An Infrastructure for Agent-Based Systems: an Interagent Approach

Francisco J. Martin, Enric Plaza and Juan A. Rodríguez-Aguilar

IIIA - Artificial Intelligence Research Institute

CSIC - Spanish Council for Scientific Research

Campus UAB, 08193 Bellaterra,Barcelona, Spain

Vox: +34-93-5809570, Fax: +34-93-5809661

{*martin,jar,enric*}*@iiia.csic.es*

Abstract

In this work we introduce an infrastructure for easing the construction of agent-based systems. We argue that such infrastructure constitutes a convenient solution for releasing agent developers from the cumbersome interaction issues inherent to any agent-based system, and therefore for helping them concentrate on the domain-dependent agents' logics issues (from the agents' inner behavior to the agents' social behavior). Our proposal relies upon two fundamental elements: *conversation protocols*, coordination patterns that impose a set of rules on the communicative acts uttered by the agents participating in a conversation (what can be said, to whom, and when), and *interagents*, autonomous software agents that mediate the interaction between each agent and the agent society wherein this is situated. Interagents employ conversation protocols for mediating conversations among agents. Thus, the management of conversation protocols is presented as *raison d'être* of interagents. We also introduce JIM, our java-based implementation of a general-purpose interagent. On the one hand, we show how it handles conversation protocols in practice, and on the other hand, how it has been endowed with the capability of dealing with agent interaction at different levels by means of the SHIP protocol. Finally, we illustrate the use of interagents by explaining the several roles that they play in FM, an agent-mediated electronic auction market.

Keywords: Agent-Based Systems, Interagents, Conversation Protocols, Interoperability

1 Introduction

Agent-based systems provide a way of conceptualizing sophisticated software applications that face problems involving multiple and (logically and often spatially) distributed sources of knowledge. In this way, they can be thought of as computational systems composed of several agents that interact with one another to solve complex tasks beyond the capabilities of an individual agent.

The development of agent-based systems can be regarded as a process composed of two well-defined, separate stages concerning respectively:

- *the agents' logics*: from the agents' inner behavior (knowledge representation, reasoning, learning, etc.) to the agents' social behavior responsible for high-level coordination tasks (the selection, ordering, and communication of the results of the agent activities so that an agent works effectively in a group setting[1, 2]).
- *the agents' interactions* taking place at several levels: *content level*, concerned with the information content communicated among agents; *intentional level*, expressing the intentions of agents' utterances, usually as performatives of an agent communication language (ACL), e.g. KQML, FIPA ACL, etc.; *conversational level*, concerned with the conventions shared between agents when exchanging utterances; *transport level*, concerned with mechanisms for

the transport of utterances; and *connection level*, contemplating network protocols (TCP/IP, HTTP, etc.).

Notice that nowadays a large amount of time during the development of agent-based systems is devoted to the management of the aforementioned time-consuming, complex agents' interactions. Since most of these appear to be domain-independent, it would be desirable to devise general solutions at this development stage that can be subsequently reused for the deployment of other multi-agent systems. In this manner, and very importantly, agent engineers could exclusively concentrate on the domain-dependent agents' logics issues. And for this purpose, they can largely benefit from their previous experiences in the application of Artificial Intelligence techniques.

In this work we propose an *infrastructure* that eases the construction of agent-based systems by taking charge of the cumbersome interaction issues inherent to this type of systems. Such infrastructure relies upon two fundamental elements: *conversation protocols*, coordination patterns that impose a set of rules on the communicative acts uttered by the agents participating in a conversation (what can be said, to whom, and when) , and *interagents*, autonomous software agents that mediate the interaction between each agent and the agent society wherein this is situated. Very importantly, interagents employ conversation protocols for mediating conversations among agents, and so the management of conversation protocols is in fact their *raison d'être*.

In our infrastructure interagents constitute the unique mean through which agents interact within a multi-agent scenario (see Figure 1). Thus, as a rule, agents' external behavior is managed by interagents, while their individual logic, knowledge, reasoning, learning, etc. are internal to agents.

We have materialized the conceptualization of interagents through the design and development of JIM: a java-based implementation of a general-purpose interagent capable of managing conversation protocols and capable also of dealing with agent interaction at different levels. JIM has been successfully applied in the development of FM, our current implementation of an agent-mediated electronic auction market.

The other sections of this article are organized as follows. Section 2 provides an overview of our infrastructure by explaining the basics of both conversation protocols and interagents, and introduces an example of multi-agent conversation extracted from FM, which in turn is explained in depth in Section 5. Next section 3 explains in depth conversation protocols. We start introducing their conceptual and formal models, followed by a further analysis concerning the properties that they must exhibit to ensure sound conversations. Furthermore, the way of negotiating, instantiating, and employing conversation protocols are also presented. Section 4 sketches out JIM along with the SHIP protocol that it employs to handle agent interaction at different levels. Section 5 illustrates the valuable use of interagents in the development of FM, an actual agent-based system. Finally, Section 6 analyzes which features distinguish our approach from related approaches, whereas Section 7 presents some final remarks.

2 An Overview of Interagents

In this work, we view conversations as the means of representing the conventions adopted by agents when interacting through the exchange of utterances[3,4] —"utterance suggests human speech or some analog to speech, in which the message between sender and addressee conveys information about the sender".[5] More precisely, such conventions define the *legal* sequence of utterances that can be exchanged among the agents engaged in conversation: what can be said, to whom and when. Therefore, *conversation protocols* are coordination patterns that constrain the sequencing of utterances during a conversation.

Interagents mediate the interaction between an agent and the agent society wherein this is situated. As explained in Section 3, the main task of an interagent is the management of conversation protocols. Here we differentiate two roles for the agents interacting with an interagent: *customer*, played by the agent exploiting and benefiting from the services offered by the interagent; and *owner*, played by the agent endowed with the capability of dynamically establishing the policies that determine the interagent's behavior. Needless to say that an agent can possibly play

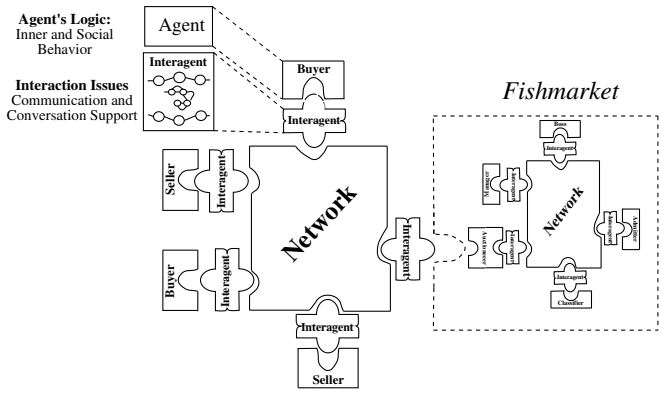

Fig. 1: Interagents conform the infrastructure of the agents composing a multi-agent system. This figure shows the Fishmarket, an electronic auction house where interactions among agents (trading agents and market intermediaries) takes place through their corresponding interagents.

both roles at the same time. Moreover, several owners can even share their property (*collective ownership*), whereas several customers can make use of the same interagent (*collective leasing*).

In what follows we provide a detailed account of the full functionality of interagents —mostly from the point of view of customers— to illustrate how they undertake conversation management. An interagent is responsible for posting utterances of its customer to the corresponding addressee and for collecting the utterances that other agents address to its customer. This *utterance management* abstracts customers from the details concerning the agent communication language and the network protocol. Each interagent owns a collection of relevant conversation protocols (CP) used for managing its customer conversations. When its customer intends to start a new conversation with another agent the interagent instantiates the corresponding conversation protocol. Once the conversation starts, the interagent becomes responsible for ensuring that the exchange of utterances conforms to the CP specification.

Before setting up any conversation the interagent must perform a *CP negotiation* process with the interagent of the addressee agent. The goal of CP negotiation is to reach an agreement with respect to the conversation protocol to be used (see 3.6). Moreover, before starting a conversation, the interagent performs a *CP verification* process. This process checks whether the CP to be used verifies the necessary conditions (liveliness, termination, deadlock and race condition free) for guaranteeing the correct evolution of an interaction. Finally, an interagent allows its customer to hold several conversations at the same time. This capability for *multiple conversations* is important because, although in the paper we consider only conversations with two participants (dialogues), conversations with any number of participants are built as a collection of simultaneous CP instances. In other words, the agent views a conversation as involving n participants while its interagent views such conversation as a collection of simultaneous dialogues represented as multiple CP instances.

Next we introduce an example of multi-agent conversation extracted from FM,[6,7] an agent-mediated electronic market that will serve to illustrate both the use and functionality of interagents. FM is an electronic auction house based on the traditional fish market auctions in which trading (buyer and seller) heterogeneous (human and software) agents can trade. The main activity within FM, the auctioning of goods, is governed by the auctioneer who makes use of the so-called *downward bidding protocol* (DBP). When the auctioneer opens a new *bidding round* to auction a good among a group of agents, he starts quoting offers downward from the chosen good's starting price. For each price called, three situations might arise during the open round: i) several buyers submit their bids at the current price. In this case, a collision comes about, the good is not sold to any buyer, and the auctioneer restarts the round at a higher price; ii) only one buyer submits a bid at the current price. The good is sold to this buyer whenever his credit can support his bid. Whenever there is

an unsupported bid the round is restarted by the auctioneer at a higher price, the unsuccessful bidder is punished with a fine, and he is expelled out from the auction room unless such fine is paid off; and iii) no buyer submits a bid at the current price. If the reserve price has not been reached yet, the auctioneer quotes a new price which is obtained by decreasing the current price according to the price step. If the reserve price is reached, the auctioneer declares the good *withdrawn* and closes the round.

The conventions that buyer agents have to comply with when interacting with the auctioneer during a bidding round are represented by means of a conversation protocol managed by the so-called *trading interagents*. These are owned by the institution, the auction house, but used by trading agents. Section 5 provides a more thorough discussion about the role played by interagents in FM.

3 Conversation Protocols

A conversation protocol (CP) defines a class of legal sequences of utterances that can be exchanged between two agents holding a conversation. We model and implement a CP as a special type of Pushdown Transducer (PDT), which can be seen in turn as a combination of a Finite-State Transducer (FST) and a Pushdown Automaton (PDA):

- An FST is simply a Finite State Automaton (FSA) that deals with two tapes. To specify an FST, it suffices to augment the FSA notation so that labels on arcs can denote pairs of symbols;[8]
- A PDA is composed of an input stream and a control mechanism —like an FSA— along with a stack on which data can be stored for later recall.[9, 10]

Therefore, a PDT is essentially a pushdown automaton that deals with two tapes. A PDA can be associated to a PDT by considering the pairs of symbols on the arcs as symbols of a PDA. The choice of PDTs as the mechanism for modeling CPs is motivated by several reasons. First, analogously to other finite-state devices a few fundamental theoretical basis make PDTs very flexible, powerful and efficient.[8] They have been largely used in a variety of domains such as pattern matching, speech recognition, cryptographic techniques, data compression techniques, operating system verification, etc. They offer a straightforward mapping from specification to implementation. The use of pairs of symbols permits allows labeling arcs with (p/d) pairs (where p stands for a performative, and d stands for a predicate). This adds expressiveness to the representation of agent conversations. PDTs, unlike other finite state devices, allow us to store, and subsequently retrieve, the contextual information of ongoing conversations.

In this section we explain CPs in depth, whose management is the *raison d'être* of interagents. We start introducing a conceptual model of CPs. Next, the formalism underpinning our model is presented in order to provide the foundations for a further analysis concerning the properties that CPs must exhibit. Finally, the way of negotiating, instantiating, and employing CPs is also explained.

3.1 Conceptual Model

Conceptually, we decompose a CP into the following elements (see Figure 3): a finite state control, an input list, a pushdown list, and a finite set of transitions.

First, the *finite state control* contains the set of states representing the communication state of the interagent's customer during an ongoing conversation. We shall distinguish several states based on the communicative actions that they allow: *send*, when only the utterance of performatives is permitted, *receive*, when these can be only received, and *mixed*, when both the utterance and reception are feasible.

The utterances heard by an interagent during each conversation are stored into an *input list*. In fact, this input list is logically divided into two sublists: one for keeping the utterances' performatives, and another one for storing their predicates. It is continuously traversed by the interagent

in search of a (p/d) pair which can produce a transition in the finite state control. Notice that the continuous traversing the input list differs from the one employed by classic FSAs whose read only input tapes are traversed from left to right (or the other way around).

Let us consider a particular CP related to an ongoing conversation. We say that an utterance is *admitted* when it is heard by the interagent and subsequently stored into the input list. Admitted utterances become *accepted* when they can cause a state transition of the finite state control. Then they are removed from the input list to be forwarded to the corresponding addressee, and thereupon the corresponding transition in the finite state control takes place. Notice that all utterances are firstly admitted and further on they become either accepted or not. Therefore, the input list of each CP keeps admitted utterances that have not been accepted for dispatching yet. From now on, these criteria will be taken as the message sending and receiving semantics used in Section3.5*.

The context of each conversation can be stored and subsequently retrieved thanks to the use of a *pushdown list*. Such context refers to utterances previously sent or heard, which later can help, for example, to ensure that a certain utterance is the proper response to a previous one. For instance, an utterance in the input list —represented in a KQML-like syntax— will be processed only if its sender, receiver and the value of the keyword *:in-reply-to* match respectively the receiver, sender and value of the keyword *:reply-with* of the topmost message on the pushdown list.

Finally, each transition in the *finite set of transitions* of a CP indicates: i) what utterance can be either sent or received to produce a move in the finite state control; and ii) whether it is necessary to store (push) or retrieve (pop) the context using the pushdown list.

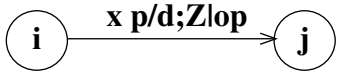

Fig. 2: Transition

Each arc of the finite state control is labeled by one or more transition specifications. The structure of a transition specification is shown in Figure 2: a transition from state i to state j occurs whenever an utterance with polarity x, performative p, and predicate d is found in the input list and the state of the pushdown list is Z. In such a case, the chain of stack operations indicated by op is processed. In order to fully specify a transition, the following definitions and criteria must be observed:

- the polarity of an utterance u, denoted as $polarity(u)$, can take on one of two values: $+$, to express that the utterance is sent, or -, to indicate that the utterance is received. From this, for a given CP p we define its *symmetric view* $\overline{p}$ as the result of inverting the polarity of each transition;
- the special symbol $p/*$ represents utterances formed by performative p and any predicate, whereas the special symbol $*/d$ stands for utterances formed by any performative and predicate d;
- when several transitions $p_1/d_1; Z|op, \ldots, p_n/d_n; Z|op$ label an arc, they can be grouped into a compound transition as $(p_1/d_1|\ldots|p_n/d_n); Z|op$;
- our model considers the following stack operations:

  `push` pushes the utterance selected from the input list onto the stack;

  `pop` pops the stack by removing the topmost utterance; and

  `nop` leaves the stack unchanged. Usually this operation is omitted in specifying transition;

- when the state of the pushdown list is not considered for a transition, it is omitted in the transition specification. In CPs, e-moves, i.e. moves that only depends on the current state of finite state control and the state of the pushdown list, are represented using the transition specification Z|op.

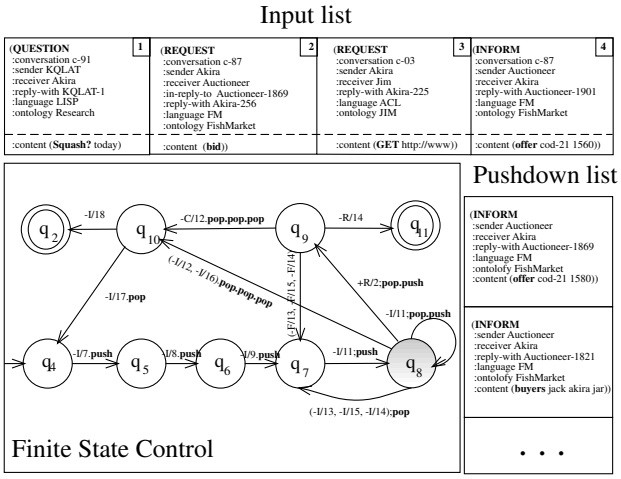

Fig. 3: Partial view of the CP DBP used by trading interagents in the Fishmarket.

For instance, Figure 3 depicts the CP employed by a trading interagent to allow its customer (buyer agent *Akira*) to participate in a bidding round open by the *auctioneer* agent as explained in Section 2. Notice that the performatives of the utterances considered in the figure follow the syntax of Table 1, and the predicates within such utterances belong to the list in Table 2. This simple example illustrates the use of the pushdown list: i) for saving the state of the bidding round (round number, good in auction, list of buyers, etc.); and ii) for ensuring that whenever a request for bidding is dispatched, the bid conveyed to the auctioneer will be built by recovering the last offer pushed by the trading interagent onto the pushdown list.

ID	Speech Act	Description
Q	QUESTION	SOLICIT the addressee to INFORM the sender of some proposition
R	REQUEST	SOLICIT the addressee to COMMIT to the sender concerning some action
I	INFORM	ASSERT + attempt to get the addressee to believe the content
C	COMMIT	ASSERT that sender has adopted a persistent goal to achieve something relative to the addressee's desires
F	REFUSE	ASSERT that the sender has not adopted a persistent goal to achieve something relative to the addressees desires

Tab. 1: Types of performatives following Cohen and Levesque[11] (extracted from Parunak[5])

3.2 Formal Definition

Now it appears convenient to formally capture the conceptual model introduced above to be able to reason, later on, about the properties that we must demand from CPs. Therefore, formally we define a conversation protocol as an 8-tuple :

$$CP = \langle Q, \Sigma_1, \Sigma_2, \Gamma, \delta, q_0, Z_0, F \rangle$$

such that:

- Q is a finite set of state symbols that represent the states of the finite state control.
- Σ_1 is a finite alphabet formed by the identifiers of all performatives that can be uttered during the conversation.
- Σ_2 is a finite input list alphabet composed of the identifiers of all predicates recognized by the speakers.

- Γ is the finite pushdown list alphabet.
- δ is a mapping from $Q \times \Sigma_1 \times \Sigma_2 \times \Gamma^*$ 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.

CPs only contemplate a finite number of moves from each state that must belong to one of the following types:

moves using the input list these depend on the current state of the finite state control, the performative and predicate of a message into the input list and the state of the pushdown list. For instance, the move expressed by the following transition allows a trading interagent to convey to the auctioneer a request for bidding received from its customer (a buyer agent) whenever an offer, received from the auctioneer, has been previously pushed upon the pushdown list

$$\delta(q_8, +\mathsf{REQUEST}, \mathsf{bid}, \mathsf{offer}Z) = \{(q_9, \mathsf{bid}Z)\};$$

e-moves these depend exclusively on the current state of the finite state control and the state of the pushdown list. e-moves are specifically employed to model and implement time-out conditions within CPs, so that interagents can handle expired messages and automatically recover from transmission errors

$$\delta(q_9, e, e, \mathsf{bid}Z) = \{(q_8, Z)\}.$$

It should be noted that at present we are only interested in deterministic CPs (DCP): a CP is said to be deterministic when for each state $q \in Q$, $p \in \Sigma_1$, $d \in \Sigma_2$ and $Z \in \Gamma^*$ there is at most one possible move, that is $|\delta(q, p, d, Z)| \leq 1$.

3.3 Instantiation

CPs can be defined declaratively and stored into conversation protocol repositories open to interagents. Each CP is identified by a unique name anonymously set by the agent society. When an interagent is requested by its customer to start a conversation with another agent it must retrieve the appropriate CP from a conversation repository, and next proceed to instantiate it. In fact, the CP must be instantiated by each one of the interagents used by the agents intending to talk.

We say that a CP becomes fully instantiated when the interagent creates a CP instance, i.e. after setting the values for the following attributes:

CP name CP class to which the CP instance belongs;

speakers identifiers of the agents to engage in conversation. Notice that we shall restrict a CP instance to consider exactly two speakers: the agent that proposes to start a conversation, the *originator*, and his speaker, the *helper*. In spite of this limitation, we are not prevented from defining multi-agent conversation, since these can be created by means of multiple CP instances;

conversation identifier a unique identifier created by the originator;

polarity this property indicates how to instantiate the polarity of each transition of the CP: if the instance polarity is positive, each transition is instantiated just as it is, whereas if it is negative each transition polarity is inverted. Notice that the helper must instantiate the symmetric view of the originator's CP in order to ensure protocol compatibility as shown in Section 3.5.

transport policies such as time-out or maximum time allowed in the input list. These can be altered during the course of a conversation whereas the rest of attributes of a CP instance remain fixed. The use of transport policies require to extend the CP being instantiated with e-moves that enable to roll back to previous conversation states.

Then, and considering the definition above, from the point of view of interagents the conversations requested to be held by its customer can progress through several states:

pre-instantiated after retrieving the requested CP from the originator's conversation repository;

instantiated a CP becomes instantiated when the originator creates a CP instance, and subsequently asks the helper for starting a new conversation accepting the terms (attributes) of the interaction expressed by the CP instance;

initiated this state is reached when both speakers agree on the value of the attributes of a new conversation, as a result of the negotiation phase described in Section 3.6;

running state attained after the first utterance;

finished a conversation is over whenever either the final state of the CP is reached, the helper refuses to start it, or an unexpected error comes about.

3.4 Instantaneous Description

An instantaneous description formally describes the state of a CP instance at a particular time. An *instantaneous description* of a CP instance p is a 7-tuple:

$$\langle o, h, p, t, q, l, \alpha \rangle$$

such that:

o is the originator agent; h is the helper agent; p is the polarity of the CP instance; t is the current setting of transport policies; q is the current state of the finite state control; i represents all utterances currently kept by the input list; and α is the current state of the pushdown list.

Figure 3 depicts the instantaneous description for an instance of the CP DBP employed by a trading interagent to allow its customer (a buyer agent) to participate in a bidding round open by the auctioneer agent. We identify buyer *Akira* as the originator, the *auctioneer* as the helper, and the colored node q_8 as the state of the finite state control.

A deterministic CP has at most —without taking into account possible e-moves for dealing with expired utterances— one possible move from any instantaneous description. However, continuously traversing the input list in search of an utterance that causes a transition can lead to *race conditions*. For instance, in the CP instance of Figure 3 the second and fourth utterances can originate a race condition since both utterances can cause a move in the finite state control. Thus, it is necessary to define criteria for deciding which utterance must be accepted as we show in the next section.

3.5 Compatibility Semantics

In order to ensure the correct exchange of utterances during a conversation, there are important, desirable properties such as termination, liveliness, deadlock and race condition free that CPs must verify. In what follows we concentrate exclusively on the two last properties, since to guarantee both termination and liveliness it suffices to assume that every CP whose set of final states is non-empty does not remain forever in the same state.

First we formulate our notion of CP compatibility, following the notion of protocol compatibility proposed by Yellin and Strom in,[12] whose work provides, in fact, the foundations of our analysis.

CPs can be assigned two different semantics: asynchronous or synchronous. Although asynchronous semantics may facilitate implementation, it makes generally harder reasoning about certain properties, such as *deadlock* which has proven undecidable under these semantics.[12] On the

contrary, under synchronous semantics, reasoning about such properties is easier, though an implementation technique must map these semantics to a particular implementation. For our purposes, we have decided for a synchronous semantic for CPs and, consequently, for devising the adequate mechanisms for implementation. Such a type of semantic requires to assume that a speaker can send an utterance to the other speaker only if that is willing to receive the utterance. Therefore, we must assume that the finite state controls of both CP instances advance synchronously, and hence that sending and receiving an utterance are atomic actions. Upon this assumption, Yellin and Strom introduced the following notion of compatibility of protocols: "Protocols p_1 and p_2 are *compatible* when they have no *unspecified receptions*, and are *deadlock free*". On the one hand, in terms of CPs, the absence of unspecified receptions implies that whenever the finite state control of a CP instance corresponding to one of the speakers is in a state where a utterance can be sent, the finite state control of the CP instance of the other speaker must be in a state where such utterance can be received. On the other hand, deadlock free implies that the finite state control of both CP instances are either in final states or in states that allow the conversation to progress. Interestingly, the authors prove the existence of an algorithm for checking protocol compatibility. By applying such algorithm, it can be proved that under synchronous semantics a CP instance p and its symmetric view $\overline{p}$ are always compatible. From this follows that two CP instances are compatible if both belong to the same CP class, and both have the same speakers but different polarity. In this way, the complete agreement on the order of the utterances exchanged between the speakers is guaranteed.

Concerning the implementation, observe that the atomicity of sending and receiving utterances cannot be guaranteed. Nonetheless, when compatible CP instances lack mixed states, the unspecified receptions and deadlock free properties can still be guaranteed. But the presence of mixed states can lead to race conditions that prevent both speakers from agreeing on the order of the messages, and therefore unexpected receptions and deadlocks might occur. In order to avoid such situations, low-level synchronization mechanisms must be provided in accordance with the interpretation given to message sending and receiving in Section 3.1.

For this purpose, we consider that the speakers adopt different conflict roles —either *leader* or *follower*— when facing mixed states. The leader will decide what to do, whereas the follower will respect the leader's directions. By extending CP instances with a new attribute —which can take on the values *leader* or *follower*— we include the role to be played by each speaker in front of mixed states. Besides, we consider two special symbols —TRY and OK— that alter the interpretation of message sending. Thus, on the one hand when a message is sent under the TRY semantics, the receiver tries to directly accept the message without requiring previous admission. On the other hand, the OK symbol confirms that the TRY was successful. Then given a CP $\langle Q, \Sigma_1, \Sigma_2, \Gamma, \delta, q_0, Z_0, F\rangle$ for each mixed state $x \in Q$ the CP instance corresponding to the follower will be augmented in the following way:

$\forall p \in \Sigma_1, \forall d \in \Sigma_2, \forall Z \in \Gamma^*, \forall Z' \in \Gamma^*, \forall q \in Q$ such that $\exists \delta(x, +p, d, Z) = \{(q, Z')\}$ then a new state $n \notin Q$ will be added $Q = Q \cup \{n\}$ and the following transitions will be added:

1. $\delta(x, +TRY(p), d, Z) = \{(n, Z)\}$
2. $\delta(n, -OK, e, Z) = \{(y, Z')\}$
3. $\forall p' \in \Sigma_1, \forall d' \in \Sigma_2, \forall Z'' \in \Gamma^*$ then $\delta(x, -p', d', Z'') = \delta(n, -p', d', Z'')$

Therefore, when the leader is in a mixed state and sends an utterance, the follower admits it and subsequently accepts it. Conversely, when the follower is in a mixed state and sends an utterance, the leader, upon reception, determines if it is admitted or not. Figure 4 illustrates how the CP instance on the left should be augmented to deal with the mixed state q_8.

Notice that the conflict role played by each speaker must be fixed before the CP becomes completely instantiated. This and other properties of CP instances need be negotiated by the speakers as explained in the next section.

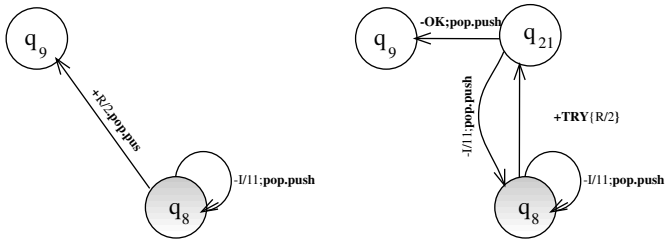

Fig. 4: Augmented CP instance

3.6 Conversation Protocol Negotiation

In Section 3.3 we introduced the several attributes of a CP instance that have to be fixed before the conversation between the speakers becomes fully instantiated, and subsequently started. The value of each one of these attributes has to be mutually agreed by the speakers in order to guarantee conversation soundness. For this purpose, interagents have been provided with the capability of negotiating such values by means of the so-called *Handshake* phase, following the directions of their customers. During this process, the initial connection between the originator and the helper is established, and next the originator conveys its multi-attribute proposal (the set of attributes' values) to the helper. Then, we distinguish two models of negotiation based on the helper's response: one-step and two-step negotiation.

In one-step negotiation the helper either automatically accepts or refuses the originator's proposal. The following sequence depicts a typical exchange for this model of negotiation, where q values indicate the degree of preference over each proposal.

```
originator:  START
             CP/DBP; id=21; polarity=+; leader=me; q=0.7
             CP/DBP; id=21; polarity=-; leader=you; q=0.3
helper:      OK
             CP/DBP; id=21; polarity=+; leader=me
```

In this example the helper accepts the second proposal from the originator.

In two-step negotiation, instead of directly accepting or refusing the originator proposal, the helper can reply with a a list of counterproposals ranked according to its own preferences. Then, the originator can either accept one of these proposals or cancel the process.

```
originator:  START
             CP/DBP; id=22; polarity=+; leader=me; timeout=500; q=0.7
             CP/DBP; id=22; polarity=-; leader=you; timeout=1000; q=0.3
helper:      NOT
             CP/DBP; id=22; polarity=-; leader=me; timeout=200; q=0.4
             CP/DBP; id=22; polarity=+; leader=you; timeout=200; q= 0.6
originator:  OK
             CP/DBP; id=22; polarity=+; leader=you; timeout=200
```

In this example the helper refuses the proposals of the originator, who finally accepts the first helper's counterproposal.

It should be noted here that the concrete conversation protocol to be instantiated can be negotiated too. For this purpose, we have introduced the CP type (f.i. the CP/DBP for the downward bidding protocol), analogously to MIME content types (text/html, image/gif, etc.). On the other hand, it is nonsense to negotiate certain attributes for some CPs. For instance, the polarity of the CP to be employed by an auctioneer attempting to open a DBP round is unnegotiable since the auctioneer cannot play the role of a buyer and vice versa.

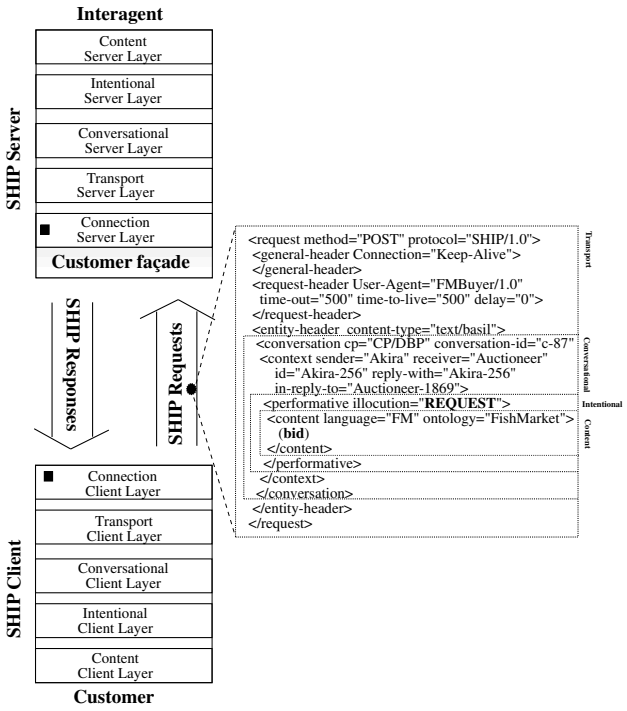

Fig. 5: All messages between an interagent and its customer are encapsulated as either requests or responses of the SHIP protocol.

4 JIM

In this section we introduce JIM, a general-purpose interagent enabled with the capability of managing CPs which has proven its usefulness in several applications (see Section 5). We precede the presentation of JIM by the introduction of the SHIP protocol since this has profoundly shaped its architecture.

4.1 SHIP An Extensible and Hierarchical Interaction Protocol

The extenSible and Hierarchical Interaction Protocol (SHIP) offers a layered approach to support all levels of agent interaction (a layer per level introduced in Section 1). Each layer groups a set of protocols with similar functionality together, and provides an abstract interface to higher layers that hides the details related to the particular protocol in use. For instance, two agents can exchange utterances without worrying about the underlying communication mechanism —message passing, tuple-space or remote procedure call. In addition to this, the hierarchical structure of SHIP allows agents to choose their interaction level. This involves the use of the sub-hierarchy of SHIP represented by the chosen layer and the layers below, but not higher layers, e.g. an agent may decide to use the transport layer and the lower levels (the connection layer in this case) but not the higher layers of SHIP. Lastly, SHIP is extensible, in the sense that it permits the incorporation (plug-in) of new protocols into each layer.

SHIP is a request/response protocol which distinguishes two roles: client and server. An interagent may play both roles at the same time, e.g. it can act as a server for its customer, and as a client for another interagent. In a typical message flow a client sends a request to the server, and this sends a response back to the client. All messages exchanged between a client and a server are encapsulated as either requests or responses of the SHIP protocol as depicted by Figure 5. Observe that both requests and responses are represented as XML[13] messages that nest the information corresponding to each level of interaction. On the sender side, messages to be sent are wrapped at each layer and passed down till getting to the communication layer responsible for the sending.

Upon reception, on the receiver side, messages are unwrapped at each layer and passed up to higher layers till the message arrives at the layer chosen by the agent for interacting.

SHIP has been built upon the following hierarchy of layers (from top to bottom): content, agent communication, conversation,transport and connection. In what follows we describe the functionality of each one of these layers taken as example the SHIP request in Figure 5 corresponding to a request for bidding of a buyer agent.

The *content layer* is concerned with the language for representing the information exchanged between agents. Such information refers to terms of a specific vocabulary of an application domain (ontology[14]). For example, a piece of information at this level might be the predicate (*bid*) codified in the *FM* language using the ontology *Fishmarket*.

The *intentional layer* codifies and de-codifies performatives in a given ACL. For example, a performative requesting for bidding at this level wraps the predicate (*bid*) received from the content layer with the illocution *REQUEST* of the Basic ACL (*Basil*) whose performatives are shown in Table 1.

The *conversational layer* is in charge of providing the necessary mechanisms for the negotiation, instantiation, and management of CPs as explained in depth in Section 3. Following the example above, the performative is wrapped with the information related to the context in which it is uttered (conversation protocol in use, sender, receiver, etc.).

The *transport layer* is concerned with the transport of utterances. For this purpose, we have devised the Inter-Agent Protocol (IAP), an extensible application-level protocol based on the Hypertext Transfer Protocol (HTTP). IAP constrains the body of a message to be an utterance. Moreover IAP overrides the set of request methods provided by HTTP. For instance, on the one hand the POST method is used to indicate that the utterance enclosed in the message body must be addressed to the receiver specified in the context. On the other hand, the GET method is used to retrieve the utterance accepted by an interagent that matches the pattern of utterance enclosed in the message body. Furthermore, a new addressing scheme has been introduced (iap://) and a new default port (2112).

IAP allows messages to be tagged with different headers to specify their lifetime:

- the delay header indicates how long an utterance queued in an interagent is postponed before being processed. In this way, an agent can tell its interagent to deliver a given utterance after a certain period of time;
- the time-out header indicates the maximum period of time that an agent commits to wait for receiving the response to a given utterance;
- the time-to-live header indicates the lifetime of the utterance once admitted by an interagent —thus, when this time expires, the utterance is thrown away by the receiving interagent and the sender receives (back) an error message.

We have introduced IAP due to the lack of an adequate transport protocol for agents' communicative acts. Currently, most ACLs such as FIPA ACL[15] or KQML[16] only define minimal message transport mechanisms —in FIPA ACL terms "a textual form delivered over a simple byte stream".[15] As an alternative to simple TCP/IP sockets, other transport protocols such as HTTP, SMTP or even GSM have been proposed. However, on the one hand these protocols offer many unnecessary features for this task, and on the other hand they lack many desirable features. Hence the convenience of designing IAP as a specialized protocol for the transport of utterances.

The *connection layer* occupies the lowest level of the SHIP protocol. It is responsible for the creation of connections for SHIP requests. We consider a connection as a virtual circuit established between two agents for the purpose of communication. Different network protocols can be employed to establish such connections. Currently, JIM interagents support a wide variety of network protocols: memory I/O streams, (TCP/IP) stream sockets, and SSL sockets; others like RMI and IIOP will be supported in the near future.

As a final remark, notice that SHIP has required the introduction of several MIME (Multipurpose Internet Mail Extensions) types for declaring the types of conversation protocol (e.g. CP/DBP), of ACL (e.g. text/basil, text/kqml, text/fipaacl, etc.), etc.

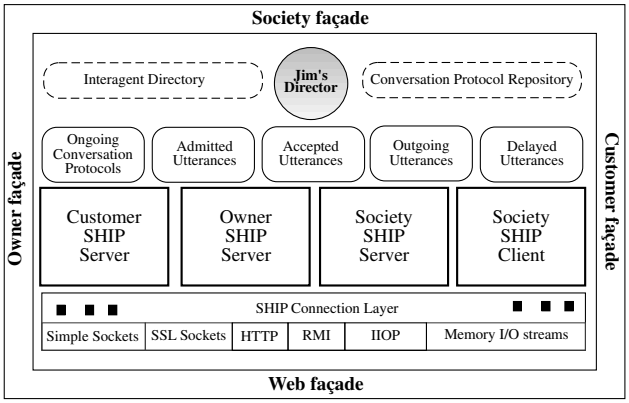

Fig. 6: An Overview of JIM's Architecture.

4.2 Architecture

The architecture of JIM(Figure 6 shows a sketch of it) is best explained in terms of its components:

Interagent Directory This component offers interagent naming services (white pages) that translate from unique identifiers of interagents into their logical addresses and vice versa. Each interagent can communicate with a subset of interagents of the agent society, named its *acquaintances*. A reflexive and symmetric, but not transitive, relation *acquaintance-of* can be established among interagents. An interagent's acquaintanceship is a set of unique identifiers and can vary dynamically, allowing that a collection of agents grow dynamically.

Director This component directs the behavior of the rest of components. It can receive specific directives from both its owner and its customer. Each CP defines a set of constraints that the director takes into account to sort the utterances stored in the different buffers.

CP repository This component stores all CP classes that can be instantiated by an interagent. Such repository can be updated in either *statically* or *dynamically*, e.g. in FM only the market intermediaries working for the institution, the auction house, can define, and store at run-time new CP classes into the CP repository of each interagent. Notice that the conversation protocol repository can either be owned by only one interagent or shared among several interagents.

Ongoing conversation protocols This component stores all *running* CPs (see Section 3.3). A CP instance is kept for each conversation in which the interagent's customer is involved. Once a CP is over (reaches the *finished* state) is thrown away (or alternatively kept for tracing purposes).

Buffer for admitted utterances This component is responsible for storing all utterances coming from other interagents in the agent society. An utterance is stored here until either it is accepted or it expires due to a time-out condition.

Buffer for accepted utterances This component stores utterances that have already been accepted by a CP but which have not been required by the customer yet. Once one of these utterances is required by the customer it is packaged up as a SHIP response and forwarded to it.

Buffer for outgoing utterances This component stores utterances that have been forwarded from a customer agent, and received by its interagent but that have not been accepted yet. Notice that the input list of each CP instance can be seen as a composition of the buffers for both incoming and ongoing utterances.

Buffer for delayed utterances This component stores the utterances that have been sent by the customer specifying a lapse of time (using the delay header of the IAP protocol) before they are actually transmitted. When such lapse of time expires, they are automatically transferred to the buffer for outgoing utterances.

Observe that the architecture of JIM presents up to four different façades: *owner, customer, society and web* façades. Each façade represents a different entry for each type of agent requesting the services provided by JIM. The services provided through each façade differ. For instance, only the owner façade allows an agent to upgrade the CP repository. The web façade simply provides access to World-Wide Web resources. For instance, this façade allows a human agent to interact with the JIM's customer through a standard WWW browser. Each façade is built on top of a different instance of the SHIP protocol. In JIM a chain of responsibility is established through the different protocol layers that conform the SHIP instance of each façade. That is to say, each layer is responsible for managing some of the services provided by JIM.

5 The Agent-based Electronic Auction Market FM

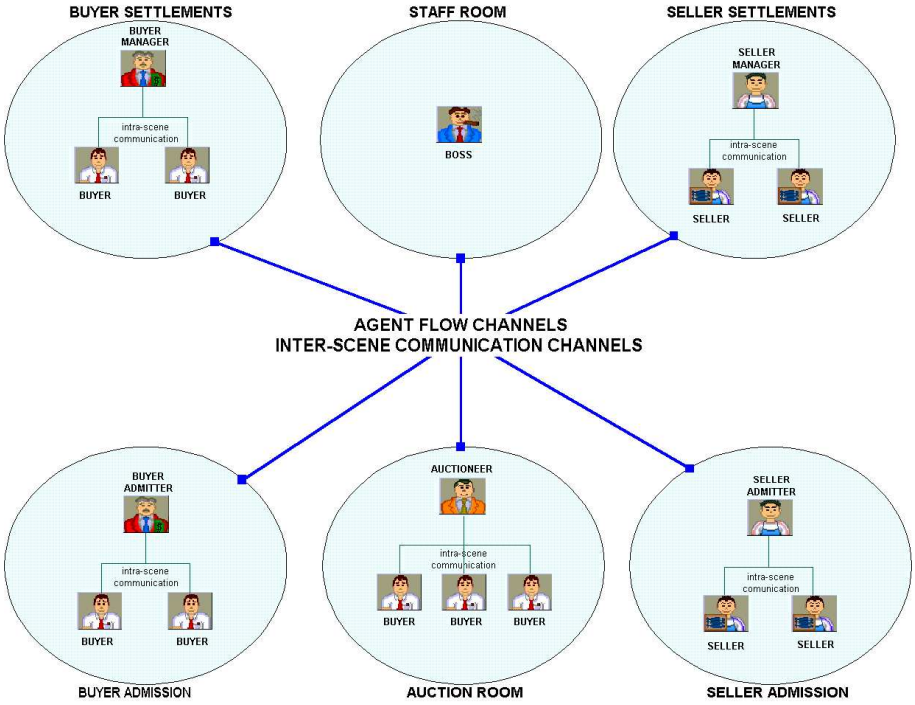

Fig. 7: FM Multi-scene Structure

In this section, we introduce FM an example of a particular agent-based system that will serve to illustrate both the practical use and functionality of interagents. In fact, most of the features of interagents have emerged as a generalization of the more ad-hoc notion of *remote control* introduced at the first stages of the development of FM.[17] In[17,18] we presented FM, an electronic auction house based on the traditional fish market auctions. Further on, FM was extended to produce the multi-agent test-bed for electronic auctions introduced in[7] and fully reported in.[6] The current version of FM, FM0.9beta,[19] is now available and can be downloaded from the FishMarket project web page.[20]

The actual fish market is an *institution* (cf. North's[21]) that establishes and enforces explicit conventions. From our view, it can be described and modeled as a place where several *scenes* run simultaneously, at different places, but with some causal continuity.[17,18] 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. Each scene is supervised by one of the market intermediaries (auctioneer, buyer admitter, buyer manager, seller admitter, seller manager, and boss) which represent and work for the institution.

In a highly mimetic way, the workings of its electronic counterpart, FM, also involve the concurrency of several scenes governed by the market intermediaries identified in the fish market. Therefore, seller agents register their goods with a seller admitter agent, and can get their earnings (from a seller manager agent) once the auctioneer agent has sold these goods in the auction room. Buyer agents, on the other hand, register with a buyer admitter agent, and bid for goods which they pay through a credit line that is set up and updated by a buyer manager agent. Figure 7 shows the conceptual model of FM as a multi-scene multi-agent scenario.

Similarly to the actual fish market, FM can be regarded as an *electronic institution* in the sense proposed by Sierra and Noriega.[18,22] 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 CPs handled by interagents that establish what utterances can be said by whom and when. We shall differentiate two types of interagents in FM: *trading interagents*, owned by the institution but used by trading agents, and *institutional interagents*, both owned and used by those agents functioning as market intermediaries. Therefore, all interagents are owned by the institution, but we identify two types of customers: the trading agents and the institution itself.

Trading interagents constitute the sole and exclusive means through which trading agents interact with the market intermediaries representing the institution. Within each scene, a trading interagent must employ a different CP to allow its customer to talk to the market intermediary in charge of it. Therefore, trading interagents are responsible for enforcing the protocols that guarantee that every trading agent behaves according to the rules of the market. For instance, in the auction room a trading interagent uses the CP/DBP depicted in Figure 3 for allowing its customer (a buyer agent) to participate in a bidding round. In general, we shall refer to the conversations supported by trading interagents as *intra-scene conversations* because they are dialogues held within each scene between the market intermediary responsible of the scene and each trading agent situated in the scene.

As to market intermediaries, they must hold several conversations at the same time with the agents in the scene that they govern. For this purpose, the institutional interagents that they employ must exploit their capability for supporting multiple conversations, as explained in Section 2, by building collections of simultaneous CP instances (one per trading agent).Thus, for example, the auctioneer's interagent maintains an instance of the CP/DBP for each buyer agent in the auction room. Moreover, not only are institutional interagents used to support conversations with trading agents, but also to allow those agents working as market intermediaries to coordinate their activities. We will refer to the conversations held between market intermediaries as *inter-scene conversations*.

Both types of interagents have been endowed with further capabilities. On the one hand, they are in charge of conveying monitoring information to the auditing agent (also called monitoring agent), so that market sessions can be monitored, and analyzed step-by-step. On the other hand, interagents provide support for agent failure handling. For example, when a trading agent either goes down or simply fails to consume the utterances conveyed by its interagent (say that the trading agent lapses into an extremely demanding deliberative process for elaborating its strategies), this interagent pro-actively unplugs its customer from the market and leaves the market on its behalf.

Finally, by applying the results deriving from our analysis in Section 3.5, the interagents in FM can dynamically (at run-time) reconfigure retrieved CPs in order to guarantee the verification of liveliness, termination, and deadlock and race condition free.

Summarizing, the successful incorporation of interagents into FM has proven their usefulness by coping with several tasks: i) to handle the interplay between trading agents and the market in-

stitution (intra-scene conversations); ii) to handle the coordination between market intermediaries' tasks (inter-scene conversations); iii) to provide support for the monitoring of market sessions; iv) to handle agents' failures; v) to reconfigure CPs so as to ensure protocol compatibility.

6 Related Work

So far, much effort in agent research concerning agent interaction has focused on the semantic and pragmatic foundations of different agent communication languages (ACLs) based on speech act theory.[11,23,24] 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 make easier their interoperation.[25] Generally, an ACL is composed of three main elements: an open-ended vocabulary appropriate to a common application area,[26] an inner language (KIF–Knowledge Interchange Format[27]) to encode the information content communicated among agents, and an outer language (KQML–Knowledge Query and Manipulation Language[28]) to express the intentions of agents. Apart from KQML, there are other ACLs in use such as FIPA ACL,[15] KQML Lite,[29] etc. However this research is currently broadening from the specification of individual utterances to include the characterization of goal-directed conversations for which agents will use ACLs. Thus new works in speech act research, exemplified by efforts such as KAoS,[30] Dooley Graphs,[5] COOL[4] and MAGMA,[31] attempt at representing and reasoning about the relationships within and among conversations, or groups of utterances. A number of formalisms have been proposed for the modeling of conversations: FSMs,[4,32] Dooley graphs,[5] Petri Nets,[33] etc. Our approach proposes a new model based on PDTs that allows to store the context of ongoing conversations, and, in contrast with other approaches, that provides a mapping from specification to implementation. Moreover, as a distinctive feature from other approaches, we provide our model with a detailed analysis that studies the properties that conversation protocols must exhibit in order to ensure protocol compatibility, and therefore the soundness of agent conversations.

Although there is a large number of software tools for developing agents,[34] not many of them happen to provide support for the specification of conversation protocols. AgentTalk,[35] COOL,[4] JAFMAS,[32] Agentis,[36] Jackal[37] and InfoSleuth,[38] do offer conversation constructs. JAFMAS, for instance, provides a generic methodology for developing speech-act based multi-agent systems using coordination constructs similar to COOL. In addition to this, as far as our knowledge goes, none of them offers dynamically and incrementally specifiable conversation protocols except for InfoSleuth.[38] We attempt to make headway in this matter with respect to other agent building tools by introducing interagents, autonomous software agents that permit both the dynamic and incremental definition of conversation protocols by both the agent engineer and the owner.

We have chosen the conceptualization of interagents as autonomous software agents instead of an agent's built-in conversation layer as proposed in other agent architectures because of the need to separate the agents' logics from the agents' interactions —such separation has proven to be valuable in the development of a particular type of agent-based systems, namely electronic institutions such as FM. Interagents —like KQML facilitators[39]— are inspired by the *efficient secretary* metaphor already introduced in the Actors model of concurrent computation.[40] Nonetheless, *interagents* —unlike KQML facilitators— offer the conversational level required by agents to cooperate in non-trivial ways.

7 Conclusions

We have introduced conversation protocols CPs as the methodological way to conceptualize, model, and implement conversations in agent-based systems. CPs allow to impose a set of constraints on the communicative acts uttered by the agents holding a conversation. We have also introduced interagents as autonomous software agents that mediate the interaction between each agent and the agent society wherein this is situated. Based on CPs and interagents we have proposed an *infrastructure* for easing the development of agent-based systems that takes charge of the complex, time-consuming interaction issues inherent to the construction of this type of systems. In this way, the overhead related to the management of the interaction tasks needed by an agent is shifted

to its interagent. Interagents employ CPs for mediating conversations among agents, and so the management of CPs is identified as their main task. Two major benefits are achieved by deploying our infrastructure from the point of view of the agent developer: on the one hand, their agents can reason about communication at higher levels of abstraction, and on the other hand they are released from dealing with interaction issues, and so they can concentrate on the design of the agents' logics. We have also introduced JIM, a general-purpose interagent. JIM is currently being used to coordinate the activities of the market intermediaries composing the *Fishmarket* system[7,20] and the interaction between the market as a whole and the participating buyers and sellers (see Figure 1). Additionally, JIM is being successfully employed by other ongoing research projects: the SMASH project,[41] that addresses the construction of prototype multi-agent systems with learning capabilities that cooperate in the solution of complex problems in hospital environments; and in the multi-agent learning framework Plural[42] which tackles the problem of sharing knowledge and experience among cognitive agents that co-operate within a distributed case-based reasoning framework.

Acknowledgments

This work has been supported by the Spanish CICYT project SMASH, TIC96-1038-C04001. Juan A. Rodríguez-Aguilar and Francisco J. Martin enjoy DGR-CIRIT doctoral scholarships FI-PG/96-8490 and FI-DT/96-8472 respectively.

References

[1] Victor R. Lesser. Reflections on the nature of multi-agent coordination and its implications for an agent architecture. *Autonomous Agents and Multi-Agent Systems*, 1:89–111, 1998.

[2] Nick R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. *The Knowledge Engineering Review*, 8(3):223–250, 1995.

[3] T. Winograd and F. Flores. *Understanding Computers and Cognition.* Addison Wesley, 1988.

[4] Mihai Barbuceanu and Mark S. Fox. Cool: A language for describing coordination in multi agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems*, 1995.

[5] H. Van Dyke Parunak. Visualizing agent conversations: Using enhanced dooley graph for agent design and analysis. In *Proceedings of the Second International Conference on Multi-Agent Systems*, 1996.

[6] Juan A. Rodríguez-Aguilar, Francisco J. Martín, Pablo Noriega, Pere Garcia, and Carles Sierra. Towards a test-bed for trading agents in electronic auction markets. *AI Communications*, 11(1):5–19, 1998.

[7] Juan A. Rodriguez-Aguilar, Francisco J. Martin, Pablo Noriega, Pere Garcia, and Carles Sierra. Competitive scenarios for heterogenous trading agents. In *Second International Conference on Autonomous Agents*, 1998.

[8] Emmanuel Roche and Yves Schabes. *Finite State Language Processing.* The MIT Press, 1997.

[9] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume I: Parsing of *Series in Automatic Computation.* Prentice-Hall, 1972.

[10] J. Glenn Brookshear. *Theory of Computation, Formal Languages, Automata, and Complexity.* The Benjamin/Cummings Publishing, 1989.

[11] P. R. Cohen and H. J. Levesque. Communicative actions for artificial agents. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 65–72, Menlo Park, CA., jun 1995. AAAI Press.

[12] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997. .

[13] Dan Connolly. *XML, Principles, Tools and Techniques.* O REILLY, 1997.

[14] Tom R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. In Nicola Guarino and Roberto Poli, editors, *Software Agents.* Kluwer Academic, 1993.

[15] FIPA 97. specification part 2: Agent communication language. Technical report, FIPA - Foundation for Intelligent Physical Agents, 1997.

[16] James Mayfield, Yannis Labrou, and Tim Finin. Evaluation of kqml as an agent communication language. In Michael Wooldridge and Jörg Müller, editors, *Intelligent Agents II*, pages 347–360. Springer Verlag, 1996.

[17] Juan A. Rodríguez-Aguilar, Pablo Noriega, Carles Sierra, and Julian Padget. Fm96.5 a java-based electronic auction house. In *Second International Conference on The Practical Application of Intelligent Agents and Multi-Agent Technology(PAAM'97)*, 1997.

[18] Pablo Noriega. *Agent-Mediated Auctions: The Fishmarket Metaphor.* PhD thesis, Universitat Autonoma de Barcelona, 1997. Also to appear in IIIA mongraphy series.

[19] Juan A. Rodríguez-Aguilar, Francisco J. Martín, Francisco J. Giménez, and David Gutierrez. Fm0.9beta users guide. Technical report, Institut d'Investigació en Intel.ligència Artificial. Technical Report, IIIA-RR98-32, 1998.

[20] The FishMarket Team. The fishmarket project. In *www.iiia.csic.es/Projects/fishmarket.*

[21] D. North. *Institutions, Institutional Change and Economics Perfomance.* Cambridge U. P., 1990.

[22] Carles Sierra and Pablo Noriega. Institucions electròniques. In *Proceedings of the Primer Congrès Català d'Intel.ligè ncia Artificial*, 1998.

[23] J. L. Austin. *How to Do Things With Words.* Oxford University Press, 1962.

[24] John Searle. *Speech Acts.* Cambridge University Press, 1969.

[25] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM, Special Issue on Intelligent Agents*, 37(7):48–53, July 1994.

[26] Tom R. Gruber. A translation approach to portable ontology specifications. Technical Report KSL–92–71, Knowledge Systems Laboratory, Computer Science Department, Standford University, April 1993.

[27] Matthew L. Ginsberg. Knowledge interchange format: The kif of death. *AI Magazine*, 12(3):57–63, 1991.

[28] Tin Finin, Yannis Labrou, and James Mayfield. Kqml as an agent communication language. In Jeff Bradshaw, editor, *Software Agents.* MIT Press, Cambridge, 1995. invited chapter.

[29] Robin MacEntire and Donal McKay. Kqml lite specification. Technical report, Lockheed Martin Mission Systems, 1998. Technical Report ALP-TR/03.

[30] J. M. Bradshaw. Kaos: An open agent architecture supporting reuse, interoperability, and extensibility. In *Tenth Knowledge Acquisition for Knowledge Based Systems*, 1996.

[31] Yves Demazeau. From interactions to collective behaviour in agent-based systems. In *European Conference on Cognitive Sciences*, 1995.

[32] Deepika Chauhan. *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation.* PhD thesis, ECECS Department, University of Cincinnati, 1997.

[33] Jean-Luc Koning, Guillaume François, and Yves Demazeau. Formalization and pre-validation for interaction protocols in multiagent systems. In *13th European Conference on Artificial Intelligence*, 1998.

[34] *AAAI-98 Workshop on Software Tools for Developing Agents*, 1998.

[35] Agenttalk: Describing multi-agent coordination protocols. http://www.cslab.tas.ntt.jp/at/.

[36] Mark d'Inverno, David Kinny, and Michael Luck. Interaction protocols in agentis. In *Third International Conference on Multi-Agent Systems*, 1998.

[37] R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, and Ian Soboroff. Jackal: a java-based tool for agent development. In *AAAI-98 Workshop on Software Tools for Developing Agents*, 1998.

[38] Marian Nodine, Brad Perry, and Amy Unruh. Experience with the infosleuth agent architecture. In *AAAI-98 Workshop on Software Tools for Developing Agents*, 1998.

[39] 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.

[40] Gul Agha. *Actors, A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[41] The smash project. http://www.iiia.csic.es/Projects/smash/.

[42] Francisco J. Martin, Enric Plaza, and Josep L. Arcos. Knowledge and experience reuse through communication among competent (peer) agents. 1998. To appear in International Journal of Software Engineering and Knowledge Engineering.

#Message	Predicate	Parameters
1	*admission*	*buyerlogin password*
2	*bid*	$[price]$
3	*exit*	
4	*deny*	*deny_code*
5	*accept*	$open\|closed$ *auction_number*
6	*open_auction*	*auction_number*
7	*open_round*	*round_number*
8	*good*	*good_id good_type starting_price resale_price*
9	*buyers*	$\{buyerlogin\}^*$
10	*goods*	$\{good\_id\ good\_type\ starting\_price\ resale\_price\}^*$
11	*offer*	*good_id price*
12	*sold*	*good_id buyerlogin price*
13	*sanction*	*buyerlogin fine*
14	*expulsion*	*buyerlogin*
15	*collision*	*price*
16	*withdrawn*	*good_id price*
17	*end_round*	*round_number*
18	*end_auction*	*auction_number*
19	*going*	$\{single\|multiple\} + \{1, 2\}$
20	*gone*	
21	*tie_break*	*buyerlogin*
22	*closed_market*	

Tab. 2: Trading Interagent Predicates