MALLET–A Multi-Agent Logic Language for Encoding Teamwork

Xiaocong Fan, *Member, IEEE,* John Yen, *Fellow, IEEE,*
Michael Miller, Thomas Ioerger, *Member, IEEE,* and Richard Volz, *Fellow, IEEE*

Abstract

MALLET, a Multi-Agent Logic Language for Encoding Teamwork, is intended to enable expression of teamwork emulating human teamwork, allowing experimentation with different levels and forms of inferred team intelligence. A consequence of this goal is that the actual teamwork behavior is determined by the level of intelligence built into the underlying system as well as the semantics of the language. In this paper, we give the design objectives of MALLET and its syntax, and define an operational semantics for MALLET in terms of a transition system. We also introduce CAST—an interpreter of MALLET, by which we have explored various forms of proactive information exchange behavior embodied in human teamwork. The semantics can be used to guide the implementation of various MALLET interpreters emulating different forms of team intelligence, and to formally study the properties of team-based agents specified in MALLET.

I. Introduction

Agent teamwork has been the focus of a great deal of research in both theories [1], [2], [3], [4] and practices [5], [6], [7], [8]. A team is a group of agents having a shared objective and a shared mental state [2]. While the notion of joint goal (joint intention) provides the glue that binds team members together, it is not sufficient to guarantee that cooperative problem solving will ensue [3]. The agreement of a common recipe among team members is essential for them to achieve their shared objective in an effective and collaborative way [4]. Languages for specifying common recipes (plans) and other teamwork related knowledge are thus highly needed both for agent designers to specify and implement cohesive teamwork behaviors, and for agents themselves to easily interpret and manipulate the mutually committed course of actions so that they could collaborate smoothly both when everything is progressing as planned and when something goes wrong unexpectedly.

The term “team-oriented programming” has been used to refer to both the idea of using a meta-language to describe team behaviors (based on mutual beliefs, joint plans and social structures) [9] and the effort of using a reusable team wrapper for supporting rapid development of agent teams from existing heterogeneous distributed agents [10], [11]. In this paper, we take the former meaning and focus on the semantics of an agent teamwork encoding language called MALLET (Multi-Agent Logic Language for Encoding Teamwork), which has been developed and used in the CAST (Collaborative Agents for Simulating Teamwork) system [8] to specify agents’ individual and teamwork behaviors.

There have been several efforts in defining languages for describing team activities [12], [13], [3]. What distinguishes MALLET from the existing efforts is three-fold. First, MALLET is a language that allows experimentation with different levels and forms of team intelligence. Depending upon the level of intelligence built into the underlying system, one would expect to program team plans somewhat differently. As more intelligence is built into the underlying system, the expression of teamwork behavior can become simpler and more natural from a human understanding perspective.

Second, MALLET is a generic language for encoding teamwork knowledge. Teamwork knowledge may include both declarative knowledge and procedural knowledge. Declarative knowledge (knowing "that") describes objects, events, and their relationships. Procedural knowledge (knowing "how") focuses on the way needed to obtain a result, where the control information for using the knowledge is embedded in the knowledge itself. MALLET supports the specification of both declarative and procedural teamwork knowledge. For instance, MALLET has reserved keywords for specifying team structure-related knowledge (such as who are in a team, what roles an agent can play) as well as inference knowledge (in terms of horn-clauses).

Third, MALLET is a richer language for encoding teamwork process. MALLET has constructs for specifying control flows (e.g., sequential, conditional, iterative) in a team process. Tidhar also adopted such an synthesized approach [9], where the notions of social structure and plan structure respectively correspond to the team structure and team process in our term. While MALLET does not describe team structure in the command and control dimension as Tidhar did, it is more expressive than the simple OR-AND plan graphs and thus more suitable for describing complex team processes. In addition, MALLET allows the constraints for task assignments, preconditions of actions, dynamic agent selection, decision points within a process and termination conditions of a process to be explicitly specified. The recipe language used in [3] lacks the support for specifying decision points in a process, which is often desirable in dealing with uncertainty. While OR nodes of a plan graph [9] can be used for such a purpose, the language cannot specify processes with complex execution orders. Team/agent selection (i.e., the process of selecting a group of agents that have complimentary skills to achieve a given goal) is a key activity for effective collaboration [14]. No existing languages except MALLET allow the task of agent-selection to be explicitly specified in a team process. Using MALLET, a group of agents can collaboratively recruit doers for the subsequent activities based on the constraints associated with

agent-selection statements.

The structure of this paper is as follows. We give some background in the rest of this section. Section II summarizes the design objectives of MALLET, and Section III introduces the syntax of MALLET. We prepare our work in Section IV and give the transition semantics in Section V. Through an example, we show how to use the transition rules to formally reason about the behaviors of team-based agents in Section VI. Comparison is given in Section VIII and Section IX concludes the paper.

A. Background: encoding teamwork knowledge

Several agent architectures have been developed for producing cooperative behaviors among intelligent agents. GRATE* is an implemented system based on the Joint Responsibility model [3]. STEAM (a Shell for TEAMwork) [5] is a hybrid teamwork model built on top of the Soar architecture [15]. The RETSINA model of teamwork (RETSINA-MAS) [7] is built upon the RETSINA individual agent architecture [16]. JACK Teams [17] provides a team-oriented modeling framework by extending JACK Intelligent Agents. In addition, Tidhar investigated team-oriented programming [9] (which is referred to as TOP below) and provided a plan description language based on plan graphs. In the following, we briefly review the expressivity of the above-mentioned approaches to encoding teamwork knowledge from four dimensions: team tasks, dynamic task allocations, decision points, and control constructs.

Team tasks in multi-agent systems can be classified into three categories: atomic team operators, joint team activities, and shared team plans. Atomic team operators refer to those atomic actions that cannot be done by single agent and must involve at least two agents to do it. For instance, lifting a heavy object is a team operator. Before doing a team operator, the associated preconditions should be satisfied by all the involving agents, and the agents should synchronize when performing the action. TOP [9] supports team operators.

A joint team activity is a long-term process involving multiple agents. To execute a joint team activity, it often requires that the involved agents establish joint and individual commitments to the activity, monitor the execution of the activity, broadcast task failures or task irrelevance whenever they occur, and replan the activity if necessary. The notion of *team operator* in STEAM [5] corresponds to this level of team tasks. A joint team activity is typically associated with a joint type specifying the execution constraints. For instance, STEAM uses three primitive role

constraints (a role is an abstract specification of a set of activities in service of a team's overall activity) to specify the relationship between sub-activities of a joint team activity: 1) AND-combination (the whole activity succeeds iff all the sub-activities succeed), 2) OR-combination (the whole succeeds iff any one sub-activity succeeds), and 3) role-dependency (the execution of one sub-activity depends on another). Such constraints can be combined to specify more complex joint team activities.

Shared team plans refer to common recipes that govern the collaboration behaviors of teammates in solving complex problems. A shared plan often involves team formation, information exchange regarding synchronization, task allocation, constraints and temporal ordering of embedded activity invocations, etc. GRATE* [3] has such a language, where trigger conditions and structure of suboperations can be specified for a recipe. RETSINA-MAS [7] also uses the concept of shared plans to coordinate individual behaviors, but it lacks an explicit team plan encoding language. Instead of providing a higher-level planning encoding language, JACK Teams [17] tried to extend a traditional programming language (i.e. Java) with special statements for programming team activities. In JACK, team-oriented behaviors are specified in terms of roles using a construct called *teamplan*. TOP [9] uses *social structures* to govern team formation and it is assumed that each agent participating in the execution of a joint plan knows the details of the whole plan. STEAM [5] lacks support for encoding shared plans.

Dynamic task assignment means the precise group of agents executing a task in a team plan is not compiled in, but can be flexibly determined at run time [5]. To allow agent teams to be adaptive to the dynamic environment, it is desirable for an agent architecture to support dynamic task assignment, especially at the language level. A STEAM agent determines the candidates for a task (role) by matching its own (or other agents') capabilities with the requirements of the role, while avoiding conflicts between the new task and the candidates' existing commitments [5]. Built upon STEAM, a role allocation algorithm was given in [18], where an unassigned role triggers the creation of a role-allocation-role with the responsibility of assigning an agent to the domain-level role through inter-proxies interactions. JACK Teams's support of dynamic task assignment also depends on the concept of role. Aside from plan body, a JACK teamplan declares the roles needed and how the task team is to be established. The plan body, specified in terms of the required roles, is independent of the actual teams performing the roles [17]. Task assignment in RETSINA is based on constrains reasoning (i.e., authority and social parameters),

as well as agent capabilities and role requirements [7]. GRATE* agent assigns tasks in a so-called solution-planning phase [3]. Thus, each agent's responsibility is already determined before executing a joint action. Upon recipe failures (e.g., untenable), GRATE* agents need to enter a replanning process to obtain another recipe workable for the changed environment. In TOP [9], the dynamic task assignment is done upon plan selection. However, none of these systems provide a language-level support for specifying task-allocations within team plans.

A complex teamwork process often involves operation steps where all the team members need to share critical information in order to correctly choose and execute the next course of action. For instance, an echelon unit's plan (e.g., the operational order for a brigade of U.S. Army) usually specifies critical decisions the commanders need to make, and information needed to make these decisions. These decisions are called decision points in the operational order, and the information needs of the commander are called Commanders Critical Information Requirements (CCIR). Based on CCIR, the intelligence officer in the unit and the scouts he/she interacts with are able to proactively deliver information related to CCIR to the commander for making better decisions. However, none of the above-mentioned approaches support the coding of decision points in a team plan.

As far as supports for programming complex team behaviors are concerned, GRATE*, JACK Teams, and TOP have various control constructs. For instance, GRATE*s recipe language has constructs PAR, WHILE and IF for composing parallel, iterative and conditional processes [3]. JACK Teams provides supports for complex team goal handling through statement @parallel, which allows several branches of activity in a teamplan to progress in parallel [17]. A @parallel statement can specify success condition, termination condition, how termination is notified, and whether to monitor and control the parallel execution. @parallel is a powerful mechanism because it allows the specification of when the statement as a whole succeeds: when all branches have terminated successfully (AND), or when any one branch succeeds (OR). TOP [9] has explicit operators for sequencing, non-deterministic choice, and parallelism to specify the ordering of actions.

B. Types of Information Needs in Agent Teamwork

An information need may state that an agent needs to know the truth value of a proposition, or wants to know the values of some arguments of a predicate, where the values could make

the predicate true. Information needs exhibit themselves in different ways. For example, prior to performing a plan or action, an agent typically needs to check whether the plan or action is both physically and epistemically feasible [19]; an agent may inform the decision maker of changes which have been made elsewhere in the process and impinge upon the current decision context [20]; when things go wrong with one agent's activities, the other group members will help exert pressure and do whatever they think is necessary (make failure mutually known) for the collective to succeed in achieving its objective [21]; a team of agents with a joint intention is required to commit to informing others when an individual agent detects that the goal has been accomplished, becomes impossible to achieve, or becomes irrelevant [1].

Yen, Fan and Volz [22] formally identified four types of information-needs usually emerging in the pursuit of team or individual goals.

Action-performing information-need This type of information-needs enables an agent to perform certain (complex) actions, which contributes to the agent's individual commitments to the whole team. Typically, an action-performing information-need is derived from the preconditions of the action.

Decision-making information-need As well as domain actions, those information-needs emerging from the mental action *decision-making* is of particular interest. It helps an agent to reduce uncertainty in the process of making decisions, and consequently enables the agent to rationally select a course-of-action (COA) from several potential choices. In the terminology of the SharedPlans theory, this kind of information-needs makes an agent better equipped to adopt an appropriate intention by reconciling potential intentions serving the same goal. Typically, a decision-point has several branches to be explored, and each branch is associated with factors (e.g., *preference* constraints) that may affect the process of decision-making. The more factors are taken into consideration (i.e., the more relevant information is known), the more likely the decision-maker will make better decisions.

Goal-protection information-need This type of information-needs allows an agent to protect a goal (or an intention) from becoming unachievable. Information regarding potential threats to a goal belongs to this category; knowing such information will help an agent to adjust its behavior to either remove or avoid the threat to its goal. Information regarding conflicts between potential desires and the adopted goals also belongs to this category; knowing such information will help an agent to rationally postpone or drop those unrealistic desires.

Goal-escape information-need A goal ultimately becomes achieved, unachievable or irrelevant. This type of information is needed by an agent to drop the impossible or irrelevant commitments (goals). A goal is achievable and relevant only when certain context holds. Thus, goal-escape information-needs can typically be derived from the context of the goal under concern. If any part of the context no longer holds and this is observed by an agent, being helpful, the agent can inform this fact to the other teammates involved in the goal so that they can stop pursuing the goal at the earliest possible opportunity.

II. OBJECTIVES

The design of MALLET aims to accomplish the following three objectives.

1. MALLET ought to be a language suitable for encoding agent teamwork knowledge, especially for composing complex teamwork behaviors. We consider this requirement from three aspects: expressivity, understandability, and reusability. First, the language should be expressive enough. As we mentioned before there exist three levels of team tasks. Rather than supporting one or two levels as did in the previous attempts, we aim to provide three levels of supports in MALLET. Second, teamwork knowledge (team structure and the team processes) should be captured in a way that is easy to understand at the team level. This requires a high-level language (rather than using classical programming languages, say, JACK Teams [17]) designed specifically for capturing teamwork knowledge.

Third, reusability of knowledge is important for reducing the cost of developing and maintaining agent systems with such knowledge. Complex cognitive behaviors have been simulated using existing agent architectures (e.g., Soar [15], ACT-R [23]). The knowledge captured in these architectures can be easily reusable because the knowledge representation languages (e.g., production rules in Soar, and rules in ACT-R) were designed to model general intelligence. Adhering to the same criterion, MALLET should be designed such that the representation of *teamwork knowledge* can be easily reused.

2. The language should encourage inferred team intelligence. One particular interest of our research is to empower agents with the capability of anticipating other teammates' information needs. An agent's information needs may emerge before it performs an action or a plan, when it is required to choose the next course of actions from several alternatives, or when its joint commitments become violated as detected by other teammates. Therefore, it is desirable that the

language can support the specification of collaboration constraints, such as action preconditions and plan termination conditions.

More generally, the language should allow experimentation with different levels and forms of inferred team intelligence. For example, one might want to include the ability of agents to observe their environment (thus providing a richer basis for inference [?]), or one might want to use decision theory to improve the inference used in proactive communication [24]. Equally important, one might want to represent different levels of synchronization among team members, and allow experimentation with varying forms of individual and mutual belief among team members during execution. This latter reflects reality in human teams. Therefore, it is desired to have a language flexible enough to emulate different levels of human teamwork behavior.

3. The language should allow the specification of adaptive team structure and team process. For instance, in a dynamic environment, agents often need to adapt their team processes to environment changes (e.g., recover from failures). In addition, effective human teams often adjust their team structure by dynamically allocating tasks to members based on their roles and other factors of the environment. This sets another requirement on the language.

III. Syntax

The syntax of MALLET is given in Appendix I. A MALLET specification is composed of definitions for agents, teams, membership of a team, team goals, initial team activities, agent capabilities, roles, roles each agent can play, individual operators, team operators, plans (recipes), and inference rules.

At the top level, MALLET allows expression of knowledge about team structure in terms of membership of a team and agent-role relationship (i.e. which agent plays which role). For example, the following specification defines team *fire-fightingTeam*, which has three members playing a role of fighter or ambulance, respectively:

(team fire-fightingTeam (John Tom Sam))
(plays-role John (fighter))
(plays-role Tom (fighter))
(plays-role Sam (ambulance)).

Operators are atomic domain actions, each of which is associated with pre-conditions and effects. Individual operators are supposed to be carried out by only one agent independently,

while the performance of a team operator requires more than one agent playing different roles as required by the operator. Before doing a team action, all the involving agents should synchronize their activities and satisfy the corresponding preconditions. For example,

$$(toper\ co\_spray(?fid)\ (num\ eq\ 3)\cdots)$$

states that team operator co_spray requires two agents work on the fire $?fid$ simultaneously.

Plans are decomposable higher-level actions, built upon lower-level actions or atomic operators hierarchically. Plans play the same role as recipes in the SharedPlans theory [4]. A plan in MALLET specifies which agents (variables), under what pre-conditions, can achieve what effects by following what a process, and optionally under what conditions the execution of the plan can be terminated.

Collaboration among team members can be coded in the process component of a team plan. A MALLET process is specified in terms of plan invocation statements and composite statements using constructs such as sequential (SEQ), parallel (PAR), iterative (WHILE, FOREACH, FORALL), conditional (IF) and choice (CHOICE). An *invocation* statement is used to directly execute an action or invoke a plan. Since there is no doers associated with invocation statements, all the agents reaching such a statement will do it individually. A **DO** process is composed of a doer specification and an embedded process. An agent coming to a **DO** statement has to check if itself belongs to the doer specification. If so, the agent will proceed to perform the embedded process; otherwise the agent has to wait to be informed of the accomplishment of the embedded process.

MALLET has a powerful mechanism for dynamically binding agents with tasks. The **Agent-Bind** construct introduces flexibility to a teamwork process in the sense that agent selection can be done dynamically based on the evaluation of certain teamwork constraints (e.g., finding an agent with specific capabilities). For example,

$$(\mathbf{AgentBind}(?f)\ (constraints\ (playsRole\ ?f\ fighter)\ (closestToFire\ ?fid)))$$

states that the agent variable $?f$ needs to be instantiated with an agent who can play the role of fighter and is the closest to the fire $?fid$ ($?fid$ already has a value from the preceding context). The selected agent is then responsible for performing later steps (operators, sub-plans, or processes) associated with *?f*. An agent-bind statement becomes eligible for execution at the point when progress of the embedding plan has reached it, as opposed to being executed when the plan is entered. The scope for the binding to an agent variable extends to either the end

of the plan in which the variable appears, or the beginning of the next agent-bind statement that binds the same variable, whichever comes first. Agent-Bind statements can be anywhere in a plan, as long as agent variables are instantiated before they are used. External semantics can be associated with the constraints described in an Agent-Bind statement. For instance, a collection of constraints can be ordered increasingly in terms of their priorities. In case that not all the constraints can be satisfied, the agent allocation task is reduced to a distributed constraint optimization problem: to satisfy as many constraints as possible, and whenever necessary, the constraint with the least priority is relaxed first.

The **Joint-Do** construct provides a means for describing multiple synchronous processes to be performed by the identified agents or teams in accordance with the specified share type. A share type is either AND, OR, or XOR. For an AND share type, all of the specified subprocesses must be executed. For an XOR, exactly one subprocess must be executed, and for an OR, at least one subprocess must be executed. A Joint-Do statement is not executed until all involved team members have reached this point in their plans. Furthermore, the statement following a Joint-Do statement in the team process can not begin until all the involved team members have completed their part of the Joint-Do.

The **Choice** construct can be used to explicitly specify decision points in a complex team process. For example, suppose a fire-fighting team is assigned a task to extinguish a fire caused by an explosion at a chemical plant. After collecting enough information (e.g., whether there are noxious chemicals or dangerous facilities near the plant), the team needs to decide how to put out the fire. They have to determine one plan if there exist several options. A Choice statement is composed of a list of branches, each of which specifies a plan (a course of actions) and may be associated with preference conditions and a priority information. The preference conditions of a branch is a collection of first-order formulas; the evaluation of their conjunction determines whether the branch is workable under that context. The priority information is used to select a branch in case that the preference conditions of more than one branch are satisfiable.

As an example, Appendix II gives a MALLET profile for a fire-fighting scenario. In this example, the fire-fighting team, composed of three fire fighters and an ambulance unit, needs to extinguish emerging fires. The agents choose how to extinguish a fire depending on the fierceness level of the fire. $extinguishM1$ is used when the fierceness level is low, and $extinguishM2$ is used when the fierceness level is high. In $extinguishM2$, **AgentBind** is used to select two fighters

capable of carrying heavy tanks. The selected fighters will perform a team operator (co_spray) that is more effective than the individual operator $spray$, while the other fighters simply perform $spray$ individually.

In response to the capturing of different kinds of information needs at the language level, MALLET supports the specification of pre-conditions for primitive operators and plans, the specification of termination conditions for team plans, and the specification of preference conditions for branches of a choice point. *PreconditionList* declares a list of conditions under which an action (plan, individual or team operator) can be performed. One or more first-order predicates are allowed to define pre-condition. The conjunction of all predicates in the list is used to evaluate the pre-condition. Each condition in the precondition list could be one of three values: true, false, or unknown. An action starts only if the conjunction of the associated preconditions is true. In case that the conjunction is false or unknown, the behavior type specified in the preconditions determines how the actors behave. There are six possible behavior types: SKIP, FAIL, WAIT-SKIP, WAIT-FAIL, ACHIEVE-SKIP, and ACHIEVE-FAIL. SKIP means the plan/operator can be ignored and the execute proceeds to the next. FAIL means the actors have to terminate the execution. WAIT-SKIP means the doers can wait for a certain period and skip it if the precondition is still false. WAIT-FAIL means the doers have to terminate the execution if the precondition is still false after a certain period. ACHIEVE-SKIP means the doers can try to bring about the precondition (e.g., triggering a plan to achieve a state satisfying the preconditions), with the permission to proceed if failed. ACHIEVE-FAIL means the doers can try to bring about the precondition, but they have to terminate if the attempt failed. The behavior type is optional and is 'FAIL' by default.

PrefCondList is similar to *PreconditionList*, except that it is used to specify preference conditions. *TermConditionsList* declares the conditions under which an action (plan) can be dropped. By evaluating these conditions, agents can judge whether the goal of the action is achievable, or whether the motivation of starting the action still holds. Similar to conditions in the precondition list, termination conditions are 3-valued. Termination type has values SUCCESS-SKIP, SUCCESS-FAIL, FAILURE-SKIP, and FAILURE-FAIL. SUCCESS-SKIP means, when the termination condition is true, the doers can skip the rest of the plan and proceed to the next statement after the plan. SUCCESS-FAIL means, when the termination condition is true, the doers have to terminate execution. FAILURE-SKIP means, when the termination condition is

false, the doers can skip the rest of the plan and proceed to the next statement after the plan. FAILURE-FAIL means, when the termination condition is false, the doers have to terminate execution. Termination type is optional and is SUCCESS-SKIP by default. For instance,
(plan rescuePeople (?fid) (term-cond (people-alive ?fid 0)(out-of-control ?fid)))
states that the agents involved in executing the plan $rescuePeople$ should terminate their actions if there is no people alive in fire *?fid* and the fire becomes out of control.

When implementing agent systems, the conditions or constraints captured as such can be used to establish information flow relationships. For instance, when implementing a team-oriented agent architecture, to conform to the Joint Intentions theory, the first agent who detects the violation of a termination condition should commit to informing others of the failure.

IV. Preparation for the formal semantics

The following notational conventions are adopted. We use i, j, k, m, n as indexes; a's [1] to denote individual agents; A's to denote sets of agents; b's to denote beliefs; g's to denote goals; h's to denote intentions; ρ's to denote plan templates; p's to denote plan preconditions; q's to denote plan effects; e's to denote plan termination-conditions; β and α's to denote individual operators; Γ's to denote team operators; s and l's to denote MALLET-process statements; ψ and ϕ's to denote first-order formulas; t's to denote terms; $\vec{t}$ and $\vec{v}$ to denote vector of terms and variables. A substitution (binding) is a set of variable-term pairs $\{[x_i/t_i]\}$, where variable x_i is associated with term t_i (x_i does not occur free in t_i). We use $\theta, \delta, \eta, \mu, \tau$ to denote substitutions. $\perp$ denotes logical inconsistency, $Wffs$ denotes the set of well-formed formulas.

Given a specification of an agent team in MALLET, let $Agent$ be the set of agent names, $Ioper$ be the set of individual operators, $TOper$ be the set of team operators, $Plan$ be the set of plans, B be the initial set of beliefs (belief base), and G be the initial set of goals (goal base).

Let $P = Plan \cup Toper \cup Ioper$. We call P template plan base, which consists of all the specified operators and plans. Every invocation of a template in P is associated with a substitution: each formal parameter of the template is bound to the corresponding actual parameter. For instance, given a template

(**plan** ρ $(v_1 \cdots v_j)$

[1] We use a's to refer to a and a with a subscript or superscript. The same notation applies to the follows.

(**pre-cond** $p_1 \cdots p_k$) (**effects** $q_1 \cdots q_m$) (**term-cond** $e_1 \cdots e_n$) (**process** s)).

A plan call $(\rho\ t_1 \cdots t_j)$ will instantiate the template by binding $\theta = \{\vec{v}/\vec{t}\}$, where the evaluation of t_i may further depend on some other environment binding μ. Note that such instantiation process will substitute t_i $(1 \leq i \leq j)$ for all the occurrence of v_i in the precondition, effects, term-condition, and plan body s. The instantiation of ρ wrt. binding θ is denoted by $\rho \cdot \theta$, or $\rho\theta$ for simplicity.

Let $\mathcal{P} = \bigcup_{\rho \in P} \Xi_\rho$, where Ξ_ρ is the set of all the instantiation of ρ. $\mathcal{P}$ denotes the universe of plan instantiations. We also define some auxiliary functions. For any operator α, $pre(\alpha)$ and $post(\alpha)$ return the conjunction of the preconditions and effects of α respectively, $\lambda(\alpha)$ returns the binding if α is an instantiated operator. For team operator Γ, $|\Gamma|$ returns the minimal number of agents required for executing Γ. For any plan ρ, in addition to $pre(\rho)$, $post(\rho)$ and $\lambda(\rho)$ as defined above, $tc(\rho)$, $\chi_p(\rho)$, $\chi_t(\rho)$, and $body(\rho)$ return the conjunction of termination-conditions, the precondition type ($\in$ {**skip**, **fail**, **wait-skip**, **wait-fail**, **achieve-skip**, **achieve-fail**, ϵ}), the termination type ($\in$ {**success-skip**, **success-fail**, **failure-skip**, **failure-fail**, ϵ}), and the plan body of ρ, respectively. The precondition, effects and termination-condition components of a plan are optional. When they are not specified, $pre(\rho)$ and $post(\rho)$ return **true** and $\chi_t(\rho) = \epsilon$. For a statement s, $isPlan(s)$ returns **true** if s is of form $(\rho\ \vec{t})$ or $(\mathbf{Do}\ A\ (\rho\ \vec{t}))$ for some A, where ρ is a plan defined in P; otherwise, it returns **false**. $(\mathbf{SEQ}\ s_1\ \cdots\ s_i)$ is abbreviated as $(s_1; \cdots; s_i)$. ε is used to denote the empty MALLET process. For any statement s, $\varepsilon; s = s; \varepsilon = s$. (**wait until** ϕ) is an abbreviation of (**while** (**cond** $\neg\phi$) (**do** self skip)) [2], where $skip$ is a built-in individual operator with $pre(skip) = true$ and $post(skip) = true$ (i.e., the execution of $skip$ changes nothing).

Messages A MALLET interpreter needs to implement inter-agent communication at two levels. First of all, as a teamwork encoding language, MALLET constructs for team operations (e.g., team operators, JointDo) require that the involved agents synchronize their activities appropriately. Secondly, communication may also come from the team intelligence built in the underlying systems or from the specific requirements of domain problems. For instance, based on the constraints encoded in a MALLET process, agents can proactively anticipate others'

[2]The keyword "$self$" can be used in specifying doers of a process. An agent always evaluate $self$ as itself.

information needs and help them without being asked [8]. This kind of communication is out of the scope of this paper. We here focus on synchronization-related communication that is required to enforce the semantics of team operations. However, it is worth noting that 1) MALLET, itself, does not define communication, although various types of communication can be simulated using operators; 2) the notion of "control message" below is rather generic. It may or may not directly correspond to the actual implementations. Control messages are used as a tool to formally characterize the operational semantics of team synchronizations.

A control message is a tuple $\langle type, a_i, g_x, \rho_k, olist \rangle$, where $a_i \in Agent$, $g_x \in Wffs$, $\rho_k \in \mathcal{P} \cup \{nil\}$, and $type \in \{$ *sync, ctell, cask, unachievable* $\}$. A message of type *sync* is used by agent a_i to synchronize with a recipient with respect to the committed goal g_x and current activity ρ_k; a message of type *ctell* is used by agent a_i to tell a recipient about the status of ρ_k (i.e., starting, ending); a message of type *cask* is used by agent a_i to request a recipient to perform ρ_k; a message of type *unachievable* is used by agent a_i to inform a recipient of the unachievability of ρ_k. $olist$ is a list of extra parameters varying from message type to message type.

Control messages are wrapped and actually sent by a built-in domain-independent operator **send***(receivers, msg)*, where $pre(\mathbf{send}) = true$. We assume that the execution of **send** always succeeds. If $\langle type, a_1, \cdots \rangle$ is a control message, the effect of $send(a_2, \langle type, a_1, \cdots \rangle)$ is that agent a_1 will assert the fact $(type\ a_1\ \cdots)$ into its belief base, and agent a_2 will do the same thing when it receives the message.

Goals and Intentions A goal g is a pair $\langle \phi, A \rangle$, where $A \subseteq Agent$ is a set of agents responsible for achieving a state satisfying ϕ. When A is a singleton, g is an individual goal; otherwise, it is a team goal.

An *intention slice* is of form $(\psi, A) \leftarrow s$, where the execution of statement s by agents in A is to achieve a state satisfying ψ. An *intention* is a stack of intention slices, denoted by $[\omega_0 \backslash \cdots \backslash \omega_k]$ $(0 \leq k)$[3], where ω_i $(0 \leq i \leq k)$ are of form $(\psi_i, A_i) \leftarrow s_i$. ω_0 and ω_k are the bottom and top slice of the intention, respectively. The ultimate goal state of intention $h = [(\psi_0, A_0) \leftarrow s_0 \backslash \cdots \backslash \omega_k]$ is ψ_0, referred to by $o(h)$. The empty intention is denoted by $\top$.

[3]The form of intentions here is similar to Rao's approach [29]. Some researchers also borrow the idea of fluents to represent intentions, see [**?**] for an example.

For $h = [\omega_0 \backslash \cdots \backslash \omega_k]$, $[h \backslash \omega'] \triangleq [\omega_0 \backslash \cdots \backslash \omega_k \backslash \omega']$. If ω_i is of form $(true, A) \leftarrow \varepsilon$ $(0 \leq i \leq k)$ for some A, then $h = [\omega_0 \backslash \cdots \backslash \omega_{i-1} \backslash \omega_{i+1} \backslash \cdots \backslash \omega_k]$.

Definition 1 (configuration): A MALLET configuration of an agent is a tuple $\langle B, G, H, \theta \rangle$, where B, G, H, θ are the belief base, the goal base, the set of intentions, and the current substitution, respectively. And, (1) $B \not\models \bot$, (2) for any goal $g \in G$, $B \not\models g$, and $g \not\models \bot$ hold.

B, G, H, θ are used in defining MALLET configurations, because beliefs, goals, and intentions of an agent are dynamically changing, and a substitution is required to store the current environment bindings for free variables. Plan base P is omitted since we assume P will not be changed at run time. Dynamic planning is out of the scope of this paper. In the follows, we write h instead of H when $H = \{h\}$.

Note that there are no beliefs, goals, and intentions global to all the agents of a team. Mallet configurations are defined with respect to individual agents. Here, B, G, H, θ should all be understood as the belief base, goal base, intention set, and current substitution of an individual agent. Of course, agents in a team may overlap in their Bs, Gs and Hs. The transitions of an agent team are made up of the transitions of member agents.

Similar to [26], we give an auxiliary function to facilitate the definition of semantics of intentions.

Definition 2: Function $agls$ is defined recursively as: $agls(\top) = \{\}$,
and for any intention $h = [\omega_0 \backslash \cdots \backslash \omega_{k-1} \backslash (\psi_k, A_k) \leftarrow s_k]$ $(k \geq 0)$,
$agls(h) = \{\psi_k\} \cup agls([\omega_0 \backslash \cdots \backslash \omega_{k-1}])$.

G specifies a set of initial top-level goals, while function $agls$ returns a set of achievement goals generated at run time in pursuing some (top-level) goal in G.

V. Operational Semantics

Usually there are two options to defining semantics for an agent-oriented programming language: operational semantics and temporal semantics. For instance, temporal semantics is given to MABLE [27]; while 3APL [28], AgentSpeak(L) [29], and ConGolog [30] have operational semantics in terms of transition systems. Temporal semantics is better for property verification using existing tools, such as SPIN (a model checking tool which can check whether temporal formulas hold for the implemented systems), while operational semantics is better for implementing interpreters for the language.

We define an operational semantics for MALLET in terms of a transition system, aiming to guide the implementation of interpreters. Each transition corresponds to a single computation step which transforms the system from one configuration to another. A computation run of an agent is a finite or infinite sequence of configurations connected by transition relation $\rightarrow$. The meaning of an agent is a set of computation runs starting from the initial configuration. We assume a belief update function $BU(B, p)$, which revises the belief base B with a new fact p. The details of BU is out the scope of this paper. For convenience, we assume two domain-independent operators over B: $\mathbf{unsync}(\psi, \rho)$ and $\mathbf{untell}(\psi, s)$. Their effects are to remove all the predicates that can be unified with $sync(?a, \psi, \rho)$ and $ctell(?a, \psi, s, ?id)$, respectively, from belief base B.

A. *Semantics of beliefs, goals and intentions in MALLET*

Suppose belief base B allows *explicit negations*. For any $b(\vec{t}) \in B$, its explicit negation is denoted by $\tilde{b}(\vec{t})$. Such treatment enables the representation of *unknown*.

Definition 3: Given a MALLET configuration $M = \langle B, G, H, \theta \rangle$, for any wff ϕ, any belief or goal formula ψ, ψ', any agent a,

1) $M \models Bel(\phi)$ iff $B \models \phi$,
2) $M \models \neg Bel(\phi)$ iff $B \models \tilde{\phi}$,
3) $M \models Unknown(\phi)$ iff $B \not\models \phi$ and $B \not\models \tilde{\phi}$,
4) $M \models Goal(\phi)$ iff $\exists \langle \phi', A \rangle \in G$ such that $\phi' \models \phi$ and $B \not\models \phi$,
5) $M \models \neg Goal(\phi)$ iff $M \not\models Goal(\phi)$,
6) $M \models Goal_a(\phi)$ iff $\exists \langle \phi', A \rangle \in G$ such that $a \in A$, $\phi' \models \phi$ and $B \not\models \phi$,
7) $M \models \neg Goal_a(\phi)$ iff $M \not\models Goal_a(\phi)$,
8) $M \models \psi \wedge \psi'$ iff $M \models \psi$ and $M \models \psi'$,
9) $M \models Intend(\phi)$ iff $\phi \in \bigcup_{h \in H} agls(h)$.

Note 1 (Consistency of belief bases): As MALLET is a language for expressing team activities, there are numerous occasions on which a team or sub-team will be required to evaluate the same condition expression in a team plan. From a programming language perspective, one might expect that all agents should achieve the same value (TRUE or FALSE) for the condition. However, humans do not always achieve this, even when they are supposed to. As

MALLET is a multi-agent language that is intended to allow emulation of human teamwork, and each agent are supposed to evaluate conditions with respect to their own belief base, this raises the question of how to achieve effective collaboration if the individual belief bases are inconsistent. We leave this issue open because we believe that it is the underlying systems, not the language, that should employ mechanisms to assure consistency, if that is desired, or to handle discrepancies (e.g., through failure conditions). For example, we have handled different levels of intelligence in implicitly managing the belief base consistency (or consistency of decision making) by incorporating advanced intelligence features in various experimental versions of CAST [8], [24], which is the underlying system within which MALLET plans are executed. This approach is consistent with our objective of making MALLET a language within which experimentation with differ forms of intelligence could be carried out.

B. The Semantics of Variable Bindings

A MALLET plan may have four types of variables: parameters, variables introduced in preconditions, variables introduced in Agent-Bind statements, variables introduced in conditions of IF or Iterative statements. They have different semantics of value bindings.

Plan parameters cannot be rebound within the plan. They are passed in and retain their values throughout the plan. The variables introduced in preconditions can be used within the plan. They are bound with values when the preconditions are evaluated (i.e., upon entering the plan). Also, they cannot be rebound and their values are retained throughout the plan.

The variables introduced in Agent-Bind statements are *agent variables*. They are bound when the constraints of the corresponding Agent-Bind statements are satisfiable with respect to the belief base of the agent who is executing the Agent-Bind. The scope for the binding to an agent variable extends to either the end of the plan, or the beginning of the next Agent-Bind statement that binds the same variable, whichever comes first.

The scope of variables that are first introduced (not been previously introduced in other statements) by the condition of an IF process is limited to the IF branch (the bindings carry over to neither the ‘else’ branch nor the statements after the IF construct). The reason for disallowing use of variables introduced in the **cond** of an IF statement within the ‘else’ clause is that one can not be certain that they were bound in the evaluation of the condition. If a variable is used

within a plan before it is used in the **cond** of an IF in the scope of that plan, the variable retains its binding and is not rebound during evaluation of the **cond**. In a variable introduced in the **cond** of an IF, and then a variable of the same name is used after the IF (or in the 'else' clause – such use is discouraged) it is considered a different variable and must be bound before use. It does not retain any value from the evaluation of the **conf** of the IF beyond the scope of the IF.

The variables that are first introduced in the conditions of WHILE statements (not been previously introduced) are rebound in every iteration. We refer to these as *iteration variables*. Other variables used in the conditions of WHILE statements retain their bindings obtained before the WHILE construct. The case in which an iteration variable is bound in the first iteration and retains the binding in the rest iterations is *not* supported. Non-iteration variables that are bound in a loop retain their last binding upon exit from the loop. However, iteration variables become undefined after a WHILE construct, as the variables could have no bindings at all (i.e., the loop condition may be false the first time it is evaluated). If the same variable name appears after a WHILE, it is considered a new variable and must be appropriately bound before use.

For constructs such as FORALL and FOREACH, the rule is similar to WHILE, except that in FORALL and FOREACH all the bindings of variables are obtained the first time the condition is evaluated. As with a WHILE, iteration variables of FOREACH and FORALL become undefined after the scope of the FOREACH or FORALL.

The effects conditions of a plan may also introduce variables. This means it must be the case that a query of the form of the effects condition would return TRUE, with bindings for unbound variables. Thus, if a query is subsequently made elsewhere in the world, it will succeed unless something has negated some part of the condition in the interim.

The above semantics of variable bindings is maintained in the 'substitution' component of configurations, with new bindings overwriting previous ones.

C. Failures in MALLET

We start with the semantics of failures in MALLET. MALLET imposes the following semantics rules on execution failures:

- There are three causes of process failures:
  - The precondition is false when an agent is ready to enter a plan or execute an operator.

The execution continues or terminates depending on the type of the precondition:
skip: skip this plan/operator and execute the next one;
fail: terminate execution and propagate the failure upward;
wait-skip: check the precondition after a certain time period, if it is still false, proceed to the next plan/operator;
wait-fail: check the precondition after a certain time period, if it is still false, terminate execution and propagate the failure upward;
achieve-skip: try to bring about the precondition (e.g., triggering another plan that might make the precondition true), if failed after the attempt then skip this plan/operator and execute the next one;
achieve-fail: try to bring about the precondition, if failed after the attempt then terminate execution and propagate the failure upward;

- An agent monitors the termination condition, if any, of a plan during the execution of the plan. The execution continues or terminates depending on the type of the termination condition:
success-skip: if the termination condition is true, then skip the rest of the plan and proceed to the next statement after the plan;
success-fail: if the termination condition is true, then terminate execution and propagate the failure upward;
failure-skip: if the termination condition is false, then skip the rest of the plan and proceed to the next statement after the plan;
failure-fail: if the termination condition is false, then terminate execution and propagate the failure upward;
- When doing **agent-bind**, an agent cannot find solutions to the agent variables;

- Process failures must propagate upward until a **choice** point:
  - If any MalletProcess in a **seq** returns fail, then the entire **seq** terminates execution and fails;
  - If any branch of a **par** fails, the entire **par** terminates and fails;
  - If the body of a **while**, **foreach**, or **forall** fails, the entire iterative statement terminates execution and fails;

  - If any branch of an **if** fails, the entire **if** terminates execution and fails;
  - If any branch of a **JointDo** fails, the **JointDo** terminates and fails;
  - If the body of a plan fails, the plan invocation fails;
- Process failures are captured and processed at a **choice** point:
  - If, except for those branches the execution of which has caused process failures, the **choice** point still has other alternatives to try, then select one and the execution continues;
  - If the **choice** point has no more alternatives to try, then propagate the failure backward/upward until another **choice** point.

Note 2: Operators are considered atomic from the perspective of MALLET; they do not have termination conditions. If there is a concern that operators may not succeed, they should be embedded in a plan and the result be checked, with use of the termination condition in the case of failure.

Note 3: MALLET allows a *skip* or *fail* mode to be included with preconditions and termination conditions (supported since version V.3). One argument for allowing both modes is that continuing operations, even when some precondition is not satisfied, is what happens in real life. To the extent that we are trying to allow agent designs to respond to real-life, we need this capability. This argument is also related to the argument that we wanted to leave as much flexibility as possible in the MALLET specification so that different implementations and levels of intelligence could be experimented with.

We thus can formally define rules for failure propagation. Given the current configuration $\langle B, G, H, \theta \rangle$, a plan template $(\rho\ \vec{v})$ and an invocation $(\rho\ \vec{t})$ or $(\mathbf{Do}\ A\ (\rho\ \vec{t}))$, let $\eta = \{\vec{v}/\vec{t}\}$.

- Assert $(\mathit{failed}\ \rho\ \eta)$ into B, if $\chi_p(\rho) =$ **fail**, and $\nexists\tau \cdot B \models pre(\rho)\theta\eta\tau$;
- Assert $(\mathit{failed}\ \rho\ \eta)$ into B, if $\chi_p(\rho) =$**wait-fail**, and $\nexists\tau \cdot B \models pre(\rho)\theta\eta\tau$ for neither before nor after the specified waiting time period;
- Assert $(\mathit{failed}\ \rho\ \eta)$ into B, if $\chi_p(\rho) =$**achieve-fail**, and $\nexists\tau \cdot B \models pre(\rho)\theta\eta\tau$ for neither before nor after the 'achieve' attempt;
- Assert $(\mathit{failed}\ \rho\ \eta)$ into B, if $\chi_t(\rho) =$**success-fail**, and $\exists\tau \cdot B \models tc(\rho)\theta\eta\tau$;
- Assert $(\mathit{failed}\ \rho\ \eta)$ into B, if $\chi_t(\rho) =$**failure-fail**, and $\nexists\tau \cdot B \models tc(\rho)\theta\eta\tau$;
- Assert $(\mathit{failed}\ s\ \eta)$ into B, where $s = (\rho\ \vec{t})$ or $s = (\mathbf{Do}\ A\ (\rho\ \vec{t}))$, if $\exists\tau \cdot B \models (\mathit{failed}\ \mathit{body}(\rho)\ \tau)$;

- Assert $(failed\ s\ \theta)$ into B, where $s = (\textbf{agent-bind}\ \vec{v}\ \psi)$, if $\nexists\tau \cdot B \models \psi\theta\tau$;
- Assert $(failed\ s\ \theta)$ into B, where $s = (l_1; \cdots l_m)$, if $\exists\theta' \cdot B \models (failed\ l_1\ \theta')$;
- Assert $(failed\ s\ \theta)$ into B, where $s = (\textbf{par}\ l_1 \cdots l_m)$, if $B \models \bigvee_{i=1}^{m} \exists\theta' \cdot (failed\ l_i\ \theta')$;
- Assert $(failed\ s\ \theta)$ into B, where $s = (\textbf{forall}\ (\textbf{cond}\ \psi)\ l_1)$ or $s = (\textbf{foreach}\ (\textbf{cond}\ \psi)\ l_1)$, if $B \models \bigvee_{\tau \in \{\eta : B \models \psi\eta\}} \exists\theta' \cdot (failed\ l_1\tau\ \theta')$;
- Assert $(failed\ s\ \theta)$ into B, where $s = (\textbf{while}\ (\textbf{cond}\ \psi)\ l_1)$, if $\exists\theta' \cdot B \models (failed\ l_1\ \theta')$;
- Assert $(failed\ s\ \theta)$ into B, where $s = (\textbf{if}\ (\textbf{cond}\ \psi)\ l_1\ l_2)$, if $\exists\theta' \cdot B \models (failed\ l_1\ \theta') \vee (failed\ l_2\ \theta')$;
- Assert $(failed\ s\ \theta)$ into B, where $s = (\textbf{JointDo}\ \mathbf{X}\ (A_1\ l_1) \cdots (A_m\ l_m))\ (\mathbf{X} \in \{\textbf{AND}, \textbf{OR}, \textbf{XOR}\})$, if $B \models \bigvee_{i=1}^{m} \exists\theta' \cdot (failed\ l_i\ \theta')$;
- Assert $(failed\ s\ \theta)$ into B, where $s = (\textbf{choice}\ l_1 \cdots l_m)$, if $B \models \bigwedge_{i=1}^{m} \exists\theta' \cdot (failed\ l_i\ \theta')$.

Note that conjunction rather than disjunction is used in the rule about **choice**. This is because the semantics of choice allows re-try upon failures: a **choice** statement fails only when all the branches have failed.

The semantics of failure is defined in terms of $failed$.

Definition 4 (semantics of failure): Let s be any Mallet statement.
$\langle B, G, H, \theta \rangle \models failed(s)$ iff $\exists\theta' \cdot B \models (failed\ s\ \theta')$.

D. Transition system

In the following transition rules, we only give the minimal semantics, allowing different levels of underlying intelligence to achieve different behaviors emulating different levels of human teamwork behavior.

We use $\mathbf{SUCCEED}$ to denote the terminal configuration where the execution terminates successfully (i.e., all the specified goals and generated intentions are fulfilled); use $\mathbf{STOP}$ to denote the terminal configuration where the execution terminates abnormally—all the remaining goals are unachievable. In particular, we use $\mathbf{STOP}(h)$ to denote the execution of intention h terminates abnormally.

Definition 5: Let $h = [h' \backslash (\psi_k, A_k) \leftarrow l_1; l_2]$. UC is defined recursively:
$UC(\top) = \top$,
$UC(h) = h$, if l_1 is of form $(\textbf{choice}\ s_1 \cdots s_m)$;
$UC(h) = UC(h')$, if l_1 is not of form $(\textbf{choice}\ s_1 \cdots s_m)$.

Function $UC(h)$ returns h', where h' is h with all the top intention slices popped until the first choice point is found.

Definition 6 (Backtracking upon failure): Let $h = [h' \backslash (\psi_k, A_k) \leftarrow s \backslash \cdots]$,

$$\frac{\langle B, G, h, \theta\rangle \models \mathit{failed}(s), UC(h) \neq \top}{\langle B, G, h, \theta\rangle \rightarrow \langle B, G, UC(h), \theta\rangle}, \quad \textbf{(F1)}$$

$$\frac{\langle B, G, h, \theta\rangle \models \mathit{failed}(s), UC(h) = \top}{\langle B, G, h, \theta\rangle \rightarrow \mathbf{STOP}(h)}. \quad \textbf{(F2)}$$

In Definition 6, **F1** is a transition rule for backtracking upon process failure. Rule **(F2)** states that the execution of an intention stops if there is no choice point backward.

Definition 7 (Goal selection):

$$\exists g = \langle \psi, A\rangle \in G,\ \exists(\rho\ \vec{v}) \in P,\ \mathit{self} \in A,$$

$$\frac{\exists \tau, (\theta\tau\ \mathit{has\ bindings\ for}\ \vec{v}), B \models \mathit{pre}(\rho)\theta\tau, \mathit{and}\ \mathit{post}(\rho)\theta\tau \models \psi}{\langle B, G, \emptyset, \theta\rangle \rightarrow \langle B, G \setminus \{g\}, \{[(\psi, A) \leftarrow (\mathbf{Do}\ A\ (\rho\ \vec{v})\theta\tau)]\}, \theta\tau\rangle}, \quad \textbf{(G1)}$$

$$\frac{\forall g = \langle \psi, A\rangle \in G, \forall(\rho\ \vec{v}) \in P \quad \nexists \tau \cdot \mathit{post}(\rho)\theta\tau \models \psi}{\langle B, G, \emptyset, \theta\rangle \rightarrow \mathbf{STOP}}, \quad \textbf{(G2)}$$

$$\frac{}{\langle B, \emptyset, \emptyset, \theta\rangle \rightarrow \mathbf{SUCCEED}}. \quad \textbf{(G3)}$$

In Definition 7, Rule **G1** states that when the intention set is empty, the agent will choose one goal from its goal set and select an appropriate plan, if there exists such a plan, to achieve that goal. Rule **G2** states that an agent will stop running if there is no plan can be used to pursue any goal in G. Rule **G3** states that an agent terminates successfully if all the goals and intentions have been achieved. **G1** is the only rule introducing new intentions. It indicates that an agent can only have one intention in focus (it cannot commit to another intention until the current one has already been achieved or dropped). **G1** can be modified to allow intention shifting (i.e., pursue multiple top-level goals simultaneously).

Definition 8 (End of intention/intention slice): Let

$h_1 = [\cdots \backslash \omega_{k-1} \backslash (\psi_k, A_k) \leftarrow \varepsilon]$,

$h_2 = [(\psi_0, A_0) \leftarrow s \backslash \cdots]$,

$$\frac{B \not\models \psi_k\theta,\ UC(h_1) \neq \top}{\langle B, G, h_1, \theta\rangle \rightarrow \langle B, G,\ UC(h_1), \theta\rangle}, \quad \textbf{(EI1)}$$

$$\frac{B \not\models \psi_k\theta,\ UC(h_1) = \top}{\langle B, G, h_1, \theta\rangle \rightarrow \mathbf{STOP}(h_1)}, \quad \textbf{(EI2)}$$

$$\frac{B \models \psi_k\theta}{\langle B, G, h_1, \theta\rangle \rightarrow \langle B, G, [\cdots \backslash \omega_{k-1}], \theta\rangle}, \quad \textbf{(EI3)}$$

$$\frac{h_2 \in H, B \models \psi_0\theta}{\langle B, G, H, \theta\rangle \rightarrow \langle B, G, H \setminus \{h_2\}, \theta\rangle}. \quad \textbf{(EI4)}$$

In Definition 8, **EI1** and **EI2** are the counterparts of rules **F1** and **F2**, respectively. According to Rule **P3** in Definition 15, the achievement goal ψ_k comes from the effects condition of some plan. The effects condition associated with a plan represents an obligation that the plan must achieve. Normally, ψ_k can be achieved unless the execution of the plan body failed. But this is not always the case (e.g., an agent simply had made a wrong choice). It is thus useful to verify that a plan has, in fact, achieved the effects condition, although this is not a requirement of MALLET. In the definition, when the execution of the top intention slice is done (the body becomes ε), the corresponding achievement goal ψ_k will be checked. If ψ_k is false, the execution backtracks to the latest choice point (**EI1**) or stops (**EI2**). If ψ_k is true, then the top intention slice is popped and the execution proceeds (**EI3**). Rule **EI4** states that at any stage if the ultimate goal ψ_0 of an intention becomes true, then drop this already fulfilled intention.

Goals in G are declarative abstract goals while intention set H including all the intermediate subgoals. Definition 7 and Definition 8 give rules for adopting and dropping goals, respectively. Later we will give other rules that are relevant to goal adoption and termination (e.g. propagation of failure in plan execution). Birna van Riemsdijk, et al. [**?**] analyzed several motivations and mechanisms for dropping and adopting declarative goals. In their terminology, MALLET supports goals in both procedural and declarative ways, and employs the landmark view of subgoals.

As we have explained earlier, the **choice** construct is used to specify explicit choice points in a complex team process, and it is a language-level mechanism for handling process failures. For example, suppose a fire-fighting team is assigned to extinguish a fire caused by an explosion at a chemical plant. After collecting enough information (e.g., there are toxic materials in the plant, there are facilities endangered, etc.), the team needs to decide how to put out the fire. They have to select one plan if there exist several options. And they have to resort to another

option if one is found to be unworkable.

In syntax, the **choice** construct is composed of a list of branches, each of which specifies a plan (a course of actions) and may be associated with preference condition and a priority information. The preference condition of a branch is a collection of first-order formulas; the evaluation of their conjunction determines whether the branch can be selected under that context. The priority information is considered when the preference conditions of more than one branch are satisfiable.

Given a configuration $\langle B, G, H, \theta\rangle$ and a statement (**choice** $Br_1\ Br_2 \cdots Br_m$) where $Br_i = (pref_i\ pro_i\ (\mathbf{DO}\ A_i\ (\rho_i\ t_i)))$, let $BR = \{Br_i | 1 \leq i \leq m\}$, $BR_- \subseteq BR$ be the set of branches in BR which have already been considered but failed. We assume that B can track the changes of BR_-. Let $BR^+ = \{Br_k |\ \exists \tau \cdot B \models pref_k \cdot \theta\tau, 1 \leq k \leq m\} \setminus BR_-$, which is the set of branches that have not been considered and the associated preference conditions can be satisfied by B. In addition, let $BR^{\oplus}$ be the subset of BR^+ such that all the branches in $BR^{\oplus}$ have the maximal priority value among those in BR^+, and $ram(BR^{\oplus})$ can randomly select and return one branch from $BR^{\oplus}$.

Definition 9 (Choice construct): Let

$h = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\mathbf{choice}\ Br_1\ Br_2 \cdots Br_m); s]$,

$h_1 = [h \backslash (true, A_k) \leftarrow (\mathbf{DO}\ A_i\ (\rho_i\ t_i)); \mathbf{cend}]$,

$h_2 = [h \backslash (true, A_k) \leftarrow \mathbf{cend}]$,

$$\frac{ram(BR^{\oplus}) = Br_i, B' = BU(B, BR_-.add(Br_i))}{\langle B, G, h, \theta\rangle \rightarrow \langle B', G, h_1, \theta\rangle}, \quad \textbf{(C1)}$$

$$\frac{self \in A_i, \langle B, G, h_2, \theta\rangle \not\models failed(\rho_i), B' = BU(B, post(\rho_i)\theta)}{\langle B, G, h_2, \theta\rangle \rightarrow \langle B', G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s], \theta\rangle}. \quad \textbf{(C2)}$$

In Definition 9, Rule **C1** applies when there exists a workable branch. The intention h is appended with a new slice ended with $\mathbf{cend}$, which marks explicitly the scope of the choice point. An agent has to wait (e.g., until more information becomes available) if there is no workable branch. Rule **C2** states that when an agent comes to the statement $\mathbf{cend}$ and the execution of ρ_i is successful, it proceeds to the next statement following the choice point. Rule **C3** states that if $failed(\rho_i)$ is true, the execution returns to the choice point to try another branch.

Note 4: First, when a selected branch has failed, according to Rule **F1** the execution backtracks to this choice point (i.e., the intention of the current configuration becomes h again). When all the branches $Br_i (1 \leq i \leq m)$ have failed (i.e., $failed(\mathbf{choice}\ Br_1\ Br_2 \cdots Br_m)$ holds), again by Rule **F1** the execution backtracks to the next choice point, if there is one.

Second, an implementation can enforce the agents in a group to synchronize with others when backtracking to a preceding choice point, although this is not required by MALLET, which, as a generic language, allows experimentation with different levels and forms of team intelligence. By explicitly marking the scope of choice points, synchronization can be enforced, if necessary, when agents reaching **cend**.

Definition 10 (Agent selection): Let intention
$h = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{agent-bind} \ \ \vec{v} \ \phi); s]$,

$$\frac{\exists \tau \cdot B \models \phi\theta\tau}{\langle B, G, h, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s], \theta\tau\rangle}. \quad \textbf{(B1)}$$

The successful execution of an agent-bind statement is to compose the substitution obtained from evaluating the constraint ϕ with θ (Rule **B1**). The execution fails if there is no solution to the constraints. Since each agent has an individual belief base, one complication can arise here if the individual agents in A_k reach a different choice for the agents to bind to the agent variables. Consequences can involve vary from two different agents performing an operation that only one was supposed to do, to some agents successfully determining a binding while others fail to do so. Different strategies can be adopted when an interpreter of MALLET is implemented. For instance, in case there is a leader in a team, one solution is to delegate the binding task to the leader, who informs the results to other teammates once it finishes. If so, **B1** has to be adapted accordingly.

Note 5: Given any configuration $\langle B, G, H, \theta\rangle$, for any instantiated plan ρ, variables in $body(\rho)$ are all bounded either by some binding τ where $B \models pre(p)\theta\tau$, or by some preceding agent-bind statement in $body(\rho)$.

Definition 11 (Sequential execution): Let intention
$h = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow l_1; \cdots; l_m]$,

$$\frac{\langle B, \emptyset, [(true, A_k) \leftarrow l_1], \theta\rangle \rightarrow \langle B', \emptyset, [(true, A_k) \leftarrow \varepsilon], \theta'\rangle}{\langle B, G, h, \theta\rangle \rightarrow \langle B', G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow l_2; \cdots; l_m], \theta'\rangle}. \quad \textbf{(SE)}$$

seq is a basic construct for composing complex processes. As shown in Definition 11, if the execution of l_1 can transform B and θ into B' and θ' respectively, the rest will be executed in the context settled by the execution of l_1.

Definition 12 (Individual operator execution): Let intention
$h = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (Do \ a \ (\alpha \ \vec{t})); s]$,

$h_2 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\alpha\ \vec{t}); s]$, where $(\alpha\ \vec{v}) \in Ioper$, $\eta = \{\vec{v}/\vec{t}\}$,

$$\frac{self = a, \exists \tau, B \models pre(\alpha)\theta\eta\tau, B' = BU(B, post(\alpha)\theta\eta\tau)}{\langle B, G, h, \theta\rangle \rightarrow \langle B', G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow l; s], \theta\rangle}, \quad \textbf{(I1)}$$

$$\frac{self \neq a}{\langle B, G, h, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow l_2; s], \theta\rangle}, \quad \textbf{(I2)}$$

$$\frac{self = a, \nexists \tau \cdot B \models pre(\alpha)\theta\eta\tau, \chi_p(\rho) = \mathbf{X}}{\langle B, G, h, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s'; s], \theta\rangle}, \quad \textbf{(I3)}$$

$$\frac{\exists \tau, B \models pre(\alpha)\theta\eta\tau, B' = BU(B, post(\alpha)\theta\eta\tau)}{\langle B, G, h_2, \theta\rangle \rightarrow \langle B', G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s]; \theta\rangle}, \quad \textbf{(I4)}$$

$$\frac{\nexists \tau \cdot B \models pre(\alpha)\theta\eta\tau, \chi_p(\rho) = \mathbf{X}}{\langle B, G, h_2, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s''; s], \theta\rangle}. \quad \textbf{(I5)}$$

where l and l_2 are points for team synchronization, if needed; s' and s'' are points for responding to different precondition types when the precondition is false.

In Definition 12, Rule **I1** states that if an agent is the assigned doer a, and the precondition of α is satisfiable wrt. the agent's belief base, then the execution of the individual operator is to update the belief base with the postcondition of the operator. Rule **I2** states that the agents other than the doer a can either synchronize or proceeds, depending on the actual implementation of MALLET interpreters. In Rule **I3**, s' can be replaced by different statements, depending on the actual precondition types. Rules **I4** and **I5** are similar to **I1** and **I3** except that the intention is of form h_2, which by default all the individual agents in A_k are the doers of α.

Note 6: The statements l, l_2, s', and s'' are left open for flexibility so that alternate interpretations of agent interaction semantics can be implemented. For instance, when l and l_2 are replaced by ε, each agent in A_k can just do their own jobs. Alternatively, if we let $l = (\mathbf{Do}\ self\ (\mathbf{send}\ A_k \setminus \{self\}, \langle ctell, self, \psi_0, \alpha\rangle))$,
$l_2 = (\mathbf{wait\ until}\ ctell(a, \psi_0, \alpha) \in B)$, then the team has to synchronize before proceeding next. Precondition failures have already been covered by Rules **F1** and **F2**. Rules **I3** and **I5** apply when the precondition is false and the precondition type is of 'skip' mode. For instance, if **X** is **skip**, then s' and s'' can be ε or statements for synchronization, depending to the agent interaction semantics as explained above. If **X** is **wait-skip**, it is feasible to let
$s' = (\mathbf{wait\ until}\ \exists \tau \cdot B \models pre(\alpha)\theta\eta\tau); (\mathbf{Do}\ self\ (\alpha\ \vec{t}))$, and
$s'' = (\mathbf{wait\ until}\ \exists \tau \cdot B \models pre(\alpha)\theta\eta\tau); (\alpha\ \vec{t})$.

To execute a team operator, all the involved agents need to synchronize. Let $Y(\psi, \Gamma) =$

$\{a'|sync(a', \psi, \Gamma) \in B\}$, which is a set of agent names from whom, according to the current agent's beliefs, it has received a synchronization message wrt. ψ and Γ.

Definition 13 (Team operator execution): Let intention

$h = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (Do\ A\ (\Gamma\ \vec{t})); s]$, where $(\Gamma\ \vec{v}) \in Toper$, $\eta = \{\vec{v}/\vec{t}\}$,

$$\frac{self \in A, \exists \tau \cdot B \models pre(\Gamma)\theta\eta\tau, sync(self, \psi_0, \Gamma) \notin B}{\langle B, G, h, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^1; s], \theta\rangle}, \quad \textbf{(T1)}$$

$$\frac{self \in A, \exists \tau \cdot B \models pre(\Gamma)\theta\eta\tau, sync(self, \psi_0, \Gamma) \in B, |Y(\psi_0, \Gamma)| < |\Gamma|}{\langle B, G, h, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^2; s], \theta\rangle}, \quad \textbf{(T2)}$$

$$\frac{self \in A, \exists \tau, B \models pre(\Gamma)\theta\eta\tau, sync(self, \psi_0, \Gamma) \in B, |Y(\psi_0, \Gamma)| \geq |\Gamma|, B' = BU(B, post(\Gamma)\theta\eta\tau)}{\langle B, G, h, \theta\rangle \rightarrow \langle B', G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^3; s], \theta\rangle}, \quad \textbf{(T3)}$$

$$\frac{self \notin A}{\langle B, G, h, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^4; s], \theta\rangle}, \quad \textbf{(T4)}$$

$$\frac{self \in A, \nexists \tau \cdot B \models pre(\Gamma)\theta\eta\tau, \chi_p(\Gamma) = \textbf{wait-skip}}{\langle B, G, h, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^5; s], \theta\rangle}. \quad \textbf{(T5)}$$

where $s^1 = (\textbf{Do}\ self\ \textbf{send}(A, \langle sync, self, \psi_0, \Gamma\rangle)); (\textbf{Do}\ A\ (\Gamma\ \vec{t}))$,

$s^2 = (\textbf{wait until}\ (|Y(\psi_0, \Gamma)| \geq |\Gamma|)); (\textbf{Do}\ A\ (\Gamma\ \vec{t}))$,

$s^3 = (\textbf{Do}\ self\ \textbf{unsync}(\psi_0, \Gamma)); (\textbf{Do}\ self\ \textbf{send}(A_k \setminus A, \langle ctell, self, \psi_0, \Gamma\rangle))$,

$s^4 = (\textbf{wait until}\ \forall a \in A \cdot ctell(a, \psi_0, \Gamma) \in B)$,

$s^5 = (\textbf{wait until}\ \exists \tau \cdot B \models pre(\Gamma)\theta\eta\tau); (\textbf{Do}\ A\ (\Gamma\ \vec{t}))$.

In Definition 13, Rule **T1** states that if an agent itself is one of the assigned doers, the precondition of the team operator holds, and the agent has not synchronized with other agents in A, then it will first send out synchronization messages before executing Γ. Rule **T2** states that an agent has already synchronized with others, but has not received enough synchronization messages from others, then it continues waiting. Rule **T3** states that the execution of Γ will update B with the effects of the team operator, and before proceeding, the agent has to retract the sync messages regarding Γ (to ensure proper agent behavior in case that Γ needs to be re-executed later) and inform the agents not in A of the accomplishment of Γ. Rule **T4** deals with the case when an agent belongs to $A_k \setminus A$—the agent has to wait until being informed of the accomplishment of Γ. Rule **T5** applies when the preconditions of Γ does not hold. Variants of **T5** can be given when $\chi_p(\Gamma)$ is **skip** or **achieve-skip**.

Note 7: As we explained earlier, here we use control messages to formally characterize the

semantics of team synchronizations. Different implementations may employ different strategies to achieve synchronization.

Note 8: Usually in the use of transition systems (as in concurrency semantics) the aspect of 'waiting' is modeled implicitly by the fact that if the proper conditions are not met the rule cannot be applied so that the transition must wait to take place until the condition becomes true. In this paper, there are a number of places where 'waiting' is included in the transitions explicitly. It is true that in some places implicit modeling of waiting can be used (say, the rule T2), but not all the 'wait' can be removed without sacrificing the semantics (say, the rule T4). We use explicit modeling of waiting mainly for two reasons. First, agents in a team typically need to synchronize with other team members while waiting. For example, the doers of a team operator need to synchronize with each other both before and after the execution. Here, the agents are not passively waiting, but waiting for a certain number of incoming messages. Second, 'wait' in the rules provides a hook for further extensions. For instance, currently the wait semantics states that an agent has to wait until the precondition of an action to be executed is satisfied. We can ascribe a "proactive" semantics to the language such that the doer of an action will proactive bring about a state that can make the precondition true or seek help from other teammates.

The semantics of **JointDo** is a little complicated. A joint-do statement implies agent synchronization both at the beginning and at the end of its execution. Its semantics is given in terms of basic constructs.

Definition 14 (Joint-Do): Let intentions

$h_1 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{JointDo AND } (A'_1\ l_1) \cdots (A'_n\ l_n)); s]$,

$h_2 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{JointDo OR } (A'_1\ l_1) \cdots (A'_n\ l_n)); s]$,

$h_3 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{JointDo XOR } (A'_1\ l_1) \cdots (A'_n\ l_n)); s]$,

$$\frac{\bigcap_{j=1}^{n} A'_j = \emptyset, self \in A'_i}{\langle B, G, h_1, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^1; s], \theta\rangle}, \quad \textbf{(J1)}$$

$$\frac{\bigcap_{j=1}^{n} A'_j = \emptyset, self \in A'_i}{\langle B, G, h_2, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^0; s^{21}; s^{22}; s^0; s], \theta\rangle}, \quad \textbf{(J2)}$$

$$\frac{self \in A'_i, isSelected(A'_i)}{\langle B, G, h_3, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^1; s], \theta\rangle}, \quad \textbf{(J3)}$$

$$\frac{self \in A'_i, \neg isSelected(A'_i)}{\langle B, G, h_3, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^0; s^0; s], \theta\rangle}, \quad \textbf{(J4)}$$

$$\frac{\forall A \in \{A'_1, \cdots, A'_n\} \cdot isSelected(A'_i) \notin B}{\langle B, G, h_3, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^4; s], \theta\rangle}, \quad \textbf{(J5)}$$

where $s^0 = (\textbf{Do } self\ (\textbf{send } \bigcup_{j=1}^{n} A'_j, \langle sync, self, \psi_0, nil\rangle))$;
$(\textbf{wait until } (\forall a \in \bigcup_{j=1}^{n} A'_j \cdot sync(a, \psi_0, nil) \in B)); (\textbf{Do } self\ (\textbf{unsync } \psi_0, nil))$;
$s^1 = s^0; (\textbf{Do } A'_i\ l_i); s^0$,
$s^{21} = (\textbf{If}(\textbf{cond } \nexists l_x, a \cdot ctell(a, \psi_0, l_x, 0) \in B)$
$(s^3; (\textbf{Do } A'_i\ l_i); (\textbf{Do } self\ (\textbf{send } \bigcup_{j=1, j\neq i}^{n} A'_j, \langle ctell, self, \psi_0, l_i, 1\rangle))\ )\ )$,
$s^3 = (\textbf{If } (\textbf{cond } \nexists a \cdot cask(a, \psi_0, l_i) \in B)$
$(\ (\textbf{Do } self\ (\textbf{send } \bigcup_{j=1, j\neq i}^{n} A'_j, \langle ctell, self, \psi_0, l_i, 0\rangle))$;
$(\textbf{Do } self\ (\textbf{send } A'_i \setminus \{self\}, \langle cask, self, \psi_0, l_i\rangle))\ )\ )$,
$s^{22} = (\textbf{while}(\textbf{cond } \exists \phi_x, a \cdot ctell(a, \psi_0, l_x, 0) \in B)$
$(\textbf{wait until } \forall b \in A'_x \cdot ctell(b, \psi_0, l_x, 1) \in B); (\textbf{Do } (\textbf{untell } \psi_0, l_x))\ )$,
$s^4 = \textbf{selectBranch}(A'_1, \cdots, A'_n); (\textbf{joint-do XOR } (A'_1\ l_1) \cdots (A'_n\ l_n))$.

We first describe the meanings of the control messages used in Definition 14. Suppose a' is the agent under our concern. $ctell(a, \psi_0, l_x, 0) \in B$ means a' believes it has received a control message (of type $ctell$) from agent a saying that activity l_x towards achieving ψ_0 has started. $ctell(a, \psi_0, l_x, 1) \in B$ means a' believes it has received a control message (of type $ctell$) from agent a saying that activity l_x towards achieving ψ_0 has finished. $cask(a, \psi_0, l_x) \in B$ means a' believes it has been asked by agent a to perform activity l_x towards achieving ψ_0.

In Definition 14, Rule **J1** defines semantics for joint-do with share type "AND". It states that before and after an agent performs its task l_i, it needs to synchronize with the other teammates involved in the **joint-do** statement. Statement s^0 is used for this purpose: after sending a synchronization message to others, an agent needs to wait until it has received a similar message from each of its teammates. The last step of s^0 is to retract all those synchronization messages so

that the agent can synchronize with other teammates without being affected by previous control messages.

A joint-do statement with share type "OR" requires that at least one sub-process has to be executed. In Rule **J2**, the joint-do statement is replaced by $s^0; s^{21}; s^{22}; s^0$. Statement s^{21} states that if an agent has not received any message regarding the start of some sub-statement l_x (i.e., this agent itself is the first ready to execute the joint-do statement), the agent will sequentially do (a) s^3: if among A'_i this agent is the first ready to execute l_i, then tell all other agents not in A'_i regarding the start of l_i (i.e., $\langle ctell \cdots 0 \rangle$), and request other agents in A'_i to execute l_i; (b) agents in A'_i together execute l_i; (c) tell other agents not in A'_i the accomplishment of l_i (i.e., $\langle ctell \cdots 1 \rangle$). Statement s^{22} states in case that this agent was informed of the start of some other sub-statement l_x, it needs to wait until being informed by all the doers of l_x that l_x has been completed. This guarantees that at least one sub-process will be executed. When agents belonging to different sub-teams reach statement s^3 at the same time, more than one branch will be executed parallely. Note that in both **J1** and **J2**, it is required that an agent cannot be involved in more than one branch. Construct **Par** can be used in case that an agent needs to do more than one activity parallely.

The semantics of joint-do with share type "XOR" is based on a special team operator **selectBranch**. The first time a team of agents reach a **joint-do** with share type "XOR", they will use Rule **J5** to collaboratively select one branch from $\{A'_1, \cdots, A'_n\}$ before doing the **joint-do** statement. Some negotiation strategies can be adopted in implementing **selectBranch**; this is left to the designers of MALLET interpreters. Here we assume the effect of **selectBranch** is to allow all the agents in the team to assert $isSelected(A'_i)$, if A'_i is selected, into their respective belief base. Rule **J3** and **J4** are used when the second time the team of agents reach the joint-do statement. Rule **J3** states that if an agent belongs to the group of selected agents, it needs to synchronize with other teammates and executes the corresponding sub-statement. Rule **J4** states that if an agent does not belong to the selected group, it simply synchronizes twice (corresponding to the start and end of A'_i's execution of l_i, respectively).

Plan execution is a process of hierarchical expansion of (sub-)plans. In Definition 15 below, Rule **P1** states that if an agent is not involved, it simply waits until ρ is done. Before entering a plan, an agent first checks the corresponding pre-conditions. Rule **P2** applies when the pre-condition is false and Rule **P3** applies when the precondition is true. Rule **P2** is defined for the

case where the precondition type is **skip**. Variants of **P2** can be given for other 'skip' modes. In Rule **P3**, s^1 states that on entering a plan, a new intention slice will be appended where the agent needs to synchronize with others (when everyone is ready the synchronization messages are dropped to ensure that this plan can be properly re-entered later), then execute the plan body instantiated by the environment binding θ and local binding τ, and then tell other agents not involved in ρ about the accomplishment of ρ. Rule **P4** states that when exiting a plan (i.e., **endp** is the only statement in the body of the top intention slice), if ρ has been successfully executed, the execution proceeds to the statement after the plan call, with B being updated with the effects of ρ. Rules **P5** and **P6** complement Rules **F1** and **F2**. Rule **F1** (**F2**) applies when $failed(\textbf{Do}\ A\ (\rho\ \vec{t}))$ holds, that is, when the execution of the body of ρ fails (including the failures propagated from sub-plans of ρ). Rule **P5** (**P6**) applies when $failed(\rho)$ holds, that is, when failures emerge from the precondition or termination condition of ρ. This means, an agent needs to monitor all the termination conditions of the calling plans. The semantics of plan invocation of form $(\rho\ \vec{t})$ (i.e., no doers are explicitly specified) can be similarly defined, except that A_k will be used as the doers of ρ.

Definition 15 (Plan entering, executing and exiting): Let

$h_1 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{Do}\ A\ (\rho\ \vec{t})); s]$,

$h_1' = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{Do}\ A\ (\rho\ \vec{t}))\theta\eta\tau; s\theta]$,

$h_1'' = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{Do}\ A\ (\rho\ \vec{t}))\theta\eta\tau; s\theta \backslash (post(\rho)\theta\eta\tau, A) \leftarrow \textbf{endp}]$,

$h_1''' = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{Do}\ A\ (\rho\ \vec{t})); s \backslash \cdots]$, where $(\rho\ \vec{v}) \in Plan$, $\eta = \{\vec{v}/\vec{t}\}$,

$$\frac{self \notin A}{\langle B, G, h_1, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^2; s], \theta\rangle}, \quad \textbf{(P1)}$$

$$\frac{self \in A,\ \nexists \tau \cdot B \models pre(\rho)\theta\eta\tau, \chi_p(\rho) = \textbf{skip}}{\langle B, G, h_1, \theta\rangle \rightarrow \langle B, G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s^0; s], \theta\rangle}, \quad \textbf{(P2)}$$

$$\frac{self \in A, \exists \tau \cdot B \models pre(\rho)\theta\eta\tau}{\langle B, G, h_1, \theta\rangle \rightarrow \langle B, G, [h_1' \backslash (post(\rho)\theta\eta\tau, A) \leftarrow s^1; \textbf{endp}], \theta\eta\tau\rangle}, \quad \textbf{(P3)}$$

$$\frac{self \in A, \langle B, G, h_1'', \iota\rangle \not\models failed(\rho), B' = BU(B, post(\rho)\iota)}{\langle B, G, h_1'', \iota\rangle \rightarrow \langle B', G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s\theta], \iota\rangle}, \quad \textbf{(P4)}$$

$$\frac{self \in A_k, \langle B, G, h_1''', \theta\rangle \models failed(\rho), UC(h_1''') \neq \top}{\langle B, G, h_1''', \theta\rangle \rightarrow \langle B, G, UC(h_1'''), \theta\rangle}, \quad \textbf{(P5)}$$

$$\frac{self \in A_k, \langle B, G, h_1''', \theta\rangle \models failed(\rho), UC(h_1''') = \top}{\langle B, G, h_1''', \theta\rangle \rightarrow \textbf{STOP}(h_1''')}. \quad \textbf{(P6)}$$

where $s^0 = (\textbf{Do}\ self\ (\textbf{send}\ A_k, \langle ctell, self, \psi_0, \rho\rangle))$;

$(\textbf{wait until } (\forall a \in A \cdot ctell(a, \psi_0, \rho) \in B)),$

$s^1 = (\textbf{Do } self \ (\textbf{send } A, \langle sync, self, \psi_0, \rho\rangle)); (\textbf{wait until } (\forall a \in A \cdot sync(a, \psi_0, \rho) \in B));$
$(\textbf{Do } self \ (\textbf{unsync } \psi_0, \rho)); body(\rho)\theta\eta\tau; s^0,$

$s^2 = (\textbf{wait until } (\forall a \in A \cdot ctell(a, \psi_0, \rho) \in B)).$

The semantics of plan invocation of form $(\rho \ \vec{t})$ (i.e., there are no doers explicitly specified) can be similarly defined. In this case, the implicit doers of ρ are the same as the doers of the higher-level plan (i.e., A_k). For instance,

```
(plan p1 (?v1 ?v2)
  (process
    (seq
      (Do (?v1 ?v2) (p2 1 2))
      (p3 1 2)
    )))
```

Suppose the doers of plan $p1$ are $\{a, b, c\}$, the binding of $?v1$ and $?v2$ are a and b respectively. Then, the doers of $p2$ are $\{a, b\}$ while the doers of $p3$ are {a,b,c}. Actually, the statement $(p3\ 1\ 2)$ is equivalent to $(\textbf{Do } self \ (p3\ 1\ 2))$, which requires every doer of $p1$ to do $p3$ separately. One thing is worth noting is that, before doing $p2$, both agent a and b need to be at the point of the **Do** statement before they can execute $p2$ (e.g., through synchronization). Further, after the execution of $p2$, all three agents must be aware of the accomplishment of $p2$ before they proceed to execute $p3$. Similarly, since $p1$ is a team plan, after execution, all the doers of $p3$ still need to let others know the accomplishment or unachievability of $p3$.

The situation is not quite so simple when one considers preconditions and effects conditions. Consider what appears to be a simpler variation of the above example.

```
(plan p1 (?v1 ?v2)
  (process
    (seq
      (Do (?v1) (p2 1 2))
      (Do (?v2) (p3 1 2))
    )))
```

In this case, however, suppose that plan p2 is supposed to result in finding a wumpus (as in the Wumpus World domain) and plan p3 is supposed to kill the wumpus that was found. One might

specify an effect condition of
(effects (wumpusfound ?wid)) with plan $p2$, and a pre-condition
(pre-cond (wumpusfound ?wid))
with plan p3. That is, agent $?v1$ is to find a wumpus and $?v2$ is to kill it. Suppose that agent $?v1$ will place a fact (wumpusfound ?wid) in its knowledgebase after finding a wumpus. Since each agent evaluates the conditions with respect to their own belief base, one problem is that, unless some mechanism is supplied to transfer the wumpusfound information from $?v1$ to $?v2$, plan $p3$ may block indefinitely (if the precondition type is WAIT) waiting for its pre-condition to be satisfied.

However, as we have claimed earlier, the actual behavior of a given program depends upon not only the MALLET program, but the underlying implementation, as well. This allows different levels of intelligence to be implemented to emulate different levels of human teamwork behavior. For example, we have explored several mechanisms in CAST variants. In the simplest CAST implementation [8], $?v1$ can explicitly *send* the information relevant to the effects to the teammate $?v2$. When the proactive information delivery behavior is implemented within CAST [24], the information about wumpusfound, which is considered as one of $?v2$'s information needs, can be automatically sent to $?v2$. In another experimental version of CAST [**?**], which allows reasoning about the visibilities of agents, the information would not be sent if $?v1$ could reason that $?v2$ could also see the wumpus.

Par is a construct that takes a list of processes and executes them in any order. When each process in the list has completed successfully, the entire **par** process is said to complete successfully. If at any point one of the process fails, then the entire **par** process returns failure and gives up executing any of the statements after that point.

Intuitively, a parallel statement with k branches requires the current process (transition) to split itself into k processes. These spawned processes each will be responsible for the execution of exactly one parallel branch, and they have to be merged into one process immediately after each has completed its own responsibility. To prevent the spawned processes from committing to other tasks, their initial transitions need to be established such that (1) the intention set only has one intention with one intention slice at its top; (2) the goal base is empty (so that the transition cannot proceed further after the unique intention has been completed). Because the original goal set and intention set has to be recovered after the execution of the parallel statement, we adopt

an extra transition, which has the same components as the original transition except that $\#$ is pushed as the top intention slice, which indicates that this specific intention is *suspended*.

Definition 16 (Parallel construct): Let $h_0 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s_k; s]$,
$h = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s_k; s \backslash \#]$, where $s_k = (\textbf{par}\ l_1\ l_2 \cdots l_m)$,
$T_j = \langle B, \emptyset, [(true, A_k) \leftarrow l_j], \theta \rangle \rightarrow^* \langle B_j, \emptyset, [(true, A_k) \leftarrow \varepsilon], \theta_j \rangle \wedge B_j \not\models failed(l_j)$, and
$P_B = \langle B, G, h, \theta \rangle \parallel \langle B, \emptyset, [(true, A_k) \leftarrow l_1], \theta \rangle \parallel \cdots \parallel \langle B, \emptyset, [(true, A_k) \leftarrow l_m], \theta \rangle$,

$$\frac{\langle B, G, h_0, \theta \not\models failed(s_k)}{\langle B, G, h_0, \theta \rangle \rightarrow P_B}, \qquad \textbf{(PA1)}$$

$$\frac{\bigwedge_{j=1}^{m}(T_j), B' = BU(\bigcup_{j=1}^{m} B_j, B), \theta' = \theta_0\theta_1 \cdots \theta_m}{\langle B, G, h, \theta \rangle \rightarrow \langle B', G, [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s], \theta' \rangle}. \qquad \textbf{(PA2)}$$

In Definition 16, Rule **PA1** states that when an agent reaches a **par** statement, if the par statement is not failed, the transition is split into $k+1$ parallel transitions. Rule **PA2** states that if all the spawned processes execute successfully, the suspended intention will be reactivated with the belief base and substitution modified.

Now, it is straightforward to define semantics for composite processes. For instance, the **forall** construct is an implied **par** over the condition bindings, whereas the **foreach** is an implied **seq** over the condition bindings. The constructs **forall** and **foreach** are fairly expressive when the number of choices is unknown before runtime.

Definition 17 (Composite plans): Let
$h_1 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{if}\ (\textbf{cond}\ \phi)\ l_1\ l_2); s]$,
$h_2 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{while}\ (\textbf{cond}\ \phi)\ l); s]$,
$h_3 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{foreach}\ (\textbf{cond}\ \phi)\ l); s]$,

$h_4 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{forall } (\textbf{cond } \phi)\ l); s]$,

$$\frac{B \models \phi\theta\tau}{\langle B, G, \{h_1\}, \theta\rangle \rightarrow \langle B, G, \{[\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow l_1\tau; s]\}, \theta\rangle}, \quad \textbf{(S1)}$$

$$\frac{\nexists\tau \cdot B \models \phi\theta\tau}{\langle B, G, \{h_1\}, \theta\rangle \rightarrow \langle B, G, \{[\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow l_2; s]\}, \theta\rangle}, \quad \textbf{(S2)}$$

$$\frac{B \models \phi\theta\tau}{\langle B, G, \{h_2\}, \theta\rangle \rightarrow \langle B, G, \{[\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow l\tau; (\textbf{while } (\textbf{cond } \phi)\ l); s], \theta\rangle}, \quad \textbf{(S3)}$$

$$\frac{\nexists\tau \cdot B \models \phi\theta\tau}{\langle B, G, \{h_2\}, \theta\rangle \rightarrow \langle B, G, \{[\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s], \theta\rangle}, \quad \textbf{(S4)}$$

$$\frac{\exists\tau_1, \cdots, \tau_k \cdot \bigwedge_{j=1}^{k} B \models \phi\theta\tau_j}{\langle B, G, \{h_3\}, \theta\rangle \rightarrow \langle B, G, \{[\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow l\tau_1; \cdots; l\tau_k; s]\}, \theta\rangle}, \quad \textbf{(S5)}$$

$$\frac{\nexists\tau \cdot B \models \phi\theta\tau}{\langle B, G, \{h_3\}, \theta\rangle \rightarrow \langle B, G, \{[\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s]\}, \theta\rangle}, \quad \textbf{(S6)}$$

$$\frac{\exists\tau_1, \cdots, \tau_k \cdot \bigwedge_{j=1}^{k} B \models \phi\theta\tau_j}{\langle B, G, \{h_4\}, \theta\rangle \rightarrow \langle B, G, \{[\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{par } l\tau_1\ \cdots l\tau_k); s]\}, \theta\rangle}, \quad \textbf{(S7)}$$

$$\frac{\nexists\tau \cdot B \models \phi\theta\tau}{\langle B, G, \{h_4\}, \theta\rangle \rightarrow \langle B, G, \{[\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s]\}, \theta\rangle}, \quad \textbf{(S8)}$$

The above has defined the semantics for MALLET statements from an individual agent's perspective. In a teamwork setting, each agent can be viewed as a transition system. An agent drives towards its goal through evaluating and executing its intentions based on the agent's belief base. Collaborative teamwork behaviors are realized through the interactions among the transition systems. Particularly, to have a joint goal and shared team plans, agents in a team will know when and how to synchronize with other teammates as they work on their joint and individual intentions.

VI. A RUN OF AN EXAMPLE AGENT TEAM

We can use the operational semantics to formally reason about the behaviors of team-based agents. In this section, using the example in Appendix II we show how team-based agents execute team plans using the transition rules.

Suppose the MALLET profile in Appendix II is shared by a team of agents $\{a_0, a_1, a_2, a_3\}$, where a_0, a_1, and a_2 are firefighters and a_3 is an ambulance unit. Also, suppose the initial transitions of all the agents are the same: $c_0 = \langle B, G, \emptyset, \emptyset\rangle$, where $G = \{\langle T1\ (extinguished\ fire1)\rangle\}$,

and the fact $(\mathit{fireLevel}\ \mathit{fire1}\ \mathit{high}) \in B_{a_i}$ (B_{a_i} is the belief base of agent a_i). Below we focus on a potential sequence of transitions of agent $a1$.

1. Since $a_1 \in T1$, and the antecedents of rule G1 hold when ρ is $workOnFire$ and $\tau = \{?f/fire1\}$, c_0 can thus be transformed using rule G1 into

$$c_1 = \langle B, \emptyset, [(\psi_0, T_1) \leftarrow (\mathbf{Do}\ T1\ (workOnFire\ \mathit{fire1}))], \theta_1 \rangle, \qquad \textbf{(rule G1)}$$

where $\psi_0 = (extinguished\ fire1)$, $\theta_1 = \{?f/fire1\}$.

2. Since the top of the intention in c_1 is a plan invocation, applying rule **P3** transforms c_1 to

$$c_2 = \langle B, \emptyset, [h_0 \backslash ((extinguished\ fire1), T_1) \leftarrow s_1; s_2; s_3; body(workOnFire)\theta_1; s_4; \mathbf{endp}], \theta_1 \rangle,$$
(rule P3)

where

$$\begin{aligned}
h_0 =& [(\psi_0, T_1) \leftarrow (\mathbf{Do}\ T1\ (workOnFire\ fire1))],\\
s_1 =& (\mathbf{Do}\ self\ (\mathbf{send}\ T1, \langle sync, self, \psi_0, workOnFire\rangle)),\\
s_2 =& (\mathbf{wait\ until}\ (\forall a \in T1 \cdot sync(a, \psi_0, workOnFire) \in B)),\\
s_3 =& (\mathbf{Do}\ self\ (\mathbf{unsync}\ \psi_0, workOnFire))\\
s_4 =& (\mathbf{Do}\ self\ (\mathbf{send}\ T1, \langle ctell, self, \psi_0, workOnFire, 1\rangle)),\\
& (\mathbf{wait\ until}\ (\forall a \in T1 \cdot ctell(a, \psi_0, workOnFire, 1) \in B))
\end{aligned}$$

3. s_1 invokes the execution of individual operator **send**, which results in belief update.

$$c_3 = \langle B_3, \emptyset, [h_0 \backslash ((extinguished\ fire1), T_1) \leftarrow s_2; s_3; body(workOnFire)\theta_1; s_4; \mathbf{endp}], \theta_1 \rangle,$$
(rule I1)

where $B_3 = B \cup \{sync(a_1, \psi_0, workOnFire)\}$.

4. When not getting all the synchronization messages, agent a_1 has to do a skip action (unfolding the loop).

$$c_4 = \langle B_3, \emptyset, [h_0 \backslash ((extinguished\ fire1), T_1) \leftarrow skip; s_2; s_3; body(workOnFire)\theta_1; s_4; \mathbf{endp}], \theta_1 \rangle,$$
(rule S3)

5. Agent a_1 can proceed until after getting all the synchronization messages from teammates.

$$c_5 = \langle B_5, \emptyset, [h_0 \backslash ((extinguished\ fire1), T_1) \leftarrow s_3; body(workOnFire)\theta_1; s_4; \mathbf{endp}], \theta_1 \rangle,$$
(rule S4)

where $B_5 = B \cup \{sync(a_0, \psi_0, workOnFire) sync(a_1, \psi_0, workOnFire),$
$sync(a_2, \psi_0, workOnFire), sync(a_3, \psi_0, workOnFire)\}$.

6. The execution of s_3 is to make sure plan $workOnFire$ can be re-entered, whenever needed. After this, the belief base recovered to B.

$$c_6 = \langle B, \emptyset, [h_0 \backslash ((extinguished\ fire1), T_1) \leftarrow body(workOnFire)\theta_1; s_4; \mathbf{endp}], \theta_1 \rangle,$$

(rule I1)

$body(workOnFire)\theta_1 =$

(**choice**

$((prefcond\ (fireLevel\ fire1\ low))(Do\ extinguishM1\ fire1))$

$((prefcond\ (fireLevel\ fire1\ high))(Priority\ 5)(Do\ extinguishM2\ fire1))$

$((prefcond\ (fireLevel\ fire1\ high))(Priority\ 2)(Do\ extinguishM3\ fire1))$

).

7. Applying rule C1, c_6 becomes

$$c_7 = \langle B_6, \emptyset, [h_1 \backslash (true, T_1) \leftarrow (DO\ T1\ extinguishM2\ fire1); \mathbf{cend}], \theta_1 \rangle, \qquad \textbf{(rule C1)}$$

where $h_1 = [h_0 \backslash ((extinguished\ fire1), T_1) \leftarrow body(workOnFire)\theta_1; s_4; \mathbf{endp}]$,
$B_6 = B \cup \{BR\_ = \{(Do\ extinguishM2\ fire1)\}\}$.

8. Now, the top intention slice is a plan invocation. We omit the steps similar to steps from 2 to 5, and suppose now the transition becomes

$$c_8 = \langle B_6, \emptyset, [h_1 \backslash (true, T_1) \leftarrow body(extinguishM2)\theta_1; s_4; \mathbf{endp}; \mathbf{cend}], \theta_1 \rangle, \qquad \textbf{(rule I1)}$$

c_8 is equivalent to c_8':

$$c_8' = \langle B_6, \emptyset, [h_1 \backslash (true, T_1) \leftarrow s_5; s_6; s_4; \mathbf{endp}; \mathbf{cend}], \theta_1 \rangle,$$

where

$$s_5 = (\textbf{agent-bind}\ (?x\ ?y)\ \phi_1)),$$

$$s_6 = (\textbf{if}\ (\textbf{cond}\ \phi_2)\ l_1\ l_2), where$$

$$\phi_1 = (playsRole\ firefighter\ ?x)\ (capableOf\ heavyTank\ ?x)$$

$$(playsRole\ firefighter\ ?y)\ (capableOf\ heavyTank\ ?y)\ (notEq\ ?x\ ?y),$$

$$\phi_2 = (notEq\ self\ ?x)\ (notEq\ self\ ?y),$$

$$l_1 = (\textbf{Do}\ self\ (extinguishFire\ fire1\ 300)),$$

$$l_2 = (\text{co-act}\ ?x\ ?y\ fire1\ 500)).$$

9. The execution of agent-bind adds more bindings.

$$c_9 = \langle B_6, \emptyset, [h_1 \backslash (true, T_1) \leftarrow s_6; s_4; \textbf{endp}; \textbf{cend}], \theta_2 \rangle, \qquad \textbf{(rule B1)}$$

where $\theta_2 = \{?f/fire1, ?x/a_1, ?y/a_2\}$, because a_1 and a_2 make ϕ_1 true.

10. ϕ_2 is false for agent a_1, so c_9 transforms to

$$c_{10} = \langle B_6, \emptyset, [h_1 \backslash (true, T_1) \leftarrow (\text{co-act}\ ?x\ ?y\ fire1\ 500); s_4; \textbf{endp}; \textbf{cend}], \theta_2 \rangle, \qquad \textbf{(rule S2)}$$

11. Then agent a_1 enters plan co-act with binding $\eta = \{?a/?x, ?b/?y, ?f/fire1, ?amount/500\}$.

$$c_{11} = \langle B_6, \emptyset, [h_2 \backslash (true, T_1) \leftarrow s_7\ s_8\ s_9; s_{11}; \textbf{endp}], \theta_3 \rangle, \qquad \textbf{(rule P1)}$$

where

$$h_2 = [h_1 \backslash (true, T_1) \leftarrow (\text{co-act}\ ?x\ ?y\ fire1\ 500); s_4; \textbf{endp}; \textbf{cend}],$$

$$\theta_3 = \{?f/fire1, ?x/a_1, ?y/a_2, ?amount/500, ?a/?x, ?b/?y\},$$

$$s_7 = (\textbf{Do}\ self\ (carryWater\ 2000)),$$

$$s_8 = (\textbf{Do}\ self\ (moveTo\ fire1)),$$

$$s_9 = (\textbf{while}\ (\textbf{cond}\ (waterMoreThan\ 500)(at\ a_1\ fire1)(at\ a_2\ fire1))\ s_9),$$

$$s_{10} = (\textbf{Do}\ (a_1\ a_2)\ (\text{co-spray}\ 500)),$$

$$s_{11} = (\textbf{Do}\ self\ (\textbf{send}\ T1, \langle ctell, self, \psi_0, \text{co-act}, 1\rangle));$$

$$(\textbf{wait until}\ (\forall a \in T1 \cdot ctell(a, \psi_0, \text{co-act}, 1) \in B)).$$

12. The execution of s_7 affects the belief base.

$$c_{12} = \langle B_{12}, \emptyset, [h_2 \backslash (true, T_1) \leftarrow s_8; s_9; s_{11}; \mathbf{endp}], \theta_3 \rangle, \quad \textbf{(rule I1)}$$

where $B_{12} = B_6 \cup \{(hasWater\ 2000)\}$.

13. Then agent a_1 enters plan *moveTo* with binding $\eta = \{?lo/fire1\}$:

$$c_{13} = \langle B_{12}, \emptyset, [h_3 \backslash (true, a_1) \leftarrow s_{12}; \mathbf{endp}], \theta_4 \rangle, \quad \textbf{(rule P3)}$$

where

$$h_3 = [h_2 \backslash (true, T_1) \leftarrow s_8; s_9; s_{11}; \mathbf{endp}],$$

$$\theta_4 = \{?f/fire1, ?x/a_1, ?y/a_2, ?amount/500, ?a/?x, ?b/?y, ?lo/fire1\},$$

$$s_{12} = (\mathbf{while}\ (\mathbf{cond}\ (not\ (at\ self\ fire1)))\ s_{11}),$$

$$s_{13} = (\mathbf{Do}\ self\ (\text{stepForward}\ fire1)).$$

14. If a_1 currently is not at the location of $fire1$, the loop unfolds.

$$c_{14} = \langle B_{12}, \emptyset, [h_3 \backslash (true, a_1) \leftarrow s_{13}; s_{12}; \mathbf{endp}], \theta_4 \rangle, \quad \textbf{(rule S3)}$$

15. Suppose now, there is no route for a_1 to get to $fire1$, i.e., the termination condition of *moveTo* holds. Then according to rule P5, the transition backtracks to the latest choice point:

$$c_{15} = \langle B_{13}, \emptyset, [h_0 \backslash ((extinguished\ fire1), T_1) \leftarrow body(workOnFire)\theta_1; s_4; \mathbf{endp}], \theta_4 \rangle,$$

(rule P4)

where $B_{13} = B_{12} \cup \{(failed\ moveTo\ fire1)\}$.

16. Now, the agents can choose another branch, if possible, to achieve the goal $(extinguished\ fire1)$.

VII. CAST–An agent architecture realizing MALLET

CAST (Collaborative Agents for Simulating Teamwork) is a team-oriented agent architecture that supports teamwork using a shared mental model (SMM) among teammates [8]. The CAST kernel includes an implemented interpreter of MALLET. At compile time, CAST translates processes specified in MALLET into PrT nets (specialized Petri-Nets), which use predicate evaluation at decision points. CAST supports predicate evaluation using a knowledge base with a Java-based backward chaining reasoning engine called JARE. The main distinguishing feature of CAST is proactive team behaviors enabled by the fact that agents within a CAST architecture

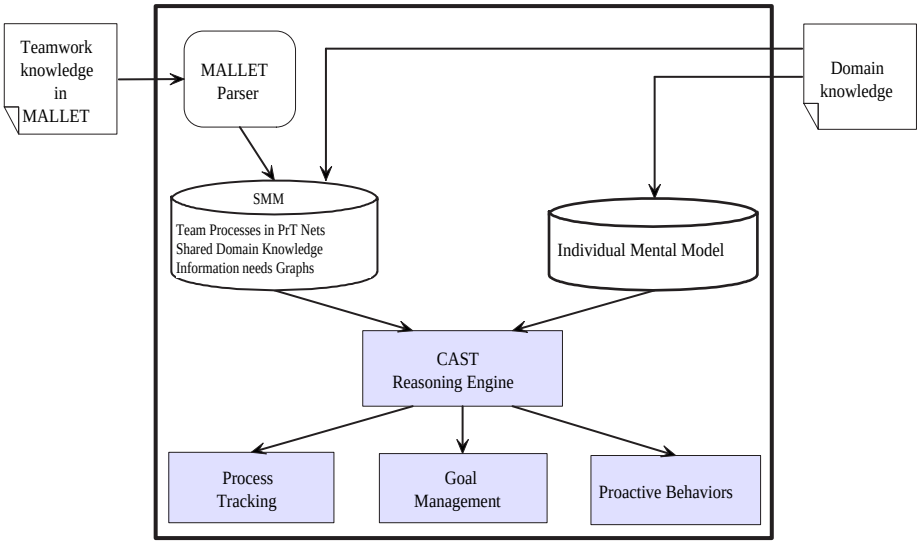

Fig. 1. The relationship between MALLET and CAST

share the same declarative specification of team structure and process as well as share explicit declaration of what each agent can observe. Therefore, every agent can reason about what other teammates are working on, what the preconditions of teammates' actions are, whether a teammate can observe the information required to evaluate a precondition, and hence what information might be potentially useful to the teammate. As such, agents can figure out what information to proactively deliver to teammates, and use a decision theoretic cost/benefit analysis for doing proactive information delivery. CAST has been used in several domains including fire-fighting, simulated battle fields [24]. Examples and practices of using MALLET can be found in [31].

Figure 1 shows the CAST architecture. A CAST agent has six components: Reasoning Engine (RE), Shared Mental Model (SMM), Individual Mental Model (IMM), Team Process Tracking (TPT), Proactive Behavior (PB), and Goal Management (GM). Based on the current states of SMM and IMM, the RE triggers appropriate algorithms in TPT, PB and GM to monitor the progress of team activities, to select goals to pursue, to anticipate others information needs and to proactively help them. The execution of these mental operations will further affect the evolution of the shared and individual mental states.

CAST supports three kinds of information-needs, which are critical for initiating help behaviors. First, CAST is implemented such that each team member commits to letting others know

its progress in the current team process. Such communication for synchronization purpose is motivated by the built-in information-needs: each agent needs to know others progress in order to maintain the SMM regarding the dynamic status of team process. The built-in information-needs provide the cohesive force [2] that binds individual CAST agents together as a team. The second kind of information-needs is those explicitly coded in a team process. CAST agents can establish partial information-flow relationships by extracting the pre-conditions, termination conditions and constraints associated with (sub-)plans in a team process. These partial relationships can be further refined at run time as the team allocates tasks. The third kind of information-needs is those explicitly requested from teammates.

While IMM stores those mental attitudes privately held by individual agents, SMM stores the knowledge and information that are shared by all team members. It has four components: team process, team structure, domain knowledge, and Information-needs Graphs. The team process component can be further split into static part and dynamic part. The static part is a collection of plans represented as PrT nets, which describe how the team is to accomplish its goals. These plans are like incomplete recipes in the SharedPlans theory, since the actors of unresolved tasks need to be determined at run time. The dynamic part is a collection of token configurations, each of which tracks the current progress of the corresponding plan. The team structure component of SMM captures those knowledge specifying roles in the team, agents in the team, and the roles each agent can play. A shared understanding about team structure enables an agent to develop a higher level abstraction about capabilities, expertise, and responsibilities of generic team members. The domain knowledge component describes domain-dependent common knowledge shared by all the team members, such as each agents observability (used to approximate nested beliefs), communication protocols, inference rules, domain expertise, etc. The Information-needs Graphs are used to maintain the dynamic information-needs relationships (i.e., make sure the information needs reflect the current status of team activities).

The TPT module is used by individual agents to interpret and manipulate a team process so that they could collaborate smoothly both when everything is progressing as planned and when something goes wrong unexpectedly. The PB module encapsulates all the proactive teamwork capabilities. One of such implemented capabilities is proactive information delivery, which depends on the anticipation of others information needs. The capability is implemented in the DIARG (Dynamic Inter-Agent Rule Generator) algorithm. Upon acquiring new information from

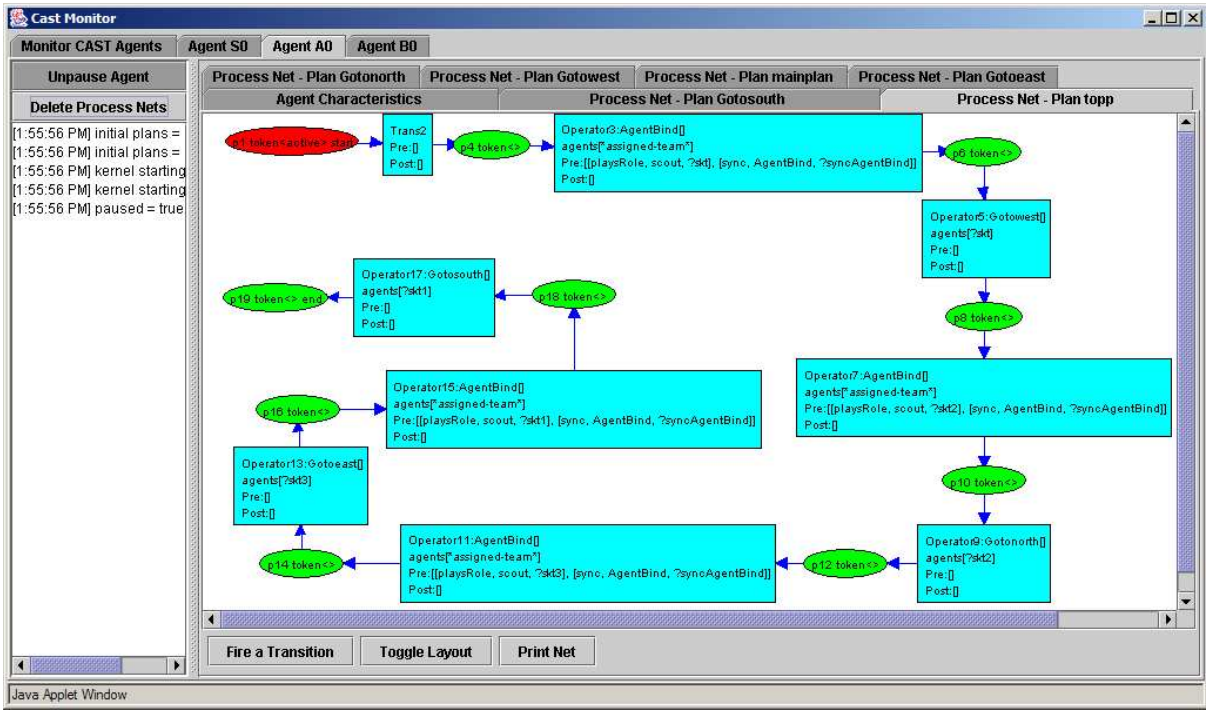

Fig. 2. The CAST Monitor

the environment, DIARG checks whether the new information matches some teammates future information-needs. If there is a match, the agent will consider sending out the new information to the corresponding needers proactively.

An agent may have multiple goals. Some are individual goals and others are team goals; some may have temporal relations while others may not. The GM module is used by a CAST agent to select a goal to pursue, or suspend the pursuit of one goal and switch to another; both are based on the agents situation assessment and cooperation requests from other agents. Once a goal is committed, GM will find a plan that can achieve the goal; the PrT net generated for the plan will become the agents work process.

Figure 2 is a screen shot of CAST monitor. CAST monitor can display the PrT nets (visual representation of MALLET plans) that a team of agents are working on. Different colors are used to indicate the progress of activities, so that a human can track the running status of a team process.

It is worth noting that MALLET is designed to be a language for encoding teamwork knowl-

edge, and CAST is just one agent architecture that realizes MALLET. It is not required that all agents in a team have to be homogeneous in that they are all implemented in the same way. Agents with different architectures can form a team and work together with CAST agents as long as their kernels conform to the semantics of MALLET and the same communication protocols.

VIII. COMPARISON

We compare MALLET with the work in JACK [17], OWL-S [32], PDDL [33], and the team-oriented programming framework [9].

Instead of providing a higher-level plan-encoding language like MALLET, JACK Teams [17] tried to extend a traditional programming language (i.e. Java) with special statements for programming team activities. In JACK Teams, a team is an individual reasoning entity that is characterized by the roles it performs and the roles it requires others to perform. To form a team is to set up the declared role obligation structure by identifying particular sub-teams capable of performing the roles to be filled.

JACK Teams has constructs particularly for specifying team-oriented behaviors. `Teamdata` is a concept that allows propagation of beliefs from teams to sub-teams and vice versa. In a sense, belief propagation in JACK is comparable to the maintenance of SMM in CAST. However, SMM in CAST is a much more general concept, which includes team plans, progress of team activities, results of task allocations, decision results of choice points, information needs graphs, etc. Agents in a team need to proactively exchange information (beliefs) to maintain the consistency of their SMM. Statements $@team\_achieve$ and $@parallel$ are used in JACK for team goal handling. $@team\_achieve$ is similar to DO statement in MALLET, except that $@team\_achieve$ is realized by sending an event to the involved sub-team, while the agents involved in a DO statement will try to perform the associated activity whenever they reach the statement along the team process. $@parallel$ allows several branches of activity in a teamplan to progress in parallel. A $@parallel$ statement can specify success condition, termination condition, how termination is notified, and whether to monitor and control the parallel execution. In semantics, $@parallel$ statement can be simulated using PAR or $CHOICE$ in MALLET. As far as failure handling is concerned, JACK Teams leverages the Java exception mechanism to throw and catch exceptions of types `TeamException`, `TeamError`, and `TeamAbort`, while in CAST $CHOICE$ points are used as places to catch failures and re-attempt the failed goals if needed, which is much more flexible

in recovery from failure at the team plan level.

OWL-S [32] is an ontology language for describing properties and capabilities of Web services. It enables users and software agents to automatically discover, invoke, compose, and monitor Web services. Similar to MALLET, OWL-S provides constructs (such as Sequence, Split, Split+Join, Choice, Unordered, If-Then-Else, Iterate, etc.) for specifying composite processes, and preconditions and effects can be associated with a process. There exist correspondences between OWL-S and MALLET. For instance, both 'Split' in OWL-S and PAR in MALLET can be used to specify components to be executed concurrently. The main difference between these two language lies in the fact that MALLET is designed for encoding team intelligence where the actors of each activity within a team process need to collaborate with each other in pursuing their joint goals, while OML-S, as an abstract framework for describing service workflows, does not consider collaboration issues from the perspective of agent teamwork.

PDDL (the Planning Domain Definition Language) [33] is a standard language for the encoding of planning domains, inspired by the well-known STRIPS formulations of planning problems. PDDL is capable of capturing a wide variety of complex behaviors using constructs such as seq, $parallel$, $choice$, $foreach$ and $forsome$. The semantics of processes in PDDL is grounded on a branching time structure. One key difference between PDDL and MALLET is that PDDL is used for guiding planning while MALLET is used for encoding the planning results. The processes defined in PDDL serve as guides for a planner to compose actions to achieve certain goals, while the processes in MALLET serve as common recipes for a team of agents to collaborate their behaviors.

The detailed comparison between MALLET and TOP is shown in Table I.

IX. Conclusion

MALLET is a language that organizes plans hierarchically in terms of different process constructs such as sequential, parallel, selective, iterative, or conditional. It can be used to represent teamwork knowledge in a way that is independent of the context in which the knowledge is used. This paper described the design objective of MALLET, defined an operational semantics for MALLET in terms of a transition system, used the transition rules to formally reason about the behaviors of an example agent team, and briefly introduced CAST–an implemented interpreter of MALLET, which uses PrT nets as the internal representation of team process.

TABLE I

A COMPARISON BETWEEN MALLET AND TOP

Items	**TOP**	**MALLET**
Expressivity	a language for specifying joint plans at team level	a language for programming team activity
	Expressions:	Control constructs:
	testing plan expression (only the specified agents can do)	IF construct (all agents reaching it can do)
sequencing	operator ';'	SEQ
parallelism	operator '&'	PAR
alternative	operator '|'	CHOICE
iterative	N/A	WHILE, FOREACH, FORALL
joint activity	N/A	JOINT-DO (AND, OR, XOR)
Plans	pre-defined	pre-defined and shared
Team structure	formed on demand based on pre-computed skill information, no run-time re-formation	pre-specified and shared, allow run-time re-formation (AGENT-BIND)
Plan body	composed of sub-goals, each raises a decision task on how to achieve that goal (dynamic plan selection)	composed of sub-processes, which frees agents from run-time plan selections.
task assignment	the acting team may outnumber the formal team, such a case will incur unnecessary communication cost	the acting team are selected such that information is exchanged only among the relevant teammates
fail handling	OR node	CHOICE, termination conditions, pre-conditions

MALLET does have several limitations, though. For instance, there is no clear semantics defined for dynamic joining or leaving a team. Also, MALLET does not specify what to do if agents do not have a plan to reach a goal. Although some of these issues can be left open to agent system designers, providing a language-level solution might be helpful in guiding the implementation of team-based agent systems. One way is to extend MALLET with certain build-in meta-plans. For instance, meta-plans, say, *resource-based-planner*, can be added so that agents could execute it to construct a plan when they need but do not have one.

APPENDIX I

THE SYNTAX OF MALLET

CompilationUnit ::=	(AgentDef | TeamDef | MemberOf | GoalDef | Start |
	CapabilityDef | RoleDef | PlaysRole | FulfilledBy | IOperDef |
	TOperDef | PlanDef | RuleDecl)*
AgentDef ::=	'(' ⟨AGENT⟩ AgentName ')'
TeamDef ::=	'(' ⟨TEAM⟩ TeamName ('(' (AgentName)+ ')')? ')'
MemberOf ::=	'(' ⟨MEMBEROF⟩ AgentName
	(TeamName | '(' (TeamName)+ ')') ')'
GoalDef ::=	'(' ⟨GOAL⟩ AgentOrTeamName (Cond)+ ')'
Start ::=	'(' ⟨START⟩ AgentOrTeamName Invocation ')'
CapabilityDef ::=	'(' ⟨CAPABILITY⟩ (AgentName | '(' (AgentName)+')')
	(Invocation | '(' (Invocation)+ ')') ')'
RoleDef ::=	'(' ⟨ROLE⟩ RoleName (Invocation | '('(Invocation)+')')')'
PlaysRole ::=	'(' ⟨PLAYSROLE⟩ AgentName '(' (RoleName)+ ')' ')'
FulfilledBy ::=	'(' ⟨FULFILLEDBY⟩ RoleName '(' (AgentName)+ ')' ')'
IOperDef ::=	'(' ⟨IOPER⟩ OperName '(' (⟨Variable⟩)* ')'
	(PreConditionList)* (EffectsList)? ')'
TOperDef ::=	'(' ⟨TOPER⟩ OperName '(' (⟨Variable⟩)* ')'
	(PreConditionList)* (EffectsList)? (NumSpec)? ')'
PlanDef ::=	'(' ⟨PLAN⟩ PlanName '(' (⟨Variable⟩)* ')'
	(PreConditionList | EffectsList | TermConditionList)*
	'(' ⟨PROCESS⟩ MalletProcess ')' ')'
RuleDecl ::=	'(' ⟨RULE⟩ (Pred)+ ')'
Cond ::=	Pred | '(' ⟨NOT⟩ Cond ')'
Pred ::=	'(' ⟨IDENTIFIER⟩ (⟨IDENTIFIER⟩ | ⟨VARIABLE⟩)* ')'
Invocation ::=	'('PlanOrOperName (⟨IDENTIFIER⟩ | ⟨VARIABLE⟩)* ')'
PreConditionList ::=	'(' ⟨PRECOND⟩ (Cond)+ (':IF-FALSE' (⟨SKIP⟩ | ⟨FAIL⟩ |
	⟨WAIT-SKIP⟩ ((⟨DIGIT⟩)+)? | ⟨WAIT-FAIL⟩ ((⟨DIGIT⟩)+)? |
	⟨ACHIEVE-SKIP⟩ | ⟨ACHIEVE-FAIL⟩))? ')'
EffectsList ::=	'(' ⟨EFFECTS⟩ (Cond)+ ')'
TermConditionList ::=	'(' ⟨TERMCOND⟩ (⟨SUCCESS-SKIP> | ⟨SUCCESS-FAIL> |
	⟨FAILURE-SKIP⟩| ⟨FAILURE-FAIL⟩)? (Cond)+')'
NumSpec ::=	'(' ⟨NUM⟩ ($'='$ | $'<'$ | $'>'$ | $'\leq'$ | $'\geq'$) (⟨DIGIT ⟩)+ ')'

PrefCondList ::=	’(’ ⟨PREFCOND⟩ (Cond)+ (’:IF-FALSE’ (⟨SKIP⟩ | ⟨FAIL⟩ |
	⟨WAIT-SKIP⟩ ((⟨DIGIT⟩)+)? | ⟨WAIT-FAIL⟩ ((⟨DIGIT⟩)+)? |
	⟨ACHIEVE-SKIP⟩ | ⟨ACHIEVE-FAIL⟩))? ’)’
Priority ::=	’(’⟨PRIORITY⟩ (⟨DIGIT⟩)+ ’) ’
ByWhom ::=	AgentOrTeamName | ⟨VARIABLE⟩ | MixedList
MixedList ::=	’(’ (⟨IDENTIFIER⟩ | ⟨VARIABLE⟩)+ ’)’
Branch ::=	’(’(PrefCondList)?(Priority)? ’(’⟨DO⟩ByWhom Invocation’)”)’
MalletProcess ::=	Invocation | ’(’⟨DO⟩ ByWhom MalletProcess ’)’
	| ’(’⟨AGENTBIND⟩ VariableList ’(’ ⟨CONSTRAINTS⟩ (Cond)+ ’)’ ’)’
	| ’(’⟨JOINTDO⟩ (⟨AND⟩ | ⟨OR⟩ | ⟨XOR⟩)?
	(’(’ ByWhom MalletProcess ’)’)+ ’)’
	| ’(’⟨SEQ⟩ (MalletProcess)+ ’)’ | ’(’⟨PAR⟩ (MalletProcess)+ ’)’
	| ’(’⟨IF⟩’(’⟨COND⟩(Cond)+’)’MalletProcess(MalletProcess)?’)’
	| ’(’⟨WHILE⟩ ’(’ ⟨COND⟩ (Cond)+ ’)’ MalletProcess ’)’
	| ’(’⟨FOREACH⟩ ’(’ ⟨COND⟩ (Cond)+’)’MalletProcess’)’
	| ’(’⟨FORALL⟩ ’(’ ⟨COND⟩ (Cond)+ ‘)’MalletProcess’)’
	| ’(’⟨CHOICE⟩ (Branch)+ ’)’

APPENDIX II
A MALLET PROFILE FOR A FIRE-FIGHTING EXAMPLE

(**team** T1 (a_1, a_2, a_3, a_4))

(**goal** T1 (extinguished fire1))

(**playsRole** firefighter a_0)

(**playsRole** firefighter a_1)

(**playsRole** firefighter a_2)

(**playsRole** ambulance a_3)

(**capableOf** heaveyTank a_1)

(**capableOf** heaveyTank a_2)

(**ioper** carryWater (?amount)

  (**effects** (hasWater ?amount)))

(**ioper** stepForward (?f)

  (**pre-cond** (closer ?w ?f) (canMove self ?w))

  (**effects** (at self ?w)))

(**ioper** spray (?amount)

```
   (pre-cond (hasWater ?w) (<= ?amount ?w) (- ?w ?amount ?left))
   (effects (hasWater ?left)))
(toper co-spray (?amount)
   (pre-cond (hasWater ?w) (<= ?amount ?w) (- ?w ?amount ?left))
   (effects (hasWater ?left))
   (num eq 2))
(plan workOnFire (?f)
   (effects (extinguished ?f))
   (process
      (choice
         ( (prefcond (fireLevel ?f low)) (Do extinguishM1 ?f))
         ( (prefcond (fireLevel ?f high)) (Priority 5) (Do extinguishM2 ?f))
         ( (prefcond (fireLevel ?f high)) (Priority 2) (Do extinguishM3 ?f))
      )))
(plan extinguishFire (?f ?amount)
   (process
      (seq
         (Do self (carryWatter ?amount))
         (Do self (moveTo ?f))
         (Do self (spray ?amount))
      )))
(plan extinguishM1 (?f)
   (pre-cond (hasResource water))
   (process
      (Do self (extinguishFire ?f 300))))
(plan extinguishM2 (?f)
   (pre-cond (hasResource water))
   (process
      (agent-bind (?x ?y)
         (constraints (playsRole firefighter ?x) (capableOf heavyTank ?x)
```

```
        (playsRole firefighter ?y) (capableOf heavyTank ?y)
        (notEq ?x ?y) ))
    (if (cond (notEq self ?x) (notEq self ?y) )
      (Do self (extinguishFire ?f 300))
      (co-act ?x ?y ?f 500)))
(plan moveTo (?lo)
  (term-cond SUCCESS-FAIL (noRouteTo ?lo))
  (process
    (while (cond (not (at self ?lo)))
      (Do self (stepForward ?lo)))))
(plan co-act (?a ?b ?f ?amount)
  (process
    (seq
      (Do self (carryWatter 2000))
      (Do self (moveTo ?f))
      (while (cond (waterMoreThan ?amount) (at ?a ?f)(at ?b ?f))
        (Do (?a ?b) (co-spray ?amount))))))
```

ACKNOWLEDGMENT

This research has been supported by AFOSR MURI grant No. F49620-00-1-0326.

REFERENCES

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

[2] P. R. Cohen, H. J. Levesque, and I. A. Smith, "On team formation," in *Contemporary Action Theory*, J. Hintikka and R. Tuomela, Eds., 1997.

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

[4] B. Grosz and S. Kraus, "Collaborative plans for complex group actions," *Artificial Intelligence*, vol. 86, pp. 269–358, 1996.

[5] M. Tambe, "Towards flexible teamwork," *Journal of AI Research*, vol. 7, pp. 83–124, 1997.

[6] C. Rich and C. Sidner, "Collagen: When agents collaborate with people," in *Proceedings of the International Conference on Autonomous Agents (Agents'97)*, 1997.

[7] J. Giampapa and K. Sycara, "Team-oriented agent coordination in the RETSINA multi-agent system," Robotics Institute, Carnegie Mellon University, Tech. Rep. CMU-RI-TR-02-34, 2002.

[8] J. Yen, J. Yin, T. Ioerger, M. Miller, D. Xu, and R. Volz, "CAST: Collaborative agents for simulating teamworks," in *Proceedings of IJCAI'2001*, 2001, pp. 1135–1142.

[9] G. Tidhar, "Team oriented programming: Preliminary report," in *Technical Report 41, AAII, Australia*, 1993.

[10] D. V. Pynadath, M. Tambe, N. Chauvat, and L. Cavedon, "Toward team-oriented programming," in *Agent Theories, Architectures, and Languages*, 1999, pp. 233–247.

[11] P. Scerri, D. V. Pynadath, N. Schurr, and A. Farinelli, "Team oriented programming and proxy agents: the next generation," in *Proc. of the 1st Inter. Workshop on Prog. MAS at AAMAS'03*, 2003.

[12] A. S. Rao, M. P. Georgeff, and E. A. Sonenberg, "Social plans: A preliminary report," in *Decentralized AI 3 –Proceedings of MAAMAW-91)*, E. Werner and Y. Demazeau, Eds. Elsevier Science B.V.: Amsterdam, Netherland, 1992, pp. 57–76.

[13] D. Kinny, M. Ljungberg, A. S. Rao, E. Sonenberg, G. Tidhar, and E. Werner, "Planned team activity," in *Artificial Social Systems (LNAI-830)*, C. Castelfranchi and E. Werner, Eds. Springer-Verlag: Heidelberg, Germany, 1992, pp. 226–256.

[14] G. Tidhar, A. Rao, and E. Sonenberg, "Guided team selection," in *Proceedings of the 2nd International Conference on Multi-agent Systems (ICMAS-96)*, 1996.

[15] J. Laird, A. Newell, and P. Rosenbloom, "SOAR: an architecture for general intelligence," *AI*, vol. 33, no. 1, pp. 1–64, 1987.

[16] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng, "Distributed intelligent agents," *IEEE Expert, Intelligent Systems and their Applications*, vol. 11, no. 6, pp. 36–45, 1996.

[17] "JACK Teams manual," in *http://www.agent-software.com/shared/demosNdocs/ JACK-Teams-Manual.pdf*, 2003.

[18] P. Scerri, L. Johnson, D. Pynadath, P. Rosenbloom, M. Si, N. Schurr, and M. Tambe, "A prototype infrastructure for distributed robot, agent, person teams," in *Proceedings of the second International Joint conference on agents and multiagent systems*, 2003.

[19] E. Davis, "Knowledge preconditions for plans," *Journal of Logic and Computation*, vol. 4, no. 5, pp. 721–766, 1994.

[20] N. Jennings, P. Faratin, M. Johnson, T. Norman, P. O'Brien, and M. Wiegand, "Agent-based business process management," *International Journal of Cooperative Information Systems*, vol. 5, no. 2&3, pp. 105–130, 1996.

[21] R. Tuomela and K. Miller, "We-intentions," *Philos. Stud*, vol. 53, no. 2&3, pp. 367–389, 1988.

[22] J. Yen, X. Fan, and V. R. A., "Towards a theory for proactive information exchange in agent teamwork," *Artificial Intelligence*, vol. (accepted), 2004.

[23] J. R. Anderson and C. ALebiere, *The Atomic Components of Thought*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1998.

[24] J. Yen, X. Fan, S. Sun, T. Hanratty, and J. Dumer, "Agents with shared mental models for enhancing team decision-makings," *Decision Support Systems, Special issue on Intelligence and Security Informatics(in press)*, 2005.

[25] M. Prasad, V. Lesser, and S. E. Lander, "Retrieval and reasoning in distributed case bases," *Journal of Visual Communication and Image Representation, Special Issue on Digital Libraries*, vol. 7, no. 1, pp. 74–87, 1996.

[26] R. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge, "Model checking agentspeak," in *Proceedings of AAMAS-2003*, 2003, pp. 409–416.

[27] M. Wooldridge, M. Fisher, M. Huget, and S. Parsons, "Model checking multiagent systems with MABLE," in *Proceedings of AAMAS-2002*, 2002.

[28] M. Dastani, B. van Riemsdijk, F. Dignum, and J.-J. C. Meyer, "A programming language for cognitive agents: Goal directed 3APL," in *Proc. of the 1st Inter. Workshop on Prog. MAS at AAMAS'03*, 2003.

[29] A. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language," in *MAAMAW'96, LNAI 1038*. Springer-Verlag: Heidelberg, Germany, 1996, pp. 42–55.

[30] G. D. Giacomo, Y. Lesperance, and H. J. Levesque, "ConGolog, a concurrent programming language based on the situation calculus," *AI*, vol. 121, no. 1-2, pp. 109–169, 2000.

[31] J. Yen and et al, "CAST manual," IST, The Pennsylvania State University, Tech. Rep., May 2004. [Online]. Available: http://faculty.ist.psu.edu/yen/Center/

[32] OWL-S, in *http://www.daml.org/services/owl-s/1.0/owl-s.html*, 2003.

[33] D. McDermott, "The formal semantics of processes in PDDL," in *Proc. ICAPS Workshop on PDDL*, 2003.