A PRACTICAL METHOD FOR PROACTIVE INFORMATION EXCHANGE WITHIN MULTI-AGENT TEAMS

A Thesis

by

RYAN TIMOTHY ROZICH

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2003

Major Subject: Computer Science

A PRACTICAL METHOD FOR PROACTIVE INFORMATION EXCHANGE WITHIN MULTI-AGENT TEAMS

A Thesis

by

RYAN TIMOTHY ROZICH

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Thomas Ioerger (Chair of Committee)	Richard Volz (Member)
Christopher Menzel (Member)	Valerie Taylor (Head of Department)

August 2003

Major Subject: Computer Science

ABSTRACT

A Practical Method for Proactive Information Exchange

within Multi-Agent Teams. (August 2003)

Ryan Rozich, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Thomas Ioerger

Psychological studies have shown that information exchange is a key component of effective teamwork. In addition to requesting information that they need for their tasks, members of effective teams often proactively forward information that they believe other teammates require to complete their tasks. We refer to this type of communication as *proactive information exchange* and the formalization and implementation of this is the subject of this thesis. The important question that we are trying to answer is: under normative conditions, what types of information needs can agent teammates extract from shared plans and how can they use these information needs to proactively forward information to teammates? In the following, we make two key claims about proactive information exchange: first, agents need to be aware of the information needs of their teammates and that these information needs can be inferred from shared plans; second, agents need to be able to model the beliefs of others in order to deliver this information efficiently. To demonstrate this, we have developed an algorithm named PIEX, which, for each agent on a team, reasonably approximates the information-needs of other team members, based on analysis of a shared team plan. This algorithm transforms a team plan into an individual plan by inserting communicative tasks in agents’ individual plans to deliver information to those agents who need it. We will incorporate a previously developed architecture for multi-agent belief reasoning. In addition to this algorithm for proactive information exchange, we have developed a formal framework to both describe scenarios in which proactive information exchange takes place and to evaluate the quality of the communication events that agents running the PIEX algorithm generate. The contributions of this work are a formal and implemented algorithm for information exchange for maintaining a shared mental model and a framework for evaluating domains in which this type of information exchange is useful.

To my family,

for their endless amount of support, encouragement and love throughout all these years.

ACKNOWLEDGMENTS

I gratefully acknowledge the support and presence in this work of many people without whom I never would have been able to complete my program. I am grateful to my advisor, Dr. Thomas Ioerger, whose support has made my graduate experience the best possible; I feel lucky to have had him as my advisor, mentor, and friend. Since I started doing research with Dr. Ioerger as an undergraduate, I haven’t made many academic, research or career decisions over the past three years without consulting him, and he has never steered me wrong. Dr. Ioerger’s enthusiasm for research, his knowledge and sense of direction has inspired me and has made what is at many times a difficult and uncertain journey, manageable - and at times, even fun.

I would also like to thank my other committee members Dr. Richard Volz, whose support for my research and leadership within our MURI group have been very valuable to me, and Dr. Christopher Menzel for taking the time to review my thesis and sit on my committee. Most of this research has been built upon the research of the CAST/MALLET project in the MURI group at Texas A&M University. I would like to thank the members of that group: Mr. Mike Miller, whose informal discussions and suggestions have proved very useful to my work on CAST-PM. Also, the other members of that group, Dr. Dianxiang Xu, Keith Biggers, Yue Zhou, Lini He, and Sen Cao.

Last, but never least I would like to thank my family, who have supported me financially and have been a never-ending source of encouragement and love throughout my

university career.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

With the advent of communication networks for computers, it is a rare situation in which computer-based intelligent agents can operate usefully in isolation. Many times, either the complexity of the task at hand or the number of interacting agents make it more useful and convenient to deal with them as teams of agents (rather than a group of interacting individual agents). The discipline of Multi-Agent Systems (MAS), which is a subset of Distributed Artificial Intelligence (DAI), is the study of how interacting, intelligent agents can pursue shared goals in a coherent way. Multi-Agent Systems research involves the study and construction of agents at the micro (individual agent) as well as the macro (group or team level). Teamwork is a structured way to collaborate, and communication plays a variety of roles in multi-agent teams, including: information exchange, task delegation[1], team coordination, negotiation and contracting, distributed planning [2], conflict resolution, and maintaining situation awareness, just to name a few.

As a basis for the study of multi-agent teams, it is useful to look at the large body of research into human teams. Teamwork, which is defined as two or more people working interdependently toward a shared goal [3], is the focus of a wide variety of research, spanning disciplines from Business Management, to Psychology, to Organizational Theory, to Philosophy. Communication is a vital part of the team process, researchers have studied the ability of teams to use communication effectively in order to maintain

---

This thesis follows the style and format of *IEEE Transactions on Systems, Man, and Cybernetics*.

situation awareness, perform joint decision making, and to request help or offer assistance. The roles of communication in teams and how teams use communication effectively has been studied in a wide variety of ways, from experiments to observation to simulation.

Building computational models that mimic naturalistic human teamwork is an important goal for the implementation of large-scale collaborative systems. The challenge is simulating the implicit team knowledge that humans seem to employ. That is, humans, without explicitly planning such team activity know the right thing to do in certain situations (i.e. backup teammates, share information, report failure). Getting agents to be this intelligent is a significant goal. In this thesis, we will be considering one piece of this puzzle, that is implementing the ability for agent teams to automatically keep each other informed of relevant information. To this end, our primary focus is how to build agent teams that communicate effectively by reasoning about the plans, responsibilities, and knowledge of their teammates. Some important challenges relating to this are, defining formal semantics for communicative actions [4-6], describing agents motivation for exchanging information [7-9], modeling the belief states of other agents [10, 11], and defining optimality conditions for communications.

Currently, there are a variety of ways in which agents communicate in collaborative environments, depending on the communicative goals of the agent (coordination, delegation, negotiation, information exchange). First we note that it is common in many interoperating systems to have communication protocols that are ad-hoc, hard-coded communication points. As the size and complexity of agents and tasks grows though, coherent action is difficult to achieve using this approach [12]. As an alternative to this approach, some practical architectures for building multi-agent systems have a method of automatic communication grounded in formal semantics. For instance, STEAM [13] uses joint intentions theory to ensure that all agents stay coordinated with respect to joint-persistent-goals (JPGs). STEAM also uses a decision-theoretic framework for communication selectivity. Another architecture is GRATE* [1], which uses a model of joint responsibilities that enable teams of agents to automatically negotiate who will be responsible for different goals, the plans for how to achieve those goals, and also how to recover from failure. Other agent architectures use communication in order to maintain a consistent knowledge base over all the agents on the team through distributed truth maintenance [11]. While others communicate to perform distributed problem solving, such as distributed constraint satisfaction [14]. Whatever the higher-level team goal of the communicative acts, we believe that this type of general, automatic, and verifiable communication is necessary for complex multi-agent systems.

A particularly important role of communication in teamwork is exchanging information. In most multi-agent environments, both knowledge and information are distributed. An important assumption in this thesis is that information exchange within teams can improve the team efficiency, since knowing required facts earlier could allow tasks to be completed more quickly. For example, if a team of military aircraft were on a mission and one aircraft were to notice an area that contained enemy surface to air missile (SAM) site, it would initiate routines to reroute around that area. In addition, this team member, knowing that his teammates also need this information, should choose to communicate this information to them. This would allow the teammates to reroute their flight paths earlier than if they each came across the information on their own, and thus operate more efficiently (and safely).

Another important role for communication, especially in domains involving tactical decision-making, is to build and maintain situation awareness, such as by assimilating information from disparate sources in order to abstract, classify and understand what is going on in the environment. Examples of this have been observed in human teams from fire brigades to battalion tactical operations centers, to air traffic controllers. Among other things, information exchange allows team members to act as if they possessed a shared mental model that is to act as if their sensors, knowledge and decision-making were centralized, rather than distributed.

There are at least two ways in which teammates can exchange information[1]:

1. Information-needers can request information needed from information-suppliers, or

2. Information-suppliers can proactively forward this information to information-needers. Most information exchange in multi-agent systems uses the first type of communication (request-reply) exclusively. We claim that the second type of communication, referred to as proactive information exchange, can enhance team efficiency in two respects, first because agents informed of needed information should be able to perform more efficiently than if they had to discover the information on their own and, secondly because agents needing information may not know who within their

[1] We recognize that there are more ways to exchange information, such as broadcasting messages or posting information to shared message boards (black boards).

team (if anyone) has the information that they desire. In the proactive information exchange model, teammates committed to achieving a shared goal anticipate each other’s information needs and automatically forward this information (if they know it) to the agent(s) that need it. A major challenge in implementing this type of communication is for an agent to determine exactly which agents need what information at any point in time. We believe that determining relevance of information can be accomplished by the analysis of a shared team plan [15] (i.e. what others on the team are responsible for and what the preconditions of those actions are). Our PIEX (Proactive Information EXchange) algorithm, which extracts information needs from shared plans, is the focus of this thesis.

Yen et al. [16] define ideal conditions under which an agent should proactively communicate with another agent. Beliefs are modeled using a modal operator B, e.g. $B_{Mike}(have\,hammer)$, with the usual possible worlds semantics [17]. Goals refer to specific steps in plans, to which agents can make commitments. This condition is as follows:

$$B_A(I) \wedge B_A\left(\neg B_B(I)\right) \wedge B_A(Goal_B(G)) \wedge \left\lfloor \neg B_B(I) \rightarrow \mathrm{p} \neg Done_B(G) \right\rfloor \wedge$$
$$\left[B_B(I) \rightarrow \neg \mathrm{p} \neg Done_B(G)\right] \rightarrow Goal_A(Inform\ \ B\ I)$$

Where □ is the temporal operator for ‘always’. This formula says that when agent A believes some information fact I *and* A does not believe that B believes I (i.e. I is not redundant) *and* A believes that B has a goal G and I is a precondition of G (i.e. I is relevant to B) then A should tell B fact I. There are two challenges to a practical implementation of this condition for information exchange: first, agents need to be able to reason about the goals (and therefore the information needs) of other agents and second, agents need to reason about the beliefs of other agents.

There is another theoretical question separate from the conditions in which proactive information exchange takes place, and that is: what motivates agents to monitor their teammates goals and to communicate when they think it is helpful to their teammates? If agents have individual goals as part of a team plan, why would they bother to help others achieve their portion? A simple answer might be that agents are simply computer programs and therefore are motivated to communicate because that is the way we design them to behave; but we must recognize that not all multi-agent systems are closed and therefore agents may need to collaborate with possibly self-interested agents that have different designers[2]. Therefore, as part of this thesis, we will describe the motivation in which agents will have to proactively exchange information. This motivation is based on joint intentions theory [18] and an expanded formal description of responsibilities presented in [9]. The motivation can be derived from their joint commitment to the team goal, and the consequent requirement to maintain mutual belief. For a team of agents jointly committed to a goal Θ, and who have delegated responsibility for some sub goal Θ' to a member of the team, each member not assigned to Θ' should actively monitor the information needs and beliefs of the responsible agent (in parallel with their own

[2] There is also a large body of research having to do with mixed human-agent teams. While we are interested in agent-only teams in this thesis, the possibility of introducing humans into the loop increases the need to be sure that our teammates will be sufficiently motivated to be helpful.

activities) and send any relevant information that they believe to the teammate that needs it.

Our goal is to specify multi-agent teams by providing the required teamwork process and knowledge (in the form of shared plans) and allowing the agent architecture to automatically generate the required information exchanges, without having to explicitly define communication points in the system. In this thesis, we will describe our general algorithm, named PIEX, for Proactive Information EXchange in MAS teams. PIEX uses a pre-processing algorithm to infer information needs of other agent teammates based on analysis of a shared plan. We also use a multi-agent belief reasoning framework to filter out redundant messages. We claim that these two pieces provide a reasonable approximation of an implementation of the formal conditions for proactive information exchange given above. These methods can be applied to other agent architectures, especially those based on Hierarchical Task Networks (HTNs) [19]. The key idea is to use a generic information-exchange plan that runs in parallel which monitors for info relevant to others and sends it automatically (if the other agent doesn't already believe the information). We then go on to describe a formal validation framework that will allow us to define optimal communication sequences for teams of agents in a variety of environments for assessing team performance. We will also describe the implementation of our PIEX algorithm in the CAST-PM agent architecture and demonstrate these agents ability to perform effective proactive information exchange in a military air combat simulation environment. Finally, we will validate the results (our agent's communicative acts) using a formal framework and evaluation criteria that we also developed as part of this thesis.

BACKGROUND

MULTI-AGENT SYSTEMS

In many environments, agents must work together to cooperatively solve problems or perform tasks. Teamwork, as defined in [20] is a special type of coordinated activity where team members work with each other in order to satisfy a shared goal. In some domains, goals or tasks may be impossible for a single agent to achieve, such as surveying land or fighting a battle. In other domains, goals such as preparing a meal or searching a space may be achieved more efficiently or effectively by teams rather than individuals.

The field of multi-agent systems (MAS) is highly interdisciplinary. Areas such as economics, philosophy, cognitive psychology, logic and computer science are all related and each look at the field differently. In this thesis, we are interested in developing practical methods for building multi-agent systems that can automatically infer information needs of other agents in their team in order to produce intelligent behaviors and solve complex problems through teamwork. Jennings et al. [21] list four key characteristics of multi-agent systems:

1) Each agent has incomplete information, or capabilities for solving the problem, thus each agent has a limited viewpoint

2) There is no global system control

3) Data is decentralized

4) Computation is asynchronous

Characteristics 1-3 are most important to our discussion of proactive information exchange and information needs in multi-agent systems. Agents in a distributed environment do not have total access to each other's senses or knowledge bases; although in many cases, agents could perform more efficiently if they did have a certain level of this access. Agent teammates exchange information (and effectively give access to each others knowledge base) through communication.

SHARED MENTAL MODELS

A mental model is an internal representation of a situation or environment, at a certain level of granularity that has the property of being complete – that is, all of details are filled in (and assumptions are made for missing information). Shared mental models extend this to a team context by representing not only an individual agent's own beliefs and activities, but also those of their teammates as well as the team structure and state. Shared mental models consist of different types of knowledge, both static and dynamic [22]. Static knowledge includes information about team structure (who is on the team, who plays what role), team goals, communication policies, agent capabilities and responsibilities. Dynamic knowledge consists of the knowledge of the team process that changes over time and includes current task assignments and status of agents, overall progress of the team, current knowledge of teammates, and the overall progress of the team towards its goal.

Studies have shown that maintaining situation awareness is critical to many teams effectiveness [23], especially in distributed command and control environments [24]. In these situations, team members receive different information and must work together to resolve any conflicts or ambiguities in order to come up with a common picture of the situation.

Maintaining a shared mental model can take on many aspects including maintaining shared ontology [7], shared goals [13], team structure and process [15, 20], and shared belief [10]. For the PIEX algorithm, we are mostly concerned with maintaining a shared mental model of team process and belief in order to estimate the information needs of other agents and decide whether to inform teammates of needed information.

The CAST project [16] is a multi-agent architecture for teamwork developed at Texas A&M University that simulates teamwork for multi-agent teams with an emphasis on maintaining a shared mental model among the team members. This allows teammates to dynamically assign tasks to team members, to anticipate information needs of teammates and to decide whether to forward relevant information to teammates. Team processes/plans, as well as knowledge about team structure are described using the MALLET (Multi-Agent Logic-based Language for Encoding Teamwork) language, which is described in more detail in the implementation section. Our reference implementation of the PIEX algorithm, which enables agents to proactively inform teammates of information that they need, fits into a specialized version of the CAST agent architecture, known as CAST-PM.

OTHER APPROACHES TO MULTI-AGENT COMMUNICATION

There are other agent architectures that incorporate automatic routines for generating agent communication. Most notably, STEAM [12], which generates communications to keep teams committed to shared goals and coordinated, Stone's work in communication for strategic teamwork in dynamic environments [25], Biggers and Ioerger's work on the TIP-C algorithm which transforms team plans into individual plans by inserting communicative acts, and Jenning's work on GRATE* which uses communication to coordinate teammates responsibilities [1].

The automatic communication routines in STEAM [12] are based on Joint Intentions theory [17, 18]. A joint intention is a mutual belief about a commitment to attempt to achieve a goal so long as the goal has not been achieved, can still be achieved and is still relevant. Communication in STEAM is used to establish and maintain joint intentions, and synchronize team action. Tambe describes a decision-theoretic framework for communication selectivity in STEAM. In this framework, the criteria for whether or not to communicate a fact, F is not only based on communication rewards and benefits but also the likelihood that relevant information is already common knowledge and hence unnecessary to communicate. Rewards and costs are measured in terms of the team, not the individual. The decision of whether or not to broadcast a fact, F to the team is made in terms of:

B – the benefit to keeping the team's knowledge of F coordinated.

τ - the probability that F is *not* shared belief in the team.

C_{mt} - the penalty for the teams beliefs about F to be out of sync

C_c - the cost of communicating

The decision of whether to broadcast fact F can be visualized as the decision tree in figure 1:

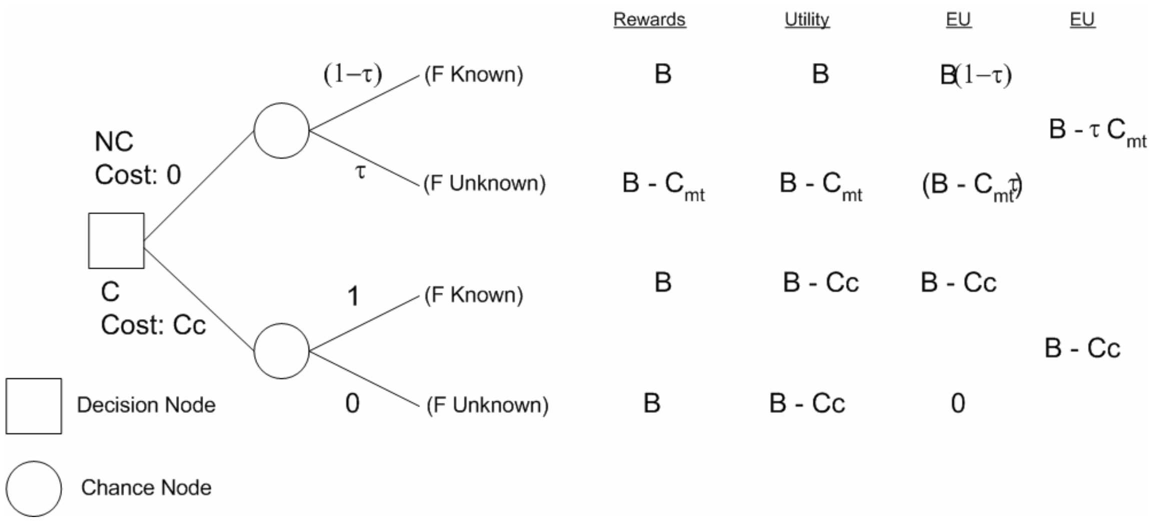


Figure 1 - Decision Tree for Selective Communication

We can see that the expected utility of not communicating $E(NC)=B-\tau C_{mt}$ and the expected utility of communicating $E(C)=B-C_c$ and therefore the agent will communicate iff $EU(C) > EU(NC)$ i.e. iff $\tau C_{mt} > C_c$.

Using this decision-theoretic framework, Tambe describes the motivation for agents to either communicate information needs of others, or to withhold such communication. While B, C_c, and C_{mt} can be considered a function of the current team situation (and fact F), τ is a value local to the individual agent making the decision to communicate. If F has already been broadcast by another teammate, the agent can be relatively certain that $\tau=0$ and it should not communicate. However, the agent can also reason about the ability of his teammates to observe F in order to get a better estimate of τ. If the agent believes that his teammates can all observe F, he can reason that τ is high and, if C_{mt} is sufficiently low and if C_c is sufficiently high, the agent can increase the team's utility by saving the team the cost of the communication. This suggests that belief reasoning can play an important role in information exchange and communication selectivity.

STEAM uses information exchange for the purposes of coordinating the team with respect to team operators and joint goals. To test the effectiveness of selective communications in STEAM, Tambe created an attack helicopter simulation with six different scenarios (varying costs of communication, visibility restrictions, etc) [13]. He also defined three team types with respect to communication – balanced, cautious and reckless – and compared the amount of communication performed between team members. Balanced agents exploit the decision theoretic communication structure described above, cautious agents always communicate, and reckless agents communicate very little. Their results show that the reckless team was rarely able to achieve its goal, and while the cautious and balanced teams were both able to achieve the team goals, the cautious team exchanges 10-20 fold more messages than the balanced team.

While STEAM focuses on communication for team coordination, we claim that information exchange may also be used to provide assistance to teammates in completing their tasks more efficiently by communicating relevant facts F to teammates if they do not currently believe F (even if they may come to believe F on their own at some time later). Whatever the use of information exchange, this decision-theoretic framework provides a useful way to describe the utility of reasoning about observability in modeling agent teammates' beliefs.

Another approach looks at the function of communication in terms of dynamic role assignment in strategic teams situated in low-bandwidth environments [25]. Stone uses PTS (Periodic Team Synchronization) to allow team members to act autonomously during low communication ("on-line") periods (due to a high cost of communication during these periods) and to coordinate and synchronize during high communication ("offline") periods. Stone implemented teams of agents in the Robocup soccer domain and therefore called the periods of high communication "locker-room agreements". These locker-room agreements allow teams to remain synchronized without making costly communications by agreeing upon roles and collections of roles (formations) and the conditions upon which to take on certain roles and configurations.

In [26], Biggers and Ioerger describe the TIP-C algorithm which takes a transformational approach to automatically generating communication in multi-agent teams. They focus on three principal goals of communication: synchronizing joint action, disambiguating shared responsibilities, and alerting of failure. TIP-C takes plans written in a multi-agent teamwork language (MALLET) and converts them to an equivalent individual plan in an individual agent language (TRL). The steps added in the individual plans keep the team synchronized and also perform the necessary communication required to generate team behavior, such as disambiguating responsibilities and failure recovery. This approach is similar to the approach taken in PIEX, in that both approaches insert communication by transforming team to individual plans.

The GRATE* agent architecture [1], also based on joint-intentions, generates communication in order to coordinate and synchronize joint action. A *situation assessment module* determines when joint action is warranted, and when an opportunity is identified, the identifying agent becomes the *organizer* of the joint action. The situation assessment module then hands control off to the *cooperation module* which attempts to identify and send messages to those acquaintances with potential for joint action. Potential teammates either accept or reject offers of joint action. The participants then negotiate the exact details of the common *recipe* (plan) for how to achieve the goals of the joint action, the joint action is then considered operational. Communication is also automatically generated when agents either drop commitments to a goal or drop commitments to the recipe for achieving the goal. Jennings has a real world team of agents running GRATE* that manages an electricity transportation network; software agents attempt to analyze alarms, identify potential blackouts, and control the flow of power to respond to blackouts.

Each of these approaches are related to our work with PIEX. These architectures take a principled approach to multi-agent communication to achieve the desired goals (coordination, synchronization, low-bandwidth usage) in order to properly simulate teamwork. Our approach is motivated by these approaches and aims to augment them by maintaining a shared mental model with respect to each agent's beliefs and automatically delivering needed information to other agents on a team.

FORMAL DESCRIPTION OF PIEX

In this section, we take two approaches to formalizing proactive information exchange: using a modal logic of belief to describe the internal conditions for an agent proactively forwarding information to another, and using a model of responsibilities built on joint intentions theory to describe the global team conditions for monitoring the information needs of teammates. Using formal methods to specify and reason about agent behavior is important because it allows us to specify agent-based systems in a concrete and verifiable way.

Yen et al. [16] presented a formalized condition upon which agents will perform proactive information exchange. In this, beliefs are modeled using a modal operator B , e.g. $B_{Mike}(have\, hammer)$, with the usual possible worlds semantics [17, 27, 28]. Goals refer to specific steps in MALLET plans, to which agents can make commitments. Proactive information exchange is formally defined as follows: Information *I* should be sent from agent *A* to a teammate agent *B* when

1) agent *A* believes the truth-value of *I*,

2) agent *A* believes that agent *B* does not currently believe *I*, and

3) *I* is an *information need* of *B*. Formally, *B* has the current goal *G*, the achievement of which depends on believing *I* (i.e. I is a pre-condition of goal G):

$$B_A(I) \wedge B_A(\neg B_B(I)) \wedge B_A(Goal_B(G)) \wedge \lfloor \neg B_B(I) \rightarrow \mathrm{p}\, \neg Done_B(G) \rfloor \wedge$$
$$[B_B(I) \rightarrow \neg \mathrm{p}\, \neg Done_B(G)] \rightarrow Goal_A(Inform\ \ B\ I)$$

Where $\Box$ is the temporal operator for 'always'. In [16] the DIARG (for Dynamic Inter-Agent Rule Generator) algorithm is presented as a first approximation to this condition in an implemented multi-agent system – particularly this implementation did not incorporate the ability for agents to reason about the beliefs of other agents. In DIARG, goals are examined for preconditions which are treated as information needs and allow agents to associate their teammates with the facts that are relevant to them. When an agent comes across a relevant fact, it generates a TELL message to that agent. However, determining relevance of information is only half of the picture for proactive information exchange; the other necessary component is *need*. Agents should only TELL other agents relevant information when they can be reasonably sure that they do not already believe it. We present PIEX as a second approximation to this condition; particularly we add the ability of agents to reason (in a principled way) about the beliefs of the other agents on their team.

This formal condition is important because it captures the notion that in teamwork, agents can determine relevance of information to each other by analyzing the preconditions of each others goals. While there may be other ways to determine relevance of information to agents (such as a publish-subscribe method), this method allows agents to determine information needs directly from their team goals. It is also important to capture the chain of mental states that an agent would go through in order to be *motivated* to exchange information with another teammate. We believe that this motivation can be derived from a concept of agent responsibility to others and to a team.

In [9] Ioerger and Johnson formally present the concept of responsibilities using a modal logic of intention [17]. They define responsibilities as similar to intentions in that they refer to mental attitudes of agents towards actions, sometimes producing commitments to action. Responsibilities are different from intentions though, in that responsibilities can be transferred (delegated) from one agent to another, whereas an agent can only intend itself to perform an action, or request that another perform the action thereby dropping the intention. A central aspect to this definition of responsibility is that, rather than treating responsibilities as irreducible mental states (like intentions are in the BDI model), Ioerger and Johnson model them as specific combinations of mutual beliefs and persistent goals. In their definition of responsibilities, agents can be responsible for a concrete action or sequence of actions (i.e. plans or sub-plans). Someone is *responsible to* someone else who wants the action done.

Fundamentally, responsibilities are asymmetric relationships *between agents* – particularly the agent that gave (delegated) the responsibility and the agents who have received the responsibility. In [9] a partial ordering of agents is defined with respect to the delegation of responsibility of a certain task θ. To illustrate, suppose that Θ is an action (or sequence of actions) and that agent A had responsibility for Θ but delegated it to B; we would then say $A >_{\Theta} B$ which can be read “B is responsible to A for performing Θ”. B can then further delegate Θ to C and we would say $C <_{\Theta} B <_{\Theta} A$. They note that these partial orders can be viewed as a directed graph (see figure 2).

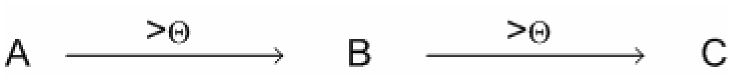


Figure 2 - Illustration of Responsibilities as a Graph

Ioerger and Johnson go on to define three fundamental types of responsibilities:

1. **Direct –** has a responsibility and does not delegate it: directResp(A, Θ) $\rightarrow$ resp(A, Θ) $\wedge$ ($\neg\exists$B A $>_{\Theta}$ B)

2. **Indirect –** has a responsibility and has delegated it to someone else: indirectResp(A, Θ) $\rightarrow$ ($\exists$B $\neg$(B=A) $\wedge$ A $>_{\Theta}$ B)

3. **Ultimate –** has a responsibility, has delegated responsibility to someone else, but is not in-turn responsible to anyone else ultimateResp(A, Θ) $\rightarrow$ resp(A, Θ) $\wedge$ ($\neg\exists$B B $>_{\Theta}$ A)

They also define concepts of RPGs (responsibility-persistent goals), and A-RPGs (accountable r-persistent goals). RPGs are goals in which agents with direct responsibility will not give up until the goal is achieved, unachievable or irrelevant. Agents with RPGs also have the option of delegating the responsibility to others. A-RPGs have the same conditions as RPGs, but agents are also committed to maintain mutual belief about the status of the goal (achieved, unachievable, irrelevant) with whoever delegated the responsibility.

This responsibilities framework can be used to facilitate the implementation of *procedural specifications* of actions that achieve goals that dictate when agents should accomplish a goal on their own, when they should wait for others to do it, when to inform others that they are going to fail, and how teams should recover from failure of a teammate. Most importantly, we will use this to implement procedural specifications that dictate when agents should monitor their teammate's responsibilities and proactively forward information needs of these teammates to them. We view information-needs of a goal/task Θ as a special type of derived goal Θ':

$$\Theta' = Goal_A\left(B_A(F)\right) \vee Goal_A\left(B_A(\neg F)\right)$$

where F = pre-conds(Θ). Note that while, at first glance $B_A(F)$ or $B_A(\neg F)$ may seem like a tautology (i.e. believing something is either true or believing that it is false), we also consider that the truth value of F can be *unknown* to the agent and therefore it is meaningful to state that agent A believes the truth value of a fact F (this is referred to as *knowing-whether*).

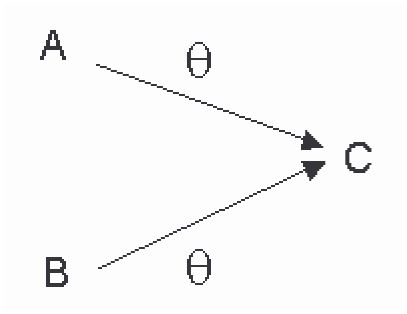


Figure 3 - Illustration of Multiple Agents Delegating Responsibility

We use an extended definition of responsibility that allows for teams of agents to delegate responsibility to sub-teams and also use concepts of joint intentions [18] to describe conditions for proactive information exchange. Our definition includes a team of agents T that is jointly committed to a goal θ, delegating responsibility for some sub-goal θ' of to an agent b∈T. Therefore, instead of responsibility being a unique set of pairs in PxP (where P is a set of agents), we consider that a subset of P can delegate responsibility to another subset of P (see figure 3).

We would then have {A, B} $>_{\Theta}$ {C}. We therefore expand the definition from of delegation (from Ioerger and Johnson) to teams. In Ioerger and Johnson, an individual agent *a* delegates responsibility for a goal θ to another agent *b*: *delegated(a,b,θ)*. We extend this by allowing teams of agents to delegate responsibility by adding the following axiom:

$a,b \in T\ delegated(T,b,\theta) \rightarrow delegated(a,b,\theta)$

This means that whenever a team T delegates the responsibility to a member of the team, essentially each member of the team has delegated the responsibility to the member and the normal axioms of individual responsibility apply. Having presented the responsibilities framework and our extension, we would like to use this to derive the following condition (which is similar to the conditions given in [16]). for proactive information exchange

$a,b \in T\ delegated(T,b,\theta) \wedge direct\text{-}resp(b,\Theta) \wedge Bel(a,\alpha) \wedge pre\text{-}cond(\alpha,\Theta) \wedge B_a(\neg B_b(\alpha)) \rightarrow$

$Intend(a,Tell(b,\alpha))$

The derivation of this condition will use our expanded definition of responsibilities and will trace an agent's *motivation* for exchanging information with teammates. The derivation follows the steps:

In the responsibilities framework, when *a* delegates responsibility for θ to *b*, *a* by definition has an indirect responsibility and therefore a persistent goal towards θ.

1. delegated(a,b,θ) → indirect-resp(a,Θ) → PGOAL(a, Done(Θ))

If ϕ is a precondition for the goal Θ, then as long as ϕ is not satisfied then Θ will never be done when ϕ is satisfies then Θ has the chance of being done. This establishes how preconditions will be used to infer that *b* needs to know ϕ.

2. pre-cond(ϕ,Θ) → $(\neg\phi \rightarrow \text{p}\ \neg Done(\Theta)) \wedge (\phi \rightarrow \neg\text{p}\ \neg Done(\Theta))$

If *a* believes ϕ and has a persistent goal for Θ being done (step 1) and if *a* believes that ***b*** has the direct responsibility for Θ and that *b* does not believe ϕ, then Θ will never be done (step 2); therefore *a* will have a persistent goal of *b* believing ϕ.

3. Bel(a, ϕ) ∧ PGOAL(a, Done(Θ)) ∧ pre-cond(ϕ,Θ) ∧ direct-resp(b, Θ) → PGOAL(a, Bel(b, ϕ))

If *a* has the goal of being in a state of mutual belief about a predicate ϕ and *a* believes that *b* does not believe ϕ, then *a* will have the intention of telling *b* ϕ.

4. PGOAL(a, Bel(b,ϕ)) ∧ Bel(a, ¬Bel(b, ϕ)) → Intend(a,Tell(b, ϕ))

A PGOAL of an agent *a* towards a goal Θ, as defined in [17], is an individual goal of *a* to do Θ until one of three conditions are satisfied: 1) the goal is achieved, 2) the agent believes the goal is unachievable, or 3) the goal becomes irrelevant. This derivation traces the path from an agent delegating responsibility to another to that agent's intention to communicate relevant facts to the agent with ultimate responsibility for the action.

There are also other approaches to formalizing the conditions for proactive information exchange. Grosz and Kraus [15, 20] describe how their axioms for shared plans lead to helping behavior. Also, in [29] Yen and Volz present a formal foundation of proactive information is presented derived from SharedPlan theory. They derive proactive information delivery behavior from the assist axiom in SharedPlan theory to describe agents motivation.

In this section we have described the proactive information exchange task from two perspectives. First, we listed the formal conditions (written in modal logic) for an individual agent to have a goal of informing another agent of a particular fact, then we built upon established formal team constructs of responsibilities and joint intentions to describe the motivation of team members to exchange information. In the next section, we will describe how common knowledge of a shared plan can help agents analyze each others information needs.

THE PIEX ALGORITHM AND IMPLEMENTATION

OVERVIEW OF APPROACH

In this section, we describe our approach to automatically generating communication within multi-agent teams. Our approach is based on analysis of a shared team plan and inserting communication acts (as calls to sub-plans) in a process of transforming the team plan into an individual plan. This involves more online reasoning than the transformational process used in Biggers' TIP-C algorithm, because agents must reason about the current belief state of the other agents on their team. PIEX is general enough to transform a shared team plan into an individual plan in most agent architectures, especially those based on hierarchical task networks (HTN) [19] in which tasks are decomposed hierarchically into subtasks until they ground out at atomic actions.

PIEX relies on three key steps to generate need-based communication between agent teammates:

1) Monitoring teammate responsibilities as encoded in a shared plan

2) Anticipating the information needs of those responsibilities, and

3) Monitoring what an agent believes his teammate believes.

The information need generation is done in a recursive way, analyzing the information needs of the highest task level (shared plans) and decomposing this task into sub-tasks, which in turn have sub-information-needs, until the tasks ground out at atomic actions, which in many cases also have information needs (i.e. their pre-conditions). This method is derived from the formal model of the team mental states described in the previous section. To illustrate our approach, we have implemented PIEX in a variation of the CAST agent architecture known as CAST-PM. CAST-PM operates on shared plans written in a teamwork modeling language called MALLET (described below).

SHARED PLANS AND INFORMATION NEEDS

Our approach is based on analysis of roles and responsibilities encoded in a shared team plan. Individuals are given responsibility to achieve certain goals, which in turn commit them to action (sub-plans or operators). We claim that information needs of agents can be automatically derived from analysis of the team plan; the following example illustrates this. Suppose two agents have a plan to rescue a hostage and their plan specifies that first, one goes around the building and disarms any alarm systems and then the other enters the building and rescues the hostage[3]:

```
(team-plan rescue-hostage
  (process
    (seq
      (do technician
        (seq (go-around-back)(disable-alarm))
      (do soldier
        (seq (enter-building)(rescue-hostage))))))
```

Implicit in this plan is that the soldier agent needs to know that the technician agent has disarmed the alarm before he can start his portion of the team activity. In other words,

[3] This is an example of MALLET syntax. It defines a plan named “rescue-hostage” the process of this plan specifies that, in sequence the technician agent first performs plan “go-around-back” then performs plan “disable-alarm”. When the technician agent is done the soldier agent performs, in sequence, the plan “enter-building” and then the plan “rescue-hostage”.

the fact that the technician agent has completed the “disable-alarm” plan is an information need of the soldier agent. In addition, the sub-plans of agents may encode information needs; in this case the technician should believe that there is no one around back before he goes around back and that the alarm is on before he attempts to disable it. The plan writer could explicitly encode communication steps for agents at given points in time, but this puts the burden of generating information needs on the person writing the plans. It would not only be timesaving but potentially more complete if agents could automatically infer the information needs of their teammates directly from the plans that they share.

The DIARG algorithm from the CAST project [16] is a preliminary implementation of the idea of proactive information exchange. DIARG generates sets of possible needers and possible providers of information through static analysis of operators. DIARG then analyzes the team plan structure (encoded as a predicate transition network) and decides who should ask and tell information in that context. PIEX differs from DIARG in three important ways. First DIARG is based on analyzing a static predicate transition network structure but PIEX does not require such a model. Second, DIARG analyzes only agents’ newly sensed pieces of information, whereas PIEX analyzes all information that agents currently believe. Finally, PIEX unlike DIARG uses a belief reasoning module to model the belief state of other agents in order to prevent communicating information that another agent may already believe.

BELIEF REASONING

As we have stated earlier, the second necessary component of proactive information exchange is the ability of agents to be reasonably sure that their teammates do not already believe the relevant information before they send it. This is because there may be a high cost of communication to the agents. In many computer-agent-only domains, the cost of communication is small and therefore agents can afford to send many (sometimes superfluous) messages to each other in order to ensure that the team is kept synchronized and information is properly distributed. A liberal proactive information exchange strategy could be for each member of a team, upon coming across a new information fact *I*, to broadcast *I* to all teammates. This may seem appropriate for domains in which the cost of communication is low. However, there are some domains in which communication may be costly; for instance, if agents are trying to maintain radio silence or if there is a high computational cost to process new information messages, then we would like to limit our communication to only the most important, relevant and needed information to each teammate. Additionally, when we have mixed human-agent teams the cost of communication is clearly higher, as humans have limited attention resources and cannot process a constant stream of communication and perform well at their given task at the same time.

While communication is important to teamwork, in psychological studies [24] it has been observed that under particularly high workload (or high tempo operations), communication in the most effective teams can actually decrease, presumably because team members begin to rely on more well-developed shared mental models and can infer what other teammates already believe or can figure out for themselves. We would like to take advantage of this type of shared mental model such that agents resist informing other agents of facts when they are reasonably sure the others already believe it. Accordingly, the PIEX algorithm requires an ability to simulate multi-agent *belief reasoning*, that is, agents maintaining a model of their teammate's beliefs. For example, it is relatively straightforward for an agent *A* to believe that agent *B* believes something if *A* has told *B* about it (or if *B* told *A* about it).

We use a method of multi-agent belief maintenance, called BOA. BOA is implemented in Java and is incorporated into our CAST-PM agent architecture. One advantage of BOA over traditional theorem-provers like JARE (an implementation of PROLOG in Java), is that BOA allows us to represent more than true or false states of belief; we can also represent that the state of an agent belief is *unknown* or *whether*. The belief state of whether means that we know that the agent believes the truth value of a fact, but we do not believe ourselves what that truth value is. For instance, if an agent A watches another agent B go into a room, agent A will believe that B believes *whether* light-on(room) is true or false, which is like saying ¬(unknown B (light-on(room)))[4]. BOA allows a knowledge engineer or agent designer to define *justifications* for different beliefs that agents may have in different situations. For instance, an important method of inferring what other agents believe is by reasoning about observability in the environment – there are many cases in which agents can see (sense) what others see, and can use that to make inferences about what they believe. Another justification for belief

[4] Semantics for (unknown B F) = ¬(BEL B F) ∧ ¬(BEL B ¬F) = ¬(WHETHER B F)

are the results of actions: often actions have known effects, and if an agent believes that another agent has completed a task, then the agent can believe that the acting agent believes the effects of those actions have occurred. Other justifications include persistence rules (to define what facts persist over time), inference rules, defaults, facts, and others. These justification rules are loaded in as a file and the BOA system processes agent senses, resolves any possible dependencies or conflicts between justifications and maintains the multi-agent knowledge base. The BOA file format and notes about BOA system are listed in Appendix G. An text-dump of an agent BOA knowledge base is listed in Appendix F.

Given the distributed nature of many multi-agent domains, it may not be possible to believe *for certain* what another agent currently believes, and therefore BOA provides a best-approximation of other agents' beliefs. Belief reasoning is very important to the PIEX algorithm because it allows us to evaluate the predicate `(BEL A ¬(BEL B I))` in the formal conditions for proactive information exchange. Belief reasoning is essential to simulating effective information exchange, particularly in domains in which the cost of communications is high, having mentioned the importance of belief reasoning to PIEX, we will not go into the details of this belief algorithm any further.

OVERVIEW OF MALLET

There are many multi-agent languages developed for specifying teamwork processes [12, 15, 22]. Each language has advantages and disadvantages and our PIEX algorithm should be general enough to work with almost any of these team process languages. Our reference implementation of PIEX analyzes plans written in the MALLET teamwork specification language. MALLET is a logic-based language that allows for description of both team structure (agents, roles, goals, capabilities) and team processes (plans). MALLET uses a LISP-like syntax (nested s-expressions).

Actions that agents are able to take in the world are defined in terms of operators. Individual operators (i-opers) are those atomic actions that an individual agent can execute in the environment. Operators, as well as plans in MALLET have STRIPS-style preconditions and effects [30]. The following is an example of an operator that moves an agent in a given direction:

```
(i-oper move (?direction)
  (pre-cond (at self ?cur-x ?cur-y) (can-move ?direction ?new-x ?new-y))
  (effects (at self ?new-x ?new-y) (not (at self ?cur-x ?cur-y)))
)
```

The above is an example of mallet syntax, variables are indicated with a '?' prefix and are bound by either the underlying theorem prover (for queries, like in the preconditions) or when parameters are passes (operators and plans are invoked like functions, so in the above ?direction is bound by passing a parameter when invoked). In an implementation of MALLET (like the CAST-PM architecture described in the next section) there is a theorem prover that unifies the variables against a logical knowledge base. Preconditions represent what must be true in the agent's knowledge base just before an action takes place; effects are what will hold directly after the action takes place (both are important for inferring information needs of other agents in shared plans). In the example above, before an agent moves, there should be a fact in its knowledge base like (at self 5 4). "can-move" is a rule, or predicate that takes two arguments `?new-x` and `?new-y`, the new coordinates of the agent after moving in the specified `?direction`, inferred by the rule.

Operators indicate the atomic actions in the plan hierarchy. Plans are more complex process descriptions – they are like operators except they also define a process definition (plan body). Plans can call sub-plans or operators by connecting them in various constructs such as *seq* (sequential), *par* (parallel), *while* (iteration), *if* (contingency), etc. Plans in MALLET are defined using the following grammar:

```
(plan <planName> (<var>*)
  [(pre-cond <cond>+)]  // pre-conditions for execution
  [(effects <cond>+)]   // effects of execution
  [(term-cond [FAILURE|SUCCESS] <cond>+)] // default is FAILURE
  (process (<proc>+))

<proc> ::=
  (seq <proc>+) |                   // sequential execution
  (par <proc>+) |                   // parallel execution
  (if (cond <cond>+) <proc> [<proc>]) |  // conditional statement
  (while (cond <cond>+) <proc>) |  // do-while loop
  (foreach <cond> <proc>) |        // iterates sequentially
  (forall (<cond>+) <proc>) |      // iterates in parallel
  (choice <proc>+)                 // handles failure
```

In addition, team plans – which specify joint actions among members of the team, are specified using the following grammar[5]:

```
<team-plan> ::=
  (team-plan <name> (<var>+)
    [(pre-cond <cond>+)]
    [(term-cond [SUCCESS|FAILURE] <cond>+)]
    [(effects <cond>+)]
    (process <tproc>)
  )

<tproc> ::=
  (seq <tproc>+) |
  (par tproc>+) |
  (do <agent> <proc>) // agent specified by <agent> is to do <proc>
```

For the most part, team plans look very similar to other plans. There are two important features which distinguish a team plan from an individual plan. First, the addition of the *do* construct allows teams to specify *which agent* will perform the specified procedure. Also, only *seq* (sequential action) and *par* (parallel action) are allowed to wrap around these ‘do’ constructs. Our implementation of PIEX determines how the MALLET implementation (agent interpreter) executes ‘do’ constructs. Complete MALLET syntax is provided in Appendix A. Next, we will describe our concrete agent implementation that interprets MALLET plans, known as CAST-PM.

OVERVIEW OF CAST-PM

CAST-PM is a derivative of the CAST agent architecture from Texas A&M University

[5] The version of MALLET that we are using is a slightly modified version of the *official* MALLET specification from Texas A&M. One of the major modifications is the addition of the <team-plan> construct to distinguish team from individual plans. Also, there is no role specification or agent-binding.

[16]. Both CAST and CAST-PM are multi-agent architectures designed to simulate teamwork, primarily thorough the simulation of shared mental models. The major difference is that CAST-PM is based on the Process Manager agent kernel and CAST is based on predicate transition networks. Process Manager is an algorithm that models the state of an agent's tasks, plans, and operators in a hierarchical way (as a tree), similar to the HTN networks in RETSINA [19, 31]. Different node types represent different types of agent process control sequences and operators. A plan node is represented by a "control" node, that is a node that is not an atomic action but a way of combining and controlling the flow of actions. Examples of control nodes are: `while, if, choice, for-each, for-all, seq, par, start`, etc. These are the direct implementation of the control-flow constructs in MALLET. All nodes contain a status tag that keeps track of the current status of that task. For instance, for atomic primitive actions (leaf nodes in the tree), nodes are marked ‘in progress’ until they are either completed successfully and marked *done* or they fail and are marked *fail*. The status of control nodes are updated according to the status of its children in a bottom-up, recursive way. The semantics of different node types are defined in terms of recursive functions *node.expand(node)* and *node.repair(node)*. Nodes get repaired at each time step, this involves checking the state of the node with respect to the current knowledge base (i.e. for while nodes the condition gets checked at each time step, plan nodes termination conditions are checked at each time step). At the beginning of each time step the process manager kernel calls *repair* on the root node, which calls *repair* on each of it’s child nodes, which in turn repair each of their child nodes. This recursive repair process

continues until all branches ground out at atomic actions in the world (which do not get repaired) and the recursion unwinds to the root node of the tree. As part of the repair process, both success and failure statuses are propagated up the tree (a special *choice* node type catches failure, similar to exception handling in other programming languages). Control nodes are expanded when they are first created and also when their children are complete. Control nodes are expanded and repaired according to their node type; below we give pseudocode for the expansion and repair steps for *seq* (sequential) and *par* (parallel) nodes.

```
Expand(node):
  if type(node)='par' then:
    for each $\alpha_i$ where proc(node) = par($\alpha_1$,…,$\alpha_n$):
      create new node($\alpha_i$) = $n_i$
      addchild($n_i$)
      expand($n_i$)
  if type(node)='seq' then:
    $\alpha_i \leftarrow \min_{\alpha_i \in seq(\alpha_1,\ldots,\alpha_n)}$(status($\alpha_i$)!='done')
    create new node($\alpha_i$) = $n_i$
    addchild($n_i$)
    expand($n_i$)
else if type = 'oper' then:
  create leaf node and mark as open

Repair(node, status):
  parent $\leftarrow$ parent(node)
  if status='fail' then:
    unlink node from parent
    repair(parent, fail)
  else if status='success' then:
    if all children done then:
      unlink node from parent
      repair(parent, success)
    else if type(parent)='seq' then:
      $\alpha_i \leftarrow \min_{\alpha_i \in seq(\alpha_1,\ldots,\alpha_n)}$(status($\alpha_i$)!='done')
      create new node($\alpha_i$) = $n_i$
      addchild($n_i$)
      expand($n_i$)
    else if type(parent)='par' then: continue
```

Control nodes can expand into both control nodes and operator (primitive action) nodes as children. The type of control node defines the way in which its children are expanded. Operator nodes are always the leaves of a tree. At any time, the current intentions of an agent can be found by gathering the operator leaves of the tree that are not marked as failed or done.

This provides a simple yet powerful way to model how an agent schedules its actions, tasks, plans and operators. CAST-PM uses BOA as its theorem prover, BOA is a forward-chaining theorem prover (similar to CLIPS) but is also a multi-agent belief maintenance system. BOA not only maintains the agent's knowledge base, it also allows the agent to query. assert and retract facts from the ‘self’ knowledge base and also other agents knowledge bases. BOA is called to evaluate the conditions in statements such as *if* and *while.*

CAST-PM is implemented in Java and provides the ability to write domain-specific *adapter classes* which act as interfaces to simulation environments. While agent designers write team plans in MALLET, and CAST-PM interprets those plans to decide on actions in an environment, CAST-PM itself does not provide the ability to interface directly with the agent domains. Instead agents interface with dynamically loadable adapter classes which in turn provide the domain application functionality layer – such as API functions, RPC (Remote Procedure Calls), or network message passing. This separation of agent reasoning from domain-specific actions has allowed us to *plug-in* CAST-PM agents to many simulation environments, including the networked team game *Unreal Tournament*, the *Robocup Soccer* simulator, and our own simulation environments without having to rewrite the code that controls the agents reasoning processes.

Next, we will describe how PIEX is implemented in CAST by illustrating how agents derive information needs from MALLET team plans and how these information needs are incorporated into the agent's individual plan to proactively inform teammates.

DERIVING INFORMATION NEEDS FROM TEAM PLANS

Our implementation of the PIEX algorithm in CAST uses the following procedure to anticipate the information needs of a teammate to perform a task. We use a method that approximates all possible information needs that this other agent requires for its assigned process. Below is a pseudo code description of the algorithm we use for generating information needs stored in a knowledge base from a MALLET plan. In this method, *proc* is the current MALLET construct (list) being processed. When an agent comes across a *do* construct (responsibility) in a team-plan for which it is not responsible, it calls itself recursively to detect all information needs of the responsible agent:

```
generateInfoNeeds(List proc, String agent, int id){
  if(proc.key=='seq'|'par'|'choice'){
    for(each process in <proc>+) generateInfoNeeds(process, agent, id);
  }
  else if(proc.key=='while'){
    generateInfoNeeds(proc.process, agent, id);
  }
  else if(proc.key=='foreach'|'forall'){
    for(each binding in proc.cond){
      bound-process=binding.unify(proc.process);
      generateInfoNeeds(bound-process, agent, id);
    }
  }
  else if(proc.key=='if'){
    cond = <if-condition>
    assert(“info-need ” id “ ” cond);
    generateInfoNeeds (proc.trueBranch(), agent, id);
    generateInfoNeeds (proc.falseBranch(), agent, id);
  }
  else if(proc.key=planName|tPlanName){
    new_id = new_unique_id();
    // see description of ‘do-id’ fact in section 7
    assert(“do-id ” new_id “ ” id “ “ agent “ “ prec.key);
    for(each cond in <pre-cond>,<term-cond>,<effects>)
      assert("info-need " new_id " " cond);
    generateInfoNeeds(proc.process, agent, new_id);
  }
  else if(proc.key=operName){
    for(each cond in <pre-cond>,<effects>)
      assert("info-need   " id " " cond);
  }
}
```

Note that whenever a new plan is expanded, it is given a new id. Each direct information need of that plan is tagged with a ‘do-id’ fact in the KB that marks the id of the plan that it is an information need of (see next section for more description). Sub-plans are given new ids, but also tagged with the parent id. When it is observed that an agent has completed a plan, the information needs of that plan are retracted; in addition, all sub-information needs of the plan are also retracted. This allows the agent to prune information needs that are no longer relevant (because when parent plans are observed to be complete, the sub-information-needs are no longer relevant).

In order to avoid exchanging *obvious* facts or functions in the world, we also define facts that are *common knowledge* among all agents, such as procedural attachments (i.e. functions such as '=','+','>','cons',etc), team facts that everyone knows *(self ?self)*, *(role ?agent)* and sensory facts that agents assert directly from the environment (i.e. *(hear ?msg) (at self ?x ?y) (see ?object ?x ?y)* etc.). The generate-info-needs step ignores these facts when they appear, to avoid sending obvious information to others.

PIEX IMPLEMENTATION IN CAST-PM

As previously described, MALLET defines special constructs for individual and team plans as well as *do* constructs for assigning responsibilities. Implementing MALLET in CAST-PM involves creating *node-types* for each construct (plan, seq, while, etc.) that get expanded and repaired in a Process Manager tree in a way that is consistent with the semantics of those constructs. Our implementation of PIEX involves implementing *do* nodes in MALLET. A *do* statement is part of a team plan (see above) that specifies a responsibility, that is - both a task and the agent that is to perform that task. A simple team plan might look like this:

```
(team-plan build-campfire
  (process
    (par
      (do ryan (gather wood))
      (do mark (shop-for (graham-crackers chocolate marshmallows)))
      (do jim (clear area))
    )
  ))
```

This specifies that, in parallel, Ryan, Mark and Jim will each perform a task in order to complete the team plan. Each member of this team (Ryan, Mark and Jim) executes this same plan. The root of each agent's process tree is a *team-plan node* which has one child *par node* which in turn has three *do node* children. These team plans are transformed into individual plans that each agent executes independently. For each of the three *do-nodes* expanded in parallel, the agent checks to see if it is the agent assigned to the task. If it is the agent assigned to the *do* task, then they do the task and broadcast that they have completed the task when they are done (or failed, if they fail at the task). If the individual agent is not the responsible agent specified, the agent derives all information needed for the assigned task and all subtasks (using the routine given in the previous section) and monitors his own knowledge base for opportunities to proactively inform his teammate of these information needs (until he hears that the subtask is complete).

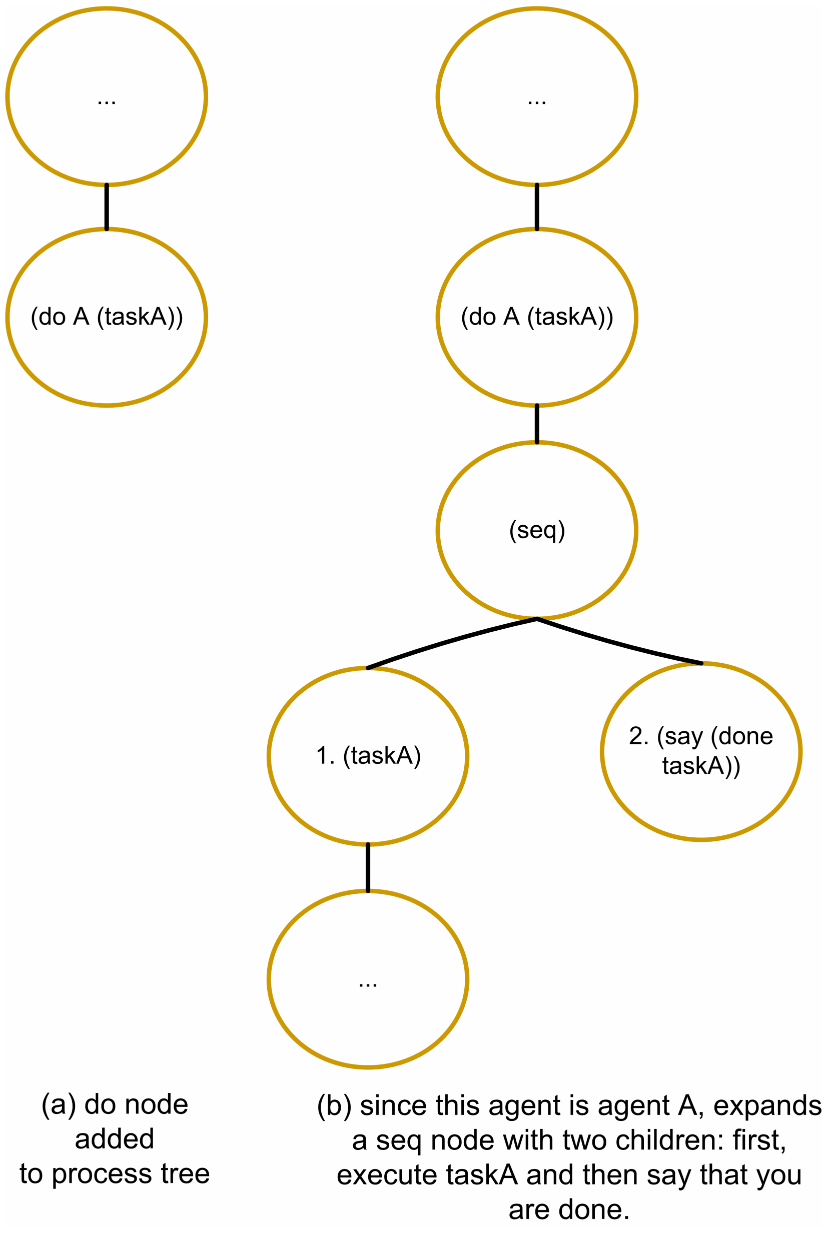

...
(do A (taskA))
...
(do A (taskA))
(seq)
1. (taskA)
2. (say (done taskA))
...
(a) do node added to process tree
(b) since this agent is agent A, expands a seq node with two children: first, execute taskA and then say that you are done.

Figure 4 - Process Tree States for Executing Own Agents Task

These figures illustrate how do nodes are expanded in the process trees. In figure 4, we can see that, if the agent is the team member specified in the do node, the agent will expand his do node into a *seq node* (sequence node) with two children: first the agent performs the specified action and then he notifies his teammates that he is done with the action.

Figure 5 shows that if the agent is not the specified agent in the do node, then he calculates the information that his teammate (the agent specified in the do node) needs for that plan and then adds an active-inform plan-node to his process tree. The active inform plan, while waiting for a ‘done’ message, actively queries the agents knowledge base at each moment to check if he believes any of the information needs of his teammate and, when he does, forwards that information (but only if he believes that his teammate does not already believe the information about to be sent).

At a more technical level, here is how the process works: When the agent comes across a *do* node for which it is *not* the agent specified, it first increments a unique id counter and tags its teammate’s do-assignment by asserting a fact of the form:

```
((do-id <id> <parent-id> <agent> <spec>))
```

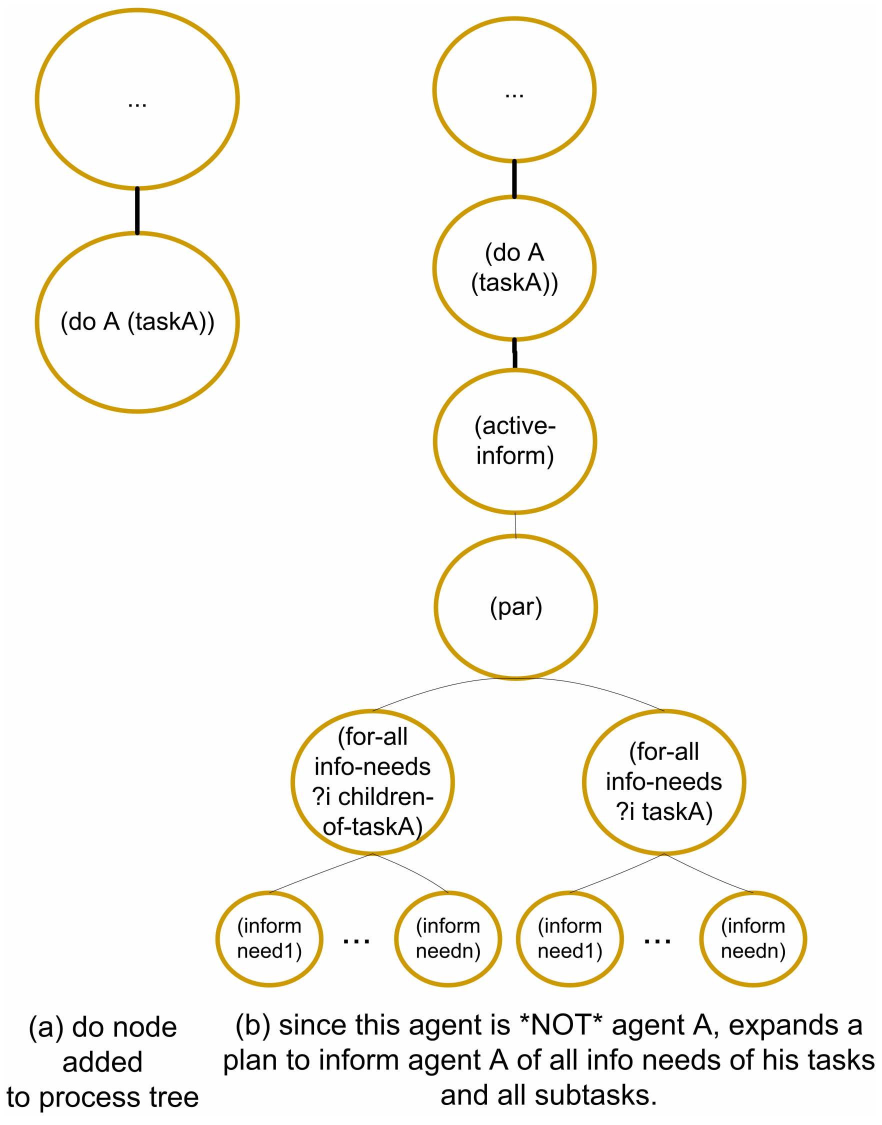

...
(do A (taskA))
...
(do A (taskA))
(active-inform)
(par)
(for-all info-needs ?i children-of-taskA)
(for-all info-needs ?i taskA)
(inform need1)
...
(inform needn)
(inform need1)
...
(inform needn)
(a) do node added to process tree
(b) since this agent is *NOT* agent A, expands a plan to inform agent A of all info needs of his tasks and all subtasks.

Figure 5 - Process Tree States for Executing Other Agents Task

Where <id> is the unique id of the plan, <parent-id> is the parent plan's unique-id if this is called as a sub-plan (null if this is the top level plan), <agent> is the agent assigned to do the action and <spec> is the specification given in the *do* statement for action. For example

```
((do-id 46 40 mark (gather firewood)))
```

Here the plan (gather firewood) is assigned to mark and is given the id of 46, it is the child of the do task with id 40. After the task is tagged with an id, there is a `generate-info-needs` step (described in the previous section) that infers the information needs of the other agent's task and asserts them as these types of facts

```
((info-need <id> <cond>))
```

For example

```
((info-need 46 (loc firewood ?x ?y)) // comes from precond
                                     // of gather firewood
((info-need 46 (loc gloves ?x ?y))
```

Where <id> is the unique id of the plan specified as part of the do-id and <cond> is an information need of <agent> to do his task (indexed by int). There will be 0 to many of these facts, depending on the task of the other agent.

Finally, there is a plan that will get added as a child to the *do* node for the agent that is *not* directly responsible. The plan specifies that the agent waits until he hears that the task for his teammate is completed and, while he is doing so, monitors his knowledge base for opportunities to communicate needed information to his teammate (provided he doesn't believe that his teammate already believes this information). This is a generic plan in MALLET, which gets expanded to form the sub-tree in figure 5.

```
; Plan Active-Inform: for PIEX
(plan active-inform (?id)
  (pre-cond (do-id ?id ?parent ?agent ?proc))
  (term-cond SUCCESS (complete ?proc))
  (effects (not (do-id ?id ?parent ?agent ?proc))
           (not (info-need ?type ?id ?pred)))
  (process
    (par
      ; active-inform all info needs of this plan
      (forall ((info-need ?type ?id ?info-need))
        (seq
          (while (cond (not ?info-need)
                       (needs-info ?agent ?type ?id ?info-need))
            (NOP))
          (if (cond ?info-need
                    (needs-info ?agent ?type ?id ?info-need))
            (say ?agent ?info-need))
        )
      )
      ; create another active inform plan for all children of this plan
      (forall ((do-id ?child-id ?id ?child-agent ?child-spec))
        (active-inform ?child-id)
      )
    )
  )
)

; BOA rules – says that an agent only ‘needs’ information when it
; doesn’t already know it – important for filtering communication

(defrule (needs-info ?agent ?type ?id ?info-need)
    (do-id ?id ?parent ?agent ?proc)
    (info-need ?type ?id ?info-need)
    (not (bel ?agent ?info-need)))
(defrule (needs-info ?agent ?type ?id ?info-need)
    (do-id ?id ?parent ?agent ?proc)
    (info-need ?type ?id ?info-need)
    (not (bel ?agent (whether ?info-need))))
```

It is interesting to note that the *active-inform* task for monitoring information needs, analyzing teammate's beliefs, informing teammates of information and cleaning up information needs and sub-information needs of completed tasks is *itself written in MALLET*. This can be loaded in as a module and makes it easier to experiment with different communication policies. Appendix D shows a visualization of an actual Process Manager tree running PIEX and the active-inform plan.

The process manager network structure (hierarchical task network) itself also aids in maintaining the information needs of teammates. Keeping track of parent-child relationships in information needs is useful for retracting information needs and sub-information needs when they are no longer relevant. For example, if a plan p_1 has a sub plan p_2 and p_2 has information needs a, b, and c, and the agent learns that p_1 is complete, we can be sure that the agent no longer needs to perform p_2 and therefore, no longer needs to know a, b, or c and they can be retracted from the set of information needs for that agent. Since the parent-child relationship of plans/sub plans are explicit in the Process Manager structure, so are the information-needs/sub-information needs of agents. So when a parent (plan/info-need) node is pruned from the tree, so are all its descendants.

In this section, we have described the implementation of the PIEX algorithm in CAST-PM. Next, we will describe the proactive information exchange validation framework developed as part of this research.

VALIDATION FRAMEWORK

We have recognized the need for an effective description and evaluation framework to both describe scenarios in which proactive information exchange would take place and to validate the proactive information exchange communication events that agents perform. The following is a formal framework for describing, from an omniscient point of view, information-exchange scenarios for agent-based teams. We describe the scenarios in terms of the agents involved, the facts relevant to those agents, the information needs of each agent, and the event-times at which each agent will come to learn each event. In addition to being a descriptive tool, it also prescribes an *optimal* communication sequence for information exchange between agents on the team. Note that this formalism should *not* be confused with PIEX, the actual information-exchange algorithm to control the communications of agents working under conditions of limited information. Instead, this is a useful framework for discussing the aspects of information exchange from an omniscient point of view. This can also be used as a performance metric in order to evaluate PIEX as a communication policy for information exchange among agent team members. One could do this validation in a *post-hoc* way, by running agent team members in a simulation environment and then determining the information needs of each agent, analyzing the sensory-events that agents received at each moment and finally, the communications that they send to each other. Alternatively, we could use this framework to specify the simulation environment from the start and evaluate their communications in the same way (this is the way in which we run our experiments in this thesis). In the final section of this proposal, we describe the method we use to evaluate PIEX.

Once again, it is important to remember that this formal description specifies scenarios at an omniscient level of knowledge, and since we assume that no agent has access to this level of knowledge, this formalism is not meant to specify *how* agents should communicate to exchange information, only to describe the environment that the agents will inhabit and the optimal communication sequence of these agents. Additionally, this formalism is independent of the agent architecture used to implement information exchange. In the following pages, we will describe the formal framework for specifying information exchange scenarios, show that we can define and derive the optimal communication sequence from this description under different optimality conditions, show how we can evaluate actual communication according to the optimal sequence, give an example scenario and finally discuss limitations and possible extensions to the model.

DESCRIPTION

The following is a formalism that allows us to set up and validate our scenarios:

S. S = <A, F, info-needs, events, inferences>

A scenario is a quintuple of: *A* – agents, *F* – facts, *info-needs* – sets of facts relevant to each agent, events is a set of *information events* for each agent, and inferences are the inference rules that agents use to make logical deductions.

A. $A = \{a_1, a_2, \ldots, a_n\}$

Set of all agents on a the team

F. $F = \{f_1, f_2, \ldots, f_m\}$

Set of all facts (propositions) in the domain required by some agents to complete their tasks. These may be actual information needs or antecedents used to infer information needs.

info-needs. $(\forall a \in A : (\text{info-needs}_a = \{f \in F, \mid a \text{ requires } f \text{ to complete a task}\}))$

info-needs$_i$ assigns, for each agent, a set of facts $f_j \in F$ that agent a_i needs to complete his individual task. The semantics of this is that, without this information, a_i will never be able to complete his task. We assume that agents will not act on an action until they believe the preconditions hold.

info-events. $(\forall a \in A : (\forall f \in F : (\text{events} = \{<a,f,t> \mid t = \text{moment at which a discovers } f_j$ independently, or through its own sensors$\})))$

events assigns, for each agent, a set events$_i$, which is a set of *information sensing event* tuples <a,f,t>. Events are a mapping from $A \times F \rightarrow \mathbf{N} \cup \{-1\}$, where **N** is the set of natural numbers. For agent a and each fact $f \in F$. events gives the time step t that the a will discover f *independently*, that is without communication from other agents (or –1 if the agent will never come across that information on its own). |events| = m. The semantics for this is that for each $<a_i,f_j,t>$, $B_a^t f$ meaning that a will believe fact f at time ≥ t (and all other times after that). We assume that knowledge bases are static, that is facts do not change truth value over time.

inferences. $\text{inferences} = \{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_x \rightarrow c \mid \alpha_1 \ldots \alpha_x, c \in F\}$

inferences are a set of rules that specify the logical conclusions that the agents can draw[6], such as:

mother(Jane, Julie) ^ mother(Julie, Helen) $\rightarrow$ grandmother(Jane, Helen)

Rules are encoded as Horn clauses, where each conjunct of the antecedent and the consequent of each rule are members of F. For the rule α_1^α_2^…^α_x $\rightarrow$ c we denote antecedent(α,c) as the fact that α is an antecedent of an inference rule with the consequent *c*. The semantics is that, if an agent believes all the antecedents of a rule, then it will believe the consequent. That is

$$\left\{\alpha_1 \wedge \ldots \alpha_n \rightarrow c\right\} \in Inferences_A \wedge \wedge_{i=1}^{n} B_A^t \alpha_i \rightarrow B_A^t c$$

By using this method of specifying scenarios for our agent teams we can represent our scenario as a directed graph with agent and fact nodes. Edges emanating from a fact node f_j (circle shaped) to an agent node a_i (square shaped) represent an *information need* for fact f_j of agent a_i. Each member of *info-needs*$_i$ for all a_i is an edge from a fact node to an agent node. Each edge emanating from an agent node a to a fact node f represents an information event $<a, f, t> \in events$. Each member of the set events for all a is an edge from an agent node to a fact node, these edges are labeled with the value of <a, f, t>, where t is the time in which a will discover f without help from his teammates (-1 if he will never learn this).

Using this formalism we can say that, if there is no $event_{ij} = <a_i, f_j, t>, t = -1$ where $f_j \in$ *info-needs*$_i$, then each agent can carry out their own tasks independently, without the

[6] We say that inference rules are common knowledge among all of the agents

help of other agents, as each agent can carry out their own tasks independently, without the help of other agents[7]. There may still be an advantage for information-exchange communication, though, because a teammate may learn information needed by another agent *before* the other agent learns it. Formally, for some fact $f_j \in$ info-needs$_i$ there could be an information event event$_{kj}$ = $<a_k, f_j, t_{kj}>$ and event$_{ij}$ = $<a_i, f_j, t_{ij}>$ such that $t_{kj} < t_{ij}$, which should trigger a communication of information f_j from a_k to a_i enabling the agent to believe it earlier than otherwise.

A SIMPLE EXAMPLE

To illustrate, let us consider a simple scenario with 2 agents, 3 facts and no inference rules.

$A = \{a_1, a_2\}$

$F = \{f_1, f_2, f_3\}$

info-needs$_1$ = $\{f_1\}$

info-needs$_2$ = $\{f_2, f_3\}$

Event Mapping $A \times F \rightarrow N \cup \{-1\}$:

	f_1	f_2	f_3
a_1	5	3	-1
a_2	4	2	3

inferences = {}

[7] Conversely, if there is any fact $f_j \in$ info-needs$_i$ and an event $<a_i, f_j, t> = -1$, then there is no way that agent a_i can complete their individual task on their own.

The graph in figure 6 encodes this scenario[8].

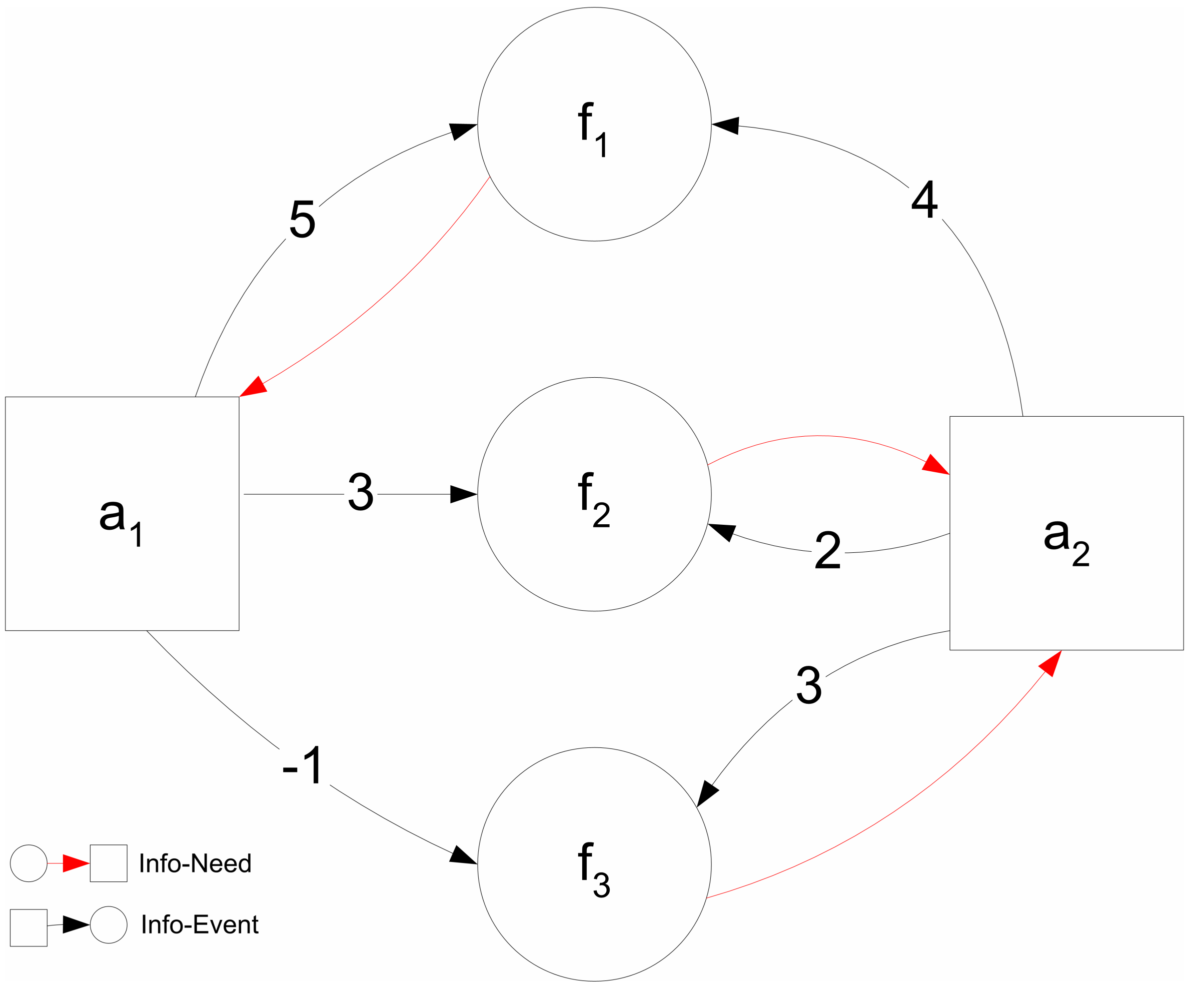


Figure 6 - Information Need/Event Graph

In this example, there is an opportunity for communication from a_2 to a_1 of fact f_1 at t=4 because there exist two information-events $<a_2, f_1, t_{21}>$ and $<a_1, f_1, t_{11}>$ such that $t_{21} < t_{11}$ and $f_1 \in$ info-needs$_1$.

[8] In larger graphs, we will probably want to leave off event edges labeled –1 (i.e. no information event)

OPTIMAL COMMUNICATION

This framework allows us to specify the *optimal* communication of teammates working in a scenario specified the formalism above. That is, we can define a policy which can be used to derive what a team will optimally communicate in any scenario. In order to do this, we need to determine exactly what we mean by the optimal communication sequence. This is relatively straightforward if we do not consider inference rules in giving the optimal communication sequence. In this case, the agent who happens to believe an information-fact first (and only that agent) should tell all others who need that information as soon as he discovers this fact. Implicit in our optimality definition is that it is better to perform fewer communications-actions and to believe information needs sooner. In other words, the optimal communications in a setting that ignores inference achieves the minimal number of communication messages possible while informing each agent of his information need as soon as it is available within the team.

The addition of inference rules, while increasing the richness of the scenario description, adds ambiguity as to what the optimal communication sequence is. We can define different optimal communication policies for how to handle implication rules combined with the facts that agents believe at different times. As an example domain, say that:

1) $a, b, c \in$ A

2) $r \in$ info-needs$_c$

3) $(p \wedge q \rightarrow \text{r}) \in$ inferences

4) $B_a^t(p)$ and $B_b^t(q)$.

In this scenario, neither *a* nor *b* can infer *r* and therefore, cannot tell *c* its information need outright. However, between the two of them, *a* and *b* have information for which *c* can infer *r*. The decision about what agents should optimally communicate to each other based on their current knowledge and the information needs of other agents is defined in an *optimal communication policy*. The definition of optimal information-exchange could vary in different domains, we therefore define 4 different *policies* of optimal communication.

a. POLICY 1 – MINIMAL COMMUNICATION MODEL

Ignores inference rules and simply states that if an agent *a* needs fact *I*, and agent *b* specifically believes *I* at time *t* before anyone else, that *b* should inform *a* of *I* at *t*.

$$\exists a,b \in A\ ,\neg\exists c \in A,\ \exists t \in \mathbf{N} \wedge B_b^t(I) \wedge (\neg\exists c \in A,\ \neg\exists t^* \in \mathbf{N} \wedge\ I \in \text{info-need}(a) \wedge (a \neq b\ ) \wedge B_c^{t^*}(I) \wedge (t^* < t)) \rightarrow \text{TELL}(a,b,I,,t)$$

In the example above involving inferences, neither agent *a* nor *b* will communicate with *c* because neither believes the information-need *r* outright. *a* or *b* will only communicate *r* when they personally have the information *r*.

b. POLICY 2 - LIBERAL COMMUNICATION MODEL

Follows the communications rule in model 1. Also, if an agent *b* believes an antecedent *N* to an information need *I* of another agent *a* at time *t*; *b* will tell *a N* at time *t*.

$$\exists a,b \in A,\ \exists n,I,J \in F,\ \exists J \in \text{Inferences} \wedge I=\text{consequent}(J) \wedge I \in \text{info-need}(a) \wedge \text{antecedent}(n,J) \wedge (\neg\exists t^* \in \mathbf{N},\ \exists t \in \mathbf{N} \quad \wedge\ B_b^t(n) \wedge B_c^{t^*}(n) \wedge (t^* < t)) \rightarrow \text{TELL}(a,b,n,,t)$$

In the example above *a* will tell *c p* and *b* will tell *c q*, even though they do not believe that any other agent believe the remaining antecedent-facts. Therefore, *a* would still inform *c* of fact *p* even if *b* did not believe *q*.

c. POLICY 3 - ORACLE-AGENT MODEL

Requires agents to have access to all other agents' knowledge. If an agent *a* needs fact *I* that either (a) some agent believes directly or (b) that all agents together have enough to infer, *a* should be informed (by one agent) of *I* at time *t*. In the example above *A* would tell *c* fact *r* only when *a* believes that *b* believes *q*. Note that this requires more insight than we normally attribute to agents, since it requires *a* to believe *b* believes *q* without *a* believing *q* itself (since it would, by implication, believe *r* and the rule in model one would apply).

$\exists$a,b$\in$ A, $\exists$I$\in$ F, $\exists$t$\in$ **N**, $\neg\exists t^* \in \mathbf{N} \wedge (\bigcup B^{t}_{a_i \in A} \models I) \wedge I \in$ info-need(a) $\wedge(\bigcup B^{t^*}_{a_i \in A} \models I) \wedge (t^*$

$< t) \rightarrow$ TELL(a,b,I,,t)

In the example above, either agent *a* or *b* should inform *c* of *I* at time *t*.

d. POLICY 4 - LIBERAL ORACLE MODEL

The liberal-oracle model has elements of both the liberal and oracle models. Agents may communicate if they only believe antecedents to information-needs and do not believe the information-need outright (unlike the minimal communication model). Agents will only communicate what they personally believe (like the liberal but unlike the omniscient model, they will only communicate facts that are in their own personal knowledge base), but will not communicate these facts until they believe that all of the antecedents of the information need are satisfied by the entire team (like the omniscient model and unlike the liberal).

$\exists$a,b$\in$A, $\exists$t$\in\mathbf{N}$ $\wedge$ J$\in$Inferences $\wedge$ I=consequent(J) $\wedge$ I$\in$info-need(a) $\wedge$ ($\forall$n$\in$F $\wedge$ n$\in$antecedents(J) $\bigcup B^{t}_{a_i \in A}$ |= I) $\wedge$ ($\exists$m$\in$F, $\neg\exists$t$^*\in\mathbf{N}$ $\wedge$ m$\in$antecedents(J) $\wedge$ B^{t}_{a}(m) $\wedge$ $B^{t^*}_{c}$(m) $\wedge$ (t* < t)) $\rightarrow$ TELL(a, b, m, t)

In the example above, agent *a* will inform *c* of *p* if and only if, among the other agents *b,c* the fact *q* is also known. This model generates more communication than the omniscient model, since more facts are communicated, but potentially less than the liberal model because no communication occurs unless the union of all of the agents' knowledge bases entails the information need fact.

CALCULATING THE OPTIMAL COMMUNICATION SEQUENCE

All of the policies above use the same formal framework for describing the scenario. The models differ in the optimal communications sequence specified for a given scenario. For our experimental results, we use policy 1 optimality conditions. Below is the algorithm which shows how we calculate the optimal communication sequence using model 1[9].

```
Set T = $\emptyset$
Set Comm = $\emptyset$
for each fact node $f_i$ in F
  T = T $\cup$ $\min_{edge=(a,f,t)}$(incoming-edges($f_i$) | edge.t $\neq$ -1)
T = $sort_t$(T)
for each edge=($a_i$,$f_i$,$t_i$)$\in$ T
```

[9] This assumes that there exists an optimal communication sequence in which all agent information needs are satisfied. We can encode scenarios in which this condition does not hold.

```
for each $a_j$ in outgoing-edge($f_i$)=($f_i$->$a_j$)
  if($a_i \neq a_j$) Comm = Comm $\cup$ tell($a_i$, $a_j$, $f_i$, $t_i$)
return Comm
```

This algorithm returns a chronologically sorted set of communicative actions from agent a_i to a_j of fact f_i at time t_i. Intuitively, this algorithm says that the first agent to learn fact $f_i \in F$ should inform all other agents who will need this information (other than themselves). This allows us to specify the *optimal* Policy 1 communication sequence of agents in this scenario. Similar algorithms can be derived for models 2-4.

MODELING AGENT DOMAINS

Now that we have a formal way to specify scenarios, we next ask, “What types of scenarios can we represent using this formalism?” As a concrete example, here we will specify a scenario of a multi-agent team whose goal it is to plan a party.

A rugby team decides to throw a party and has put three players A1, A2, and A3 in charge of making the party arrangements and collecting money from the other members of the club. In addition A4 is the treasurer of the club, the treasurer must make all purchases for the club – although he has no time to actually plan the party. A1 is in charge of figuring out the details of the food, disc jockey and location of the party. A2 is responsible for getting a permit from the city to hold the party, and determining where to purchase the drinks for the party. A3 is a social chair of the club, so he is responsible for contacting the members of the team, collecting money from members, getting a count of the number of people planning to attend the party, the best day to hold the party and preferred music to play at the party.

Notice that A4 is completely dependant on information from the other three agents to complete his task (purchase items for the party). The other agents can complete their tasks on their own, but have opportunities to help their teammates by communicating information they come across in the course of their tasks. For instance, A3 happens to believe exactly how much it will cost to hire the disc jockey since his roommate hired one at his last party, he should tell A1, since he believes that it is an information need of A1. When A1 talks to the dj at time 4, the dj tells him that techno will be the best choice of music. A1 believes that A3 is responsible to gather this information – so he should communicate this, and so on. The facts and info needs are as follows:

A = {a_1, a_2, a_3, a_4}

F = {f_1 = best_caterer(buppys),

f_2 = best_day_for_party(Saturday),

f_3 = amount_collected($500),

f_4 = num_people(50),

f_5 = drink_merchant(Kroger),

f_6 = place_to_file_permit(municipal-building),

f_7 = amount_of_permit($20),

f_8 = best_dj(mark),

f_9 = cost_of_dj($50),

f_{10} = music_pref(techno),

f_{11} = party_location(marks-house),

$f1_2$ = cost_of_drinks($300),

f_{13} = cost_of_food($100)}

info-needs$_1$ = {f_1, $f_{8,}$ f_9, f_{11}, f_{13}}

info-needs $_2$ = {f_5, $f_{6,}$ f_7, f_{12}}

info-needs $_3$ = {f_2, $f_{3,}$ f_4, f_{10}}

info-needs $_4$ = {f_3, $f_{5,}$ f_7, f_8, f_9, f_{12}, f_{13}}

inferences = {} // for this example, we ignore inferences.

Numbers in this list are the time steps at which agents will learn facts. The number -1 means that the agent will never learn the information.

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}
a_1	5	-1	-1	-1	-1	4	-1	3	4	4	9	-1	5
a_2	-1	-1	10	-1	7	5	8	-1	-1	-1	-1	10	-1
a_3	-1	5	6	4	5	-1	-1	1	-1	7	-1	3	-1
a_4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Below is a graph of this scenario, edges emanating from facts and terminating at an agent are dashed and represent information needs. Edges emanating from agents and terminating at a fact are solid and represent information-events; these event-edges are labeled with the time in which they occur.

Using the algorithm listed in appendix B, we can show that the optimal communication sequence is:

t	From	To	Message
-	----	--	-------
1	a_3	a_1	f_8
1	a_3	a_4	f_8
3	a_3	a_2	f_{12}
3	a_3	a_4	f_{12}
4	a_1	a_3	f_{10}
4	a_1	a_2	f_6
4	a_1	a_4	f_9
5	a_3	a_2	f_5
5	a_3	a_4	f_5
5	a_1	a_4	f_{13}
6	a_3	a_4	f_3
8	a_2	a_4	f_7

Figure 7 shows the information needs and event graph for the rugby party domain.

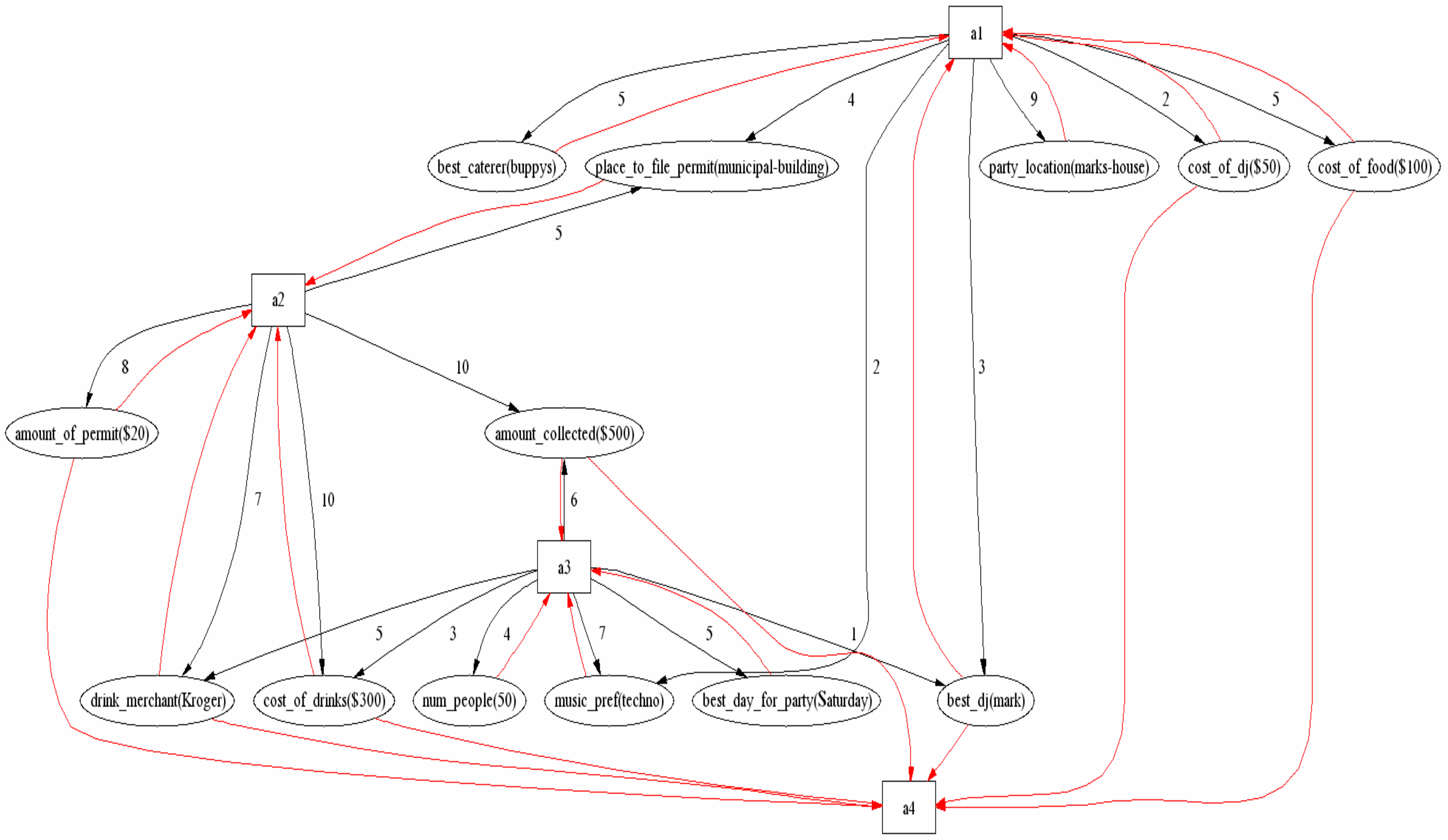


Figure 7 - Information Need/Event Graph for Party Domain

This communication sequence can be plotted graphically by connecting agents with edges labeled with the time and fact of the communication sent (see figure 8).

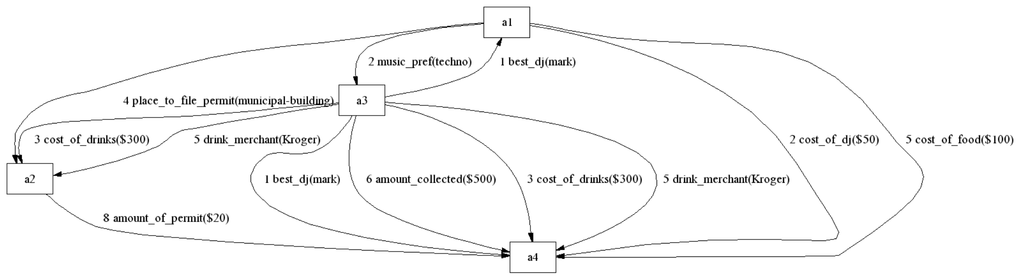


Figure 8 - Information Exchange Graph for Party Domain

EVALUATION FRAMEWORK

We have described a formal framework for specifying information exchange scenarios and how to calculate the optimal communications between agents in these scenarios, as well as our algorithm for generating the communications in a multi-agent setting. In this section, we discuss how we will evaluate the agents' performance using our definition of the optimal communication sequence. We start by looking at the set of optimal communication actions *OPT* = {<*a*,*f*,*t*> | agent *a* is optimally informed of fact *f* at time *t*} calculated from the description of the domain and the actual communications delivered by each agent ACT = {<*a*,*f*,*t*> | agent a is actually informed of fact f at time t} operating in the scenario. We can specify an error function *err* that assigns a numerical value to the difference between actual and optimal communication and use this to score our agents' information-exchange actions. We can say that:

$$err = \sum_{af_i \in ACT \cap OPT} ACT_{af_i} - OPT_{af_i} + \sum_{af \mid af \in ACT, af \notin OPT} \beta + \sum_{af \mid af \notin ACT, af \in OPT} \gamma$$

where ACT_{af} is the time at which agent *a* is actually informed of fact *f*, OPT_{af} is the time at which agent *a* is optimally informed of f_i. β is a numeric error value for unnecessary communication and γ is a numeric error value for missed communication opportunity. These two penalties can be set in a domain-dependent way; in some domains it is more costly not to communicate information needs and in others it is more costly to communicate unnecessarily. Note that if ACT ≡ OPT then the agent has optimally informed his teammates and $err = 0$.

Note that we do not expect our agents to achieve the optimal communication for proactive information exchange, since it is not guaranteed that they will maintain a perfect model of their teammate's beliefs. Nonetheless, our hypothesis is that they will be able to come reasonably close to the optimal communication in certain situations. In this section, we have discussed how we will quantitatively evaluate the communication events generated from the PIEX algorithm using the criteria for optimal communications described in this section. In the example section, we will create a military aircraft mission scenario, implement agents running PIEX and evaluate the communication that they perform.

LIMITATIONS AND EXTENSIONS

We do not claim that this framework can capture or describe all of the nuances of proactive information exchange scenarios. Having said that, we do think that this formalism captures the fundamental aspects of the proactive information exchange task, and we use this for our experiments. Below we list limitations of this framework, which can be viewed as possible extensions for future work.

a. Ability to communicate This framework does not consider whether agents *can* communicate with each other. In many domains agents do not have unlimited communication ability. For instance, agents may have a sensory range, which includes sensing communications, and agents might not be able to send/receive a message if they are out of range. This model does not take this into account; we assume that agents can send and receive messages to any member of the team at any time.

b. Information need contingency For instance, some information needs may be dependent on the knowledge of other facts, such as (*if x then info-need(y) else info-need(z)*). That is, we do not handle the case in which a_i needs to believe either *y* or *z* depending on the value of *x*. The agent therefore, will not need either *y* or *z* unless he first believes the value of *x*. For instance, in the party-planning example above, the information need that *best_dj(mark)* may be dependent on the agent believing that *preferred_music(techno)*. This culd be accounted for by using dependency trees which is left for future work.

c. Time dependent needs Some information needs might have temporal constraints, such as “I need this information before x” or “This information will be relevant to *a* after time *t*” or “During time (t_s,t_f) this information is relevant to agent *a*. In this framework, we only consider static information needs.

d. Dynamic truth values We assume that the facts in the world have a fixed truth value, that is fact *b* will not toggle between true,false,true, etc.

e. Weighted error (cost function) In evaluating our team information exchange performance, some facts may be more critical than others and communication errors with respect to these pieces of information can cost the team more than errors on other information-facts. The error function *err* defined above can be modified to take this cost of each fact into account, although it currently does not do so.

f. Team structure Sometimes, teams are structured in such a way that communication should not travel point-to-point from information-possessors to information-needers. Instead the structure of the team could dictate that certain agents only communicate with other agent, which may then possibly act as an intermediary, forwarding this information according to the team protocol. This description of scenarios assumes that point-to-point communication is acceptable.

While we admit that these limitations restrict our ability to model all possible communication scenarios for proactive information exchange, we feel that our approach captures the most salient features of the problem and that we can construct interesting scenarios for proactive-information-exchange algorithm evaluation.

EXAMPLE

In this section, we will implement a scenario for a team of military aircraft agents running CAST-PM/PIEX to demonstrate qualitatively that PIEX generates communication reasonably close to optimal. This scenario involves five military aircraft on a bombing mission. Two scout aircraft (s1 and s2) and fighter jet (p1) fly ahead of the two bombers (b1 and b2). The scouts verify the location of the enemy targets, status of air-defense and radar systems, and the number of enemy forces at the target. The support fighter jet provides assistance to and helps protect the reconnaissance aircraft. When the scouting is complete, the bombers are informed and they strike their targets. The five aircraft then return to the base together. The MALLET plan for this team is listed in Appendix C and the file used to specify the information scenario is listed in Appendix D. The CAST-PM agents analyze this MALLET plan to detect the information needs of their teammates. For instance, the bomber agents have the goal of striking target ‘target1’ and ‘target2’, these are plans with preconditions of knowing the locations of each target. Since the team believes that agent b1 is assigned to strike target ‘target1’ – the location of target1 is an information need of b1. Sub-plans and sub-task information needs are expanded and the information is categorized as sub-information needs of the parent tasks, this allows agents to prune sub-information needs when they believe that the parent task is complete.

In this example scenario (appendix D), while scout one (s1) is responsible for scouting positions of enemy radar, scout two (s2) comes to believe of the location of radar2 first (at time 13) and therefore has an opportunity to let s1 believe before he would come to

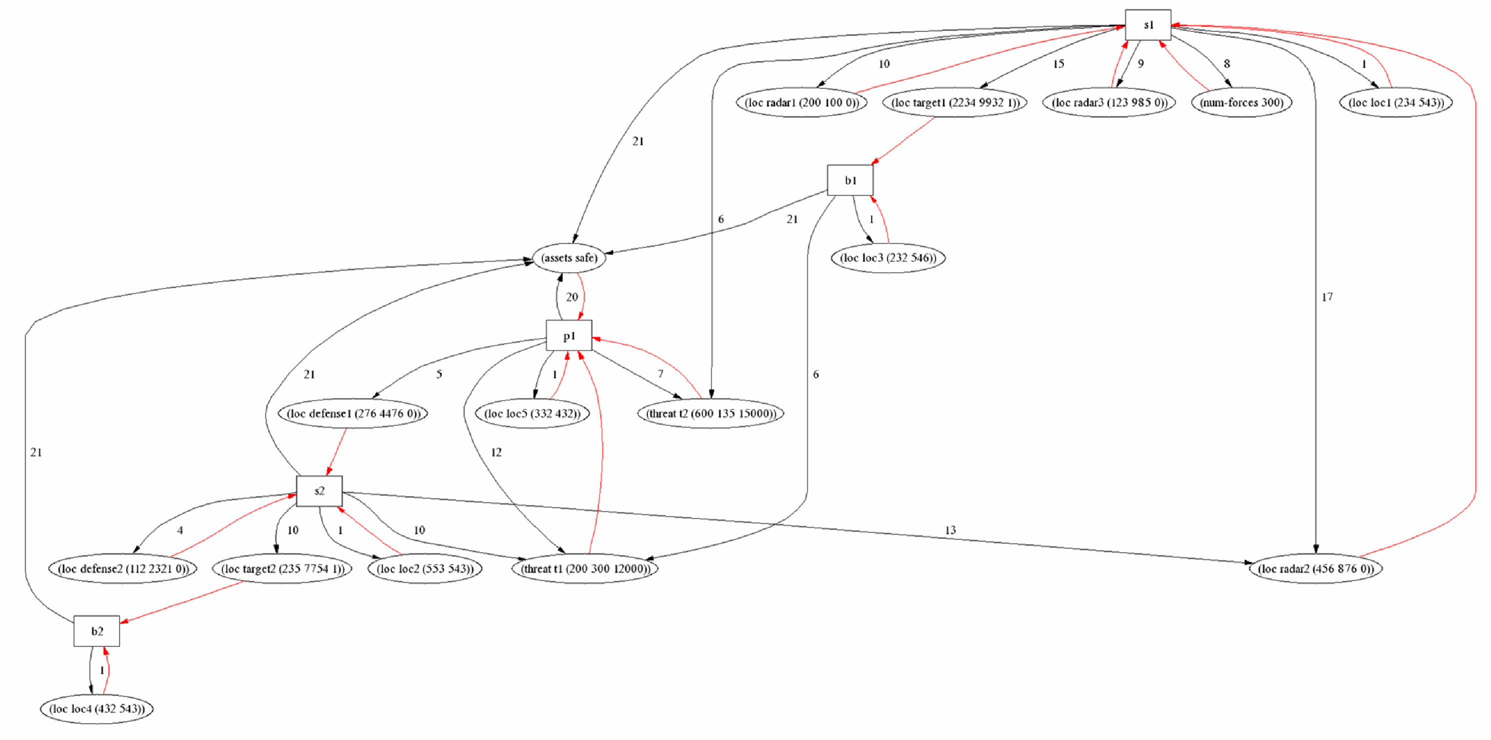


Figure 9 - Graphical Representation of Military Aircraft Scenario

believe the fact himself (time 17). Other such opportunities are available throughout the course of this plan.

Figure 9 shows the graph representing this scenario and Figure 10 shows the optimal communication graph. Table 1 lists the optimal communication sequence [10]for the agents. In our validation, we will use the scenario listed in Appendix D to run a simulation for a team of agents executing the MALLET plan listed in Appendix F.

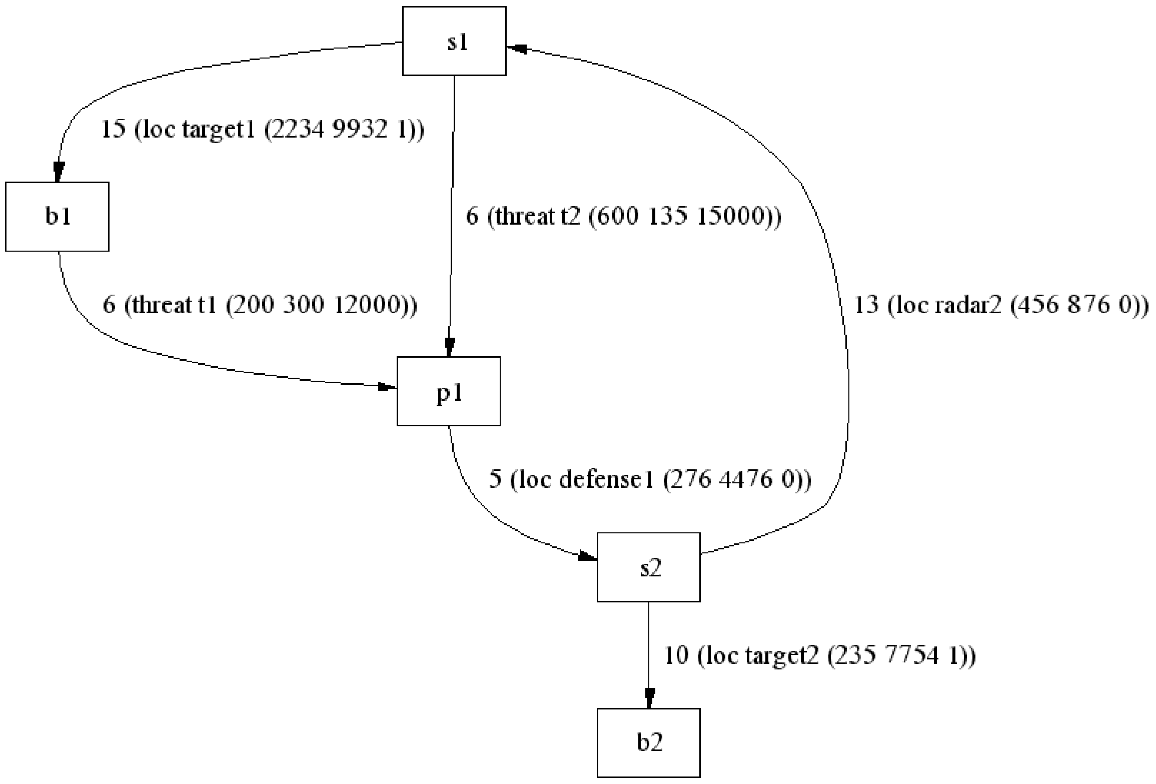

s1
b1
p1
s2
b2
15 (loc target1 (2234 9932 1))
6 (threat t2 (600 135 15000))
6 (threat t1 (200 300 12000))
13 (loc radar2 (456 876 0))
5 (loc defense1 (276 4476 0))
10 (loc target2 (235 7754 1))

Figure 10 - Graphical Representation of Optimal Communication for Military Air Combat Scenario

[10] Using the minimal communication optimality condition described above.

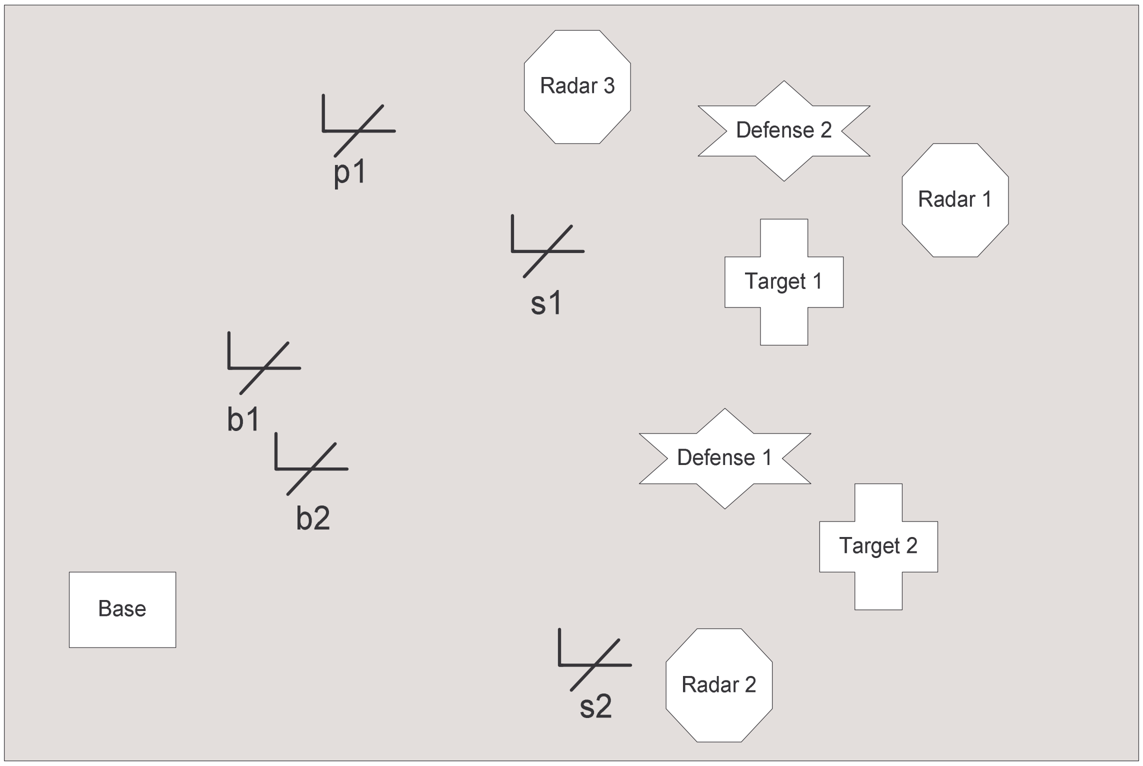


Figure 11 - Map of Air Combat Simulation

Figure 11 shows a view of this air combat scenario. The scenario simulation server outputs the optimal communication sequence for the agents running the scenario, delivers the information events at the specified times and keeps track of the actual communication events that take place between teammates. We will now examine the communication behavior of the teammates running the plans listed in appendix D to see how closely the communication generated by PIEX comes to the optimal communication. The optimal communication sequence is listed in table 1.

Table 1 - Optimal Communication for Military Aircraft Simulation

t	from	to	fact
5	p1	s2	(loc defense1 (276 4476 0))
6	b1	p1	(threat t1 (200 300 12000))
6	s1	p1	(threat t2 (600 135 15000))
10	s2	b2	(loc target2 (235 7754 1))
13	s2	s1	(loc radar2 (456 876 0))
15	s1	b1	(loc target1 (2234 9932 1))

The simulation engine delivers the information events to agents at the prescribed times (the information events specified in the scenario description file in Appendix D). Agents run in their own processes and have no access to each others' knowledge bases or plan states, other than through the messages that agents send to each other through the simulation server. The actual communication events that the agents perform are listed in table 2[11]:

Table 2 - Actual Communication for Military Aircraft Simulation

t	from	to	fact
6	p1	s2	(loc defense1 (276 4476 0))
7	s1	p1	(threat t2 (600 135 15000))
7	b1	p1	(threat t1 (200 300 12000))
11	s2	b2	(loc target2 (235 7754 1))
14	s2	s1	(loc radar2 (456 876 0))
16	s1	b1	(loc target1 (2234 9932 1))

Comparing table 1 (optimal communication) and table 2 (actual communication) we see that none of the optimal communication points are missed and no superfluous communication is generated by this team. The actual and optimal communication times

[11] Note that we only list the information exchange events, in order to compare the actual to the optimal communication sequences for information exchange. The PIEX algorithm also automatically generates communication to inform teammates when the agent has completed a task or plan.

differ by only one time step, which is due to the way the simulator and agents interact – when agents are delivered senses, they must process the senses and re-evaluate their plans (Process Manager tree) according to any new information received, then afterwards the agents decide on an action and perform it, during the next time step.

We also notice that the BOA belief reasoning module is useful in filtering out unnecessary communications. For instance, agent B1 will come to believe fact (threat t1 (200 300 12000)) at time 6, S2 believes this at time 10, and P1 believes this at time 12. This fact is an information need of agent P1. The optimal communication sequence specifies that P1 only needs to be informed once and as soon as possible (which has B1 telling P1 at time 6). In order for agent S2 to avoid sending redundant information at time 10, S2 needs to reason that since B1 has already told P1 the information need, that P1 now believes this. The following BOA action justification encodes this knowledge:

```
; says that when we know that agent ?ag tells agent ?to
; message ?msg then we know that ?to knows ?msg
(action (do ?ag (say ?to ?msg))
        (effects (bel ?to ?msg)))[12]
```

And one of the conditions of the active-inform plan generating a communication act is the rule:

[12] Of course, this seems like a trivial rule that would be easy enough to hard code into an agent architecture. The real utility of BOA is when it is used in complex environments where several justifications can attempt to alter the value of the same fact, and the reasoning system has to resolve these conflicts. Since our purpose is to demonstrate PIEX and not BOA, it is sufficient to show that the BOA system is integrated with CAST enough to maintain models of teammate's beliefs.

```
(defrule (needs-info ?agent ?type ?id ?info-need) (do-id ?id ?parent
?agent ?proc) (info-need ?type ?id ?info-need) (not (bel ?agent ?info-
need)) (not (bel ?agent (whether ?info-need))))
```

this says that an agent only *needs* information if it is both listed as an information-need and also that the information needing agent does not already believe the given fact. The combination of the BOA justification for how to update an agents model of his teammates knowledge coupled with this rule about who needs to be told information allows the agent to perform *selective communication*.

In addition to filtering out unnecessary communication, PIEX has improved the efficiency of this team activity. In the communication of the threat1 in the previous example, the fighter jet was able to intercept the threat at time 6 rather than time 12, which helps ensure the safety of the team.

Appendix E shows a screenshot of the visualization of the Process Manager tree for agent B1 during the simulation. Appendix F shows the state of agent p1's knowledge base and process manager tree in time steps 5 and 6. At time 5, p1 is delivered the information (loc defense1 (276 4476 0)), and we can see the change in both the agents knowledge base and process manager tree which commits him to communicating that fact to agent s2, who needs that information.

In this section, we have run actual CAST-PM software agents running the PIEX algorithm in a military aircraft simulation. We have demonstrated that PIEX comes quite close to optimal communication in this environment.

CONCLUSION

The goal of this research was to enable agents to maintain a shared mental model for the purposes of enhancing proactive teamwork behavior, specifically proactive information exchange. Our claim is that by analyzing a shared team plan and with a way to maintain knowledge about the beliefs of agent teammates, agent architectures can automatically transform a shared team plan into individual plans with appropriate communication points for need-based information exchange. We have defined formal conditions for when proactive information takes place as well as the motivation for proactive information exchange. We have described our framework for generating information testing scenarios for validating agent behavior and tested our agents in one such scenario.

We feel that our work on practical methods for proactive information exchange in agent teams is a significant contribution to the area of multi-agent systems; although we have only scratched the surface in this area. Future work might focus on the costs and utility of communication in different domains, in these domains teams may use decision-theoretic methods to decide when and if to deliver information (similar to the decision-theoretic communication routines in STEAM). Another open area involves identifying characteristics of agent domains that would make proactive information exchange more useful or difficult. We hypothesize that in less observable environments, agents will be less able to maintain models of each others beliefs and would deviate farther from optimal communications; although in these environments teams also stand to benefit more from proactive information exchange even more since information is more distributed. The formal framework presented might be used to define certain characteristics of agent domains and test the performance teams under different environmental conditions. One might experiment with other types of communication protocols such as broadcast vs. multicast vs. point-to-point communication.

It would also be interesting to investigate possible applications of proactive information exchange for agent teams in the real world. Recently, the inability of organizations (especially in government) to keep other agents or organizations properly informed of relevant information has been blamed for many blunders. Also, organizations such as the CDC might benefit from the ability to have relevant information from health centers around the country proactively forwarded to them to aid in assessing the possibility of outbreaks of disease or other health hazards. The amount of information that such organizations have to analyze and interpret is growing rapidly and the ability to pre-filter only relevant information could also possibly boost the efficiency of such organizations.

Future work may also look at other applications of belief reasoning systems like BOA. In this work, we have leveraged the ability to reason about beliefs of others for the purposes of need based delivery of information. There could be other multi-agent applications of modeling the beliefs of others; for instance, coordination, negotiation, role assignment, etc. The primary contribution of this work is the enhancement of team efficiency by monitoring information needs and beliefs of other agents to properly maintain a shared mental model.

REFERENCES

[1] N. R. Jennings, "Controlling cooperative problem-solving in industrial multiagent systems using joint intentions," *Artificial Intelligence*, vol. 75, no. 2, pp. 195-240, 1995.

[2] E. H. Durfee and V. R. Lesser, "Using partial global plans to coordinate distributed problem solvers," in *Proc. Tenth International Joint Conference on Artificial Intelligence*, pp. 875-883, 1987.

[3] E. Salas, T. L. Dickinson, S. A. Converse, and S. I. Tannenbaum, "Towards an understanding of team performance and training," in *Teams: Their Training and Performance*, R. W. Swezey and E. Salas, Eds. Norwood, NJ: Ablex Pub. Corp., 1992, pp. 3-29.

[4] J. Searle, *Speech Acts: An Essay in the Philosophy of Language*. Cambridge: Cambridge University Press, 1970.

[5] P. R. Cohen and H. J. Levesque, "Communicative actions for artificial agents," in *Proc. First International Conference on Multi-Agent Systems*, San Francisco, CA, pp. 65-72, 1995.

[6] P. R. Cohen and C. R. Parrault, "Elements of a plan-based theory of speech acts," *Cognitive Science*, vol. 3, pp. 177-212, 1979.

[7] L. Gasser, "Social conceptions of knowledge and action: DAI Foundations and Open Systems Semantics," *Artificial Intelligence*, vol. 47, pp. 107-138, 1991.

[8] N. R. Jennings, "Commitments and conventions: The foundation of coordination in multi-agent systems," *Artificial Intelligence*, vol. 75, 1994.

[9] T. R. Ioerger and J. C. Johnson, "A formal model of responsibilities in agent-based teams," in *Proc. Fifth International Conference on Autonomous Agents*, Montreal, Canada, pp. 157-158, 2001.

[10] H. Isozaki and H. Katsuno, "A semantic characterization of an algorithm for estimating others' beliefs from observation," in *Proc. of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference, vol. 2*, H. Sharobe, Ed. Menlo Park, CA, 1996, pp. 543--549.

[11] M. N. Huhns and D. M. Bridgeland, "Multiagent truth maintenance," *IEEE Transactions on Systems Man and Cybernetics*, vol. 21, no. 6, pp. 1437-1445, 1991.

[12] M. Tambe, "Towards flexible teamwork," *Journal of Artificial Intelligence Research*, vol. 7, no. 1, pp. 83-124, 1997.

[13] M. Tambe, "Agent architectures for flexible, practical teamwork," in *Proc. National Conference on Artificial Intelligence*, pp. 22-28, 1997.

[14] M. Yokoo and K. Hirayama, "Algorithms for distributed constraint satisfaction: A review," *Autonomous Agents and Multi-Agent Systems*, vol. 3, no. 2, pp. 185-207, 2000.

[15] B. Grosz and S. Kraus, "Collaborative plans for complex group actions," *Artificial Intelligence*, vol. 86, no. 2, pp. 269-357, 1996.

[16] J. Yen, J. Yin, T. R. Ioerger, M. S. Miller, D. Xu, and R. A. Volz, "CAST: Collaborative agents for simulating teamwork," in *Proc. 17th International Joint Conference on Artificial Intelligence,* Seattle, WA, pp. 1135-1144, 2001.

[17] P. R. Cohen and H. J. Levesque, "Intention is choice with commitment," *Artificial Intelligence*, vol. 42, pp. 213-261, 1990.

[18] P. R. Cohen and H. J. Levesque, "Teamwork," *Nous*, vol. 25, no. 4, pp. 487-512, 1991.

[19] K. Erol, J. Hendler, and D. S. Nau, "HTN planning: Complexity and expressivity," in *Proc. Twelfth National Conference on Artificial Intelligence*, Seattle, Washington, pp. 1123-1128, 1994.

[20] B. Grosz and S. Kraus, "The evolution of shared plans," in *Foundations and Theories of Rational Agency*, A. Rao and M. Wooldridge, Eds., Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 227-262, 1998.

[21] N. R. Jennings, K. Sycara, and M. Wooldridge, "A roadmap of agent research and development," *Autonomous Agents and Multi-Agent Systems*, vol. 1, pp. 275-306, 1998.

[22] E. Sonenberg, G. Tidhar, E. Werner, D. Kinny, M. Ljungberg, and A. Rao, "Planned team activity," Australian Artificial Intelligence Institute, Melbourne, Australia, Technical Notes 26 1992.

[23] E. Salas, C. Prince, D. Baker, L. Shrestha., "Situation awareness in team performance: Implications for measurement and training," *Human Factors*, vol. 37, no.

1, pp. 123-136, 1995.

[24] J. M. Orasanu, "Shared mental models and crew decision making," Cognitive Sciences Laboratory, Princeton, NJ, CSL Report 46, 1990.

[25] P. Stone and M. Veloso, "Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork," *Artificial Intelligence*, vol. 110, no. 2, pp. 241-273, 1999.

[26] K. E. Biggers and T. R. Ioerger, "Automatic generation of communication and teamwork within multi-agent teams," *Applied Artificial Intelligence*, vol. 15, pp. 875-916, 2001.

[27] S. Kripke, "Semantical analysis of modal logic.," *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, vol. 9, pp. 67-96, 1963.

[28] J. Hintikka, *Knowledge and Belief*. Ithaca, NY: Cornell University Press, 1962.

[29] J. Yen, X. Fan, and R. A. Volz, "On proactive delivery of needed information to teammates," AAMAS Workshop on Teamwork and Coalition Formation, pp. 53-61, 2002.

[30] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, pp. 189-208, 1971.

[31] M. Paolucci, D. Kalp, A. S. Pannu, O. Shehory, and K. Sycara, "A planning component for RETSINA agents," in *Lecture Notes in Artificial Intelligence, Intelligent Agents VI*, Springer, New York, 1999.

APPENDIX A

MALLET SYNTAX

```
/**
 * Mallet 2
 * Created: 08/15/2001
 */

(agent <agentName>)

(object-type <name>)

(role <roleName> (<agentName>+))

(capable <agentName> <operName>)

<pred> ::= (<predName> <argument>*)
<cond> ::= <pred> | (not <pred>)
<var>  ::= ?<string>

(i-oper <operName> (<var>*)
  [(pre-cond <cond>+)]
  [(effects <cond>+)])

(t-oper <operName> (<var>*)
  [(pre-cond <cond>+)]
  [(effects <cond>+)]
  [(share-type AND|OR|XOR)]) // default share-type = OR

(plan <planName> (<var>*)
  [(pre-cond <cond>+)]
  [(effects <cond>+)]
  (term-cond [FAILURE|SUCCESS] <cond>+) // default is FAILURE
  (process (<proc>+)) // procedures in process
                      // are implicitly sequential
<team-plan> ::=
  (team-plan <name> (<var>+)
    [(pre-cond <cond>+)]
    [(term-cond [SUCCESS|FAILURE] <cond>+)]
    [(effects <cond>+)]
    (process <tproc>)
  )

<proc> ::=
  (seq <proc>+) |
  (par <proc>+) |
  (if (cond <cond>+) <proc> [<proc>]) |
  (while (cond <cond>+) <proc>) |
  (foreach <cond> <proc>) |
  (forall <cond> <proc>) |
  (choice <proc>+) |
  (do <agentName>|<roleVar> <call>) |
  (select-role <roleVar> <roleName> [(constraint <cond>+)] <proc>)

<call> ::= (<planName>|<operName> <args>*)

<tproc> ::=
  (seq <tproc>+) |
  (par tproc>+) |
  (do <agent> <proc>) // agent specified by <agent> is to do <proc>
```

APPENDIX B

GRAPH ALGORITHM FOR OPTIMAL COMMUNICATION

```
Set $T = \varnothing$
Set $Comm = \varnothing$
for each fact node $f_i$ in F
  $T = T \cup \min_{edge=(a,f,t)}$(incoming-edges($f_i$) | edge.t $\neq$ -1)
$T = sort_t(T)$
for each edge=$(a_i,f_i,t_i) \in T$
  for each $a_j$ in outgoing-edge($f_i$)=($f_i$->$a_j$)
    if($a_i \neq a_j$) Comm = Comm $\cup$ tell($a_i$, $a_j$, $f_i$, $t_i$)
return Comm
```

APPENDIX C

MALLET PLAN FOR MILITARY AIRCRAFT TEAM

Note that the files for all five agents are identical except for the first line which asserts the ‘self’ fact, letting the agent know which one they are.

```
; boa belief reasoning
(declare (self b1))
(declare (agents s1 s2 p1 b1 b2))

; initialize flight location
(init (bel s1 (loc loc1 (234 543))))
(init (bel s2 (loc loc2 (553 543))))
(init (bel b1 (loc loc3 (232 546))))
(init (bel b2 (loc loc4 (432 543))))
(init (bel p1 (loc loc5 (332 432))))

; information need persist
(persist (bel ?ag (assets safe)))
(persist (bel ?ag (threat ?threat (?lat ?long ?alt))))
(persist (bel ?ag (loc ?obj ?loc)))
(persist (bel ?ag (num-forces ?num)))

; says that when we know that agent ?ag tells agent ?to
; message ?msg then we know that ?to knows ?msg
(action (do ?ag (say ?to ?msg))
        (effects (bel ?to ?msg)))
; when you hear someone broadcast that they are done
; you know the task is done
(action (do ?ag (say (complete ?id)))
        (effects (complete ?id)))

; done persists
(persist (bel ?ag (complete ?any)))

; info needs persist
(persist (bel ?ag (do-id ?id ?parent ?who ?what)))
(persist (bel ?ag (info-need ?type ?do-id ?fact)))
(persist (bel ?ag (done ?any)))

; individual operators (atomic actions in the world)
(i-oper fly-to (?loc)
  (pre-cond (loc ?loc (?lat ?long))))
(i-oper deploy-payload (?lat ?long ?alt))
(i-oper say (?to ?msg))
(i-oper broadcast (?msg))

; main team plan
```

```
(team-plan bombing-run ()
  (process
    (par
      (do p1 (protect-assets))
      (seq
        (par
            (do s1 (fly-to loc1))
            (do s2 (fly-to loc2))
            (do b1 (fly-to loc3))
            (do b2 (fly-to loc4))
            (do p1 (fly-to loc5))
          )
        (par
          (do s1 (scout-for-radar))
          (do s1 (scout-for-num-forces))
          (do s2 (scout-for-air-defense))
          (do b1 (strike-target target1))
          (do b2 (strike-target target2))
        )
          (par
            (do s1 (fly-to base))
            (do s2 (fly-to base))
            (do p1 (fly-to base))
            (do b1 (fly-to base))
            (do b2 (fly-to base))
          )
      )
    )
  )
)

(plan protect-assets ()
  (term-cond SUCCESS (assets safe))
  (process
    (while (cond (= 1 1))
      (if (cond (threat ?threat ?loc))
        (print (INTERCEPTING ?threat AT ?loc))
        (NOP)
      )
    )
  )
)

(plan scout-for-radar ()
  (term-cond SUCCESS (loc radar1 ?loc1)(loc radar2 ?loc2)(loc radar3
?loc3))
  (process
    (while (cond (= 1 1))
      (NOP) ; scout around
    )
  )
)

(plan scout-for-num-forces ()
  (term-cond SUCCESS (num-forces ?num))
  (process
    (while (cond (= 1 1))
      (NOP) ; scout around
    )
  )
)

(plan scout-for-air-defense ()
  (term-cond SUCCESS (loc defense1 ?loc1)(loc defense2 ?loc2))
  (process
    (while (cond (= 1 1))
      (NOP) ; scout around
    )
  )
)

(plan strike-target (?target)
  (term-cond SUCCESS (done ?agent (deploy-payload deploy-payload ?a ?b
?c)))
  (process
    (while (cond (= 1 1))
      (if (cond (loc ?target (?lat ?long ?alt)))
        (deploy-payload ?lat ?long ?alt)
        (NOP)
      )
    )
  )
)

(plan active-inform (?id)
  (pre-cond (do-id ?id ?parent ?agent ?proc))
  (term-cond SUCCESS (complete ?proc))
  (effects (not (do-id ?id ?parent ?agent ?proc))
           (not (info-need ?type ?id ?pred)))
  (process
    (par
      ; active-inform all info needs of this plan
      (forall ((info-need ?type ?id ?info-need))
        (seq
          (while (cond (not ?info-need)(needs-info ?agent ?type ?id
?info-need)) (NOP))
          (if (cond ?info-need (needs-info ?agent ?type ?id ?info-
need)) (say ?agent ?info-need))
          ;(if (cond (success-cond ?type)) (assert (finshed ?id))) ;
see boa rule success-cond below
        )
      )
      ; create another active inform plan for all children of this plan
      (forall ((do-id ?child-id ?id ?child-agent ?child-spec))
        (active-inform ?child-id)
      )
    )
  )
```

```
)

(defrule (needs-info ?agent ?type ?id ?info-need) (do-id ?id ?parent
?agent ?proc) (info-need ?type ?id ?info-need) (not (bel ?agent ?info-
need)) (not (bel ?agent (whether ?info-need))))

; 1) if you hear the agent tell you he is done
(defrule (done ?id)(hear (done ?proc))(do-id ?id ?parent ?agent ?proc))
```

APPENDIX D

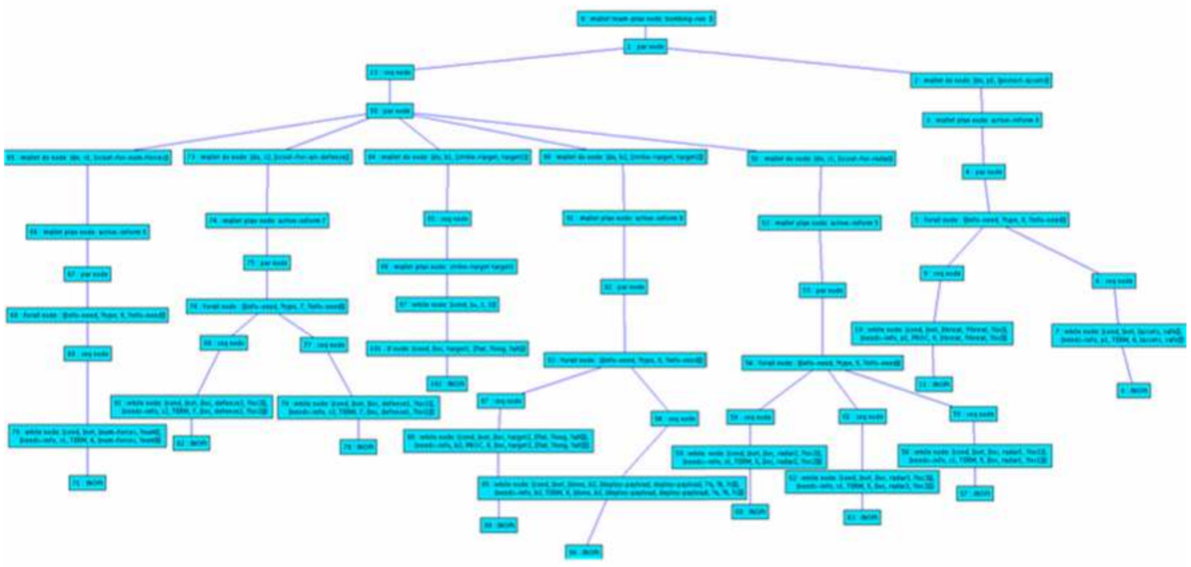
INFORMATION EXCHANGE FILE FOR MILITARY AIRCRAFT DOMAIN

```
A=s1,s2,b1,b2,p1
F=(assets safe),(threat t1 (200 300 12000)),(threat t2 (600 135
15000)),(loc radar1 (200 100 0)),(loc radar2 (456 876 0)),(loc radar3
(123 985 0)),(num-forces 300),(loc defense1 (276 4476 0)),(loc defense2
(112 2321 0)),(loc target1 (2234 9932 1)),(loc target2 (235 7754
1)),(loc loc1 (234 543)),(loc loc2 (553 543)),(loc loc3 (232 546)),(loc
loc4 (432 543)),(loc loc5 (332 432))
info-need-1=4,5,6,7,12
info-need-2=8,9,13
info-need-3=10,14
info-need-4=11,15
info-need-5=1,2,3,16
event-1=21,-1,6,10,17,9,8,-1,-1,15,-1,1,-1,-1,-1,-1
event-2=21,10,-1,-1,13,-1,-1,-1,4,-1,10,-1,1,-1,-1,-1
event-3=21,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,-1,-1
event-4=21,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1,-1
event-5=20,12,7,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,1
```

APPENDIX E

PROCESS MANAGER TREE FOR AIRCRAFT AGENT

This is a picture of the process manager tree for the aircraft agent, the size of the graphic prohibits the reading of text inside each node.



APPENDIX F

KNOWLEDGE BASE AND PROCESS MANAGER STATES IN THE AIRCRAFT DOMAIN

```
Agent p1 time 5

******** KB **********
2:(bel b1 (loc loc3 (232 546)))
2:(bel b2 (loc loc4 (432 543)))
2:(bel p1 (complete (fly-to loc1)))
2:(bel p1 (complete (fly-to loc4)))
2:(bel p1 (complete (fly-to loc2)))
2:(bel p1 (complete (fly-to loc5)))

2:(bel p1 (complete (fly-to loc3)))
2:(bel p1 (loc loc5 (332 432)))
7:(bel p1 (agent s1))
7:(bel p1 (agent s2))
7:(bel p1 (agent p1))
7:(bel p1 (agent b1))
7:(bel p1 (agent b2))
7:(bel p1 (self p1))
2:(bel p1 (unknown (info-need PRE 0 (loc loc1 (?lat ?long)))))
2:(bel p1 (unknown (info-need PRE 1 (loc loc2 (?lat ?long)))))
2:(bel p1 (unknown (info-need PRE 2 (loc loc3 (?lat ?long)))))
2:(bel p1 (unknown (info-need PRE 3 (loc loc4 (?lat ?long)))))
2:(bel p1 (info-need TERM 4 (loc radar1 ?loc1)))
2:(bel p1 (info-need TERM 4 (loc radar2 ?loc2)))
2:(bel p1 (info-need TERM 4 (loc radar3 ?loc3)))
2:(bel p1 (info-need TERM 5 (num-forces ?num)))
2:(bel p1 (info-need TERM 6 (loc defense1 ?loc1)))
2:(bel p1 (info-need TERM 6 (loc defense2 ?loc2)))
2:(bel p1 (info-need TERM 7 (done ?agent (deploy-payload deploy-payload
?a ?b ?c))))
2:(bel p1 (info-need PROC 7 (loc target1 (?lat ?long ?alt))))
2:(bel p1 (info-need TERM 8 (done ?agent (deploy-payload deploy-payload
?a ?b ?c))))
2:(bel p1 (info-need PROC 8 (loc target2 (?lat ?long ?alt))))
2:(bel p1 (unknown (do-id 0 null s1 (fly-to loc1))))
2:(bel p1 (unknown (do-id 1 null s2 (fly-to loc2))))
2:(bel p1 (unknown (do-id 2 null b1 (fly-to loc3))))
2:(bel p1 (unknown (do-id 3 null b2 (fly-to loc4))))

2:(bel p1 (do-id 4 null s1 (scout-for-radar)))
2:(bel p1 (do-id 5 null s1 (scout-for-num-forces)))
2:(bel p1 (do-id 6 null s2 (scout-for-air-defense)))
2:(bel p1 (do-id 7 null b1 (strike-target target1)))
2:(bel p1 (do-id 8 null b2 (strike-target target2)))
3:(bel p1 (needs-info s1 TERM 4 (loc radar1 ?loc1)))
3:(bel p1 (needs-info s1 TERM 4 (loc radar2 ?loc2)))
3:(bel p1 (needs-info s1 TERM 4 (loc radar3 ?loc3)))
```

```
3:(bel p1 (needs-info s1 TERM 5 (num-forces ?num)))
3:(bel p1 (needs-info s2 TERM 6 (loc defense1 ?loc1)))
3:(bel p1 (needs-info s2 TERM 6 (loc defense2 ?loc2)))
3:(bel p1 (needs-info b1 TERM 7 (done ?agent (deploy-payload deploy-
payload ?a ?b ?c))))
3:(bel p1 (needs-info b1 PROC 7 (loc target1 (?lat ?long ?alt))))
3:(bel p1 (needs-info b2 TERM 8 (done ?agent (deploy-payload deploy-
payload ?a ?b ?c))))
3:(bel p1 (needs-info b2 PROC 8 (loc target2 (?lat ?long ?alt))))
2:(bel s1 (loc loc1 (234 543)))
2:(bel s2 (loc loc2 (553 543)))


******** Process Manager **********
0[0]: MALLET team-plan node: bombing-run []
  1[0]: par node
    2[0]: MALLET do node: [do, p1, [protect-assets]]
      3[0]: seq node
        4[0]: MALLET plan node: protect-assets
          5[0]: while node: [cond, [=, 1, 1]]
            126[?]: if node: [cond, [threat, ?threat, ?loc]]
    8[0]: seq node
      57[0]: par node
        58[0]: MALLET do node: [do, s1, [scout-for-radar]]
          59[0]: MALLET plan node: active-inform 4
            60[0]: par node
              61[0]: forall node : [[info-need, ?type, 4, ?info-need]]
                62[0]: seq node
                  63[0]: while node: [cond, [not, [loc, radar1,
?loc1]], [needs-info, s1, TERM, 4, [loc, radar1, ?loc1]]]
                    127[?]: (NOP)
                65[0]: seq node
                  66[0]: while node: [cond, [not, [loc, radar2,
?loc2]], [needs-info, s1, TERM, 4, [loc, radar2, ?loc2]]]
                    128[?]: (NOP)
                68[0]: seq node
                  69[0]: while node: [cond, [not, [loc, radar3,
?loc3]], [needs-info, s1, TERM, 4, [loc, radar3, ?loc3]]]
                    129[?]: (NOP)
        72[0]: MALLET do node: [do, s1, [scout-for-num-forces]]
          73[0]: MALLET plan node: active-inform 5
            74[0]: par node
              75[0]: forall node : [[info-need, ?type, 5, ?info-need]]
                76[0]: seq node
                  77[0]: while node: [cond, [not, [num-forces, ?num]],
[needs-info, s1, TERM, 5, [num-forces, ?num]]]
                    130[?]: (NOP)
        80[0]: MALLET do node: [do, s2, [scout-for-air-defense]]
          81[0]: MALLET plan node: active-inform 6
            82[0]: par node
              83[0]: forall node : [[info-need, ?type, 6, ?info-need]]
                84[0]: seq node
                  85[0]: while node: [cond, [not, [loc, defense1,
?loc1]], [needs-info, s2, TERM, 6, [loc, defense1, ?loc1]]]
```

```
                    131[?]: (NOP)
                87[0]: seq node
                  88[0]: while node: [cond, [not, [loc, defense2,
?loc2]], [needs-info, s2, TERM, 6, [loc, defense2, ?loc2]]]
                    132[?]: (NOP)
        91[0]: MALLET do node: [do, b1, [strike-target, target1]]
          92[0]: MALLET plan node: active-inform 7
            93[0]: par node
              94[0]: forall node : [[info-need, ?type, 7, ?info-need]]
                95[0]: seq node
                  96[0]: while node: [cond, [not, [done, b1, [deploy-
payload, deploy-payload, ?a, ?b, ?c]]], [needs-info, b1, TERM, 7,
[done, b1, [deploy-payload, deploy-payload, ?a, ?b, ?c]]]]
                    133[?]: (NOP)
                98[0]: seq node
                  99[0]: while node: [cond, [not, [loc, target1, [?lat,
?long, ?alt]]], [needs-info, b1, PROC, 7, [loc, target1, [?lat, ?long,
?alt]]]]
                    134[?]: (NOP)
        102[0]: MALLET do node: [do, b2, [strike-target, target2]]
          103[0]: MALLET plan node: active-inform 8
            104[0]: par node
              105[0]: forall node : [[info-need, ?type, 8, ?info-need]]
                106[0]: seq node
                  107[0]: while node: [cond, [not, [done, b2, [deploy-
payload, deploy-payload, ?a, ?b, ?c]]], [needs-info, b2, TERM, 8,
[done, b2, [deploy-payload, deploy-payload, ?a, ?b, ?c]]]]
                    135[?]: (NOP)
                109[0]: seq node
                  110[0]: while node: [cond, [not, [loc, target2,
[?lat, ?long, ?alt]]], [needs-info, b2, PROC, 8, [loc, target2, [?lat,
?long, ?alt]]]]
                    136[?]: (NOP)

Agent p1 time 6

******** KB **********
2:(bel b1 (loc loc3 (232 546)))
2:(bel b2 (loc loc4 (432 543)))
2:(bel p1 (complete (fly-to loc1)))
2:(bel p1 (complete (fly-to loc4)))
2:(bel p1 (complete (fly-to loc2)))
2:(bel p1 (complete (fly-to loc5)))

2:(bel p1 (complete (fly-to loc3)))
2:(bel p1 (loc loc5 (332 432)))
2:(bel p1 (loc defense1 (276 4476 0)))
7:(bel p1 (agent s1))
7:(bel p1 (agent s2))
7:(bel p1 (agent p1))
7:(bel p1 (agent b1))
7:(bel p1 (agent b2))
7:(bel p1 (self p1))
2:(bel p1 (unknown (info-need PRE 0 (loc loc1 (?lat ?long)))))
```

```
2:(bel p1 (unknown (info-need PRE 1 (loc loc2 (?lat ?long)))))
2:(bel p1 (unknown (info-need PRE 2 (loc loc3 (?lat ?long)))))
2:(bel p1 (unknown (info-need PRE 3 (loc loc4 (?lat ?long)))))
2:(bel p1 (info-need TERM 4 (loc radar1 ?loc1)))
2:(bel p1 (info-need TERM 4 (loc radar2 ?loc2)))
2:(bel p1 (info-need TERM 4 (loc radar3 ?loc3)))
2:(bel p1 (info-need TERM 5 (num-forces ?num)))
2:(bel p1 (info-need TERM 6 (loc defense1 ?loc1)))
2:(bel p1 (info-need TERM 6 (loc defense2 ?loc2)))
2:(bel p1 (info-need TERM 7 (done ?agent (deploy-payload deploy-payload
?a ?b ?c))))
2:(bel p1 (info-need PROC 7 (loc target1 (?lat ?long ?alt))))
2:(bel p1 (info-need TERM 8 (done ?agent (deploy-payload deploy-payload
?a ?b ?c))))
2:(bel p1 (info-need PROC 8 (loc target2 (?lat ?long ?alt))))
2:(bel p1 (unknown (do-id 0 null s1 (fly-to loc1))))
2:(bel p1 (unknown (do-id 1 null s2 (fly-to loc2))))
2:(bel p1 (unknown (do-id 2 null b1 (fly-to loc3))))
2:(bel p1 (unknown (do-id 3 null b2 (fly-to loc4))))
2:(bel p1 (do-id 4 null s1 (scout-for-radar)))
2:(bel p1 (do-id 5 null s1 (scout-for-num-forces)))
2:(bel p1 (do-id 6 null s2 (scout-for-air-defense)))
2:(bel p1 (do-id 7 null b1 (strike-target target1)))
2:(bel p1 (do-id 8 null b2 (strike-target target2)))
3:(bel p1 (needs-info s1 TERM 4 (loc radar1 ?loc1)))
3:(bel p1 (needs-info s1 TERM 4 (loc radar2 ?loc2)))
3:(bel p1 (needs-info s1 TERM 4 (loc radar3 ?loc3)))
3:(bel p1 (needs-info s1 TERM 5 (num-forces ?num)))
3:(bel p1 (needs-info s2 TERM 6 (loc defense1 ?loc1)))
3:(bel p1 (needs-info s2 TERM 6 (loc defense2 ?loc2)))
3:(bel p1 (needs-info b1 TERM 7 (done ?agent (deploy-payload deploy-
payload ?a ?b ?c))))
3:(bel p1 (needs-info b1 PROC 7 (loc target1 (?lat ?long ?alt))))
3:(bel p1 (needs-info b2 TERM 8 (done ?agent (deploy-payload deploy-
payload ?a ?b ?c))))
3:(bel p1 (needs-info b2 PROC 8 (loc target2 (?lat ?long ?alt))))
2:(bel s1 (loc loc1 (234 543)))
2:(bel s2 (loc loc2 (553 543)))


******** Process Manager **********
0[0]: MALLET team-plan node: bombing-run []
  1[0]: par node
    2[0]: MALLET do node: [do, p1, [protect-assets]]
      3[0]: seq node
        4[0]: MALLET plan node: protect-assets
          5[0]: while node: [cond, [=, 1, 1]]
            138[?]: if node: [cond, [threat, ?threat, ?loc]]
    8[0]: seq node
      57[0]: par node
        58[0]: MALLET do node: [do, s1, [scout-for-radar]]
          59[0]: MALLET plan node: active-inform 4
            60[0]: par node
              61[0]: forall node : [[info-need, ?type, 4, ?info-need]]
```

```
                62[0]: seq node
                  63[0]: while node: [cond, [not, [loc, radar1,
?loc1]], [needs-info, s1, TERM, 4, [loc, radar1, ?loc1]]]
                    139[?]: (NOP)
                65[0]: seq node
                  66[0]: while node: [cond, [not, [loc, radar2,
?loc2]], [needs-info, s1, TERM, 4, [loc, radar2, ?loc2]]]
                    140[?]: (NOP)
                68[0]: seq node
                  69[0]: while node: [cond, [not, [loc, radar3,
?loc3]], [needs-info, s1, TERM, 4, [loc, radar3, ?loc3]]]
                    141[?]: (NOP)
        72[0]: MALLET do node: [do, s1, [scout-for-num-forces]]
          73[0]: MALLET plan node: active-inform 5
            74[0]: par node
              75[0]: forall node : [[info-need, ?type, 5, ?info-need]]
                76[0]: seq node
                  77[0]: while node: [cond, [not, [num-forces, ?num]],
[needs-info, s1, TERM, 5, [num-forces, ?num]]]
                    142[?]: (NOP)
        80[0]: MALLET do node: [do, s2, [scout-for-air-defense]]
          81[0]: MALLET plan node: active-inform 6
            82[0]: par node
              83[0]: forall node : [[info-need, ?type, 6, ?info-need]]
                84[0]: seq node
                  143[0]: if node: [cond, [loc, defense1, ?loc1],
[needs-info, s2, TERM, 6, [loc, defense1, ?loc1]]]
                    144[0]: MALLET ioper node: (say s2 (loc defense1
(276 4476 0)))
                87[0]: seq node
                  88[0]: while node: [cond, [not, [loc, defense2,
?loc2]], [needs-info, s2, TERM, 6, [loc, defense2, ?loc2]]]
                    145[?]: (NOP)
        91[0]: MALLET do node: [do, b1, [strike-target, target1]]
          92[0]: MALLET plan node: active-inform 7
            93[0]: par node
              94[0]: forall node : [[info-need, ?type, 7, ?info-need]]
                95[0]: seq node
                  96[0]: while node: [cond, [not, [done, b1, [deploy-
payload, deploy-payload, ?a, ?b, ?c]]], [needs-info, b1, TERM, 7,
[done, b1, [deploy-payload, deploy-payload, ?a, ?b, ?c]]]]
                    146[?]: (NOP)
                98[0]: seq node
                  99[0]: while node: [cond, [not, [loc, target1, [?lat,
?long, ?alt]]], [needs-info, b1, PROC, 7, [loc, target1, [?lat, ?long,
?alt]]]]
                    147[?]: (NOP)
        102[0]: MALLET do node: [do, b2, [strike-target, target2]]
          103[0]: MALLET plan node: active-inform 8
            104[0]: par node
              105[0]: forall node : [[info-need, ?type, 8, ?info-need]]
                106[0]: seq node
```

```
                  107[0]: while node: [cond, [not, [done, b2, [deploy-
payload, deploy-payload, ?a, ?b, ?c]]], [needs-info, b2, TERM, 8,
[done, b2, [deploy-payload, deploy-payload, ?a, ?b, ?c]]]]
                    148[?]: (NOP)
                109[0]: seq node
                  110[0]: while node: [cond, [not, [loc, target2,
[?lat, ?long, ?alt]]], [needs-info, b2, PROC, 8, [loc, target2, [?lat,
?long, ?alt]]]]
                    149[?]: (NOP)
```

APPENDIX G

BOA SYNTAX AND NOTES

BOA - Multi-agent Belief Maintenance and Theorem-Prover
Ryan Rozich

3/19/2003

Introduction

The multi-agent belief maintenance module was specified and implemented in Java by Dr. Thomas Ioerger. This document was written by Ryan Rozich as a set of personal notes for learning how to use this software. This document might be turned into a quick start guide for anyone looking to use this software as a multi-agent belief maintenance module for intelligent agents. This does not go into the formal specifications or semantics of the belief reasoning process; Dr. Ioerger has written two papers (currently unpublished) about the formal semantics for this system.

Why do we need this specialized theorem prover for belief-maintenance? Can’t we just implement a specialized predicate “bel” in a normal theorem-prover like JARE to get the same effect? Or why can’t we implement more theoretical models of belief like modal-logics in order to do belief reasoning. Using a generic theorem prover to do belief reasoning by adding some special predicates masks some of the intricacies of performing belief reasoning and maintenance in a multi-agent environment. There are two things that make multi-agent belief reasoning difficult. First, many times agents will come to *conflicting conclusions* about a certain belief, for instance the agent may receive a communication message that the light is off in the room when the agent happens to be in the room and can see that the light is on. Resolving conflicting values for belief predicates involves reasoning about the *justification* for that belief. Second, certain belief predicates are *dependant* upon the values of other predicates and therefore must be sorted by dependency before being evaluated. These two aspects of belief maintenance/reasoning make it difficult to use a generic theorem prover for this task.

On the other hand, a large chunk of the belief reasoning literature involves using modal logics to reason about beliefs. While this is a powerful theoretical tool for reasoning about belief, the task of implementing these formal logics is difficult and we are not aware of any practical implementations of them at this time.

While not as powerful as the formal models of belief based on modal logics (i.e. this does not handle things like nested belief), this is a practical, pragmatic implementation with a formal semantics for reasoning about beliefs of others and ourselves. It handles things such as assumptions, persistence, inference, it models the belief state of a predicate as true, false, unknown, or *knows-whether-or-not*, it handles different types of justifications and dependencies among different beliefs. This module was written in Java and is designed to be a drop-in replacement for the JARE theorem-prover.

1. Concepts
1a. Belief
A Belief is a fact (*what*) belonging to an agent (*who*) that has a *strength* (credibility of the belief), and finally the belief has a *status* (known, unknown, whether) and a value (true, false, or other value). If a belief is a normal fact then the value will be either true or false, a belief can also be a function that can take on my values such as (val (room-temp) 65), the value in this case is not true or false, it is 65. The agent's beliefs (about his own or others beliefs) are contained in the BeliefDB.

The syntax for a belief in this module is:

```
<bel> ::= (bel <agt> <bel-pf>) | <bel-pf>
<bel-pf> ::= <bel-p> | <bel-f>
<bel-p> ::= <pred> | (not <pred>) | (unknown <pred>) | (whether <pred>)
<bel-f> ::= (val <pred> <any>) | (val <pred> known)| (val <pred>
unknown)
<pred> ::= (<name> <arg>*)
```

One of the first things that you will notice as a new feature of this system (over theorem provers like JARE) is that beliefs can explicitly be tagged false (not) unknown, or whether. Also there is the addition of values (functions) which do not evaluate to true or false but some other value.

Syntax Note: The status flag for functions are treated differently:

```
(VAL <PRED> KNOWN) → STATUS=WHETHER

(VAL <PRED> UNKNOWN) → STATUS=UNKNOWN

(VAL <PRED> <VALUE>) → STATUS=KNOWN
```

In other words, if you want to define the function value as unknown or whether use these constructs and not something that looks like this:

```
(unknown (val <pred>)) ; wrong
(whether (val <pred))  ; wrong
```

1b.BeliefDB

The BeleifDB holds, for a single agent, all of his own beliefs and his model of other agents beliefs. This object allows us to insert new beliefs in a consistent way and also query for the beliefs of others or ourselves. The BeliefDB is the drop-in replacement for JARE as a theorem prover. While the API calls to JARE and BeliefDB are identical, the file formats are slightly different and some work needs to be done to convert the syntax of JARE files into BeliefDB files.

1c. Justifications

Justifications are a property of any new fact/belief that is about to be added to a beliefDB, it can be thought of as *what reason (justification) does this agent have to believe this fact*. This is needed because agents might believe this because they just observed it, they inferred it form other facts in their KB, they simply persist in believing the fact over time (in the absence of other information). Justifications provide a way for certain facts to take precedence over others in order to maintain a consistent KB (to avoid the KB from containing conflicting information). That is, beliefs backed by justifications of higher strength are never overwritten by beliefs of lower strength.

Justifications listed in order of strength (see the section below for more detailed syntax and semantics):

Default - Believed in absence of any other information
Persist - These facts persist from one moment to the next
Infer - Define rules to derive new facts from know information
Action- Define what agents know about the effects of others actions
Obs - Define what agents can observe in the environment
DirObs - Define facts that we can directly infer from our senses
Fact - Tautological facts in the world.
Init - Things that are initially true in the world. Is this really a stronger form of belief than fact?? Why don’t init justs show up when I do a (showj)??

Direct Assertion - Things directly asserted

Expanded versus unexpanded justifications??

1d. File Header

First lines must be

```
  (declare (agents <agent names>))
  (declare (self <self-name>))
```

1e. Special Facts

```
  (fact (agent Ag1))
  (fact (agent Ag2))
  (fact (self Ag1))
```

2. Belief Maintenance Shell
The belief maintenance shell is a way to experiment and test the functionality of this belief maintenance module interactively by giving commands interactively to the shell.

2a. Running the Shell

```
  >java Jare.Bel
```

2c. Commands
The following is a list of commands available to the user of the shell:

1. **load** – load a belief file into the system
   ```
   <load> ::= (load <filename>)
   ```

   For example
   ```
   (load rules.txt)
   ```

2. **quit** – Exit the shell
   <quit> ::= quit

3. **query** – query the database
   ```
   <query> ::= (query <bel>)
   <bel> ::= (bel <agt> <bel-pf>) | <bel-pf>
   <bel-pf> ::= <bel-p> | <bel-f>
   <bel-p> ::= <pred> | (not <pred>) | (unknown <pred>) | (whether <pred>)
   <bel-f> ::= (val <pred> <any>) | (val <pred> known)| (val <pred>
   unknown)
   ```

Some examples

```
(query (color red))
(query (color ?x))
(query (bel Ag1 (color ?x)))
(query (bel ?ag (color ?x)))
(query (unknown (alive ?x)))
(query (bel Ag1 (unknown (alive ?x))))
(query (val (loc Ag2) ?loc))
(query (bel Ag1 (val (loc Ag2) ?loc)))
```

4. **update -** Creates a new beliefDB from the current beliefDB and current beliefs.
```
<update> :: = “(update)”
```

5. **assert** – Assert a direct-assertion belief into the database
```
<assert> ::= (assert <bel>)
<bel> ::= (bel <agt> <bel-pf>) | <bel-pf>
<bel-pf> ::= <bel-p> | <bel-f>
<bel-p> ::= <pred> | (not <pred>) | (unknown <pred>) | (whether <pred>)
<bel-f> ::= (val <pred> <any>) | (val <pred> known)| (val <pred>
unknown)
<pred> ::= (<name> <arg>*)
```

6. **showb -** Lists all of the beliefs for all of the agents. Sorted by agent.
```
<showb> ::= ”(showb)”
```

Output is of the form

```
8:(bel Ag1 (unknown (alive wumpus)))
8:(bel Ag1 (Q))
8:(bel Ag1 (val (loc Ag2) roomA))
8:(bel Ag1 (val (loc Ag1) roomB))
8:(bel Ag1 (not (light-on roomA)))
8:(bel Ag1 (light-on roomB))
7:(bel Ag1 (color red))
7:(bel Ag1 (color green))
7:(bel Ag1 (color blue))
7:(bel Ag1 (self Ag1))
7:(bel Ag1 (agent Ag1))
7:(bel Ag1 (agent Ag2))
3:(bel Ag1 (not (P)))
7:(bel Ag2 (color red))
5:(bel Ag2 (not (light-on roomA)))
```

The first number is the *strength* of the justification (See Justifications section above) followed by (bel <agent> <pred>) where <agent> is the name of the agent that believes <pred>.

7. **showj** – Show all justifications

```
<showj> :== (showj)
```

8. **defrule** – Create an inference justification

```
(defrule <bel> <cond>)
<bel> ::= (bel <agt> <bel-pf>) | <bel-pf>
<bel-pf> ::= <bel-p> | <bel-f>
<bel-p> ::= <pred> | (not <pred>) | (unknown <pred>) | (whether <pred>)
<bel-f> ::= (val <pred> <any>) | (val <pred> known)| (val <pred>
unknown)
<pred> ::= (<name> <arg>*)
<cond> ::= <bel>*         // note: for >1, use no AND or extra parens
```

9. **fact** – Create a fact justification

Tautological facts in the world. These cannot be changed. For instance if we define (fact (color red)) and later try to (assert (not (color red))), (not (color red)) will not be asserted because it conflicts with the previous fact.

```
(fact <bel>)          // just a rule with 0 antecedents
<bel> ::= (bel <agt> <bel-pf>) | <bel-pf>
<bel-pf> ::= <bel-p> | <bel-f>
<bel-p> ::= <pred> | (not <pred>) | (unknown <pred>) | (whether <pred>)
<bel-f> ::= (val <pred> <any>) | (val <pred> known)| (val <pred>
unknown)
<pred> ::= (<name> <arg>*)
```

10. **init** – Create an init justification

```
(init <bel>)
<bel> ::= (bel <agt> <bel-pf>) | <bel-pf>
<bel-pf> ::= <bel-p> | <bel-f>
<bel-p> ::= <pred> | (not <pred>) | (unknown <pred>) | (whether <pred>)
<bel-f> ::= (val <pred> <any>) | (val <pred> known)| (val <pred>
unknown)
<pred> ::= (<name> <arg>*)
```

11. **default** – Create a default justification

```
<default> :: = (default <bel>)
<bel> ::= (bel <agt> <bel-pf>) | <bel-pf>
<bel-pf> ::= <bel-p> | <bel-f>
<bel-p> ::= <pred> | (not <pred>) | (unknown <pred>) | (whether <pred>)
<bel-f> ::= (val <pred> <any>) | (val <pred> known)| (val <pred>
unknown)
<pred> ::= (<name> <arg>*)
```

12. **persist** – Create a persist justification

Persist specifies the predicates whose values persist from one moment to the next. Notice that instead of taking a <bel> like the other constructs, this takes an <unspec-b> which basically indicates the name of the predicate whose truth value persists, without giving the actual truth value. For instance if we specify that:

```
(persist (bel ?ag (wumpus-state dead)))
```

than any beliefs (wumpus-state dead) or (not (wumpus-state dead)) persist from one moment to the next (i.e. they do not go away). Therefore, it is not nessesary (and bad form) to declare `(persist (bel ?ag (not (wumpus-state dead))))`. This also works for values, just leave the value off of the end, such as:

```
(persist (bel ?ag (val (room-temp))))
```

Therefore `(val (room-temp) 65) (val (room-temp) 70)` etc, etc will all persist in all agents KB as a result of this statement.

```
<persist> :: = (persist <unspec-b>)
<unspec> ::= <pred> | (val <pred>)  // doesn't commit to value
<unspec-b> ::= (bel <ag> <unspec>)
```

13. **obs** – Create an observability justification
Obs is used to track *other agents* (i.e. not the *self* agents) beliefs based on conditions in which that agents can observe things.

```
<obs> ::= (obs (bel <ag> <unspec>) <cond>)//new: must name believer,
assume not self
<unspec> ::= <pred> | (val <pred>)  // doesn't commit to value
```

14. **direct-obs** – Create a direct-observability justification
Direct-obs is used to track the *self agents* own beliefs (i.e. not beliefs of other agents – use 'obs' for that). <bel-pf> is asserted if the agent receives the given <sense> and if the condition <cond> unifies.

```
<direct-obs> ::= (direct-obs <sense> <bel-pf> <cond>)
<bel-pf> ::= <bel-p> | <bel-f>
<bel-p> ::= <pred> | (not <pred>) | (unknown <pred>) | (whether <pred>)
<bel-f> ::= (val <pred> <any>) | (val <pred> known)| (val <pred>
unknown)
<cond> ::= <bel>*   // note: for >1, use no AND or extra parens
<bel> ::= (bel <agt> <bel-pf>) | <bel-pf>
```

?? when I make an assertion and do a '(showj)' the 'cond' portion of the of the DirectObs justification does not have the sense predicate in it. It seems like it shoud ??

15. **action** – Create an action justification
```
<action> ::=
 (action <act> [(context <cond>)] [(pre-cond <cond>)] [(effects
<cond>)])
<act> ::= (do <ag> <pred>)
<bel> ::= (bel <agt> <bel-pf>) | <bel-pf>
<bel-pf> ::= <bel-p> | <bel-f>
<bel-p> ::= <pred> | (not <pred>) | (unknown <pred>) | (whether <pred>)
<bel-f> ::= (val <pred> <any>) | (val <pred> known)| (val <pred>
unknown)
```

```
<pred> ::= (<name> <arg>*)
<cond> ::= <bel>*         // note: for >1, use no AND or extra parens
```

16. **declare** - ??

3. Belief File Format
Taken from the comments in Dr. Ioergers Bel.java code:

Syntax
  the first two expressions of the file must be:

```
 (declare (agents <name>*))
 (declare (self <name>))


(defrule <bel> <cond>)
(fact <bel>)          // just a rule with 0 antecedents
(init <bel>)
(default <bel>)
(persist <unspec-b>)
(obs (bel <ag> <unspec>) <cond>)//new: must name believer, assume not
self
(direct-obs <sense> <bel-pf> <cond>)
(action <act> [(context <cond>)] [(pre-cond <cond>)] [(effects
<cond>)])


<bel> ::= (bel <agt> <bel-pf>) | <bel-pf>
<bel-pf> ::= <bel-p> | <bel-f>
<bel-p> ::= <pred> | (not <pred>) | (unknown <pred>) | (whether <pred>)
<bel-f> ::= (val <pred> <any>) | (val <pred> known)| (val <pred>
unknown)
<pred> ::= (<name> <arg>*)


<cond> ::= <bel>*         // note: for >1, use no AND or extra parens
<unspec> ::= <pred> | (val <pred>)  // doesn't commit to value
<unspec-b> ::= (bel <ag> <unspec>)
<act> ::= (do <ag> <pred>)
```

Note: if head of rule specifies a believer, then it will only be applied to the model of that agent's beliefs; however, the KE is responsible for ensuring that antecedents don't depend on other agent's beliefs (except for possibly self)

Also, if bel has agent variable, then it must only depend on other beliefs of same variable in antecedents, or possibly self

Whole rule can have at most one agent variable otherwise would have to take cross product of dependencies (Ag x Ag) just make a special check on this and punt if detected

Suffix symbol with :ns, as in ?ag:ns will get recognized during macro expansion (would have liked to use ?ag/ns, but this causes parser to not halt...)

VITA

Born in Joliet, Illinois, Ryan Rozich received his B.S. degree from the Department of Computer Science at Texas A&M University in 2001. As an undergraduate, he was a research assistant, working on “FURL” (for Fuzzy Rule Learner) a machine learning/theory revision approach to learning fuzzy rules. This work was published in IEEE International Conference on Fuzzy Systems (FUZZ-IEEE) 2002. As a graduate student at Texas A&M, he worked as a research assistant with the MURI (Multi University Research Initiative) team, on multi-agent systems for team training. During this time, he worked on “Process Manager” an individual agent kernel and “CAST-PM” which is an implementation of that CAST (Collaborative Agents Simulating Teamwork) multi-agent architecture which builds on the Process Manager kernel. He also worked on the MALLET team process language and the BOA multi-agent belief maintenance system.

His research interests include artificial intelligence, machine learning/data mining, and multi-agent systems.

Ryan Rozich

12445 Alameda Trace Circle Apt. 936

Austin, TX 78727