The Semantics of MALLET–An Agent Teamwork Encoding Language

Xiaocong Fan[1], John Yen[1], Michael S. Miller[2], and Richard A. Volz[2]

[1] School of Information Sciences and Technology,
The Pennsylvania State University, University Park, PA 16802
[2] Department of Computer Science,
Texas A&M University, College Station, TX 77843
{zfan, jyen}@ist.psu.edu, {mmiller, volz}@cs.tamu.edu

Abstract. MALLET is a team-oriented agent specification and programming language. In this paper, we define an operational semantics for MALLET in terms of a transition system. The semantics can be used to guide the implementation of MALLET interpreters, and to formally study the properties of team-based agents specified in MALLET.

1 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 twofold. First, 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).

Second, 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. Section 2 gives the syntax of MALLET and Section 3 gives some preparations. We give the transition semantics in Section 4, and in Section 5 introduce the CAST architecture, which has implemented a MALLET interpreter. Section 6 gives comparisons and discussions and Section 7 concludes the paper.

2 Syntax

The syntax of MALLET is given in Table 1. 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, agents playing a certain role, individual operators, team operators, plans (recipes), and inference rules.

Operators are atomic domain actions, each of which is associated with preconditions and effects. Individual operators are supposed to be carried out by

Table 1. The Abstract 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	::=	'(' (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⟩ (′=′ |′ <′ |′ >′ |′ ≤′ |′ ≥′) (⟨DIGIT ⟩)+ ')'
PrefCondList	::=	'(' ⟨PREFCOND⟩ (Cond)+ (':IF-FALSE' (⟨FAIL⟩ |
		⟨WAIT⟩ ((⟨DIGIT⟩)+)? | ⟨ACHIEVE⟩))? ')'
Priority	::=	'('⟨PRIORITY⟩ (⟨DIGIT⟩)+ ') '
ByWhom	::=	AgentOrTeamName | ⟨VARIABLE⟩ | MixedList
MixedList	::=	'(' (⟨IDENTIFIER⟩ | ⟨VARIABLE⟩)+ ')'
Branch	::=	'('(PrefCondList)?(Priority)? '('⟨DO⟩ByWhom Invocation')"')'
MalletProcess	::=	Invocation | '('⟨DO⟩ ByWhom MalletProcess ')'
		| '('⟨AGENTBIND⟩ VariableList '(' (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)+ ')'

only one agent independently, while team operators can only be invoked by more than one agent who play specific roles as required by the operators. Before doing a team action, all the involving agents should synchronize their activities and satisfy the corresponding preconditions.

Plans are decomposable higher-level actions, which are built upon lower-level atomic operators hierarchically. Plans play the same role as recipes in the SharedPlan theory. 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.

The process component of a plan plays essential role in supporting coordinations among team members. A process can be specified 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 associated doer specification, each agent coming to 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 simply does the action and moves on; otherwise the agent waits until being informed of the ending of the action. A joint-do process (JOINTDO) specifies a share type (i.e., *AND, OR, XOR*) and a list of (*ByWhom process*) pairs. A joint-do of share type "*AND*" requires all the involved agents acting simultaneously— the joint-do succeeds only after all the pairs have be executed successfully. For an "*XOR*", exactly one must be executed to avoid potential conflicts, and for an "*OR*", at least one must be executed (with no potential conflicts). An agent-bind statement is used to dynamically select agents satisfying certain constraints (e.g., finding an agent that is capable of some role or action). 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 a variable extends to either the end of the embedding plan, or the beginning of the next agent-bind statement that also binds this variable, whichever comes first.

3 Preparation

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's and α's to denote individual operators; Γ's to denote team operators; s's and l's to denote Mallet process statements; ψ's and ϕ's to denote first-order formulas; t's to denote terms; bold $\boldsymbol{t}$ and $\boldsymbol{v}$ to denote vector of terms and variables. A substitution (binding) is a set of variable-term pairs

[1] We use a's to refer to a and a with a subscript or superscript. The same applies to the description of other notations.

$\{[x_i/t_i]\}$, 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. *Wffs* is the set of well-formed formulas.

Given a team specification 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 the plan (template) 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 $(v_1 \cdots v_j)$

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

We define some auxiliary functions. For any operator α, $pre(\alpha)$ and $post(\alpha)$ return the conjunction of the preconditions and effects specified for 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 any statement s, $isPlan(s)$ returns **true** if s is of form $(\rho\ \boldsymbol{t})$ or (**Do** A $(\rho\ \boldsymbol{t})$) for some A, where ρ is a plan defined in P; otherwise, it returns **false**. (**SEQ** $s_1\ \cdots\ s_i$) is abbreviated as $(s_1; \cdots; s_i)$. ε is used to denote the empty Mallet process statement. 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 Control messages are needed in defining the operational semantics of MALLET. A control message is a tuple $\langle type, aid, gid, pid, \cdots\rangle$, where $aid \in Agent$, $gid \in Wffs$, $pid \in P \cup \{nil\}$, and $type \in \{sync, ctell, cask, unachievable\}$. A message of type *sync* is used by agent *aid* to synchronize with the recipient with respect to the committed goal *gid* and the activity *pid*; a message of type

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

ctell is used by agent *aid* to tell the recipient about the status of *pid*; a message of type *cask* is used by agent *aid* to request the recipient to perform *pid*; a message of type *unachievable* is used by agent *aid* to inform the recipient of the unachievability of *pid*.

MALLET has a built-in domain-independent operator **send***(receivers, msg)*, which is used for inter-agent communications. $pre(send) = true$. 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 $(typ\ a_1\ \cdots)$ 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 $\top$. 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 $h = [\omega_0 \backslash \cdots \backslash \omega_{i-1} \backslash \omega_{i+1} \backslash \cdots \backslash \omega_k]$. Let H denote the intention set.

Definition 1 (configuration). *A Mallet configuration is a tuple* $\langle B, G, H, \theta \rangle$, *where* B, G, H, θ *are the belief base, the goal base, the intention set, and the current substitution, respectively*[4]. *And, (1)* $B \not\models \perp$, *(2) for any goal* $g \in G$, $B \not\models g$, *and* $g \not\models \perp$ *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.

Similar to [17] 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* $h = [\omega_0 \backslash \cdots \backslash \omega_{k-1} \backslash (\psi_k, A_k) \leftarrow s_k]$ $(k \geq 0)$, $agls(h) = \{\psi_k\} \cup agls([\omega_0 \backslash \cdots \backslash \omega_{k-1}])$.

Note that goals in G are top-level goals specified initially, while function *agls* returns a set of achievement goals generated at run time in pursuing some (top-level) goal in G.

[3] The form of intentions here is similar to Rao's approach [15]. Some researchers also borrow the idea of fluents to represent intentions, see [16] for an example.

[4] There are no global beliefs, goals, and intentions. Mallet configurations are defined with respect to individual agents. The transitions of an agent team are made up of the transitions of member 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, for agents in a team, their Bs, Gs and Hs may overlap.

4 Operational Semantics

Usually there are two ways of defining semantics for an agent-oriented programming language: operational semantics and temporal semantics. For instance, temporal semantics is given to MABLE [18]; while 3APL [19] and AgentSpeak(L) [15] have operational semantics, and transition semantics is defined for ConGolog based on Situation calculus [20]. 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 B.

4.1 Semantics of Beliefs, Goals and Intentions in MALLET

We allow *explicit negation* in B, and for each $b(\boldsymbol{t}) \in B$, its explicit negation is denoted by $\tilde{b}(\boldsymbol{t})$. 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. $M \models \neg Goal(\phi)$ *iff* $M \not\models Goal(\phi)$, $M \models \neg Goal_a(\phi)$ *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* $\phi \in \bigcup_{h \in H} agls(h)$.

4.2 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 1. 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 2. 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 $(\rho\ \boldsymbol{v})$ and an invocation $(\rho\ \boldsymbol{t})$ or $(\mathbf{Do}\ A\ (\rho\ \boldsymbol{t}))$, let $\eta = \{\boldsymbol{v}/\boldsymbol{t}\}$.

– Assert $(\mathit{failed}\ \rho\ \eta)$ into B, if $\chi_p(\rho) = \mathbf{fail}$, and $\nexists \tau \cdot B \models \mathit{pre}(\rho)\theta\eta\tau$;
– Assert $(\mathit{failed}\ \rho\ \eta)$ into B, if $\chi_p(\rho) = \mathbf{wait\text{-}fail}$, and $\nexists \tau \cdot B \models \mathit{pre}(\rho)\theta\eta\tau$ for neither before nor after the specified waiting time period;
– Assert $(\mathit{failed}\ \rho\ \eta)$ into B, if $\chi_p(\rho) = \mathbf{achieve\text{-}fail}$, and $\nexists \tau \cdot B \models \mathit{pre}(\rho)\theta\eta\tau$ for neither before nor after the 'achieve' attempt;
– Assert $(\mathit{failed}\ \rho\ \eta)$ into B, if $\chi_t(\rho) = \mathbf{success\text{-}fail}$, and $\exists \tau \cdot B \models \mathit{tc}(\rho)\theta\eta\tau$;
– Assert $(\mathit{failed}\ \rho\ \eta)$ into B, if $\chi_t(\rho) = \mathbf{failure\text{-}fail}$, and $\nexists \tau \cdot B \models \mathit{tc}(\rho)\theta\eta\tau$;
– Assert $(\mathit{failed}\ s\ \eta)$ into B, where $s = (\rho\ \boldsymbol{t})$ or $s = (\mathbf{Do}\ A\ (\rho\ \boldsymbol{t}))$, if $\exists \tau \cdot B \models (\mathit{failed}\ \mathit{body}(\rho)\ \tau)$;
– Assert $(\mathit{failed}\ s\ \theta)$ into B, where $s = (\mathbf{agent\text{-}bind}\ \boldsymbol{v}\ \psi)$, if $\nexists \tau \cdot B \models \psi\theta\tau$;
– Assert $(\mathit{failed}\ s\ \theta)$ into B, where $s = (l_1; \cdots l_m)$, if $\exists \theta' \cdot B \models (\mathit{failed}\ l_1\ \theta')$;
– Assert $(\mathit{failed}\ s\ \theta)$ into B, where $s = (\mathbf{par}\ l_1 \cdots l_m)$, if $B \models \bigvee_{i=1}^{m} \exists \theta' \cdot (\mathit{failed}\ l_i\ \theta')$;
– Assert $(\mathit{failed}\ s\ \theta)$ into B, where $s = (\mathbf{forall}\ (\mathbf{cond}\ \psi)\ l_1)$ or $s = (\mathbf{foreach}\ (\mathbf{cond}\ \psi)\ l_1)$, if $B \models \bigvee_{\tau \in \{\eta : B \models \psi\eta\}} \exists \theta' \cdot (\mathit{failed}\ l_1\tau\ \theta')$;
– Assert $(\mathit{failed}\ s\ \theta)$ into B, where $s = (\mathbf{while}\ (\mathbf{cond}\ \psi)\ l_1)$, if $\exists \theta' \cdot B \models (\mathit{failed}\ l_1\ \theta')$;
– Assert $(\mathit{failed}\ s\ \theta)$ into B, where $s = (\mathbf{if}\ (\mathbf{cond}\ \psi)\ l_1\ l_2)$, if $\exists \theta' \cdot B \models (\mathit{failed}\ l_1\ \theta') \vee (\mathit{failed}\ l_2\ \theta')$;
– Assert $(\mathit{failed}\ s\ \theta)$ into B, where $s = (\mathbf{JointDo}\ \mathbf{X}\ (A_1\ l_1) \cdots (A_m\ l_m))$ $(\mathbf{X} \in \{\mathbf{AND}, \mathbf{OR}, \mathbf{XOR}\})$, if $B \models \bigvee_{i=1}^{m} \exists \theta' \cdot (\mathit{failed}\ l_i\ \theta')$;
– Assert $(\mathit{failed}\ s\ \theta)$ into B, where $s = (\mathbf{choice}\ l_1 \cdots l_m)$, if $B \models \bigwedge_{i=1}^{m} \exists \theta' \cdot (\mathit{failed}\ l_i\ \theta')$.

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.* $\langle B, G, H, \theta \rangle \models \mathit{failed}(s)$ *iff* $\exists \theta' \cdot B \models (\mathit{failed}\ s\ \theta')$.

4.3 Transition System

We use **SUCCEED** to denote the terminal configuration where the execution terminates successfully (i.e., all the specified goals and generated intentions are fulfilled); use **STOP** to denote the terminal configuration where the execution terminates abnormally—all the remaining goals are unachievable. In particular, we use **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* $h = [h'\backslash(\psi_k, A_k) \leftarrow s\backslash \cdots]$,

$$\frac{\langle B, G, h, \theta\rangle \models \mathit{failed}(s),\ UC(h) \neq \top}{\langle B, G, h, \theta\rangle \to \langle B, G,\ UC(h), \theta\rangle}, \qquad (\mathbf{F1})$$

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

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).

$$\frac{\begin{array}{c}\exists g = \langle \psi, A\rangle \in G,\ \exists(\rho\ \boldsymbol{v}) \in P,\ self \in A,\\ \exists\tau, (\theta\tau\ has\ bindings\ for\ \boldsymbol{v}), B \models pre(\rho)\theta\tau, and\ post(\rho)\theta\tau \models \psi\end{array}}{\langle B, G, \emptyset, \theta\rangle \to \langle B, G \setminus \{g\}, \{[(\psi, A) \leftarrow (\textbf{Do}\ A\ (\rho\ \boldsymbol{v})\theta\tau)]\}, \theta\tau\rangle}, \qquad (\mathbf{G1})$$

$$\frac{\forall g = \langle \psi, A\rangle \in G, \forall(\rho\ \boldsymbol{v}) \in P\ \ \nexists\tau \cdot post(\rho)\theta\tau \models \psi}{\langle B, G, \emptyset, \theta\rangle \to \mathbf{STOP}}, \qquad (\mathbf{G2})$$

$$\frac{}{\langle B, \emptyset, \emptyset, \theta\rangle \to \mathbf{SUCCEED}}. \qquad (\mathbf{G3})$$

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*
$h_1 = [\cdots\backslash\omega_{k-1}\backslash(\psi_k, A_k) \leftarrow \varepsilon]$,
$h_2 = [(\psi_0, A_0) \leftarrow s\backslash \cdots]$,

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

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

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

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

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. [21] 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 $Br_i = (pref_i\ pro_i\ (\textbf{DO}\ A_i\ (\rho_i\ t_i)))$, let $BR = \{Br_i | 1 \le i \le m\}$, $BR_- \subseteq BR$ 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 $BR^+ = \{Br_k |\ \exists\tau \cdot B \models pref_k \cdot \theta\tau, 1 \le k \le m\} \setminus BR_-$, 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*
$h = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{choice}\ Br_1\ Br_2 \cdots Br_m); s]$,
$h_1 = [h \backslash (true, A_k) \leftarrow (\textbf{DO}\ A_i\ (\rho_i\ t_i)); \textbf{cend}]$,
$h_2 = [h \backslash (true, A_k) \leftarrow \textbf{cend}]$,

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

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

In Definition 9, Rule **C1** applies when there exists a workable branch. The intention h is appended with a new slice ended with **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 **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 3. 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 $Br_i (1 \le i \le m)$ have failed (i.e., $failed$(**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*
$h = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\textbf{agent-bind}\ \ \boldsymbol{v}\ \phi); s]$,

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

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 4. 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 preceeding agent-bind statement in $body(\rho)$.

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

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

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

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

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

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

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

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

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 5. 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 $l = (\textbf{Do}\ \mathit{self}\ (\textbf{send}\ A_k \backslash \{\mathit{self}\}, \langle \mathit{ctell}, \mathit{self}, \psi_0, \alpha\rangle))$, $l_2 = (\textbf{wait until}\ \mathit{ctell}(a, \psi_0, \alpha) \in B)$, 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
$s' = (\textbf{wait until}\ \exists\tau \cdot B \models \mathit{pre}(\alpha)\theta\eta\tau); (\textbf{Do}\ \mathit{self}\ (\alpha\ \boldsymbol{t}))$, and
$s'' = (\textbf{wait until}\ \exists\tau \cdot B \models \mathit{pre}(\alpha)\theta\eta\tau); (\alpha\ \boldsymbol{t})$.

To execute a team operator, all the involved agents need to synchronize. Let $Y(\psi, \Gamma) = \{a' | \mathit{sync}(a', \psi, \Gamma) \in B\}$, 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* $h = [\omega_0 \backslash \cdots \backslash(\psi_k, A_k) \leftarrow (Do\ A\ (\Gamma\ \boldsymbol{t})); s]$*, where* $(\Gamma\ \boldsymbol{v}) \in \mathit{Toper}$*,* $\eta = \{\boldsymbol{v}/\boldsymbol{t}\}$*,*

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

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

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

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

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

where $s^1 = (\textbf{Do}\ \mathit{self}\ \textbf{send}(A, \langle \mathit{sync}, \mathit{self}, \psi_0, \Gamma\rangle)); (\textbf{Do}\ A\ (\Gamma\ \boldsymbol{t}))$,
$s^2 = (\textbf{wait until}\ (|Y(\psi_0, \Gamma)| \geq |\Gamma|)); (\textbf{Do}\ A\ (\Gamma\ \boldsymbol{t}))$,
$s^3 = (\textbf{Do}\ \mathit{self}\ \textbf{unsync}(\psi_0, \Gamma)); (\textbf{Do}\ \mathit{self}\ \textbf{send}(A_k \setminus A, \langle \mathit{ctell}, \mathit{self}, \psi_0, \Gamma\rangle))$,
$s^4 = (\textbf{wait until}\ \forall a \in A \cdot \mathit{ctell}(a, \psi_0, \Gamma) \in B)$,
$s^5 = (\textbf{wait until}\ \exists\tau \cdot B \models \mathit{pre}(\Gamma)\theta\eta\tau); (\textbf{Do}\ A\ (\Gamma\ \boldsymbol{t}))$.

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 6. 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*
$h_1 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\mathbf{JointDo\ AND}\ (A'_1\ l_1) \cdots (A'_n\ l_n)); s]$,
$h_2 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\mathbf{JointDo\ OR}\ (A'_1\ l_1) \cdots (A'_n\ l_n)); s]$,
$h_3 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\mathbf{JointDo\ XOR}\ (A'_1\ l_1) \cdots (A'_n\ l_n)); s]$,

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

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

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

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

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

In Definition 14, Rule **J1** defines semantics for joint-do with share type "AND". It states that before and after an agent does its task l_i, it needs to synchronize (i.e., s^0) with the other teammates wrt. l_i. 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), it 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 that if 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 that l_x has been completed. The semantics of joint-do with share type "XOR" is based on a function *isSelected*()[5]: if an agent belongs to the group of selected agents, it simply synchronizes and executes the corresponding sub-statement (Rule **J3**); otherwise, only synchronization is needed (Rule **J4**).

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 precondition 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

[5] Some negotiation strategies, even social norms [22], can be employed to allow agents to know each others' commitments [23] in determining the selected agents in *isSelected*. We leave such an issue to the designers of MALLET interpreters.

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 $\mathit{failed}(\textbf{Do}\ A\ (\rho\ \boldsymbol{t}))$ 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 $\mathit{failed}(\rho)$ 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 $(\rho\ \boldsymbol{t})$ (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*
$h_1 = [\omega_0\backslash\cdots\backslash(\psi_k, A_k) \leftarrow (\textbf{Do}\ A\ (\rho\ \boldsymbol{t})); s]$,
$h_1' = [\omega_0\backslash\cdots\backslash(\psi_k, A_k) \leftarrow (\textbf{Do}\ A\ (\rho\ \boldsymbol{t}))\theta\eta\tau; s\theta]$,
$h_1'' = [\omega_0\backslash\cdots\backslash(\psi_k, A_k) \leftarrow (\textbf{Do}\ A\ (\rho\ \boldsymbol{t}))\theta\eta\tau; s\theta\backslash(\mathit{post}(\rho)\theta\eta\tau, A) \leftarrow \textbf{endp}]$,
$h_1''' = [\omega_0\backslash\cdots\backslash(\psi_k, A_k) \leftarrow (\textbf{Do}\ A\ (\rho\ \boldsymbol{t})); s\backslash\cdots]$, *where* $(\rho\ \boldsymbol{v}) \in \mathit{Plan}$, $\eta = \{\boldsymbol{v}/\boldsymbol{t}\}$,

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

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

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

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

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

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

where $s^0 = (\textbf{Do}\ \mathit{self}\ (\textbf{send}\ A_k, \langle \mathit{ctell}, \mathit{self}, \psi_0, \rho\rangle));$
$\quad (\textbf{wait until}\ (\forall a \in A \cdot \mathit{ctell}(a, \psi_0, \rho) \in B))$,
$s^1 = (\textbf{Do}\ \mathit{self}\ (\textbf{send}\ A, \langle \mathit{sync}, \mathit{self}, \psi_0, \rho\rangle)); (\textbf{wait until}\ (\forall a \in A \cdot \mathit{sync}(a, \psi_0, \rho) \in B));$
$\quad (\textbf{Do}\ \mathit{self}\ (\textbf{unsync}\ \psi_0, \rho)); \mathit{body}(\rho)\theta\eta\tau; s^0$,
$s^2 = (\textbf{wait until}\ (\forall a \in A \cdot \mathit{ctell}(a, \psi_0, \rho) \in B))$.

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* $s_k = (\mathbf{par}\ l_1\ l_2 \cdots l_m)$, $T_j = \langle B, \emptyset, [(true, A_k) \leftarrow l_j], \theta\rangle \rightarrow^* \langle B_j, \emptyset, [(true, A_k) \leftarrow \varepsilon], \theta_j\rangle \wedge B_j \not\models failed(l_j)$, *and*
$P_B = \langle B, G, h, \theta\rangle \parallel \langle B, \emptyset, [(true, A_k) \leftarrow l_1], \theta\rangle \parallel \cdots \parallel \langle B, \emptyset, [(true, A_k) \leftarrow l_m], \theta\rangle$,

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

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

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*
$h_1 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\mathbf{if}\ (\mathbf{cond}\ \phi)\ l_1\ l_2); s]$,
$h_2 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\mathbf{while}\ (\mathbf{cond}\ \phi)\ l); s]$,
$h_3 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\mathbf{foreach}\ (\mathbf{cond}\ \phi)\ l); s]$,
$h_4 = [\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow (\mathbf{forall}\ (\mathbf{cond}\ \phi)\ l); s]$,

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

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

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

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

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

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

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

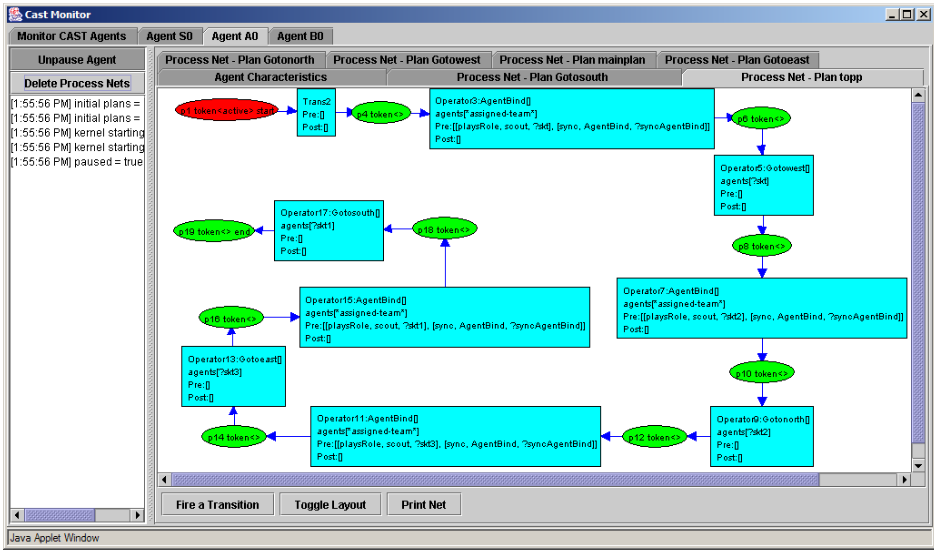
$$\frac{\nexists\tau \cdot B \models \phi\theta\tau}{\langle B, G, \{h_4\}, \theta\rangle \rightarrow \langle B, G, \{[\omega_0 \backslash \cdots \backslash (\psi_k, A_k) \leftarrow s]\}, \theta\rangle}, \qquad \textbf{(S8)}$$

5 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 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 [25].

Figure 1 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 knowledge, and CAST is just one agent architecture that realizes


Cast Monitor
Monitor CAST Agents
Agent S0
Agent A0
Agent B0
Unpause Agent
Delete Process Nets
Process Net - Plan Gotonorth
Process Net - Plan Gotowest
Process Net - Plan mainplan
Process Net - Plan Gotoeast
Agent Characteristics
Process Net - Plan Gotosouth
Process Net - Plan topp
p1 token<active> start
Trans2 Pre:[] Post:[]
p4 token<>
Operator3:AgentBind[] agents["assigned-team"] Pre:[[playsRole, scout, ?skt], [sync, AgentBind, ?syncAgentBind]] Post:[]
p6 token<>
Operator5:Gotowest[] agents[?skt] Pre:[] Post:[]
p8 token<>
Operator7:AgentBind[] agents["assigned-team"] Pre:[[playsRole, scout, ?skt2], [sync, AgentBind, ?syncAgentBind]] Post:[]
p10 token<>
Operator9:Gotonorth[] agents[?skt2] Pre:[] Post:[]
p12 token<>
Operator11:AgentBind[] agents["assigned-team"] Pre:[[playsRole, scout, ?skt3], [sync, AgentBind, ?syncAgentBind]] Post:[]
p14 token<>
Operator13:Gotoeast[] agents[?skt3] Pre:[] Post:[]
p16 token<>
Operator15:AgentBind[] agents["assigned-team"] Pre:[[playsRole, scout, ?skt1], [sync, AgentBind, ?syncAgentBind]] Post:[]
p18 token<>
Operator17:Gotosouth[] agents[?skt1] Pre:[] Post:[]
p19 token<> end
Fire a Transition
Toggle Layout
Print Net
Java Applet Window

```
[1:55:56 PM] initial plans =
[1:55:56 PM] initial plans =
[1:55:56 PM] kernel starting
[1:55:56 PM] kernel starting
[1:55:56 PM] paused = true
```

Fig. 1. The CAST Monitor

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 they conform to the semantics of MALLET and the same communication protocols.

6 Comparison and Discussion

We compare MALLET with JACK Teams [26], OWL-S [27], PDDL [28], and the team-oriented programming framework [9].

JACK Teams [26], instead of providing a higher-level plan-encoding language like MALLET, extends a traditional programming language (i.e. Java) with special statements for programming team activities. In JACK Teams, a team is an individual reasoning entity 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. CAST 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 the *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 can start to perform the associated activity whenever they reach the statement along the team process. 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* statements 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 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 [27] 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 composing composite processes, to which preconditions and effects can be specified. There exist correspondences between OWL-S and MALLET. For instance, both 'Split' in OWL-S and *PAR* in MALLET can be used to specify concurrent activities. The main difference between these two languages 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 teamwork.

PDDL (the Planning Domain Definition Language) [28], inspired by the well-known STRIPS formulations of planning problems, is a standard language for the encoding of planning domains. 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.

In summary, MALLET has been designed as a language for encoding teamwork knowledge, 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.

MALLET does have several limitations. 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.

7 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. In this paper, we defined an operational semantics for MALLET in terms of a transition system, which is important in further studying the formal properties of team-based agents specified in MALLET. The effectiveness of MALLET in encoding complex teamwork knowledge has already been shown in the CAST system [8], which implements an interpreter for MALLET using PrT nets as the internal representation of team process.

Acknowledgments

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

References

1. Cohen, P.R., Levesque, H.J.: Teamwork. Nous **25**:487–512, (1991)
2. Cohen, P.R., Levesque, H.J., Smith, I.A.: On team formation. In Hintikka, J., Tuomela, R., eds.: Contemporary Action Theory. (1997)
3. Jennings, N.R.: Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. Artificial Intelligence **75** (1995) 195–240
4. Grosz, B., Kraus, S.: Collaborative plans for complex group actions. Artificial Intelligence **86** (1996) 269–358
5. Tambe, M.: Towards flexible teamwork. Journal of AI Research **7** (1997) 83–124
6. Rich, C., Sidner, C.: Collagen: When agents collaborate with people. In: Proceedings of the International Conference on Autonomous Agents (Agents'97). (1997) 284–291
7. Giampapa, J., Sycara, K.: Team-oriented agent coordination in the RETSINA multi-agent system. Technical Report CMU-RI-TR-02-34, CMU (2002)
8. Yen, J., Yin, J., Ioerger, T., Miller, M., Xu, D., Volz, R.: CAST: Collaborative agents for simulating teamworks. In: Proceedings of IJCAI'2001. (2001) 1135–1142
9. Tidhar, G.: Team oriented programming: Preliminary report. In: Technical Report 41, AAII, Australia. (1993)
10. Pynadath, D.V., Tambe, M., Chauvat, N., Cavedon, L.: Toward team-oriented programming. In: Agent Theories, Architectures, and Languages. (1999) 233–247

11. Scerri, P., Pynadath, D.V., Schurr, N., Farinelli, A.: Team oriented programming and proxy agents: the next generation. In: Proc. of the 1st Inter. Workshop on Prog. MAS at AAMAS'03. (2003) 131–138
12. Rao, A.S., Georgeff, M.P., Sonenberg, E.A.: Social plans: A preliminary report. In Werner, E., Demazeau, Y., eds.: Decentralized AI 3 –Proceedings of MAAMAW-91), Elsevier Science B.V.: Amsterdam, Netherland (1992) 57–76
13. Kinny, D., Ljungberg, M., Rao, A.S., Sonenberg, E., Tidhar, G., Werner, E.: Planned team activity. In Castelfranchi, C., Werner, E., eds.: Artificial Social Systems (LNAI-830), Springer-Verlag: Heidelberg, Germany (1992) 226–256
14. Tidhar, G., Rao, A., Sonenberg, E.: Guided team selection. In: Proceedings of the 2nd International Conference on Multi-agent Systems (ICMAS-96). (1996)
15. Rao, A.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: MAAMAW'96, LNAI 1038, Springer-Verlag: Heidelberg, Germany (1996) 42–55
16. Pozos-Parra, P., Nayak, A., Demolombe, R.: Theories of intentions in the framework of situation calculus. In Leite, J., Omicini, A., Torroni, P., Yolum, P., eds.: Declarative Agent Languages and Technologies (DALT 2004), LNCS 3476, Springer-Verlag (2005). In this volume.
17. Bordini, R., Fisher, M., Pardavila, C., Wooldridge, M.: Model checking agentspeak. In: Proceedings of AAMAS-2003. (2003) 409–416
18. Wooldridge, M., Fisher, M., Huget, M., Parsons, S.: Model checking multiagent systems with MABLE. In: Proceedings of AAMAS-2002. (2002) 952–959
19. Dastani, M., van Riemsdijk, B., Dignum, F., Meyer, J.J.C.: A programming language for cognitive agents: Goal directed 3APL. In: Proc. of the 1st Inter. Workshop on Prog. MAS at AAMAS'03. (2003) 111–130
20. Giacomo, G.D., Lesperance, Y., Levesque, H.J.: ConGolog, a concurrent programming language based on the situation calculus. AI **121** (2000) 109–169
21. van Riemsdijk, M.B., Dastani, M., Dignum, F., Meyer, J.J.C.: Dynamics of declarative goals in agent programming. In Leite, J., Omicini, A., Torroni, P., Yolum, P., eds.: Declarative Agent Languages and Technologies (DALT 2004), LNCS 3476, Springer-Verlag (2005). In this volume.
22. Robertson, D.: A lightweight coordination calculus for agent systems. In Leite, J., Omicini, A., Torroni, P., Yolum, P., eds.: Declarative Agent Languages and Technologies (DALT 2004), LNCS 3476, Springer-Verlag (2005). In this volume.
23. Winikoff, M., Liu, W., Harland, J.: Enhancing commitment machines. In Leite, J., Omicini, A., Torroni, P., Yolum, P., eds.: Declarative Agent Languages and Technologies (DALT 2004), LNCS 3476, Springer-Verlag (2005). In this volume.
24. Yen, J., Fan, X., Sun, S., Hanratty, T., Dumer, J.: Agents with shared mental models for enhancing team decision-makings. Decision Support Systems, Special issue on Intelligence and Security Informatics (In press) (2004)
25. Yen, J., et al: CAST manual. Technical report, IST, The Pennsylvania State University (2004)
26. JACK Teams Manual. http://www.agent-software.com/shared/demosNdocs/JACK-Teams-Manual.pdf. (2004)
27. OWL-S. http://www.daml.org/services/owl-s/1.0/owl-s.html (2003)
28. McDermott, D.: The formal semantics of processes in PDDL. In: Proc. ICAPS Workshop on PDDL. (2003)