

# JIM

## A Java Interagent for Multi-Agent Systems

Francisco J. Martin, Enric Plaza, Juan A. Rodríguez-Aguilar,  
and Jordi Sabater\*

IIIA - Artificial Intelligence Research Institute  
CSIC - Spanish Council for Scientific Research  
Campus UAB, 08193 Bellaterra, Barcelona, Spain  
Vox: +34-3-5809570, Fax: +34-3-5809661  
{martin,enric,jar,jsabater}@iia.csic.es

### Abstract

In this paper we introduce an *interagent* as an autonomous software agent which manages (intermediates) the communication and coordination between an agent and the agent society wherein this is situated. According to our proposal, interagents shall constitute the sole and exclusive means through which agents within a multi-agent scenario interact. With this aim, we have developed JIM, a general-purpose interagent that provides agents with a highly versatile range of programmable – before and during the agent’s run-time – communication and coordination services. The development of JIM lies in the framework of the SMASH project. SMASH addresses the construction of multi-agent systems to tackle complex problems of distributed nature in hospital environments. Two main benefits stem from the usage of JIM: on the one hand, it permits agents to reason about both communication and coordination at a higher level of abstraction, whereas on the other hand, it provides a complete set of facilities that allows agent engineers to concentrate on the design of their agents’ inner and social behaviour.

## 1 Introduction

There exists a number of problems which involve multiple sources of knowledge and, thereby, can best be addressed using a multi-agent system – a computational system composed of several interacting agents which cooperate<sup>1</sup> with one another to solve complex tasks. Furthermore, the deployment of multi-agent systems permits to benefit from a number of advantages – such as parallelism, robustness or scalability – that a single agent working isolatedly can not offer itself.

---

\*AAAI-98 Workshop on Software Tools for Developing Agents, Madison, Wisconsin, July 1998. This work has been supported by the Spanish CICYT project SMASH, TIC96-1038-C04001 and the DGR-CIRIT doctoral scholarships FI-PG/96-8490 and FI-DT/96-8472.

<sup>1</sup>“*Cooperation* with other agents is paramount: it is the *raison d’être* for having multiple agents in the first place in contrast to having just one” H. S. Nwana [15].

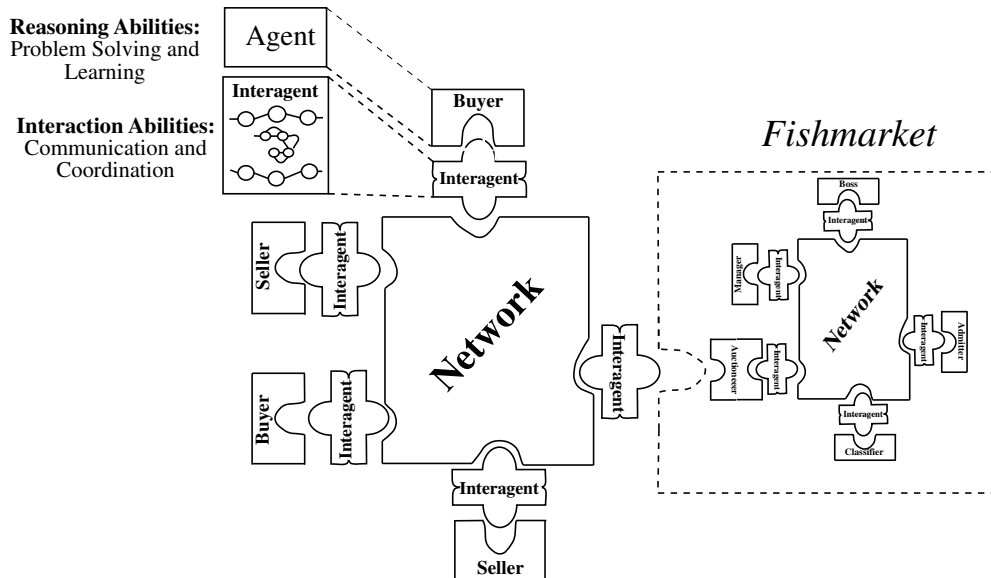


Figure 1: Fishmarket: A multi-agent system using interagents

Currently, we are partners of the SMASH project[2], a collective, joint effort involving several research institutions that addresses the construction of a general-purpose heterogeneous rational multi-agent architecture, and the development –on a computational implementation of this architecture– of prototype multi-agent systems with learning capabilities that cooperate in the solution of complex problems in hospital environments. The development of such multi-agent systems poses the question of how to integrate a set of heterogeneous agents – agents developed by different people for different purposes and in different languages – within a common setting. In order to achieve this goal, two major issues need to be addressed: (highly flexible) communication and coordination among the agents composing the multi-agent system. Instead of letting agents deal themselves with such issues, our proposal opts for introducing an autonomous software agent that we call *interagent* which manages (intermediates) the communication and coordination between the agent it is attached to (its *owner*) and the agent society wherein the owner is situated (see Figure 1). According to our proposal, interagents shall constitute the sole and exclusive means through which agents within a multi-agent scenario interact.

We are developing JIM, a general-purpose interagent that provides agents with a highly versatile range of programmable – before and during the agent’s run-time – communication and coordination services. JIM is being implemented in Java in order to ensure platform independence.

The remainder of this paper is organized as follows. Section 2 analyzes what features distinguish interagents from related approaches. Section 3 describes the communication services offered by interagents. In Section 4 the coordination services supported by interagents are presented. In section 5 we illustrate how JIM is being used in different agent-based applications. Finally, Section 6 summarizes some concluding remarks.

## 2 Related Work

For several years, agent-based software engineering has faced the matter of enabling heterogenous programs written by different people, at different times, in different languages and with different interfaces to communicate and interoperate [9]. Researchers in the ARPA Knowledge Sharing Effort have proposed agent communication languages (ACLs) as the means to allow the exchange of knowledge among software agents in order to facilitate their interoperation [9]. Generally, an ACL is composed of three main elements: an open-ended vocabulary appropriate to a common application area, an inner language (KIF–Knowledge Interchange Format) to encode the information content communicated among agents, and an outer language (KQML–Knowledge Query and Manipulation Language) to express the intentions of agents [13].

Nowadays, KQML has become the communication language par excellence in agent-based systems. However, when several computational entities interact by exchanging messages a higher level of interaction concerned with the conventions that they share during the exchange should be addressed [6]. This level of interaction is not supported by KQML, whereas coordination languages –like COOL[6] – allow such conventions to be explicitly expressed. Making shared conventions explicit allows interdependencies among agents’ activities to be managed.

Apart from COOL, there exist other agent building tools such as Agent-Talk[1] or JAFMAS [7] which also provide coordination constructs and many more agent building tools that have not addressed this issue yet<sup>2</sup>. For instance, JAFMAS provides a generic methodology for developing speech-act based multi-agent systems using coordination constructs similar to COOL.

Interagents –likewise KQML facilitators[16]– are inspired by the *efficient secretary* metaphor already introduced in the Actors model of concurrent computation [3]. Nevertheless, *interagents* (unlike KQML facilitators) offer the coordination level required by agents to cooperate in non-trivial ways. On the other hand, unlike KQML facilitators interagents have no knowledge about the reasoning capabilities of their owners [21], though they are aware about their owners’ plans thanks to conversation protocols.

By introducing interagents, and concretely JIM, we try to make headway with respect to other agent building tools offering a programmable communication and coordination module whose behaviour can be specified before or during its owner’s run-time by both the agent engineer and the (owner) agent itself. In addition, another major advantage of using interagents is that they permit agents to reason about communication and coordination at a higher level of abstraction, making implementation details transparent to their owners.

Finally, notice that in some sense interagents extend to agents the concept of synchronizers presented in [8] to coordinate distributed objects.

---

<sup>2</sup>For an account of other agent building tools refer to <http://www.ececs.uc.edu/~mnoschan/tools.html>.

### 3 Communication Services

In our proposal, communications among agents are based on *message-passing*<sup>3</sup>. However, agents do not exchange messages directly but by means of interagents. Thus, an interagent is informed by its owner about the message to be sent and its addressee, and then the interagent carries out all the operations needed to deliver it correctly.

An interagent and its owner can communicate in two ways:

- a) through (TCP-)stream-sockets —in case that an interagent and its owner are two distinct computational processes (residing in the same computer or not)
- b) through shared memory —in case that an interagent and its owner are two distinct threads residing in the same process space

Concerning communication, an interagent provides its owner with the following communication services based upon TCP/IP:

- queueing of outgoing messages from its owner and queueing of incoming messages from (the interagents of) other agents;
- *asynchronous communication* between agents;
- *synchronous communication* between agents (implemented on top of *buffered asynchronous communication* between interagents);
- agent naming services (white pages);
- handling of expired messages and automatic recovery of transmission errors.

Two communication protocols have been devised (agent-to-interagent and interagent-to-interagent<sup>4</sup>) whose communication language is based on KQML. Therefore, both agent-to-interagent messages and interagent-to-interagent messages are expressed as KQML performatives. However, at present interagents support a subset of KQML performatives, whose syntax has been extended with the reserved parameter keywords shown in Table 1:

- Each performative exchanged between agents is associated to a particular conversation specified by the `:conversation` parameter.
- The `:delay` parameter indicates how long a message queued in an interagent is postponed before this starts to process it. In this way, an agent can tell its interagent to deliver a given performative after a certain amount of time.
- The `:time-out` parameter indicates the maximum period of time that an agent conforms to await for receiving a reply to the performative.
- The `:time-to-live` parameter indicates the life time of the performative once it has been queued in an interagent. Thus, when this time expires, the message is thrown away by the interagent where it is queued, and the sender receives back an error message.

**`:content`** when *response* to a *response* to the of the term definitions) performative

---

<sup>3</sup>The message-passing paradigm provides functionality equivalent to that found in *remote procedure call* (RPC) or *tuple-space* paradigms [10].

<sup>4</sup>Notice that communication between interagents is transparent to their owners.

Table 1: Reserved parameter keywords and their meanings introduced by JIM

Keyword	Meaning
:conversation	Identifier of the conversation wherein the performative is uttered
:delay	Indicates how long a message must be delayed by an interagent before it starts to deliver it to the addressee.
:time-out	Maximum period of time an agent accepts to wait for receiving a reply to the performative.
:time-to-live	Life time of the performative after being queued in an interagent.

## 4 Coordination Services

Interagents offer the coordination level required by agents to cooperate in non-trivial ways. An interagent allows interdependencies between agents' communicative acts to be ordered. These interdependencies can be defined declaratively inside each interagent by means of a *conversation protocol*. A conversation protocol represents the conventions adopted by agents when interacting through the exchange of messages. A conversation protocol can also be seen as an agent's plan to achieve some goal[6]. We model and implement conversation protocols as a special type of pushdown automaton. Pushdown automata, unlike finite state machines, allow context to be stored and to be subsequently retrieved for an ongoing conversation.

Conceptually, we have decomposed a conversation protocol into (see Figure 2):

- A finite state control. Each state in the finite state control represents the situation of the interagent's owner during an ongoing conversation.
- An input list is continuously traversed in search of a performative which can produce a transition in the finite state control. If such message is found, it is dispatched and, thereby, removed from the input list<sup>5</sup>. Note that the way of traversing the input list differs from the one employed by classic pushdown automata whose read only input tapes is traversed from left to right (or the other way around).
- A pushdown list where the context of an specific conversation can be stored and subsequently retrieved.
- A finite set of transitions. Each transition in a conversation protocol indicates:
  1. what message has to be either sent or received to produce a move in the finite state control; and
  2. whether it is necessary to store (push) or retrieve (pop) the context using the pushdown list.

---

<sup>5</sup>Conversation protocols lack *e*-moves, what makes a significant difference with respect to classic pushdown automata.

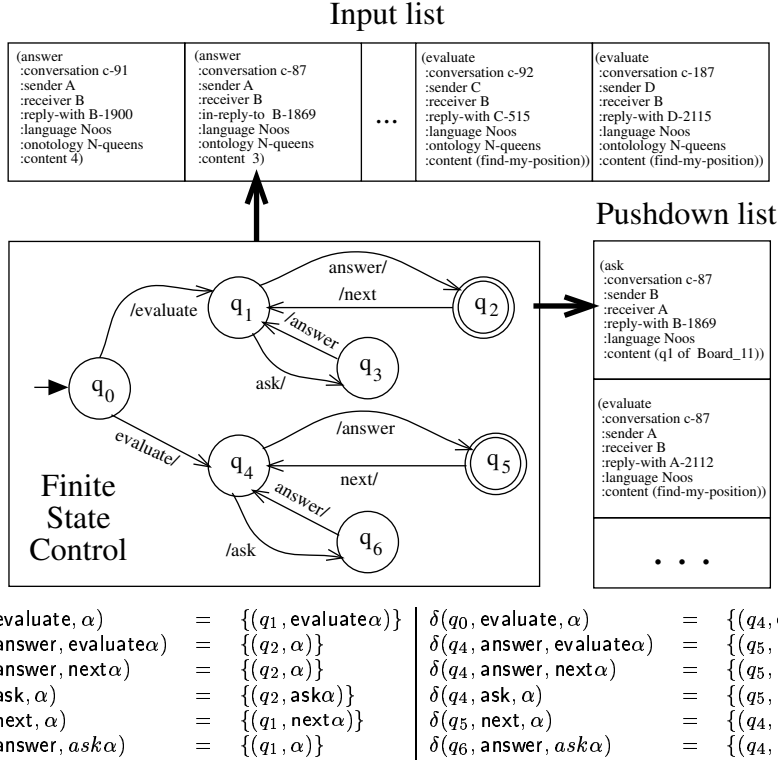


Figure 2: Conversation protocol for the foreign evaluation capability of Plural [11]. The conversation protocol followed by Plural agents during a foreign evaluation is modelled and implemented in an interagent as a pushdown automaton  $P$  such that:  $P = \langle \{q_0, q_1, q_2, q_3, q_4, q_5\}, \{\text{evaluate}, \text{ask}, \text{answer}\}, \{Z, \text{evaluate}, \text{ask}, \text{answer}\}, \delta, q_0, Z, \{q_2, q_5, q_6\} \rangle$ . Messages followed by / stand for performatives sent by the interagent's owner, whilst messages preceded by / stand for performatives received by the interagent's owner.

Formally, a conversation protocol, like a pushdown automaton [4], is a 7-tuple :

$$P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$$

where

- $Q$  is a finite set of state symbols that represent the states of the finite state control;
- $\Sigma$  is the finite input list alphabet composed of all possible performatives that an interagent can deal with;
- $\Gamma$  is the finite pushdown list alphabet composed of all possible performatives that an interagent can store;
- $\delta$  is a mapping from  $Q \times \Sigma \times \Gamma$  to the finite subsets of  $Q \times \Gamma^*$  which indicates all possible transitions that can take place during a conversation.
- $q_0 \in Q$  is the initial state of a conversation.

- $Z_0 \in \Gamma$  is the start symbol of the pushdown list.
- $F \subseteq Q$  is the set of final states representing possible final states of a conversation.

Messages queued by *interagents* can queue-jump only if they produce a transition in the conversation protocol. Therefore, in some sense can it be said that interagents constrain what an agent can utter and hear, and when. For instance, Figure 2 shows instantiation *c-87* of a conversation held by agents *A* and *B*. The following transition:

$$\delta(q_6, \text{answer}, \text{ask}\alpha) = \{(q_4, \alpha)\}$$

indicates in such conversation protocol that when:

- such conversation is in state  $q_6$ ; and
- the performative corresponding to the topmost message on the pushdown list is `ask`; and
- a message with the performative `answer` is in the input list, and its sender, receiver and the label in keyword `:in-reply-to` match respectively the receiver, sender and label in keyword `:reply-with` of the topmost message on the pushdown list.

then the following move

$$\delta(q_6, \text{answer}, \text{ask}\alpha) \vdash (q_4, \alpha)$$

takes place in the finite state control. As a result, conversation *c-87* switches to state  $q_4$ , the corresponding message is extracted from the input list and forwarded to the corresponding addressee, and the message on the top of the pushdown list is popped out.

## 4.1 Conversation Protocol Definition

An interagent can support a wide range of conversation protocols that can be defined declaratively and stored into the conversation protocol library that each interagent has associated. Such library can be upgraded in two ways:

**Statically.** Before the interagent's owner run-time, as an item provided by the agent engineer.

**Dynamically.** An agent can interactively define new conversation protocols (or modify existing ones) at run-time using a *conversation protocol definition and manipulation language* based on the set of reserved coordination performatives in Table 2. These performatives allow to either fully define a new conversation protocol or modify stored conversation protocols by adding or deleting states or transitions.

Once defined, conversation protocols must become instantiated in order to be used for coordinating the interaction between agents.

This capability of allowing agents to define and modify themselves their conversation protocols at run-time happens to be an innovative feature of our proposal, distinguishing interagents from other approaches like COOL[6] or JAFMAS[7]. architecture.

Table 2: Reserved coordination performatives, for agent A and interagent I

Performative	Meaning
define-conversation	A defines in I a new conversation protocol
add-state	A adds a new state to a conversation protocol residing in I
delete-state	A deletes a state from a conversation protocol residing in I
add-transition	A adds a new transition to a conversation protocol residing in I
delete-transition protocol	A deletes an existing transition from a conversation residing in I

## 5 Applications

JIM our Java-based implementation of a general-purpose interagent, is being used as the communication and coordination support for the agents developed in the framework of the SMASH project. Concretely, in this section we describe Plural – an extension of the knowledge representation language Noos for developing agent-based systems with learning capabilities – and *Fishmarket* – an agent-mediated electronic marketplace<sup>6</sup>.

### 5.1 Plural Noos

Noos is an object-centered language based on feature terms [5]. Noos can furnish applications with the reasoning capabilities required by intelligent agents, nonetheless, applications developed in Noos lack communication abilities except for graphical user interface. Thereby, we are developing Plural, a seamless extension of Noos based on interagents which promotes Noos to an agent-oriented language [11]. On top of the basic mechanism offered by interagents Plural extends Noos with two new constructs with the same nature as the rest of Noos constructs: *defforeign* and *defmobile*. These constructs provide Noos with two new ways of constructing feature terms –*foreign refinements* and *mobile refinements*– which, in turn, allow an agent to remotely evaluate methods owned by other agent –*foreign evaluation*– and send methods to other agents to solve problems on its behalf –*mobile problem solving methods* [12].

In this way, in Plural agents do not communicate directly with one another, instead, they rely on interagents which offer a range of programmable communication and coordination facilities. Each agent has attached its own interagent which constitutes the sole and exclusive means through which a Plural agent interacts. An interagent gives a permanent identity to its owner and enforces the *conversation protocols* (defined for each construct) –thus establishing what messages can be forwarded, to whom and when.

The declarative fashion of the conversation protocols offered by *interagents* is what allow Plural to incorporate new capabilities which require that agents follow some convention in the exchange of messages. These conventions will be provided to *interagents* by means of conversation protocols. We have provided interagents

---

<sup>6</sup>In fact, an interagent has emerged as a generalization of the notion of *remote control* introduced during the implementation of this system.

with a conversation protocol for each new Plural construct incorporated at the level of the knowledge representation language. For instance, Figure 2 shows the conversation protocol for the foreign evaluation capability of Plural. Interagents allow all the underlying exchange of messages needed by those constructs to be transparent to Plural agents.

The new capabilities embedded into Plural enable agents to adequately communicate and coordinate in order to exchange knowledge. Plural can be thought as an extension of a knowledge representation language with both an agent communication language and a agent coordination language which are implicitly provided at the knowledge representation level by means of some constructs and thanks to interagents. These constructs allow the exchange of knowledge to be performed at the knowledge representation level transparently and independently to the agent communication and coordination languages chosen.

The capabilities that now Plural incorporates are those currently under active research by new programming paradigms, namely distributed state, foreign method invocation, and remote evaluation. In order to provide Noos with such capabilities there was no need to re-write Noos. Thus, interagents shows a way in which legacy software can also profit from mobile code paradigms.

The foreign evaluation and mobile problem solving methods capabilities of Plural have been used to devised two cooperation modes among agents with learning capabilities –*Distributed Case-based Reasoning* (DistCBR) and *Collective Case-based Reasoning* (ColCBR) [17]. These modes of cooperation are based on reusing the experience acquired by other agents. Which agent owns the similarity-based reasoning method used to solve a problem –the *sender* of the problem or the *addressee* agent– is the basic difference between both methods [17]. In DistCBR an agent is delegated to solve a task on behalf of another agent. DistCBR is supported by the *foreign evaluation* capability of Plural. In ColCBR, an agent in addition to the task to be achieved sends the method to solve that task. ColCBR is supported by the *mobile methods* capability of Plural. Such cooperation modes are being used in CHROMA a distributed system for recommending a plan for the purification of proteins from tissues and cultures [17] and in CoDiT [18], a multi-agent system for therapy recommendation in diabetic patients in the framework of the SMASH project[2].

## 5.2 The *Fishmarket*

The fish market can be described as a place where several *scenes* run simultaneously, at different places, but with some causal continuity. The principal scene is the auction itself, in which buyers bid for boxes of fish that are presented by an auctioneer who calls prices in descending order – the *downward bidding protocol*. However, before those boxes of fish may be sold, fishermen have to deliver the fish to the fish market, at the *sellers' registration scene*, and buyers need to register for the market, at the *buyers' registration scene*. Likewise, once a box of fish is sold, the buyer should take it away by passing through a *buyers' settlements scene*, while sellers may collect their payments at the *sellers' settlements scene* once their lot has been sold.

In [20, 19, 14] we present the *Fishmarket*, our current implementation of an electronic auction house based on the traditional fish market metaphor, subsequently extended to become a multi-agent testbed[20]. This implementation allows to run auctions over the Internet that permit both human and software

agents to participate. Thus, buyer and seller agents can trade goods as long as they comply with the *Fishmarket institutional* conventions. Those conventions that affect buyers and sellers have been coded into their interagents, which constitute the sole and exclusive means through which each trading agent interacts with the market institution. An interagent enforces its owner (a trading agent) *conversation protocols* that establish what illocutions can be uttered by whom and when. Not only are interagents utilized to allow trading agents to interact with the market institution, but also to allow those agents working as market intermediaries to coordinate their activities. Figure 1 depicts a conceptual view of the *Fishmarket* system which differentiates trading interagents (attached to buyers and sellers) from market interagents (attached to market intermediaries). Both types of interagents communicate asynchronously through TCP-stream sockets with their owners, making use of the communication services detailed in section 3.

For the current version of the system both trading and market interagents instantiate the conversation protocols stored in their libraries of conversation protocols. Nonetheless we must recall from section 4.1 that the high flexibility of interagents would permit to dynamically reconfigure *Fishmarket* without changing the implementation. For instance, say that the boss of the market decides, during the system’s run-time, that the auctioneer employs new bidding protocols unknown by the trading interagents. In that case, the boss would dynamically define the conversation protocol required for the new bidding protocol in the trading interagents so that buyer agents were capable of bidding under the new auction rules.

## 6 Conclusions

An interagent provides an agent with the basic mechanisms to interact (communicate and coordinate) with other members of an agent society. Therefore, an interagent intermediates between its owner and the multi-agent environment as a whole. In this way, the overload related to the management of the communication and coordination tasks needed by an agent to live in a multi-agent system is shifted to its interagent, that relieves its owner from such a “tedious” work. Moreover, and more importantly, interagents allow to coordinate interdependencies between agents’ activities by means of highly flexible conversation protocols.

Two major benefits are gained from employing interagents. On the one hand, it permits agents to reason about both communication and coordination at a higher level of abstraction, whereas on the other hand it provides a complete set of facilities that allows agent engineers to concentrate on the design of their agents’ inner and social behaviour.

JIM is currently being used in two directions:

- to promote the knowledge representation language Noos to an agent-oriented language[11];
- to coordinate the activities of the market intermediaries composing the *Fishmarket* system[22, 20] and the interaction between the market as a whole and the participating buyers and sellers (see Figure 1).

## References

- [1] Agenttalk: Describing multi-agent coordination protocols. <http://www.cslab.tas.ntt.jp/at/>.
- [2] The smash project. <http://www.iiia.csic.es/Projects/smash/>.
- [3] G. Agha. *Actors, A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.
- [4] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume I: Parsing of *Series in Automatic Computation*. Prentice-Hall, 1972.
- [5] J. L. Arcos. *The Noos representation language*. PhD thesis, Universitat Politècnica de Catalunya, 1997.
- [6] M. Barbuceanu and M. S. Fox. Cool: A language for describing coordination in multi agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems*, 1995.
- [7] D. Chauhan. *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation*. PhD thesis, ECECS Department, University of Cincinnati, 1997.
- [8] S. Frolund. *Coordinating Distributed Objects*. The MIT Press, 1996.
- [9] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM, Special Issue on Intelligent Agents*, 37(7):48–53, July 1994.
- [10] M. Lejter and T. Dean. A framework for the development of multiagent architectures. *IEEE Expert*, 11(6):47–59, 1996.
- [11] F. J. Martin, E. Plaza, and J. L. Arcos. Interagents: Providing knowledge representation languages with agent-oriented capabilities. 1998. Submitted.
- [12] F. J. Martin, E. Plaza, and J. L. Arcos. Mobile problem solving methods in multi-agent systems. 1998. Submitted.
- [13] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of kqml as an agent communication language. In M. Wooldridge and J. Müller, editors, *Intelligent Agents II*, pages 347–360. Springer Verlag, 1996.
- [14] P. Noriega. *Agent-Mediated Auctions: The Fishmarket Metaphor*. PhD thesis, Universitat Autònoma de Barcelona, 1997. Also to appear in IIIA monography series.
- [15] H. S. Nwana. Software agents: an overview. *The Knowledge Engineering Review*, 11(3):205–244, 1996.
- [16] R. S. Patil, R. E. Fikes, P. F. Patel-Schneider, D. McKay, T. Finin, T. R. Gruber, and R. Neches. The darpa knowledge sharing effort: Progress report. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, 1992.

- [17] E. Plaza, J. L. Arcos, and F. Martín. Cooperative case-based reasoning. In G. Weiss, editor, *Distributed Artificial Intelligence Meets Machine Learning. Learning in Multi-agent Environments*, number 1221 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1997.
- [18] E. Plaza, J. L. Arcos, and F. J. Martin. Knowledge and experience reuse through communication among competent (peer) agents. 1998. Submitted.
- [19] J. Rodríguez-Aguilar, P. Noriega, C. Sierra, and J. Padget. Fm96.5 a java-based electronic auction house. In *proc. of PAAM'97*, pages 207–224, 1997.
- [20] J. A. Rodriguez-Aguilar, F. J. Martin, P. Noriega, P. Garcia, and C. Sierra. Competitive scenarios for heterogenous trading agents. In *Second International Conference on Autonomous Agents*, 1998.
- [21] N. Singh and M. Gisi. Coordinating distributed objects with declarative interfaces. In *Coordination Languages and Models*, number 1061 in Lecture Notes in Computer Science, pages 368–385. Springer, 1996.
- [22] T. F. Team. The fishmarket project. In [www.iia.csic.es/Projects/fishmarket](http://www.iia.csic.es/Projects/fishmarket).