A Knowledge-Based Approach for Designing Intelligent Team Training Systems

Jianwen Yin, Michael S. Miller, Thomas R. Ioerger, John Yen, Richard A. Volz

Department of Computer Science

Texas A&M University

College Station, Texas 77843-3112

{jianweny, mmiller, ioerger, yen, volz}@cs.tamu.edu

ABSTRACT

This paper presents a knowledge approach to designing team training systems using intelligent agents. We envision a computer-based training system in which teams are trained by putting them through scenarios, which allow them to practice their team skills. There are two important roles that intelligent agents can play; these are virtual team members, and tutors. To carry out these functions, these agents must be equipped with an understanding of the task domain, the team structure, the selected decision-making process and their beliefs about other team members' mental states. Even though existing agent teamwork models incorporate many of the elements listed above, they have not focused on analyzing information needs of team members to support proactive agent interactions. To encode the team knowledge, we have developed a representation language, based on the BDI model, called MALLET. A Petri Net model of an individual agent's plans and information needs can be derived from the role description represented in MALLET, and the IARG (Inter-Agent Rule Generator) algorithm is introduced to detect information flow and generate team interactions.

Keywords

Teamwork, Agent-based team training, collaboration, coordination, multi-agent teams

1. INTRODUCTION

Training is a multi-billion dollar/year industry critical to the success of many industries. However, training is very expensive, hence there is a great need to reduce costs through automation. One aspect of training that has recently become important is team training [2]. Teamwork is important in many disciplines, from business management to sports to emergency response.

Our definition of a team is a group of entities (humans or agents) that are working together to achieve a goal that could not be accomplished as effectively (or at all) by any one of them alone. Team members often play unique roles, which require unique skills and resources. Many teams are hierarchical, with a chain-of-command and leadership or authority roles. All teams have to deal in one way or another with sharing information and distributed decision-making (also called cooperation or collaboration). In team training, the focus is not on each individual's skills (which are typically learned offline), but on optimizing interactions, such as communications efficiency, maintaining situational awareness, and the effectiveness of group decision-making [14].

A great deal of research has been invested in developing computer-based technologies for training, such as multimedia and ITS (Intelligent Tutoring Systems) [15]. ITS systems attempt to carry out user modeling to determine what the student knows or doesn't know. They tend to fall into two categories. The first is "error taxonomies", in which the potential deficiencies in knowledge are anticipated beforehand and are directly looked for in the testing. The second is "overlay approaches" [16], in which a cognitive model of the student's understanding is constructed and compared to an expert's model of the task to determine differences. Current ITS systems typically focus only on individuals.

Intelligent agents can help extend these methods to build ITTS (Intelligent Team-Training Systems). We envision a computer-based training system in which teams are trained by putting them through scenarios (simulations) which allow them to exercise and refine their team skills. There are two important roles that intelligent agents can play in such systems. First, we can have virtual agents that substitute for other team members. This allows for partial team training, which could provide huge cost savings, but relies on producing as realistic of behavior and interactions as possible. A second major role for agents to play is the role of coaches, which relieves the load on human trainers.

To carry out these functions, the intelligent agents must be equipped with an understanding of the task domain, the team structure, and the selected decision-making process. An ITTS agent must reason not only about its goals and capabilities, but also about the goals of the team and other team members and

about shared responsibilities or commitments. This requires what is known as belief reasoning. This view is based on the vast amount of literature that has been written about "shared mental models" [17], which are thought to be essential to effective teamwork.

In this paper, we introduce a new multi-agent architecture that supports the type of belief reasoning that is required for building ITTS. Our approach has three components.

First, we will introduce the team-description language called MALLET (Multi-Agent Logic Language for Encoding Teamwork), which defines the syntax used for encoding teamwork based on our team ontology. Our language will provide relations such as "role", "responsibility", "stage", etc., along with plans and operators, for describing the domain and ideal team behavior. Our ontology defines the semantics of those predicates used in encoding teamwork. Although there are other methods for automatically generating coordinated team-behaviors (joint intentions [1,11]; contract nets [6], multi-agent decision theory [13]), we take a more knowledge-based approach because:

- We are interested in structured domains with well-defined roles and well documented procedures,
- The virtual agents must be able to interact with human trainees and communicate in realistic ways,
- The explicit representation of goal hierarchies and intentions will be important for diagnosing problems with team behavior and providing useful feedback.

Tambe's work [1] focuses on establishing the joint intentions of team members in trying to achieve a joint goal, but not on the information needs of a team member in order to provide information proactively. This is a focus of our approach.

Second, given a description of a team in MALLET, we will then define an interpreter that converts this description into an "executable" form. General belief reasoning (i.e. via theorem proving) can have high computational complexity [10]. The actions and interactions of a team can easily be encoded in Petri Nets, however, which is a natural representation for actions, synchronization, parallelism, etc. Georgeff's PRS system [3] is designed for executing plans in a dynamic environment, but does not incorporate any analysis about information needs for the plan. However, we wish to find the information needs of each team member through the analysis of the plan. To do this, we use Petri Nets to explicitly capture plan knowledge as well as information needs. We show how descriptions in our formal language can be translated into Petri Nets for each agent in the team. There are some team situations that Petri Nets cannot represent, but we feel that this is an adequate starting point. This commitment is a choice on the spectrum of tradeoffs between expressiveness and efficiency.

The third component is to provide a decision-making algorithm. It is not enough for a virtual agent to make decisions based on its own Petri Net. As we have argued above, it is crucial to consider the mental states of other team members. This is where the belief reasoning comes in: each agent tracks the state of the other agents (virtual or human), and can then choose actions based on their beliefs. We define an Inter-Agent Rule Generator (IARG), which is an algorithm that makes these decisions for an agent in the context of the whole team. Hence the focus of our approach is on collaborative execution of plans within a dynamic environment, rather than planning or organizing.

Understanding the actions of an individual on a team is more complicated because their decision-making explicitly involves reasoning about the other members of the team (e.g. their beliefs, roles, etc.), and their actions may be implicitly in support of a team goal or another agent. Our approach is to model team members as maintaining simplified models of the mental states of all the other members on the team. To avoid issues of computational complexity with belief reasoning (e.g. via modal logics), we use Petri Nets as an approximate representation of these mental states. Then when a team member needs to decide what to do, they can not only reason about what actions would achieve their own goals, but they can reason about the state and needs of others. In particular in this paper, we focus on two effects: by making teamwork efficient through anticipating the actions and expectations of others (e.g. by knowing others roles, commitments, and capabilities), and by information exchange (knowing who to ask for information, or providing proactively just when it is needed by someone else to accomplish their task).

2. MALLET: A Multi-Agent Language for Encoding Teamwork

The ontology underlying our framework is based on the BDI model (Belief represents the knowledge of the agent, Desire represents the general goals of the agent and Intention represents the selected plans of the agent [12]). The purpose of presenting an ontology is to identify the general concepts and relationships that occur in teamwork across multiple domains and give them formal definitions that can be used as the basis of a team-description language with primitive terms with well-specified meanings. MALLET is a language based on predicate logic that allows the encoding of teamwork. Being a logic-based language, MALLET provides a number of pre-defined predicates that can be used to express how a team is supposed to work in each domain. Some basic predicates allow the declaration of the types of terms being used, i.e. they are unary predicates that associate symbols with specific classes of teamwork concepts. Other predicates represent the relation between these terms.

2.1 Basic Object Type Predicates

Some concepts in the team domain need to be associated with a specific class of objects.

- *Team-member(x)*

We need to have a way to define who is in the team. By using the unary predicate *Team-member(x)*, we define that x is a member of the team. *Virtual-agent(x)* is used to define *Team-member(x)* as a virtual agent and *Human-trainee(x)* is used to define *Team-member(x)* as a human trainee.

- *Role(x), Plays_role(x,y)*

The team structure and process is described generically in the terms of roles (rather than specific individuals as team members). The unary predicate *Role(x)* defines x as a role. For example, in a volleyball team, we might have *Role(server)*, *Role(setter)*, *Role(spiker)*, etc. It makes more sense to assign responsibilities and relate capabilities to these generic roles. Only if a member of a team plays a certain role, does he/she become interesting to us in the team. *Plays_role(x,y)* defines the

relationship between a member x and the role y. We have *[∀x,y Plays_role(x,y) =>Team-member(x) ∩Role(y)]*.

- *Stage(x)*

Stages turn out to be very useful for describing teams in many real world domains. For example, in flight control for the space shuttle, there are a sequence of specific stages that the team must go through, such as launch, orbit, and landing. The unary predicate *Stage(x)* define x as a stage. Team goals and individual responsibilities can change from stage to stage. We assume that the stages are totally ordered, which can be represented in Interval Logic as *Before(x,y)*.

- *Condition(x), Goal(G,<set-of-conditions>)*

Condition (x) defines a subset of states of the world. In MALLET, we treat goals as propositional conditions, although some agent environments might involve more complex constraints. We also allow goals to be specified as conjunctions of conditions as well. Note that *Goal (G, <set-of-conditions>)* is only used to associate the name G with a set of conditions, not to specify who has it.

2.2 Actions and Plans

Actions are assumed to be primitive and cannot be interrupted. Plans can be viewed as a hierarchical structure composed of sub-plans, and the leaves of the hierarchy representing primitive actions. In MALLET, an action can be defined as a 3-tuple, similar to STRIPS operators: *Action(Action-name, <Set-of-pre-conditions>,<Set-of-post-conditions>)*, in which Action-name is the name of the action, and Set-of-pre-conditions and Set-of-post-conditions are two sets of conditions. Plans can serve multiple goals, and for one goal, there could be multiple plans for how to achieve it. In a manner similar to that introduced in [3], we represent plans through a procedural language as introduced in [9]. A plan can be defined by a 6-tuple in MALLET as *Plan(plan-name, <Set-of-pre-conditions>, <Set-of-actions>, <Set-of-post-conditions>, <Set-of-causal-links>, <Set-of-action-ordering-constraints>)*. We assume that for any pre-condition of any action in the <Set-of-actions> of the plan, if it is not a post-condition of any action in the <Set-of-action> of the plan, then it should be a pre-condition in the <Set-of-pre-conditions> of the plan. Similarly, we assume that for any post-condition of any action in the <Set-of-actions> of the plan, if it is not a pre-condition of any action in the <Set-of-action> of the plan, then it should be a post-condition in the <Set-of-post-conditions> of the plan. We define action-ordering and causal-links constraints as: *Before(A1,A2)* which means action A1 must occur sometime before action A2. According to [9], "A causal link is written as Si $\xrightarrow{c}$ Sj and read as 'Si achieves c for Sj.' Causal links serve to record the purpose(s) of steps in the plan: here a purpose of Si is to achieve the precondition C of Sj." In MALLET, such a casual link is represented as *Achieve(C, Si, Sj)*. To relate goals with plans, we define *Goal-to-plans(goal,<set-of-plans>)*, which identifies the plans for achieving the goal.

2.3 Responsibilities

The core of MALLET is based on several predicates that specify relations among the basic classes of objects, which will be used for the team interaction generation. We begin by defining responsibilities. Responsibilities are relationships between a set of roles, a goal, and a stage: *Responsibility(<role-set>, <goal>, <stage>)*, is used to mean that a set of roles share the responsibility to reach goal <goal> in stage <stage>. The semantics of these expressions is based on the work of Cohen and Levesque [11], in that responsibilities map onto joint intentions and have the same effect of committing agents to actions. To further differentiate kinds of responsibilities, we define:

- Redundant responsibilities

Consider responsibilities shared by multiple agents such that any one of the agents can carry them out independently and if multiple agents take actions towards them, there will not be any damage in accomplishing them except for unnecessary effort. We name this kind of responsibility *redundant responsibility*. For example, in a battlefield, more than two agents in a team take the responsibility of monitoring the movement of the enemy and reporting to the commander. If both of them report the same information to the commander, this causes no harm, except it may waste some communication bandwidth. In MALLET, we represent redundant responsibilities as *Or-responsibility(<role-set>, <goal>, <stage>)*. When there is only one role in the role-set, the responsibility becomes an individual responsibility.

- Shared competitive responsibility

Consider those responsibilities shared by multiple agents such that any one of the agents can carry them out independently, but if multiple agents take actions toward them, there will be some risk of failure because of conflict; we name this kind of responsibility *shared competitive responsibility*. For example, in the volleyball domain, all team members have the responsibility to save the ball in the defense stage, but if more than one of them takes action, they may collide with each other and lose the point. In MALLET, we represent shared competitive responsibilities as *Xor-responsibility(<role-set>, <goal>, <stage>)*.

- Shared complementary responsibility

Consider those responsibilities that require multiple agents to cooperate in carrying out those responsibilities. We name them *shared complementary responsibility*. For example, two agents share the responsibility of moving a heavy table together. In MALLET, we define shared complementary responsibility as *And-responsibility(<role-set>, <goal>, <stage>)*, which requires simultaneous action, and possible communication among the roles to synchronize their actions.

We have the following three observations related to the concepts of different kinds of responsibilities.

- If the team has *Or-responsibility(R, G, s)*, ∀r1, r2 ∈R, r1≠r2, r1 and r2 need to communicate to decide that at least one of them should have the persistent goal G in stage s.
- If the team has *Xor-responsibility(R, G, s)*, ∀r1, r2 ∈R, r1≠r2, r1 and r2 need to communicate to decide that one of them should have the persistent goal G and all the others will not have the goal G in stage s.
- If the team has *And-responsibility(R, G, s)*, then ∀r1, r2 ∈R, r1≠r2, r1 and r2 need to communicate to build the joint intention G in stage s and synchronize so that each waits to act until the other is ready.

2.4 Capabilities

Capabilities are a relationship between roles and actions. In contrast to responsibilities, capabilities specify what team members *can* do, instead of *should* do. To play a certain role, an agent should have the right kind of capability. Or to put it another way, we can say that each role requires a certain set of capabilities. In MALLET, we define a capability as *Capability (<role-set>, <action>).* Every role in the role-set has the capability for that action. Similarly, to further distinguish the different kinds of capabilities, we define:

- Backup capability

There are certain kinds of actions that can be taken by more than one team member at the same time without interfering with each other, we call them backup capabilities. In MALLET, we represent it as *Or-capability(<Role-set>, <action>).* If the capability is an individual capability with only one role in the Role-set, it can be represented as *Capability(role, <action>).*

- Shared conflicting capability

There are certain kinds of actions that can be taken by more than one role independently, but if taking the action at the same time, the action will turn out to be a failure due to conflict, we call them shared conflicting capabilities. In MALLET, we represent them as *Xor-capability(<Role-set>, <action>).*

- Shared cooperative capability

There are certain kinds of actions that can not be taken by one role independently without the cooperation of other agents; we define these as shared cooperative capabilities. In MALLET, we represent them as *And-capability(Role-set, <action>).*

2.5 Beliefs

Beliefs are important for teamwork, for example to reason about and anticipate the actions or needs of others. Most of the expressions in MALLET can be extended to incorporate beliefs in a straightforward way. Beliefs about roles and responsibilities can be asserted for specific team members via *Bel(agent1,Responsibility(agent2,goal,stage))*, etc. For simplicity, we make the Mutual Belief Assumption, which is that, by default, agents are assumed to all share common knowledge about the roles, capabilities and responsibilities that they are involved in within the team, and they believe that all other agents have the same beliefs. This assumption might have to be relaxed for human team members. Facts or propositions can also be represented in beliefs, but they are more dynamic and agent-specific (e.g. *Bel(agent1,valve-open)*). Finally, we note that individual action operators should be converted to belief statements. For example, if P and Q are pre-conditions for an agent A to take an action X, from another agent's perspective, what is really required for A to try to execute X (besides motivation), is that agent A *believes* that P and Q are true. This has implications for collaboration, since A may elect to achieve P himself, or ask another agent to help; either way, A eventually comes to believe that P becomes true. Simulating this type of multi-agent belief reasoning presents a challenge, since in general, reasoning about beliefs can be intractable [10]. Our Petri Net algorithms introduced below, address this implicitly, at the cost of expressiveness and completeness; this leads to an adequate and efficient tradeoff for approximating the belief reasoning involved.

2.6 Communication

Communication is very important for teamwork and it is communication that provides a way for team members to coordinate and collaborate. Communication is more complicated than Petri Nets can represent. As introduced by Barbuceanu and Fox in [4] and Bradshaw, et. al. in [5], communication can be represented as a 4-tuple Com (Purpose, Protocol, FSM, Intc), in which Purpose is the invocation condition, Protocol is the way to communicate, FSM is a Finite State Machine to describe the communication process and Intc is the interrupt condition. If the FSM ends in a successful state, the communication achieves its purpose successfully, otherwise the communication fails.

Following is a list of protocols that can be used in a multi-agent team: Non-intrusive broadcasting are ones that can be ignored by a team member, intrusive broadcasting for announcements that the team must respond to, intrusive multicasting for use in a sub-team, and one-to-one communication.

The FSM itself is defined as a 5-tuple FSM($Q, \Sigma, \delta, q_0, F$), in which Q is a set of states, Σ is a set of atom actions in communication, such as propose, accept, reject, etc., δ is a transition function, q_0 is the initial state and F is a set of final states which defines the communication policies [5].

Communication is a very interesting topic and we hope to incorporate these ideas into MALLET. However we will not focus our discussion on communication in this paper.

3. An Example Team Domain

In order to illustrate our ontology and representation of teams in MALLET, and to help present our algorithm for constructing Petri Nets and generating team interactions, we introduce a real world example of a team-training domain.

The NASA Mission Control Center consists of a team that is arranged in a hierarchical manner with clearly delineated roles for each team member. The Flight Director (FD) oversees 10 disciplines which each monitor functions on the Space Shuttle. Each discipline is hierarchically organized with a discipline leader and one or more sub-team specialists.

Propulsion Systems (PROP) is one of those disciplines. The Propulsion Systems Officer is responsible for the operation of the Space Shuttle Orbital Maneuvering System (OMS) and Reaction Control System (RCS). These secondary engines are used for orbital corrections, docking operations, and the De-orbit burn. The PROP is assisted by the OMS/RCS Engineering Officer (OREO) and the Consumables Officer (CONS).

The PROP officer is in a vertical chain of command, but he/she must also interface with other disciplines. The sub-team members such as the OREO and CONS officers sit on consoles in a separate room from the Flight Control Room (FCR) called the Multipurpose Support Room (MPSR).

During the launch stage, the FD asks the PROP officer for a status check to see if the discipline is ready for launch. The PROP officer checks with his sub-team. Each sub-team member checks his/her own console and reports to the PROP officer. The PROP officer reports the status back to the FD that they are ready for launch.

Let us take a look at the MALLET representation for the two roles CONS and OREO. We have:

Role(CONS) and **Role**(OREO).

Goal(Report_launch_status).

And-responsibility({CONS, OREO},
Report_launch_status, Launch_stage).

Capability({OREO}, check_OREO_status)

Capability({CONS}, check_CONS_status)

Xor_capability({CONS, OREO},
report_own_status_to_PROP).

We assume that there is only one communication channel to reach PROP by CONS and OREO. Thus they can not report to PROP simultaneously, that is why we have Xor_capability({CONS, OREO},report_own_status_to_PROP).

We assume virtual agents as shown in Figure 1 play CONS and OREO. Based on the *Mutual belief assumption*, all the shared responsibilities and capabilities are mutually understood by all agents involved.

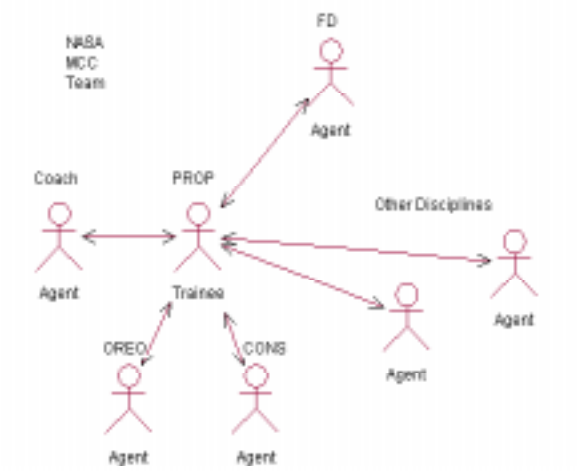

Figure 1. NASA MCC Team

OREO has the following plan:

Plan(
Name: OREO-report-PROP-status,
Pre-condition{ OREO_monitor_system_normal},
Actions:{
Action(check_OREO,{OREO_monitor_system_normal},{OREO_status_ready, OREO_monitor_system_normal })
Action(report_own_status_to_PROP,
{OREO_status_ready} ,{ OREO_finish_reporting_status})
},
Post-condition: {OREO_finish_reporting_status,
OREO_monitor_system_normal},
Causal link: Achieve (OREO_status_ready, check_OREO, report_own_status_to_PROP)
Action-ordering-constraints: before(check_OREO, report_own_status_to_PROP)
)

CONS is similar to OREO.

4. Generating Petri Nets from Plans

Petri Nets have previously been suggested as an appropriate implementation for both intelligent agents [8] and teamwork [7]. Petri Nets are particularly good at representing actions in a symbolic/discrete framework. They can represent the dependence of actions on pre-conditions in a very natural way, i.e. via input places to a transition. The effects of the action become simply output places. However we need to connect these possible transitions to goals, which can be simulated as extra input places that enable firing when a token is present. We can use this mechanism in controlling the choices among different plans. We can handle team stages in a similar way. Petri Nets are also good for representing synchronization requirements. Figure 2 is an example Petri Nets model of the OREO's plan OREO-report-PROP-status generated by our Petri Nets constructing algorithm introduced below.

G: OREO-report-PROP-status (goal place)
PrA1: OREO_monitor_system_normal
A1: check_OREO
PoA1: OREO_status_ready
A2: report_own_status_to_PROP
PoA2: OREO_finish_reporting_status

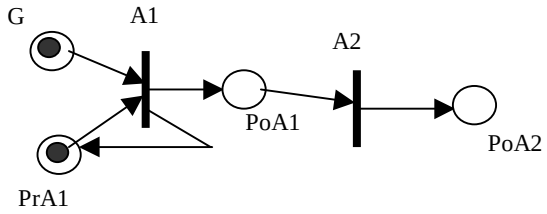

Figure 2. Petri Nets model of OREO plan OREO-report-PROP-status

However, Petri Nets are not able to represent beliefs in such a straightforward way. In order to incorporate agents' beliefs into Petri Nets, we introduce a new type of node called a "belief node" with a more suitable semantics. Belief nodes can hold two types of tokens: T representing that the proposition is believed to be true, and F representing that it is believed to be false. Absence of either type of token indicates lack of belief, or uncertainty, which is an important aspect of many team situations. Hence this becomes a colored Petri Net. Like standard colored Petri Nets, we allow arcs between transitions and belief nodes to be labeled with T or F to restrict the type of token that can satisfy an input to a transition, or is deposited to an output from a transition. However, belief nodes have three unique characteristics:

- If multiple tokens are deposited into a single place, they are either reduced (e.g. multiple T's to one T), or canceled (e.g. if T's and F's occur together; canceling reverts the interpretation back to uncertainty of belief).
- Places corresponding to the same proposition throughout the network are merged. This is important for maintaining consistency. For example, if one transition has an effect P, then the belief that P is true should be propagated to any other transition that depends on this belief.

- "Back-arcs," or connections, are added from the output of each transition back to belief nodes that are its input places, since by default, most actions do not undo pre-conditions (persistence assumption), unless explicitly listed as effects.

Now we are ready to introduce our algorithm for generating Petri Nets for each role on the team. For each role, the algorithm *ConstructPetriNet(Role)* will recursively call the algorithm *ConstructPlanNet(Plan, Role)* to generate a sub-net of the plan passed as an argument and combine the sub-nets into one Petri Net by merging the propositional nodes with the same names. Here the relationship between Role and Plan is indirect. Role is related to goal through *responsibility* and goal is related to plans through the relation *Goal-to-plans*. For each action in a plan, the algorithm *ConstructPlanNet(Plan, Role)* will first translate each pre-condition into an input place, each effect into an output place, and link these to a new transition. After recursively constructing a sub-net for each step in the plan, the algorithm will merge the belief nodes corresponding to the same propositions and connect the sub-nets based on causal links.

In the algorithm *ConstructPlanNet*, goal places serve as a control to the Petri Nets. We know that in a Petri Net, if each input place has a token, then a transition can fire. In our Petri Net model of an agent, we need to have a way to choose which plan to execute. Whenever a goal becomes the agent's current intention, a token will be put into the goal place so that the relevant plan can execute when all other pre-conditions hold. By applying the algorithm, we constructed the Petri Nets of the OREO's plan OREO-report-PROP-status as shown in Figure 2.

ConstructPetriNet(Role)

SubNets =∅.

For each responsibility R such that Role is in its role-set.

Let G be the goal associated with responsibility.

Let P be the plan that can be used to solve goal G.

Append ConstructPlanNet(P,Role) to SubNets.

ConstructPlanNet(Plan, Role)

//The following mentioned sets are all related to the Role.

Create a place labeled by the name of the plan(goal place).

Create input places for each pre-condition of the plan.

Create output places for each post-condition of the plan.

For each action in the set-of-actions, create sub-net P with

Create a transition labeled by the action,

input places for each pre-condition of the action,

And output places for each post-condition of the action.

For each causal link in the set-of-causal-links

merge the output place for the condition in the casual link (a post-condition of the first action) with the input place for the condition in the casual link (a pre-condition of the second action).

For each ordered constraint in the set-of-action-ordering constraints. If there is no causal link between these two actions

Then add a place node between the two action transitions based on their ordering constraints.

For each input place and output place of any action.

Merge those input places and output places of an action transition that are labeled with the same name.

5. Generating Interactions from Petri Nets

In addition to generating the Petri Nets for each agent, we need to determine the interactions that will take place among the agents. In particular, we are concerned with both detecting ambiguities that arise when multiple agents have responsibilities for accomplishing a goal, and determining useful information exchange among agents. In this section, we present an algorithm that accomplishes the following:

1. Detects ambiguities of responsibilities within the team based on the shared knowledge of responsibilities that each agent has, and initiates communication with other team members to resolve the detected ambiguities.
2. Detects information needs of this agent and resolves these needs through requests for information to other agents.
3. Detects the information needs of other agents and resolves these needs by providing known information to other agents that need it.

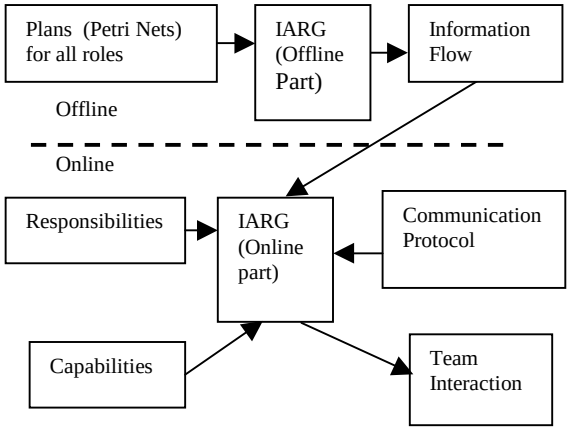

Figure 3. Team Interaction Generation

We accomplish these tasks with our IARG algorithm, which can simulate information flow and generate team interactions. IARG uses both offline and online components, as shown in Figure 3. An agent analyzes the Petri Nets of all the other agents in order to derive information flow and identify propositions that other agents need to know. We define information flow as a 3-tuple: *<Proposition, Providers, Needers>*. *Proposition* is a truth-valued piece of information. *Providers* is the set of roles that can provide the information (i.e. perhaps has the responsibility of achieving and/or maintaining it). *Needers* is the set of roles that need this information. An agent is said to need a piece of information in the sense that the proposition maps onto an input place of a transition of the *Needer* in the Petri Net corresponding to an action that that agent can execute to carry out one of its responsibilities.

After the Petri Net models are generated for all agents involved in a team, we can determine the information flow. For each

agent's Petri Net model, let *input-places(τ)* be the set of all input places of transition τ, and let *output-places(τ)* be the set of all the output places of transition τ[1]. Then we have the following steps to generate the information flow offline. The providers of information are determined by agents that have a transition that outputs tokens to a given belief node (proposition), and the agents that need the information are those that have this proposition as an input to a transition.

- For each agent A, let *Provides(A)* be the set of all belief-node places that are outputs from a transition but not inputs to the same transition: *Provides*(A) = {π | ∃τ s.t. π∈ output-places(τ), and π∉ input-places(τ)), in the Petri Net for A}
- For each agent A, let *Needs(A)* be the set of all belief-node places (places related to belief) that are inputs to a transition of agent A but not in the set of Provides(A): *Needs*(A) = {π | ∃τ s.t. π∈ input-places(τ) in the Petri Net for A, and π∉ *Provides*(A) }
- *Providers*(P) = {A | P∈ *Provides*(A)}, where P is a proposition and A is an agent.
- *Needers*(P) = {A | P∈ *Needs*(A)}, where P is a proposition and A is an agent.

These two sets give us the sets of flows called Information-flow: *{<P, Providers(P), Needers(P)>}*. This is all done offline, as illustrated in the top half of Figure 3. In the online part, based on the plan, the agents' knowledge of information flow, and current beliefs, the IARG algorithm generates team interactions. This is done by determining each agent's current responsibilities, identifying information they need via empty input places to transitions for carrying out those responsibilities – and cases of this in other agents' Petri Nets - and then exchanging the pertinent information via Ask and Tell operators (performatives). This algorithm only determines the set of candidate team interactions (e.g. communications, which supplement the set of actions the agent can take directly). It does not resolve domain-dependent issues such as how specific actions (both direct and communicative) are prioritized and selected, or the choice of communication channels used within the team.

The algorithm starts by identifying all active goals of the agent in the current stage, and then either informing co-responsible agents that the agent intends to accomplish the goal, or invoking communication to resolve conflicts or synchronize, depending on the type of responsibility. Then the agent finds all blocked transitions within its own Petri Net to generate requests for information. Finally, for any propositions that are effects of its own actions, which it knows the truth-value of, the agent communicates this information to other agents needing it (i.e. those agents that have a goal-enabled transition that depends on the proposition, but the corresponding place has no tokens or an incorrect token in the Petri Net for that agent).

Generate-team-interactions*(Role R, Goal G, Information-flow IF)*

PossibleActions=∅

For all goals G that are responsibilities of role R

- If Or-responsibility({R,R_1,R_2,…}, G, stage), then for all R_i (i=1,2,…), append "Tell(R_i,intend(R,G))" to *PossibleActions*
- If And-responsibility({R,R_1,R_2,…}, G, stage), then append "Synchronize[2]({R,R_1,R_2,…},G)" to *PossibleActions*
- If Xor-responsibility({R,R_1,R_2,…}, G, stage), then for all R_i (i=1,2,…), append "Select-Among[1](R,R_i)" to *PossibleActions*

∀τ such that τ is a transition in the Petri Net for R, and ∃γ such that γ∈ input-places(τ), and γ∈ GoalNodes and tokens(γ)≠∅, then

- ∀π ≠ γ such that π∈ input-places(τ) and tokens(π)=∅,

  If <π,*Providers*,*Needers*>∈ *IF* and R∈ *Needers*, then ∀A_i∈ *Providers* append "Ask(A_i,π)" to *PossibleActions*

∀τ such that τ is a transition in the Petri Net for R, and ∀π such that π∈ output-places(τ) and tokens(π)≠∅,

If <π,*Providers*,*Needers*>∈ *IF* and R∈ *Providers* then

∀A_i∈ *Needers* if:

∃ω such the ω is a transition in the Petri Net for A_i with an input goal node with a token in it, and π∈ input-places(ω), and tokens(π,*PetriNet*(A_i)) ≠ tokens(π,*PetriNet*(R))

then append "Tell(A_i,π,token(π))" to PossibleActions

return *PossibleActions*

6. Example

To illustrate this process, we can go back to the shuttle controller team example. First the OREO receives the command from PROP to resolve the goal Report_launch_status, he/she checks his/her responsibilities and finds it is an And-responsibility involving both him/herself and OREO. Second he/she communicates with OREO to setup a joint intention. In Figure 2 we assume now a joint intention has been built. The OREO puts a token in his goal place. Third, OREO will check all the input places of the current transition A1 and finds all the input places that have a T token. Fourth, OREO then fires the transition, deleting all tokens from its input places and adding tokens to its output places. In this case, since PrA1 is a belief that is not changed by A1 so a token is looped back to reset it. Fifth, OREO checks and finds all the input places for the current

[1] We can specify a particular Petri Net as an argument to *input-places()* and *output-places()*, if it is unclear from context.

[2] We look at communications from a high-level perspective. For example communication could occur through observation, actions, speech, or networks. Thus Synchronize means to us that agents agree upon a certain time by which each agent can perform its actions in order to achieve an And-responsibility. Select-Among is a weaker form of Synchronize in that one agent is selected from a team of agents to achieve a Xor-responsibility. Other protocols such as Tell and Ask can be referred to [5].

transition A2 that each needs a token. Sixth, OREO fires the transition A2 to report to the PROP officer of his/her status and reaches the goal state and stops the execution of the plan.

7. Conclusion and future work

In this paper, we have presented the team-description language MALLET based on our ontology of teamwork, which specifies the semantics of roles, responsibilities, and capabilities of a team as well as goals, actions and plans of the team. It also enables the representation of a team member's belief about other member's mental states. Based on this knowledge, a goal-directed agent can be designed to behave as a virtual team member in training. A unique feature of our approach is that it dynamically generates proactive team interactions by analyzing the information flow between agents using the IARG algorithm. The agent-based teamwork model can not only be used to implement virtual team members in an intelligent team training system, it can also serve as the "expert teamwork model" for a coaching agent to assess the process and the performance of a team being trained. In order to do coaching, the agent has to build an appropriate model of the user that explains his actions, which centrally depend on interacting with, and relying on others, in a team context. Our Petri Nets can serve as the user model, and errors can be identified as structural differences (e.g. missing links) compared to an expert model. Such an intelligent team training system can reduce the time and overall cost of training a team staff for domains such as battlespace management, mission control centers, and other team-oriented applications. Future work will focus on doing plan recognition by inferring probable token markings of Petri Nets based on observed actions of agents so as to model their behavior and diagnose problems with cooperation and information flow.

8. Acknowledgement

This research was partially supported by GANN fellowship grant P200A80305 and seed funds from the Texas Engineering Experiment Station for the Training System Sciences and Technology Initiative.

9. References

[1] Tambe, M. Agent architectures for flexible, practical teamwork. In Proceedings of the 14th National Conference on Artificial Intelligence (Providence, Rhode Island 1997), 22-28.

[2] Van Berlo, M. P. W. Systematic development of team training: A review of the literature. TNO Human Factors Research Institute technical report TM-96-B010, 1996.

[3] Georgeff, M. P. and Lansky, A. L. Reactive reasoning and planning. In Proceedings of the 6th National Conference on Artificial Intelligence (1987), 677-682.

[4] Barbuceanu, M. and Fox, M. S. Cool: A language for describing coordination in multi agent systems. In Proceedings of the First International Conference on Multi-Agent Systems (1995), 17-24.

[5] Bradshaw, J.M., Dutfield, S., Benoit, P. and Woolley, J.D. KaoS: Toward an industrial-strength open agent architecture. In Bradshaw, J.M., editor, Software Agents. AAAI/MIT Press, (1997).

[6] Smith, R.G. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. IEEE Transactions on Computers, (1980), C-29 (12): 1104-1113.

[7] Coovert, M.D. and McNelis, K. Team decision making and performance: A review and proposed modeling approach employing Petri Nets. Teams: Their Training and Performance. R.W.Swezey and E. Salas(eds.) (1992).

[8] Moldt, D. and Wienberg, F. Multi-Agent-Systems Based on Colored Petri Nets. In Proceedings of 18th International Conference, ICATPN'97, (1997), 82-101.

[9] Russell, S. and Norvig, P. Artificial Intelligence A Modern Approach. Prentice Hall, Upper Saddle River, New Jersey, (1995).

[10] Halpern, J.Y. and Moses, Y. A guide to completeness and complexity for modal logics of knowledge and belief. Artificial Intelligence, (1992), 54:319-379.

[11] Cohen, P.R. and Levesque, H.J. Teamwork. Nous 25(4), Special Issue on Cognitive Science and Artificial Intelligence (1991), 487-512.

[12] Rao, A., and Georgeff, M. BDI Agents: from theory to practice. In Proceedings of the First International Conference on Multiagent Systems (ICMAS), (1995).

[13] Sen, S. and Arora, N. Using limited information to enhance group stability. International Journal of Human-Computer Studies, (1998), 48:69-82.

[14] Canon-Bowers, J. A. and Salas, E. A framework for developing team performance measures in training. In: Team Performance Assessment and Measurement: Theory, Methods, and Applications. M. T. Brannick, E. Salas, and C. Prince (eds.). Lawrence Erlbaum Associates: Hillsdale, NJ. (1997).

[15] Self, J. Hanging by two Threads: The Evolution of Intelligent Tutoring System Research. In Proceedings of the 4th International Conference on Intelligent Tutoring System. (1998).

[16] Anderson, J.R., Boyle, C.F., Corbett, A.T., and Lewis, M.W. Cognitive modeling and intelligent tutoring. Artificial Intelligence (1990), 42:7-49.

[17] Blickensderfer, E., Cannon-Bowers, J.A., and Salas, E. Theoretical bases for team self-correction: Fostering shared mental models in Advances. Interdisciplinary Studies of Work Teams, JAI Press, (1997), V.4, 249-279.