

Real-time Role Coordination For Ambient Intelligence

Paolo Busetta,
Mattia Merzi
ITC-irst
Povo, Trento - Italy
busetta@itc.it
merzi@itc.it

Silvia Rossi^{1,2}
¹Università di Trento
Povo, Trento - Italy
²Istituto di Cibernetica, CNR
Pozzuoli (NA) - Italy
silvia.rossi@dit.unitn.it

François Legras
Onera DCSD
Toulouse - France
francois.legras@cert.fr

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 in which everybody hears everything being said. The issues that have to be faced (e.g., changing users’ preferences and locations; performance constraints; redundancies of sensors and actuators; agents on mobile devices continuously joining and leaving) 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. A set of protocols, designed for unreliable group communication, are used to negotiate a coordination policy, and for team coordination. We sketch a general computational model for an agent participating to an implicit organization.

General Terms

Algorithms, Reliability, Design

Keywords

Pervasive computing, coordination, group communication, role, 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. Some active environments are being developed for the PEACH project [15], whose domain is interactive cultural information delivery within museums or archaeological sites. An “active museum” can be seen as a large scale multi-user, multi-media, multi-modal system. 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.

Our agents must immediately adapt to changes in the focus of attention or to 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 use a form of group communication, called *channeled multicast* [2], for coordinating agents. 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 [5], monitoring [10], 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 our current work on an agent coordination technique based on channeled multicast. Rather than addressing the well-known problems of task decomposition or sub-goal negotiation, we focus on achieving robustness and tolerance to failure in a setting where agents can be redundant, communication is unreliable, hardware can be switched off, and so on. Such an environment can evolve faster than the agents execute their task. We want to avoid centralized or static solutions like mediators, facilitators or brokers; rather, we aim at a fully distributed and flexible system, without necessarily looking for optimality.

More specifically, we defined a general set of social conventions for establishing and enforcing a coordination policy among agents playing the *same* role. From these conventions, we derived protocols, designed to work under the assumption of unreliable communication, and a generic agent computational model, in which coordination and task-specific capabilities are kept separated as much as possible. A formalization, based on the Joint Intention Theory [7], is available in [4].

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 following three sections describe a few coordination policies, discuss how an organization decides its own, and show some examples (Sections 5, 6, and 7 respectively). Some architectural considerations and a computational model are presented in Sec. 8.

2. SCENARIO AND SETTINGS

As mentioned above, PEACH [15] 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 is a very challenging application domain: 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 visitors, they will simply ignore it or, worse, will become annoyed.

A typical situation that may arise in PEACH is the following. A visitor is supposed to receive a multimedia presentation on a particular subject (e.g., a painting she is close to), 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); (2) the visitor is close to several actuators (screens, speakers) each able to show 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 nearby visitors are potentially using the same actuators, and of course if the visitor does not get a presentation after a few seconds she will leave.

The main communication infrastructure we use in PEACH, called LoudVoice, is based on the concept of channeled multicast [2]. LoudVoice uses the fast but inherently unreliable IP multicast – which is not a major limitation when considering that, as said above, our communication media is unreliable by nature. Channels in LoudVoice can be easily discovered by their *themes*, that is, by the main subjects of conversation; a theme is just a string taken from an application-specific taxonomy of subjects, accessible as an XML file via its URL. Having discovered one or more channels, an agent can freely “listen” to and “speak” on them by means of FIPA-like messages, encoded as XML documents. The header of a message includes a performative, its sender and one or more destinations; the latter can be agent identifiers, but any other expression is accepted (for instance, we use role names – see Sec. 4). LoudVoice has been developed in Java, is being ported to C++, and runs both on standard desktops and on handheld devices.

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 [2], an auction system with two main roles: the auctioneer (which calls for bids, collects them and declares 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 [16], 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 includes different roles, and is formed in order to achieve a specific goal; as said above, an implicit organization includes all agents playing a given 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 towards 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 5 describes some policies we use in our domain, and our computational model (Sec. 8.2) 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, this case is effectively indistinguishable 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 [8] – generates 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, unre-

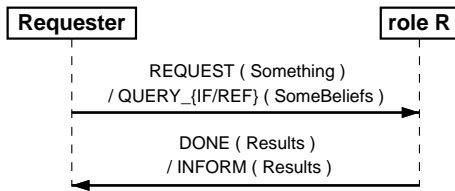


Figure 1: A role-based interaction

liable 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.

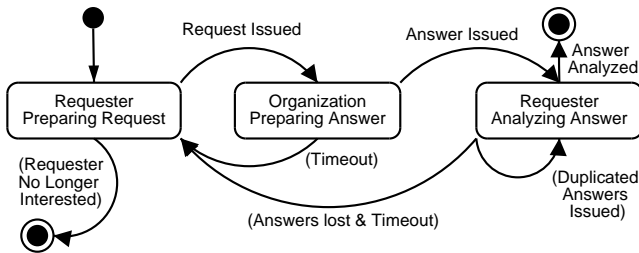


Figure 2: Interacting with unreliable communication – a simplified protocol machine

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 [9]), our current choice is to have the requester to consider whatever answer it receives first as the valid one, 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 [5, 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. COORDINATING AN ORGANIZATION

A high-level formalization of the coordination within an implicit organization needed to achieve a goal – that is, how the Organization Preparing Answer state of Fig. 2 is implemented – is provided in [4]. It prescribes two main steps: first, negotiating a coordination policy, if one has not been already established; second, in a policy-specific way, achieving the goal. Negotiation is described in Sec. 6; we focus here on the second step.

Recall, from Sec. 3, that when an implicit organization receives a requests (i.e., the state Organization Preparing Answer is entered), its members with available resources form a sub-team. This happens silently, i.e. there is no explicit group formation message.

Below, we discuss some specific policies in use in our applications, but first we need to define organizational coordination in abstract terms. Any coordination policy follows a straightforward three-phase schema (Fig. 3). Before doing anything, the sub-team coordinates 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).

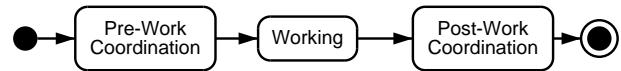


Figure 3: Abstract coordination policy, UML state diagram

The abstract schema above is summarized by Figure 4 from the perspective of a single member of the sub-team, which in turn is reflected by the general computational model presented in Sec. 8.2. 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.

In the following, we describe the four 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. Practical examples of their application will be shown later, in Sec.7.

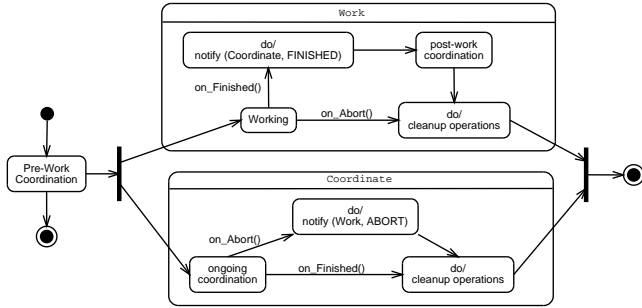


Figure 4: Abstract behavior of a member of a goal-specific subteam – UML state diagram

5.1 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. 4, 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 accept whatever answer comes first and ignore the others.

5.2 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 to finish 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 messages and synthesizes the common result, in a goal-dependent way. 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.

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.

5.3 Multicast Contract Net

This policy is a simplification of the well-known Contract Net protocol [14], 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 sent.

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 declares 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 (Sec. 6), 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.

5.4 Master-Slave

This policy has many similarity with the Multicast Contract Net; the essential difference 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 necessarily 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 send an INFORM to the role declaring their availability. The master agent (which may or may not be one of those available to work) collects all declarations, and issues an INFORM to the role nominating the slave, which acknowledges 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 end up believing 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.

6. NEGOTIATING A POLICY

We discuss here how an implicit organization establishes its own coordination policy. This is done formally in [4] by means of the Joint Intention Theory (JIT) [7, 13]. Here, we outline the protocol and present the algorithm applied by each agent to implement it; the latter has been designed for unreliable communication. The protocol is executed every time a member of an implicit organization joins or leaves a channel or an inconsistency is detected, as described later.

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. The protocol can be restarted at any time, e.g. when an inconsistency is found or there is no common policy instance (in which case agents have to take some actions, such as notifying their users, before re-negotiating). In the normal case, i.e. when the protocol is not restarted mid-way, the total number of messages sent on a channel for a negotiation is equal to the number of agents in the implicit organization multiplied by a tunable parameter (called `max_repeats`, typically set to 2 or 3), plus one (e.g., with 50 agents and `max_repeats` set to 2, 101 messages are sent).

Recall, from Sec. 3, that a policy instance is a tuple with the name of a policy and ground values for all its parameters; for instance, `< 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 [3], we describe a language for expressing sets of policy instances in a compact way by means of constraints on parameters, including infinite or undetermined ranges; operators are provided that combine policy instance statements in order to obtain the union or the intersection of the represented sets. 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. Some simple examples of statements are the following:

```
name = MulticastContractNet
  param = Currency constraint = (one of Dollar, Euro)
        suggested = (Euro)
  param = WinningCriteria value = lowest
  param = BidTimeout constraint = (in 100..2000 )
        suggested = (1000)
name = MasterSlave
  param = Master constraint = (not one of Agent1, Agent2)
        suggested = (Agent3, Agent4)
```

From the perspective of an individual member, the coordination policy of an organization, independently of 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 then to negotiate it with the other members. We show in the algorithm below how the negotiation is triggered.

As said above, the negotiation is done in two straightforward steps. In the first, a mutual belief about the policy instances common to the entire organization is established. This can be easily achieved by having each agent sending INFORMs to the role on what it is able to support, and intersecting the contents of all received messages. The second step consists in having one agent – called the *oracle* – selecting one policy among those common to everybody and notifying the organization of its decision, by means of a simple ASSERT sent to the role.

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 is 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. 7). The algorithm presented below: (1) 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; (2) provides a default oracle election mechanism if an external one is not present; and, (3) handles conflicting oracles by forcing a re-negotiation.

For the negotiation protocol to work in an unreliable environment, one additional information and a few more messages to those already mentioned are needed.

All messages concerning policy negotiation are marked with a *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 coherence 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 known one are actually handled.

The protocol is made robust in two other ways. First, during the first phase of the protocol, rather than simple INFORMs on the supported policies, agents exchange 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 and the policy reminder cause the organization to re-negotiate the policy.

6.1 Implementing the protocol

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 [3].

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 messages (INFORM, ASSERT, QUERY) to the role for which the policy is being negotiated; 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.

Each agent has three main beliefs related to the policies: the set of policies it supports, the set of policies believed to be common to all agents in the organization, 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. 8.2).

The three main beliefs described above and the state of the negotiation are represented by the following variables:

```
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 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 message 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 re-configurations 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 re-joining, 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 believes – 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”.

```

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

```

7. 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 Sec. 5. Interactions will be illustrated with a simplified, FIPA-like syntax.

7.1 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>...</doc3> ) )

```

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 an 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 causes its agent to tune into the local channel for its role; consequently, in a typical peer-to-peer fashion, a new 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.

7.2 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 her 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* works 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 *PPs* collaborate – sensibly merging multi-media presentations is a hard task even for a human. Observe that, in realistic situations, redundancy of *PPs* is a necessity, e.g. to handle the workload imposed by hundredth of simultaneous visitors. Redundancy can be obtained in various ways, for instance by putting identical *PPs* 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 means specifying criteria for bidding in a *Multicast Contract Net*, accepting a *Master-Slave* policy but preventing its own *PP* from becoming a master, and so on.

7.3 Choosing among Multiple Screens

Smart rooms may contain multiple places where to show things to users, e.g. large screens on different walls, the users' own PDAs, computers scattered in the room. Location, but also quality and logical congruence with the tasks being performed by the user, are

all important factors. Also, it is not necessarily the case that a single screen is a good choice – for instance, a presentation to a group of people may be better shown in multiple locations 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 a wall screen, or mobile, if on board of 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.

8. ARCHITECTURAL CONSIDERATIONS

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

8.1 Communication channels

So far, we have described all protocols as if all messages were 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 one channel (at least) 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 traffic (requests, queries, and their answers), which also plays a part in organizational coordination, is addressed to its “natural” destination role on its “natural” channel created by the agents themselves or configured by a network manager. 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 extemporary messaging only. By contrast, we leave to application designers total freedom on the messaging 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.

8.2 An agent computational model

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

The computational model presented here reminds of the work by Chen and Decker [6]. 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 code 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 goal-achieving code, in accordance to the schema of Fig. 4. 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 under-performs.

We do not dictate any specific agent architecture, either BDI [12, 11], 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.

`control` exports the current information about an implicit organization (the state of its policy – being negotiated or decided –, supported, common, and current policies) to `application`.

Interactions between `control` and `application` have two main purposes. The first is feeding `control` with the policies supported by `application`. Typically, this happens at start-up 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, contractNet (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 Plain Competition and Multicast Contract Net, 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. 4. `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 `application`. Consider, for instance, Plain 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 wasteful if not harmful.

Consequently, the model of interaction between `application` and `control` outlined below is a bit convoluted, but is architecture independent and fully separates – and thus enables reusability of – the two components:

1. a `REQUEST` or `QUERY` is handled by `application`, which decides whether or not to participate to the subteam achieving the goal, based e.g. on the current availability of resources. If `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. 4. If this agent is selected for achieving the goal, it returns a positive answer to `application`. If the answer is negative, `application` does any required clean up operation and the rest of this flow is not applied;
4. `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, `application` has to notify `control` before performing anything else. The latter may send (depending on the policy) a `FAIL` message;
7. if `application` succeeds in achieving the goal, it has to call `control` with the results. The latter will issue the final `DONE` message, possibly after invoking `application` again 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 [12].

9. CONCLUSIONS AND FUTURE WORKS

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

In the near future, we will focus on practical experimentation and application to our domain, i.e. multi-media, multi-modal cultural information delivery in smart rooms (“active museums”). Preliminary performance results on the negotiation algorithm show that, on a real, busy LAN, 20 agents can negotiate their own coordination policy in less than a second, without any external oracle.

Longer term research should focus on two main areas: overhearing and computational models. We envision the creation of overhearers agents, whose goals are: helping in achieving robustness by catching and recovering partial message losses; supervising the behavior of implicit organizations; and, applying machine learning or other techniques for deciding the “best” policy for a role in a given environment. Finally, we will investigate a tighter integration of the proposed computational model with the BDI architecture.

Acknowledgments

This work has been supported by the PEACH and TICCA projects, funded by the Autonomous Province of Trento.

10. REFERENCES

- [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, A. Donà, and M. 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.
- [3] P. Busetta, M. Merzi, S. Rossi, and F. Legras. Intra-role coordination using channeled multicast. Technical Report 0303-02, ITC-IRST, Trento, Italy, 2003.
- [4] P. Busetta, M. Merzi, S. Rossi, and F. Legras. Intra-Role Coordination Using Group Communication: A Preliminary Report. In *Proc. of the Agent Communication Languages and Conversation Policies workshop at AAMAS 2003*. ACM Press, 2003.
- [5] 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.
- [6] Wei Chen and Keith Decker. Applying coordination mechanisms for dependency relationships under various environments. In *Workshop on MAS Problem Spaces and Their Implications to Achieving Globally Coherent Behaviour, AAMAS2002*, Bologna, Italy, July 15-19, 2002.
- [7] P. R. Cohen and H. J. Levesque. Teamwork. Technical Report 504, AI Center, SRI International, Menlo Park, CA, 1991.
- [8] Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification. <http://www.fipa.org/repository/cas.html>.
- [9] 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.
- [10] 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.
- [11] Anand S. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. In *MAAMAW'96: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, LNAI 1038. Springer-Verlag, January 1996.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] O. Stock and M. Zancanaro. Intelligent Interactive Information Presentation for Cultural Tourism. In *Proc. of the International CLASS Workshop on Natural Intelligent and Effective Interaction in Multimodal Dialogue Systems*, Copenhagen, Denmark, 28-29 June 2002.
- [16] Gil Tidhar. *Organization-Oriented Systems: Theory and Practice*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, Australia, 1999.