiagent Systems, Interaction, Concurrency, Speech Acts.

1 Introduction

Over the past few years there has been quite a lot of interest in *multi-agent systems* [20, 9, 18]. They are the subject of a stream of research within distributed AI, studying systems made up of multiple heterogeneous intelligent agents where competition or cooperation is possible between them [18]. This research area differs from *distributed problem solving* because there is no common global goal to be solved which is known at design time; on the contrary, it is assumed that programs of intelligent agents are written by different people, in different languages, at different times for different purposes.

A central issue is the design of a language to build such systems. This includes the design of both a *knowledge representation* language (to specify the internal behaviour of agents) and a *communication* language (to specify the interacts among agents). Recently, two languages have been proposed for multi-agent systems, which are both based on the *speech act theory* [19], a formal model of human communication developed by philosophers and linguists. The first one is the Agent-Oriented Programming (AOP) framework, proposed by Shoham [20]. AOP specializes object oriented programming by defining the state of an agent, and restricting the kinds of messages which agents can send or receive. The second language is Agent Communication Language (ACL) [9], provided in the context of the ARPA Knowledge Sharing Effort. ACL includes two main components: (i) a representation language for the contents of messages (called Knowledge Interchange Format – KIF), which is basically an extension of first-order logic; and (ii) a communication language KQML (Knowledge Query and Manipulation Language) [12], which consists of a set of communication primitives - called *performatives* - which aim to support cooperation among agents in distributed applications. The KQML performatives enable agents to exchange and request knowledge, and to cooperate during problem solving.

Both AOP and KQML are languages similar to those in the family of so-called *coordination languages* [4]. These extend sequential languages with constructs to support concurrency and coordination. In a similar way, AOP and KQML extend knowledge representation formalisms with knowledge communication primitives, and focus on defining *knowledge-level coordination languages*, which can be used to specify a range of cooperation strategies. Knowledge-level coordination languages are situated at a higher level of abstraction with respect "normal" coordination languages of distributed computing, as they support coordination not at the symbol-level but at the knowledge-level [16]. A previous paper by one of the authors [8] represents a preliminary attempt towards a formal analysis of the interaction problems in this class of languages. In particular, a formal operational semantics for a subset of KQML is provided, which helped in showing relations between knowledge-level communication primitives based on speech acts and traditional concurrent formalisms known in distributed computing. In the same way as CCS [13] defines the basic communication mechanisms for synchronous message passing languages in distributed computing, we would like to capture a minimal set of primitives for knowledge-level coordination languages. Indeed, as Robin Milner stated in his Turing award lecture "a theory of concurrency and interaction requires a new conceptual framework, not just a refinement of what we find natural for sequential computing" [15].

In this paper, we present a first step towards a rigorous approach to the semantics of speech act based languages. First, we investigate the basic communication mechanisms of agent communication languages which, we claim, are close to those of the actor model [1]. Then, we define an algebra of actors (SAA, for short), which is assumed static (i.e., without dynamic creation) for the sake of simplicity. The actor algebra we are proposing includes many basic communication and synchronization primitives which, apparently, are enough to specify speech act based languages. SAA is equipped with a structured operational semantics and with a behavioural equivalence which equates actor systems whenever they are to be considered indistingushable by external observers. Finally, we show how a subset of KQML can be traslated into SAA. In this way we can import in KQML the theory defined for SAA. For instance, two KQML programs can be defined equivalent if so are their SAA translations. In this way, we provide a rigorous formal base to reason about this language.

We think that the contents of this paper is not only relevant for the AI community, indeed the actor algebra can be considered as one (maybe the most) promising attempt in order to fully understand the basic mechanisms of concurrent object oriented programming, providing an alternative to other approaches [3], [15], [21] and to the current work on the foundation of actors [2]. This extended abstract will necessarily concentrate mainly on the AI aspects of our research, thus relations with other work in the field of semantics of concurrency is only scketched. This will be reported in the forthcoming paper [5].

The paper is organized as follows. In Section 2 we analyse a subset of KQML, trying to understand which are the basic communication mechanisms which are needed for a knowledge-level coordination language. In Section 3 we introduce our actor algebra with its operational semantics. In Section 4 we show how a subset of KQML can be translated into the actor algebra. Finally, we add some concluding remarks.

2 Agents and Interaction Mechanisms

In this section we analyze multi-agent systems and their communication protocols based on speech act theory, in order to single out a set of basic interaction primitives.

The multi-agent systems we are considering are systems composed of a set of intelligent software agents, communicating by means of knowledge-level coordination primitives defined according to the speech act theory. That theory, much broader than necessary to our aims, is a general framework to model human communication, which partly influenced the theory of multi-agent systems. The kinds of messages that can be used are rather different. Messages can be classified according to their type (also called *performative*). We introduce here a variation of a classification presented in [12] which distinguishes three classes of performatives, including most of the performatives defined in knowledge-level coordination languages. A message can be:

1. *assertive*, when it states a fact;

2. *directive*, if it includes a command, a request or a suggestion;

3. *declarative*, when an agent sends to another agents an information on his internal capabilities.

Languages based on speech acts introduce a fixed structure for messages. For instance a KQML message is defined by the following abstract syntax:

$Performative(Recipient, Sender, Contents)$. A typical assertive primitive is tell(B,A,P) means that the agent A states to B that P is true. (Note that the notion of true is intendend here with respect to a particular agent in this case A believes P to be true.) A typical instance of a directive primitive is ask(B,A,P), meaning that agent A wants to know from B if P is true, while a typical instance of a declarative primitive is export(B,A,P), meaning that A informs B that he is able to answer queries on P.

2.1 Agent Identity

In multi-agent systems there is a notion of *agent-identity* which is similar to the notion of object-identity in object-oriented programming [11]. Identity is that property of an agent which distinguishes each agent from all others. Support for identity in multi-agent systems can be achieved by associating to each agent a unique name; this is the approach followed by KQML. The name of the agent is also used as basic dispatching mechanisms in message passing. This notion of agent-identity based on names has a natural mapping in the Actor model [1]. Actors are named objects with a behaviour which is a function of incoming communications. Each actor has a unique name (mail address) determined at the time of its creation. This name is used to specify the recipient of a message. Conversely, agent-identity is not easely embeddable in formalisms such as CCS [13] or π-calculus [14], where message dispatching is performed by the mean of channels. In these formalisms the association address-process is not unique a process may have several ports (channels) from which it receives messages and the same channel can be accessed by different processes.

2.2 Synchronous vs Asynchronous Message Passing

We distinguish here two versions of message passing: the synchronous version does not allow the buffering of messages, thus a message to be delivered needs a "rendez-vous" between the sending and the receiving agent, this is the mechanism adopted by CCS [13]; the asynchronous version assumes that the buffering of messages is possible, thus it is not required an agreement we the receiving agent to send a message, this is the approach adopted by actors [1] and by asynchronous process algebras [6].

Since our goal is to select a set of primitive interaction mechanisms, we need to specify which kind of message passing should be considered as primitive in this context. We claim that asynchronous message passing is the most suitable basic interaction mechanism for multi-agent systems. Firstly, because synchronous communication can be modelled providing adequate synchronization constraints, while, viceversa, if we want to model asynchronous message passing with a synchronous language, we need to introduce an extra agent to deal with the buffering of messages. Secondly, according with the thesis presented in [8], asynchronous message passing guarantees a higher-(knowledge)-level programming style.

2.3 Explicit vs Implicit Receive

Given that the basic message passing mechanism is asynchronous, and that the name of an agent is used to specify the recipient of a message, we have still to establish which is the form of receive primitive which is more suitable in multi-agent systems. The receive primitive may be blocking or non-blocking, and may also be explicit or implicit. A blocking receive primitive stops the execution of an agent until a message is available, while a non-blocking receive allows the agent to continue its execution. A receive operation is explicit when it appears in the program, while it is implicit when does not correspond to an operation in the programming language and it is performed implicitly at certain points of the computation: typically in actor systems the receive is implicit and is performed only when the actor is idle, in this case the implicit receive can be only blocking since there is no computation to carry on when the actor is idle.

Although, some of the performatives which commonly appears in speech acts languages, such as those of KQML, sometimes include an implicit receive primitive, the semantics of all the performatives assumes that a certain message is transmitted from a sender agent to a recipient agent and there are no receive performatives. For instance, when an agent send an "ask" message to another agent, it is assumed also that it starts waiting for the answer, thus, it performs an implicit receive operation. In any case, the "ask" message must be transmitted to the other agent.

For this reason, we think that the receive should be implicit in a basic calculus for multi-agent systems. By the way, note that asynchronous message passing with an implicit receive mechanism is a basic feature of the actor model.

2.4 Staticity vs Dynamicity

An aspect which is not central in the theory of speech act is dynamicity, i.e., the possibility of creating new agents at run time. As an example, KQML does not include performatives to create new agents. Although one may obiect that any agent which "speaks" KQML will be able to communicate with other agents (KQML is an open system), there is still no way to control how these new created agents will coperate with the pre-existing ones (for instance, what agent names are known by the new agent).

On the other hand, this issue is central in the current work on the theory of distributed computing [15] and in particular in the recent approaches to the formal description of concurrent object oriented programming languages [3, 2, 21]. This may sound quite disappointing: after all, the actor model was introduced by Carl Hewitt about 20 years ago [10] and for a long time it has been the unique model of concurrent computations able to deal with dynamic aspects, (π-calculus was designed about 10 years after) while most of the current work on multi-agent systems does not consider dynamicity as a key issue.

This is exactly the opposite of what it is happening now in distributed computing where dynamic aspects of the languages are taking a central role. As applications running on wide area networks require the dynamic activation of software agents on different sites, these new agents must be able to cooperate with the other heterogeneous pre-existing agents, and new proposals for new languages try to handle these issues, as Obliq [3] does by exploiting cloning, distributed lexically scoped variables and delegation.

For the limited aims of the present paper, however, we restrict our attention to static systems only. Nonetheless, as we believe that dynamic features of knoledge-level coordination languages should be taken into account, we also developed a dynamic actor algebra with the usual "create" operator of actors, reported in [5].

2.5 Discussion

In summary the actor model captures all the basic interaction mechanisms we have pointed out above providing: support for agent identity, asynchronous message passing, an implicit receive mechanism and support for dynamicity, thus we choose the actor model as a basic model for multi-agent systems. However the mapping between multi-agent systems and actors is not immediate; the main difference are the following: actors does not allow non-terminating programs to be executed when answering a message; the order of messages is not relevant in an actor-based system and, finally, actors do not provide symmetric primitives such as ask and tell.

We make the assumption that our set of agents is statically defined and allocated on a set of processors, thus we consider a static variant of the actor model where primitives to create actors are not allowed in programs. We define an associated actor algebra and we study the mapping of a subset of the KQML performatives into this algebra.

3 An Algebra of Actors

In this section we introduce an algebra of actors as a basic formalism to model multi-agent systems. The innovative aspect of the algebra we are proposing is that it is not defined over actor configurations (including also incoming messages as in [2]), but over actors only. This allows to reason on a particular actor system independently of the context, i.e., the multiset of incoming messages.

In the actor model the behaviour of an object is a function of incoming communications. Actors are self-contained agents that communicate by asynchronous message passing making use of three basic primitives: *create*, to create new actors; *send*, to send messages to other actors; and *become*, to change the behaviour of an actor.

3.1 A Static Actor Algebra

A Static Actor Algebra (SAA) is an algebra which models a *static* actor system, i.e., composed of a fixed set of actors where create is not a legal primitive and where actor names (except the name of the sender) are not allowed in messages.

Let A_{actors} be a finite set of actors names and let M_{actors} be a set of expressions representing the possible contents of a message, an actor term is defined according to the following syntax:

ActorTerm ::= Actor | *Actor* || *ActorTerm*
Actor ::= Idle_Actor | *Busy_Actor*
Idle_Actor ::= <*Name, Behaviour*>
Busy_Actor ::= <*Name, Program, Behaviour*>
Behaviour ::= C(x,y,z) where $C(x,y,z) \stackrel{def}{=}$ *Program*
Program ::= E_1:$Program_1$+E_2:$Program_2$+...+E_n:$Program_n$ |
become(*Behaviour***).***Prog* |
send(*Message***).***Program* | $\surd$
Prog ::= **send(***Message***).***Prog* | $\surd$
Message ::= [Recipient,Sender,Contents]
Recipient ::= Name
Sender ::= Name
Name ::= $a \in A$
Contents ::= $m \in M_{actors}$

where $E_1 : Program_1 + E_2 : Program_2 + \ldots + E_n : Program_n$ is a guarded command and $E_1, E_2, \ldots, E_n$ are conditions on the state of the actor and/or the incoming message. The constants defining actor programs take three arguments which represent respectively: the sender of the incoming message, the contents of the message and the state of the actor (when arguments are not significant they will be omitted).

Definition 3.1 - Actor Algebra. *An actor algebra is a triple* (A, K, ρ) *where* $A \subseteq A_{actors}$*, K is a set of Behaviour, and* ρ *is a funcion that maps actor names into their initial Behaviour.*

3.2 Operational Semantics

We model the operational semantics of our actor algebra by means of labelled transitions systems following the structural operational semantics approach (SOS) of Plotkin [17].

A transition system modelling an actor algebra is represented by a triple $(ActorTerms, T, \{\stackrel{\alpha}{\to}: \alpha \in T\}$ where $ActorTerms = \{a : aisanActorTerm\}$, T is a set of labels and $\stackrel{\alpha}{\to}$ is the minimal transition relation satisfying the following axioms and rules.

send1) $< a, send([x, self, m]).P, C(x,y) > \stackrel{send(x,m)}{\longrightarrow} < a, P, C(x,y) >$

send2) $< a, send([b, self, m]).P, C > \stackrel{send(b,m)}{\longrightarrow} < a, P, C >$

bec) $< a, become(C').P, C > \stackrel{become(a)}{\longrightarrow} < -, P, C > \| < a, C' >$

rec) $< a, C(x,y) > \stackrel{receive(a,x,y)}{\longrightarrow} < a, P, C(x,y) >$ where C(x,y) $\stackrel{def}{=}$ P

par1) $\dfrac{A \stackrel{\alpha}{\longrightarrow} A'}{A \| B \stackrel{\alpha}{\longrightarrow} A' \| B}$ par2) $\dfrac{B \stackrel{\alpha}{\longrightarrow} B'}{A \| B \stackrel{\alpha}{\longrightarrow} A \| B'}$

sum) $\dfrac{E_i = true}{< a, E_1 : P_1 + E_2 : P_2 + \ldots + E_n : P_n, C > \stackrel{\tau}{\longrightarrow} < a, P_i, C >}$

idle) $< a, \surd, C > \stackrel{\surd}{\longrightarrow} < a, C >$

end) $< -, \surd, C > \stackrel{\surd}{\longrightarrow}$

Two rules are needed to model the execution of the send primitive *send*1, *send*2. We must be able to distinguish between a recipient which is explicitely specified (b) from a recipient thet will be bound to the sender of an incoming message (x). The rule *bec* modifies the behaviour of an actor, a new actor with the same name is created and the rest of the computation is executed by an actor which is no more reachable (it will be destroied by the *end* rule). The implicit receive is performed only when an actor is idle, (i.e., when an actor ends its computation, see the rule *idle*). The name of the sender and the contents of the message are recorded in the label. This allows us to describe correctly actor computations when the multiset is added to configurations: an actor configuration is a pair (A,MS) where A is an actor term and MS is the multiset of incoming messages. The following rule models the implicit receive in this context:

$$\frac{A \stackrel{receive(a,x,y)}{\longrightarrow} A'}{(A, \{[a,b,m], MS\}) \longrightarrow (A'', MS)} \text{ where } A'' = A'[x/b, y/m]$$

The transition systems on actors allow us to define when two actor terms are equivalent independently from the multiset of incoming messages. This is obtained by exploiting a weak notion of bisimulation [13] which intuitively does not depends on the order in which communication actions are performed [5]. This is natural for actors since there is arrival-order non-determinism in message delivery. The given equivalence has also be proved to be a congruence with respect to the parallel composition operator $\|$. Thus, if two actor systems are proved to be equivalent also their compositions with any other composable actor system are equivalent [5].

4 Translating KQML into SAA

The set of performatives used in KQML is large, as different communication protocols are supported, thus it is not possible to provide a formal account of the whole system in this paper. Therefore we will only consider a subset of the KQML performatives (insert, ask-if, and tell).

We assume to have given a fixed set of agents A_{KQML} which communicate by means of the above subset of KQML. Each agent is composed of a name, a set of programs and a *virtual knowledge base* (VKB).

Agents react to messages received from other agents and from the outside world. We model only communication actions and we assume that each communication action has associated a contents in a given knowledge representation formalism, we denote with M_{KQML} the set of all the possible contents and with M,M',... its elements.

The messages that can be received from an agent named a in our subset of KQML are: $insert(a, b, M)$, the agent a is asked to insert the proposition M into its VKB; $ask(a, b, M)$, the agent b wants to know if M is in the VKB of a; and $tell(a, b, M)$, the sentence M is in the VKB of b. Each agent has an associated *handler function* [12] [1] which maps the received message into a KQML program which executes it. We assume that all the program are associated to constant by means of definig equantions ($C \stackrel{def}{=} KQMLProgram$) and that the set of the constants is denoted by K_{KQML}.

Programs specify the way in which agents deal with messages. The syntax of programs is shown below.

$$
\begin{aligned}
&\mathit{KQMLProgram} ::= \mathit{CommAction.KQMLProgram} \mid \surd \mid C \\
&\mathit{CommAction} ::= \mathit{assert(Recipient,Sender,Contents)} \mid \\
&\qquad\qquad\qquad \mathit{tell(Recipient,Sender,Contents)} \mid \\
&\qquad\qquad\qquad \mathit{ask(Recipient,Sender,Contents)} \\
&\mathit{Sender} ::= a \in A_{KQML} \\
&\mathit{Receiver} ::= a \in A_{KQML} \\
&\mathit{Contents} ::= M \in M_{KQML}
\end{aligned}
$$

The symbol $\surd$ represents a terminating internal program which does not involve communication actions. A constant C is an agent program whose meaning is given by a defining equation $C \stackrel{def}{=} Program$. This mechanism is useful as it allows us to define agents with infinite behaviour [13] by providing recursive definitions, such as $C = ask(a, b, M).C'$ and $C' = assert(a", d, M').C$.

The state of an agent can be: busy, the agent is executing a program; idle, the agent is idle; waiting, when the agent is waiting for an answer. There are no explicit receive primitive inside programs.

An operational semantics for a synchronous and an asynchronous variant of this subset of KQML has been presented in [8]. The main difference with the system we are modelling now is that we include also the contents of messages in the language.

In this paper we show how the semantics of the asynchronous variant of KQML can be expressed by means of translation into our actor algebra. The asynchronous KQML proposed in [8] makes use of an asynchronous transport level based on a tuple space abstraction and exploits a non-blocking ask primitive. Requests of knowledge do not block

[1] This function is called dispatch in [8]

the agent executing them; it can continue its current execution. This implies that agents receive answers to ask questions only when they are idle.

Definition 4.1 - *(KQML system) A KQML system is a triple* $(A_{KQML}, K_{KQML}, handler)$ *where handler is the set of handler functions.*

Let $Q, Q', \ldots$ be KQML programs and let ϕ a function that maps KQML a contents into M_{actors}. The translation of a KQML system into the actor algebra is defined as follows:

1. each KQML agent is translated into an actor with the same name, thus $A = A_{KQML}$.
2. each actor a has a initial behavior $C_a = E_1 : P_1 + E_2 : P_2 + \ldots + E_{n_a} : P_{n_a}$ encoding the *handler* function where the P_i are actor programs defined below, thus $\rho(a) = C_a$
3. each KQML program is translated by means of a function Φ which takes as an argument an agent name and a KQML program and returns a set of *Behaviour*. Φ is defined recursively on the struture of programs.

The translation function is defined as follows:

$$\Phi(b, ask-if(a,b,M).Q) = send([a,b,(ask,\phi(M))]), become(new(b)), send([self,self,go]), \surd$$
$$\Phi(b, tell(a,b,M).Q) = send([a,b,(tell,\phi(M))]), become(new(b)), send([self,self,go])), \surd$$
$$\Phi(b, assert(a,b,M).Q) = send([a,b,(assert,\phi(M))]), become(new(b)), send([self,self,go])), \surd$$
$$\Phi(b, C) = become(\Phi(b, C) \; where \; C \stackrel{def}{=} \Phi(b,Q) \; if \; C \stackrel{def}{=} Q$$
$$\Phi(b, \surd) = become(C_b(x,y,z)), send(self, self, go), \surd$$

where $new(b)$ returns a fresch constant symbol $C'_a(u,v,z)$ associated to the actor b such that $C'_a(u,v,z) \stackrel{def}{=} \Phi(b,Q)$. The translation presented does not aim to be complete and and it has been only introduced to illustrate the technique. It deals mainly with the handling of possible infinite behaviour of agents, which are achieved in the actor algebra exploiting the become primitive. On the other hand, we do not model here a protocol to guarantee an ordered message delivery. However, different translation shema based on different protocols are possible.

5 Conclusion

We have presented a study on the interaction properties of multi agent systems with the aim to find out a set of basic mechanisms to model interation. We have defined an algebra of actors that captures some of the highlighted basic interaction primitives which is the basis of a semantics by translation modelling the interaction feautures of KQML. Our semantics approach is complementary to other approaches to the semantics of KQML, such that the one presented in [12], which is based on the description of the cognitive states of the agents that exploit speech acts. In that report, in fact, the emphasis is not on operational description, rather on logical-based properties that preserved or transformed by program executions.

Possible arguments against this kind of research are that it may be considered relevant for the theory of concurrency but not for AI and that it is not necessary to study the language used for the transport level which can be a message passing language, Linda [4], or other, because the mapping between speech acts messages and the transport level is trivial: each performative may be associated to a message at the transport level.

Our position with respect to these criticism is that the trivial mapping between the performatives and the transport level does not captures the interaction properties of performatives and the consequent causal relations which arises among messages. As an example, if we consider two performatives which belong to the class of directives a request such as "ask" and a command such as "achieve" [7], we see they have different interaction behaviour. We expect there will be an answer to the query, while no answer may be required from a command. Thus, doing an "ask" does not mean only to send an ask message, the sending agent must also change its internal state or behaviour in order to take into account that it is waiting for an answer. Moreover, there is a causal relationship between the query and the answer, whenever a query is performed we espect that an answer will be given.

In summary, since speech act messages have associated interaction properties, i.e., they may enforce causal relationship among low level messages, a careful analisys of them is needed, if we want to provide a rigorous model for knowledge-level coordination languages.

Acknowledgements

This paper has been partially supported by the Italian Ministry of University (MURST) and by the ESPRIT BRA Project 9102- Coordination.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, Cambridge, MA, 1986.

[2] G. Agha, I. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computation. In *Proc. of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 564–579. Springer Verlag, 1992.

[3] L. Cardelli. A language with distributed scope. Technical report, Digital Equipment Corporation, June 1994.

[4] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[5] A. Dalmonte, M. Gaspari, and R. Gorrieri. A algebra of actors. Technical report, Comp. Science Laboratory, Università di Bologna, Italy, January 1995.

[6] F. DeBoer, J. Klop, and C. Palamidessi. Asynchronous Communication in Process Algebra. In *Proc.* 7^{th} *annual IEEE Symposium on Logic in Computer Science*, Santa Crux, California, 1992.

[7] T. Finin, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzson, D. McKay, J. McGuire, R. Pevalin, and S. Shapiro. Specification of the kqml agent communication language. Technical report, The ARPA Knowledge Sharing Initiative External Interface Working Group, June 1993.

[8] M. Gaspari and E. Motta. Symbol-level Requirements for Agent-level Programming. In A. Cohn, editor, *ECAI94 the 11th European Conference on Artificial Intelligence*, pages 364–368. John Wiley and SonsLtd, 1994.

[9] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):49–53, July 1994.

[10] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[11] S. Khoshafian and G. Copeland. Object Identity. In *Proc. OOPSLA '86*, pages 406–416, September 1986.

[12] Y. Labrou and T. Finin. A semantics approach for KQML – a general purpose communication language for software agents. In *Proc. the Third International Conference on Information and Knowledge Management (CIKM'94)*, 1994.

[13] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[14] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[15] R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):79–89, January 1993.

[16] A. Newell. Reflections on the Knowledge Level. *Artificial Intelligence*, 59:31–38, 1993.

[17] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981.

[18] J. Rosenschein and G. Zlotkin. Designing Concensions for Automated Negotiation. *The AI Magazine*, pages 29–46, Fall 1994.

[19] J. Searle. *Speech Acts.* Cambridge Univ. Press, 1969.

[20] Y. Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60:51–92, 1993.

[21] V. Vasconcelos. Typed Concurrent Objects. In *8th European Conference on Object Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1994.