Published in F. Dignuma and M. Greaves (Eds.) Issues in Agent Communication. Lecture Notes in Computer Science 1916, p. 249-263.

Conversation Protocols: Modeling and Implementing Conversations in Agent-Based Systems

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

Abstract

We present Conversation Protocols (CPs) as a methodological way to conceptualize, model, and implement conversations in agent-based systems. CPs can be thought of as 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). Our proposal relies upon *interagents*, autonomous software agents that mediate the interaction between each agent and the agent society wherein this is situated. Thus, Interagents employ conversation protocols for mediating conversations among agents.

1 Introduction

Interaction among agents can take 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); *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.

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[4, 27, 9, 13, 2, 17]. However, new works in speech act research, exemplified by efforts such as KAoS[6], Dooley Graphs[23], COOL[5] and MAGMA[11], attempt at representing and reasoning about the relationships within and among conversations, or groups of utterances. A number of formalisms have been proposed for modeling conversations: FSMs[5, 8], Dooley graphs[23], Petri Nets[15], etc.

In this work we present Conversation Protocols (CPs) as the methodological way to conceptualize, model, and implement conversations in agent-based systems. Our approach proposes a new model based on a special type of Pushdown Transducers (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.

We view conversations as the means of representing the conventions adopted by agents when interacting through the exchange of utterances[28, 5] —"utterance suggests human speech or some analog to speech, in which the message between sender and addressee conveys information about the sender"[23]. 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.

Our proposal relies upon *interagents*[20, 19], autonomous software agents that mediate the interaction between each agent and the agent society wherein it is situated. Interagents employ conversation protocols for mediating conversations

among agents.

Interagents are responsible for posting utterances of its *customer*[1] to the corresponding addressee and for collecting the utterances that other agents address to its customer. Each interagent has 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. 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.

The remainder of this article is organized as follows. Section 2 introduces a conceptual model of CPs. Next, in Section 3 the formalism underpinning our model is presented. Section 4 explains the way of instantiating CPs. Next, in Section 5 we introduce the notion of CP compatibility, in order to ensure the correct exchange of utterances during a conversation. In Section 6 we describe two ways of negotiating the attributes of a CP instance. Finally, Section 7 presents some concluding remarks.

2 Conceptual Model

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[24];
- 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[3, 7].

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: i) analogously to other finite-state devices a few fundamental theoretical basis make PDTs very flexible, powerful and efficient [24]; ii) 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.; iii) they offer a straightforward mapping from specification to implementation; iv) PDTs, unlike other finite state devices, allow us to store, and subsequently retrieve, the contextual information of ongoing conversations; and, finally, v) the use of pairs of symbols to label arcs adds expressiveness to the representation of agent conversations.

Conceptually, we decompose a CP into the following elements (see Figure 1): 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. The input list is continuously traversed by the interagent in search of a (p/d) pair (where p stands for a performative, and d stands for a predicate)

[1] We call customer the agent exploiting and benefiting from the services offered by an interagent

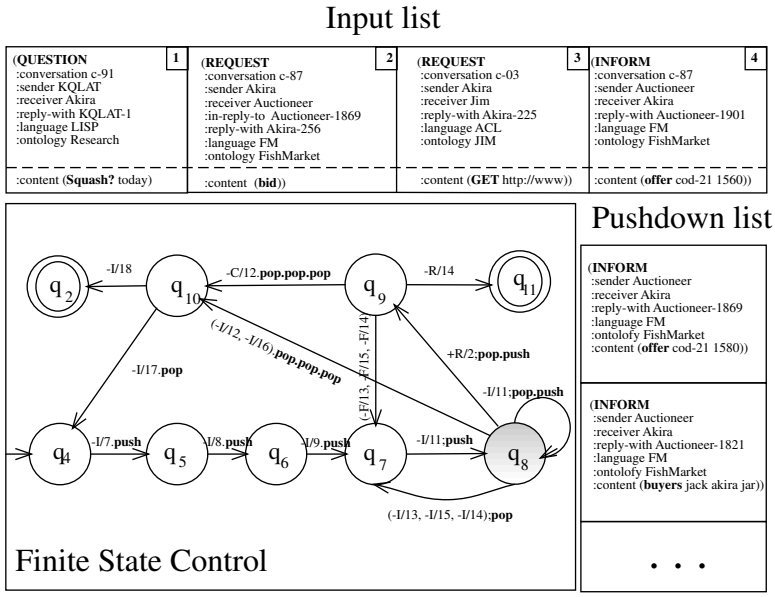

Fig. 1: Partial view of the CP DBP used by trading interagents in the Fishmarket.

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 Section5.

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.

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. Moreover, for a given CP c we define its *symmetric view* $\overline{c}$ 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 label an arc, $p_1/d_1; Z|op, .., p_n/d_n; Z|op$, they can be

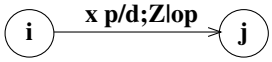

Fig. 2: Transition

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 a 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.

For instance, Figure 1 depicts the CP employed by the interagent used by a buyer agent to bid for items presented by an auctioneer agent who calls prices in descending order —the downward bidding protocol (DBP). Each item is adjudicated to the buyer that stops the descending sequence of prices called by the auctioneer following the rules of the DBP implemented in FM[25]. 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. Such performatives and predicates belong to the FM communication language and ontology. 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.

3 Formal Definition

Now it's time to formally capture the conceptual model introduced above in order to be later able to reason 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 $Q\times\{+,-\}\cdot\Sigma_1\times\Sigma_2\times\Gamma^*$ 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 moves depend on the current state of the finite state control, the performative/predicate pair of a message into the input list, and the state of the pushdown list. For instance, the move expressed by the following transition: $\delta(q_8, +\mathsf{REQUEST}, \mathsf{bid}, \mathsf{offer}Z) = \{(q_9, \mathsf{bid}Z)\}$ 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.

e-moves These moves 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. For instance, the

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[9] (extracted from Parunak[23])

#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 Predicates

following transition $\delta(q_9, e, e, \mathsf{bid}Z) = \{(q_8, Z)\}$ could allow a trading interagent to roll back to a previous state.

It should be noted that 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 $|\delta(q, p, d, Z)| \leq 1$.

4 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 its addressee, 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 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.

From the point of view of interagents the conversations requested to be held by its customer agent 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 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.

4.1 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; l represents all utterances currently kept by the input list; and α is the current state of the pushdown list.

Figure 1 depicts a partial 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 1 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.

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 [29], 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[29]. 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 con-

trols 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 2.

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 3 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.

6 Conversation Protocol Negotiation

In Section 4 we introduced the 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 sequence in Figure 4 depicts a typical exchange for this model of negotiation, where q values indicate the degree of preference over each proposal.

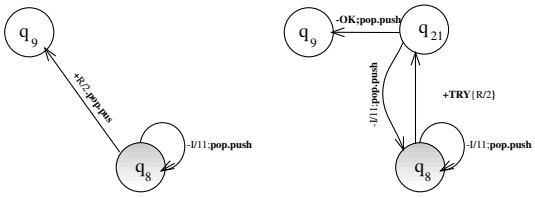

Fig. 3: Augmented CP instance

```
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
```

Fig. 4: One-step negotiation. In this example the helper accepts the second proposal from the originator.

In two-step negotiation (see Figure 5), instead of directly accepting or refusing the originator proposal, the helper can reply with a a list of counter-proposals ranked according to its own preferences. Then, the originator can either accept one of these proposals or cancel the process.

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 (e.g. the CP/DBP for the downward bidding protocol), analogously to MIME content types (text/html, image/gif, etc.). On the other hand, it makes no sense 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.

7 Final Remarks

We have introduced conversation protocols as the methodological way to conceptualize, model, and implement conversations in agent-based systems. We have also explained the way of negotiating, instantiating and ensuring compatibility semantics of CPs. CPs allow to impose a set of constraints on the communicative acts uttered by the agents holding a conversation. Other finite state models have been largely used for network protocols, and subsequently adapted to speech act theory. Though valid for specifying the states through which a conversation may progress, they lack of mechanisms for maintaining contextual information valuable for tracking the evolution of the conversation in time. CPs, on the contrary, allow to store the context of an ongoing conversation.

Our proposal relies upon *interagents*[19], 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. We think that CPs together with interagents constitute a convenient *infrastructure* for easing the development of agent-based systems. 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 details, and so they can concentrate on the design of 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 [16, 14]).

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 [21]. Although there is a large number of software tools for developing agents[1], not many of them happen to provide support for the specification of conversation protocols. AgentTalk[2], COOL[5], JAFMAS

[2] http://www.cslab.tas.ntt.jp/at/

```
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
```

Fig. 5: Two-step negotiation. In this example the helper refuses the proposals of the originator, who finally accepts the first helper's counterproposal.

[8], Agentis[12], Jackal[10] and InfoSleuth[22], 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[22]. We attempt to make headway in this matter with respect to other agent building tools by introducing interagents, that permit both the dynamic and incremental definition of conversation protocols. We have chosen such conceptualization 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.

JIM has been successfully applied in the development of FM[3][25, 26], our current implementation of an agent-mediated electronic auction market. Additionally, JIM is being successfully employed by other ongoing research projects: the SMASH[4] project, that addresses the construction of prototype multi-agent systems with case-based reasoning capabilities that cooperate in the solution of complex problems in hospital environments; and in the multi-agent learning framework Plural[18] 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 and the COMRIS project, ESPRIT LTR 25500; Juan A. Rodríguez-Aguilar and Francisco J. Martin enjoy DGR-CIRIT doctoral scholarships FI-PG/96-8490 and FI-DT/96-8472 respectively.

[3] http://www.iiia.csic.es/Projects/fishmarket

[4] http://www.iiia.csic.es/Projects/smash/

References

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

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

[3] 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.

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

[5] 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.

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

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

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

[9] 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.

[10] 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.

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

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

[13] 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.

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

[15] Jean-Luc Koning, Guillaume François, and Yves Demazeau. Formalization and prevalidation for interaction protocols in multi-agent systems. In *13th European Conference on Artificial Intelligence*, 1998.

[16] 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.

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

[18] 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.

[19] Francisco J. Martin, Enric Plaza, and Juan A. Rodriguez. An infrastructure for agent-based systems: an interagent approach. 1999. To appear in International Journal of Intelligent Systems.

[20] Francisco J. Martin, Enric Plaza, Juan A. Rodriguez-Aguilar, and Jordi Sabater. Java interagents for multi-agent systems. In *AAAI-98 Workshop on Software Tools for Developing Agents*, 1998.

[21] Francisco J. Martin, Enric Plaza, Juan A. Rodriguez-Aguilar, and Jordi Sabater. Jim: A java interagent for multi-agent systems. In *1r Congrés Català d'Intel.ligència Artificial*, pages 163–171, Tarragona, Spain, 1998.

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

[23] 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.

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

[25] 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.

[26] 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.

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

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

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