CENTRO PER LA RICERCA
SCIENTIFICA E TECNOLOGICA

38050 Povo (Trento), Italy
Tel.: +39 0461 314312
Fax: +39 0461 302040
e−mail: prdoc@itc.it − url: http://www.itc.it

INTRA−ROLE COORDINATION USING CHANNELED MULTICAST

Busetta P., Merzi M., Rossi S., Legras F.

March 2003

Technical Report # 0303−02

PEACH:

Personal Experience with Cultural Heritage

Intra-Role Coordination Using Channeled Multicast

Paolo Busetta
Mattia Merzi
Silvia Rossi
François Legras

Title:	**Intra-Role Coordination Using Channeled Multicast**
Version:	**1.0**
Date:	**April 2003**
Contact:	**Busetta, Merzi, Rossi, Legras**
Work Package:	**WP1**

Table of Content

Abstract

We propose group communication for agent coordination within "active rooms" and other pervasive computing scenarios featuring strict real-time requirements, inherently unreliable communication, and a large but continuously changing set of context-aware autonomous systems. Messages are exchanged over *multicast channels*, which may remind of chat rooms where everybody hears everything being told. The issues that have to be faced (e.g., changing users' preferences and locations; performance constraints; redundancies of sensors, actuators, and services; evolving sources of information; the continuous joining and leaving of agents on mobile devices) require the ability of dynamically selecting the "best" agents for providing a service in a given context. Our approach is based on the idea of *implicit organization*, which refers to the set of all agents willing to play a given role on a given channel; an implicit organization is a special form of team with no explicit formation phase and a single role involved. No middle agent is required by an implicit organization; instead, a set of protocols, designed for unreliable group communication, are used for two purposes: first, to negotiate a coordination policy; second, for actual team coordination. Preconditions and effects of these protocols are formalized by means of the joint intention theory (JIT). We sketch a general computational model for an agent participating to an implicit organization.

1 Introduction

So-called "active rooms" or "active environments" are pervasive computing scenarios providing some form of "ambient intelligence", i.e. some form of automatic, sophisticated assistance to humans performing physical or cognitive tasks by specialized devices present in the same place. Active environments often feature a large and continuously changing set of context-sensitive, partly mobile autonomous systems; thus, a multi-agent architecture seems to be a natural choice. As an example, our current domain of application is interactive cultural information delivery within museums or archeological sites, for which we develop multi-user, multi-media, multi-modal systems. Agents, which are distributed on both static and mobile devices, guide visitors, provide presentations, supervise crowds, and so on, exploiting whatever sensors and actuators are close to the users during their visit. The agents must immediately adapt to changing focus of attention or movements of users, in order not to annoy them with irrelevant or unwanted information; to make things even harder, wireless networks (required to support mobility) are intrinsically unreliable for a number of reasons. Consequently, agents have to deal, in a timely and context-dependent way, with a range of problems that include unexpectedly long reaction times by cooperating agents, occasional message losses, and even network partitioning.

We propose a form of group communication, called *channeled multicast* [3], as the main technique for agent coordination in ambient intelligence scenarios. Channeled multicast often reduces the amount of communication needed when more than two agents are involved in a task, and allows overhearing of the activity of other agents. Overhearing, in turn, enables the collection of contextual information, pro-active assistance [2], monitoring [12], even partial recovery of message losses in specific situations, and is exploited by the protocols described in this paper. Our current implementation of channeled multicast is based on IP multicast, thus it features almost instantaneous message distribution on a local area network but suffers from occasional message losses.

Objective of this paper is to describe an agent coordination technique we are exploring and provide its initial formal ground. Specifically, we define a general set of social conventions, formalized with the Joint Intention Theory [5], for establishing and enforcing a coordination policy among agents playing the *same* role; this addresses some issues related to redundancies as well as adaptation to the context without the intervention of middle agents. Protocols are designed to work under the assumption of unreliable communication. From the social conventions and the protocols, we outline a generic computational model for the agents, in which coordination and task-specific capabilities are kept separated as much as possible.

This paper is organized as follows. Next section provides preliminary background information. Sec. 3 introduces the concept of *implicit organization*, that is, a team of agents coordinating to play a role on a channel. Sec. 4 describes the interaction between agents and implicit organizations. The behaviour of an organization is formally described in Sec. 5. The following three sections discuss how organizations decide their own coordination policies, describe a few ones, and show some examples (Sections 6, 7, and 8 respectively). Some architectural considerations and a computational model are given in Sec. 9. Sec. 10 compares works available in the literature with ours. We conclude with final remarks and future directions (Sec. 11).

2 Background

2.1 Reference scenario

Pervasive computing is a very active and promising research domain that brings new challenges to the agent community. The PEACH project [17] aims at creating "active museums", i.e. smart environments featuring multimodal I/O where a visitor would be able to interact with her environment via various sensors and effectors, screens, handheld devices, etc. This application domain provides a very challenging environment: agents can come and go dynamically, visitors move about the museum (potentially carrying handheld devices), communication media are heterogeneous and unreliable (a mix of wired and wireless networks

like WiFi or Bluetooth are likely). An important consideration is that, if the system does not react timely to the movements and interests of the visitors, they will simply ignore it or, worse, will become annoyed.

Here is a typical problem that such a system should be able to solve: a visitor is supposed to receive a multimedia presentation on a particular subject, but (1) several agents are able to produce it with different capabilities (pictures + text or audio + video) and variable availabilities (CPU load or communication possibilities) and (2) the visitor is close to several actuators (screens, speakers) each able to "display" the presentation, and the visitor carries a PDA which is also able to display it, albeit in a more limited way. In addition, several other visitors are nearby potentially using the actuators, and of course if the visitor does not get a presentation after a few seconds she will leave.

2.2 Settings and assumptions

We deal here with cooperation with unreliable communication and highly dynamic environment. We do not address the well-known problems of task decomposition or sub-goal negotiation. We assume that once agents are set to execute a task they know how to do it. Rather, we aim at achieving robustness and tolerance to failure in a setting where agents can be redundant, communication is unreliable, hardware can be switched off, etc. Such an environment can evolve faster than the agents execute their task, so it is not feasible to use "traditional" techniques e.g. guided team selection [24] or shared planning [10].

Our objective is to avoid centralized or static solutions like mediators, facilitators or brokers, but rather to have a fully distributed and flexible system, without looking for optimality.

We use a multicast communication infrastructure LoudVoice [3] to support most of the communication needs of the agents. LoudVoice uses the fast but inherently unreliable IP multicast and XML for message encoding. Multicast possesses an inherent unreliability, but our communication media is unreliable by nature (possibly including WiFi for example). LoudVoice is language-independent: we currently have a Java implementation of the API and a beta version of the C++ porting that runs both on handheld devices and PCs. LoudVoice identifies channels in three different ways: with an identifier (a name), with a theme (i.e., the main subject of conversations on the channel), and with a multicast IP communication address. A theme is just a list of strings taken from an application-specific taxonomy of subjects; this taxonomy is represented as an XML file accessible to all agents via its URL, and is used by agents to discover which channels are available. Having discovered or otherwise identified one or more channel, an agent can freely "listen" and "speak" on them by means of FIPA-like messages. These are encoded as XML documents, and have a common header, which includes a performative, sender and destination. Other details on LoudVoice can be found in [3].

3 Implicit Organizations

Following a common convention in multi-agent systems, we define a *role* as a communication-based API, or abstract agent interface (AAI), i.e. one or more protocols aimed at obtaining a cohesive set of functions from an agent. A simple example is mentioned in [3], an auction system with two main roles: the auctioneer (which calls for bids, collects them and declare the winner) and the bidder (which answers to calls for bids and commits to perform whatever transaction is requested when winning an auction). An agent may play more than one role, simultaneously or at different times depending on its capabilities and the context.

We adopt the term *organization* from Tidhar [25], to refer to teams where explicit *command*, *control*, and *communication* relationships (concerning team goals, team intentions, and team beliefs respectively) are established among subteams.

We call *implicit organization* a set of agents tuned on the same channel to play the same role and willing to coordinate their actions. The word "implicit" highlight the facts that there is no group formation phase (joining an organization is just a matter of tuning on a channel), and no name for it – the role and the channel uniquely identify the group, indeed. It is important to stress that, in this work, we focus on implicit

organizations formed by agents all able to play the same role but possibly in different ways – redundancy (as in traditional fault tolerant or high capacity, load-balanced systems) is just a particular case where all agents are perfectly identical. This situation is commonly managed by putting a broker or some other form of middle agent supervising the organization. By contrast, our objective is to explore advantages and disadvantages of an approach based on unreliable group communication, in a situation where agents can come and go fairly quickly, their capabilities can change or evolve over time, and it is not necessarily known a-priori which agent can achieve a specific goal without first trying it out.

An implicit organization is a special case of team. Generally speaking, a team include different roles, and is formed in order to achieve a specific goal; as said above, an implicit organization includes all agents for a role on a channel at any given time. Goals for an implicit organization are automatically established by requests to achieve something and queries addressed to its role. In Tidhar's terms, this is to say that a command relationship is established between any agent performing a goal-establishing communicative action (the "commanding agent") and the implicit organization, whose consequence is that the latter is committed at achieving the goal. Section 4 below discusses the corresponding protocol.

An implicit organization is in charge of defining its own control policy, which means: (1) how a sub-team is formed within the organization in order to achieve a specific goal; and, (2) how the intentions of this sub-team are established. For this initial work, a goal-specific sub-team is fixed to be simply *all agents that are willing to commit immediately at achieving the organizational goal at the time this is established*; i.e., there is no explicit sub-team formation, rather introspection by each agent to decide whether or not it has enough resources immediately available. A *coordination policy* established for the organization is then used within a sub-team to decide who actually works toward achieving its goal, and possibly to coordinate the agents if more than one is involved.

We assume that policies are well-known to agents; Section 7 describes some policies we use in our domain, and our computational model (Sec. 9.2) assumes the existence of a library of application-independent policies. Thus, it is possible to refer to a policy simply by its name. A policy, however, may have parameters that need to be negotiated before use – for instance, the currency used for auctions, or the master agent in a master-slave environment. We call *policy instance* a tuple composed of a policy name and the ground values for its parameters. Section 6 discusses how a policy instance is negotiated and established as organizational coordination policy.

Note that a goal-specific sub-team may well be empty, e.g. when all agents of the implicit organization are busy or simply no agent is part of the organization. With unreliable communication setting, this case is effectively undistinguishable from the loss of the goal-establishing message (unless overhearing is applied; this will be the objective of future works) or even from a very slow reaction; consequently, it must be properly managed by the commanding agent. These considerations have an important impact on the protocol between commanding agents and implicit organizations, as discussed below.

4 Role-based communication

In this initial work, we assume that any request – by which we mean any REQUEST and QUERY, using the FIPA performatives [9] – becomes a commitment by an implicit organization to perform the necessary actions and answer appropriately (strategic thinking by the organization is left to future work). Thus, in principle the interactions between commanding agents and implicit organizations are straightforward, and can be summarized in the simple UML sequence diagram of Fig. 1. A generic *Requester* agent addresses its request to a role *R* on a channel; the corresponding implicit organization replies appropriately. As mentioned above, however, unreliable channels, continuous changes in the number of agents in the organization, and strict real-time constraint, substantially complicate the picture. Fig. 2 is a finite state machine that captures, with some simplifications, the possible evolutions of the protocol. The events on top half represent the normal case - no message loss, a goal-specific subteam achieves the goal. The events in brackets on the lower half represent degraded cases.

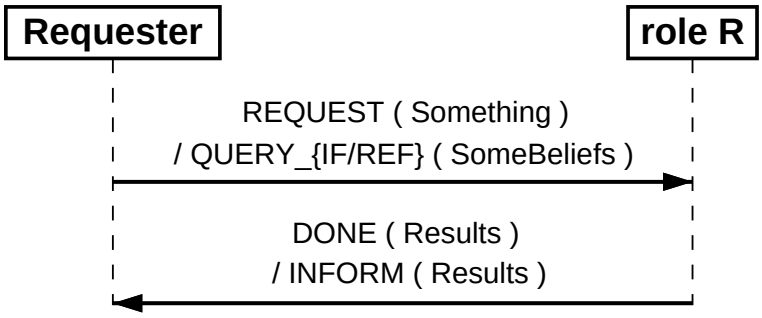

Figure 1: A role-based interaction

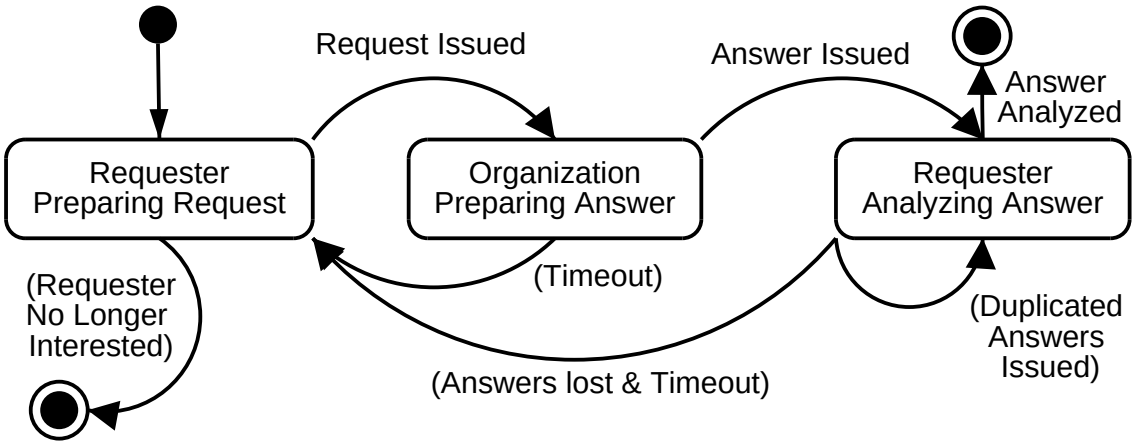

Figure 2: Interacting with unreliable communication – a simplified protocol machine

Consider, for instance, the cases where a request or its answer are lost, or no goal-specific sub-team can be formed. A timeout forces the commanding agent to reconsider whether its request is still relevant – e.g., the user's context has not changed – and, if so, whether to resend the original message. It follows that an implicit organization must be able to deal with message repetitions. This implies that agents should discard repeated messages or re-issue any answer already sent; also, the coordination policies should contain mechanisms that prevent confusion in the cases of partial deliveries or new agents joining the organization between two repetitions.

Similarly, the commanding agent has to deal with repeated answers, possibly caused by its own repeated requests. In the worse case, these repeated answers may even be different – e.g., because something has changed in the environment, a new agent has joined the organization, and so on. Rather than introducing middle-agents or making the organizational coordination protocols overly complicated to prevent this to happen (which is likely to be something impossible to achieve anyway, following [11]), our current choice is to have the requester to consider as valid whatever answer it receives first, and ignore all others.

Not even to mention, the usability of the protocol presented above is limited to non-safety critical applications, or at least to situations where it is possible to tolerate some level of uncertainty. This is definitely the case in our multi-media environments, for instance, where quality of communications is fairly high and the objective is nothing more critical than providing some guidance and context-sensitive cultural information to visitors of museums.

Observe that third parties overhearing a channel, in accordance to [2, 1], may help in making the interaction with implicit organizations much more robust – for instance, by detecting some message losses,

or whether a goal-specific sub-team has been established. We plan to explore some options in future work.

5 Formalizing the behaviour of an organization

This section provides a high-level formalization of the coordination within an implicit organization. This is done by means of a logic specifically designed for conversation policies, the Joint Intention Theory (JIT) [5, 20]. JIT was born as a follow-up of a formalization of the theory of speech acts [21]. Recently, it has been applied to group communication [13], and a particular form of diagram, composed of *landmark expressions*, has been introduced. This diagram "resembles state machines but instead of specifying the state transitions, it specifies a partially ordered set of states" [14] called *landmarks* and corresponding to JIT formulas.

Strictly speaking, JIT is not applicable to our domain because we adopt unreliable communication, but we use it anyway because it is a very convenient way to capture certain aspects of team behavior we are interested in; we return on this point in Sec. 5.1 below.

JIT is expressed as a modal language with connectives of the first order logic and operators for propositional attitudes. Greek letters are used for groups, and lowercase variables for agents.

We use the definitions from the papers mentioned above [20, 13, 14], to which we add a few new ones in order to simplify the formulas introduced later. We identify an implicit organization with the role it plays, and indicate it with ρ; $\rho(x)$ is true if x is member of ρ, which means that x is playing the role ρ on a given channel. We simplify the definition of group mutual belief given in [13], that is:

$$(MB\ \tau_1\ \tau_2\ p) \equiv (BMB\ \tau_1\ \tau_2\ p) \wedge (BMB\ \tau_2\ \tau_1\ p)$$

for the special case of "all agents of a group towards their group and viceversa":

$$(MB\,\rho\ p) \equiv \forall x\,\rho(x)\ \supset\ (BMB\,x\,\rho\ p) \wedge (BMB\,\rho\,x\,p)$$

Analogously, we extend the definitions of mutual goal (*MG*), joint persistent goal (*JPG*), and introduce group extensions of some others from [21], as follows. For simplicity, we consider the relativizing condition q always true, thus it will be omitted in all the following formulas.

A *mutual goal* is defined as:

$$(MG\ \rho\ p) \equiv (MB\ \rho\ (GOAL\ \rho\ \Diamond p))$$

A *weak achievement goal* is:

$$\begin{aligned}(WAG\ x\ \rho\ p) \equiv\ & [(BEL\ x\ \neg p)\ \wedge\ (GOAL\ x\ \Diamond p)] \vee \\ & [(BEL\ x\ p) \wedge (GOAL\ x\ \Diamond (MB\ \rho\ p))] \vee \\ & [(BEL\ x\ \Box\neg p)\ \wedge\ (GOAL\ x\ \Diamond (MB\ \rho\ \Box\neg p))]\end{aligned}$$

that is, an agent x has a WAG toward a group ρ when it believes that p is not currently true, in which case it has a goal to achieve p, or if it believes p to be either true or impossible, in which case it has a goal to bring about the corresponding mutual beliefs.

A *weak mutual goal* is:

$$\begin{aligned}(WMG\ \rho\ p) \equiv\ & \forall x\,\rho(x)\ \supset (MB\ \rho\ (WAG\ x\ \rho\ p)\ \wedge \\ & (WAG\ \rho\ x\ p))\end{aligned}$$

where: $(WAG\ \rho\ y\ p) \equiv \forall x\,\rho(x)\ \supset\ (WAG\,x\,y\,p)$. A weak mutual goal is a mutual belief that each agent has a weak achievement goal towards its group for achieving p, and conversely the group has the goal towards its members.

A *joint persistent goal* is defined as:

$$(JPG\ \rho\ p) \equiv (MB\ \rho\ \neg p)\ \wedge\ (MG\ \rho\ p)\ \wedge \\ (UNTIL\ [(MB\ \rho\ p)\ \vee\ (MB\ \rho\ \Box\neg p)] \\ (WMG\ \rho\ p))$$

that is, a group of agents ρ has a joint persistent goal p when there is a mutual belief that p is not currently true, there is a mutual goal to bring about p, and p will remain a weak mutual goal until there is a mutual belief that p is either true, or will never be true.

The expression $(DONE\ \rho\ p)$, used in the following, is a group-extension of the definition of DONE described in [6] i.e. any agent in the group ρ believes that the expression p happened immediately before the present time.

Finally, we define $Coord(P_i\ \sigma\ r\ p)$, where P_i is a coordination policy, σ is a group of agents, r is another agent, and p is a goal, as a function that computes the sequence of actions that must be performed by σ under policy P_i in order to achieve p and notify r when done. This sequence is composed of coordination actions and sub-goals assigned to individual agents, and must terminate with a DONE or INFORM message to r on the results. Recall that, by using a channeled multicast infrastructure such as LoudVoice, everybody listening to a channel receives everything sent on it; thus, the action of sending the results to r also establishes – via overhearing – a belief in the listeners that the goal has been reached (or is not reachable).

With the definitions given above, we can now formalize the protocol that an implicit organization, having a goal p to achieve, commanded by a request from agent r to the role ρ (Section 4), has to follow. *CurrentPolicyDecided* represents whether or not the organization has established its own coordination policy, i.e. that there is a mutual belief $(CurrentPolicy\ P_i)$ where P_i is a *policy instance* (Sec. 3); thus, $CurrentPolicyDecided \equiv \exists P_i\ (CurrentPolicy\ P_i)$. σ represents the goal-specific sub-team instantiated to achieve p. The protocol is represented by the landmark diagram of Figure 3, where the landmarks correspond to the following JIT expressions:

L1 : $\neg(DONE\ \rho\ p) \wedge (JPG\ \rho\ ((DONE\ \rho\ p) \wedge (BEL\ r\ p)))$

L2 : $(MB\ \rho\ CurrentPolicyDecided)$
$\wedge(INTEND\ \sigma\ Coord(P_i\ \sigma\ r\ p))$

L3 : $(MB\ \rho\ ((DONE\ \rho\ p) \vee \Box\neg(DONE\ \rho\ p)))$

L4 : $(MB\ \rho\ \neg CurrentPolicyDecided)$
$\wedge\ (JPG\ \rho\ CurrentPolicyDecided)$
$\wedge(JPG\ \rho\ ((DONE\ \rho\ p) \wedge (BEL\ r\ p)))$

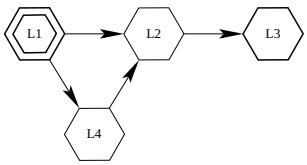

Figure 3: Coordination in an implicit organization

The protocol is started (i.e., landmark L1 is entered) when a request for p from r arrives to an implicit organization ρ, and concluded (i.e., L3 is reached) when the request is satisfied. In other words, Fig. 3 is an expansion of the state `Organization Preparing Answer` of the diagram of Fig. 2. Landmark L1

says that the requested goal has not been achieved yet, and that ρ has a joint persistent goal to achieve it and notify *r*. However, achieving *p* is possible only if the organization knows how to coordinate. If this is the case, i.e. a coordination policy has already been established, then the protocol moves to landmark L2, otherwise to L4. The transition from L4 to L2 is called *policy negotiation*, and is discussed in Sec. 6 below.

In the transitions from L1 to L2 and from L4 to L2, a sub-team σ is formed and given an intention to achieve *p* on behalf of ρ. As specified in Sec. 3, we assume that σ are all agents able to achieve *p* at the time the request arrives (i.e., when L1 is entered). In a policy-specific way, σ does whatever is required to achieve *p*. The notification of the results to *r* is overheard by everybody in ρ, thus it establishes the mutual belief within the organization that satisfies its joint goal (landmark L3).

5.1 Common knowledge, JIT, and LoudVoice

The Joint Intention Theory relies on the ability to achieve *common knowledge* [11] within a team. However, this goes against our choice of using an *unreliable* group communication infrastructure called LoudVoice, based on the principle of channeled multicast [3], within our domain. By design, LoudVoice privileges speed to reliability; to this end, it has been implemented by using multicast IP, which is an extension of IP to group communications [7]. In our settings (typically a standard LAN extended with a wireless infrastructure), we can assume practically instantaneous transport of messages on a channel with respect to the evolution of the environment, and a certain probability that – for whatever reason, including network jams and inherent unreliability of wireless links – messages can be occasionally lost and temporary network partitions happen (typically when mobile devices move through areas not covered by any signal). As shown in [11], which formalizes well-known results in distributed systems theories, with unreliable communication (as well as non-instantaneous delivery of messages), "perfect" common knowledge cannot be achieved; specifically, [11] demonstrates that synchronized action based on perfect common knowledge is not possible.

To get out of this contradiction, we exploit Morris and Shin's results on *probabilistic coordination* [16], obtained by applying game theory. In short, they show that, in *non-strategic games* – i.e. where all agents play by rules determined a-priori independently of their individual objectives and preferences – and a reasonably small probability of losing messages, it is possible to design protocols which achieve coordination (in game theory-speak, maximize payoff) with a probability arbitrarily close to the desired one. Non-strategic behaviour, pre-defined rules, and small probability of loss, are all characteristics of our domain. We designed our protocols to be robust against single message losses or the occasional network partitioning, thus giving a high level of confidence on their outcomes. Some mechanisms, such as the periodic reminders sent on the control channel about the current policy adopted by an implicit organization (Sec. 6.3), have the specific purpose of increasing that confidence by detecting problems as soon as possible.

The conclusion is that, in spite of the fact that we cannot use it to automatically derive protocols (as done by [14]), we feel confident in using JIT, thanks to the robustness of the underlying coordination protocols we defined.

6 Negotiating a coordination policy

This section formalizes the negotiation protocol for the organizational coordination policy, and describes an algorithm to be applied with unreliable group communication. In summary, the negotiation protocol works in two phases: first, the set of *policy instances* common to all agents is computed; then, one is selected.

Recall, from Sec. 3, that a policy instance is a tuple with the name of a policy and ground values for all its parameters; e.g.,

$$< auction, Euro >$$

represents an "auction" policy whose first parameter (presumably the currency) is "Euro". An important

issue with our two-phase protocol is that, potentially, an agent may support a very large (even infinite) or undetermined number of policy instances: this happens, for instance, when a parameter may take any integer value, or has to be set to the name of one of the agents of the organization. In Sec. 6.4, we define a language for expressing sets of policy instances in a compact way by means of constraints on parameters, including infinite or undetermined ranges. This language is currently limited to only two data types (integers and strings), but this is enough for our purposes and enables the two-phase negotiation protocol described in the following.

6.1 Formalization

From the perspective of an individual member, the coordination policy of an organization, independently from the specific policy instance it assumes, goes through three different states: “unknown”, “negotiating”, and “decided”.

The policy is unknown when the agent joins the organization, i.e. when it tunes on a channel to play the organization’s role; the first goal of the agent is to negotiate it with the other members. At that point, the following applies:

$$(\nexists\ P_i\ (BEL\ x\ (CurrentPolicy\ P_i)))\wedge$$
$$(INTEND\ x\ SEND(\rho, REQUEST\ (CurrentPolicy\ ?)))$$

that is, the agent does not hold any belief on the current policy, and asks to the other members about it.

Receiving a request about the current policy is interpreted by the organization as a notification that a new member has joined. This triggers the negotiation protocol, described in Fig. 4. The corresponding landmarks are:

L1 : $(JPG\ \rho\ CurrentPolicyDecided)$

L2 : $(MB\ \rho\ (CommonPolicies\ \Pi))$
$\wedge(GOAL\ o\ (MB\ \rho\ (CurrentPolicy\ \ selectFrom(o,\Pi))))$
$\wedge(JPG\ \rho\ CurrentPolicyDecided)$

L3 : $(MB\ \rho\ (CommonPolicies\Pi))$
$\wedge(MB\ \rho\ (CurrentPolicy\ P_i))$

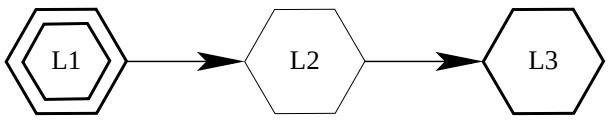

Figure 4: Landmarks for a generic policy negotiation protocol

Landmark L1 corresponds to the start of the negotiation protocol. There is a mutual belief that no policy has been currently decided within the organization. This happens, for instance, when a new agent joins - the REQUEST it sends makes the organization aware that the mutual belief on the current policy, if it had ever held before, does no longer hold and must be established (or re-established). Thus, ρ sets for itself a joint persistent goal of agreeing on the current policy.

As said above, this is done in two steps. In the first, corresponding to the transition from L1 to L2, a mutual belief about the policy instances common to the entire group is established. This is easily achieved by exchanging reciprocal INFORMs on what each agent is able to support, and intersecting their contents; the resulting set of policy instances is the mutual belief $(CommonPolicies\Pi)$.

At this stage (landmark L2), a special agent *o*, called the *oracle*, assumes the goal of establishing the *CurrentPolicy* of the role; the function $selectFrom(o, \Pi)$ is evaluated by *o* and returns a member of its input set. Thus, the second step of the protocol (the transition from L2 to L3) consists of the decision-making by the oracle and the establishment of the current policy, which can be done by simply sending an ASSERT to ρ about the value returned by $selectFrom(o, \Pi)$. This assertion, in turn, causes the agents to know that *CurrentPolicyDecided* is now true, and thus the related joint goal is automatically satisfied.

The oracle can be any agent, either a member or external to the organization (in the latter case, recall that it can overhear all messages sent on a channel, thus it can be informed on the common policies). It can apply whatever decision criteria it deems more appropriate – from a random choice, to a configuration rule, to inferring on previous policies based on machine learning, and so on. In this work, we do not elaborate on how the oracle is chosen, nor on its logic (but we will give some examples in Sec. 8). The algorithm presented below allows for any external agent (such as a network monitor or an application agent interested in enforcing certain policies) to intervene just after the common policies have been established, provides a default oracle election mechanism if an external one is not present, and handles conflicting oracles by forcing a renegotiation.

6.2 From theory to practice

As discussed in Sec. 5.1, JIT does not work with unreliable communication. A practical protocol for the decision of a policy has to take this issue into account, as well as the problem of a highly dynamic environment where agents can join and leave very quickly. To this end, the protocol implemented by the algorithm described below adds one information and a few more messages to the formal model given above.

All messages concerning policy negotiation are marked with a so-called *Negotiation Sequence Number* (NSN). A NSN is the identifier of a negotiation process, and is unique during the lifetime of an organization. NSNs form an ordered set; an `increment(nsn)` function returns a NSN that is strictly greater than its input *nsn*. In our current implementation, a NSN is simply an integer; the first agent joining an organization sets the NSN of the first negotiation to zero.

Goal of the NSN is to help in guaranteeing coherency of protocols and integrity of mutual beliefs in the cases of message losses, network partitioning, and new agents joining mid-way a negotiation. As described in the algorithm below, messages related to policy negotiation are interpreted by an agent depending on the NSN. Messages containing an obsolete NSN are simply discarded, possibly after informing the sender that it is out of date. Messages that have a NSN newer than the one known to the agent are also ignored but cause the agent to enter into a negotiating state. Only messages whose NSN is equal to the one known are actually handled.

The protocol is made robust in two other ways. First, the reciprocal INFORMs on the supported policies (transition from L1 to L2) are changed into repeated INFORMs on the known common policies and participating agents. Second, a *policy reminder* message, consisting of an INFORM on what is believed to be the current policy, is periodically sent by each agent after the end of the negotiation, allowing recovery from the loss of the oracle announcement and consistency checking against other problems.

Note that a network partitioning may cause two sub-organizations, each living on its own partition, to increment their NSN independently; when the partitions are rejoined, the integrity checks on the NSNs described above cause the organization to re-negotiate the policy.

A member of an organization can be modeled as having three main beliefs directly related to policies: the supported policies, the common policies, and the current policy.

As its name suggest, the supported policies belief contains all the policy instances that an agents supports:

$$SupportedPolicies(\rho, \{P_1 \dots P_k\})$$

where ρ is the role and $\{P_i\}$ is a set of policy instances.

The common policies belief contains a set of agents participating to a negotiation (identified by its NSN) and the policies supported by everybody, i.e. the intersection of their supported policies:

$$CommonPolicies(\rho, NSN, \{P_1 \dots P_k\}, \{A_1 \dots A_i\})$$

where ρ is the role and $\{P_i\}$ is the intersection of all the policy instances supported by agents $\{A_1 \dots A_i\}$.

The current policy is the policy instance decided by the oracle for a given negotiation (identified by its NSN):

$$CurrentPolicy(\rho, NSN, P_k)$$

6.3 The algorithm

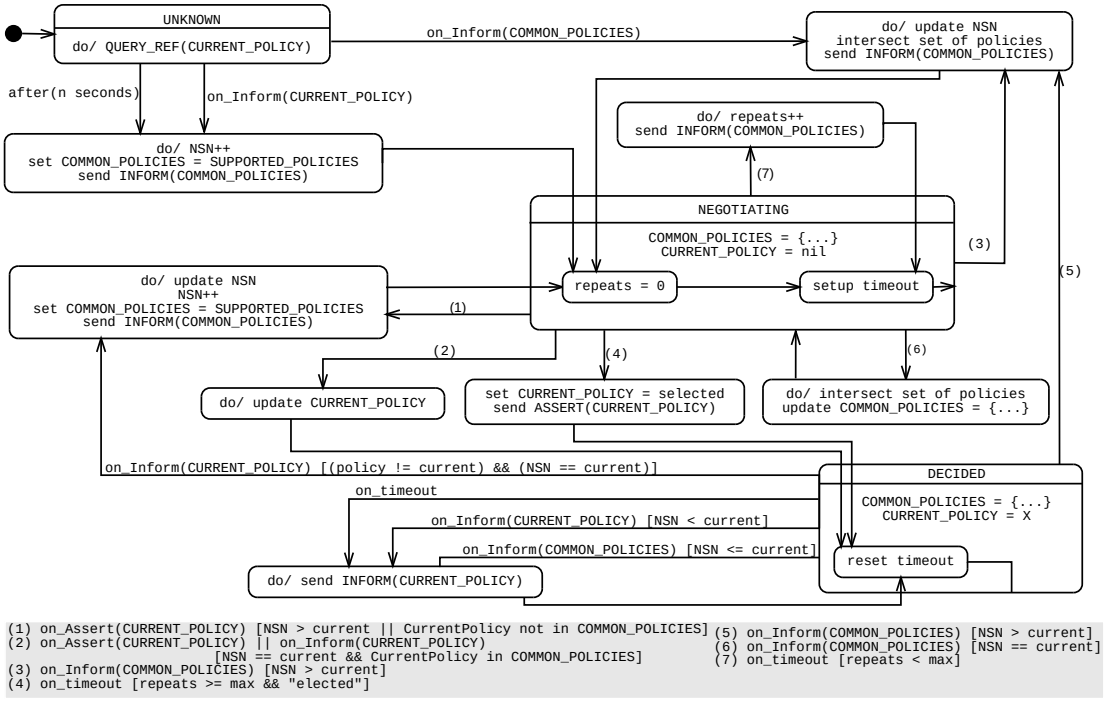

Figure 5: Behaviour of a negotiating agent - UML state diagram

This section describes, as pseudo-code, the negotiation algorithm performed by each agent of an implicit organization. The same algorithm is illustrated as a state machine in Figure 5.

In the following, we assume that an agent is a simple event-driven rule engine. We adopt some obvious syntactical conventions, a mixture of generic event-action rules, C (for code structuring, $==$ for testing equality and $!=$ for diversity) and Pascal (variable declarations as `name: type`). The `suspend` primitive blocks the execution of the calling rule until its input condition (typically a timeout) is satisfied; during the time of suspension, other rules can be invoked as events happen. A few primitives are used to send FIPA messages (INFORM, ASSERT, REQUEST) to the role for which the policy is being negotatied; for simplicity, we assume that the beliefs being transmitted or queried are expressed in a Prolog-like language. For readability, the role is never explicitly mentioned in the following, since the algorithm works for a single role at the time.

As described in Sec. 6.2 above, each agent has three main beliefs related to the policies: the set of supported policies, the set of common policies, and the current policy, which is the result of the negotiation.

The supported policies set is provided for each role played by an agent, typically by the agent developer as described in the computational model (Sec. 9.2).

The following global variables are used by the algorithm. They correspond (in a simplified form) to the three main beliefs of an agent described above, plus information concerning the negotiation process.

```
negotiationState: {UNKNOWN, NEGOTIATING, DECIDED};

commonPolicies: set of Policy;
negotiatingAgents: set of Agent_Identifier;

supportedPolicies: set of Policy;

currentPolicy: Policy;

myNSN: Negotiation_Sequence_Number;
myself: Agent_Identifier;
```

When the agent starts playing a role, it needs to discover the current situation of the organization, in particular its current Negotiation Sequence Number (NSN). This is done by sending a query to the role about the current policy. If nothing happens, after a while the agent assumes to be alone, and forces a new negotiation to start; potential messages losses are recovered during the rest of the algorithm.

```
on_Start() {
  set negotiationState = UNKNOWN;
  set myNSN = MIN_VALUE;
  set myself = getOwnAgentIdentifier();
  REQUEST (CURRENT_POLICY(?,?));
  suspend until timeout;
  if ((currentPolicy == nil)
      AND (negotiationState == UNKNOWN)) {
    negotiate( increment(myNSN),
               supportedPolicies,
               set_of (myself) );
  }
}

on_Request ( CURRENT_POLICY (input_NSN, input_policy) ) {
  if ((input_NSN == ? ) AND (input_policy == ? )
      AND (currentPolicy != nil))
       INFORM ( CURRENT_POLICY(myNSN,currentPolicy) );
  }
  else
    ......
}
```

The negotiation process mainly consists of an iterative intersection of the policies supported by all agents, which any agent can start by sending an INFORM with its own supported policies and a NSN higher than the one of the last negotiation (see `negotiate()` later on). Conversely, if the agent receives an INFORM on the common policies whose NSN is greater than the one known to the agent, it infers that a new negotiation has started, and joins it.

The iterative step consists of intersecting the contents of all INFORMs on the common policies that are received during the negotiation. If the resulting common policies set is empty, i.e. no policy can be agreed upon, the agent notifies a failure condition, waits for some time to allow network reconfigurations or agents to leave, and then restart the negotiation again.


```
on_Inform ( COMMON_POLICIES ( input_NSN,
                 input_policies, input_agents ) ) {
    if (input_NSN > myNSN)
      negotiate ( input_NSN,
          intersect (supportedPolicies, input_policies),
          union (input_agents, set_of(myself)) );
    else
    if ((input_NSN == myNSN) AND
        (negotiationState == NEGOTIATING)) {
      commonPolicies =
        intersect (commonPolicies, input_policies);
      negotiatingAgents =
        union (negotiatingAgents, input_agents);
      if (commonPolicies == {empty set}) {
        {notify failure to user};
        suspend until timeout;
        negotiate( increment(myNSN), supportedPolicies,
                   set_of (myself) );
      }
    }
    else
    if (negotiationState == DECIDED)
      INFORM (CURRENT_POLICY (myNSN,currentPolicy));
    else
    if (negotiationState == NEGOTIATING)
      INFORM(COMMON_POLICIES (myNSN, commonPolicies,
                              negotiatingAgents));
}
```

A negotiation is normally started by the agent setting the common policies to those it supports, unless it joins a negotiation started by somebody else (see above). No matter the initial parameters, during the first phase of a negotiation the agent informs the channel about the policies it knows as common, then waits for a period, during which it collects INFORMs from the other members of the organization, as described above. This process is repeated for (at most) `max_repeats` times, to allow recovery of any lost message by having agents repeating more than once what they know about the common policies. Of course, `max_repeats` depends on the reliability of the transport media in use for the channels: if the reliability is very high, the `max_repeats` value is low (two or three). The set of negotiating agents, which is not exploited by this algorithm, may be used in future for a more sophisticated recovery (e.g., by comparing the incoming set of agents with those whose messages have been received).

After `max_repeats` repetitions, the set of common policies should correspond to the intersection of all policies accepted by all negotiating agents. It is now time to decide a policy, taken from the set of those supported by everybody. This is done either by an oracle, as mentioned in the previous section, or – if no oracle is present – by an agent chosen with an arbitrary heuristic. Below, we choose the agent with the lowest identifier (after checking the NSN, to prevent confusion when negotiate() is called recursively by a new negotiation starting in the middle of another). The self-nominated oracle can apply whatever criteria it prefers to pick one policy; here, we use a simple random choice.

```
procedure negotiate (negotiation_NSN: NSN,
           initial_policies: set of Policy,
           initial_agents: set of Agent_Identifier ) {
  set negotiationState = NEGOTIATING;
  set myNSN = negotiation_NSN;
  set commonPolicies = initial_policies;
  set negotiatingAgents = initial_agents;
  currentPolicy = nil;
  INFORM (COMMON_POLICIES (myNSN, commonPolicies,
```


```
                              negotiatingAgents));
  repeat max_repeats times {
    suspend until timeout;
    if (currentPolicy != nil)
        break;   /// out of the 'repeat' block
    INFORM (COMMON_POLICIES (myNSN, commonPolicies,
                      negotiatingAgents));
  }
  suspend until (timeout OR currentPolicy != nil);
  if ((currentPolicy == nil) AND
      (myNSN == negotiation_NSN) AND
      (LowestId (negotiatingAgents) == myself)) {
    set currentPolicy = random_choice(commonPolicies);
    ASSERT (CURRENT_POLICY(myNSN,currentPolicy) );
  }
  while (currentPolicy == nil) {
   suspend until (timeout OR currentPolicy != nil);
  }
  set negotiationState = DECIDED;
}
```

When an ASSERT of the current policy is received, the agent does a few checks to detect inconsistencies – for instance, that the NSN does not refer to a different negotiation, or that two independent oracles have not attempted to set the policy. If everything is fine, the assertion is accepted, causing `negotiate()` to finish (see above). Otherwise, the assertion is either refused, or triggers a new negotiation.

```
on_Assert( CURRENT_POLICY ( input_NSN, input_policy ) ) {
  if (input_NSN == myNSN) {
    if (((currentPolicy == nil) AND
       commonPolicies.contains(input_policy)) OR
       (currentPolicy == input_policy))
         set currentPolicy = input_policy;
    else
         negotiate ( increment(myNSN), supportedPolicies,
                      set_of (myself) );
  }
  else {
    if (myNSN < input_NSN)
       negotiate ( increment (input_NSN),
          supportedPolicies, set_of (myself) );
  }
}
```

Since the assertion of the current policy can be lost, and to prevent inconsistencies caused for instance by network partitioning and rejoining, periodically each agent reminds to the group what it believes to be the current policy. The frequency of the reminders may change depending on the reliability of the transport media. When an agent receives a remainder, it checks for consistency with what it knows – i.e., that the NSN and the policy are the same. If not, depending on the situation it may either react with an inform (to let the sender know of a likely problem and possibly re-negotiate) or by triggering a negotiation itself.

```
on_PolicyReminder_Timeout() {
  INFORM (CURRENT_POLICY (myNSN,currentPolicy));
  set policy_reminder_timeout = getPolicyReminder_Timeout();
}

on_Inform(CURRENT_POLICY (input_NSN, input_policy)) {
```


```
    if (input_NSN > myNSN)
      negotiate( increment(input_NSN),
          supportedPolicies, set_of(myself));
    else
    if (input_NSN < myNSN) {
       if (negotiationState == DECIDED))
         INFORM (CURRENT_POLICY (myNSN, currentPolicy));
    }
    else  /** that is, input_NSN == myNSN **/
    if ((currentPolicy == nil) AND
          commonPolicies.contains(input_policy))
         set currentPolicy = input_policy;
    else
    if (input_policy != currentPolicy)
      negotiate( increment(myNSN),
          supportedPolicies, set_of(myself));
    }
}
```

Finally, when an agent leaves the channel, it has a social obligation to start a new negotiation process, to allow the others to adapt to the new situation. This is done by triggering the negotiation process with an INFORM about the common policies, with an incremented NSN and the policies set to a special value any which means "anything is acceptable" (see Sec. 6.4 below).

```
on_Leave() {
  increment(myNSN);
  INFORM(COMMON_POLICIES(myNSN,set_of("any"),{empty set}));
}
```

6.4 A language for defining policy instances

The policy negotiation protocols works on intersecting sets of policy instances. A policy may have one or more parameters, whose values often depends on the context (e.g., electing an agent among those available, selecting the currency for auctions).

For this initial work, we have designed a relatively simple language which allows the definition of constraints on, and suggested values for, policy parameters. An expression in this language indicates a (possibly infinite) *set* of policy instances. *Merging* two expressions generates one or more expressions corresponding to the union of the two sets. *Reducing* two expressions generates zero or more expressions corresponding to the intersection of the two sets. We exploit the associative property of the union operation on sets ($A \cup \{ b_1..b_n \} = \cup_{i=1}^{n} (A \cup \{ b_i \})$) and the distributive property of intersection ($A \cap \{ b_1...b_n \} = \cup_{i=1}^{n} (A \cap \{ b_i \})$) for operating on policy instance sets; specifically, the set of supported policies for a role by an agent is generated by merging all expressions defined by an agent programmer for the scripts or plans for that role; the intersection of policies performed by agents during negotiation is generated by the reduction of the expressions in the sets exchanged by the protocol.

A policy instance set is expressed with the following grammar:

```
<PolicyIstance> ::= name = <PName> <ParameterList>
         | name = any
<ParameterList> ::=
           <empty>
         | param = <PName> value = <Value> <ParameterList>
         | param = <PName> constraint = <ConstrExpr>
                           suggested = ( <Enumeration> )
                           <ParameterList>
```


```
<ConstrExpr> ::=
           any
         | ( one of <Enumeration> )
         | ( not one of <Enumeration> )
         | ( in <Range> )
<Range> ::= integer..integer
<Enumeration> ::= <Value> | <Value>, <Enumeration>
<PName> ::= string
<Value> ::= integer | string
```

The any policy name is used to indicate *all* possible policy instances, and should be used by agents leaving, or external to, an organization to trigger a policy negotiation in which they are no longer involved. The `param=P value=V` statement is a shorthand for `param=P constraint=(one of V) suggested=(V)`. Examples of policy instance statements are:

```
name = SimpleAuction param = Currency value = Dollar
name = SimpleAuction
    param = Currency costraint = (one of Dollar, Euro)
                    suggested = (Euro)
name = SimpleAuction param = Currency value = Euro
    param = MinBid constraint = (in 100..1000 )
                  suggested = (500)
    param = MaxBid constraint = (in 1000..2000 )
                  suggested = (1500)
name = MasterSlave
    param = Master constraint = (not one of A1, A2)
                  suggested = (A3, A4)
```

The `constraint` expression determines the possible values that a parameter can take. For now, only scalars are admitted (but note that ranges can be expressed somehow, by means of two `max` and `min` parameters). `One of` and `in` have fairly obvious semantics. `any` means that anything is accepted; `not one of` is like `any` but with the exclusion of some specific values. The suggested value must respect the constraint. Note that a statement where all parameters are constrained to a single value (e.g., they are all of the form `param=P value=V`) designs a single policy instance.

The `suggested` value is the value that an agent would assign to a parameter if it was the one to decide, i.e. its preferences if it has any.

Merging two policy instance sets (say A and B) means to identify one or more statements which include all and only the instances of A and B, and combines suggestions from both accordingly. The following rules apply:

1. if one of the policy names is any, the result is any;
2. if the policy names are different, the result is the two original instance sets, A and B;
3. if the parameter lists contain different parameter names, an inconsistency has been detected. For this version, the agent signals an error to its user and stops operating; sophisticated future version may attempt to reconcile different lists;
4. if A and B are equal, then the result is just one (say A);
5. if A and B have equal constraints but different suggestions for only one parameter, then the result is a policy instance set with the same constraints and the suggestion changed to the union of those in A and B;
6. if A and B differ for the constraints of only one parameter, then:

(a) if the constraint of either A or B is any, then the result is a policy instance set with any as constraint and suggestion equal to the union of those in A and B;

(b) if both are enumeration, then the result is a policy instance set with the constraint and the suggestion changed to be the union of the original;

(c) if both are negated enumeration, then the result is a policy instance set with the constraint changed to be the intersection of the original ones and the suggestion is the union of the original values;

(d) if both are ranges that do not overlap, then the result are still A and B;

(e) if both are range that overlap, then the result is a policy instance set with the union of the two ranges and suggestions;

(f) if one is a range and the other an enumeration, then the result are two policy instance sets: one with the range, the second the enumeration less the values that fall in the range (if empty, the second set is not generated). The suggestions are split accordingly;

(g) if one is a range or an enumeration and the other a negated enumeration, then the result is the negated enumeration less those values that fall in the range or enumeration (or any if no negated value is left), and the suggestion is the union of the original ones.

(h) for all other cases [are there any?], the result is still the two original statements.

7. if A and B differ for the constraints of two or more parameters, then:

   (a) if, for *all* different parameters, the constraints of A are less restrictive than the constraints of B (e.g., any, ranges and enumerations of B are included in A, and so on), the result is a single statement with the same constraints of A (the less restrictive) and the union of the suggestions;

   (b) same with A and B inverted;

   (c) otherwise, the result is still A and B.

Reducing two policy instance sets (A and B) means to identify one or more statements which include all and only the instances which are common to both A and B. This means that an intersection operation is performed on the constraints of A and B, while all suggestions that fall within the new constraint must be combined (recall that these are simply preferred choices within the set of possibilities indicated by the constraints). Thus, the following rules apply:

1. if the policy name of A is any, the result is B; same with A and B inverted;
2. if the policy names are different, the result is empty;
3. if the parameter lists contain different parameter names, an inconsistency has been detected. This is dealt with as in the of merging;
4. if A and B are equal, then the result is just one (say A);
5. if A and B have equal constraints but different suggestions for only one parameter, then the result is a policy instance set with the same constraints and the suggestion changed to the union of those in A and B;
6. if A and B differ for the constraints of only one parameter, then:

   (a) if the constraint of either A or B is any, then the result is a policy instance set with the other constraint and the suggestion equal to the union of those in A and B less values that are incompatible with the constraint;

   (b) if both are enumeration, then the result is a policy instance set with the constraint and the suggestion changed to be the intersection of the original. If the constrained set is empty, no statement is generated;

(c) if both are negated enumeration, then the result is a policy instance set with the constraint changed to be the union of the original and the suggestion is the union of the original values less those in the new enumeration (an empty suggestion may be generated);

(d) if both are ranges that do not overlap, then the result is empty;

(e) if both are ranges that overlap, then the result is a policy instance set with the intersection of the two ranges, and the suggestions the union of the original less those that fall outside the range;

(f) if one is a range and the other an enumeration, then the result is the enumeration without the values that fall without the range (if empty, no statement is generated). The suggestions are united and reduced as above;

(g) if one is a range and the other a negated enumeration with values in the range, then the result is a set of instance sets, each for each split of the range caused by the negated values (possibly just one statement if all negated values are outside of the range). Suggestions are united and split accordingly.

(h) if one is an enumeration and the other a negated enumeration, the result is an instance set with the enumeration less the negated values. Suggestions are treated accordingly;

(i) for all other cases [is there any?], the result is empty.

7. if A and B differ for the constraints of two or more parameters, then:

(a) if, for *all* different parameters, the constraints of A are less restrictive than the constraints of B (i.e., `any`, ranges and enumerations of B are included in A, and so on), the result is a single statement, similar to B (the most restrictive) but enlarged with those suggestions from A which fall within the constraints of B;

(b) same with A and B inverted;

(c) otherwise, the result is empty.

7 Organizational Coordination Protocols

Recall, from Sec. 3, that when an implicit organization receives a requests (i.e., the state `Organization Preparing Answer` of Fig. 2 is entered), its members with available resources form a sub-team. This happens silently, i.e. there is no explicit group formation message. As described in Sec. 6, if an organizational coordination policy has not been established yet, then one is negotiated among all members of the organization (these are the transitions from L1 to L4 and then to L2 in Fig. 3). Once the policy has been decided, the goal-specific subteam can start working.

Before describing some specific policies in use in our applications (Sec. 7.1), we define organizational coordination in abstract terms. Any coordination policy has to follow a straightforward three-phase schema, summarized in Fig. 6. Before doing anything, the sub-team coordinate to form a joint intention on how to achieve the goal (`Pre-Work Coordination`). The agents in charge perform whatever action is required, including any necessary on-going coordination (`Working`). When finally everybody finishes, a `Post-Work Coordination` phase collects results and replies to the requester (which corresponds to the `Answer Issued` event of Fig. 2).

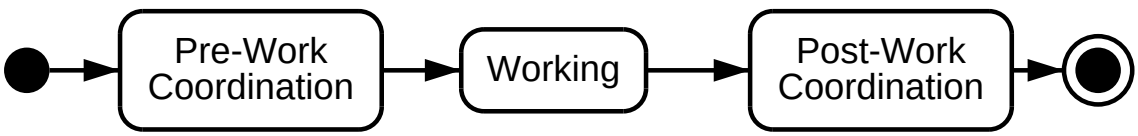

Figure 6: Abstract coordination policy, UML state diagram

The above abstract schema is summarized by Figure 7 from the perspective of a single member of the sub-team, which in turn is reflected by the general computational model presented later (Sec. 9.2). The `Working` phase of the abstract coordination policy here is expanded in two concurrent, cooperating state machines (`Work` and `Coordinate`), one performing actions toward the goal, the other coordinating with the other members of the subteam. Observe that, after `Pre-Work Coordination`, an agent may immediately stop working; this is typically the case when it is no longer involved in achieving the goal.

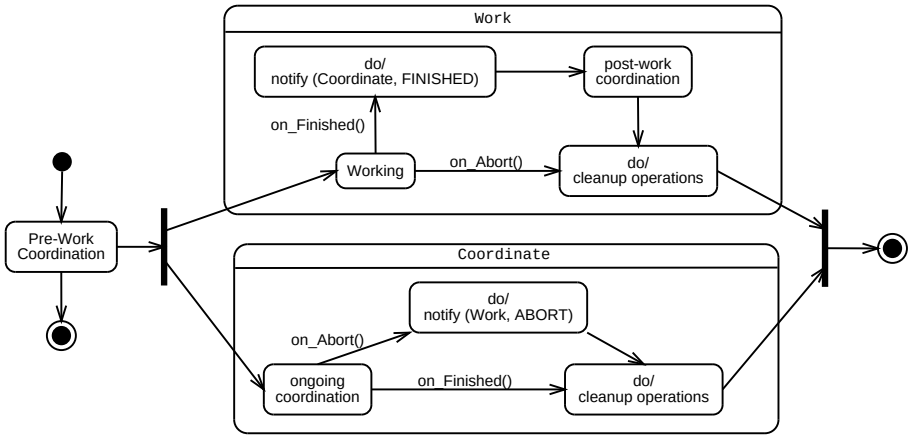

Figure 7: Abstract behaviour of a member of a goal-specific sub-team – UML state diagram

7.1 Some coordination policies

We describe here the three basic organizational coordination policy that we use in our domain. Many other variants and alternatives can be designed to meet different requirements, such as Quality of Service objectives, efficiency, and so on.

7.1.1 Plain Competition

This policy is nothing more than "everybody races against everybody else, and the first to finish wins". It is by far the easiest policy of all: no pre-work nor post-work coordination is required, while the on-going coordination consists in overhearing the reply sent by who finishes first.

In summary, the policy works as follows: when a role receives a request, any agent able to react starts working on the job immediately. When an agent finishes, it sends back its results (as an INFORM or DONE to the requester, in accordance to Fig. 1). The other agents overhear this answer and stop working on the same job; that is, with reference to Fig. 7, `Coordinate` enters `do/notify(Work,ABORT)`.

A race condition is possible: two or more agents finish at the same time and send their answers. This is not a problem, as explained in Sec. 4, since the requester must accepts whatever answer it gets first and ignore the others.

7.1.2 Simple Collaboration

This policy consists of a collaboration among all participants for synthesizing a common answer from the results obtained by each agent independently. This policy does not require any pre-work coordination; the on-going coordination consists in declaring what results have been achieved, or that work is still on-going;

finally, the post-work coordination consists in having one agent (the first to finish) collecting answers from everybody else and sending the answer to the requester.

The policy works as follows. As in Plain Competition, all agents able to react start working on the job as soon as the request is received. The first agent that finishes advertises his results with an INFORM to the role, and moves to the post-work phase. Within a short timeout (a parameter negotiated with the policy, usually around half a second), all other members of the subteam must react by sending, to the role again, either an INFORM with their own results, or an INFORM that says that they are still working followed by an INFORM with the results when finally done. The first agent collects all these message and synthesize the common result, in a goal-dependent way.

As in the previous case, a race condition is possible among agents finishing at the same time, so more than one may be collecting the same results; as always, multiple answers are not a problem for the requester, and generally imply a minor waste of computational resources because of the synthesis by multiple agents. Let us stress that, to support this policy, an agent must have the capabilities both to achieve the requested goal, and to synthesize the results.

7.1.3 Multicast Contract Net

This policy is a simplification of the well-known Contract Net protocol [22], where the Manager is the agent sending a request to a role, and the award is determined by the bidder themselves, since everybody knows everybody else's bid. Thus, effectively this policy contemplates coordination only in the pre-work phase, while neither on-going nor post-work are required. This policy has three parameters: the winning criteria (lowest or highest bid), a currency for the bid (which can be any string), and a timeout within which bids must be send.

The policy works as follows. As soon as a request arrives to the role, all participating agents send their bid to the role. Since everybody receives everybody else's offer, each agent can easily compute which one is the winner. At the expiration of the timeout for the bid, the winning agent declare its victory to the role with an INFORM repeating its successful bid, and starts working.

Some degraded cases must be handled. The first case happens when two or more agents send the same winning bid; to solve this issue, as in the policy negotiation protocol, we arbitrarily chose a heuristics, that consists in taking as winner the agent with the lowest communication identifier. The second case happens because of a race condition when the timeout for the bid expires, or because of the loss of messages; as a consequence, it may happen that two or more agents believe to be winners. This is solved by an additional, very short wait after declaring victory, during which each agent believing to be the winner listens for contradictory declarations from others.

In spite of these precautions, it may happen that two or more agents believe to be the winners and attempt to achieve the goal independently. The winner declaration mechanism, however, reduces its probability to the square of the probability of losing a single messages, since at least two consecutive messages (the bid from the real winner and its victory declaration) must be lost by the others.

7.1.4 Master-Slave

This policy has many similarity with the Multicast Contract Net; the essential differences is that a master decides which agent is delegated to achieve a goal, rather than a bidding phase. The master is elected by the policy negotiation protocol. Typically, agents that support this policy either propose themselves as masters, or accept any other agent but refuse to be master themselves; this is because the master is mandatorily part of any goal-specific sub-team, i.e. it must always be able to react to any request and must have an appropriate logic for the selection of the slave delegated to achieve a goal.

The policy works as follows. On reception of a request, all agents of the subteam sends an INFORM to the role declaring their availability. The master agent (which can be one of those) collects all declara-

tions, and issue an INFORM to the role with the slave, which answers by repeating the same INFORM; message loss is recovered by the master handling a simple timeout between its declaration and the reply, and repeating the INFORM if necessary.

Of course, there is no way that two agents may believe to be slaves at the same time. It is well possible, by contrast – as with any other policy – that no agent is available to be the slave for a request.

8 Some practical examples

We elaborate on three examples, which have been chosen to show some practical implicit organizations and the usage of the policies discussed in earlier sections. Interactions will be illustrated with a simplified, FIPA-like syntax.

8.1 Collaborative Search Engines

A `CitationFinder` accepts requests to look for a text in its knowledge base and returns extracts as XML documents. For the sake of illustration, we model searching as an action (e.g., as in scanning a database) rather than a query on the internal beliefs of the agent. An example of interaction is:

```
REQUEST From: UserAssistant033
        Receiver: CitationFinder
        Content: find ( Michelangelo )
DONE    To: UserAssistant033
        Content: done (
           find (Michelangelo),
           results (
              <doc1>Michelangelo born in Italy</doc1>,
              <doc2>...</doc2>,
              <doc3>...</doct3> )
           )
```

Typically, different CitationFinders work on different databases. Any coordination policy of those presented above seems to be acceptable for this simple role. Particularly interesting is Simple Collaboration, where any agent, when done with searching, accepts to be the merger of the results; indeed, in this case, merging is just concatenating all results by all agents. Consider, for instance, the situation where CitationFinders are on board of PDAs or notebooks. A user entering a smart office cause its agent to tune on the local channel for its role; consequently, in a typical peer-to-peer fashion, any user adds her knowledge base to those of the others in the same room. This could be easily exploited to develop a collaborative work (CSCW) system. In this case, collaboration may be enforced by the CSCW agent by acting as the "oracle" during policy negotiation.

8.2 Competing Presentation Planners

The interactive museum we are developing, as other information delivery systems, need `PresentationPlanners` (*PP* for short) for generating multi-media presentations on specific topics relevant to the context where a user, or a group of users, is. For instance, a user getting close to an art object should receive – depending on its interests, profile, previous presentations she received, etc. – a personalized presentation of the object itself, of its author, possibly of related objects in the same environment or other rooms. A typical interaction would look like the following:

```
REQUEST From: UserAssistant033
```


```
        Receiver: PresentationPlanner
        Content: preparePresentationForUser (621)
DONE    To: UserAssistant033
        Content: done (
           preparePresentationForUser (621),
           results (
               file (http://workNode/user621/pres1377.ram),
               bestResolution ( 800, 600 ),
               includeVideo ( true ),
               includeAudio ( true )
               )
           )
```

Typically, a *PP* work on a knowledge base containing text, audio and video tracks. When a request arrives, the *PP* collects data on the user and her contexts, e.g. by querying roles as `UserProfiler`, `RoomLayout`. Then, it queries its own knowledge base and, if information is available, it builds a multimedia presentation, by connecting audio and video tracks, generating audio via text-to-speech systems, and so on.

A *PP* is often the leader of its own team, formed by highly specialized agents. By contrast, it is unlikely that different *PP*s collaborate – sensibly merging multi-media presentations is a hard task even for a human. Observe that, in realistic situations, redundancy of *PP*s is a necessity, e.g. to handle the workload imposed by hundredths of simultaneous visitors. Redundancy can be obtained in various ways, for instance by putting identical *PP*s working on the same knowledge bases, or by specializing them by objects, or by rooms, or by user profiles.

Given the variety of possible configuration choices, the best policies for a *PP* are *Plain Competition* and *Multicast Contract Net* based on some quality parameter; it may also be that, in well controlled situations, a *Master* can be elected (or, more likely, imposed by an oracle). Thus, the developer of a *PP* should enable a number of non-collaborative policies, which includes specifying criteria for bidding, using negated enumerations to accept a Master-Slave policy but excluding its own *PP* from becoming a master, and so on.

8.3 Choosing among Multiple Screens

Smart rooms may have multiple places where to show things to users, e.g. large screens on different walls, the users' own PDAs, computer screens dispersed on the room. Location, but also quality and logical congruence with the tasks being performed by the user, are all important factors. Also, not necessarily *one* screen is a good choice – for instance, a presentation to a group of people may be better shown on multiple places simultaneously.

For our interactive museum, we are working on `SmartBrowsers` (*SB* for short). A *SB* is an agent able to show a multi-media presentation (video and audio) and aware of its position (which may be static, if its display is wall screens, or mobile, if running on a PDA). A typical interaction looks like the following:

```
REQUEST From: UserAssistant033
        Receiver: SmartBrowser
        Content: showMultiMedia (
            user (621),
            file (http://workNode/user621/pres1377.ram),
            bestResolution ( 800, 600 ),
            includeVideo ( true ),
            includeAudio ( true )
            )
DONE    To: UserAssistant033
        Content: done (
```


```
showMultiMedia (),
results (
    completed
    )
)
```

SBs should accept a policy that allows a clear selection of one, or (better) a fixed number of agents at request time. Thus, *Plain Competition* and *Simple Collaboration* should be avoided; *Master-Slave* works, but seems unduly restrictive in a situation where *SBs* are context aware. We are working on a *Multicast Contract Net* policy where bids are computed as a single number combining screen resolution, distance from the user, impact on other people in the same room (e.g. when audio is involved); only *SBs* visible to the user from her current position, having all required capabilities, and not busy showing something else, can participate to the sub-team bidding for a multi media presentation.

9 Architectural considerations

This section examines how the protocols described have been implemented on top of our channeled multicast platform (LoudVoice, based on multicast IP), and the computational model we adopt for our agents.

9.1 Communication channels

So far, we have described all protocols as if all messages where exchanged on the same channel. However, considerations on network traffic, on separation of concerns between organizational coordination and normal application-related exchanges, and on overhearing drive us to keep channels separated.

To this end, we exploit LoudVoice's ability to create and search channels based on *themes* of conversations. We dedicate at least one channel to messages for pure coordination, including policy negotiation; we will refer to it as `control` channel. Coordination messages are addressed to a `Control` role, played by any agent participating to one or more organizations, rather than to the role being coordinated; this implies that all messages need to specify to which role and application channel they apply. Application-related requests, queries, and their answers, all playing a part in organizational coordination, are addressed to their natural destination roles on the appropriate channels created by the agents or configured by a network manager.

Note that this separation of traffic has some implications on the engineering of the agents, as discussed in next section.

LoudVoice supports two messaging styles: *extemporary*, i.e. messages are sent with no relation to any specific conversation, a style very suitable to event notifications; and *conversational*, in which messages are related to threads of conversation, which in turn need to be explicitly created, joined by interested agents, and destroyed when their objective has been reached. LoudVoice conversations are suited for short-term collaborative tasks within closed teams, and offer some programming features (such as filtering) that make message processing very efficient.

Given the nature of the protocols and the openness of implicit organizations (agents can join and leave at any time, even in the middle of a policy negotiation), coordination on `control` adopts only extemporary messaging. By contrast, we leave to application designers total freedom on the conversation style to adopt when interacting with implicit organizations. A clean design choice is to create a new conversation for each request submitted to a role, and destroy it when the first answer arrives; however, the overhead imposed may not be justified, especially for simple queries.

9.2 An agent computational model

The computational model for an agent participating an implicit organization has to support a few basic requirements, deriving from the protocols and the network architecture described above: many roles may be played simultaneously; one of these, `Control`, deals with all policy negotiations; overhearing may be required by some policies, reducing the messaging dedicated to coordination only.

The model presented in this section aims at developing re-usable coordination libraries. A more traditional approach, where coordination and goal-specific actions are mixed together, may lead to programs that are sometimes easier to analyze, but lacks the ability of reusing generic coordination, as well as goal-specific, code.

We do not dictate any specific agent architecture, either BDI [19, 18], rule engines, planning or scripting systems. However, our model assumes a basic multi-threading capability, or equivalently the ability to handle events and perform multiple procedures (or intentions, in BDI terms) concurrently but at different priority levels. Also, the scheduler must give high priority to the handlers of requests and queries, in order to give the agent the chance to decide whether or not to participate to goal-specific subteams.

We distinguish a general `control` module from application-specific code (respectively called `control` and application from now on, for brevity). Goals of `control` are: (1) handling all the tasks of the `Control` role; and (2) handling organizational coordination, in collaboration with application. `control` owns a high priority thread (or its architecture-specific equivalent) dealing with negotiations and policy reminders, thus it opens the `control` channel. The `control` module exports the current information about an implicit organization (the state of its policy – being negotiated or decided –, supported, common, and current policies) to the application code.

Interactions between `control` and application have two main purposes. The first is feeding `control` with the policies supported by the application. Typically, this happens at startup time, when the agent analyzes its own libraries of plans, rules, configuration information, or other architecture-specific artifacts, to determine which roles and which coordination policies it can support. For instance, we envisage that BDI systems will allow plan annotations, as follows:

```
handler-for-message: REQUEST to: R1 content: G(params)
supported-role: R1;
supported-policies: competition, auction (bid = f(params));
pre-conditions: ...
body: ...
post-conditions: ...
```

In this example, `supported role` and `supported policies` are used for two purposes: first, to state that the agent supports role `R1` under at least `competion` and `auction`, the latter with `bid` set to be some function f; second, to augment the preconditions checked by the BDI interpreter, so that this plan will not be applied when the current policy for `R1` is, say, collaboration. A developer, then, builds its agent by providing the plans required for the policies that she deems suitable for a specific role; when starting up, the agent checks if it has at least one plan for each request supported by the role, computes which policies it actually supports, and uses them for negotiations with the organization.

The second reason for application and `control` to interact concerns the implementation of the coordination policies, in accordance to the general state machine of Fig. 7. `control` does not directly handle any message on the application channels, however it needs to overhear at least part of the communication to be able to coordinate with other agents and give feedback to the application. Consider, for instance, `competition`, where the first `DONE` sent in reply to a request sets the mutual belief within the organization that the request has been satisfied, thus making any further effort unuseful if not harmful.

As a consequence, the general model of interaction between application and `control` outlined below is a bit convoluted, even if it is architecture independent and allows for a full separation – and thus reusability – of the two:

1. a `REQUEST` or `QUERY` is handled by the application, which decides whether or not to participate to the subteam achieving the goal, based e.g. on the current availability of resources. If the application decides of *not* participating, `control` is never involved and the rest of this flow is not applied;
2. `control` is informed of the participation, and possibly fed with policy-specific parameters (e.g., bids for auctions);
3. `control` performs whatever is required by the `Pre-Work Coordination` of Fig. 7. If this agent is selected for achieving the goal, it returns a positive answer to the application. If the answer is negative, the application does any required clean up operation and the rest of this flow is not applied;
4. the application starts executing the required task-specific actions, subject to a *maintenance condition* (as in BDI) or other architecture-specific mechanism to abort its execution. The condition for aborting includes a signal from `control`;
5. The handlers for messages related to the goal being achieved must call `control` before dealing with them. Thus, `control` is able to enforce the coordination policy by overhearing the application channel, in addition to handling timeouts and messaging on the `control` channel;
6. if it fails to achieve the goal or the abort condition becomes true, the application has to notify `control` before performing anything else. This may send (depending on the policy) a `FAIL` message;
7. if the application succeeds in achieving the goal, it has to call `control` with the results. This will issue the final `DONE` message, possibly after invoking further application for operations such as merging of results under collaborative policies.

Of course, simpler application/`control` interaction models are possible with a tight integration with the agent architecture. For instance, steps 5 to 7 would be easily eliminated from BDI by appropriately extending its interpreter or adding some so-called *meta-level* plans [19].

10 Related Works

Team programming based on joint intentions has been explored by various authors, in addition to those already mentioned in Sections 3 and 5. For instance, STEAM [23] combines Joint Intentions Theory and SharedPlans [10], and adds decision theoretic communication selectivity in order to adapt type and amount of communication to the context, by taking in account its costs and its benefits with respect to the team's goals.

Dignum et al. [8] deal with team formation via communication, something that is clearly related to our work. Our primary objective is not team formation, since this is solved by the very definitions of implicit organization and goal-specific sub-team – however, as mentioned in Sec. 3, the way the latter is formed may become more sophisticated in future. Moreover, with respect to the decomposition in steps required to form a team proposed by the model in [8], we can draw a parallel between their "potential recognition" step, where an agent takes the initiative to find a team, and the situations where a new agent joins an organization, or takes the initiative of renegotiating the coordination policy (and consequently how tasks are allocated among agents) because, for instance, of the poor performance of the current policy.

The computational model presented in Sec. 9.2 reminds of the work by Chen and Decker [4]. They supply a library of coordination protocols, analyze the structure of agent plans and insert calls to the coordination library where appropriate. Over time, agents can learn which coordination mechanism is more suitable to achieve a given goal in a given situation. Our approach is similar, in that we also supply a library of coordination protocols, but differs for at least three reasons. First, the coordination protocol is negotiated beforehand by the organization and for all goals achievable by a role. Second, we expect agent developers to supply plans tailored to specific coordination protocols, which is to say that goals may be achieved differently depending on the policy chosen by the organization; moreover, coordination actions are left

out of the body of the plans, in accordance to the schema of Fig. 7. Third, anything learned concerning coordination is not used while achieving goals, but when negotiating; and, learning can be performed by any agent outside the organization, by overhearing its internal communication, tracing its performance, acting as oracle during negotiation, and possibly even forcing negotiation when the organization underperforms.

A generic framework for developing computer-supported collaborative systems and modelling their coordination policies, called COCA (Collaborative Object Coordination Architecture) [15], has many analogies with the work presented here. Users explicitly adopt roles; coordination policies are explicitly and collectively specified in terms of relationships among roles, using a logic-based specification language. COCA uses IP multicast as an efficient technique for group communication, encountering the same kind of difficulties with unreliability that we deal with. It is important to stress that roles and policies in COCA concern the way users – not software components, as in our case – cooperate towards a common task; for instance, policies are used to decide who "controls the floor" (i.e. who is authorized to operate on the shared workspace). Having that in mind, the main difference with our work arises from COCA's different interpretation of the concepts of "cooordination" and "role". They are defined in a way similar to data types or classes in traditional programming languages, in the sense that they imply certain constraints on the behaviour of their instances. A set of users who are governed by the same set of coordination policies is said to be playing the same role, which is to say that there is no distinction between coordination policies and roles, or that policies are a kind of role definition augmented with constraints. Another difference with our work is that, to keep the coordination processes and applications separated, COCA provides a general layered architecture, called the COCA virtual machine. This architecture is composed by a communication layer and an application layer at the two extremes, while a collaboration layer in the middle interprets the coordination policies and glues the other two together. Coordination policies are then written according to the application, rather than being independent.

11 Conclusions and Future Works

We proposed the concept of *implicit organizations* for the coordination of agents able to play the same role possibly in different ways, exploiting the ability of performing group communication and overhearing in environments where messages may be occasionally lost and agents can come and go very quickly. We presented a protocol for negotiating a common coordination policy, gave the general outline of an organizational coordination protocol, and discussed a few examples. Some architectural considerations and a preliminary computational model have been presented.

In the near future, we will focus on practical experimentation and application to our domain, i.e. multimedia multi-modal cultural information delivery in smart rooms ("active museums").

Future directions of reasearch include: the creation of specialized overhearers agents, whose goals include helping in achieving robustness by catching and recovering partial message losses, supervising the behaviour of implicit organizations, and applying machine learning or other techniques for deciding the "best" policy for a role in a given environment; and, a tight integration of the computational model within the BDI architecture.

Bibliography

1. M. Aiello, P. Busetta, A. Donà, and L. Serafini. Ontological Overhearing. In *Proceedings of the Eighth Int. Workshop on Agent Theories, Architectures, and Language (ATAL)*, Seattle, USA, August 2001.

2. P. Busetta, L. Serafini, D. Singh, and F. Zini. Extending Multi-Agent Cooperation by Overhearing. In *Proceedings of the Sixth Int. Conf. on Cooperative Information Systems (CoopIS 2001)*, Trento, Italy, 2001.

3. Paolo Busetta, Antonia Doná, and Michele Nori. Channeled multicast for group communications. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 1280–1287. ACM Press, 2002.

4. Wei Chen and Keith Decker. Applying coordination mechanisms for dependency relationships under various environments. In *Workshop on MAS Problem Spaces and Their Inplications to Achieving Globally Coherent Behaviour, AAMAS2002*, Bologna, Italy, July 15-19, 2002.

5. P. R. Cohen and H. J. Levesque. Teamwork. Technical Report 504, AI Center, SRI International, Menlo Park, CA, 1991.

6. Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261, 1990.

7. S. E. Deering. RFC 1112: Host extensions for IP multicasting, August 1989.

8. Frank Dignum, Barbara Dunin-Kęplicz, and Rineke Verbrugge. Agent theory for team formation by dialogue. *Lecture Notes in Computer Science*, 1986, 2001.

9. Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification. `http://www.fipa.org/repository/cas.html`.

10. Barbara J. Grosz and Sarit Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357, 1996.

11. J. Y. Halpern and Y. O. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the Association for Computing Machinery*, 37:549–587, 1990.

12. A. Kaminka, D. Pynadath, and M. Tambe. Monitoring Teams by Overhearing: A Multi-Agent Plan-Recognition Approach. *Journal of Artificial Intelligence Research*, 17:83–135, 2002.

13. S. Kumar, M. J. Huber, D. McGee, P. R. Cohen, and H. J. Levesque. Semantics of agent communication languages for group interaction. In *Proceedings of the 17 th Int. Conf. on Artificial Intelligence*, pages 42–47, Austin, Texas, 2000.

14. Sanjeev Kumar, Marcus J. Huber, and Philip R. Cohen. Representing and executing protocols as joint actions. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 543–550. ACM Press, 2002.

15. Du Li and Richard R. Muntz. COCA: Collaborative objects coordination architecture. In *Computer Supported Cooperative Work*, pages 179–188, 1998.

16. Stephen Morris and Hyun Song Shin. Approximate common knowledge and co-ordination: Recent lessons from game theory. *Journal of Logic, Language and Information*, 6(2):171–190, 1997.

17. O. Stock and M. Zancanaro. Intelligent Interactive Information Presentation for Cultural Tourism. In *Proceedings of the International CLASS Workshop on Natural Intelligent and Effective Interaction in Multimodal Dialogue Systems*, Copenhagen, Denmark, 28-29 June 2002.

18. Anand S. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. In *MAAMAW'96: 7th European Workshop on Modelling Autonomous Agen ts in a Multi-Agent World*, LNAI 1038. Springer-Verlag, January 1996.

19. Anand S. Rao and Michael P. Georgeff. An Abstract Architecture for Rational Agents. In W. Swartout C. Rich and B. Nebel, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, San Mateo, CA, 1992. Morgan Kaufmann Publishers.

20. I. Smith, P. Cohen, J. Bradshaw, M. Greaves, and H. Holmback. Designing conversation policies using joint intention theory. In *Proceedings of Third International Conference on Multi-Agent Systems (ICMAS98)*, 1998.

21. Ira A. Smith and Philip R. Cohen. Toward a semantics for an agent communication language based on speech acts. In Howard Shrobe and Ted Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference, Vol. 2*, pages 24–31, Menlo Park, California, 1996. AAAI Press.

22. R. G. Smith. The contract net protocol: High level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, 1980.

23. M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

24. G. Tidhar, A. S. Rao, and E. A. Sonenberg. Guided team selection. In *Proceedings of the Second International Conference on Multi-Agent Systems*, Kyoto, Japan, December 1996. AAAI Press.

25. Gil Tidhar. *Organization-Oriented Systems: Theory and Practice*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, Australia, 1999.